Asset Management in Software Engineering – What is it after all?

Ehsan Zabardast*
Blekinge Institute of Technology
Karlskrona, Blekinge, Sweden
ehsan.zabardast@bth.se

Julian Frattini
Blekinge Institute of Technology
Karlskrona, Blekinge, Sweden
julian.frattini@bth.se

Javier Gonzalez-Huerta
Blekinge Institute of Technology
Karlskrona, Blekinge, Sweden
javier.gonzalez.huerta@bth.se

Daniel Mendez
Blekinge Institute of Technology
and fortiss GmbH
Karlskrona, Blekinge, Sweden
daniel.mendez@bth.se

Tony Gorschek
Blekinge Institute of Technology
Karlskrona, Blekinge, Sweden
tony.gorschek@bth.se

ABSTRACT

When developing and maintaining software-intensive products or services, we often depend on various "assets", denoting the inherent value to selected artefacts when carrying out development and maintenance activities. When exploring various areas in Software Engineering, such as Technical Debt and our work with industry partners, we soon realised that many terms and concepts are frequently intermixed and used inconsistently. Despite the central role of assets to software engineering, management, and evolution, little thoughts are yet invested into what assets eventually are. A clear terminology of "assets" and related concepts, such as "value" or "value degradation", just to name two, are crucial for setting up effective software engineering practices.

As a starting point for our own work, we had to define the terminology and concepts, and extend the reasoning around the concepts. In this position paper, we critically reflect upon the resulting notion of Assets in Software Engineering. We explore various types of assets, their main characteristics, such as providing inherent value. We discuss various types of value degradation and the possible implications of this on the planning, realisation, and evolution of software-intensive products and services over time.

With our work, we aspire to contribute to a more standardised definition of Asset Management in Software Engineering and foster research endeavours and their practical dissemination in a common, more unified direction.

CCS CONCEPTS

• **Software and its engineering → Software creation and management**; *Designing software.*

KEYWORDS

Assets, Asset Management, Technical Debt, Asset Degradation

*All authors contributed equally to this research.

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1234/1122445.1122456

ACM Reference Format:
Ehsan Zabardast, Julian Frattini, Javier Gonzalez-Huerta, Daniel Mendez, and Tony Gorschek. 2020. Asset Management in Software Engineering – What is it after all?. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1234/1122445.1122456

1 INTRODUCTION

Assets and their effective management are known to be crucial for efficient quality control in the engineering, maintenance, and evolution of software-intensive products or services. Such quality control, however, is expensive and, thus, should be implemented with care and only as a means to an end rather than a means in itself; it should be justified and concentrate on managing the quality of only those artefacts that are worth being kept at a high level of quality throughout their life-cycle. Those artefacts are typically coined as assets: Artefacts which are seen to play a vital role to project success and have a certain value when carrying out engineering and maintenance activities in the organisation.

The term, however, is still too often used carelessly without putting much thought into what an asset eventually constitutes and, in consequence, how – as well as why and under which conditions – assets should be managed. While the term seems to be well-established, or at least while it is frequently being used in other fields such as economics, in Software Engineering, we still lack a profound understanding of the term and its implications for Software Engineering practices in general. We, therefore, postulate the importance of a well-defined vocabulary, not only to avoid ambiguous and fuzzy discussions and shallow approaches rooted in those discussions but also to open fruitful avenues for future research and practice.

In Software Engineering, one term often used as an equivalent for an asset is *artefact*. As we will discuss, this term shares similarities, and yet it is not the same, the difference having, in fact, some major implications. One of the key characteristics of assets (in contrast to software artefacts) is the inherent value they bring to an organisation or to a project environment, such that they justify continuous quality control or quality assurance effort. This might not necessarily be per-se the case for all artefacts.

In this position manuscript, we contribute a critical definition of the terminology surrounding assets, and we extend the reasoning around related concepts. We explore various types of assets, their main characteristics, such as providing inherent value and the related notion of value degradation. We discuss various types of value

degradation and the possible implications of this on the planning, realisation, and evolution of software-intensive products and services over time. Our contribution is rooted in our own long-term academia-industry collaborations such as reflected in projects like the Software Engineering Rethought project [1] We will conclude with a discussion of future perspectives for research and practice to hopefully foster contemporary research endeavours in the community of Software Engineering researchers and practitioners in a common, more unified direction.

