Technical Credit

IAN GORTON*, Northeastern university, USA
ALESSIO BUCAIONI†, Mälardalen university, Sweden
PATRIZIO PELLICCIONE‡, Gran Sasso science institute (GSSI), Italy

ACM Reference Format:
Ian Gorton, Alessio Bucaioni, and Patrizio Pelliccione. 2024. Technical Credit. 1, 1 (August 2024), 5 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

Technical debt (TD) is an established concept in software engineering encompassing an unavoidable side effect of software development [1]. It arises due to tight schedules, which often prioritizes short-term delivery goals over long-term product quality concerns [1]. Even when long-term planning is feasible, the continuous evolution of requirements and technology platforms necessitates design decisions and code revision. Inevitably, even the best software designs will deteriorate over time, leading to TD [2].

Addressing TD is an essential task that entails continuous efforts to refactor code bases, update integrated third-party components, and resolve low-priority bugs [3]. Just like managing personal finances or parenting teenagers, a living software system requires constant attention to TD. Simply, TD is an unavoidable aspect of the software development process. TD has been a subject of extensive research and analysis within the software engineering community. In many ways, TD is the gift that keeps on giving for the research community, delivering an endless source of problems to study from the vast global ecosystem of the software industry. It is simply an intrinsic problem in software engineering that will never go away. Not while humans write code.

Surprisingly, the software engineering literature has yet to explore the opposite concept of TD, namely *Technical Credit* (TC). While TD creates friction that decreases a project's velocity over time, TC reduces development friction by greasing the wheels of evolution for a software system. Recognizing this gap in research, the primary objective of this viewpoint article is to introduce the concept of TC in the context of software engineering. In particular, we propose a definition of TC and present an abstract model to provide a comprehensive understanding of the concept. Furthermore, we provide real-world examples of TC and outline a set of research questions that must be addressed to effectively implement TC management in practice.

WHAT IS TECHNICAL CREDIT?

Essentially, TC characterizes system features that can yield long term benefits as the system evolves. By highlighting the benefits of strategic investment, TC contrasts with the conventional TD-driven focus on the drawbacks of sub-optimal choices. Rather, TC advocates for a paradigm shift towards recognizing the positive potential of forward-thinking

Authors' addresses: Ian Gorton, i.gorton@northeastern.edu, Northeastern university, USA; Alessio Bucaioni, alessio.bucaioni@mdu.se, Mälardalen university, Sweden; Patrizio Pelliccione, patrizio.pelliccione@gssi.it, Gran Sasso science institute (GSSI), Italy.

Manuscript submitted to ACM

decisions. We believe that framing TC as a counterbalance to the dominant technical debt-centric view can unlock a more nuanced, rounded understanding of a system's attributes, qualities, and its intrinsic value and investment.

While largely absent from the software engineering literature, TC was described by Berenbach in systems engineering [4]. Berenbach defines TC as extra effort put into designing, and building systems in anticipation of future benefits from *emergent properties* — systems engineering terminology for unknown future requirements. Essentially, all valuable systems evolve to add new functionalities. While the exact path of this evolution is unpredictable, some paths are likelier than others.

For example, an online payment system is unlikely to transform into an electronic health record system. Yet, it might need to adapt to new payment methods, support emerging payment devices and new countries, and so on. Hence, when developing applications within a particular domain, having knowledge about that domain, product plans and road-maps can provide valuable insights into the likely directions that the product may take. These insights make it possible for developers to strategically engineer capabilities into a product to reduce the cost and effort required for likely enhancements. These forward-thinking capabilities collectively embody a system's technical credit.

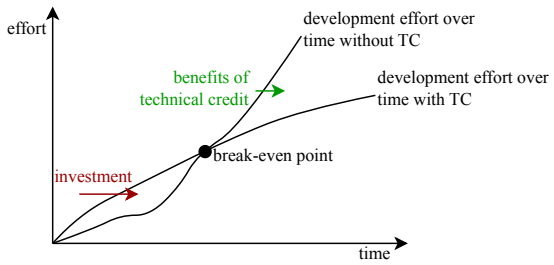

Fig. 1. Technical credit.

Technical credit is not a cost-free endeavor. It necessitates an initial investment to establish the necessary capabilities to enhance future modifications. Figure 1 illustrates that such an investment can lead to returns over time, once the break-even point is achieved. The key is to minimize the investment in technical credit while maximizing effort and cost saving that result from the technical credit investment downstream. In light of the above, we define technical credit as follows:

> *Technical credit is the benefits that result from making strategic design decisions that require a higher initial investment, but offer highly advantageous long-term effects for the evolution of a system.*

To exemplify this concept, imagine a system that relies on an external provider for email delivery. In this setup, there are two primary ways to structure the system: (i) each service interfaces directly with the email vendor's API, and (ii) the system uses an intermediary edge service that wraps the email vendor's API, with all services sending emails through this edge service.

