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:	April 2, 2025

Contents

List of Figures

List of Tables

List of Abbreviations

BSI	Federal Office for Information Security
CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Delivery
CD	Continuous Delivery
CPM	Customer Product Management
DOD	Department of Defense
EVB-IT	Ergänzende Vertragsbedingung für die Beschaffung von IT-Leistungen
NRC	National Research Council
QA	Quality Assurance
SASPF	Integrated Standard-Application-Software-Product Family
TDD	Test Driven Development
VS-NFD	Verschlusssache - Nur für den Dienstgebrauch
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.

2.1.1 Software Decay

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].

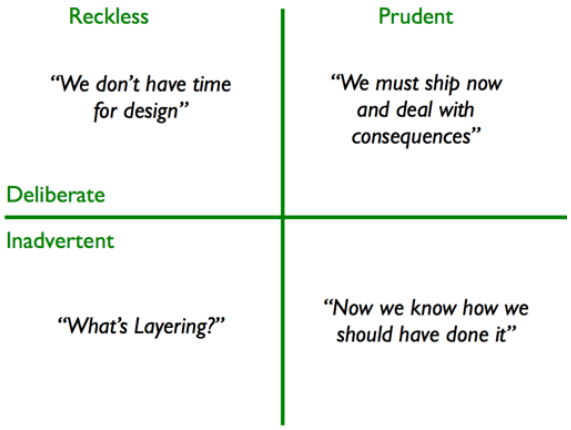
2.1.2 Technical Debt

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.

2.1.3 Conclusion

This chapter has outlined the foundational challenges inherent to software engineering and introduced two critical concepts: software decay and technical debt. Drawing on Frederick Brooks' distinction between essential and accidental difficulties, it becomes evident that complexity, changeability and lack of uniformity are deeply rooted in the nature of software itself and cannot be fully resolved. These enduring challenges are the foundation of the concepts of software decay and technical debt.

Software Decay refers to the gradual degradation of a software system's internal quality, leading to reduced maintainability, increased complexity and higher change effort. This is a

[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.

natural outcome of continuous system evolution in a response to changing requirements and environments. As shown through empirical work of Eick et al. and the theoretical contributions of Lehman and Parnas, software that is not actively maintained becomes harder to main, less modular and more fragile, making decay a systematic and long-term threat to software sustainability.

On the other hand, technical debt captures the intentional or unintentional compromises made during software development that create short-term advantages but lead to long-term costs. Originating as a metaphor, it has since evolved into a structured framework for describing, architectural design, code or process-related liabilities. While software decay can be seen as a developing property of ongoing change, technical debt often originates from conscious decisions. Whether this is due to deadlines, insufficient knowledge or lack of resources, it manifests through debt like code quality deterioration, architectural erosion or insufficient testing.

Importantly, technical debt and software decay are closely related: unmanaged technical debt accelerates software decay, while software decay can lead to the accumulation of technical debt. Empirical studies from Potdar, Li and Banker confirm the measurable impact of these phenomenon on maintenance costs, defect rates and overall system quality.

To effectively manage these issues, the software industry has developed strategies that go beyond reactive maintenance. Structured, proactive practices such as continuous refactoring, rigorous quality assurance, technical debt management and automated testing are essential to prevent the accumulation of debt and decay. Based on these foundations, the next chapter will explore concrete mitigation strategies in commercial environments providing a comprehensive overview of the most common strategies, techniques and frameworks used in the industry to address technical debt and software decay.

2.2 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 resulting in a more expensive and less competitive product. With these challenges in mind, practitioners and researchers

have developed a variety of strategies which 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.

2.2.1 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 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 software quality and mitigation of software decay.
Similarly, CI/CD introduces rigorous automation, rapid feedback loops and early detection of defects to proactively control 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.

2.2.1.1 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 broader 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'

[47] *Leite*, *G. d. S.* et al., Technical Debt Management in Agile Software Development, 2024.
[48] Ibid., p. 318.

(1997)[49]. It has since become one 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 the most used artifacts for identifying technical debt in Scrum are the Sprint and Product Backlog.[50] By explicitly managing these items with their workflow, teams effectively reduce both debt and software entropy, improving overall software maintainability.

