Technical Debt: Showing the Way for Better Transfer of Empirical Results

Forrest Shull, Davide Falessi, Carolyn Seaman, Madeline Diep, and Lucas Layman

Abstract

In this chapter, we discuss recent progress and opportunities in empirical software engineering by focusing on a particular technology, Technical Debt (TD), which ties together many recent developments in the field. Recent advances in TD research are providing empiricists the chance to make more sophisticated recommendations that have observable impact on practice.

TD uses a financial metaphor and provides a framework for articulating the notion of tradeoffs between the short-term benefits and the long-term costs of software development decisions. TD is seeing an explosion of interest in the practitioner community, and research in this area is quickly having an impact on practice. We argue that this is due to several strands of empirical research reaching a level of maturity that provides useful benefits to practitioners, who in turn provide excellent data to researchers. They key is providing observable benefit to practitioners, such as the ability to tie technical debt measures to business goals, and the ability to articulate more sophisticated value-based propositions regarding how to prioritize rework. TD is an interesting case study in how the maturing field of empirical software engineering research is paying dividends. It is only a little hyperbolic to call this a watershed moment for empirical study, where many areas of progress are coming to a head at the same time.

F. Shull (✉) • D. Falessi • C. Seaman • M. Diep • L. Layman
Fraunhofer Center for Experimental Software Engineering, 5825 University Research Court, Suite 1300, College Park, MD 20740-3823, USA
e-mail: fshull@fc-md.umd.edu; dfalessi@fc-md.umd.edu; cseaman@fc-md.umd.edu; mdiep@fc-md.umd.edu; llayman@fc-md.umd.edu

1 Introduction

Software engineering is an exceedingly dynamic field. Since the term "software engineering" was coined at the 1968 NATO conference, the field has seen an explosion in terms of the number of applications and products that use software, an immense growth in the sophistication and capabilities of those products, multiple revolutions in the way software relates to the hardware and networks over which it runs, and an ever-changing set of technologies, tools, and methods promising more effective software development.

Similarly and not surprisingly, the field of empirical software engineering has been dynamic as well. In the decades in which empirical studies have been performed, we have seen evolutions in the objects of study, study methodologies, and the types of metrics used to describe those study objects. Also, as with software engineering in the large, empiricists have seen our own fads come and go, with different types of studies being introduced, becoming (over-)popular, and then settling into a useful niche in the field. Articles elsewhere in this book [1] have reflected on some of these trends. In this chapter, we discuss some recent progressions and opportunities in the area of empirical software engineering by focusing on a particular technology, Technical Debt, which ties together many recent developments in the field. We use Technical Debt to discuss recent advances that are providing empiricists the chance to make more sophisticated recommendations and to have more of an impact on practice. We also use this concept as a launching point to look at how some of these recent progressions may extend into the future.

2 Technical Debt: What Is It?

Before describing the opportunities that TD brings to empirical software engineering, let's try to understand what TD is. First coined in 1992 [2], the underlying ideas come from the mid-1980s and are related to Lehman and Belady's notion of software decay [3] and Parnas' software aging phenomenon [4]. TD is a metaphor, and while it lacks a formal definition, it can be seen as "the invisible results of past decisions about software that negatively affect its future" [5]. The reference to financial "debt" implies that the notion of tradeoffs between short-term benefits and long-term costs is central to the concept.

Because TD is a metaphor, it can be applied to almost any aspect of software development, encompassing anything that stands in the way of deploying, selling, or evolving a software system or anything that adds to the friction from which software development endeavors suffer: test debt, people debt, architectural debt, requirement debt, documentation debt, code quality debt, etc. [6]. Research into TD often bears a superficial resemblance to earlier empirical work on defect identification; TD identification often takes the form of identifying deficiencies in software development artifacts (requirements, architecture, code, etc.).

