



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: May 16, 2025

Contents

List of Figures	V
List of Tables	VI
List of Abbreviations	VII
List of Symbols	VIII
1 Introduction	1
2 Literature Review	2
2.1 Problems of Software Engineering	2
2.1.1 Software Decay	4
2.1.2 Technical Debt	5
2.1.3 Conclusion	8
2.2 Mitigation Strategies in Commercial Environments	10
2.2.1 High-Level Mitigation Strategies	10
2.2.1.1 Agile Methodologies	10
2.2.1.2 Continuous Integration/Continuous Deployment	12
2.2.2 Code-Level Practices for Preventing Decay	13
2.2.3 Architectural Strategies for Long-Term Quality	15
2.2.4 Process and Organizational Measures	16
2.2.5 Tool Support for Continuous Quality Assurance	17
2.2.6 Synthesis of Commercial Mitigation Strategies	18
2.3 Unique Constraints in Military Software Development	19
2.3.1 Introduction to Military Software Context	19
2.3.2 Regulatory and Procurement Constraints in the Bundeswehr	20
2.3.3 Security and Compliance Requirements	22
2.3.4 Legacy Systems and Extended Lifecycles	23
2.3.5 Conclusion	25
2.4 Summary of the Literature Review	26
2.4.1 Problems of Software Engineering	26
2.4.2 Mitigation Strategies in Commercial Environments	26
2.4.3 Constraints in the Military Software Environments	27
2.4.4 Conclusion	27

3	Methodology	28
3.1	Research Design	28
3.2	Data Collection	28
3.3	Data Analysis	29
3.4	Limitations of the Methodology	30
4	Results	31
4.1	Introduction	31
4.2	Challenges in Managing Long-Lived Military Software	31
4.2.1	System Complexity	31
4.2.2	Legacy Technologies and Compatibility Issues	32
4.2.3	Organizational Challenges	33
4.3	Impact of Procurement and Approval Processes	33
4.3.1	Financial Constraints	33
4.3.2	Procurement Barriers	34
4.3.3	Slow Approval Processes	34
4.4	Technical Debt and Software Decay Practices	34
4.4.1	Tracking and Visibility of Technical Debt	35
4.4.2	Consequences of Technical Debt	35
4.4.3	Sources of Technical Debt	35
4.4.4	Handling of Existing Technical Debt	36
4.5	Mitigation Strategies in Military Software Projects	36
4.5.1	Documentation Improvements	36
4.5.2	Automation	37
4.5.3	Refactoring	37
4.5.4	Code Reviews	37
4.6	Quality Assurance and Testing	38
4.6.1	Testing Challenges	38
4.6.2	Use of Metrics and Tools	39
4.6.3	Test Coverage and Test Maintenance	39
4.6.4	Testing Practices	39
4.7	Agile Methodologies in Military Projects	40
4.7.1	Knowledge Sharing and Collaboration	40
4.7.2	Limitations in Agile Implementations	40
4.7.3	Use and Impact of Agile Methods	41
4.8	Influence of Regulatory and Security Requirements	41
4.8.1	Impact on Flexibility	41
4.8.2	Standards Compliance	42

4.8.3	Security and Reliability Requirements	42
4.9	Successful Examples	43
4.9.1	Effective Measures against Software Decay	43
4.10	Recommendations For Future Practices	44
4.10.1	Agile Expansion	44
4.10.2	Focus on Test Maintenance	44
4.10.3	Continuous Documentation	45
4.10.4	Early Investment	45
5	Discussion	46
5.1	Introduction	46
5.2	Causes and Symptoms of Software Decay in the Military Domain	46
5.3	Mitigation Strategies and Their Effectiveness	48
5.4	Recommendations for Future Practice	50
5.5	Limitations of the Study	52
6	Conclusion	53
	Bibliography	55
	Appendix	60

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
UI	User Interface
VS-NFD	Verschlusssache - Nur für den Dienstgebrauch
XP	Extreme Programming

List of Symbols

1 Introduction

Software Decay describes the gradual deterioration in software quality and maintainability, which creates significant challenges across various software domains. The challenges become even more pronounced in environments where the systems are expected to operate for decades, such as in military software context. The motivation for this thesis stems from observations of the difficulties of managing aging software projects, particularly in the military domain. Over time the systems become increasingly problematic to maintain and modify, which highlights the importance of proactively maintaining the software quality and thereby ensuring long-term viability.

The long system lifecycle is very common in military software projects, which need to be robust and dependable, while also having regulatory requirements, high security standards and complex stakeholder demands. With these unique factors in mind, usual software engineering practices to mitigate software decay could face limitations and require certain adaptations. Therefore this thesis specifically investigates how software decay can be prevented or slowed down with the specific requirements of the German military sector.

The main research question of this thesis is:

How can software decay in military systems be prevented or slowed down?

To answer this overarching question, the following sub-questions have been formulated:

- What are the main causes and symptoms of software decay specific to the military software domain?
- Which mitigation strategies are currently in use at T-Systems IFS, Chapter Defense Systems, and how effective are they?

The main goal of this thesis is to identify the specific constraints influencing software decay within military software projects and to evaluate existing practices that could effectively mitigate the decay. Through a literature review and a semi-structured expert interview conducted with three practitioners from T-Systems IFS, Chapter Defense Systems, the thesis uses both theoretical and practical insights to answer the research questions.

The thesis is structured as follows: Initially a literature review is conducted to establish the theoretical foundation. This includes general information about software decay and technical debt, as well as common mitigation strategies and unique constraints in the military domain. The second part of the thesis presents the methodology and explains the research design and analysis of the expert interviews. In the third part, the findings

of the interviews are presented in key themes derived from the data. These findings are further interpreted and discussed in the fourth part of the thesis, which integrates the theoretical insights with the practical findings. Concluding, the final chapter synthesizes the key findings of the research, highlighting the implications for practitioners, as well as limitations and future research opportunities.

The primary audiences for this thesis include industry practitioners in the military software domain, as well as stakeholders from military institutions seeking to understand and improve their long-term software management practices.

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

¹ *Brooks, F.*, No Silver Bullet, 1987, p. 3.

² *Ibid.*, p. 3.

³ *Ibid.*, p. 3.

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 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 time¹² - a phenomenon often described as software decay.

⁴ *Brooks, F.*, No Silver Bullet, 1987, p. 4.

⁵ *Ibid.*, p. 4.

⁶ *Ibid.*, p. 4.

⁷ *Ibid.*, 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.

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 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.

¹³ Eick, S. G. et al., Does Code Decay?, 2001, p. 1.

¹⁴ Ibid., 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.

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 that software inevitably reaches a state where changes become more risky, costly and error-prone²³.

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 Intentional²⁶. 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

²¹ Parnas, D. L., Software Aging, 1994, p. 280.

²² Ibid., p. 280.

²³ Ibid., pp. 280–281.

²⁴ Cunningham, W., WyCash Portfolio, 1992, p. 2.

²⁵ Ibid., p. 2.

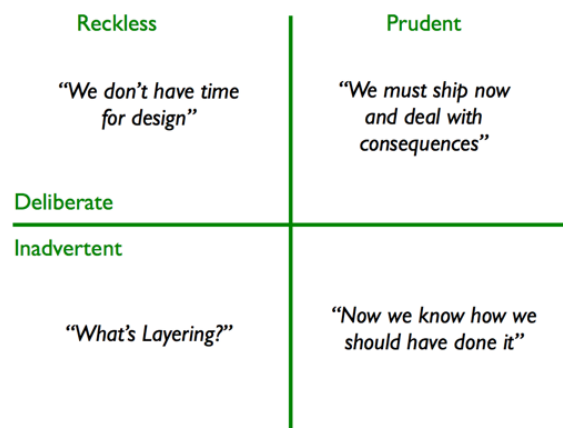
²⁶ McConnell, S., Managing Technical Debt, 2017, p. 3.

²⁷ Ibid., p. 3.

²⁸ Ibid., p. 3.

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.

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

²⁹ McConnell, S., Managing Technical Debt, 2017, p. 4.

³⁰ Fowler, M., 2009.

³¹ Ibid.

³² Kruchten, P., Nord, R. L., Ozkaya, I., Technical Debt, 2012, p. 18.

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 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.

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

³³ Kruchten, P., Nord, R. L., Ozkaya, I., Technical Debt, 2012, p. 19.

³⁴ Ibid., p. 19.

³⁵ Ibid., 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.

studies⁴¹.

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 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

⁴¹ Li, Z., Avgeriou, P., Liang, P., Mapping of Technical Debt, 2015, p. 215.

⁴² Ibid., 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.

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 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 large scale 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

As discussed in the previous section, 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. This will be viewed on different levels: high-level strategies, code-level practices, architectural level as well as process and organizational measures.

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

⁴⁷ Leite, G. d. S. et al., Technical Debt Management in Agile Software Development, 2024.

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’ (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). 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 entropy⁵¹. 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 decay⁵⁴.

⁴⁸ Leite, G. d. S. et al., Technical Debt Management in Agile Software Development, 2024, 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, pp. 5–6.

⁵² Moser, R. et al., Case Study Refactoring, 2008, p. 262.

⁵³ Ibid., p. 262.

⁵⁴ Beck, K., Extreme Programming Explained, 1999, pp. 5–7.

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.

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'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 by maintaining a specific code quality, 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.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 hours and 50 minutes to 11 hours.⁶¹ This issue, together with the fact that SonarQube can only detect code related debt if correctly configured as well as no other debt such as infrastructure or requirements debt, makes these tools useful but not a complete solution.

⁵⁵ Fowler, M., 2006.

⁵⁶ Ibid.

⁵⁷ Ibid.

⁵⁸ SonarQube, 2025.

⁵⁹ Murillo, M. I., Jenkins, M., Technical Debt Measurement Sonarqube, 2021.

⁶⁰ Ibid., p. 5.

⁶¹ Ibid., p. 4.

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'⁶². 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'⁶³.

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'⁶⁴. Additionally they describe the benefits refactoring brings as 'multidimensional'⁶⁵. Furthermore it is not consistent across different metrics, leading to their recommendation of a tool to monitor the impact of refactoring across these metrics⁶⁶. Similarly, Tempore 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.⁶⁷

Tempore 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. To eliminate these barriers, Tempore et al. suggest refactoring should be goal-oriented instead of operations-oriented

⁶² Humble, J., Farley, D., Continuous Delivery, 2010, p. 17.

⁶³ Ibid., p. 19.

⁶⁴ Kim, M., Zimmermann, T., Nagappan, N., Refactoring Challenges, 2014, p. 17.

⁶⁵ Ibid., p. 17.

⁶⁶ Ibid., p. 17.

⁶⁷ Tempore, E., Gorschek, T., Angelis, L., Barriers to Refactoring, 2017, p. 60.

⁶⁸ Ibid., p. 60.

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 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 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, 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.⁷¹ 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.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) 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.⁷⁵

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.

⁶⁹ *Tempero, E., Gorschek, T., Angelis, L., Barries to Refactoring, 2017, 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.*

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.⁷⁶ One common strategy to mitigate these risks, 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 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.⁷⁸ 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.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 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

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 Architectural Technical Debt, 2012, p. 99.

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.

2.2.4 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'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 deadlines⁸² and the 'single most cited effect of TD is delivery delay.'⁸³ 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.

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 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, accelerating and simplifying the process, 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,

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.

⁸⁵ Ibid., p. 205.

⁸⁶ McIntosh, S. et al., Impact Code Review, 2016, p. 6.

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, 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.⁹⁰ 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.⁹¹ 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

⁸⁷ *Mcintosh, S. et al., Impact Code Review, 2016, 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.*

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 TeamCity⁹³. They concluded that their approach ‘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 as well as 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 code⁹⁶. Based on these project parameters Caracciolo et al. ‘consider that to be a positive outcome.’⁹⁷

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.’⁹⁸ 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.⁹⁹ 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

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 Ibid., p. 43.

⁹⁸ Silva, R. B. T., Bezerra, C., CI Bad Practices, 2022, p. 4.

⁹⁹ Ibid., p. 4.

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 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. However, nonetheless not all the problems are able to be combated based on the essential difficulties defined by Brooks.

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'¹⁰⁰. 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.¹⁰¹ 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.¹⁰²

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.¹⁰³ Although the book specially focuses on the

¹⁰⁰ *National Research Council, Comparison Defense Commerical*, 2015, p. 31.

¹⁰¹ *Ibid.*, p. 32.

¹⁰² *Ibid.*, p. 32.

¹⁰³ *Ibid.*, p. 32.

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 acquisition process’¹⁰⁴. especially because the downstream benefits are not quantifiable for the developers.¹⁰⁵

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.¹⁰⁶ These projects often have a lifecycle of decades compared to the commercial software lifecycle of a few years.¹⁰⁷ 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.¹⁰⁸ Whereas the the architecture has to last for decades, soft- and hardware changes much faster, often resulting in a patchwork of quick fixes.¹⁰⁹ 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.¹¹⁰

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

¹⁰⁴ *National Research Council*, Comparison Defense Commerical, 2015, p. 33.

¹⁰⁵ *Ibid.*, p. 33.

¹⁰⁶ *Shyu, H.*, Military Software Needs, 2017, p. 11.

¹⁰⁷ *Ibid.*, p. 14.

¹⁰⁸ *Ibid.*, p. 14.

¹⁰⁹ *Ibid.*, p. 15.

¹¹⁰ *Ibid.*, p. 17.

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 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.¹¹¹ It includes detailed guidelines for all parts of the software development process, from roles and requirements engineering to Quality Assurance (QA) and product quality.¹¹² 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 not possible.

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.'¹¹³ 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.¹¹⁴

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.

¹¹¹ *Bundeswehr, V-Modell Bw*, 2013, p. 6.

¹¹² *Ibid.*, pp. 20-21.

¹¹³ *Strukturkommission der Bundeswehr, Strukturkommissionsbericht Bundeswehr*, 2010, p. 36.

¹¹⁴ *Schröder, H., Agile Beschaffungsprojekte*, 2022.

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 and maintenance practices. The resulting constraints accelerate the accumulation of technical debt and software decay, which will be explained in the following paragraphs.

Like all federal agencies in Germany, the Bundeswehr must adhere to the Federal Office for Information Security (BSI) guidelines for IT security, called IT-Grundschutz¹¹⁵. This defines a comprehensive security framework that includes technical, personnel and infrastructural security measures through the use of several standards. Even though these 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.¹¹⁶

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 practices¹¹⁷. 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 Verschlussache - Nur für den Dienstgebrauch (VS-NFD). Systems classified under VS-NFD require access control, encrypted communications, dedicated network infrastructure and periodic security accreditation.¹¹⁸ Developers and system administrators need to adhere to strict usage guidelines, which dictate the permissible software, hardware and methods of data handling.¹¹⁹ Compliance significantly restricts the usage of modern

¹¹⁵ Bundesamt für Sicherheit in der Informationstechnik, Tasks of BSI, n. d.

¹¹⁶ Bundesamt für Sicherheit in der Informationstechnik, BSI Standards, n. d.

¹¹⁷ North Atlantic Council, 2002, Enclosure B, pp. 1–3, Enclosure F, pp. 1–2.

¹¹⁸ Bundesministerium für Wirtschaft und Klimaschutz (BMWK), VS-NFD, 2023, see Part 2, pp. 1–3, Part 3, pp. 1–6.