XP was first introduced by Kent Beck in his influential book 'Extreme Programming Explained' (1999)[51]. It explicitly integrates practices to enhance software quality and directly prevent software decay. XP practices such as pair programming, Test Driven Development (TDD), 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 securing higher-quality code through collaborative review and knowledge sharing between developers. CI provides regular, frequent code integration, significantly reducing integration complexity and associated decay risks. TDD establishes robust test coverage, catching defects early and preventing quality erosion. The effectiveness of refactoring, a cornerstone XP practice, has been proven empirically. For example, in their case study (2008) Moser et al. demonstrated that refactoring explicitly 'prevents an explosion of complexity'[52] and promotes simpler, easier-to-maintain designs. They found it drove developers toward simpler designs, reducing complexity, coupling and long-term maintenance issues thereby directly counteracting software decay.[53] 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.

On the whole, 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.

[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, p. 262.
[53] Ibid., p. 262.

2.2.1.2 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'[54]. 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.'[55]. Furthermore, another critical prerequisite is 'that they can correctly build the code,'[56] thus guaranteeing that code changes consistently integrate without issues.

To further prevent technical debt and broader software decay, quality analysis tools such as static code analyzers like 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.[57] In their paper 'Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study' (2021)[58] 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.[59] However during their research they noticed if they changed SonarQubes default rules by just 26 rules, the technical debt effort would increase from 1 hours and 50 minutes to 11 hours.[60] This issue, together with the fact that SonarQube can only detect code related debt and not other debt such as 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) 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'[61]. 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,

[54] *Fowler*, *M.*, 2006.
[55] Ibid.
[56] Ibid.
[57] *SonarQube*, 2025.
[58] *Murillo*, *M. I.*, *Jenkins*, *M.*, Technical Debt Measurement Sonarqube, 2021.
[59] Ibid., p. 5.
[60] Ibid., p. 4.
[61] *Humble*, *J.*, *Farley*, *D.*, Continuous Delivery, 2010, p. 17.

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’[62].

2.2.2 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 practices have been previously introduced: continuous refactoring and maintaining software quality.
The benefits of refactoring have been demonstrated in the previous section. However, deeper empirical research illustrates the impacts and challenges in a commercial environment. Kim et al. (2014) observe in their paper ‘An empirical study of refactoring challenges and benefits at Microsoft’ that ‘[d]evelopers perceive that refactoring involves substantial cost and risks’[63]. Additionally they describe the benefits refactoring brings as ‘multidimensional’[64]. Furthermore it is not consistent across different metrics, leading to their recommendation of a tool to monitor the impact of refactoring across these metrics[65].
Similarly, Tempero et al. (2017) established 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 classes, even though they thought it would be beneficial.[66]
Tempero et al. claims the reasons were ‘lack of resources, of information identifying consequences, of certainty regarding risk and of support from management’[67] even though developers did not state a lack of refactoring tools as a reason. To eliminate these barriers, Tempore et al. suggest refactoring should be goal-oriented instead of operations-oriented and a better quantification of the benefits refactoring should bring to better reach a decision.[68]
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 are aware of 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

[62] *Humble*, *J.*, *Farley*, *D.*, Continuous Delivery, 2010, p. 19.
[63] *Kim*, *M.*, *Zimmermann*, *T.*, *Nagappan*, *N.*, Refactoring Challenges, 2014, p. 17.
[64] Ibid., p. 17.
[65] Ibid., p. 17.
[66] *Tempero*, *E.*, *Gorschek*, *T.*, *Angelis*, *L.*, Barries to Refactoring, 2017, p. 60.
[67] Ibid., p. 60.
[68] Ibid., p. 61.

that writing tests before the implementation encourages cleaner code and gives developers the confidence to tackle complex problems.[69] This practice promotes modularity, testability and clean design, 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, although its effect on overall software quality was more limited.[70] Importantly, they note that TDD significantly improves test coverage, a key factor in preventing unintentional decay. These gains, however, were accompanied by higher development effort.[71]

This is consistent with the case study by Bhat and Nagappan (2006), who reported a 15-35% increase in development time when using TDD.[72] 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.'[73] Similarly, Bhadauria et al. (2020) confirm 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.[74]

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.