While TD research builds upon many of the empirical lessons learned from defect identification, we argue that TD research introduces an important new dimension into empirical studies of software defects and software quality: context-dependent short-term versus long-term quality tradeoffs. Consider the progression of empirical work in software quality:

- *Approximating quality via defect counts:* Many studies of software quality have used defect counts as a proxy for a technique's impact on software quality. For example, in studies of software V&V methods, it is often assumed that the more defects identified by a technique, the bigger the resulting improvement to software quality that can result from its application. Some examples include [7, 8].
- *Value-Based Software Engineering:* It was always recognized that defect counts were simply a proxy for software quality, but the Value-Based Software Engineering (VBSE) paradigm gave the community more tools required for a sophisticated view of the problem. VBSE articulated the idea that value propositions in software development need to be made explicit, so that software engineers can determine whether stakeholder values are being met and, indeed, whether they can be reconciled [9]. Studies reflecting a VBSE point of view tend to weight defects differently in terms of their severity for different stakeholders, or according to the operational scenarios under which those defects would be detected. In short, the studies were designed on the assumption that the true impact on quality can vary greatly from one defect to another.
- *Quality is relative in time and context:* Work on TD extends the VBSE considerations even further. First, in TD, the issues needing rework are more tightly coupled to the team's specific quality goals. For example, deficient documentation may not represent a "defect" in the sense that it will lead to incorrect software, but this deficiency may represent TD for teams that prioritize reuse and maintainability. In contrast, deficient documentation would not be considered as TD by a team that is developing a throw-away prototype. Second, TD instances do not automatically represent deficiencies in the system; rather they represent a tradeoff that was made in order to achieve some other short-term goal. TD may even be healthy in the short term, such as trading off a perfectly maintainable design to add quickly a feature needed immediately by an important customer. The TD metaphor stresses that other considerations need to be taken into account by teams contemplating rework of a TD issue, such as how much effort it would take to correct that instance and how much it is "costing" to have that instance in the system.

TD research has inherited much from prior generations of empirical studies that looked at software quality: approaches to counting discrete instances of items for potential rework (whether they be instances of defects or TD); the need for taxonomies to categorize those instances and provide insights regarding root causes; and the goal of characterizing various manual and automated techniques in terms of the number and type of instances that they uncover.

TD encapsulates new aspects of empiricism by providing **a context-dependent way of thinking about software quality across lifecycle phases, and in a way**

tractable to quantitative analysis and hence objective observations. Thus, the primary contribution of TD to the empirical community is useful guidance for: (1) analyzing the cost tradeoffs of software engineering decisions; and (2) effectively transmitting the results of empirical research to practitioners by recognizing the existence and management of these tradeoffs.

3 Technical Debt: A Boundless Challenge

One important distinction is between unintentional and intentional debt [10]. Unintentional debt occurs due to a lack of attention, e.g., lack of adherence to development standards or unnoticed low quality code that might be written by a novice programmer. Intentional debt is incurred proactively for tactical or strategic reasons such as to meet a delivery deadline. Intentional debt has been further broken down into short-term debt and long-term debt, which represent, respectively, small shortcuts like credit card debt, and strategic actions like a mortgage. Based on this classification, Fowler created a more elaborate categorization composed of two dimensions—deliberate/inadvertent and reckless/prudent [11]. These dimensions give rise to four categories: deliberate reckless debt, deliberate prudent debt, inadvertent reckless debt, and inadvertent prudent debt. This classification is helpful in finding the causes of technical debt, which lead to different identification approaches. For example, to identify reckless and inadvertent debt, especially design debt, source code analysis may be required.

Technical debt can also be classified in terms of the phase in which it occurs in the software lifecycle—design debt, testing debt, defect debt, etc. [12]. Design debt refers to the design that is insufficiently robust in some areas or the pieces of code that need refactoring; testing debt refers to the tests that were planned but not exercised on the source code. This type of classification sheds light on the possible sources and forms of technical debt, each of which may need different measures for identification, and approaches for management. For example, comparison to coding standards may be required to identify and measure design debt, while testing debt measures require information about expected testing adequacy criteria. Other types of debt, based on the lifecycle phase or the entity in which the debt occurs, have been suggested in the literature, e.g., people debt, infrastructure debt, etc. Some types of TD grow organically, without any actions on the part of developers, because software and technology inevitably become out of date with respect to their environment.

Another important dimension along which instances of TD can be categorized is visibility to the customer and end-users. Instances of visible TD include poor usability and low reliability. Instances of invisible TD include violations of architectural rules and missing documentation. Some, but not all, definitions of TD exclude debt that is visible to the end-user.

