FOM Hochschule für Oekonomie & Management

university location Hamburg

Bachelor Thesis

in the study course Business Information Systems

to obtain the degree of

Bachelor of Science (B.Sc.)

on the subject

Avoiding Software Decay in Military Software Development

by

Karl Wedrich

Advisor:	Prof. Dr. Ulrich Schüler
Matriculation Number:	597615
Submission:	March 26, 2025

Contents

List of Figures

List of Tables

List of Abbreviations

CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Delivery
CD	Continuous Delivery
CPM	Customer Product Management
EVB-IT	Ergänzende Vertragsbedingung für die Beschaffung von IT-Leistungen
NRC	National Research Council
QA	Quality Assurance
TDD	Test Driven Development
XP	Extreme Programming

List of Symbols

1 Introduction

T This is the Introduction.

2 Literature Review

2.1 Problems of Software Engineering

Software has become an integral part of our daily lives. Certain expectations exist regarding the quality of software in terms of reliability, security and efficiency. These expectations come with challenges for the software developers across all industries. For decades, software engineers have tried to develop methods and guides to overcome these issues. However, in 1986 Frederick Brooks published his paper 'No Silver Bullet' in which he argues that: '. . . building software will always be hard. There is inherently no silver bullet.'[1]. He based this statement on the fact that there two types of difficulties in software development: the essential and the accidental. The essential difficulties he names are complexity, conformity, changeability and invisibility.

With complexity, Brooks wants to describe the inherit intricacy of software systems: 'Software entities are more complex for their size than perhaps any other human construct, because no two parts are alike'.[2] This complexity makes 'conceiving, describing, and testing them hard'[3].

The second essential difficulty Brooks names is conformity. To explain this, he compares software development to physics. Even though they are similarly complex, physics has the advantage of relying on a single set of laws or 'creator'. The same cannot be said for software engineers. Brooks claims that the complexity is 'arbitrary [. . .], forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform'[4]. This is due to software being perceived as 'the most comfortable'[5] element to change in a system.

Brooks explains the third issue, changeability, by comparing software to other products like cars or computers. With these types of products, changes are difficult to make once the product is released. Software however is just 'pure thought-stuff, infinitely malleable.'[6]

---

1 *Brooks*, *F.*, No Silver Bullet, 1987, p. 3.
2 Ibid., p. 3.
3 Ibid., p. 3.
4 Ibid., p. 4.
5 Ibid., p. 4.
6 Ibid., p. 4.

Another major issue regarding changeability is the fact that software often ‘survives beyond the normal life of the machine vehicle for which it is first written’[7]. This means that software has to be adapted to new machines causing an extended life time of the software.
Invisibility is the last essential difficulty Brooks names. With this he means the difficulty to visualize software compared to other products. This not only makes the creation difficult but also ‘severely hinders communication among minds’[8]. According to Brooks, these issues are in ‘the very nature of software’[9]. These difficulties are unlikely to be solved, in comparison with the accidental difficulties.

In contrast, the accidental difficulties arise from limitations of current languages, tools and methodologies. According to Brooks, this involves issues such as inefficient programming environments, suboptimal development processes and integration challenges which can be overcome as the industry improves its practices and technologies.[10] For example, the adaptation of agile methodologies, integrated development environments and continuous integration have helped to overcome some of these accidental difficulties.

The persistent nature of these challenges presented by Brooks have since been substantiated by further empirical research. For instance, Lehman and Ramil (2003) discussed in their paper ‘Software evolution—Background, theory, practice’ that software systems which are left unchecked will experience a decline in quality over time.[11] This phenomenon is encapsulated in Lehman’s laws of software evolution, which he formulated in multiple papers. Lehman and Ramil present empirical observations supporting the notion that software quality tends to deteriorate over time[12] - a phenomenon often described as software decay.

2.2 Software Decay and Technical Debt

The term software decay or erosion was empirically studied and statistically validated by Eick et al. in their influential paper ‘Does Code Decay? Assessing the Evidence from Change Management Data’(2001). They begin by stating that ‘software does not age or "wear out" in the conventional sense.’[13] If nothing in the environment changes, the software

[7] *Brooks*, *F.*, No Silver Bullet, 1987, p. 4.
[8] Ibid., p. 4.
[9] Ibid., p. 2.
[10] Ibid., pp. 5–6.
[11] *Lehman*, *M. M.*, *Ramil*, *J. F.*, Software Evolution, 2003, p. 34.
[12] Ibid., p. 42.
[13] *Eick*, *S. G.* et al., Does Code Decay?, 2001, p. 1.

could run forever. However, this is almost never the case as there is a constant change in several areas, predominantly with respect to the two areas of the hard- and software environments and the requirements of the software.[14]

The previous statement is in accordance with the first two laws of Program Evolution Dynamics formulated by Belady and Lehman (1976). The first law states: 'A system that is used undergoes continuing change until it is judged more cost effective to freeze and recreate it.'[15] Building on this, their second law declares: 'The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.'[16]

The analysis of Eick et al. provide empirical validation for these theoretical laws, offering 'very strong evidence that code does decay.'[17] This conclusion is based on their findings that 'the span of changes increases over time'[18] meaning that modifications to the software tend to effect increasingly larger parts of the system as the software evolves. This growth in the span of changes indicates and potentially leads to a breakdown in the software's modularity. Consequently the software becomes 'more difficult to change than it should be,'[19] measured specifically by three criteria: cost of change, time to implement change and the resulting quality of the software.[20] Therefore, the combination of theoretical insights from Lehman and Belday and empirical data from Eick et al. paints a clear picture: software decay is an inevitable consequence of ongoing evolution unless consciously and proactively managed through structured efforts such as continuous refactoring and architectural vigilance.

The concept of software decay aligns closely with earlier theoretical discussions by David Parnas (1994). In his influential paper 'Software Aging', Parnas describes software aging as a progressive deterioration of a program's internal quality primarily due to frequent, inadequately documented modifications which he termed 'Ignorant surgery'[21], as well as the failure to continuously adapt the architecture to evolving needs which he called 'Lack of movement'[22]. Without this proactive maintenance and refactoring effort, Parnas argues

[14] *Eick*, *S. G.* et al., Does Code Decay?, 2001, p. 1.
[15] *Belady*, *L.*, *Lehman*, *M.*, Large Program Development, 1976, p. 228.
[16] Ibid., p. 228.
[17] *Eick*, *S. G.* et al., Does Code Decay?, 2001, p. 7.
[18] Ibid., p. 7.
[19] Ibid., p. 3.
[20] Ibid., p. 3.
[21] *Parnas*, *D. L.*, Software Aging, 1994, p. 280.
[22] Ibid., p. 280.

that software inevitably reaches a state where changes become more risky, costly and error-prone[23].