2.2.3 Architectural Strategies for Long-Term Quality

In addition to code-level practices, the maintenance of 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.[75] One common strategy to mitigate these risks,

[69] *Beck*, *K.*, Test-Driven Development, 2002, pp. 8-9.
[70] *Mäkinen*, *S.*, *Münch*, *J.*, Effects of TDD, 2014, p. 13.
[71] Ibid., p. 13.
[72] *Bhat*, *T.*, *Nagappan*, *N.*, Evaluating the Efficacy of Test-Driven Development, 2006, p. 361.
[73] Ibid., p. 361.
[74] *Bhadauria*, *V.*, *Mahapatra*, *R.*, *Nerur*, *S.*, Performance Outcomes of Test-Driven Development, 2020, p. 1058.
[75] *De Silva*, *L. R.*, *Balasubramaniam*, *D.*, Controlling Software Architecture Erosion, 2012, p. 1.

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.[76]

Beyond mere conformance, teams must also 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.[77] A key component in this proactive approach is managing the architectural technical debt. These structural design decisions 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.[78]

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.[79]

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 issue discussed thoroughly in the subsequent section on Tool Support 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.

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

2.2.4 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'[80]. 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[81] and the 'single most cited effect of TD is delivery delay.'[82] This indicates that time pressure is not only a cause of 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 the principle of continuous improvement, 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.[83] He found that 'inspections increase productivity and improve final program quality.'[84] 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.[85] They found that coverage, while important, is not the only factor that influences the post-release defects.[86] They state that 'review participation should be considered when making integration decisions.'[87] Additionally, they recommend that if an

[80] *Ramač*, *R.* et al., Prevalence, Common Causes and Effects of Technical Debt, 2021, p. 40.
[81] Ibid., p. 40.
[82] Ibid., p. 40.
[83] *Fagan*, *M. E.*, Code Inspections, 1976, p. 183.
[84] Ibid., p. 205.
[85] *Mcintosh*, *S.* et al., Impact Code Review, 2016, p. 6.
[86] Ibid., p. 39.
[87] Ibid., p. 39.

expert in the matter is not available for the original code, they should be included in the review process to prevent defects.[88]

In conclusion, structured approach to managing technical debt and preventing software decay is crucial to maintain 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.

2.2.5 Tool Support for Continuous Quality Assurance

To further support the proactive management of 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 which 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 and rectify them before they are introduced into the code base . While the concept of quality gates is not new, the automation of these gates have recently 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.[89] To allow for this kind of automation, they suggest tools like SonarQube, Sigrid or Maverix.ai. These tools provide real-time feedback allowing for an enforcement of quality criteria.[90] Utilizing this approach enables 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.[91] In their case study 'Evaluating an Architecture Conformance Monitoring Solution' (2016), Caracciolo et al. investigated three different tools to detect

[88] *Mcintosh*, *S.* et al., Impact Code Review, 2016, p. 39.
[89] *Uzunova*, *N.*, *Pavlič*, *L.*, *Beranič*, *T.*, Quality Gates, 2024, p. 8.
[90] Ibid., p. 8.
[91] *Thomas*, *J.*, *Dragomir*, *A. M.*, *Lichter*, *H.*, Architecture Conformance Checking, 2017, p. 6.

architecture violations: SonarQube, Sonargraph and TeamCity[92]. They concluded that their approach 'can be applied in an industrial context.'[93] They furthermore argue that the tools can be conveniently added to a dashboard, allowing for a short feedback loop as well as proactive management of architectural violations.[94] 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[95].

Numerous empirical studies show the effectiveness of these tools. 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.'[96] However they also observed the impact of bad practices in the implementation of CI. They found that the quality levels were incorrectly set and the standard configuration tools were used instead of adjusting them to the project needs, which overall harmed the software quality indicators.[97] This indicates that, for the tools to be effective, they need to be correctly configured and adjusted to the project needs.

2.2.6 Synthesis of Commercial Mitigation Strategies

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

Organizational measures such as technical debt tracking, dedicated maintenance cycles and structured code reviews provide essential process-level support, enabling teams to

[92] *Caracciolo*, *A.* et al., Evaluating Architecture Monitoring, 2016, p. 43.
[93] Ibid., p. 44.
[94] Ibid., p. 44.
[95] Ibid., p. 43.
[96] *Silva*, *R. B. T.*, *Bezerra*, *C.*, CI Bad Practices, 2022, p. 4.
[97] Ibid., p. 4.

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.

2.3 Unique Constraints in Military Software Development

2.3.1 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 differences between commercial and military software development.