TD has been recognized to exist in every sector of the software industry, and the research community has been active on this topic. Formal, scholarly investigation of TD is just beginning and is starting to produce usable improvements. The number of TD research outputs is increasing rapidly. There have been to date three Workshops on Managing Technical Debt, the first one resulting in a joint research agenda [13] and the remaining two co-located with ICSE 2011 and 2012. Two more workshops are planned for 2013. A recent *IEEE Software* special issue contains several articles on the multifaceted concept of TD [5]. Unfortunately, these workshops and published papers have to date failed to produce a universally agreed-upon and used definition of TD. Thus, there are signs that the term TD has been overloaded and is losing its meaning. Every new software engineering technique or empirical investigation has (to a greater or lesser extent) an impact on, or is affected by, TD. For instance, an empirical study comparing the effectiveness of two V&V techniques could support mitigation of TD because future TD decisions about which V&V activity to implement can be based on such an observation. There is a noticeable trend in titling software engineering papers to relate them to TD, even if that relationship is tenuous. However, the lack of a concrete definition of TD makes it difficult to argue against such titling. Such broad labeling of published studies makes aggregation of results hard. Clearly, an ongoing challenge for the TD research community is to find a way to define the term broadly enough to encompass all relevant research, but concretely enough to draw a useful boundary and give guidance to authors.

4 Technical Debt Brings Empirical Opportunities

The concept of technical debt is one that resonates strongly with the developer community as evidenced by the number of practitioner-authored blog posts, presentations, and webinars conducted on the topic. It has been our experience that practitioners desire research results on this topic more so than on many other subjects of empirical research. What is the reason for this surge of interest? We argue that an important reason is maturity of the empirical research methods applied to TD—many of the most powerful and mature empirical methods have come together in a mutually-supportive way to study TD, to engage real-world problems, and to communicate useful results to practitioners.

4.1 Identifying and Predicting TD Costs Is Improved by Empiricism

In general, managing TD consists of estimating, analyzing, and reasoning about: (1) where TD exists in a system so that it can be tagged for eventual removal, (2) the cost of removing TD (i.e., the principal), and (3) the consequences of not removing TD (i.e., the interest). Regarding point (1), there is a large body of software engineering literature [6, 14] related to how to identify TD. Points (2) and (3) require a more careful discussion. In the context of TD, the term "principal" refers to the cost of fixing the technical problem (i.e., removing the debt). For instance, the principal related to the debt affecting a component of a system with high coupling and cohesion refers to the effort necessary to refactor the component to achieve a lower level of coupling and cohesion (e.g., refactoring the component is estimated to cost $500). Principal needs to be estimated, and the principal estimate will be more accurate and the resulting decisions more sound with a reliable knowledge base. However, in the absence of historical data, a rough estimate (e.g., high, medium, low) based on expert opinion is more helpful than no estimate at all.

In the context of TD, the term "interest" refers to the cost that will be incurred by not fixing the technical problem (i.e., the consequences of not removing the debt). For example, the interest related to a component of a system with high coupling and cohesion refers to the extra effort that will be necessary to maintain the component in the future. We note that, unlike the principal, the interest is not certain but has an associated probability of occurrence. In other words, you can be sure that refactoring a component will cost you something (i.e., $500), but you cannot be certain about the consequences of not refactoring it. Therefore, estimating interest means estimating both the amount and its probability of occurrence. These estimates are difficult to make, and in practice a rough estimate of high, medium, and low is the best that can be obtained. Even these rough estimates are more reliable if they are based on historical data.

Managing TD encompasses several estimation activities that are clearly of an empirical nature. In practice, it is difficult to predict anything without a reliable knowledge base.

4.2 The Pivotal Role of Context and Qualitative Methods

TD concepts like principal and interest are context-specific. In fact, the same TD in one organization (e.g., a specific level of coupling and cohesion) can have a low or high principal (i.e., can be easy or hard to eliminate) and a low or high debt (i.e., can have a low or high impact in the future) depending on the project or even the subsystem within a project. Thus, there is a need for quantitative methods to elicit meaningful representations of interest and debt in context.

Unfortunately, we do not yet know how to determine when one project is similar enough to another to experience similar results. Even when working in the same exact context, the future will always differ from the past. For example, when working with the same industrial partner, they may experience employee turnover that threatens the applicability of past results. Even if employees do not change, their experience and performance inevitably changes over time. Moreover, almost every software engineering technique is dependent on others. Thus, it is questionable if the assessment of a given technology still holds when the related technologies changed.