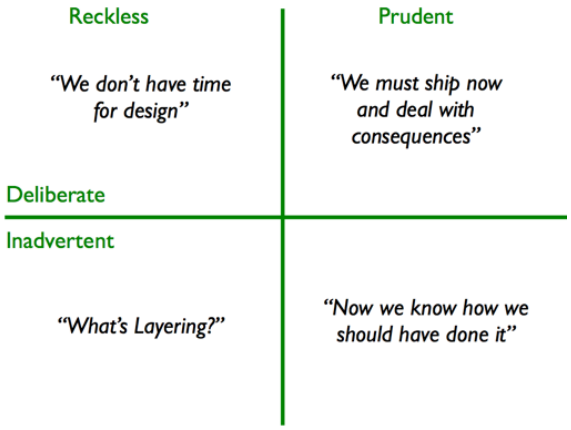
//TODO: Emperical studies like Banker or Bieman maybe?

The term 'Technical Debt' was first coined by Ward Cunningham in his paper 'The WyCash Portfolio Management System' (1992). This metaphor was used to describe the trade-off between a quickly implemented solution and a thought-out process. Using the quick solution 'is like going into debt.'[24] Cunningham argues that this debt accumulates interest if not repaid or rewritten. If this does not happen, Cunningham warns that 'Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation'[25].

This term was further built upon and refined by the industry through white papers like 'Technical Debt' by Steve McConnell (2008) or the 'Technical Debt Quadrant' by Martin Fowler (2009). McConnell differentiates between two types of technical debt: Unintentional and Intentional[26]. The first results from bad code, inexperience or unknowingly taking over a project with technical debt. The second type is taken on purpose 'to optimize for the present rather than for the future.'[27] As the first is not planned, it is difficult to avoid. The second type, however, can be managed and controlled.
Additionally, McConnell differentiates between different types of intentional debt. According to him, debt can be taken on a short-term or long-term basis. The short-term debt is taken on to meet a deadline or to deliver a feature. Therefore it is 'taken on tactically or reactively'[28]. The long-term debt on the other hand is more strategic and is taken on to help the team in a larger context. The difference between those two is that short-term debt 'should be paid off quickly, perhaps as the first part of the next release cycle'[29], while long-term debt can be carried by companies for years.

[23] *Parnas*, *D. L.*, Software Aging, 1994, pp. 280–281.
[24] *Cunningham*, *W.*, WyCash Portfolio, 1992, p. 2.
[25] Ibid., p. 2.
[26] *McConnell*, *S.*, Managing Technical Debt, 2017, p. 3.
[27] Ibid., p. 3.
[28] Ibid., p. 3.
[29] Ibid., p. 4.

Figure 1: Fowler's Technical Debt Quadrant



Martin Fowler, on the other hand, warned against taking on too much deliberate debt. He argues that 'Even the best teams will have debt to deal with as a project goes on - even more reason not to overload it with crummy code.'[30] He created a quadrant between reckless and prudent and deliberate and inadvertent debt. For Fowler, the difference between reckless and prudent is the way the debt is taken on. Reckless debt happens without an appropriate evaluation of the consequences, risking difficulties in the future. Alternatively, prudent debt is taken on with the trade-offs in mind and the knowledge of the future costs. Fowler differentiates between deliberate and inadvertent in a similar way to McConnell's differentiation between intentional and unintentional debt. The various combinations of these four elements in the quadrant results in four different approaches. Reckless and deliberate would mean quick solutions without considering the long-term impact. Reckless and inadvertent results in flawed design or implementation, either carelessly or unknowingly. Prudent and deliberate is purposefully taking on debt to gain a short-term advantage with plans of repayment and finally prudent and inadvertent means taking on debt due to lack of knowledge or experience.[31]

In their article 'Technical Debt: From Metaphor to Theory and Practice' (2012) Kruchten et al. criticize the concept of technical debt to be 'somewhat diluted lately'[32], stating that every issue in software development was called some form of debt. Therefore they set out to define 'a theoretical foundation'[33] for technical debt.
Kruchten et al. state that technical debt has become more than the initial coding shortcuts and rather encompasses all kinds of internal software quality comprises.[34] According to

[30] *Fowler*, *M.*, 2009.
[31] Ibid.
[32] *Kruchten*, *P.*, *Nord*, *R. L.*, *Ozkaya*, *I.*, Technical Debt, 2012, p. 18.
[33] Ibid., p. 19.
[34] Ibid., p. 19.

them, this includes architectural debt, ‘documentation and testing’[35] as well as requirements and infrastructure debt. All these debt types allow engineers to better discuss the trade-offs with stakeholders and to make better decisions.

TODO: More about Kruchten and Theory?

There have been many studies providing empirical evidence for the theoretical concepts of technical debt. Highly influential studies were undertaken by Potdar and Shihab (2014) as well as by Li et al. (2015).
In their study Potdar and Shihab analyzed four large open source projects to find self-admitted technical debt as well as the likelihood of debt being removed. They found that ‘self-admitted technical debt exists in 2.4% to 31% of the files.’[36] Additionally, they found that ‘developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of the self-admitted technical debt.’[37] They also discovered that ‘only between 26.3% and 63.5% of the self-admitted technical debt gets removed’[38]. This relatively low removal rate of self-admitted technical debt indicates a wider challenge: developers recognize the issues of their implementation, but defer remediation potentially leading to a major impact on long-term maintainability.
Another approach to provide empirical evidence towards technical debt was taken by Li et al. . They conducted a systematic mapping study to ‘get a comprehensive understanding of the concept of "technical debt"’[39], as well as obtaining an overview of the current research in the field. Areas of investigation included existing types of technical debt (TD), the effect of technical debt on software quality and quality attributes (QAs) as well as the limit of the technical debt metaphor.
They established that the ‘10 types of TD are requirements TD, architectural TD, design TD, code TD, test TD, build TD, documentation TD, infrastructure TD, versioning TD, and defect TD.’[40] Additionally they found that ‘[m]ost studies argue that TD negatively affects the maintainability [. . .] while other QAs and sub-QAs are only mentioned in a handful of studies’[41].
During their studies, Li et al. observed that the inconsistent and arbitrary use of the term ‘debt’ among researchers and practitioners can cause confusion and hinder effective

[35] *Kruchten*, *P.*, *Nord*, *R. L.*, *Ozkaya*, *I.*, Technical Debt, 2012, p. 20.
[36] *Potdar*, *A.*, *Shihab*, *E.*, Self-Admitted Debt, 2014, p. 1.
[37] Ibid., p. 1.
[38] Ibid., p. 1.
[39] *Li*, *Z.*, *Avgeriou*, *P.*, *Liang*, *P.*, Mapping of Technical Debt, 2015, p. 194.
[40] Ibid., p. 215.
[41] Ibid., p. 215.

management of technical debt.[42] Additionally practitioners 'tend to connect any software quality issue to debt, such as code smells debt, dependency debt and usability debt.'[43] This indicates an inflationary use of the term, which is important to keep in mind when speaking about technical debt.