According to the NRC there are three main distinctions. The first regards the 'sheer size and complexity of defense systems'[98]. These systems often have a vast amount of individual elements which need to work together, and also evolve over time as the architecture between the different stages of the lifecycle changes.[99] 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.[100]

The second distinction is the different perspectives of the actors involved. Unlike in the commercial environment, where it is usually only the project manager who controls the vision and the goals of the project, in the military environment there are multiple stakeholders with different goals in mind.[101] Although the book specially focuses on the American military, the same can be said for the German military.

The third difference given by the NRC are the concerns 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 most of the risk, as they will be using the product in the end. The NRC claims that this means 'the system developers do not have a strong incentive to address reliability goals early in the

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

acquisition process'[102]. especially because the downstream benefits are not quantifiable for the developers.[103]
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, who must interoperate with each other in real time.[104] These projects often have a lifecycle of decades compared to the commercial software lifecycle of a few years.[105] Due to these long lifecycles, the software needs to be developed so it can be easily maintained and updated, while still working with legacy systems which often have very specific requirements.[106] Whereas the the architecture has to last for decades, soft- and hardware changes much faster, often resulting in a patchwork of quick fixes.[107] Heidi Shyu provides examples of necessary software solutions such as a flexible architecture which allows for easy updates and additions, self-testing modular code and intuitive, easy-to-use interfaces.[108]

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 all serve to create a complex environment that requires specialized strategies to manage software decay and technical debt effectively.

2.3.2 Regulatory and Procurement Constraints in the Bundeswehr

The procurement and regulatory environment significantly shapes military software development in the German Armed Forces (Bundeswehr). Unlike in commercial settings, Bundeswehr software projects are tightly governed by formalized frameworks, rigorous 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. It is a customized variant of the standard V-Modell XT, specifically tailored to Bundeswehr

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

needs and closely integrated with the Bundeswehr's overarching procurement framework, Customer Product Management (CPM). The V-Modell XT Bw is a comprehensive framework that encompasses the planning and execution of software projects in the Bundeswehr.[109] It includes detailed guidelines for all parts of the software development process, from roles and requirements engineering to Quality Assurance (QA) and product quality.[110] 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 commission in 2010 found that: 'Die Streitkräfte erhalten ihre geforderte Ausrüstung zumeist weder im erforderlichen Zeit- noch im geplanten Kostenrahmen.'[111]. 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 contract for government IT solutions, hinders agile development. In his article from 2022, Holger Schröder claims that multiple different contract blueprints would have to be combined to allow for an agile development project.[112]

Collectively, all these requirements and procurement constraints strongly influence software development practices. Control and predictability are prioritized, making an iterative and agile framework difficult to use. While these procedures promote accountability and reduce risk, they can severely limit a system's ability to evolve over time, thereby fostering conditions under which software decay and technical debt accumulate. Adding to this complexity are strong security and compliance requirements, which will be discussed in the following subsection.

2.3.3 Security and Compliance Requirements

Military software is obliged to 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. These strict requirements result in severe difficulties regarding the lifecycle of the product, the software architecture

[109] *Bundeswehr*, V-Modell Bw, 2013, p. 6.
[110] Ibid., pp. 20-21.
[111] *Strukturkommission der Bundeswehr*, Strukturkommissionsbericht Bundeswehr, 2010, p. 36.
[112] *Schröder, H.*, Agile Beschaffungsprojekte, 2022.

and maintenance practices. The resulting constraints accelerate the accumulation of technical debt and software decay.

Like all federal agencies in Germany, the Bundeswehr must adhere to the Federal Office for Information Security (BSI) guidelines for IT security, called IT-Grundschutz[113]. This defines a comprehensive security framework that includes technical, personnel and infrastructural security measures through the use of several standards. Compliance with these standards involves extensive documentation, risk management processes and regular audits. This leads to an increase in the complexity and effort required for ongoing software maintenance and updates.[114]

In addition to the national regulations, Bundeswehr software systems are obligated to comply with the NATO security standards. These guidelines establish requirements for the handling of classified data, encryption standards and security clearance of personnel. This is illustrated by the NATO Security Policy (C-M(2002)49) which dictates the handling of NATO-classified information, constraining software flexibility and maintenance practices[115]. Compliance with NATO standards mandates extensive security accreditation procedures, detailed documentation and comprehensive vetting of software and hardware systems. These processes slow down software development cycles and restrict flexibility due to the necessity of re-certifying software after significant changes.