In the context of TD, principal and interest clearly vary among environments and there is a need to know one's customers and collaborate closely with them. Context factors can be elicited in a number of ways. Qualitative methods are needed when it is not clear which factors are relevant, or when there is a desire to discover new unknown context factors. Given the appropriate prompts, developers, managers, and other stakeholders can all provide important context information through interviews, focus groups, or observation.

There was a time when the software engineering research community debated whether qualitative research methods are appropriate, and proponents had to advocate for more adoption [15, 16]. These qualitative methods are now an integral part of the TD work and one of the reasons why TD tech transfer has been so effective.

When managing or studying TD in particular, two important elements of context are: (1) the software qualities of interest (i.e., what are the most important success criteria?); and (2) the "pain points" (i.e., where has the interest on TD been felt most acutely?). For example, if an organization is most concerned with on-time delivery, then they would be most interested in dealing with TD that causes late-lifecycle surprises, such as inadequate testing. If an organization has a history of damaging cost overruns during maintenance on a large legacy system, then they would be most interested in controlling design debt by refactoring code that is brittle, overly complex, or hard to maintain. Such subtle elements of context are often not documented, and can only be discovered through asking the right questions.

As an example, in [17], practitioners were interviewed and shared a variety of long-term pains resulting from TD. These pains varied from fragile code to poor performance to the added complexity of problems found at the customer site. They also shared varying contexts that influence decisions made about TD, such as the difference between short-lived, non-critical software and software whose longevity is uncertain. The open-endedness of the interviews made it possible to elicit elements of context that had not previously been reported in the literature, and also to gain a richer understanding of previously-known factors.

Sometimes, context information that relates to quality goals and to "pain points" can be found by mining the data archives of a project. In [18], the authors used archival data to conduct an in-depth retrospective study of a particularly high-interest instance of TD. The interest was incurred due to a decision to delay an upgrade in the infrastructure software, then a decision to make a substantial change in the architecture, which then necessitated a greatly increased amount of work later when the upgrade could no longer be delayed. The data analysis revealed the historical sequence of events and decisions that made this TD more expensive in the long run than it initially appeared.

In another study [19], we examined the use of reference architecture across projects in a mid-sized software development company that focuses on database-driven web applications. By collaborating with practitioners, we found that technical debt arises when developers design their own solutions and avoid reuse, and that designing the system to be in compliance with the reference architecture leads to greater understandability and maintainability over time in the future. In this same context, we also observed that code smells and out-of-date documentation can be realistic indicators of technical debt.

Finally, by observing a team developing high-performance code for supercomputers, we noticed that they solve the challenge of optimizing the use of the parallel processors by strongly separating calls to the parallelization libraries from the code doing scientific simulation, thereby allowing both the computer scientists and the domain experts to focus on what they know best [20]. In this context, the instances where this separation of concerns breaks down should be treated as technical debt—by detecting and fixing where the planned architecture of the system is not followed, we can help the developers create a more maintainable and flexible system.

We note that in all these industry collaborations, discussions with the whole team were essential to validating our concepts of TD in that context. For example, the idea of separation of concerns may not be applicable on a more homogeneous team that does not have to deal with multiple types of specialized complexity, and we do not expect that specific rules for identifying "code smells" would be applicable for every development team.

4.3 Tool Support Enables Empiricism

Much of the measurement "infrastructure" that is our legacy from the empirical software engineering research community (e.g., GQM, QIP) was originally defined as a set of methodologies without an explicit tool-supported component. Contemporary empirical studies, in contrast, rely heavily on tool support and automation in order to deal with the size and complexity of today's software engineering products when collecting, analyzing, and exploring metrics data. Moreover, automated or computer-assisted approaches are necessary to get buy-in from project teams who are used to doing all of their other project activities online. TD is no exception.

To date, several methods and tools for detecting anomalies in source code (automated static analysis, code smell detectors, etc.) have been developed, and these tools show promise for the task of identifying TD [19]. However, these tools have not yet been integrated with developers' day-to-day work practices and tools and, more importantly, with management's day-to-day decision-making processes. These shortcomings have led to limited adoption of existing methods to manage TD in industry and a lack of understanding about what can be gained from managing debt.