2 ASSETS IN SOFTWARE ENGINEERING - WHAT ARE THEY AFTER ALL?

An *asset*, in our definition, is a software artefact that has value to an organisation and / or project. It is used for, and / or enables the inception, development, delivery, or evolution of software-intensive products or services over time.

Note that just like software artefacts, assets are self-contained work results that have a context-specific purpose and constitute a physical representation, a syntactic structure, and a semantic content [7]. In that sense, assets constitute work products which are created, modified, and used in the course of several tasks comprised in the development of software-intensive products or services [7]. Key to assets, however, is their persistence. While artefacts may not be persistent at all or at least not over the whole project life-cycle, assets need to be persistent to enable effective quality control. We postulate that software artefacts that are not persistent, or that are just intermediate (short term, or one-off) work results of a task, should therefore not be considered as assets. In our understanding, an asset is, therefore, per definition an artefact, but not necessarily the other way around; certain artefacts might not qualify to be considered assets according to this definition.

Characteristics of Assets in context of Software Engineering

Here, we summarise the distinctive properties and characteristics of assets defined above:

- Assets have inherent value for the organization (which is in tune with the ISO definition of assets [4]). In our definition, assets have - same as artefacts - value to a role.
- Assets are persistent and not inherently transient, i.e., we do not consider intermediate artefacts as assets. In our definition, artefacts that are created with a particular purpose and then immediately transformed to other artefacts are not assets.

Asset Management

We define asset management as an umbrella over the administration of assets and the activities that are related to creating and maintaining them as well as controlling their value. *Asset Management*, thus, considers the explicit control of assets throughout their life-cycle with a particular focus on the assets' degradation and emendation.

[1] See also www.rethought.se.

Assets: The Importance of Value
Let us consider the following example of code that is automatically generated through a model transformation. In some cases, the code might never be maintained or modified, and the modifications are performed only on the models rather than the generated code. The generated code will be of interest to one role – the one that is executing the transformation – while the models and the transformation are of inherent value to the organisation and in the scope of quality control. In other words, the models are the assets while other artefacts, such as the code that is automatically generated, are not.

While artefacts are driven by a specific purpose to selected roles and, thus, have value to those roles, assets should ideally justify their central (quality) control by having value to an entire (project) organisation. This demands a certain persistence of the assets. Same as the value of assets to an organisation may change over time, so does in consequence, the (need for) persistence.

Assets: Persistence as a Consequence for Ensuring Quality Control
Let us consider the example of project reports created as a time-dependent snapshot of current project management status. Such reports may very well have value to a project manager – one role –, at least for the time being. However, the reports might have been created for one sole purpose only, and once the purpose has been achieved, the value might not be given anymore. Hence, controlling the quality of that report, for example, is not justified anymore after archival.

Other Perspectives on Assets

Here, we summarise the different perspectives and their definitions of assets before proceeding to describe the characteristics and attributes of assets from our perspective.

- **Assets in Software Product Lines.** Software Product Lines (SPL) are defined as "set[s] of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way. [8]" In this definition, the *core assets* are "those reusable artefacts and resources that form the basis for the software product line. [8]" In this particular context, assets are not rigorously defined, and the term is used in a rather general way. They focus on assets that can have some degree of embedded variability, or that can be selected during the product development stage, supporting somehow the customization of the software product.
- **Assets in Software Business.** Here, assets are defined as "any type of software, including components and services that can be used for achieving the business objective for a specific system, product or service being developed. [11]", thus, leveraging the general notion of value. The definition gives rise to the following characteristics: *i. Origin*, which denotes where the asset comes from; *ii. Type*, which denotes what kind of asset they are; and, *iii. Attribute*, which denotes the assets' functional and non-functional properties [9]. Similar to the notion of assets in SPL, assets in this context are not rigorously defined, and they are used in a generic way generally related to strategic business aspects.