The implications these studies have for the software industry are significant. They show that software decay and technical debt are tangible and measurable in real world software projects. In their paper 'Software complexity and maintenance costs' (1993) Banker et al. empirically demonstrated that 'software maintenance costs are significantly affected by the levels of existing software complexity.'[44] This finding emphasizes the important of proactively managing the software quality and addressing debt early in the project lifecycle, to keep the complexity and therefore cost to a minimum.
To address these effects, practitioners strongly recommend refactoring. Fowler argued in his book 'Refactoring: Improving the Design of Existing Code' (2019) that '[w]ithout refactoring, the internal design - the architecture - of software tend to decay.'[45] To prevent this, he suggests '[r]egular refactoring [to] help[s] keep the code in shape'[46].
To prevent long-term issues, practitioners recommend actively managing technical debt through refactoring, tracking and other strategies that integrate debt management into the software development process.

In summary, software decay describes the gradual deterioration of software quality over time, driven by continuous modifications, environmental changes and increasing complexity. This as theoretically established Belady, Lehman and Parnas and empirically validated by Eick et al. and others. Closely related yet conceptually distinct, technical debt captures the intentional and unintentional compromises made during software development. This leads to future costs if not proactively managed. While decay views a border spectrum of deterioration, technical debt specifically highlights how certain software engineering decisions increases the problem. Empirical research by Potdar and Shihab, Li et al., and Banker et al. underscores the tangible impact technical debt and software complexity have on real-world maintenance costs and software quality. Consequently, addressing software decay effectively in practice demands a combination of proactive quality management, regular refactoring, explicit technical debt tracking and structured maintenance strategies. With these foundational concepts clarified, the next section explores concrete mitigation

[42] *Li*, *Z.*, *Avgeriou*, *P.*, *Liang*, *P.*, Mapping of Technical Debt, 2015, p. 211.
[43] Ibid., p. 212.
[44] *Banker*, *R.* et al., Software Complexity, 1993, p. 12.
[45] *Fowler*, *M.*, Refactoring, 2019, p. 58.
[46] Ibid., p. 58.

strategies employed in commercial software environments, providing valuable insights into preventing software decay within the unique constraints of military software development.

2.3 Mitigation Strategies in Commercial Environments

Technical debt and software decay have been recognized as significant challenges in the software industry. They can lead to increased maintenance costs, reduced software quality and decreased developer productivity, overall leading to a more expensive and less competitive product. To address the challenges, practitioners and researchers have developed a variety of strategies to manage, prevent and mitigate technical debt. This section will provide an overview of the most common strategies, techniques and frameworks used in commercial environments to address technical debt.

2.3.1 High-Level Mitigation Strategies

To efficiently mitigate software decay and technical debt, proactive management strategies are essential. These strategies aim to prevent the accumulation of debt and address software entropy and decay directly by integrating quality assurance directly into everyday development process. Such strategies include Agile methodologies like Scrum or Extreme Programming (XP) as well as technical practices such as Continuous Integration/Continuous Delivery (CI/CD). These practices are designed to embed ongoing maintenance and quality assurance into routine workflows, thus combating software entropy at its core.

Agile methods emphasize frequent iterations, close collaboration between developers and stakeholders as well as continuous refactoring to prevent the gradual degradation of the software quality and mitigation software decay.

Similarly, CI/CD introduces rigorous automation, rapid feedback loops and early detection of defects to proactively controlling both technical debt accumulation and broader software quality decay. Collectively, these methodologies create a culture of continuous improvement, adaptability and quality assurance, ensuring software maintainability and long-term project sustainability.

2.3.1.1 Agile Methodologies

In their paper 'Technical Debt Management in Agile Software Development: A Systematic Mapping Study' (2024)[47] Leite et al. investigated how agile methods can be used to manage technical debt. They found that '... Scrum and Extreme Programming are the most utilized methodologies for managing technical debt.'[48] While this study focuses explicitly on technical debt, both Scrum and XP also inherently address the boarder issue of software decay by encouraging proactive quality management and continuous improvement practices.
Scrum was first presented by Ken Schwaber in his paper 'SCRUM Development Process' (1995)[49]. It has since become on of the most popular agile frameworks in the software industry. Scrum explicitly manages software quality and technical debt through iterative cycles called sprints. After each sprint, the team reflects on their work, identifies quality issues, technical debt and potential decay indicators and plans improvements in retrospectives. Leite et al. found that for identifying technical debt in Scrum the Sprint and Product Backlog are the most used artifacts.[50] By explicitly managing these items with their workflow, teams effectively reduce both debt and software entropy, improving overall software maintainability.

XP, first introduced by Kent Beck in his influential book 'Extreme Programming Explained' (1999)[51], explictly integrates practices to enhance software quality and prevent software decay directly. XP practices such as pair programming, Test Driven Development (TDD) and Continuous Integration (CI) and continuous refactoring help maintain high software quality, thus preventing both debt accumulation and broader software entropy. Pair programming prevents decay by ensuring higher-quality code through collaborative review and knowledge sharing between developers. Continuous Integration ensures regular, frequent code integration, significantly reducing integration complexity and associated decay risks. TDD ensures robust test coverage, catching defects early and preventing quality erosion. Refactoring, a cornerstone XP practice, has proven its effectiveness empirically; for example, Moser et al. demonstrated in their case study(2008)[52] that refactoring explicitly 'prevents an explosion of complexity'[53] and promotes simpler, easier-to-maintain designs. They found it drove developers toward simpler designs, reducing complexity, coupling, and

[47] *Leite*, *G. d. S.* et al., Technical Debt Management in Agile Software Development, 2024.
[48] Ibid., p. 318.
[49] *Schwaber*, *K.*, SCRUM Process, 1997.
[50] *Leite*, *G. d. S.* et al., Technical Debt Management in Agile Software Development, 2024, p. 315.
[51] *Beck*, *K.*, Extreme Programming Explained, 1999.
[52] *Moser*, *R.* et al., Case Study Refactoring, 2008.
[53] Ibid., p. 262.

long-term maintenance issues—directly counteracting software decay.[54] Beck argues that XP's incremental, continuous quality practices consistently maintain software quality and adaptability throughout development, directly addressing both debt and broader software decay.
Overall Agile methodologies, particularly Scrum and XP, systematically manage technical debt and proactively prevent software decay by fostering continuous improvement, structured quality management and adaptability. Complementing these Agile practices, the adoption of automated CI/CD pipelines further enhances the proactive management of both technical debt and broader software decay through rigorous quality control and systematic automation.