¹¹⁹ Ibid., Part 3, No. 3.1–3.7, pp. 1–5.

technologies such as cloud services, unless explicitly cleared¹²⁰. 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.¹²¹

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 which will be discussed in the following paragraphs.

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.¹²² To address this issue, the Bundeswehr initiated the Integrated Standard-Application-Software-Product Family (SASPF) program,

¹²⁰ *Bundesministerium für Wirtschaft und Klimaschutz (BMWK)*, VS-NFD, 2023, Part 3, No. 3.4.1–3.4.5, pp. 4–5.

¹²¹ *genua GmbH*, Genua VS-NFD, 2024.

¹²² *Stein, S.*, BPM in the Bundeswehr, 2011.

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.¹²³ These integrations were further complicated by stringent regulatory requirements, diverse stakeholder interests and interoperability challenges with outdated technologies. Consequently, significant technical debt and complexity accumulated, extending the modernization effort well over a decade and clearly demonstrating the severe maintenance implications of legacy systems.¹²⁴

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.¹²⁵ Continuous software updates have been crucial throughout this extensive period, including notable upgrades like the integration of the RecceLite reconnaissance system in 2009.¹²⁶ 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. As Sebastian Sprenger reported in 2018 for Defense News, the 'digitization of the head-up-display likely will require the costly replacement of the aircraft's main computer, officials wrote.'¹²⁷ This upgrade was a NATO mandatory requirement, indicating how the outdated system architecture and hardware makes new mandatory requirements difficult to implement.

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.¹²⁸ Carver emphasizes that maintaining these aging software infrastructures consumes significant financial and operational resources, limiting investment in innovation and proactive modernization strategies.¹²⁹

In conclusion, effectively managing technical debt and mitigating software decay within military systems requires strategic, long-term investments, proactive modernization plans

¹²³ Stein, S., BPM in the Bundeswehr, 2011.

¹²⁴ Ibid.

¹²⁵ Skiba, T., Der Tornado, 2024.

¹²⁶ bundeswehrTornado

¹²⁷ Sprenger, S., 2018.

¹²⁸ Carver, K., Technical Debt, 2022.

¹²⁹ Ibid.

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 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 inherent 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.

3 Methodology

3.1 Research Design

This thesis employs a qualitative, exploratory research design to investigate how software decay and technical debt are addressed in the context of military software development. Given the previously established challenges of complexity, extended lifecycles and high security requirements, these factors cannot be fully captured by quantitative methods.

Alternatively a qualitative approach was chosen to allow for a deep insight directly from the practitioners. The study is based on a semi-structured expert interview, conducted with software developers and project managers at T-Systems IFS, Chapter Defense Systems. The goal of the interviews is to identify core challenges, strategies and constraints related to managing long-lived software systems in the military domain.

The object is not to derive generalizable results for the entire industry, but rather to uncover key themes and practices specific to this organization and operational context. The interviews were designed based on the findings of the literature review and structured around topics like the management of legacy systems, use of CI/CD, architectural conformity, code quality and organizational or regulatory constraints.

3.2 Data Collection

The interview guidelines were developed based on the key topics identified in the literature review, with a focus on the following areas:

- Handling of legacy systems and long-term maintenance
- Use of Agile methodologies and CI/CD
- Architectural decision-making and conformance
- Code quality assurance practices
- Organizational and regulatory constraints in military projects

The participants were selected through purposeful sampling, all being practitioners currently working on military software systems within T-Systems IFS, Chapter Defense Systems. In total, three interviews were conducted. All participants hold several years of experience in software engineering, testing or project management in military software projects. They currently work as Project Manager, DevOps Engineer and Test Engineer with all of them having a background in Software Engineering. Due to this, the three interviewees cover a wide spectrum of the different fields in software development.

The interviews were conducted in person, recorded with consent and transcribed for analysis. The duration of each interview was between 20 and 40 minutes.

3.3 Data Analysis

The transcribed interview data was analyzed using Thematic Analysis, following the six-phase approach outlined by Braun and Clarke (2006). This method was chosen for its flexibility and suitability in identifying and interpreting patterns across qualitative datasets. According to Braun and Clarke, the first step is the familiarization with the data to become aware of potential patterns and themes.¹³⁰ The second step is the generation of the initial coding. Codes are defined by Braun and Clarke as ‘a feature of the data (semantic content or latent) that appears interesting to the analyst’¹³¹ and capture important features for the research question. The third step uses those codes to create broader themes. This is done by collecting all relevant codes and forming theme piles.¹³² These themes are then reviewed and refined in the fourth step, as: ‘it will become evident that some candidate themes are not really themes, while others might collapse into each other’¹³³. In the fifth step, clear definitions and names for each theme are developed. This includes writing a detailed analysis of each theme and how it relates to the research questions.¹³⁴ Finally, the

¹³⁰ Braun, V., Clarke, V., 2006, p. 16.

¹³¹ Ibid., p. 18.

¹³² Ibid., pp. 19–20.

¹³³ Ibid., p. 20.

¹³⁴ Ibid., p. 22.

sixth step is the production of the report. This is created through the combination of the identified themes, supported by selected interview quotes to illustrate the findings.¹³⁵ The coding process followed a hybrid approach, combining both deductive elements based on the literature review as well as inductive elements to account for unexpected themes specific to the military software context. The analysis was conducted manually, using MAXQDA, a program to support the coding process and facilitate the organization of themes and codes.

3.4 Limitations of the Methodology

While the qualitative approach chosen for this thesis provides valuable insights to uncover practical challenges and strategies, it is not without limitations.

Firstly, the study is based on a small number of expert interviews, conducted exclusively with practitioners from T-Systems IFS, Chapter Defense Systems. While the participants provided rich and relevant insights, the limited sample size means the findings cannot be generalized to the entire military software industry. Instead, the goal is to capture in-depth perspectives on key challenges and mitigation strategies in a specific organizational context.

Secondly, the semi-structured interview format, while allowing flexibility may introduce inconsistent coverage of topics between the interviews. Some participants may have focused on certain aspects, while others may have limited knowledge in specific areas. This may lead to uneven data quality across the thematic areas.

Thirdly, the subjective nature of qualitative analysis means that the findings are influenced by the researcher's interpretation of the data. While efforts were made to ensure rigor and transparency in the coding process, there is always a risk of bias or misinterpretation.

Finally, given the sensitive nature of military software projects, participants were limited in what they could disclose. This was either due to confidentiality agreements or the nature of the projects. This may lead to a partial representation of practices and challenges, especially in areas of security, architecture and procurement.

¹³⁵ Braun, V., Clarke, V., 2006, p. 23.

However, these limitations do not undermine the value of the findings. The small, focused sample allowed for a deep exploration of software decay with military constraints, capturing insights quantitative methods might overlook. Additionally, the semi-structured format while flexible, enabled participants to share their practical issues and strategies in their experience. The confidentiality constraints further show the real life conditions which are encountered in military software projects which enhances the authenticity of the findings.

Therefore, in spite of these limitations, the methodology chosen for this thesis contributes valuable insights into managing software decay and technical debt. Furthermore, the limitations show potential areas for future research, like broader studies to generalize the findings.

4 Results

4.1 Introduction

This chapter will present the findings of the qualitative content analysis of the three semi-structured expert interviews. The interviews were analyzed by the coding framework of Braun and Clarke as described in the methodology chapter. The results are structured into nine key themes, each containing several subthemes that emerged from recurring topics in the data.

4.2 Challenges in Managing Long-Lived Military Software

As established in the literature review, military software systems are typically designed for extended operational lifespans, sometimes for decades. This leads to significant challenges across technical, organizational and structural dimensions. From the expert interviews, the following subthemes emerged: System Complexity, Legacy Technologies and Compatibility Issues and Organizational Challenges.

4.2.1 System Complexity

One key challenge in maintaining long-lived military software lies within the complexity of the systems. This is not only due to the accumulation of new features but also the strict

requirements regarding reliability, testability and compliance.

As one interviewee stated: 'Je kritischer die Software, desto höher ist eben auch der Einfluss, den es auf unsere Arbeit hat, weil desto höher ist eben auch der Testaufwand [...] oder desto höher sind die Ansprüche an die Software.'

This highlights how the need for reliable and operationally safe systems leads to higher development and maintenance demands. Additionally, early architectural decisions which were often made decades ago, still constrain the current development process. For example, existing system architectures can no longer be easily modified: 'gewisse Entscheidungen [sind] vielleicht auch im Vorwege schon limitiert.'

Such constraints make implementing modern solutions or adapting to changing requirements difficult. This indicates a need for proactive simplicity and modularity in initial designs.

4.2.2 Legacy Technologies and Compatibility Issues

The continued reliance on outdated technologies represents another significant challenge in maintaining military systems. Many components are based on frameworks, languages and tools that are no longer widely used or supported. As one interviewee expressed, 'wir arbeiten noch mit sehr, sehr alten Versionen, [...] es ist alles sehr alt und es passt mit den Sachen nicht mehr zusammen.'

The problem not only exists in development but also in testing and tool integration. Test environments often rely on virtual machines and tools that require substantial manual effort, and are therefore prone for errors like misconfigurations or copy-paste mistakes. Moreover, forced tool usage from the customer side can lead to further complications: 'Wenn wir eine Vorgabe bekommen, dass wir mit bestimmten Tools testen müssen, dann ist das natürlich wieder mit extra Aufwand verbunden', as the tool might not be as commonly used and therefore might require additional training.

These issues contribute to software decay. Several participants described how initial shortcuts like skipping broken automated tests because they passed locally led to accumulated technical debt and a decrease in test reliability.

4.2.3 Organizational Challenges

Finally, organizational factors significantly impact the long-term sustainability of software. Knowledge loss due to employee turnover and the lack of documentation were recurring themes in the interviews. As one interviewee noted: 'Wenn du selbst Quellcode geschrieben hast, ihn dann zehn Jahre später noch warten und verstehen zu müssen, ist eben eine Herausforderung.' This challenge is reinforced by the retirement of experienced employees and the onboarding of younger developers unfamiliar with the legacy systems.

The interviews revealed that documentation and review practices were inconsistent. Review processes only took place before major releases or were conducted informally. One participant stated: 'Früher fast gar nicht. Also früher gab es quasi einen, der es gemacht hat und einen, der die offiziellen Tests dafür gemacht hat.' Today, although formalized practices like mandatory reviews and issue tracking are in place, their absence in older project phases continues to impact the maintainability.

4.3 Impact of Procurement and Approval Processes

The procurement and approval processes in Bundeswehr software projects have a significant impact on the evolution and maintainability of long-lived systems. Multiple participants across the interviews highlighted that even though technical shortcomings are often recognized, procedural, financial or contractual constraints obstruct the implementation of fixes. As one participant observed, although issues are raised, 'oft ist das dann für den Kunden attraktiver, ein neues Feature zu nehmen, statt irgendwas Bestehendes nochmal zu korrigieren.' This indicates a systemic preference for visible new features over addressing underlying technical debt.

4.3.1 Financial Constraints

One central barrier is a limited budget flexibility. In several cases, technical debt was acknowledged and documented, yet not addressed due to financial constraints. As one project manager noted, 'Wir hatten halt ein festes Budget, [...] aber da hat sich jetzt kein Folgeauftrag ergeben, deswegen liegen sie da jetzt erstmal.' In this context, the willingness to fix issues exists, but funding remains a bottleneck: 'Grundsätzlich ist der Wille da, es zu beheben, aber das Geld nicht.' These conditions create an environment where known issues remain unsolved and accumulate over time.

4.3.2 Procurement Barriers

In addition to funding, rigid procurement processes restrict the ability of developers to act proactively. Retroactive code improvements or technical debt remediation often require formal authorization, as developers are not permitted to modify code already delivered without explicit approval: ‘Ohne dass wir einen Auftrag bekommen, dürfen wir halt auch nichts machen.’ Even when shortcomings are identified, requests may be stalled due to insufficient contractual flexibility. This is particularly problematic under traditional methodologies like the V-Model, where ‘ein Riesenwust an falschen Umsetzungen’ can emerge from misinterpreted or outdated requirements. As one interviewee pointed out, greater contractual flexibility and agility could enable a more responsive development process: ‘Wenn man halt in den Verträgen schon leichte Spielräume schaffen würde, [...] wäre schon dem Problem viel geholfen.’

4.3.3 Slow Approval Processes

The formal structure of software certification further increases the lack of flexibility. Even minor adjustments are subject to lengthy testing and approval processes. One participant described how a simple removal of unreachable code is often not pursued: ‘Weil die Prozesse eben so aufwendig sind, wird das dann eben normalerweise nicht gemacht.’ This is not due to technical difficulty but to procedural formalities: ‘Die Softwareänderungen [dauern] manchmal 10 Minuten, aber der Prozess mit Testen und Nachweis und Lieferung und Freigabe dann durchaus mal eine Woche.’ These approval processes are not only time-consuming but also discourage iterative improvement and create a cautious approach to change.

4.4 Technical Debt and Software Decay Practices

The interviews revealed that technical debt and software decay are recognized challenges in military software development. While they are not always tracked formally, they are commonly acknowledged, with participants stating that both strategic and pragmatic decisions about when to accept or mitigate them are made. Four subthemes emerged from the interviews: tracking and visibility, consequences, sources and the handling of existing technical debt.

4.4.1 Tracking and Visibility of Technical Debt

The interviews showed that tracking of technical debt varies between projects. In some cases, teams maintained issues of known technical debt in a backlog, even when they could not address them immediately. One participant explained, ‘die potenziellen Verbesserungen [werden] als Issue aufgenommen und mit einem Won’t do Label markiert, [...] sodass wir im Nachgang jederzeit gucken können, das sind Dinge, die wir besser machen könnten.’ However, others stated that such documentation is often missing. As another interviewee noted, ‘Also getrackt wird es auf jeden Fall nicht. Das wüsste ich jetzt nicht, dass sich das jemand irgendwie merkt, dass da und da irgendwie was noch zu tun wäre.’ This discrepancy indicates inconsistent practices regarding the visibility of known technical debt, often depending on project maturity or individual initiative.

4.4.2 Consequences of Technical Debt

Unaddressed technical debt was widely viewed as having long-term negative consequences on development efficiency. One interviewee reflected, ‘Man hat, glaube ich, eine Menge Zeit da verloren und man hätte, glaube ich, wenn man im Vorwege schon solche Dinge gemacht hätte, [...] eine Menge Aufwand sparen können.’ This sentiment was confirmed by others, highlighting that the cost of technical debt is often not immediately visible but manifests in the long run, leading to increased maintenance efforts and reduced system reliability.

4.4.3 Sources of Technical Debt

Through the interviews, several sources of technical debt, ranging from outdated infrastructure to organizational pressure could be identified. Time pressure emerged as a dominant cause, with participants stating that deadlines often constrain quality practices: ‘Andererseits haben wir auch die Deadlines [...] von daher ist es immer eine Abwägung [...], gerade eher gegen Ende der Laufzeit [...], wenn Software nicht optimal ist, solange eben keine Fehler drin sind.’ This acceptance of a ‘good enough’ solution under time constraints is a recurring pattern.

Additionally, legacy systems contributed significantly to the accumulation of technical debt. Outdated tooling and incomplete test maintenance were cited repeatedly as evident in the

following example: ‘Da hat man die Testskripte ewig nicht angefasst [...] und jetzt habe ich gefühlt wieder ein paar Monate gebraucht, um die alle auf Stand zu kriegen [...]. Das ist halt [...] diese technische Schuld, die wir im Vorweg hätten verhindern können.’ Moreover, knowledge gaps introduced by inconsistent documentation and employee turnover were another root cause: ‘Wenn Dinge behoben worden sind im Quellcode, aber in den Testskripten nicht’.