3 ASSET DEGRADATION: WHAT IS IT AND WHY IS IT IMPORTANT?

Why care about Asset Degradation: It seems that when we refer to software development and assets, the term Technical Debt (TD) [1, 3] is becoming a *catch-for-all* that works for all and every negative consequence that may or may not happen to assets. However, TD seems to be more tailored to code-related assets, and that tends to depict the consequences of non-optimal design decisions [1]. However, software engineering typically implies making trade-offs in solutions of competing qualities, and the option to go for the highest possible quality might just lead to over-engineering the solution. Is it really always worth deciding for the highest quality solution or is a "good-enough" solution preferably (whatever this eventually is and however this might eventually be measured)? That is to say; often, we say that an asset has Technical Debt because it does not adhere to usually academic and purely normative, one-size-fits-all gold standards. However, this is not necessarily a problem. Sometimes, we tend to oversimplify the phenomenon of TD. In some cases, TD has been only operationalised well for certain assets. It turns out that we might be measuring something that does not reflect the actual problem we are interested in controlling, but just some symptoms. Moreover, since we have well understood these symptoms, we try to employ a treatment to fix the problems without trying to understand the root cause of the actual problem (i.e. the disease), resembling the saying: "If all you have is a hammer, everything starts looking like a nail." Thus, the ultimate goal can only be to define effective vaccines and use them to mitigate the actual problem instead of applying treatments to symptoms all the time.

Key Definitions: Nailing Down the Asset Degradation Concept

- **Asset Degradation (AD):** The loss of value that an asset suffers due to intentional or unintentional decisions caused by technical or non-technical manipulation of the asset, or associated assets, during all stages of the product's life-cycle. All assets can degrade, which will affect their *usability* in different ways.
- **Usability of an Asset:** Usability of an asset is the extent to which it supports the intent for what it has been created. It represents the asset's actual fitness: how well it supports the development or value creation process. It is any form of actionable metrics that allows us to compare the effectiveness and efficiency of the development life-cycle. It will allow us to measure how well (or bad) a certain asset fits the purpose, and assess how effective and efficient we are when we use the asset.

The degradation of certain assets, e.g., code, has been widely studied. The reason for that is that it is more tangible, and it allows us to measure how much the actual asset is deviating from the – often academic – gold standard, or from what is expressed in some other asset. Figure 1 depicts one example, putting the main concepts related to asset degradation into context. By analyzing the architecture, we can measure how our code's structure or structures, the current architecture (i.e., the *asset*), deviates from the architectural documentation. There are other types of asset degradation that have not been studied so thoroughly, for example, the degradation of test cases with regards to requirements: What happens if I change my requirements but not the test cases (also depicted in Figure 1)? We certainly can say that those test cases have degraded, but the impact and how to fix the problem is not well understood. In general, we can say that asset degradation matters, and it is causing pain to organizations, impacting the effectiveness and efficiency of the development pipeline.