2.3.1.2 Continuous Integration/Continuous Deployment

CI was first introduced by Beck in the context of XP and later refined by Martin Fowler in his influential article 'Continuous Integration'[55]. Fowler describes CI as not only the frequent, automated integration of code into the main repository but also the systematic automation of building and testing process. According to Fowler, 'Self-testing code is so important to Continuous Integration that it is a necessary prerequisite.'[56]. Furthermore, another critical prerequisite is 'that they can correctly build the code,'[57] thus guaranteeing that code changes consistently integrate without issues.

To further prevent technical debt and broader software decay, a quality analysis tools (e.g. static code analyzers such as SonarQube or DeepSource) are frequently integrated into CI pipelines. These tools often provide a metric to evaluate technical debt which is calculated based on the effort in minutes to fix the found maintainability issues.[58] In their paper 'Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study' (2021)[59] Murillo et al. found that SonarQube is a useful tool for early debt detection. The estimated remediation effort metric allows for a good debt management prioritization.[60] However during their research they noticed if they changed SonarQubes default rules by just 26 rules, the technical debt effort would increase from 1

[54] *Moser*, *R.* et al., Case Study Refactoring, 2008, p. 262.
[55] *Fowler*, *M.*, 2006.
[56] Ibid.
[57] Ibid.
[58] *SonarQube*, 2025.
[59] *Murillo*, *M. I.*, *Jenkins*, *M.*, Technical Debt Measurement during Software Development Using Sonarqube, 2021.
[60] Ibid., p. 5.

hours and 50 minutes to 11 hours.[61] This and the fact that SonarQube can only detect code related debt and not for example infrastructure or requirements debt, makes these tools useful but not a complete solution.

Continuous Delivery (CD), introduced by Jez Humble and David Farley in their foundational book 'Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation' (2010)[62] extends CI by automating the entire software release pipeline. CD ensures that the software is always in a releasable state. According to Humble and Farley implementing a functional CD pipeline 'creates a release process that is repeatable, reliable, and predictable'[63]. Beyond predictability, additional significant benefits include team empowerment, deployment flexibility and substantial error reduction. Specially, CD effectively reduces errors, particularly those introduced by poor configuration management, including problematic areas such as 'configuration files, scripts to create databases and their schemas, build scripts, test harnesses, even development environments and operating system configurations'[64].

2.3.2 Code-Level Practices for Preventing Decay

On the code level, a variety of practices can be used to prevent software decay and technical debt. The two major have been previously introduced: continuous refactoring and maintaining software quality.
The benefits of refactoring have been demonstrated int the previous section. However, deeper empirical studies illustrate the impacts and challenges in a commercial environment. Kim et al. (2014) observed in their study 'An empirical study of refactoring challenges and benefits at Microsoft' observed that '[d]evelopers perceive that refactoring involves substantial cost and risks'[65]. Additionally they found that the benefits refactoring brings 'multidimensional'[66] and that its not consistent across different metrics, which is why they recommended a tool to monitor the impact of refactoring across these metrics. Similar, Tempero et al. (2017) found certain barriers to refactoring in their study 'Barriers to refactoring'. They found that at least 40% of the developers in their study would not refactor

[61] *Murillo*, *M. I.*, *Jenkins*, *M.*, Technical Debt Measurement during Software Development Using Sonarqube, 2021, p. 4.
[62] *Humble*, *J.*, *Farley*, *D.*, Continuous Delivery, 2010.
[63] Ibid., p. 17.
[64] Ibid., p. 19.
[65] *Kim*, *M.*, *Zimmermann*, *T.*, *Nagappan*, *N.*, Refactoring Challenges, 2014, p. 17.
[66] Ibid., p. 17.

classes, even though they thought it would be beneficial.[67] Tempero et al. claims the reasons were 'lack of resources, of information identifying consequences, of certainty regarding risk, and of support from management.'[68] even though developers did not state a lack of refactoring tools as a reason. Tempore et al. suggested to eliminate these barriers, refactoring should be goal-oriented instead of operations-oriented and a better quantification of the benefits refactoring should bring to better reach a decision.[69]
Both studies show that the theoretical benefits of refactoring are not always directly translated into the industry. Developers often perceive refactoring as risky and costly, even though they see the benefits.

TDD is another code-level practice previously introduced. It was first formally described by Kent Beck in his book 'Test Driven Development: By Example' (2002) in which he argues that writing tests before the implementation encourages cleaner code and gives developers the confidence to tackle complex problems.[70] This practice promotes modularity, testability and clean design, which are key characteristics in preventing software decay. By focusing solely on code that fulfills predefined tests, developers are guided toward creating small and focused code, which is loosely coupled. These qualities make systems easier to maintain and refactor, directly counteracting long-term degradation. Multiple studies have confirmed the quality-enhancing effects of TDD. In an empirical study, Mäkinen and Münch (2014) found that TDD most commonly led to a reduction in defects and increased maintainability, though its effect on overall software quality was more limited.[71] Importantly, they noted that TDD significantly improved test coverage, a key factor in preventing unintentional decay. These gains, however, were accompanied by higher development effort.[72]
This is consistent with the case study by Bhat and Nagappan (2006), who reported a 15-35% increase in development time when using TDD.[73] However they also observed that 'the resulting quality was higher than teams that adopted a non-TDD approach by an order of at least two times.'[74] Similarly, Bhadauria et al. (2020) confirmed TDD's positive effect on code quality and defect reduction. Interestingly, their study found that for less experienced developers, TDD did not lead to increased development time, and was associated with higher developer satisfaction.[75]

[67] *Tempero*, *E.*, *Gorschek*, *T.*, *Angelis*, *L.*, Barries to Refactoring, 2017, p. 60.
[68] *Tempero*, *E.*, *Gorschek*, *T.*, *Angelis*, *L.*, Barries to Refactoring, 2017, p. 60.
[69] Ibid., p. 61.
[70] *Beck*, *K.*, Test-Driven Development, 2002, pp. 8-9.
[71] *Mäkinen*, *S.*, *Münch*, *J.*, Effects of TDD, 2014, p. 13.
[72] Ibid., p. 13.
[73] *Bhat*, *T.*, *Nagappan*, *N.*, Evaluating the Efficacy of Test-Driven Development, 2006, p. 361.
[74] Ibid., p. 361.
[75] *Bhadauria*, *V.*, *Mahapatra*, *R.*, *Nerur*, *S.*, Performance Outcomes of Test-Driven Development, 2020, p. 1058.

These findings indicate that TDD can significantly improve code quality and maintainability - two critical factors in preventing software decay. However, the trade-off in development effort and time must be carefully considered, which may explain TDD's limited adoption in the industry.