Furthermore the Bundeswehr adheres to national classification rules, such as Verschlusssache - Nur für den Dienstgebrauch (VS-NFD). Systems classified under VS-NFD require access control, encrypted communications, dedicated network infrastructure and periodic security accreditation.[116] Developers and system administrators need to adhere to strict usage guidelines, which dictate the permissible software, hardware and methods of data handling.[117] Compliance significantly restricts the usage of modern technologies such as cloud services or external software libraries, unless explicitly cleared[118]. This can be seen in the example of Genua's secure remote working solution, which complies with both VS-NfD and NATO security requirements while offering functionality typically restricted in classified environments.[119]

[113] *Bundesamt für Sicherheit in der Informationstechnik*, Tasks of BSI, n. d.
[114] *Bundesamt für Sicherheit in der Informationstechnik*, BSI Standards, n. d.
[115] *North Atlantic Council*, 2002, Enclosure B, pp. 1–3, Enclosure F, pp. 1–2.
[116] *Bundesministerium für Wirtschaft und Klimaschutz (BMWK)*, VS-NFD, 2023, see Part 2, pp. 1–3, Part 3, pp. 1–6.
[117] Ibid., Part 3, No. 3.1–3.7, pp. 1–5.
[118] Ibid., Part 3, No. 3.4.1–3.4.5, pp. 4–5.
[119] *genua GmbH*, Genua VS-NFD, 2024.

Although essential for operational security, these security and compliance requirements markedly constrain software adaptability and agility. Extensive documentation, mandatory security certifications and regular audits mandated by national and international regulations considerably increase the maintenance workload and complexity. Consequently, prolonged approval processes and heightened architectural complexity often accelerate software decay and technical debt accumulation, complicating extensive refactoring or proactive architectural evolution.

In summary, security and compliance requirements play a pivotal role in shaping Bundeswehr software development practices, significantly influencing architecture design, maintenance methodologies and lifecycle management. While crucial for safeguarding operational security and classified information, these rigorous standards often inhibit software adaptability and innovation, exacerbating the challenges of software decay and technical debt inherent in military software systems.

2.3.4 Legacy Systems and Extended Lifecycles

Legacy systems and extended lifecycles impose particularly significant constraints for military software projects, especially within the Bundeswehr. Unlike commercial software, which typically operates for a limited period, military systems frequently remain operational for decades due to high investment costs, complexity and risk aversion to adopting new technologies. Such prolonged operational lifespans inevitably contribute to technical debt accumulation and progressive software decay, significantly complicating maintenance and increasing lifecycle costs.

A prominent example of these challenges emerged within the Bundeswehr's logistics and support IT systems at the turn of the century. According to Sebastian Stein's 2011 blog entry, the Bundeswehr managed around 1,200 distinct outsourced IT systems, creating a highly fragmented and inefficient infrastructure.[120] To address this issue, the Bundeswehr initiated the Integrated Standard-Application-Software-Product Family (SASPF) program, consolidating procurement, accounting, and logistics systems into a single, integrated SAP-based solution. However, standard SAP software met initially only around 70% of the Bundeswehr's unique operational requirements, necessitating extensive custom development and temporary workarounds.[121] These integrations were further complicated by stringent regulatory requirements, diverse stakeholder interests and interoperability challenges with outdated technologies. Consequently, significant technical debt and

[120] *Stein, S.*, BPM in the Bundeswehr, 2011.
[121] Ibid.

complexity accumulated, extending the modernization effort well over a decade and clearly demonstrating the severe maintenance implications of legacy systems.[122]

Another illustrative case within the Bundeswehr is the Luftwaffe's Tornado fighter jet program. Development began in 1981 and the aircraft entered operational service in 1992. Today, approximately 80 Tornado jets remain operational, with plans for replacement only by 2030 resulting in a lifecycle of nearly 50 years.[123] Continuous software updates have been crucial throughout this extensive period, including notable upgrades like the integration of the RecceLite reconnaissance system in 2009.[124] However, each update has progressively become more complex and costly due to aging hardware, software architecture limitations and stringent compliance demands. This incremental layering of software modifications directly contributes to accelerated software decay, increasing the difficulty and expense of maintaining operational effectiveness.