A long-term vision for tool support for TD decision-making, which will always by necessity include a human component, is a set of integrated tools that continually monitor and diagnose the forms of TD that are accumulating and that threaten the goals of the project, providing a continual stream of actionable information to human decision-makers. Such a toolkit must be integrated so that one single, seamless interaction is available to practitioners for all steps involved in choosing and applying TD identification techniques, aggregating the results, analyzing the choices available for a particular decision, and recording the outcome of the decision itself. The toolkit must also be extensible to incorporate new technical debt identification techniques as well as other decision approaches as they mature. The toolkit must be accompanied by a methodology that describes not only the process of applying the toolkit and choosing among the available options, but also how the use of the toolkit should fit into software maintenance project management practices already in place.

Such tool support is necessary for technology transfer in this area, not only because industry adoption depends on effective tools, but also because such tools will allow experimentation with our findings in development environments and assist in the collection of necessary data to support further research.

4.4 Smarter Dashboards

The idea of dashboards to provide an aggregated view over key metrics is not a new one. The recent trend in this area, however, is the development of a set of principles that exploit the typical data-rich development environment to provide more effective and useful guidance. Across a number of different software domains, some of these common principles include the need for metrics dashboards to be:

- *Built on automated data collection*: Making the collection of measures an extra step for developers who are already overloaded is rarely a recipe for success. Metrics programs with staying power use metrics that are already available for other reasons, e.g., the use of timesheet or time accounting systems to track effort and tasking, or the use of JIRA and other bug tracking tools to measure defect backlog and closure rate. Effective dashboards are those which are built upon integrating data streams from these existing sources, and tying these measures to the questions of interest.
- *Easily Changeable*: In software acquisition environments, data comes from a number of different sources and may change from period to period (within contractually mandated limits). Effective dashboards are those which can easily be adapted to new data schemas as necessary.
- *Trustability:* The data used by the dashboard must be checked to help provide confidence in the results presented. Usually, the quantity of data requires at least some level of automation to verify the quality of the data and the dashboard results.
- *Allowing details-on-demand:* The ability to roll-up low level data, such as data measured at the subsystem level, separate components, or external sources, is necessary if the dashboard is to provide high-level status monitoring. If not done properly, however, roll-up data may camouflage valuable insight. Recent advances in reusable graphical libraries ensure that data exploration can happen quickly and efficiently.

Having a dashboard that pulls together and visualizes TD information would be beneficial during the decision-making process, and hence work has been ongoing in this area [21]. The dashboard should present information that is relevant to both engineers and managers, at varying levels of abstraction. For example, the dashboard could show how many TD instances have been identified, and provide capability for the engineers to drill down to the source of each TD tied to the development work products, e.g., location of the source code that could benefit in refactoring, problematic requirement statements, etc. Such information enables traceability between each TD instance and the affected documentation, as well as allowing the engineers to understand the extent and context of the TD. Additionally, business-level information, such as comparison between the principal cost and estimated interest of the TD, may be more relevant for the managers. The cost can be rolled up to show the collective impact of TD for the whole system/program. At the same, on demand, managers could drill down and understand how the impact of TD is distributed across the subsystems or program modules or other development work products.

More ambitiously, we envision future TD dashboards that include what-if analysis capability, where managers could play out various scenarios corresponding to the removal of one or more TD instances. For each scenario, the dashboard could visualize how the removal of a source of TD impacts the landscape of the remaining TD (the principal and the interest). Additionally, the dashboard could offer the view of the TD landscape over time—one TD instance's cost may increase at a more significant rate than others as the cost of principal and interest (likelihood and additional effort) may vary over time. All these different perspectives could assist in the prioritizing of TD removal effort.

5 Conclusion

In this chapter, we have shown how research in one of the fastest-growing areas of empirical study, Technical Debt, relies on a variety of results and methods from past empirical software engineering research. This represents an important development in its own right, since software engineering research has long suffered from an inability to build on previous results. We have also shown how the success of TD research is indebted to recent trends in all of those areas. In this sense, TD makes an interesting case study in how the current level of maturity in empirical studies is paying dividends—it is perhaps only a little hyperbolic to call this a watershed moment for empirical study, where many areas of progress are coming to a head together.

The TD metaphor itself is an important focus of further work since it provides a framework that is both compelling to practitioners and ties together research results on many different topics. This chapter provided our vision of where the research can go: producing an improved and context-specific approach to software measurement that provides tighter feedback loops and more information to developers when they can best use it. Just as importantly, the refinements to empirical research methodologies and principles in the TD work (e.g., accounting for context and business value) will be crucial to other areas of software engineering research by strengthening the ability to influence software development practice.