2.3.3 Architectural Strategies for Long-Term Quality

In addition to code-level practices, maintaining the software architecture is essential for preventing software decay and architectural-level technical debt. Architecture erosion, characterized by increasing divergence from the intended design due to continuous modifications and changing requirements, significantly impacts maintainability and introduces substantial technical debt.[76] To mitigate these risks, one common strategy is architecture conformance checking, a process ensuring code changes comply with predefined design rules. De Silva and Balasubramaniam (2012) emphasize that proper documentation, dependency analysis and compliance monitoring are critical prerequisites for effectively employing this strategy.[77]

Beyond mere conformance, teams also must embrace managed architectural evolution, adapting architectures systematically rather than reactively. Such evolution should actively balance new requirements with long-term sustainability to avoid uncontrolled erosion.[78] A key component in this proactive approach is managing the architectural technical debt. These structural design decision could impact future adaptability if not carefully controlled. Besker et al. (2016) suggest a framework to systematically identify, analyze and address architectural debt to preserve structural integrity.[79]

Nevertheless, when architectural decay becomes substantial, teams face critical decisions between incremental refactoring and radical redesigns. Nord et al. (2012) highlight that systematic impact analysis, guided by explicit architectural metrics, is crucial for making informed decisions on when substantial redesign becomes necessary to substantially restore software quality.[80]

Practical tooling, such as automated architectural monitoring and recovery solutions, plays a significant supportive role in these efforts. The detailed application and empirical evidence of such tools are discussed thoroughly in the subsequent section on Tool Support

[76] *De Silva*, *L. R.*, *Balasubramaniam*, *D.*, Controlling Software Architecture Erosion, 2012, p. 1.
[77] Ibid., p. 135.
[78] *Li*, *R.* et al., Understanding Software Architecture Erosion, 2022, p. 34.
[79] *Besker*, *T.*, *Martini*, *A.*, *Bosch*, *J.*, Managing Architectural Technical Debt, 2018, p. 11.
[80] *Nord*, *R. L.* et al., Managing Architecrual Technical Debt, 2012, p. 99.

for Continuous Quality Assurance.

In summary, proactively maintaining software architecture is essential for mitigating long-term software decay. Architecture conformance checking prevents the inadvertent erosion by ensuring adherence to established design principles. Managed architectural evolution further ensures that the architecture can sustainably adapt to changing requirements without introducing uncontrolled structural degradation. Moreover, explicitly recognizing and managing architectural technical debt enables targeted interventions before significant decays occurs. Ultimately, strategic decisions on refactoring versus comprehensive redesign should be guided by systematic architectural analysis, metrics and empirical evaluations to ensure long-term maintainability and quality.

2.3.4 Process and Organizational Measures

Organizational and process-oriented practices form the third pillar in combating software decay. One crucial practice is the management of technical debt. Many companies track technical debt items like outdated modules, quick fixes or known architectural shortcomings. This can be done through issue trackers or backlogs to allow for structured approach and time allocation to address them regularly. An industry-wide survey conducted by Ramač et al. (2021) found that 47% 'had some practical experience with TD identification and/or management'[81]. By visualizing and tracking technical debt, teams can prioritize and address debt items systematically, preventing their accumulation and broader software decay. Ramač et al. also found that the most common cause for technical debt was time pressure caused by deadlines[82] and the 'single most cited effect of TD is delivery delay.'[83] This indicates that time pressure is not only a cause for technical debt but also a consequence, leading to a vicious cycle of debt accumulation and delivery delays. To break this cycle, a balance of new development and maintenance is crucial to prevent a border software decay. This is promoted by agile methodologies through principle of continuous improvement, through practices like including refactoring in their definition of done or allowing dedicated maintenance sprints.

Another important practice is conducting regular code reviews as part of the development workflow. This practice dates back to 1976 when Michael Fagan reviewed the benefits of peer code inspections.[84] He found that 'inspections increase productivity and improve

[81] *Ramač, R.* et al., Prevalence, Common Causes and Effects of Technical Debt, 2021, p. 40.
[82] Ibid., p. 40.
[83] Ibid., p. 40.
[84] *Fagan, M. E.*, Code Inspections, 1976, p. 183.

final program quality.'[85] Since then code reviews have become more lightweight compared to the inspection proposed by Fagan, increasing participation. By removing in-person meetings and reviewer checklists, code reviews have become a standard practice in software development and usually occur before code is merged into the main repository. McIntosh et al. (2016) investigated the impact of code reviews on software quality, by comparing the review coverage, participation and expertise of the reviewer against the post-release defects.[86] They found that coverage, while important, is not the only factor that influences the post-release defects.[87] They state that 'review participation should be considered when making integration decisions.'[88] Additionally, they recommend that if an expert in the matter is not available for the original code, they should be included in the review process to prevent defects.[89]

In conclusion, having a structured approach to managing technical debt and prevent software decay is crucial to maintaining software quality and long-term project sustainability. Process-integrated practices like code reviews, explicit technical debt control and a culture of continuous refactoring create an environment that proactively manages software decay and technical debt.

2.3.5 Tool Support for Continuous Quality Assurance

To further support the proactively manage software decay, technical debt and the overall code health over time, a variety of tools have established themselves in the industry. Automated quality assurance tools are often integrated into modern development processes. A common approach, as previously mentioned, is the use of CI pipelines. These can be used in combination with quality gates that use static code analysis to block additions to the code base that do not meet the quality standards e.g. a certain code coverage, complexity or maintainability threshold. This allows developers to catch issues early before they are introduced into the code base and rectify them. While the concept of quality gates is not new, the automation of these gates have been investigated by Uzunova et al. (2024). They found that these gates can serve as a check point to assess software metrics like code coverage, bug density or compliance with coding standards.[90] To allow for this kind of automation, they mentioned tools like SonarQube, Sigrid or Maverix.ai. These tools allow for real-time feedback allowing for an enforcement of quality criteria.[91]

[85] Ibid., p. 205.
[86] *Mcintosh*, *S.* et al., Impact Code Review, 2016, p. 6.
[87] Ibid., p. 39.
[88] Ibid., p. 39.
[89] Ibid., p. 39.
[90] *Uzunova*, *N.*, *Pavlič*, *L.*, *Beranič*, *T.*, Quality Gates, 2024, p. 8.
[91] Ibid., p. 8.

Utilizing this approach allows the developers to catch issues early and prevent the introduction of technical debt and software decay into the codebase.