Comparable challenges are also prevalent among international NATO allies. A notable example was highlighted by Kynan Carver (2022), describing how the U.S. Department of Defense (DOD) continued operating critical strategic systems reliant on severely outdated technologies, including 8-inch floppy disks, until the late 2010s.[125] Carver emphasizes that maintaining these aging software infrastructures consumes significant financial and operational resources, limiting investment in innovation and proactive modernization strategies.[126]

In conclusion, effectively managing technical debt and mitigating software decay within military systems requires strategic, long-term investments, proactive modernization plans and an explicit focus on maintainability. Addressing the unique challenges posed by legacy systems and extended lifecycles is thus crucial for sustaining operational capability and flexibility within an increasingly dynamic military technology environment.

2.3.5 Conclusion

Military software development is fundamentally shaped by demands that are far beyond those of the commercial environments. The need for extremely reliable, long lasting systems which still meet strict classification regimes and multi-layered stakeholder requirements creates a highly regulated and risk-shy context. These characteristics collectively contribute to a development landscape where software decay and technical

[122] *Stein*, *S.*, BPM in the Bundeswehr, 2011.
[123] *Skiba*, *T.*, Der Tornado, 2024.
[124] bundeswehrTornado
[125] *Carver*, *K.*, Technical Debt, 2022.
[126] Ibid.

debt are not merely byproducts of poor practices but structural risks embedded within the system's lifecycle.

As shown throughout this section, four constraint areas amplify these challenges. Firstly, the inherit complexity of military systems, often designed to operate for decades, creates pressure to extend the viability of aging platforms. Secondly, the rigid procurement and development frameworks, such as the V-Modell XT Bw and the CPM process, prioritize upfront control and traceability over agility, hindering adaptive or incremental development practices. Thirdly, extensive national and NATO security requirements impose detailed compliance burdens, restricting the use of modern tools and leading to prolonged certification cycles. Finally, the heavy reliance on legacy systems and technologies, as illustrated by the Tornado jet or SASPF, reinforces technical inertia and raises the cost and difficulty of modernization efforts.

Together, these factors present a software ecosystem where continuous improvement, architectural evolution and proactive quality management are severely constrained. Consequently, technical debt tends to accumulate more rapidly and opportunities to refactor or modernize the system are limited due to procedural, regulatory and operational constraints.

Therefore it is crucial to understand how military software practitioners operate within this setting. The next section of this thesis will examine how experts working on Bundeswehr-related software systems experience and address these limitations in practice. The main focus will be their strategies in managing software decay, handling long-lived legacy systems and the impact of the regulatory environment on their work.

2.4 Summary of the Literature Review

Software development faces inherent challenges that have existed for decades. The previous chapters examined these problems from three key perspectives: the foundational problems of software engineering, the mitigation strategies used in commercial environments and the unique constraints faced in military projects. This summary will highlight the key insights to establish a foundation regarding software decay and technical debt, thereby laying the groundwork for the empirical investigation in the following chapters.

2.4.1 Problems of Software Engineering

The complexity of software systems, combined with their extended lifecycle and constant evolution, creates significant challenges. As presented by Brooks (1986), essential difficulties such as complexity, conformity, changeability and invisibility are inherent in software development. In contrast are the accidental difficulties which stem from the tooling and process problems of the current software development practices. These foundational matters create the two following common issues.

The first is software decay which refers to the gradual deterioration of software quality, due to evolving requirements and the constant need for updates and maintenance. This has been empirically validated by Eick et al and theoretically supported by Lehman, Belady and Parnas.
The second issue is Technical Debt, a metaphor introduced by Cunningham. Technical Debt represents the cost of implementation or design shortcuts taken to meet immediate needs, which can lead to long-term maintenance challenges. This concept has been further developed by McConnel, Fowler and Kruchten, highlighting the risks and management needs associated with both intentional and unintentional technical debt. Together, these two concepts create a complex environment where long-term degradation is inevitable unless proactively managed.