4.4.4 Handling of Existing Technical Debt

Participants reported both proactive and passive strategies for handling technical debt. In more proactive cases, debt was addressed during related work. One participant described this approach as follows: ‘Wenn wir jetzt neue Dinge überarbeiten, gucken wir, können wir das an anderen Stellen auch vereinfachen, können wir Sachen zentralisieren.’ In contrast, there were also instances where technical debt was ignored due to effort or time constraints: ‘Wenn es dann wirklich nur um nicht optimale Implementierung geht, dann wurde es dann meistens doch eher unter den Teppich gekehrt.’ This underscores the role of effort-benefit considerations in managing long-term quality.

4.5 Mitigation Strategies in Military Software Projects

In order to address the accumulation of technical debt and the long-term effects of software decay, the interviewed experts described several mitigation strategies that have been implemented in their projects over time. These include improvements to the documentation, an increase in automation, use of refactoring and formalized code review processes.

4.5.1 Documentation Improvements

While documentation was often described as historically lacking, recent practices have begun to address this gap. Lightweight documentation tools like Markdown became the preferred choice for documenting over traditional tools like Word. This is due to ease of use and the ability to integrate them directly into the development workflow. One expert emphasized the convenience of ‘einfach in der Entwicklungsumgebung heraus, da schnell was zu dokumentieren,’ particularly using Markdown-based approaches. Additionally, issue tracking systems like Jira were mentioned not only for task management but to trace changes and problem resolution, providing a project memory. Nevertheless, some

participants acknowledged that automated document generation is still not used widely due to complexity.

4.5.2 Automation

Automation emerged as a particularly influential strategy, seen as both a time-saver and quality enabler. All three interviews highlighted a move from legacy CI tools like Jenkins toward more integrated solutions like GitLab. Automation is now widely used in builds, tests and static code analysis. One practitioner noted that, ‘man kann es nicht auslassen [...] es wird ja automatisch gemacht’, referring to the inevitability of running the automated test pipelines on commits to the codebase. This shift has not only allowed for faster feedback loops but also for parallel testing through containers as well as enhanced scaling and efficiency. Test reliability and frequency have improved as well, especially in regression testing.

4.5.3 Refactoring

Refactoring is practiced increasingly, although with some constraints. While historically uncommon, agile workflows have made it more normalized. According to experts, this is due to code reviews or developer initiative and sometimes results in a complete rewrite of problematic components. One developer recounted the rewrite of a poorly maintainable test script: ‘wir schmeißen ihn weg und schreiben ihn komplett neu, [...] dadurch ist er jetzt viel besser, viel wartbarer.’ This illustrates how refactoring can increase long-term maintainability, even at the cost of short-term effort. Nevertheless, the previously mentioned procurement and approval processes hinder such refactoring efforts, especially of older, already established code. Adding on to this are resource constraints, limiting the formalization of refactoring tickets in some teams.

4.5.4 Code Reviews

Systematic code reviews were described consistently as a critical practice for modern mitigation efforts. The standard has shifted from informal peer checks to mandatory multi-eye reviews, often documented in GitLab creating a clear audit trail. As one

interviewee explained, ‘jede Änderung wird eben nochmal gereviewt’ underscoring that no code is merged without review. These processes are not only seen as a quality assurance measure but also as a knowledge transfer opportunity. Review discussion often helps to find hidden issues and prevent accidental technical debt early on. In some contexts, less experienced developers also participate, allowing for a teamwide transfer of code knowledge.

4.6 Quality Assurance and Testing

Ensuring long-term software quality is essential in military software projects, particularly due to the extended lifespans and strict regulatory demands. The interviews revealed a combination of different strategies to ensure quality, an approach which has evolved over time. Core elements included overcoming testing challenges, the use of tools and metrics, maintaining test coverage and developing systematic testing processes.

4.6.1 Testing Challenges

Participants noted several recurring challenges in maintaining test quality. The most significant of these was sustaining automated testing pipelines: ‘Wenn man jetzt ein Schwäche nennen müsste, ist es natürlich der Aufwand dahinter, das Ganze so weit zu automatisieren.’ Additionally, legacy testing processes and infrequent test execution in earlier project phases were cited as obstacles. As a result problems with the tests were detected much later, resulting in high maintenance efforts. As one interviewee noted, ‘bei gewissen Projekten war es halt so, da hat man die Testskripte ewig nicht angefasst,’ which highlights how neglecting test maintenance can cause significant issues in the future.

Another challenge was the lack of traceability, especially when tests were changed much later than the code. One interviewee stated that ‘wenn man dann nicht nachvollziehen kann, was im Quellcode geändert worden ist, [...] dann ist das schon sehr lästig’, indicating the importance of maintaining a clear connection between code changes and test updates. Furthermore, a lack of consistency in responsibility for test maintenance was noted. This responsibility was often divided between developers and testers, which created additional coordination issues. Requirements themselves were sometimes found to be vague, making precise testing difficult.

4.6.2 Use of Metrics and Tools

Quantitative quality metrics have been increasingly used to assess and improve software quality. Metrics such as code coverage, complexity and static code analysis findings were frequently mentioned. One interviewee stated: ‘Wir haben die Codecoverage als Kennzahl und wir haben die statische Codeanalyse, die uns dann Findings quasi hochwirft.’ Tools like Logiscope and Jenkins play a key role in these evaluations and are integrated into the CI pipelines for continuous feedback.

The use of unit test metrics and graphical feedback was also mentioned as a way to improve code quality. Thresholds for metric violations were set by the client, creating external pressure as well as standardization. In addition to technical tools, project management tools like Jira contribute to traceability and error tracking, showing that quality encompasses both code and process aspects.

4.6.3 Test Coverage and Test Maintenance

Ensuring high test coverage as well as maintaining test scripts were identified as key strategies to prevent software decay. Automated coverage reports allow developers to monitor how well the code is covered by tests: ‘dann halt auch so Messwerte wie Test-Coverage mir auch ausgeben lassen [...] und kann da halt gegebenenfalls nachbessern.’ The integration of test automation into CI/CD pipelines was described as both a reliability and a knowledge preservation mechanism.

Participants emphasized the need to regularly update test scripts immediately after code changes, especially in long-lived projects. One expert advised: ‘Testskripte immer direkt nach der Softwareanpassung auch mit anpassen,’ warning that even developers might struggle to remember the reasons for their own code changes after a few months. The benefit of nightly test runs and regression detection was also highlighted, described as crucial for sustainable quality assurance.

4.6.4 Testing Practices

Over the last years, testing practices have evolved from manual execution to a more automated and systematic approach. Several interviewees recalled earlier practices where tests had to be executed manually over days: ‘Vor zehn Jahren [...] mussten viele von diesen Testprozessen per Hand durchgeklickt werden.’ In contrast, today’s pipelines

integrate code compilation, test execution and reporting using tools like GitLab or Jenkins. The resulting transparency was described as a significant improvement, allowing developers to receive detailed failure reports and even screenshots in case of User Interface (UI) errors. As one expert stated, 'den Überblick zu haben [...] ist schon ein großer Gewinn.' Despite these advancements, some areas like system-level tests still rely on manual testing. Nevertheless, the overall trend is towards a more automated and integrated testing approach and is seen as a major improvement in software quality assurance.

4.7 Agile Methodologies in Military Projects

The introduction of agile methodologies into military software development marks a significant shift from traditional approaches. While usually reliant on plan-driven models such as the V-Modell XT, participants described how agile methodologies, particularly Scrum, are being adopted in some form in certain projects. The reported effects of this shift include a change in team dynamics, software quality and the ability to adapt to changes. Nonetheless, several constraints still remain.

4.7.1 Knowledge Sharing and Collaboration

A key benefit of agile practices in the military context is improved knowledge sharing and collaboration. Participants emphasized the positive impact of regular stand-ups, code reviews and a collaborative problem-solving culture: 'Dadurch, dass wir täglich miteinander reden und täglich Fragen stellen und im Review mit anderen Personen zusammensitzen, da wird sowas dann halt immer [...] gefördert.' Agile methods work against the silo knowledge structure typical in legacy projects. By encouraging broader ownership of code and increased onboarding support for new team members, a participant noted a shift away from the 'Keller-Mentalität'. According to him, this shift creates more learning opportunities as well as project resilience.

4.7.2 Limitations in Agile Implementations

Despite the positive aspects, the implementation of agile methodologies faces several structural challenges in military software projects. Firstly, agile practices are still relatively new: 'Agil ist tatsächlich ein neues Thema im Bereich militärisch. Das gab es bisher gar

nicht.’ Secondly, organizational and contractual constraints restrict the iterative nature of agile development. For example, procurement officers would be required to be more involved and responsive in iterative decisions, which would increase their workload. Additionally, early-phase resource investments, common in agile approaches, clash with the traditional planning and fixed-scope contracts: ‘Es wird natürlich im Vorwege teurer, damit man hinten raus [...] technische Schuld [...] abfedert.’ These constraints highlight the need for an adapted agile model that can accommodate the regulatory and hierarchical structures of military software development.

4.7.3 Use and Impact of Agile Methods

In practice, tools like GitLab and Jira allow for better transparency and allowing for a better integration of tasks, coding and deployment. Participants emphasized that these tools facilitate iterative development cycles, quicker feedback loops and improved visibility of ongoing work, aligning closely with agile principles. Agile processes also improve responsiveness. Teams can now react more easily to changing requirements or design flaws. By breaking work into smaller increments and regularly revisiting priorities, scope issues can be identified and addressed earlier: ‘Der Scope [ist] zu groß geworden und [wir] haben das dann abgekapselt in ein eigenes Item.’ Moreover, the agile approach of having a potentially shippable product at the end of each sprint has improved planning discipline and improvement cycles. Nevertheless, participants stressed that the full potential of these benefits can only be realized with a more flexible and responsive organizational structure.

4.8 Influence of Regulatory and Security Requirements

The development and maintenance of military software is heavily influenced by regulatory and security requirements. These requirements not only shape the development process but also impact both technical decisions and organizational flexibility. The interviews revealed that while these requirements are essential for maintaining safety and reliability, they also introduce significant challenges that can hinder adaptive development practices.

4.8.1 Impact on Flexibility

A recurring theme which emerged is the limited flexibility in making changes to the software once it has been delivered. Participants described how even minor adjustments need to

pass through extensive testing and approval processes, which can take weeks or even months. As one interviewee noted: ‘Software kann nicht einfach geändert werden, sondern wir müssen einen recht komplexen Prozess durchlaufen,’ which highlights the bureaucratic complexity that can restrict maintenance and optimization efforts. Even clearly obsolete components such as unreachable or ‘dead’ code often remain in the system due to the time and effort required for formal approval.

The flexibility issue has a direct impact on the maintainability. As another expert pointed out, even if a better solution is available, ‘wir dürfen nicht einfach beliebig irgendwelche Sachen ändern,’ due to certification status and client approval. A lack of contractual agility further increases the problem, with interviewees emphasizing that small changes which could improve quality are often not pursued because ‘die Spielräume aktuell zu wenig genutzt werden.’

4.8.2 Standards Compliance

The interviews underscored the stringent compliance requirements around coding standards and documentation. All projects adhere to formal coding guidelines defined in cooperation with the client and use tools like Logiscope to automatically enforce these standards. These metrics not only insure code quality and maintainability but also serve to enforce compliance with the regulatory framework. As one expert noted, ‘wir haben im Projekt auch feste Codierrichtlinien [...] die dafür gemacht sind, den Code am Ende wartbar zu halten.’ Furthermore, a formal review is required for any delivered code, with audit trails and standardized documentation ensuring traceability and accountability. One participant described how ‘es mussten regelmäßige Reports angelegt werden, die dann auch auf Nachfrage dem Kunden vorzulegen sind,’ reinforcing how compliance processes are integrated into the development.

4.8.3 Security and Reliability Requirements

The security and reliability requirements tied to critical military systems add another layer of complexity. Systems with higher criticality are required to have elevated test coverage, often reaching 95%, and must adhere to strict validation practices. That was framed both as a challenge and a motivator as one participant remarked that ‘je kritischer die Software, desto höher ist eben auch der Testaufwand.’

Despite these requirements, the interviews suggest that the security and reliability standards

are also intrinsically motivating for the team. One interviewee observed that teams are ‘eher intrinsisch getrieben,’ indicating a strong internal commitment to quality, beyond the external regulations. Some participants even suggested that, in certain cases, regulatory pressure may be less than the actual quality standards that the teams set for themselves, describing the standards as ‘eher ein bisschen lascher.’

4.9 Successful Examples

Despite the many challenges discussed throughout the interviews, the participants also described a number of successful practices that have mitigated software decay and reduced technical debt in their projects. These successful measures offer insights into the strategies which have worked well in practice and could be applied to other projects.

4.9.1 Effective Measures against Software Decay

One of the most frequently mentioned successful measures was the introduction of automation, particularly in the areas of testing and continuous integration. The shift from isolated VM-based tests to containerized, pipeline-driven environments was described as a key improvement. This change enabled faster, parallel test execution and improved visibility across the codebase. As one participant stated, the move to GitLab and automated pipelines allowed for ‘schnelle Ergebnisse’ and ensured that ‘jeder, der ein Issue bearbeitet, auch sicherstellen muss, dass die Tests [...] durchlaufen.’

Another significant improvement came from refactoring efforts, often initiated through broader modernization efforts. During the migration from SVN to Git, teams took the opportunity to ‘Ballast ab[zu]werfen’ removing unnecessary legacy code and simplifying complex structures. These efforts directly improved code clarity and maintainability, making it easier for new developers to engage with the system: ‘Viele Methoden sind eben kürzer geworden, leichter lesbar ... [so dass es] auch einem Laien dann leichter fällt ... zu verstehen.’

The CI/CD infrastructure was described as a continuously evolving system. Originally implemented as a basic build pipeline, it has grown to incorporate automated testing, static code analysis and compiler warning tracking. This evolution has made the development process ‘praktischer und bequemer,’ illustrating how technical investments translate into

tangible benefits for the team.

Lastly, the use of agile practices was highlighted as a driver of responsive and adaptive development. In one case, an agile approach allowed a fast response to a customer request that otherwise would have been ignored due to implementation complexity: ‘... sehr schnell auf einen Kundenwunsch reagiert ... korrigiert und mit wenig Aufwand umgesetzt.’ This example illustrates how agile and technical practices can directly reduce software decay by addressing issues early and efficiently.

4.10 Recommendations For Future Practices

The final theme of this study focuses on the recommendation for future practices by practitioners to reduce long-term software decay and technical debt. Interviewees consistently highlighted proactive measures and early intervention as key strategies. This is reflected in the recommendations targeting agile adoption, continuous documentation, sustained test maintenance and upfront investment.

4.10.1 Agile Expansion

Agile practices, while still uncommon in military software contexts, were regarded as promising in counteracting technical debt. Participants reported greater responsiveness to emerging needs and improved alignment with user priorities when using agile methods. As one interviewee described, agile approaches made it possible to ‘mit wenig Aufwand’ adapt systems quickly to customer demands. The iterative nature of sprints also enabled isolation of complex issues and allowed for more focused problem-solving. In a broader view, agile working was seen as an opportunity to address usability shortcomings and allow for better prioritization. However, participants noted that realizing these benefits requires contractual flexibility and the ability to reallocate resources mid-project, an area that still is problematic due to the rigid procurement settings.