As discussed in a previous section, ensuring that changes do not divert from the original architecture design is crucial to prevent the erosion of the architecture. To support this, tools have been developed to automatically detect architecture violations. These tools are able to collect information about the architecture from the source code and compare it to the proposed architecture.[92] In their case study 'Evaluating an Architecture Conformance Monitoring Solution' (2016) Caracciolo et al. investigated three different tools to detect architecture violations: SonarQube, Sonargraph and TeamCity[93]. They concluded that their approach was 'can be applied in an industrial context.'[94] They furthermore argue that the tools can be conveniently added to a dashboard, allowing for a short feedback loop, allowing proactive management of architectural violations.[95] These benefits can be seen in their case study, where they observed that the overall violations decreased from 606 to 600 in an 18 year old system with one million lines of code.[96].

There have been empirical studies that show the effectiveness of these tools. For example in their paper 'Empirical investigation of the influence of continuous integration bad practices on software quality' (2022) Silva and Bezerra found that 'CI can improve software quality, especially about cohesion.'[97] However they also viewed the impact of bad practices in the implementation of CI. They could observe that the quality levels were incorrectly set and the standard configuration tools were used instead of adjusting them to the project needs, overall harmed the software quality indicators.[98] This indicates that while the tools can be effective, they need to be correctly configured and adjusted to the project needs to be effective.

2.3.6 Synthesis of Commerical Mitigation Strategies

Effectively managing software decay and technical debt requires a balanced combination of technical architectural and organizational strategies. At the highest level, Agile

[92] *Thomas*, *J.*, *Dragomir*, *A. M.*, *Lichter*, *H.*, Architecture Conformance Checking, 2017, p. 6.
[93] *Caracciolo*, *A.* et al., Evaluating Architecture Monitoring, 2016, p. 43.
[94] Ibid., p. 44.
[95] Ibid., p. 44.
[96] Ibid., p. 43.
[97] *Silva*, *R. B. T.*, *Bezerra*, *C.*, CI Bad Practices, 2022, p. 4.
[98] Ibid., p. 4.

methodologies and CI/CD frameworks foster proactive maintenance cultures, reducing entropy through incremental improvement and automated quality assurance. On the code-level, disciplined practices such as continuous refactoring and TDD have empirically demonstrated their effectiveness in maintaining modularity, testability and reducing technical debt - with the downside of a higher development effort. Architectural strategies complement these by ensuring structural integrity through systematic conformance checks, managed evolution and explicit handling of architectural technical debt, especially as projects scale or requirements evolve.

Organizational measures - such as technical debt tracking, dedicated maintenance cycles and structured code reviews - provide essential process-level support, enabling teams to systematically prioritize and address decay risks before they escalate. Finally, the strategic use of modern quality-assurance tools, integrated into CI pipelines and architectural monitoring systems, ensures continuous, automated and actionable feedback, thereby enabling developers to maintain software quality proactively. Ultimately, integrating these diverse practices into a combined quality culture is vital for sustainable software development, reduced maintenance costs and the long-term viability of software products.

2.4 Unique Constraints in Military Software Development

2.4.1 Introduction to Military Software Context

Software development for military significantly differs from commercial software due to unique demands for high reliability, rigorous security, and extended system lifecycles. The National Research Council (NRC) book 'Reliability Growth: Enhancing Defense System Reliability' (2015) highlights the difference between commercial and military software development. According to them there are three main differences. The first is about the 'sheer size and complexity of defense systems'[99]. These systems often have a vast amount of individual elements that need to work together, that also evolve over time, as the architecture changes between the different stages of the lifecycle.[100] In addition, the NRC highlights the fact that new systems have to interact with legacy systems, which can be a challenge due to the different technologies and architectures used.[101] The second difference are the different perspectives of the actors involved. Unlike in the commercial environment, where it is usually only the project manager that controls the vision and the

[99] *National Research Council*, Comparison Defense Commerical, 2015, p. 31.
[100] Ibid., p. 32.
[101] Ibid., p. 32.

goals of the project, in the military environment there are multiple stakeholders with different goals in mind.[102] even though the book specially talks about the American military, the same can be said for the German military. The third difference the authors name are the concerns of risk. In the commercial environment, the manufacturer has a self interest in the product being successful and reliable. In the military environment however, the government is the customer and often holds the most risk, as they are the ones that have to use the product in the end. The NRC claims that this result in 'the system developers do not have a strong incentive to address reliability goals early in the acquisition process'[103]. especially because the downstream benefits are not quantifiable for the developers.[104]
In her presentation on 'A perspective on military software needs' Heidi Shyu highlights additional challenges in military software development. She argues that in addition to the complex setting, there are often a multitude of contractors working together, having to interoperate in real time with each other.[105] Often these projects have a lifecycle of decades compared to the commercial software lifecycle of a few years.[106] Due to these long lifecycles, the software needs to developed so it can be easily maintained and updated, while still working with legacy systems which often have very specific requirements.[107] Therefore, Heidi Shyu claims, the architecture has to last for decades, however soft- and hardware changes much faster, often resulting in a patchwork of quick fixes.[108] She names examples for needed software solutions like a flexible architecture which allows for easy updates and addition, self-testing modular code and intuitive, easy to use interfaces.[109]

In summary, these challenges in military software development are unique to the industry. The vast size and complexity of defense systems, their long lifecycles and the multitude of stakeholders involved create a complex environment that requires specialized strategies to manage software decay and technical debt effectively.

2.4.2 Regulatory and Procurement Constraints in the Bundeswehr

The procurement and regulatory environment significantly shapes military software development in the German Armed Forces (Bundeswehr). Unlike commercial settings, Bundeswehr software projects are tightly governed by formalized frameworks, rigorous

[102] *National Research Council*, Comparison Defense Commerical, 2015, p. 32.
[103] Ibid., p. 33.
[104] Ibid., p. 33.
[105] *Shyu*, *H.*, Military Software Needs, 2017, p. 11.
[106] Ibid., p. 14.
[107] Ibid., p. 14.
[108] Ibid., p. 15.
[109] Ibid., p. 17.

approval processes and comprehensive regulations designed primarily to ensure accountability, security, and reliability. However, these structures can inadvertently limit agility, prolong software updates and contribute significantly to software decay and technical debt over time.

Bundeswehr software utilizes the 'V-Modell XT Bw' as the project lifecycle model. This model is a variant to the V-Modell XT and is specifically tailored to the needs of the Bundeswehr and is linked to the Customer Product Management (CPM) process of the Bundeswehr. The V-Modell XT Bw is a comprehensive framework that encompasses the planning and execution of software projects in the Bundeswehr.[110] It includes detailed guidelines for all parts of the software development process, from roles and requirements engineering to Quality Assurance (QA) and product quality.[111] However, due to its rigid and sequential nature, the waterfall-based V-Modell XT Bw is known to limit flexibility, making iterative adjustments and agile methodologies challenging to implement.