In this scenario, using an edge service represents TC. With multiple email vendors in the market, their prices and features might change, necessitating a vendor switch. In such a case, updating the edge service to accommodate a new vendor is faster and more cost-efficient than altering each service's direct API calls to the email vendor. When you switch vendor, you cash in your technical credit. However, if you never switch email vendors, you never exploit TC, representing over-engineering or a YAGNI (You Ain't Gonna Need It) feature — something agilistas caution against.

This highlights why having deep knowledge of the domain, business, and product is essential as not every investment will yield returns, underscoring the importance of strategic planning.

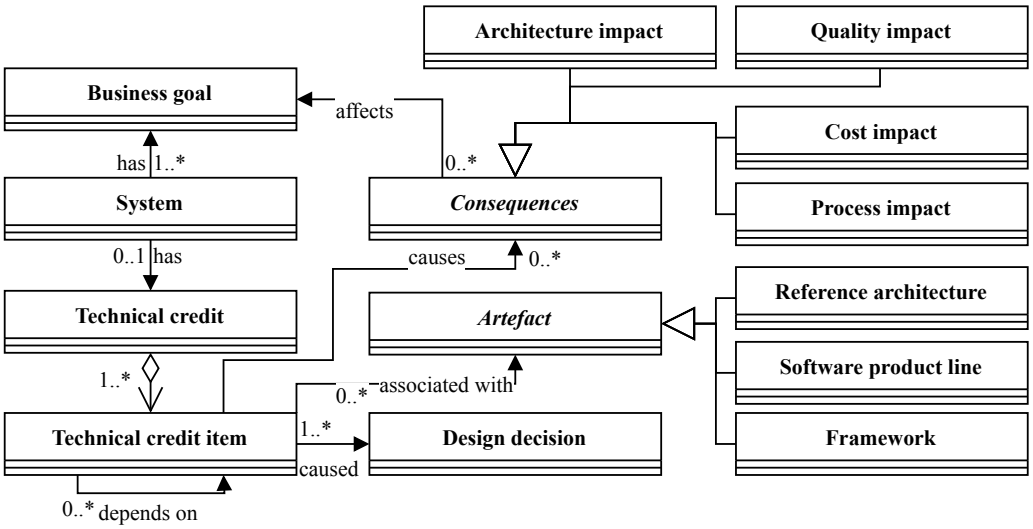

Fig. 2. Conceptual model of technical credit [5].

During the Dagstuhl Seminar 16162, Avgeriou et al. developed a conceptual model for TD [5]. Building on this, Figure 2 introduces a conceptual model for TC. TC comprises of a set of items, each with associated causes and consequences. The cause of TC is one or more design decisions. The consequences of TC impact the architecture and its qualities, the system costs, and development process. For example, a particular design decision may enhance the modifiability of a system, reducing the cost of future changes. Similarly. TC can affect the development process by reducing delays when prescriptive and informative instruments are utilized. A TC item relates to one or more specific, tangible artefacts. It should be noted that the above conceptual model does not account for approaches to TC measurement and analysis, the activities necessary to manage TC, or the various states that credit may go through.

TECHNICAL CREDIT IN PRACTICE

Decisions leading to TC are inherently connected to a system's business domain and product roadmap. However, not all design decisions yield true TC: those not enabling cost-effective modifications in line with business objectives might be deemed over-engineering. We create TC by crafting design decisions that ease future modifications anticipated by the product's roadmap. Thus, a design decision yields technical credit upon its utilization.

Drawing on extensive experience with product, architecture, and code reviews, we have identified examples of TC across different application domains.

Platform Abstraction Layer. This strategy creates abstraction layers around third-party products and libraries, protecting application code from external API changes. When these products need replacing, adjustments are made only to the abstraction layer, not the application code. This approach exemplifies TC in reducing the effort and cost of major modifications.

Architecture Fitness Function. A fitness function is an invariant that verifies the persistence of desired system properties against changes [6]. For example, a run-time fitness function might monitor response times to keep them under a set threshold, while a build-time fitness function might check for circular dependencies in the code. Fitness functions can be viewed as a form of TC, providing developers with automatic alerts to potential issues that could cause adverse effects down the line.