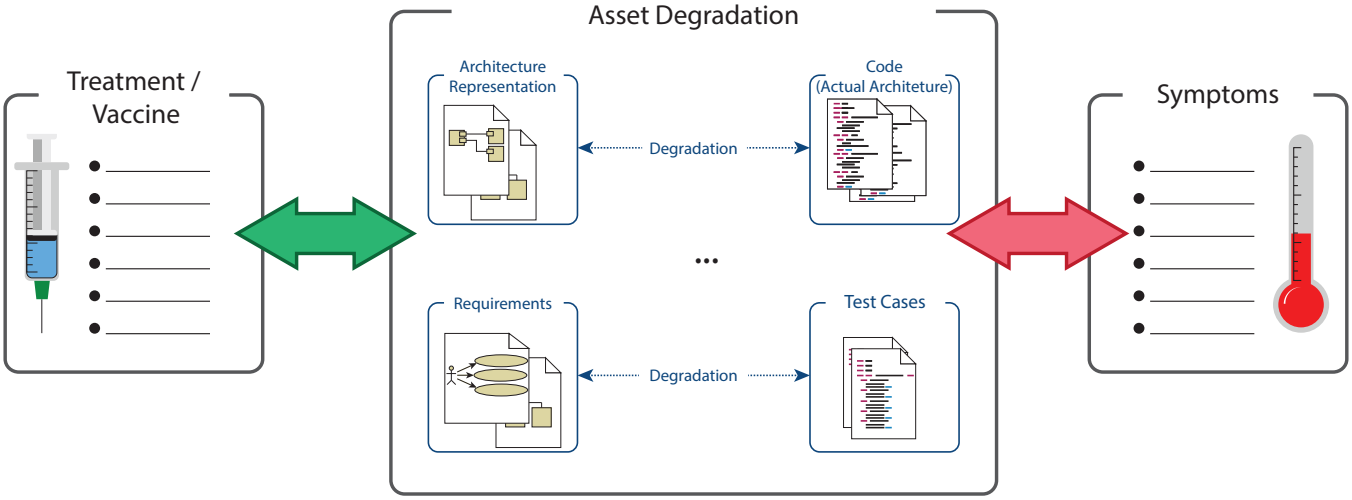

Figure 1: Conceptual Framework for Asset Degradation

Types of Asset Degradation

- **Deliberate**: The asset is degraded as a result of a modification comprising a conscious non-optimal decision. We take a shortcut, knowing its consequences, and consciously paying its prize. There is awareness of the fact that the usability of the asset is being hindered. This type of degradation is clearly avoidable.
- **Unintentional**: The asset is degraded as a result of a modification comprising an unconscious non-optimal decision. We might be or might not be aware of the other alternatives, and we do not foresee the consequences. There is no awareness of the fact that the usability of the asset can be hindered. This type of degradation is not directly avoidable. Only raising awareness, we would be able to understand the consequences of non-optimal decisions.
- **Entropy**: Similar to the second law of thermodynamics, the disorder of software assets will never decrease. *"As a software program is evolved its complexity increases unless work is done to maintain or reduce it"* and *"will be perceived as of declining quality unless rigorously maintained and adapted to a changing operational environment"* [5, 6]. Entropy might introduce degradation in the form of natural decay on assets even when we have quality management mechanisms to avoid such degradation.

The side-box *Types of Asset Degradation*, presents how assets can degrade. However, the *Unintentional* and the *Entropy* are, by definition, almost impossible to separate. Some of the unconscious decisions can be caused or hidden by the growth on complexity and size, and therefore entropy is causing the introduction of even more *Unintentional* degradation.

Degradation and its consequences: Tornhill discusses in [10] that what we tend to measure when we talk about TD might not depict the actual pain of the code's degradation, but only some superficial symptoms. What does it mean that a tool states that we have accumulated years of TD in our code-repositories? Is all TD equally painful? Tornhill gives the problem another angle, by analyzing the modules that more frequently require maintenance and the ones with the highest complexity [10], and how the code ages. Needless to say, this makes much sense, and yet: is it relevant? We argue that although analyzing the impact of the problem in the development process, by looking at which parts of the system are causing more problems over time, might be a right way of tackling the actual problem. However, it still misses the most crucial perspective: before solving a problem properly, we might better investigate which proper problem to solve.

Still, the main question remains unanswered: can we identify the real consequences of an asset's degradation? Suppose we can measure the fitness – the actual *usability* of the asset –. In that case, we can have an actionable metric that allows us to compare the effectiveness and efficiency using the asset in the development life-cycle. We will be able to discuss the actual consequences that the degradation might have over the development pipeline. We can measure how good (or bad) a certain asset is, and we can measure how effective and efficient we are when using the asset.

Discussions usually revolve around the symptoms level, but instead, we aim at characterising: i) which are the relevant assets, ii) how assets degrade with regards and which *type of degradation* is affecting them, iii) the degradation that really matters - the one that really affects the usability of the involved assets, iv) and finally the potential *treatments* and *vaccines*. We cannot treat all our problems as one big blob; we must break it down to different assets, characterise their degradation, its consequences for the organisation, and the system's quality, cost and value [1]. However, what is even more relevant: since not all degradation is equal, not all types of degradation matter to the same extent. The focus should be put on the degradation that severely affects the usability of the asset.