The procurement procedures of the Bundeswehr are outlined by the aforementioned CPM which was revised in 2012 after a comission in 2010 found that TODO: Zitat aus der 2010 kommission. Due to these problems, the revised CPM has an increased focus on upfront requirement definition and comprehensive risk assessments. This creates long lead times for approval and makes quick technological changes difficult to realize. In addition, the Ergänzende Vertragsbedingung für die Beschaffung von IT-Leistungen (EVB-IT), the standardized contracts for government IT solutions, hinders agile development. This is according to Holger Schröder in an article from 2022 where he claims that multiple different contract blueprints would have to be combined to allow for an agile development project. TODO: zitat Schröder

Collectively, all these requirements and procurement constraints strongly influence software development practices. Control and predictability are prioritized, making a iterative and agile framework difficult to use. While accountability and risk reduction are realized, these regulations can significantly complicate long-term software adaptability and maintenance. Adding to this complexity are strong security and compliance requirements, which are discussed in the following subsection.

[110] *Bundeswehr*, V-Modell Bw, 2025, p. 6.
[111] Ibid., pp. 20-21.

2.4.3 Security and Compliance Requirements

Military software must meet stringent security and compliance standards beyond the usual commercial requirements. Due to the sensitive nature of military operations and data, the software must adhere to both national and NATO standards. TODO: check if nato needs to be spelled out These strict requirements create sever difficulties towards the lifecycle of the product, the software architecture and maintenance practices. These constrains accelerate the accumulation of technical debt and software decay.

Appendix

Bibliography

Banker, *Rajiv*, *Datar*, *Srikant*, *Kemerer*, *Chris*, *Zweig*, *Dani* (Software Complexity, 1993): Software Complexity and Maintenance Costs, in: Communications of the ACM, 36 (1993), Nr. 11

Beck, *Kent* (Extreme Programming Explained, 1999): Extreme Programming Explained: Embrace Change, s.l.: Addison-Wesley Professional, 1999

Beck, *Kent* (Test-Driven Development, 2002): Test-Driven Development : By Example, s.l.: Boston : Addison-Wesley, 2002-11

Belady, *L.*, *Lehman*, *M.* (Large Program Development, 1976): A Model of Large Program Development, in: IBM Systems Journal (1976)

Besker, *Terese*, *Martini*, *Antonio*, *Bosch*, *Jan* (Managing Architectural Technical Debt, 2018): Managing Architectural Technical Debt: A Unified Model and Systematic Literature Review, in: Journal of Systems and Software, 135 (2018), pp. 1–16

Bhadauria, *Vikram*, *Mahapatra*, *Radha*, *Nerur*, *Sridhar* (Performance Outcomes of Test-Driven Development, 2020): Performance Outcomes of Test-Driven Development: An Experimental Investigation, in: Journal of the Association for Information Systems, 21 (2020), Nr. 4

Bhat, *Thirumalesh*, *Nagappan*, *Nachiappan* (Evaluating the Efficacy of Test-Driven Development, 2006): Evaluating the Efficacy of Test-Driven Development: Industrial Case Studies, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, ISESE '06, s.l.: Association for Computing Machinery, 2006-09, pp. 356–363

Brooks, *Frederick* (No Silver Bullet, 1987): No Silver Bullet Essence and Accidents of Software Engineering, in: Computer, 20 (1987), Nr. 4, pp. 10–19

Bundeswehr (V-Modell Bw, 2025): V-Modell XT Bw, s.l.

Caracciolo, *Andrea*, *Lungu*, *Mircea*, *Truffer*, *Oskar*, *Levitin*, *Kirill*, *Nierstrasz*, *Oscar* (Evaluating Architecture Monitoring, 2016): Evaluating an Architecture Conformance Monitoring Solution, in: 2016 7th International Workshop on Empirical Software Engineering in Practice (IWESEP), s.l., 2016-03, pp. 41–44

Cunningham, *Ward* (WyCash Portfolio, 1992): The WyCash Portfolio Management System, in: SIGPLAN OOPS Mess. 4 (1992), Nr. 2, pp. 29–30

De Silva, *Lakshitha Ramesh*, *Balasubramaniam*, *Dharini* (Controlling Software Architecture Erosion, 2012): Controlling Software Architecture Erosion: A Survey, in: Journal of Systems and Software, 85 (2012), Nr. 1, pp. 132–151

Eick, *Stephen G.*, *Graves*, *Todd L.*, *Karr*, *Alan F.*, *Marron*, *J. S.*, *Mockus*, *Audris* (Does Code Decay?, 2001): Does Code Decay? Assessing the Evidence from Change Management Data, in: IEEE Trans. Softw. Eng. 27 (2001), Nr. 1, pp. 1–12

Fagan, *Michael E.* (Code Inspections, 1976): Design and Code Inspections to Reduce Errors in Program Development, in: IBM Syst. J. 15 (1976), Nr. 3, pp. 182–211

Fowler, *Martin* (Refactoring, 2019): Refactoring: Improving the Design of Existing Code, s.l.: Addison-Wesley, 2019

Humble, *Jez*, *Farley*, *David* (Continuous Delivery, 2010): Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, s.l.: Addison-Wesley, 2010

Kim, *Miryung*, *Zimmermann*, *Thomas*, *Nagappan*, *Nachiappan* (Refactoring Challenges, 2014): An Empirical Study of RefactoringChallenges and Benefits at Microsoft, in: IEEE Transactions on Software Engineering, 40 (2014), Nr. 7, pp. 633–649

Kruchten, *Philippe*, *Nord*, *Robert L.*, *Ozkaya*, *Ipek* (Technical Debt, 2012): Technical Debt: From Metaphor to Theory and Practice, in: IEEE Software, 29 (2012), Nr. 6, pp. 18–21

Lehman, *Meir M.*, *Ramil*, *Juan F.* (Software Evolution, 2003): Software Evolution—Background, Theory, Practice, in: Information Processing Letters, To Honour Professor W.M. Turski's Contribution to Computing Science on the Occasion of His 65th Birthday, 88 (2003), Nr. 1, pp. 33–44

Leite, *Gilberto de Sousa*, *Vieira*, *Ricardo Eugênio Porto*, *Cerqueira*, *Lidiany*, *Maciel*, *Rita Suzana Pitangueira*, *Freire*, *Sávio*, *Mendonça*, *Manoel* (Technical Debt Management in Agile Software Development, 2024): Technical Debt Management in Agile Software Development: A Systematic Mapping Study, in: Proceedings of the XXIII Brazilian Symposium on Software Quality, SBQS '24, s.l.: Association for Computing Machinery, 2024-12, pp. 309–320