2.4.2 Mitigation Strategies in Commercial Environments

To combat software decay and manage technical debt, commercial software projects have developed a wide range of strategies. These include high-level practices such as agile methodologies like Scrum, XP or CI/CD pipelines which embed quality into the daily workflow. Such practices are complemented by code level-techniques like continuous refactoring, TDD or code reviews which help manage the complexity and ensure maintainability. Architectural strategies like conformance checking and managing architectural technical debt support long-term structural integrity. This is further supported by organizational practices such as backlog tracking of debt, balancing time pressure and cultivating a quality-focused culture. Finally, the use of modern tooling through automated analysis tools (e.g. SonarQube, TeamCity) allows enforcement of standards and early detection of issues.

These approaches aim to proactively manage quality and control the complexity, thereby reducing the long-term cost of maintenance.

2.4.3 Constraints in the Military Software Environments

While the challenges of decay and technical debt are universal, military software projects face the following unique constraints that complicate their management.

The project structure in the Bundeswehr is defined by the V-Modell XT Bw and the CPM processes, which emphasize documentation, predictability and rigid lifecycle phases, making agile practices difficult to implement. The procurement contracts like the EVB-IT further limit flexibility and iterative development. The security and compliance requirements from the BSI, NATO policies and VS-NfD create additional burdens through mandatory certifications, audits and limitations on tooling or external software. The extended lifecycles and legacy systems that exist in every military environment create decade long systems that need service and have to remain operational while also adapting to new requirements.

These conditions result in higher maintenance burdens and slower update cycles and also reduce the ability to refactor or modernize systems. This creates a software ecosystem where decay and technical debt accumulation are accelerated.

2.4.4 Conclusion

In conclusion, software decay and technical debt are persistent and measurable risks in all software systems. While commercial environments have developed effective mitigation strategies, their applicability in military projects is limited by structural, regulatory and operational constraints.

Understanding these differences is essential for designing effective strategies to manage decay and technical debt in military software projects. The next chapter will explore how professionals in military related software projects deal with these constraints in practice, offering real-world insights through expert interviews. They will provide a deeper understanding on how decay and debt are managed, despite the rigid frameworks and complex environments.

Appendix

Interviewfragebogen

Einleitung:

Vielen Dank für Ihre Teilnahme. Ziel dieser Studie ist es, zu verstehen, wie in Ihrem Unternehmen aktuell Strategien zur Vermeidung und Minderung technischer Schuld und Softwareverfall (Software Decay) genutzt werden und wie effektiv diese Maßnahmen sind.

1. Allgemeine Informationen

1. Welche Rolle haben Sie aktuell inne und wie lange arbeiten Sie bereits im Bereich der Softwareentwicklung militärischer Projekte?
2. Mit welchen Arten von Softwaresystemen arbeiten Sie aktuell hauptsächlich?

2. Agile Methoden und technischer Schuld

3. Werden agile Methoden (z.B. Scrum, XP) in Ihren Projekten eingesetzt? Wenn ja, wie genau?
4. Wie bewerten Sie die Effektivität agiler Methoden zur Prävention und Minderung technischer Schuld und Softwareverfall? Erläutern Sie bitte Ihre Erfahrungen.
5. Falls agile Methoden nicht möglich sind, wie geht Ihr Team dann mit technischer Schuld um? Welche konkreten Praktiken nutzen Sie?

3. CI/CD-Praktiken

6. Nutzen Sie aktuell CI/CD? Falls ja, beschreiben Sie kurz Ihre CI/CD-Implementierung (z.B. Tools, Abläufe).
7. Wie wirksam schätzen Sie CI/CD zur Reduktion technischer Schuld oder Softwareverfall ein?
8. Welche Faktoren erschweren aktuell die Einführung oder optimale Nutzung von CI/CD in Ihren Projekten?

4. Code Reviews

9. Werden regelmäßig Code Reviews durchgeführt? Falls ja, wie genau (z.B. Peer Review, automatisierte Tools)?
10. Wie effektiv sind Code Reviews aus Ihrer Sicht, um technische Schuld oder Softwareverfall zu reduzieren? Nennen Sie gerne konkrete Erfahrungen.
11. Welche Faktoren könnten die Effektivität von Code Reviews weiter verbessern?

5. Architekturmanagement und Konformität

12. Wie wird in Ihren Projekten die Softwarearchitektur aktuell verwaltet und dokumentiert?
13. Wie stellen Sie sicher, dass die Implementierung der geplanten Architektur entspricht? Nutzen Sie Tools oder automatisierte Checks?
14. Was sind die größten Herausforderungen im Bereich Architekturkonformität, speziell bei langlaufenden oder Legacy-Projekten?