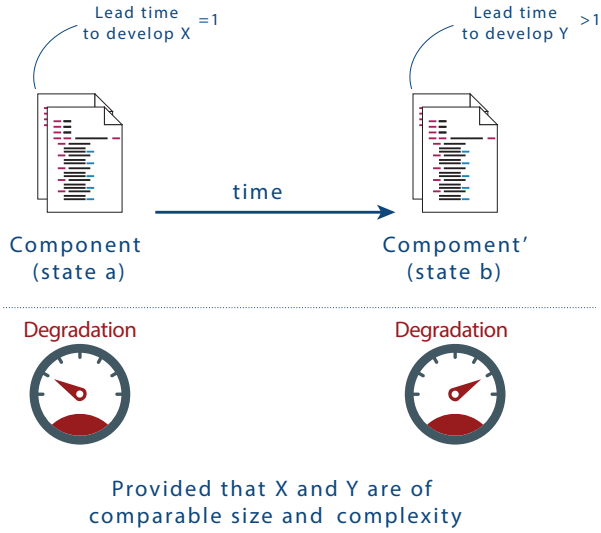

Figure 2: What we mean by impact on Lead Time

> ***Usability of an Asset: Impact on Lead Time***
> Let us assume that we have a component in the system that we suspect is degrading. We can analyze lead time and other metrics to estimate how "painful" it is to use this asset in the development. If we have similarly complex, comparable maintenance tasks affecting this component in different moments of time, we see that its degradation is growing, and we observe that the lead time is growing over time, then we could conclude that the degradation of this asset is relevant (see Figure 2). However, if we observe the opposite, that the asset is degrading, but we see no effect (or even a positive effect) on lead-time or other relevant performance metrics over time. We can conclude that this type of degradation might not impact the effectiveness and efficiency of its usage (the lead time in this case). We aim for actionable metrics that allow us to measure the assets' internal usability and prioritize the asset degradation that has a real impact on our development life-cycle.

4 FUTURE PERSPECTIVES FOR RESEARCH AND PRACTICE

A shift in the view on assets, their value, and their management – starting with a proper and concise terminology – will enable a new way of considering how the metaphor of debt can be applied in software engineering and, related, building tangible asset taxonomies. We explain both in the following.

4.1 Going Beyond Technical Debt

Technical Debt as a metaphor mapped from economic sciences runs the risk of not reflecting the essence of the accrued debt due to a limited scope on the mapping. This may manifest in both directions.

On the one hand, technical debt is often underestimated: while the *principal* of clearing debt is quite clear most of the time, and with a limited scope – as it often boils down to something comparable to rephrasing a requirement, refactoring a module, or rewriting a test – however, the scope of the accrued *interest* of a debt item is much harder to encompass. Often hidden chain-reactions and ripple effects of degradation might have a significant impact on the development life-cycle and pose a major risk if remaining undetected. Following up the example of rephrasing a requirement, what would the interest of that debt be as we continue developing that requirement or new ones? What is the scope for the calculation of this accrued *interest*?

On the other hand, over-engineering is a prominent pit-fall in software development, well portrayed by Kent Beck [2] with his *the design for today vs design for tomorrow* metaphor. Over-engineering equally threatens proper risk-management of the development life-cycle: debt items with a negligible *interest rate* do not deserve a time- and cost-intensive overhaul, as their impact, does not justify the effort. What if we accrue debt in a module that is never touched again? Does degradation matter? Those questions reflect the differentiation that has to be made to assess and allocate resources for reworking.

Both hazards of over- and under-engineering are mainly caused by a lack of understanding how degradation is propagated in a development project. Research on traceability reasonably succeeds in exposing relationships between certain assets, e.g., between requirements and code, or code and tests, but fails to provide a holistic view of assets and the relations among them. Rethinking the significance of assets and accordingly adjusting risk mitigation strategies is a first step towards rectifying this misalignment.

4.2 Asset Anatomy

The challenge regarding assets stems from their complex distribution in software-intensive product development. They are present in all stages of the inception, development, delivery, and evolution of the software product; various roles utilise them, and they are valuable for the development organisation. It is important to create taxonomies of assets to capture such a vast body of knowledge. These taxonomies allow us to cluster and classify the different assets, and to uncover implicit relationships and dependencies among them. Assets do not exist in isolation; in contrast, they are ever-evolving entities which have properties, characteristics, and relations and dependencies among each other.

These relationships and dependencies are key to be able to identify and characterize the different types of degradation assets are subject to. The assets and the relationships among them will become the *Anatomy of Assets*, similar to Gray's Anatomy for the human body, and will ultimately allow us to frame the chain reactions and ripple effects of degradation on assets. Understanding the relations between assets while considering their properties and characteristics can help us identify the potential propagation of degradation among assets, detect its symptoms, and treat its root causes. These chain-reactions and ripple effects might manifest as a result of the changes and evolution of other assets. Capturing such convoluted phenomena is hard, but it is an crucial step to create any asset management framework (i.e., any framework for impact analysis).