References

1. Basili, V.R.: A personal perspective on the evolution of empirical software engineering. In: Münch, J., Schmid, K. (eds.) Perspectives on the Future of Software Engineering: Essays in Honor of Dieter Rombach. Springer (2013)
2. Cunningham, W.: The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger **4**(2), 29–30 (1993). doi:10.1145/157710.157715
3. Lehman, M.M., Belady, L.A.: Program Evolution - Processes of Software Change, APIC Sudies in Data Processing No. 27, Academic (1985)
4. Parnas, D.L.: Software aging. In: Proceedings of 16th International Conference on Software Engineering, pp. 279–287. IEEE Computer Society Press. doi:10.1109/ICSE.1994.296790 (1994)
5. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012). doi:10.1109/MS.2012.167
6. Sterling, C.: Managing Software Debt: Building for Inevitable Change. Agile Software Development Series. Addison-Wesley Professional (2010). ISBN-10: 0321554132
7. Basili, V.R., Selby, R.W.: Comparing the effectiveness of software testing strategies. IEEE Trans. Softw. Eng. **SE-13**(12), 1278–1296 (1987). doi:10.1109/TSE.1987.232881
8. Basili, V.R., Green, S., Laitenberger, O., Shull, F., Sørumgård, S., Zelkowitz, M.V.: The empirical investigation of perspective-based reading. Retrieved from http://dl.acm.org/citation.cfm?id=241252 (1995)
9. Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grünbacher, P.: Value-Based Software Engineering (Google eBook), p. 388. Springer. Retrieved from http://books.google.com/books?id=CAlM6nNPcsgC&pgis=1 (2006)
10. McConnell, S.: 10x software development. Retrieved from http://forums.construx.com/blogs/stevemcc/archive/2007/11/01/technical-debt-2.aspx (2007)
11. Fowler, M.: Technical debt quadrant. Retrieved from http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html (2009)
12. Rothman, J.: An incremental technique to pay off testing technical debt. Retrieved from http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=11011&tth=DYN&tt=siteemail&iDyn=2 (2006)
13. Brown, N., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., Cai, Y., et al.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research – FoSER'10, p. 47. ACM Press, New York. doi:10.1145/1882362.1882373 (2010)
14. Zazworka, N., Shaw, M. A., Shull, F., Seaman, C.: Investigating the impact of design debt on software quality. In: Proceeding of the 2nd Working on Managing Technical Debt – MTD'11, p. 17. ACM Press, New York. doi:10.1145/1985362.1985366 (2011)
15. Dybå, T., Prikladnicki, R., Rönkkö, K., Seaman, C., Sillito, J.: Qualitative research in software engineering. Empirical Softw. Eng. **16**(4), 425–429 (2011). doi:10.1007/s10664-011-9163-y
16. Dittrich, Y., John, M., Singer, J., Tessem, B.: For the special issue on qualitative software engineering research. Inf. Softw. Technol. **49**(6), 531–539 (2007). doi:10.1016/j.infsof.2007.02.009
17. Lim, E., Taksande, N., Seaman, C.: A balancing act: what software practitioners have to say about technical debt. IEEE Softw. **29**(6), 22–27 (2012). doi:10.1109/MS.2012.130
18. Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F. Q. B., Santos, A. L. M., et al.: Tracking technical debt – an exploratory case study. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 528–531. IEEE. doi:10.1109/ICSM.2011.6080824 (2011)
19. Schumacher, J., Zazworka, N., Shull, F., Seaman, C., Shaw, M.: Building empirical support for automated code smell detection. In: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement – ESEM '10, p. 1. ACM Press, New York. doi:10.1145/1852786.1852797 (2010)

20. Hochstein, L., Shull, F., Reid, L.B.: The role of MPI in development time: a case study. In: 2008 SC – International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–10. IEEE. doi:10.1109/SC.2008.5213771 (2008)
21. Zazworka, N., Basili, V.R., Shull, F.: Tool supported detection and judgment of nonconformance in process execution. In: 2009 3rd International Symposium on Empirical Software Engineering and Measurement, pp. 312–323. IEEE. doi:10.1109/ESEM.2009.5315983 (2009)