6. Umgang mit Legacy-Systemen und langen Lebenszyklen

15. Welche besonderen Herausforderungen bestehen bei der Wartung und dem Management technischer Schuld in Legacy-Systemen oder Projekten mit besonders langen Laufzeiten?
16. Welche Strategien oder Maßnahmen haben sich in Ihrem Team konkret bewährt, um Softwareverfall und technische Schuld bei Legacy-Systemen zu reduzieren?

7. Bewertung und Effektivität der aktuellen Mitigationsstrategien

17. Welche konkreten Strategien oder Maßnahmen setzen Sie aktuell hauptsächlich ein, um technische Schuld und Softwareverfall aktiv zu verhindern oder zu verringern?
18. Wie bewerten Sie insgesamt die Effektivität dieser Strategien?
19. Welche Hindernisse oder Herausforderungen erschweren aktuell die erfolgreiche Umsetzung Ihrer Mitigationsstrategien?

20. Gibt es Arten technischer Schuld, die besonders schwierig zu handhaben sind? Falls ja, warum?
21. Können Sie ein konkretes Beispiel für eine erfolgreiche Mitigationsmaßnahme nennen? Was genau machte diese Maßnahme erfolgreich?
22. Welchen Einfluss haben externe Faktoren (z.B. regulatorische Vorgaben, Sicherheitsanforderungen, Beschaffungsprozesse) auf Ihre Fähigkeit, technische Schuld effektiv zu managen?

Abschluss

23. Welche Maßnahme wäre Ihrer Erfahrung nach besonders wirksam, um technische Schuld und Softwareverfall in Ihrer Organisation zukünftig effektiver zu managen?
24. Gibt es weitere relevante Erfahrungen oder Einsichten, die Sie mitteilen möchten?

Vielen Dank für Ihre Teilnahme!

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

Bundesministerium für Wirtschaft und Klimaschutz (BMWK) (VS-NFD, 2023): Handreichung Zum Geheimschutz in Der Wirtschaft: VS-NfD (Verschlusssache – Nur Für Den Dienstgebrauch), s.l., 2023

Bundeswehr (V-Modell Bw, 2013): V-Modell XT Bw, s.l., 2013

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 Refactoring Challenges 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 Software Engineering, 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 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 Architectural 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

North Atlantic Council (2002): Security within the North Atlantic Treaty Organisation (NATO), tech. rep. C-M(2002)49, s.l.: North Atlantic Treaty Organization, 2002

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)

Schröder, *Holger* (Agile Beschaffungsprojekte, 2022): Ungeeignet für agile IT-Beschaffungsprojekte, in: Staatsanzeiger Baden-Württemberg (2022), p. 32

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

Strukturkommission der Bundeswehr (Strukturkommissionsbericht Bundeswehr, 2010): Bericht der Strukturkommission der Bundeswehr - Vom Einsatz her denken - Konzentration, Flexibilität, Effizienz, tech. rep., s.l.: Bundesministerium der Verteidigung, 2010-10

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

Bundesamt für Sicherheit in der Informationstechnik (BSI Standards, n. d.): BSI-Standards, (keine Datumsangabe) [Access: 2025-03-27]
Bundesamt für Sicherheit in der Informationstechnik (Tasks of BSI, n. d.): BSIFAQ, (keine Datumsangabe) [Access: 2025-03-27]

Carver, *Kynan* (Technical Debt, 2022): Technical Debt: The Cybersecurity Threat Hiding in Plain Sight, (2022-12) [Access: 2025-03-31]

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

genua GmbH (Genua VS-NFD, 2024): Genusecure Suite: The Comprehensive Security Solution for Classification Level German VS-NfD Compliant Workplaces, (2024-08)

Skiba, *Thomas* (Der Tornado, 2024): Der Tornado: 50 Jahre Spitzenleistung in den Diensten der Luftwaffe, (2024-04) [Access: 2025-03-31]
SonarQube (2025): Understanding Measures and Metrics | SonarQube Server Documentation, (2025) [Access: 2025-03-19]
Stein, *Sebastian* (BPM in the Bundeswehr, 2011): Business Process Management in Big Organizations - Bundeswehr | ARIS BPM Community, (2011-05) [Access: 2025-03-31]

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, 2.4.2025

(Location, Date)

(handwritten signature)