4.10.2 Focus on Test Maintenance

Interviewees emphasized the importance of maintaining and adapting test scripts. Neglect in this area was seen as a major source of avoidable technical debt. One expert reflected

that without timely updates, even small code changes resulted in significantly high maintenance efforts. Automated pipelines helped reduce this burden, but maintaining the relevance and accuracy of test scripts remains a manual task. In long-lived software systems, proactively synchronizing code and test logic was viewed as a foundational element of sustainability.

4.10.3 Continuous Documentation

Improving documentation emerged as another important mitigation approach. Experts pointed out the necessity of reducing the threshold of developers to contribute to documentation, particularly by adopting lightweight and integrated tools. For example, Git and Markdown were favoured due to their accessibility and ease of use. Beyond technical documentation, traceability through issue tracking and automated records of reviews was also seen as a step towards preserving architectural and design knowledge over time. As one interviewee stated, clear documentation ensures that ‘auch Jahre später ... nachvollzogen werden kann’, supporting maintainability across generational changes.

4.10.4 Early Investment

A recurring insight across all interviews was the value of early, deliberate investment in software maintainability. Participants stressed the benefits of planning from the beginning for a system’s long lifecycle. Strategic choices such as introducing modularity, automated tools and test infrastructure early on were seen to yield long-term benefits. One practitioner noted that ‘wenn man es direkt zu Beginn macht, ... hat [man] auch den geringsten Aufwand nachher’, capturing the concept that frontloading technical quality reduces costs and complexity in the long term. Moreover, even small-scale improvements, if made consistently and early, were regarded as effective in preserving system health over time.

5 Discussion

5.1 Introduction

This chapter interprets the results of the study and discusses their implications, based on the guiding research questions and the literature review. The goal is to deepen understanding of how software decay manifests in military software systems and how it can be mitigated successfully, despite having to deal with highly regulated environments and long lifecycles. The central research question of this thesis is:

How can software decay in military systems be prevented or slowed down?

To explore this overarching question, two sub-questions were examined:

- What are the main causes and symptoms of software decay specific to the military software domain?
- Which mitigation strategies are currently in use at T-Systems IFS, Chapter Defense Systems and how effective are they?

The following discussion is structured around these questions. Firstly, the main cause of software decay and contributing factors are discussed, based on the challenges identified in the previous chapters. Particular attention is given to legacy constraints, procurement processes and organizational practices unique to the military domain. Thereafter, the mitigation strategies currently employed at T-Systems IFS are evaluated, focusing on their practical implications, effectiveness and limitations. Finally, the implications of these findings are assessed in terms of their relevance to broader software engineering practices, particularly in regulated, long-lived systems.

The discussion aims to bridge the gap between empirical observations and theoretical concepts as well as to provide practical recommendations for future projects to reduce technical debt and maintain long-term software quality in military systems.

5.2 Causes and Symptoms of Software Decay in the Military Domain

Addressing the first sub-question regarding the main causes and symptoms of software decay, the findings of the interviews align with existing literature. The interviews highlighted the inherent complexity of long-lived military systems, reinforcing Brooks' (1987) concept of

‘essential difficulties’, like complexity, conformity and changeability, which increase over time.

One of the most prominent causes is the long life cycle of military systems, which often leads to accumulation of outdated code, architectural rigidity and the use of obsolete technologies such as older C++ or Ada versions. This is a problem throughout the military domain, as can be seen by the previously mentioned example of the German Air Force, where the Tornado system is still in use after 40 years. This supports the findings of Lehman and Belady (1976) who argued that software evolution inevitably leads to increased complexity and decay, especially in systems with long lifecycles. Additionally, interviewees reported difficulties caused by loss of knowledge when original developers retire, an issue Parnas’ (1994) highlighted regarding documentation inadequacies and implicit knowledge.

Another recurring theme was the lack of early investment into maintainability of software as well as adaptability of the architectures. Several interviewees pointed out that many systems were designed decades ago, meaning the original architects and developers are no longer available. This makes it difficult to fully understand the system’s architecture and design decisions, in turn making modifications and updates more risky and time-consuming. Over time, this contributes to a buildup of technical debt, particularly when quick solutions are chosen to meet delivery deadlines. This aligns with Parnas’ ‘Lack of movement’ theory, where a lack of proactive maintenance and refactoring leads to these problems. These issues were also noted by Shyu (2017), who emphasized that due to the long lifecycles, patchworks of legacy code are very common. She proposed more use of flexible architectures as well as self-testing modular code to allow for easier updates and modifications.

Another notable finding is the influence of regulatory rigidity on software decay. The military domain’s stringent compliance and documentation requirements, while necessary for safety and reliability, lead to an increase in software decay, due to limited flexibility. The constraints prevent proactive refactoring and architectural improvements, resulting in a focus on short-term fixes over long-term stability.

Organizational challenges also play a central role. Teams often work under rigid procurement and approval processes, where minor improvements may require extensive justification and approval from the stakeholders. This leads to non-functional aspects such as internal refactoring being neglected. Often developers are aware of weaknesses in the code but are unable to address them due to lack of authority and budget. This was also mentioned by Schröder (2022) who stated that the current standard contracts do not allow

any form of agility or iteration.

Furthermore, the interviews revealed a form of acceptance towards technical debt as an inevitable cost of doing business. This acceptance not only allows technical debt to persist but also can create a cycle where decay is managed reactively rather than proactively. Such an acceptance of technical debt underscores the need for a cultural shift on all levels, whereby technical debt is recognized as a more critical issue to be addressed than the implementation of new features. These findings align with McConnell's (2007) observations regarding the impact of organizational constraints on technical debt management.

Symptoms cited by participants included unreliable test scripts, incomplete documentation as well as system behaviour that is difficult to predict or reproduce. A particularly significant indicator of software decay is the difficulty in onboarding new developers. When new personnel struggle to understand the system architecture, legacy code and the rationale behind design decisions, it often reflects insufficient documentation, lack of modularization and a high need for implicit knowledge. These problems can be found in the definition of technical debt from Kruchten et al. (2012), which encompasses not only the code itself but also the surrounding processes and practices. These symptoms reflect structural issues such as lack of modularization, outdated documentation practices and insufficient test infrastructure, confirming theoretical assertions by Cunningham (1992) regarding the long-term maintenance consequences of technical shortcuts.

In summary, long lifecycles, legacy constraints, regulatory rigidity and organizational challenges not only align with theoretical concepts but also provide a practical understanding of how software decay manifests in military systems.

5.3 Mitigation Strategies and Their Effectiveness

Regarding the second sub-question about current mitigation strategies and their effectiveness, the analysis of the interviews reflects practices that align with other commercial practices, albeit with some unique military constraints.

One of the most prominent developments across all interviews was the increased use of automation in testing, building and deployment. The introduction of containerized environments and CI/CD pipelines allows for regular execution of tests and static code analysis, without the need for manual intervention. This not only improves the frequency of tests but also their reliability, as regressions and integration issues can be detected

earlier. This is due to the fact that developers can no longer skip tests easily, since they are now part of the automated build process. Such automation forces discipline that might otherwise be neglected under time pressure. Additionally, the automated pipelines allow for a scaled development process and ensure consistent quality, which is particularly important in long-lived systems. However, as Fowler stated, a functioning test suite and build process is crucial for CI to work correctly. Therefore both factors need special attention to make sure the pipeline is used as intended. Additionally, a correct configuration of the pipeline must be ensured for, as Silva and Bezerra (2022) found, the wrong quality levels and configuration can lead to a decrease in software quality.

Another practice used against software decay was the formal code review. This has evolved from inconsistent and informal peer checks into structured, documented processes involving multiple reviewers. Participants emphasized that code reviews not only help to discover mistakes and prevent technical debt but also serve as a way to share knowledge and create a team wide quality assurance. Code reviews are often combined with GitLab-based workflows, allowing a transparent history of changes and discussions. This is backed by McIntosh et al. (2016) their study found that the right participants play a major role in the effect of the the review, which was seen not as important by the interviewees.

Refactoring, while not always practiced over the entire codebase, is used in projects that have a more agile approach. Interviewees stated that in these projects, problematic components are occasionally rewritten or restructured. However, resource constraints and rigid approval processes limit a more systematic approach to refactoring. As one developer explained: 'Wir haben jetzt keine festen Refactoring-Tickets, aber prinzipiell ist es jedem freigestellt, eigene Tickets zu erstellen.' This indicates that, while there is a local initiative to address technical debt, a border process is still lacking. These reasons against refactoring align with the studies of Kim et al. (2014) and Tempero et al. (2017), which named costs, risks and lack of support from management as the main reasons why developers did not refactor, even though they saw it as beneficial.

Interviewees also stressed the importance of documentation improvements, especially the use of lightweight tools like Markdown or systems like Jira to track changes and decisions. Documentation is often a weak spot in legacy projects, but newer practices emphasize the importance of keeping documentation up to date by lowering the barrier for contributions by embedding it into the development process.

Despite these promising strategies, their effectiveness are often limited by domain constraints. These are either procurement, budgeting or legacy constraints. Automation, for example, can be highly effective once implemented, but requires significant initial investment which is not always feasible in the military domain. Similarly, while reviews and refactoring add value, they require time and the freedom to act, which is often limited by rigid processes and approval hierarchies.

In summary, mitigation strategies such as automation, code reviews, refactoring and better documentation are being adopted to prevent software decay. Although they are deemed as effective by practitioners, their impact on each project and its restrictions can vary considerably. Team maturity and organizational willingness to invest in the long-term health of the software are crucial factors in determining the success of these strategies. This aligns with the literature, which suggests that technical debt management is not only a technical challenge but also cultural and organizational.

5.4 Recommendations for Future Practice

Based on the current practices and challenges, the interviewed experts outlined several recommendations directed at better preventing software decay in future projects. These suggestions aim to improve the long-term maintainability and reduce the accumulation of technical debt within the constraints of the military domain.

A key recommendation was the expansion of agile practices, enabling a more flexible and iterative approach to the development. Agile methods were not only seen as useful to provide short feedback loops but also to prevent a misinterpretation of requirements. Additionally, interviewees found that sprints, issue prioritization and iterative delivery allowed a better communication with stakeholders and prevented technical debt. This aligns with the findings of Leite et al. (2024), who found that agile practices were the most commonly used method for managing technical debt and by integrating them into the workflow, teams could reduce technical debt accumulation. However, successful and effective agile practices require the contractual framework to allow more flexibility and adaptability, which is not the case in many projects.

Another important theme was the prioritization of test maintenance. Several participants stressed the fact that test script should be updated with the code changes, rather than

being an afterthought. Interviewees emphasized that even with your own code, the difficulty of understanding the code increases with time. By maintaining the tests parallel to the code, consistency of the tests is ensured as well as knowledge loss prevented. Through this, a common issue is also avoided where old tests decay while new features are introduced.

Closely related to this is the recommendation to work on continuous documentation, especially in a domain where software is bound to have a long lifespan. Documentation should be integrated into the development process, rather than being a separate task. Tools like GitLab or Jira allow the easy documentation of changes, decisions and the history of the project. In addition to the technical documentation, a focus on clear code comments is recommended. All these practices also help new developers to onboard faster, as the issue of rotating or retiring developers was mentioned as a major challenge.

Finally, the experts underlined the importance of early investment into maintainability. They emphasized that making initial costly decisions like modularization, testability or automation pays off significantly over time. As the average lifecycle of military systems is often decades, the initial investment is mostly negligible compared to the long-term costs of maintaining a decaying system. To prevent the initial cost being too high, the recommendation was to start small and gradually build up these practices, rather than trying to implement them all at once. This would allow them to be implemented even under tight timelines and budget constraints.

One aspect that was not mentioned as often by the interviewees as in the literature is the refactoring process. While Moser et al. (2008) provided evidence in a case study that regular refactoring decreases complexity and prevents maintenance issues, the interviewees showed a reluctance to refactor. This is largely due to the fact that a new contract from the customer is needed to refactor old code, and rarely happens as new features are prioritized over refactoring. Additionally, the risk of introducing new bugs is often seen as too high, especially in a domain where safety and reliability are paramount. This could be prevented by having a functioning test suite, which would allow for a more confident refactoring process. With a proper test suite as a safety net, the customer could be convinced to allow for refactoring, as the risk of introducing new bugs is significantly reduced.

Another practice that was not mentioned by the interviewees is the use of quality gates in the pipelines. While the interviewees confirmed the use of static analysis tools, there is no direct enforcement of the metrics. As the metrics are already investigated, the

recommendation would be to enforce certain thresholds in the pipeline, preventing addition of new code that does not meet the requirements. Uzunova et al. (2024) found that the use of quality gates allows for real time feedback and allows developers to catch issues early in the development process.

Overall, these recommendations show a clear awareness among practitioners of the challenges posed by software decay and the need for a proactive approach for mitigation. While the constraints of the military domain are significant and unlikely to change in the near future, even small procedural and cultural changes can make a substantial difference in the long-term maintainability of military software systems.

5.5 Limitations of the Study

Due to the nature of the qualitative research design, this study is not without limitations. Firstly, the empirical results are based on a semi-structured interview with a relatively small group at a single organization. While this allows for a detailed, in-depth exploration of specific organizational practices it also introduces context-specific bias, potentially limiting the applicability of the findings to other military organizations or domains.

Secondly, the selection of interviewees could create an additional bias, as the participants were directly involved with military projects within a single organization. Other, potentially external stakeholders such as procurement agencies or regulatory bodies were not captured. This omits different viewpoints that could have provided further context or insights into the constraints and practices.

Furthermore, the qualitative approach involves a subjective influence of the researcher. While efforts were made to employ a rigorous coding process, the risk of bias remains unavoidable to some extent.

Lastly, given the dynamic nature of software development, even within the military domain, the findings present a view of the current conditions and practices. Changes in regulations, technology or organizational practices may alter the relevance of the findings over time.

Despite these limitations, the insights gained from this study offer a valuable understanding into practical challenges and mitigation strategies within the military domain. Potential further research could expand on these findings by using a quantitative approach to validate the results across a broader sample of organizations and projects.

6 Conclusion

This thesis aimed to investigate the question of how software decay can be prevented or slowed down in military systems. To support this goal, the domain-specific challenges as well as the evaluation of mitigation strategies were examined. To aid this investigation, two sub-questions were explored: the main causes and symptoms of software decay in German military software projects and also the mitigation strategies and their effectiveness at T-Systems IFS, Chapter Defense Systems.

The study found several causes of software decay that are specific in the military domain. These include the long lifecycles of the systems, which result in outdated technologies and code as well as rigid architectures. This rigidity is further increased by the high regulatory requirements and procurement frameworks which often prioritize short-term fixes over long-term solutions, thus increasing technical debt. Organizational challenges such as rigid approval processes and an acceptance of technical debt as inevitable further contribute to the decay. Key symptoms identified include unreliable test scripts, incomplete documentation, unpredictable system behaviour and a significant difficulty in onboarding new developers.