5 CONCLUSION

In this positioning, we provide a clearer, standardized definition for assets and asset management, introducing the concept of asset degradation, and concluding with an outlook on the perspectives for research and practice.

In our experience, we see that the term Technical Debt can be misleading, and with the introduction of a coherent set of concepts and the corresponding, concise terminology, such as degradation, symptoms, root causes, treatments, and vaccines, we may improve how we view and handle assets, in particular when discussing potential actions in practice.

Building on this asset anatomy, we specifically aspire to construct a framework for asset degradation that includes also means for effective impact analysis to support evidence-driven risk management approaches properly. Those can then take: i) Propagation of degradation from one asset to all inter-related assets using the asset anatomy. ii) Measurement metrics for degradation on the impacted assets. iii) Evaluation of severity of the propagated degradation to estimate the *interest rate* of the debt item. iv) Strategies of emendation for managing the risk and clearing the accrued debt. Our vision is that this framework provides a guideline on applying the concept of asset degradation in a practical context. At the time of writing this manuscript, we are working on this framework. Though a full project-encompassing, an oracle-type prediction is unrealistic, this framework shall rather provide the best possible, holistic view for evidence-based risk management.

The hope we associate with this manuscript is to have not only contributed to a more standardised terminology for asset management in Software Engineering but to have also motivated the need and ways for future work in this area. We cordially invite researchers and professionals to join us in this direction.

ACKNOWLEDGMENTS

We would like to acknowledge that this work was supported by the KKS foundation through the SHADE KKS Hög project with ref: 20170176, and through the S.E.R.T. Research Profile project at Blekinge Institute of Technology.

REFERENCES

[1] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, Carolyn Seaman, and Carolyn Seaman. 2016. Managing Technical Debt in Software Engineering. In *Dagstuhl Reports*, Vol. 6. 110–138. https://doi.org/10.4230/DagRep.6.4.110

[2] Kent Beck. 1999. *Extreme Programming Explained: Embrance Change.* Addison-Wesley Professional.

[3] Ward Cunningham. 1993. The WyCash portfolio management system. *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92) (Addendum)* 4, 2 (apr 1993), 29–30. https://doi.org/10.1145/157710.157715

[4] ISO/IEC/IEEE. 2014. *Asset management - Overview, principles, and terminology ISO/IEC/IEEE 55000:2014.* Technical Report.

[5] M.M. Lehman. 1979. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software* 1 (1 1979), 213–221. https://doi.org/10.1016/0164-1212(79)90022-0

[6] Manny M Lehman. 1996. Laws of software evolution revisited. In *European Workshop on Software Process Technology*. Springer, 108–124.

[7] Daniel Méndez Fernández, Wolfgang Böhm, Andreas Vogelsang, Jakob Mund, Manfred Broy, Marco Kuhrmann, and Thorsten Weyer. 2019. Artefacts in software engineering: a fundamental positioning. *Software and Systems Modeling* 18, 5 (oct 2019), 2777–2786. https://doi.org/10.1007/s10270-019-00714-3 arXiv:1806.00098

[8] Linda Northrop, Paul Clements, Felix Bachmann, John Bergey, Gary Chastek, Sholom Cohen, Patrick Donohoe, Lawrence Jones, Robert Krut, Reed Little, et al. 2007. A framework for software product line practice, version 5.0. *SEI–2007–http://www. sei. cmu. edu/productlines/index. html* (2007).

[9] Efi Papatheocharous, Kai Petersen, Antonio Cicchetti, Séverine Sentilles, Syed Muhammad Ali Shah, and Tony Gorschek. 2015. Decision support for choosing architectural assets in the development of software-intensive systems: The GRADE taxonomy. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*. 1–7.

[10] Adam Tornhill. 2018. *Software Design X-Rays - Fix Technical Debt with Behavioral Code Analysis.* The Pragmatic Programmers, L.L.C. 252 pages.

[11] Claes Wohlin, Krzysztof Wnuk, Darja Smite, Ulrik Franke, Deepika Badampudi, and Antonio Cicchetti. 2016. Supporting strategic decision-making for selection of software assets. In *International conference of software business*. Springer, 1–15.