Stateless services. These services operate without maintaining conversational state, treating each request independently with request-specific state stored externally, like in a cache or database. They are a critical component of scalable applications as they allow the addition of service replicas and hardware instances without altering the code. Adopting stateless services creates TC by easing the scaling process to accommodate higher loads.

Architecture Decision Records (ADRs). ADRs document key architectural design decisions and the involved trade-offs, such as the results of a proof-of-concept prototype that led to the selection of a specific database over other candidates. ADRs simplify understanding a system's architecture, creating TC by streamlining planning for future changes.

Circuit Breakers. This pattern is fundamental to building resilient micro-services-based systems, acting as safeguards against transient performance degradation or complete failures of called services. Circuit breakers are TC as they guard against cascading failures caused by slow responses to requests that lead to congestion, overload and eventual failure in calling services. They ensure system resilience and facilitate its evolution.

Reference architectures (RAs). RAs capture the essence of the architecture of a collection of systems in a given domain, serving prescriptive, descriptive, and informative roles. They guide the development, standardization, and evolution of architectures within an application domain and aid decision-making by disseminating architectural knowledge. As such, RAs are a form of technical credit.

A RESEARCH AGENDA FOR TECHNICAL CREDIT

We believe TC warrants comprehensive investigation and scrutiny from the software engineering community. Analogous to technical debt, we see three broad areas of research to explore [5]: (i) defining, comprehending, and operationalizing the concept of value in relation to TC, (ii) investigating phenomena closely related to how TC operates in practice, and (iii) establishing a shared infrastructure that enables all our research activities. Some specific Research directions that can benefit from a more in-depth exploration of TC are discussed in the following.

- *Identifying technical credit - How should a project team decide where to invest in technical credit?* Answering this question involves careful evaluation of criteria such as potential architectural impact, cost-benefit analysis, and the likelihood of realizing the anticipated credit. All of these require insights into future product road-maps and plans, which are inherently uncertain. Developing a solution to this challenge would allow teams to identify design decisions that are most likely to yield high payback. The ICE Scoring Method[1], commonly used in product planning, offers a structured way to assess the impact, confidence, and ease of implementation for product

[1] https://www.productplan.com/glossary/ice-scoring-model/

features. Adapting a similar method could help in identifying the most promising areas for TC investment by quantifying their potential benefits.

- *Identifying technical credit - How to benefit from technical credit?* To realize the benefits of TC, it is important for teams to be able to easily identify which TC items are generating value and which ones are not. This information can help teams to better understand the impact of their decisions and to make more informed ones in the future.
- *Quantifying technical credit - How to measure technical credit?* Tools like SonarQube[2] can be used to quantify technical debt by estimating the effort to resolve it. In a similar way, teams need tools that can automatically recognize and value TC, helping to track its benefits and guide resource allocation for the greatest effect.
- *Deriving product value - How to measure the Return on Investment (ROI)?* If we can quantify both TD and TC, it may be possible to assign a value to individual design decisions, and to complete products. This could become a crucial factor in evaluating decisions and product ROI, along with other elements such as revenue and customer satisfaction. However, determining how to calculate this value requires further investigation and research.

In summary, we believe the concept of technical credit can positively impact the future of software engineering. This article aims to inspire the software engineering research community to explore the concept of technical credit. By identifying opportunities for practical implementation in conjunction with software practitioners, we can collectively bring this valuable concept into engineering practice.

REFERENCES

[1] W. Cunningham, The wycash portfolio management system, in: Addendum to the Proceedings on Object-Oriented Programming Systems, Languages, and Applications (Addendum), OOPSLA '92, Association for Computing Machinery, New York, NY, USA, 1992, p. 29–30. doi:10.1145/157709.157715.
URL https://doi.org/10.1145/157709.157715

[2] P. Kruchten, R. L. Nord, I. Ozkaya, Technical debt: From metaphor to theory and practice, Ieee software 29 (6) (2012) 18–21.

[3] M. Lemaire, Refactoring At Scale, " O'Reilly Media, Inc.", 2020.

[4] B. Berenbach, On technical credit, Procedia Computer Science 28 (2014) 505–512.

[5] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, Managing technical debt in software engineering (dagstuhl seminar 16162), in: Dagstuhl reports, Vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[6] N. Ford, R. Parsons, P. Kua, Building Evolutionary Architectures: Support Constant Change, 1st Edition, O'Reilly Media, Inc., 2017.

[2] https://www.sonarsource.com/products/sonarqube/