The study also evaluated the mitigation strategies currently in use at T-Systems IFS, Chapter Defense Systems. These include practices like automation, including containerized environments and CI/CD pipelines, structured code reviews, improvements to the documentation process as well as occasional refactoring. Automation has proven to be particularly effective as it enforces regular testing thus improving reliability, however requires careful configuration as well as an extensive early investment. Structured code reviews not only help catch mistakes in the code but also spread the knowledge across the team, which combats the prior issue of the encapsulated knowledge of the developers. Nevertheless, studies have shown that the full potential of code reviews rely heavily on involving the appropriate reviewers. The documentation processes were improved by integrating them into the development process using tools like Jira or GitLab to create a comprehensive

history of changes and decisions, significantly improving the maintainability of the systems. Refactoring, a very beneficial practice as shown by studies, can only be restrictively used, due to resource constraints and the regulatory and contractual requirements.

These findings have several implications for practitioners in the military software domain. Firstly, there is a clear need for some form of agile practices within the military context, to allow for iterative feedback and better stakeholder communication. Secondly, prioritizing regular test maintenance and continuous documentation could substantially combat knowledge loss and technical debt accumulation, a core problem in long lifecycle systems. Additionally, early investments into the maintainability of the system through modularity and automation will most likely pay off due to their long term nature. Through the robust test infrastructure, refactoring could become less risky and potentially more frequent, thus improving the overall quality of the systems. Finally, by integrating quality gates into the automated pipelines, the quality of the code can be controlled more directly, thereby preventing further decay.

This study is not without limitations. Due to the qualitative approach, small sample size and single organization, the findings may not be generalizable across the military software domain. Confidentiality restrictions also limited the depth of the analysis and the ability to provide specific examples. Nonetheless, these constraints reflect the real-world challenges underscoring the practical relevance and applicability of the findings.

Further research could involve a broader quantitative study, aiming to validate the effectiveness and viability of the identified mitigation strategies, particularly in various military settings. Additionally, a comparison across national and organizational contexts could provide deeper insights into universal and context specific practices.

Overall, despite the regulatory, procedural and organizational constraints of the military domain, this study shows that proactive measures can substantially mitigate software decay and technical debt. Recognizing software maintainability as an important investment rather than a cost is crucial for ensuring the long-term sustainability and reliability of military systems.

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
- Braun, Virginia, Clarke, Victoria* (2006): Using Thematic Analysis in Psychology, in: Qualitative Research in Psychology, 3 (2006), Nr. 2, pp. 77–101
- 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 Blencio 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* (Barriers 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]

Sprenger, Sebastian (2018): Germany Fears Steeper Tornado Costs after the UK Ditches Its Planes, (2018-04) [Access: 2025-05-15]

Stein, Sebastian (BPM in the Bundeswehr, 2011): Business Process Management in Big Organizations - Bundeswehr | ARIS BPM Community, (2011-05) [Access: 2025-03-31]

Appendix

Interviewleitfaden – Bachelorarbeit Softwareverfall

Einstieg und Hintergrund

1. Welche Rolle haben Sie aktuell im Bereich der Softwareentwicklung militärischer Systeme? Seit wann sind Sie in diesem Umfeld tätig und wie sieht Ihre typische Projektarbeit aus?
2. Mit welchen Arten von Softwaresystemen arbeiten Sie hauptsächlich? (z. B. eingebettete Systeme, einsatzkritische Plattformen, Verwaltungssoftware, Legacy-Systeme)

Legacy-Systeme und lange Lebenszyklen

3. Welche Herausforderungen begegnen Ihnen beim Umgang mit Legacy-Systemen oder langlaufenden Projekten?
4. Kommt es im Projektverlauf regelmäßig zu Problemen, die durch veraltete Strukturen oder historisch gewachsene Systeme entstehen?
5. Welche Auswirkungen haben lange Lebenszyklen auf Wartbarkeit und Weiterentwicklung?

Strategien zur Qualitätssicherung

6. Welche Maßnahmen setzen Sie ein, um Softwareverfall oder technische Schuld zu vermeiden oder zu reduzieren? (z. B. Refactoring, Reviews, Automatisierung) Sehen Sie diese Maßnahmen als sinnvoll?
7. Gibt es Bedingungen, unter denen technische Schuld bewusst in Kauf genommen wird? Wird bewusste Schuld getrackt?

Agile Methodik

8. Verwenden Sie agile Methodiken wie Scrum oder XP? Falls ja, wie wirken sich diese Praktiken auf den Umgang mit technischer Schuld und Softwareverfall aus?
9. Wenn keine agilen Methodiken verwendet werden, wie werden Refactoring und Schuldenabbau sonst durchgeführt?

Architekturmanagement

10. Wie wird die Systemarchitektur geplant, dokumentiert und gepflegt?
11. Wie wird sichergestellt, dass sich die Umsetzung an der vorgesehenen Architektur orientiert, insbesondere bei älteren Projekten?
12. Welche Herausforderungen treten bei der Weiterentwicklung von Architekturentscheidungen auf?

Automatisierung und CI/CD

13. Setzen Sie CI/CD in Ihren Projekten ein? Wie sehen die Prozesse konkret aus und hilft CI/CD gegen die zuvor genannten Probleme?
14. Welche Werkzeuge und Metriken verwenden Sie zur Qualitätssicherung im Build- oder Deployprozess? Was sind Ihre Erfahrungen mit diesen Werkzeugen? Gab es bestimmte Stärken oder Schwächen, die sich auf die Codequalität oder Wartbarkeit ausgewirkt haben?

Reviews und Tests

15. Wie wird bei Ihnen mit Code Reviews gearbeitet? Wer beteiligt sich, und wie wird der Prozess organisiert?
16. Welche Rolle spielen Tests? Nutzen Sie automatisierte Verfahren wie Unit-Tests oder Test-Driven Development? Gibt es bestimmte Tests, die Ihrer Meinung nach am besten zur Vermeidung von technischen Schulden helfen?

Besondere Rahmenbedingungen im militärischen Umfeld

17. Welche Anforderungen an Sicherheit, Zuverlässigkeit oder regulatorische Vorgaben wirken sich auf Ihre Arbeit aus?
18. Inwiefern beeinflussen Beschaffungsprozesse oder lange Freigabeschleifen Ihre Projekte in Bezug auf technische Schulden und Softwareverfall?

Abschluss

19. Können Sie ein Beispiel nennen, in dem eine Maßnahme zur Reduktion von technischer Schuld oder zur Vermeidung von Softwareverfall besonders erfolgreich war? Was genau machte diese Maßnahme aus Ihrer Sicht erfolgreich?
20. Gibt es weitere Erfahrungen oder Empfehlungen, die Sie im Zusammenhang mit Softwareverfall oder technischer Schuld teilen möchten?

Interview 1

Interviewer: Dann schon mal vielen Dank, dass du das Interview mit mir machst. Mal so ein bisschen zum Einstieg: Was ist deine aktuelle Rolle im Bereich der Softwareentwicklung und wie lange bist du dort schon tätig und was ist so deine typische Projektarbeit, die du machst?

Int-1: Ja, also ich bin jetzt insgesamt, die Ausbildung in dem Bereich hier mit reingezählt, 15 Jahre dabei. Während diesen 15 Jahren habe ich hauptsächlich Softwareentwicklung, Integration und Tests gemacht. Jetzt mittlerweile bin ich tatsächlich primär im Bereich Integration und Test unterwegs, mache aber teilweise noch im Bereich Softwareentwicklung, Fehlerbehebung, Softwarelieferung und so weiter.

Interviewer: Was sind deine Hauptssysteme oder wie sehen die aus, woran du quasi hauptsächlich arbeitest? Also sind das eingebettete Systeme, Verwaltungssoftware oder Altsysteme?

Int-1: Also überwiegend würde ich sagen, könnte man es glaube ich als Altsysteme bezeichnen, weil die Projekte schon sehr lange laufen. Im Grunde genommen sind das die Führungssysteme, wo dann die einzelnen Softwarekomponenten integriert sind.

Interviewer: Und wenn du sagst, die laufen schon eine Weile, was für Herausforderungen hattest du im Umgang mit diesen alten Systemen und langlaufenden Projekten bereits gehabt?

Int-1: Also überwiegend würde ich sagen, gerade in meinem Bereich Testintegration sind es sehr oft Kompatibilitätsprobleme. Das sehen wir auch jetzt zum Beispiel im Bereich Tests mit irgendwelchen Testtools, die einfach auf neueren Versionen nicht mehr laufen oder wo wir die Lizenzen teilweise schon gar nicht mehr bekommen zu erträglichen Preisen. Und ja, im Prinzip halt wirklich dadurch, dass das Ganze so alt ist und es auch vorgegeben ist, wie das Ganze auszusehen hat, also diese ganze Umgebung ist ja quasi vorgegeben, die wir bekommen haben, passen irgendwelche neuen Modell-Tools, die wir gerne benutzen würden, da halt nicht rein. Da muss man dann gucken, dass man Kompatibilitätsprobleme

aus dem Weg kommt.

Interviewer: Und kommt es im Projektverlauf regelmäßig zu Problemen durch die veralteten Strukturen, die du ja auch ein bisschen genannt hattest und historisch gewachsene Systeme, auch auf dem Code-Level?

Int-1: Ja, auf dem Code-Level würde ich sagen auch. Es ist natürlich so, dass auch teilweise ältere C++-Versionen oder sowas benutzen, wo neue Befehle etc. dann gar nicht unterstützt werden. Auch im Bereich Test, ist jetzt nicht direkt Softwareentwicklung, aber wenn wir jetzt über Python nachdenken – ich weiß gar nicht, welche die aktuellste Python-Version ist, wir arbeiten noch mit sehr, sehr alten Versionen –, das heißt auch da Kompatibilitätsprobleme. Es ist immer wieder das gleiche Problem, es ist alles sehr alt und es passt mit den neuen Sachen nicht mehr zusammen.

Interviewer: Und was für Auswirkungen hat das dann auf die Wartbarkeit und die Weiterentwicklung von diesem System?

Int-1: Ja, Problem ist natürlich gerade, wenn man viele ältere Dinge hat, die vielleicht auch gar nicht mehr so geschult oder gelehrt werden, wenn man jetzt neue Kollegen kriegt oder so, die kennen sich mit diesen alten Sachen halt nicht mehr so gut aus. Das heißt, eine Herausforderung ist natürlich, dass diese Know-how-Träger, die es damals mal gemacht haben, vor Ewigkeiten, dass die vielleicht auch nicht mehr da sind und das schafft natürlich eine Menge Probleme. Das hat auf jeden Fall negative Auswirkungen auf das Ganze.

Interviewer: Und was für Maßnahmen nutzt ihr, um den Softwareverfall oder technische Schulden zu vermeiden oder zu reduzieren?

Int-1: Ja, also ich muss sagen, lange Zeit, also wie gesagt, seit 15 Jahren bin ich schon da und ich habe das Gefühl, dass es lange Zeit gar nicht gemacht worden ist oder sehr wenig. Es wurde sehr schlecht dokumentiert, es wurden wenige Tests automatisiert und auch nicht regelmäßig durchgeführt. Mittlerweile nutzen wir Tools, die halt automatisch die Testkette durchlaufen lassen, dass man auch regelmäßig mal sieht, ob da irgendwie was kaputt gegangen ist oder nicht. Es gibt Tools für Codechecks und so weiter. Also es wird viel normalisiert in dem Bereich, damit man dann den Verlauf in fünf Jahren oder so immer noch gucken kann, ob es auch funktioniert.

Interviewer: Und das merkt man dann auch im Nachhinein, dass das früher nicht gemacht wurde oder ist das mittlerweile relativ gut ausgebügelt?

Int-1: Also es kommt drauf an, also je nach Projekt. Es gibt Projekte, wo es noch nicht so gut aufgebügelt worden ist. Es gibt Projekte, wo man das jetzt mittlerweile nachgeholt hat, aber man hat, glaube ich, eine ganze Menge Zeit verloren und man hätte, glaube ich, wenn man im Vorwege schon solche Dinge gemacht hätte oder auch andere Maßnahmen, hätte man sich, eine Menge Aufwand sparen können.

Interviewer: Und zum Thema Reviews und Refactoring: Macht ihr das mittlerweile auch

verstärkt, und war das früher eher weniger auch der Fall?

Int-1: Früher fast gar nicht. Also früher gab es quasi einen, der es gemacht hat und einen, der die offiziellen Tests dafür gemacht hat. Das war es. Also, so ein inoffizielles Review nochmal zwischendurch gab es gar nicht. In gewissen Projekten, aber auch nicht in allen, gibt es jetzt mittlerweile, dass es Reviews, Refactoring auch immer zwischendurch gibt. Also diesen agilen Ansatz. Aber nicht in allen Projekten.

Interviewer: Und in der Historie betrachtet, sieht man schon, dass es jetzt einen Vorteil gibt, oder merkst du da keinen Unterschied zwischen, wie es vorher lief und ob jetzt weniger technische Schuld gibt?

Int-1: Also ich glaube, es ist ein bisschen schwer zu bewerten, weil wir das noch nicht so lange machen. Meine Vermutung ist aber, dass man es auf jeden Fall merkt. Man muss natürlich auch dazu sagen, dass sowas wie Reviews und Refactoring auch wieder Zeit kosten. Also man investiert quasi ein bisschen mehr, um hinten raus hoffentlich was zu sparen. Aber ich glaube, es ist noch ein bisschen zu früh, um jetzt zu sagen, dass es so ist.

Interviewer: Und gibt es Bedingungen, wo ihr technische Schuld in Kauf nehmt und wird das dann explizit getrackt oder wird das einfach weitergetragen, wenn irgendwelche Abkürzungen oder so verwendet werden?

Int-1: Also getrackt wird es auf jeden Fall nicht. Das wüsste ich jetzt nicht, dass sich das jemand irgendwie merkt, dass da und da irgendwie was noch zu tun wäre. Es wird auf jeden Fall bewusst in Kauf genommen, das habe ich auch sehr oft erlebt. Gründe dafür sind natürlich sehr unterschiedlich. In der Regel würde ich sagen, Zeitmangel.

Interviewer: Benutzt ihr agile Methodiken, sei es Scrum oder andere? Und wenn ja, wie wirken sich diese Praktiken auf technische Schuld und Softwareverfall aus?

Int-1: Ja, also agil ist tatsächlich ein neues Thema im Bereich militärisch. Das gab es bisher gar nicht. Wir haben es jetzt, also ich war ja in unterschiedlichen Projektes unterwegs, in einem Projekt haben wir es aktiv gemacht. Ich würde sagen, dass es sich grundsätzlich sehr gut auswirkt, dass es sehr sinnvoll ist, auch wie du auch schon sagtest, was Felix sagte, dass man diese Wissensträger nicht so zentriert hat, sondern dass man das so ein bisschen verteilt. Es ist natürlich trotzdem so, dass es nach wie vor Leute gibt, die das eine besser können und welche das andere etwas besser können. Und man investiert natürlich im Vorwege mehr Zeit dafür. Es wird natürlich im Vorwege teurer, damit man hinten raus halt so ein bisschen diese technische Schuld versucht schon mal ein bisschen abzufedern. Ich glaube komplett lässt es sich trotzdem nicht vermeiden, aber zumindest, dass es besser dadurch werden wird.

Interviewer: Kannst du etwas zur Architektur sagen von den Projekten, wie die geplant, dokumentiert und wie die gepflegt wird?

Int-1: Also ich kann zumindest sagen, dass es da in unserem Bereich feste

Vorgehensmodelle dafür gibt, also auch feste Dokumente etc., die abgeliefert werden müssen. Das ist bei uns nach dem V-Modell XT. Das ist ja ein Vorgehensmodell, das dafür genutzt wird. Und ich weiß, dass wir quasi intern Strukturen haben oder intern Strukturen benutzen, die daran angelehnt sind, weil wir es halt auch machen müssen. Wir sind ja dazu verpflichtet.