Li, *Ruiyin*, *Liang*, *Peng*, *Soliman*, *Mohamed*, *Avgeriou*, *Paris* (Understanding Software Architecture Erosion, 2022): Understanding Software Architecture Erosion: A Systematic Mapping Study, in: Journal of Software: Evolution and Process, 34 (2022), Nr. 3, e2423

Li, *Zengyang*, *Avgeriou*, *Paris*, *Liang*, *Peng* (Mapping of Technical Debt, 2015): A Systematic Mapping Study on Technical Debt and Its Management, in: Journal of Systems and Software, 101 (2015), pp. 193–220

Mäkinen, *Simo*, *Münch*, *Jürgen* (Effects of TDD, 2014): Effects of Test-Driven Development: A Comparative Analysis of Empirical Studies, in: vol. 166, Lecture Notes in Business Information Processing, s.l., 2014-01

McConnell, *Steve* (Managing Technical Debt, 2017): Managing Technical Debt, tech. rep., s.l., 2017-02, chap. Resources

Mcintosh, *Shane*, *Kamei*, *Yasutaka*, *Adams*, *Bram*, *Hassan*, *Ahmed E.* (Impact Code Review, 2016): An Empirical Study of the Impact of Modern Code Review Practices on Software Quality, in: Empirical Softw. Engg. 21 (2016), Nr. 5, pp. 2146–2189

Moser, *Raimund*, *Abrahamsson*, *Pekka*, *Pedrycz*, *Witold*, *Sillitti*, *Alberto*, *Succi*, *Giancarlo* (Case Study Refactoring, 2008): A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, in: *Meyer*, *Bertrand*, *Nawrocki*, *Jerzy R.*,

Walter, *Bartosz* (eds.), Balancing Agility and Formalism in Software Engineering, vol. 5082, s.l.: Springer Berlin Heidelberg, 2008, pp. 252–266

Murillo, *María Isabel*, *Jenkins*, *Marcelo* (Technical Debt Measurement during Software Development Using Sonarqube, 2021): Technical Debt Measurement during Software Development Using Sonarqube: Literature Review and a Case Study, in: 2021 IEEE V Jornadas Costarricenses de Investigación En Computación e Informática (JoCICI), s.l., 2021-10, pp. 1–6

National Research Council (Comparison Defense Commerical, 2015): Defense and Commercial System Development: A Comparison, in: Reliability Growth: Enhancing Defense System Reliability, s.l.: The National Academies Press, 2015, pp. 19–33

Nord, *Robert L.*, *Ozkaya*, *Ipek*, *Kruchten*, *Philippe*, *Gonzalez-Rojas*, *Marco* (Managing Architecrual Technical Debt, 2012): In Search of a Metric for Managing Architectural Technical Debt, in: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, s.l., 2012-08, pp. 91–100

Parnas, *David Lorge* (Software Aging, 1994): Software Aging, in: Proceedings of 16th International Conference on Software Engineering, s.l., 1994, pp. 279–287

Potdar, *Aniket*, *Shihab*, *Emad* (Self-Admitted Debt, 2014): An Exploratory Study on Self-Admitted Technical Debt, in: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME '14, s.l.: IEEE Computer Society, 2014-09, pp. 91–100

Ramač, *Robert*, *Mandić*, *Vladimir*, *Taušan*, *Nebojša*, *Rios*, *Nicolli*, *Freire*, *Sávio*, *Pérez*, *Boris*, *Castellanos*, *Camilo*, *Correal*, *Darío*, *Pacheco*, *Alexia*, *Lopez*, *Gustavo*, *Izurieta*, *Clemente*, *Seaman*, *Carolyn*, *Spinola*, *Rodrigo* (Prevalence, Common Causes and Effects of Technical Debt, 2021): Prevalence, Common Causes and Effects of Technical Debt: Results from a Family of Surveys with the IT Industry, in: Journal of Systems and Software, 184 (2021), p. 111114

Schwaber, *Ken* (SCRUM Process, 1997): SCRUM Development Process, in: *Sutherland*, *Jeff*, *Casanave*, *Cory*, *Miller*, *Joaquin*, *Patel*, *Philip*, *Hollowell*, *Glenn* (eds.), Business Object Design and Implementation, s.l.: Springer London, 1997, pp. 117–134

Shyu, *Heidi* (Military Software Needs, 2017): A Perspective on Military Software Needs, Presented at the Carnegie Mellon University Software Engineering Institute (SEI) Symposium, s.l., 2017-03

Silva, *Ruben Blenicio Tavares*, *Bezerra*, *Carla* (CI Bad Practices, 2022): Empirical Investigation of the Influence of Continuous Integration Bad Practices on Software Quality, in: Workshop de Visualização, Evolução e Manutenção de Software (VEM), s.l.: SBC, 2022-10, pp. 51–55

Tempero, *Ewan*, *Gorschek*, *Tony*, *Angelis*, *Lefteris* (Barries to Refactoring, 2017): Barriers to Refactoring, in: Commun. ACM, 60 (2017), Nr. 10, pp. 54–61

Thomas, *Jan*, *Dragomir*, *Ana Maria*, *Lichter*, *Horst* (Architecture Conformance Checking, 2017): Static and Dynamic Architecture Conformance Checking: A Systematic, Case Study-Based Analysis on Tradeoffs and Synergies, in: s.l., 2017-12, p. 13

Uzunova, *Nadica*, *Pavlič*, *Luka*, *Beranič*, *Tina* (Quality Gates, 2024): Quality Gates in Software Development: Concepts, Definition and Tools, in (2024)

Internet sources

Fowler, *Martin* (2006): Continuous Integration, (2006) [Access: 2025-03-18]
Fowler, *Martin* (2009): Technical Debt Quadrant, (2009-10) [Access: 2025-03-10]

SonarQube (2025): Understanding Measures and Metrics | SonarQube Server Documentation, (2025) [Access: 2025-03-19]

Declaration of authorship

I declare that this paper and the work presented in it are my own and has been generated by me as the result of my own original research without help of third parties. All sources and aids including AI-generated content are clearly cited and included in the list of references. No additional material other than that specified in the list has been used.

I confirm that no part of this work in this or any other version has been submitted for an examination, a degree or any other qualification at this University or any other institution, unless otherwise indicated by specific provisions in the module description.

I am aware that failure to comply with this declaration constitutes an attempt to deceive and will result in a failing grade. In serious cases, offenders may also face expulsion as well as a fine up to EUR 50.000 according to the framework examination regulations. Moreover, all attempts at deception may be prosecuted in accordance with § 156 of the German Criminal Code (StGB).

I consent to the upload of this paper to thirdparty servers for the purpose of plagiarism assessment. Plagiarism assessment does not entail any kind of public access to the submitted work.

Hamburg, 26.3.2025

(Location, Date)

(handwritten signature)