Interviewer: Und das heißt, es gibt auch einen Prozess, der sichergestellt, dass die Umsetzung der Architektur richtig ist, auch wenn das Projekt schon eine lange Zeit läuft?

Int-1: Ja, ich würde sagen, es ist bei älteren Projekten so ein bisschen wahrscheinlich, dass auch hier und da mal was vernachlässigt wurde. Aber grundsätzlich gibt es diese Strukturen und es wird auch versucht, sicherzustellen, dass sie eingehalten werden. Natürlich dann eher durch Qualitätssicherung und der Kunde, der halt sagt, dass er gewisse Dinge haben möchte.

Interviewer: Hast du von Herausforderungen mitbekommen, die entstehen, wenn man die Architektur noch weiterentwickeln möchte?

Int-1: Ja, auch hier kommt wahrscheinlich das Thema Alter wieder so ein bisschen ins Spiel. Also, dass irgendwelche Architekturen oder Systeme sehr alt sind und deswegen gewisse Entscheidungen vielleicht auch im Vorwege schon limitiert sind, wie man sich treffen kann oder halt auch nicht. Ansonsten war ich da jetzt aber nicht so stark involviert.

Interviewer: Vielleicht etwas wo du mehr involviert bist: Benutzt ihr eine Form von CI/CD in euren Projekten? Und wenn ja, wie genau benutzt ihr das? Wie sehen die Prozesse dabei aus?

Int-1: Ja, also mittlerweile benutzen wir es. Das ist auch etwas, was vor einigen Jahren erst dazugekommen ist und ein bisschen angetrieben worden ist. Wir nutzen es für die automatisierten Tests. Für die Dokumentengenerierung und so und nutzen es aktuell noch nicht. Das ist ein bisschen zu komplex, was wir da haben, zumindest aktuell. Aber wir nutzen es quasi, um regelmäßig mal die Testkette durchlaufen zu lassen, um zu gucken, ob alles noch funktioniert. Auch wenn man nichts angefasst hat, sagt die Erfahrung, dass irgendwie immer wieder was schief geht. Und so hat man dann immer viel im Überblick. Weil es das ist, was in der Vergangenheit sehr stark vernachlässigt geworden ist, würde ich sagen.

Interviewer: Und kannst du sehen, dass das gegen die zuvor genannten Probleme einen Mehrwert bringt?

Int-1: Ja, auf jeden Fall. Einfach den Überblick zu haben, auf welchem Stand ist das gerade, auch wenn vielleicht irgendwas nicht funktioniert. Aber noch zu wissen, wie viel nicht funktioniert, ist es schon großer Gewinn. Weil da waren wir in der Vergangenheit sehr oft im Dunklen.

Interviewer: Und was für Werkzeuge und Metriken benutzt ihr zur Qualitätssicherung? Sei

es beim Build- oder beim Deploy-Prozess?

Int-1: Ja, hier wird viel noch auf Manual-Reviews gesetzt. Also zum Beispiel, dass unsere Qualitätssicherung sich das alles nochmal anguckt. Ansonsten viel automatisierte Testskripte. Gerade bei den Unit-Geschichten, also wenn wir auf Komponenten-Ebene gucken, es gibt ja auch System-Tests, da wird noch viel per Hand gemacht. Also es wird viel noch durchgeklickert mit Anleitungen, jetzt klicken wir darauf und dann warten wir und so weiter. Ansonsten, ja, mittlerweile nutzen wir auch ein paar Tools, sowas wie Jira oder so, ich weiß nicht, ob das damit gemeint ist, Qualitätssicherung und mit all diesen Issues und so. Sowas wie Jenkins für die automatisierten Test-Sachen und sowas. Und ja, die Erfahrungen damit sind bisher auch sehr gut, würde ich sagen.

Interviewer: Gibt es entweder Stärken oder Schwächen, die man dann gemerkt hat an der Code-Qualität, wenn man solche Werkzeuge benutzt hat, dass sich das irgendwie positiv oder negativ auf die Wartbarkeit ausgewirkt hat?

Int-1: Ja, also natürlich gerade, wenn man Jira-Issues aufmacht und die so ein bisschen verfolgt, also irgendwelche Fehler einspielt und dann mit Kommentaren, wie man das behoben hat und welche Fehler und dass das einer nochmal nachtestet und so. Das ist natürlich sehr, sehr hilfreich, also dass man noch mal gucken kann, was wurde eigentlich geändert. Auch Dinge, wo man dann auch sowas sieht, ich merke es jetzt gerade bei mir, dass irgendwelche Dinge behoben worden sind mit Quellcode, aber in den Testskripten nicht und dann müssen die Testskripte ein paar Jahre später nachgezogen werden. Das ist wahrscheinlich so das Paradebeispiel für technische Schuld und wenn man dann nicht nachvollziehen kann, was im Quellcode geändert worden ist, außer sich vielleicht angucken, welche Zeit sich geändert hat, aber vielleicht nicht warum und wie das funktioniert, dann ist das schon sehr lästig.

Interviewer: Zu Code-Reviews: Wie wird damit bei euch gearbeitet und wie sieht das aus, wer beteiligt sich und wie wird der Prozess organisiert?

Int-1: Ja, also gerade bei Code-Reviews gibt es ja im Prinzip zwei Möglichkeiten: Es gibt einmal das Code-Review, dass man sich mit einem anderen Entwickler zusammen hinsetzt, also das wäre jetzt im agilen Umfeld zum Beispiel, dass man sich mit einem anderen Kollegen hinsetzt aus diesem agilen Team, der jetzt vielleicht auch gar nicht viel Entwicklungserfahrung hat, aber zumindest einfach mal drüber guckt. Manchmal kann das auch schon hilfreich sein. Ansonsten gibt es dann noch die automatisierte statische Code-Analyse. Da gibt es ja auch Tools für. Die werden bei uns auch in Jenkins quasi automatisiert gemacht. Da wird dann geguckt, ob diese Code-Richtlinien eingehalten werden. Was im Endeffekt ja auch darauf basiert, dass man den Quellcode qualitätssichern möchte.

Interviewer: Und wie war das in der Historie, wurde das benutzt? Oder ist das alles quasi

in den letzten paar Jahren implementiert worden?

Int-1: Also gerade dieses, ich sag mal, gemeinsam drauf gucken, das gab es früher eher so: „kannst da kurz mal drüber gucken“, aber nicht so formal. Also so wie jetzt, dass es halt wirklich vorgesehen ist und auch abgehakt wird, als eigenes Issue. Automatisiert ist es, meine ich, nie passiert, regelmäßig. Die Tools selber gab es schon. Das wurde dann aber, wenn die Software geliefert worden ist, mal gemacht. Weil es dann halt mitgeliefert werden muss, dieses Zertifikat, dass es gesichert worden ist. Aber ansonsten wurde es nicht zwischendurch mal gemacht, um zu gucken, ob noch alles gut ist. Hätte man vielleicht machen müssen.

Interviewer: Und was für eine Rolle spielen Tests? Also du meinst ja schon, dass es die Automatisierung und die Unit-Tests gibt. Aber wie sieht das in den Projekten aus?

Int-1: Ja, also ich sagte ja schon, also überwiegend nutzen wir eigentlich automatisierte Testskripte. Und das halte ich auch für sehr sinnvoll, gerade weil man die halt automatisiert über Nacht laufen lassen kann. Und das ist aus meiner Sicht auch ein sehr gutes Mittel, gegen diese technische Schuld so ein bisschen, gerade über lange Sicht. Also ich habe es jetzt gesehen, bei gewissen Projekten war es halt so, da hat man die Testskripte ewig nicht angefasst. Aber hier und da haben wir so kleine Änderungen gemacht, aber dies dürften wir eigentlich nicht. So, wie das immer so ist. Das dürfte eigentlich nichts kaputt machen. Und jetzt habe ich gefühlt wieder ein paar Monate gebraucht, um die alle auf Stand zu kriegen. Und das ist halt genau diese technische Schuld, die wir quasi im Vorweg hätten verhindern können, gerade wenn es dann Quellcode-Änderungen gibt, dann die Test-Skripte einfach nochmal direkt durchzufahren, vielleicht auch direkt anzupassen und das nicht erst ein Jahr später zu machen.

Interviewer: Also das heißt, die Test-Skripte werden dann getrennt von dir quasi aktualisiert und der Entwickler macht damit quasi gar nichts direkt. Also er ändert den Code, aber dass die Tests laufen, ist nicht sein Auftrag.

Int-1: Ja, so sollte es vielleicht sein, ist es vielleicht auch, aber das ist oft nicht gemacht worden, sagen wir es mal. Also das ist bei uns sehr unterschiedlich, manchmal macht es der Entwickler, es gibt ja durchaus auch Komponenten, wo die Entwickler bei den Tests mitgearbeitet haben, oder manchmal ist es halt auch eine komplett externe Person. In meinem Fall war es immer mal gemixt.

Interviewer: Und spürst du die Anforderungen an Sicherheit und Zuverlässigkeit und auch die regulatorischen Vorgaben stark bei deiner Arbeit, sodass sie sich darauf auswirken?

Int-1: Ja, teilweise schon. Also es kommt noch ein bisschen darauf an. Ich arbeite ja jetzt eher im Test-Integrationsbereich. Da ist es natürlich so, dass ich viel gegen die Anforderungen arbeite. Das heißt, je besser die Anforderungen sind, desto genauer müssen ja auch die Tests sein. In unserem Umfeld sind sie teilweise nicht so genau, dass

die Tests dann vielleicht auch nicht immer ganz so zuverlässig und deterministisch sind, wie sie vielleicht sein sollten. Ich glaube aber, es gibt durchaus Projekte, wo ich jetzt aber nicht ganz so beteiligt war, wo im Bereich der Softwareentwicklung, dadurch, dass Programmiersprachen wie Ada oder sowas genutzt werden, wo man das dann schon sehr stark zu spürt.

Interviewer: Hast du eine Idee, wie die Beschaffungsprozesse und die langen Freigabeschleifen, die existieren, sich auf die Projekte auswirken? Zum Thema technische Schuld und Softwareverfahren?

Int-1: Was ist genau gemeint mit Beschaffungsprozesse und Freigabeschleifen?

Interviewer: Felix meinte zum Beispiel, dass es häufig, wenn man dann irgendwelche Änderungen machen möchte, erstmal wieder zurück zum Kunden gehen muss. Und dass man halt nicht sofort Änderungen einführen kann, die man eigentlich gerne machen würde.

Int-1: Ja, das merkt man doch sehr stark bei uns. Dadurch das wir sehr viele formale Geschichten haben. Also dieser ganze formale Abnahmeprozess ist bei uns deutlich komplexer als bei anderen, würde ich sagen. Also gerade auch, was wahrscheinlich mit Freigabeschleifen gemeint ist, da haben wir natürlich ganz andere Standards als wahrscheinlich welche, die nicht militärische Software machen. So ist es dann so, dass die Softwareänderungen manchmal 10 Minuten dauern, aber der Prozess mit Testen und Nachweis und Lieferung und Freigabe dann durchaus mal eine Woche in Anspruch nimmt.

Interviewer: Und das heißt, es ist auch häufig so, dass Sachen nach hinten geschoben werden, wenn sie vielleicht nicht so dringend sind, dass man sagt, machen wir uns jetzt nicht den Aufwand, sodass dann halt viel irgendwie einfach liegen bleibt?

Int-1: Ja, tatsächlich. Das ist glaube ich auch das Problem, was ich vorhin auch angesprochen hatte, dass dann teilweise Testskripte rumliegen und das muss dann ganz schnell gehen mit der Softwarelieferung und dann machen wir später irgendwann mal die Testskripte. Dann stellt man ein paar Monate später fest, die laufen ja gar nicht und keiner weiß warum.

Interviewer: Kannst du ein Beispiel nennen von einer Maßnahme zur Reduktion von technischer Schuld oder zur Vermeidung von Softwareverfall, die für dich sehr erfolgreich war?

Int-1: Ja, da würde ich die Automatisierung der Testskripte tatsächlich nennen. Dadurch, dass man sie jetzt nicht mehr per Hand anstoßen muss, die über Nacht laufen können und ich quasi ein komplett gesammeltes Ergebnis über alle Komponenten hinweg sehen kann, ist das schon ein großer Gewinn. Also vorher mussten wir die per Hand starten. Wir mussten die Testlogs extra auswerten und so weiter. Das passiert jetzt alles vollautomatisiert. Zu Zeiten, wo niemand im Büro sein muss, das ist auf jeden Fall ein großer Gewinn.

Interviewer: Gibt es weitere Erfahrungen oder Empfehlungen, die du im Zusammenhang mit Softwareverfall oder technischer Schuld gerne teilen möchtest?

Int-1: Ja, also aus meiner Erfahrung: Testskripte immer direkt nach der Softwareanpassung mit anpassen. Weil selbst wenn man der Entwickler ist und das selber gemacht hat vor zwei Monaten, auch dann kann es schwierig werden, die Testskripte nochmal anzupassen. Also wenn man im Thema drin ist, dann wirklich alles einmal gerade ziehen. Am besten auch Dokumentation, wenn die irgendwo da ist, den Quellcode dokumentieren. Also Kommentare schreiben ist auch sehr, sehr hilfreich, sehr wichtig. Sowohl in der richtigen Software als auch an den Testskripten. Das darf man auch nicht unterschätzen. Dadurch kann man, glaube ich, eine ganze Menge Softwareverfall und auch gerade technische Schuld irgendwie in den Griff kriegen. Und man tut der nächsten Generation auch einen großen Gefallen, wenn man das einigermaßen lesbar und gut dokumentiert hinterlässt.

Interview 2

Interviewer: Erstmal zum Einstieg generell: Was für eine Rolle hast du im Bereich der Softwareentwicklung von militärischen Systemen inne, seit wann bist du in dem Umfeld tätig und was ist so die typische Projektarbeit für dich?

Int-2: Also prinzipiell, ich bin seit über 20 Jahren hier im Bereich tätig. Ich habe dementsprechend auch öfters gewechselt. Derzeit bin ich Product Owner für ein Teilsegment oder ein Teilsystem, das wir pflegen und warten, das eben auch schon seit 20 Jahren in Betrieb ist. Ich habe aber in der Zeit vorher auch als Entwickler, Tester, Qualitätssicherer, Testmanager, also alles Mögliche mal gearbeitet.

Interviewer: Und was ist so deine typische Projektarbeit derzeit?

Int-2: Es gibt jetzt zwei Projekte, an denen ich hauptsächlich arbeite. Das eine ist eine Softwareänderung, an der wir gerade arbeiten. Also ich bin in beiden Fällen eben Product Owner. Das heißt für mich eben hauptsächlich, die Arbeitspakete sinnvoll zu verteilen und zu schneiden und eben auch mit unseren ARGE-Partnern, also mit der Thales und der Atlas zusammen abzustimmen, wie unsere Anforderungen eben zu interpretieren und umzusetzen sind. Und gleichzeitig versuche ich eben auch als Entwickler und Qualitätssicherer nochmal einen Blick drauf zu werfen, was gemacht wird und in letzter Instanz eben auch zu kontrollieren, dass das vernünftig gemacht wird, was am Ende dann auch geliefert werden soll.

Interviewer: Und an was für Arten an Software-Systemen arbeitest du derzeit hauptsächlich? Also eingebettete Systeme, einsatzkritische Systeme oder Verwaltungssoftware oder sowas in die Richtung?

Int-2: Wir haben hauptsächlich ein verteiltes System, in dem eben Prozesse auf diversen Konsolen laufen, die sich dann über Producer-Consumer-Listen oder Publisher-Subscriber eben anmelden können, welche Nachrichten sie kriegen und schreiben wollen und dann darüber Informationen austauschen mit anderen Systemen.

Interviewer: Du hast ja schon gesagt, dass die Anwendung 20 Jahre alt ist. Was für Herausforderungen begegnen dir dann im Umgang von solchen Altsystemen oder allgemein langlaufenden Projekten?

Int-2: Ja, das hat oft damit zu tun, dass Leute das Projekt irgendwann verlassen und neue Leute dazu kommen. Aber auch wenn du selbst Quellcode geschrieben hast, ihn dann zehn Jahre später noch warten und verstehen zu müssen, ist eben eine Herausforderung. Das lehrt einen, zurückhaltend zu sein, was komplexe Softwarestrukturen angeht und möglichst den Code lieber ein bisschen einfacher zu halten, besser zu kommentieren und eben dafür zu sorgen, dass jeder den Code später auch noch pflegen kann. Dazu kommt eben, dass wir viele ältere Technologien auch benutzen müssen, weil es vorgegeben ist durch das Projekt und die meisten Leute, die neu in unsere Umfeld der hineinkommen, dass die eben keinerlei Erfahrung mit diesen Technologien haben. Deswegen ist es auch da sinnvoll, die Software so zu schreiben, dass eben auch jemand, der noch keine Erfahrung mit diesen ganzen Systemen hat, da eben trotzdem gut reinkommen kann. Also dementsprechend ist ein hoher Maßstab an die Qualitätssicherung zu setzen, auch an die Code-Inspektion, dass der Code gut gepflegt ist, verständlich ist und dadurch eben auch langfristig am Leben gehalten werden kann.

Interviewer: Das war dann auch ein Wandel mit der Zeit oder war das schon immer?

Int-2: Nein, das war tatsächlich schon immer so. Also wir haben im Projekt auch feste Codierrichtlinien, wo genau solche Sachen festgeschrieben sind. Also vielleicht nicht in diesem Detailgrad, aber wir haben diese Richtlinien, die eben auch dafür gemacht sind, den Code am Ende wartbar zu halten.

Interviewer: Und kam es im Projektverlauf regelmäßig zu Problemen durch die veralteten Strukturen oder historisch gewachsenen Systeme oder läuft das eigentlich relativ glatt?

Int-2: Eigentlich läuft es relativ glatt. Wir haben eher hauptsächlich den Wechsel der Mitarbeiter, der immer wieder stark ins Gewicht fällt. Dass natürlich immer wieder Leute in Rente gehen und dementsprechend sich immer wieder neue Leute einarbeiten müssen und dadurch immer wieder so ein Umdenken stattfinden muss. Wir hatten es früher sehr stark, dass wir eine Person hatten, die sich jeweils um ein einziges Segment gekümmert hat und die eben auch die Hauptansprechperson dafür war. Davon versuche ich jetzt eben auch wegzukommen, indem wir das Wissen eben auch mehr verteilen, weil es völlig klar ist, dass nicht alle Leute bis zum Ende ihrer Laufbahn hier immer nur in einem Softwareprojekt arbeiten werden.

Interviewer: Und was für Auswirkungen haben lange Lebenszyklen auf Wartbarkeit und Weiterentwicklung deiner Erfahrung nach?

Int-2: Prinzipiell ist es, wie schon gesagt, eben sehr wichtig, dass von Anfang an darauf geplant wird, dass die Software eben eine lange Laufbahn hat und dass dementsprechend sich jedes bisschen, was man in bessere Wartbarkeit steckt, langfristig rentiert. Normalerweise ist es ja so, dass Quellcode nach ein paar Jahren eben einfach nicht mehr benutzt wird und neu geschrieben werden kann. Das ist bei uns eben auch durch diverse Rahmenbedingungen eben nicht so weit möglich. Dadurch, dass wir auch sehr komplizierte Prozesse haben, kann die Software nicht einfach geändert werden, sondern wir müssen einen recht komplexen Prozess durchlaufen, bevor Software dann geändert, verbessert werden kann. Dementsprechend ist es auch dadurch eben sehr wichtig, dass wir es schaffen, die Software vernünftig zu schreiben, gleich von Anfang an.

Interviewer: Und was für Maßnahmen benutzt ihr, um Softwareverfall und technische Schuld zu vermeiden oder zu reduzieren? Also Refactoring, Reviews, irgendwelche Automatisierungen?

Int-2: Alles davon. Also prinzipiell benutzen wir immer das Vier-Augen-Prinzip. Neuerdings, also früher war es noch nicht so, aber das habe ich in den letzten Projekten eingeführt, dass wir nach Scrum arbeiten und dabei dann eben auch eingeführt haben, dass alles nochmal gereviewt wird: jede Zeile Code, jedes Dokument, das geschrieben wird. Auch früher hatten wir die Vorgabe für den Quellcode, der am Ende dann an den Kunden geliefert wird, dass immer eine Code-Inspection gemacht werden musste. Dafür wurden auch regelmäßige Reports angelegt, die auch auf Nachfrage dem Kunden vorzulegen sind. Von daher, da wurde schon sehr viel darauf geachtet, aber das wird jetzt eben noch deutlich stärker gemacht. Und wenn das dann nötig ist, kann es eben auch im Review rauskommen, dass wir Codestellen nochmal schreiben, Refactoren durchführen. Dies, wenn eben rauskommt, dass manche Teile so einfach nicht pflegbar sind oder nicht vernünftig laufen könnten. Aus dem dritten Punkt, Review, Refactoring und Automatisierung. Genau, also wir haben eben auch sehr viel Arbeit in die Automatisierung von Tests gesteckt, also bei Build und Testautomatisierung haben wir sehr viel gemacht. Einerseits, um eben sicherzustellen, dass keine Seiteneffekte sich einschleichen in die Software, aber andererseits, weil es eben auch eine Art von Wissenskonservierung ist. Also, dass eben langfristig dadurch sichergestellt werden kann, dass kein Wissen verloren geht, weil eben alles in diese Prozesse schon eingebrannt ist und dass jederzeit dadurch nachgeprüft werden kann, wie diese Prozesse funktionieren und es eben auch von jedem nachvollzogen werden kann.

Interviewer: Das heißt, du siehst diese Maßnahmen als sinnvoll?

Int-2: Ja, definitiv.

Interviewer: Gibt es Momente, wo ihr bewusst technische Schuld in Kauf nehmt?

Int-2: Prinzipiell klar, also, was heißt technische Schuld? Also, prinzipiell, jeder Quellcode kann natürlich immer noch weiter verbessert werden. Es ist immer ein Abwägen zwischen, die Software muss fertig werden, aber sie soll eben auch gut sein am Ende, also möglichst fehlerarm und pflegbar, wartbar, all das soll sie eben erfüllen. Aber andererseits haben wir auch die Deadlines zu der die Software fertig sein muss. Und die Kosten würden natürlich auch explodieren, wenn wir noch mehr Refactorn oder noch mehr Arbeit reinstecken, um den Quellcode zu optimieren. Von Daher ist es immer eine Abwägung, und natürlich ist es gerade eher gegen Ende der Laufzeit, dass dann doch mal eher in Kauf genommen wird, wenn Software nicht optimal ist, solange eben keine Fehler drin sind. Das ist natürlich auch eine Abwägung. Es geht ja jetzt nur um technische Schulden, sprich, oder was heißt nur, also es geht eben mehr darum, natürlich könnte man den Quellcode noch verbessern an anderen Stellen, aber solange die Software keine Fehler drin hat, von denen wir wissen, kann sie dann natürlich durch die Teststufen durch und eben auch geliefert werden am Ende.

Interviewer: Und wenn die dann die Schulden kaufen, werden die dann getrackt oder sind die dann quasi, wenn das Projekt abgegeben wird, weg?

Int-2: Das hängt davon ab, was das jetzt für Schulden sind. Also wenn tatsächlich nach dem Test zum Beispiel noch einzelne Fehler auftreten, werden die getrackt. Wenn es jetzt nur um Code-Verbesserungen geht, haben wir teilweise in Jira-Stories dafür mal angelegt oder jetzt in GitLab die dann auch hinterlegt. Aber im großen Ganzen, ja, weil es eben auch so aufwendig ist, solche Sachen dann später nochmal zu verbessern. Wenn es dann wirklich nur um nicht optimale Implementierung geht, dann wurde es dann meistens doch eher unter den Teppich gekehrt.

Interviewer: Du hast es hier gerade angesprochen, dass ihr Scrum verwendet. Und weil das ja erst vor Kurzem implementiert wurde: Merkst du einen Unterschied zwischen den Praktiken zum Thema technische Schuld und Softwareverfall?

Int-2: Es ist schwierig, das an Scrum festzumachen, weil generell eben auch ein starker Umbruch hier in den letzten Jahren stattgefunden hat. Also, ich habe Anfang 2000 hier angefangen, und da war es eben noch ein etwas älterer Bereich. Viele der Kollegen waren schon deutlich älter als ich. Die haben das dann über Jahrzehnte gemacht und eben auch fortgeführt. Viele von diesen Leuten, die das eben jahrzehntelang gemacht haben, sind vor fünf, sechs Jahren nach und nach in Rente gegangen, sodass sowieso jetzt ein Umdenken stattgefunden hat. Dadurch haben wir eben auch viel stärker lernen müssen, dass eben nicht eine Person immer das Gleiche machen kann, sondern dass immer neue Leute reinkommen. Wir haben viele junge Studenten dazu bekommen oder Leute, die ihr Studium abgeschlossen haben, die jetzt zu uns gewechselt sind. Allein dadurch ist es immer wieder vorgekommen, dass wir da umgedacht und verbessert haben, unser ganzes

Verhalten. Und ich fand, auch da hat Scrum eben sehr stark geholfen, dass neue Mitglieder im Team leichter an Bord geholt werden konnten, weil eben weniger diese Keller-Mentalität gelebt wird – dieses ich nehme ein Software-Paket mit, bastle ein halbes Jahr im Keller und komme dann mit dem fertigen Paket wieder. Indem man sich eben viel mehr untereinander austauscht und dementsprechend eben auch viel mehr mit anderen Leuten redet, können neue Leute besser an Bord geholt werden. Und jeder hat besser die Möglichkeit, überall mal reinzuschnuppern und im besten Fall nach ein paar Monaten eben alles von dem Projekt gelernt hat, was es zu lernen gibt.

Interviewer: Aber das heißt, ihr benutzt auch Scrum, um Refactoring und Schuldenabbau quasi effektiv zu managen. Habt ihr dann quasi spezifische Tickets, wo ihr dann Refactoring mit einplant, oder wie sieht das aus?

Int-2: Ja, es ist eine Art von Review eingeplant und ein Teil des Reviews kann es natürlich auch sein, dass dann hinterher beschlossen wird: So das war Mist, den Quellcode müssen wir wegschmeißen und neu schreiben. Ist gerade erst passiert bei einer unserer Testprozeduren, also kann immer wieder passieren, wir haben jetzt keine festen Refactoring-Tickets, aber prinzipiell ist es jedem freigestellt, eigene Tickets zu erstellen. Wenn also ein Entwickler jetzt feststellt, dass manche Teile dringend überarbeitet werden müssen, kann er dafür ein Ticket erstellen, aber auch da gibt es wieder das Problem, dass wir eben in den zu liefernden Quellcode nicht einfach ohne Verfahren eingreifen dürfen.

Interviewer: Und ein kleiner Sprung zur Architektur. Wie wird die geplant, dokumentiert und gepflegt?

Int-2: Genau, zu dem ganzen Block werde ich dir nicht zu viel erzählen können, weil die wurde eben ursprünglich Ende der 90er Jahre entworfen, also sogar noch vor meiner Zeit. Und die ist mit den ARGE-Partnern, also unseren Arbeitskollegen aus den anderen Firmen, zusammen abgestimmt. Und an dieser Architektur orientieren wir uns nach wie vor. Das heißt, es werden immer mal neue Prozesse eingeführt oder einzelne Blöcke ausgetauscht, aber die Architektur ist seitdem unverändert und kann eben auch nicht ohne weiteres verändert werden. Wir haben Diagramme, die dafür gepflegt werden, in denen diese Architektur festgehalten ist und nach denen leben wir.

Interviewer: Und gibt es einen Prozess, der sichergestellt, dass die Umsetzungen sich an der Architektur orientieren?

Int-2: Diese Prozesse sind eher technischer Natur. Wir haben Schnittstellen zu den ARGE-Partnern. Das bedeutet, wir müssen genau definieren wie unsere Prozesse funktionieren, was sie verschicken oder eben verarbeiten. Und dementsprechend ist es für uns sowieso schon schwierig, die Architektur jetzt grundlegend zu verändern, weil wir das alles immer mit den ARGE-Partnern abstimmen müssten. Es gibt immer einzelne Softwarepakete, die auf die Firmen verteilt sind und diese Pakete werden dann von dieser

jeweiligen Firma verarbeitet und die Schnittstellen nach außen sind eben fest. Die können nicht ohne weiteres geändert werden.

Interviewer: Und weißt du von irgendwelchen Herausforderungen, die bei der Weiterentwicklung von der Architektur aufgetreten sind?

Int-2: Nö, da wüsste ich nichts zu.

Interviewer: Und jetzt zum Thema Automatisierung, die du ja auch schon angesprochen hattest. Benutzt ihr eine Form von CI/CD? Und wenn ja, wie sehen die Prozesse dabei aus? Und hilft CI/CD merklich gegen die zuvor genannten Probleme?

Int-2: Also wir benutzen GitLab neuerdings. Hatten vorher Jenkins benutzt, sind jetzt aber auf GitLab umgestiegen. Vorweggenommen, also ich finde, das hilft enorm, dadurch, dass viele Prozesse, die vorher sonst per Hand gemacht werden mussten, immer sehr viel Expertise erfordert haben, oder dann doch mal ausgelassen wurden, wenn die Zeit knapp war. Und so ist man quasi gezwungen, das einmal umzusetzen, zu implementieren, diese ganze CI/CD-Geschichte. Und von daher, wir haben jetzt Pipelines, die bei jeder Änderung durchlaufen oder auch über Nacht angeschmissen werden, sodass wir jederzeit erkennen können, wenn Sachen eben nicht mehr funktionieren, die vorher alle per Hand gemacht werden mussten. Also vor zehn Jahren war es eben noch so, dass viele von diesen Testprozessen zum Beispiel eben per Hand durchgeklickt werden mussten, was dann zwei bis drei Tage pro Prozess dauern konnte. Was jetzt eben alles vollautomatisch innerhalb von, ich sage mal, der Längstläufer, den wir haben, ich will nicht zu sehr ins Detail gehen, aber es funktioniert eben automatisch im Hintergrund. Von daher muss man selber da keine Arbeit mehr reinstecken. Und vor allem, man kann es nicht auslassen. Also es wird ja automatisch gemacht, sodass man automatisch einen auf die Finger bekommt, wenn sich da irgendwelche Sachen eingeschlichen haben.

Interviewer: Und das ist jetzt eine Veränderung, die in den letzten paar Jahren kam?

Int-2: Also Jenkins haben wir schon länger am Laufen, aber wir haben es eben nach und nach immer weitergetrieben. Also wir haben die CI/CD-Pipeline immer mehr erweitert. Ursprünglich war es eben eine Build Pipeline, wir haben aber viel Zeit in die Testautomatisierung gesteckt. Wir haben noch andere Tools mit in die Pipeline aufgenommen, die auch die Codequalität prüfen. Die Build Pipeline wird eben auch ausgewertet, ob durch den Compiler irgendwelche Warnings ausgeworfen werden und sowas. Von daher, die Pipeline wächst immer weiter und dadurch wird immer weiter automatisiert und das Arbeiten wird immer praktischer und bequemer auch für die Kollegen.

Interviewer: Du hast es ja gerade schon angesprochen und vielleicht musst du nicht alles erwähnen, aber was für Werkzeuge und Metriken verwendet ihr für die Qualitätssicherung? Und was sind deine Erfahrungen mit diesen Werkzeugen bezüglich der Codequalität?

Int-2: Genau, wir benutzen Logiscope, nennt sich das Tool. Das ist ein Tool, das eigene

Metriken hinterlegt hat, das aber auch erweitert werden kann um neue Metriken. Und die Grenzwerte für diese Metriken, die sind innerhalb der ARGE festgelegt worden, zusammen mit dem ÖAG, also mit unserem Auftraggeber. Und diese Richtlinien sind eben auch einem Prozess unterworfen, durch den die definiert werden. Das ist teilweise eben auch Teil der Schwierigkeit, diese Metriken genau festzulegen, ich weiß nicht, inwiefern du auf Metriken und sowas eingehst in deiner Arbeit. Das Problem ist eben, dass ja immer Grenzwerte geprüft werden, ob diese verletzt werden oder nicht. Also zum Beispiel, ob eine Methode zu komplex ist vom Aufbau. Also zu kompliziert, unverständlich dadurch am Ende. Das wird ja auch anhand von festen Parametern geprüft. Und diese Parameter können eben festgelegt werden. Die sind noch ein bisschen im Wandel, weil wir da eben selbst noch in der Findungsphase sind, was da vernünftige Werte sind, weil sich teilweise die Metriken eben auch widersprechen. Also das ist eben auch ein bisschen so, man muss Erfahrung mit reinbringen aus den Projekten, was dann jetzt am Ende vernünftige Werte dafür sind. Aber trotzdem ist es eben wichtig, dass am Ende dann nochmal jemand drauf guckt, weil es immer Verstöße gegen Metriken geben wird und die dann am Ende bewertet, ob sie an diesen Stellen dann gerechtfertigt sind oder ob sie ausgearbeitet werden sollten. Aber trotzdem finde ich, es ist enorm hilfreich, dass man auf diese Metriken erstmal hingewiesen wird, automatisiert durch die Software. Im besten Fall arbeiten wir eben auch daran, das auch in die Pipeline hineinzubringen, damit die Pipeline das dann eben auch direkt macht, aber dass dann eben auch so ein aktives Nachdenken darüber stattfindet. Als Entwickler neigt man doch eher dazu, den schnellen Weg zu nehmen, einfach eine Prozedur runterzuschreiben, wie man sich das vorstellt und wenn dann hinterher eine Metrik dann nochmal sagt, oh, die ist aber nicht gut strukturiert, ist das sehr hilfreich, wenn man das eben schon beim Schreiben dieser Prozedur berücksichtigt.

Interviewer: Habt ihr bemerkt, dass sich dieses Achten auf die Qualität und Wartbarkeit ausgewirkt hat?

Int-2: Definitiv. Also kurz zum Beispiel, viele Methoden sind eben kürzer geworden, leichter lesbar und sowas. Das es eben auch einem Laien dann leichter fällt, der neu ins Projekt reinkommt, zu verstehen, was da jeweils passiert.

Interviewer: Du hattest ja schon angesprochen, dass ihr jetzt sehr stark auf Code Reviews achtet. Wer beteiligt sich und wie wird das Ganze organisiert?

Int-2: Prinzipiell, da wir GitLab benutzen, funktionieren viele Teile dieser Code Reviews einfach online. Da können sich die Entwickler nun selber darunter absprechen. Also wie gesagt, jede Änderung wird eben nochmal gereviewt. Und das hängt so ein bisschen davon ab, was für eine Art von Review da notwendig ist. Also gerade bei unseren Test-Skripten sind wir natürlich relativ frei. Die verlassen unser Haus nicht. Da ist unser eigener Anspruch das einzig Entscheidende. Und eben, dass unsere Qualitätssicherungen am

Ende sagt, unsere Tests sind ausreichend gewesen. Ein etwas höherer Maßstab muss eben bei Software angelegt werden, die wir an den Kunden liefern, die am Ende dann auch in Betrieb gehen sollen. Und da sind noch ein bisschen höhere Maßstäbe dran, wie das dokumentiert wird. Und am Ende dann eben auch nachvollziehbar gezeigt werden kann, dass solche Reviews stattgefunden haben, was die Ergebnisse waren. Das wird jetzt alles auch in GitLab dokumentiert, sodass es auch Jahre später dann noch nachvollzogen werden kann.

Interviewer: Zum Thema Tests hattest du schon gesagt, dass ihr auch auf Automatisierung umgestiegen seid. Haben bisher diese Tests auch geholfen, technische Schulden zu vermeiden und Softwareverhältnisse zu verhindern?

Int-2: Das ist schwierig, weil wir die automatisierten Tests eben nach und nach eingeführt haben. Und wie gesagt, es ist eben schwierig, im Nachhinein die Software zu ändern, wenn wir keinen Auftrag haben, dafür genau diesen Quellcode anzupassen. Deswegen sind wir etwas eingeschränkt mit den Möglichkeiten. Trotzdem haben wir die Möglichkeit eben genutzt, wenn Software sowieso geändert werden sollte, dann haben wir da eben auch Verbesserungen eingeführt. Trotzdem waren unsere Tests auch vorher schon sehr gut, muss ich sagen. Es ist hauptsächlich der Komfort, der dadurch gestiegen ist und dadurch eben auch die Möglichkeit, die Tests regelmäßiger und öfter durchzuführen. Das Problem war selten, dass die Software nicht gut funktioniert hat, sondern meistens, dass die Tests dann eher nicht oft genug durchgefahren wurden und deswegen noch Probleme in den Tests vielleicht drin waren. Und die Qualität unserer Tests ist jetzt einfach deutlich gestiegen und sie werden einfach viel öfter durchgefahren, wodurch einfach viel früher auftaucht, wenn Probleme in der Software enthalten sind.

Interviewer: Haben die Anforderungen an Sicherheit, Zuverlässigkeit und auch regulatorische Vorgaben einen Einfluss auf eure Arbeit?

Int-2: Sehr großen Einfluss, je nachdem, was für Software wir da gerade pflegen. Es gibt verschiedene Einstufungen. Je kritischer die Software, desto höher ist eben auch der Einfluss, den es auf unsere Arbeit hat, weil desto höher ist eben auch der Testaufwand, den wir zum Beispiel dann betreiben müssen oder desto höher sind die Ansprüche an die Software, die dann die Software auch befriedigen muss.

Interviewer: Wie beeinflussen die Beschaffungsprozesse und die Freigabeschleifen das Projekt in Bezug auf technische Schulden und Softwareverfall?

Int-2: Wären die Prozesse einfacher, wäre die Software wahrscheinlich besser. So ist es doch oft das Problem, wie du schon meintest vorher, dass wir irgendwelche Fehler feststellen, oder nicht Fehler, Fehler ist noch eine andere Sache, aber das wir Sachen in der Software feststellen wie toten Code, wo man normalerweise sagen würde: ok, schmeißen wir raus. Code, der nicht erreicht werden kann, sollte rausgeschmissen werden. Weil die

Prozesse ebenso aufwendig sind, wird das dann eben normalerweise nicht gemacht.

Interviewer: Also kommt das auch von der Kundenseite her, dass das quasi ignoriert werden sollte, oder kommt das von eurer Seite, dass man sagt, das ist es nicht wert?

Int-2: Beides so ein bisschen. Also die Prozesse sind ja mit den Kunden abgestimmt oder sind auf Wunsch des Kunden. Und die Prozesse haben auch durchaus ihren Sinn. Weil unsere Software so kritisch ist, ist es eben durchaus richtig, dass nicht jeder einfach so in der Software irgendwie rumfuscht und dann vielleicht nicht gut genug getestete Software am Ende produktiv geht. Von daher ist es sehr, sehr wichtig, dass wir diese Prozesse haben, aber andererseits führt es natürlich genau zu diesem besprochenen Problem.

Interviewer: Hast du ein Beispiel, in dem eine Maßnahme zur Reduktion von technischer Schuld oder Verbreitung von Softwareverfall besonders erfolgreich war, deiner Meinung nach?

Int-2: Also wir hatten jetzt gerade eben, hatte ich vorhin schon angesprochen, einen Testfall, der eben nicht gut war, also schlecht zu lesen. Aber der damalige Kollege arbeitet hier nicht mehr. Von daher musste jemand Neues diesen Quellcode eben pflegen, wo wir genau in das Problem reinlaufen von schlecht wartbarem Code. Es kommt immer wieder vor, dass andere Leute den Quellcode pflegen müssen als die, die ihn vorher geschrieben haben. Und da haben wir, nachdem sich der Kollege da zwei Wochen mit dem Quellcode beschäftigt hat, beschlossen, wir schmeißen ihn weg und schreiben ihn komplett neu. Weil in diesem, also es war ein Test-Skript, von daher kein echter Quellcode, der geliefert wurde. Von daher haben wir eben auch die Freiheit einfach zu sagen, wir werfen ihn weg und machen ihn neu. Dadurch ist er jetzt viel besser, viel wartbarer, viel lesbarer geworden, viel kompakter und hat am Ende auch keinen Fehler mehr geworfen.

Interviewer: Und gibt es noch weitere Erfahrungen oder Empfehlungen, die du im Zusammenhang mit Softwareverfall und technischer Schuld teilen möchtest?

Int-2: Außer nochmal eben die Aussage, Wartbarkeit ist eben sehr, sehr wichtig, langfristig, mehr wüsste ich jetzt nicht.

Interview 3

Interviewer: Zum Einstieg: Was ist denn eine aktuelle Rolle im Bereich der Softwareentwicklung und wie lange bist du in dem Umfeld tätig?

Int-3: Ja, also ich mache jetzt für eine Komponente der F124, die jetzt gerade weiterentwickelt wird, an ein paar Stellen kleinere Änderungen. Da mache ich die Automatisierung in Pipeline, in GitLab, gucke, dass die Oberflächentests automatisiert durchlaufen in einen Container. Dafür habe ich den Container gebaut und dafür gesorgt,

dass die Automatisierung da vorangeht, damit wir immer einen getesteten Zustand haben, weil wir in der Vergangenheit auch oft gemerkt haben, wenn man Test lange nicht macht, laufen sie dann irgendwann nicht mehr und man wusste nicht warum. In Zukunft sollen sich bei jedem Commit, muss also alles getestet werden und nächtlich auch jede Komponente. Daran arbeite ich und ja, nebenbei noch eine Automatisierung der Dokumentation. Das ist mein Schwerpunkt aktuell.

Interviewer: Und wenn du von diesen Systemen redest, welcher Art sind die hauptsächlich? Sind das eingebettete Systeme oder irgendwelche Legacy-Systeme oder Verwaltungssoftware?

Int-3: Ja, also die Software, die da eingesetzt wird, die ist teilweise schon, ich glaube, von 1999 oder so, also die ist schon länger im Einsatz. Ich würde es nicht als Embedded-Systeme betrachten, aber es sind halt wie so, ja, geht schon so ein bisschen Richtung Microservices, wo man halt kleine Prozesse hat, die über eine Infrastruktur, eine Kommunikationsinfrastruktur mit anderen Prozessen kommunizieren können, indem sie Pakete schreiben, Pakete empfangen und das halt teilweise auf Oberflächen darstellen können.

Interviewer: Und welche Herausforderungen hattest du bisher, wenn du mit Legacysystemen und langlaufenden Projekten umgegangen bist?

Int-3: Also für mich die größte Herausforderung ist, dass man die Sachen nicht einfach starten kann auf seinem Rechner. Wir haben halt eine Entwicklungs-VM gehabt, und man musste dann immer auf einer anderen VM entwickeln, wenn man mal was testen will, den Source-Code auf die HDI, also auf die VM zum Testen und zum Bauen packen. Dort wurde die Software erst gebaut und dort konnten dann die Tests durchlaufen werden. Das ist halt schon ein langwieriger Prozess, wo dann auch schnell mal irgendwie bei Copy-Paste irgendwas falsch kopiert werden kann. Und ja, das macht es grundsätzlich schwieriger, loszulegen, weil da sehr viele Dinge ineinandergreifen, die man berücksichtigen muss, um einen einzelnen Test zu machen.

Interviewer: Kommt es im Projektverlauf regelmäßig zu Problemen, die durch veraltete Strukturen oder historisch gewachsene Systeme entstehen?

Int-3: Ja, das es gab schon. Also, was ich häufig erlebe, ist, dass es tote Dateien gibt, die über die Historie gewachsen sind, dass irgendwelche Skripte in abgewandelter Form vielleicht vorliegen, die gar nicht mehr verwendet werden und wenn man sich dann initial reinfuchst, um das zu automatisieren, dann kann es halt schnell passieren, dass man mal in einem falschen Skript unterwegs ist. Aber auch ansonsten, in verwendeten Skripten haben wir oft eine Redundanz, die ich bemerkt habe. Das heißt, wenn man dann irgendwie eine Optimierung an einer Stelle macht, muss man sie auch an anderer Stelle machen und ja, das macht das ein bisschen herausfordernd.

Interviewer: Und was haben die langen Lebenszyklen für eine Auswirkung auf die Wartbarkeit und Weiterentwicklung, die du beurteilen kannst?

Int-3: Ja, also das größte Problem, denke ich, ist vielleicht, dass Leute vor 25 Jahren oder 20 Jahren etwas gemacht haben, die vielleicht gar nicht mehr da sind. Man kann die nicht fragen. Es wurden zwar Dinge in der Erinnerungshistorie festgehalten, aber es kann sein, dass das nicht vollständig gepflegt ist. Das macht es zum einen schwierig, und zum anderen wurden vielleicht über die lange Laufzeit Dinge ausprobiert und nachher nicht wieder richtig entfernt. Also ein simples Beispiel ist, da hat man vielleicht mal hier in der Oberfläche was zeigen wollen, während sie im Test durchlief, hat dann Sleep von 1000 Sekunden eingebaut und vergessen den rauszunehmen. Ja, so was kann halt passieren. Ja, aber grundsätzlich auch durch die Historie, hätte man früher von Anfang an gewusst, wie es am Ende eingesetzt wird, hätte man vielleicht versucht, Redundanzen mehr zu vermeiden. Man hat teilweise auch bei den Tests zum Beispiel viele Dinge doppelt gemacht für einzelne Tests, wo man es hätte vielleicht zentralisiert rauskapseln können, was jetzt zum Teil auch passiert, wo wir aktuell auch viel tun.

Interviewer: Das greift schon ein bisschen in den nächsten Punkt, was für Maßnahmen setzt ihr derzeit ein, um Softwareverfall oder technische Schuld zu vermeiden? Also zum Beispiel Refactoring, Code Reviews, Automatisierung, was du ja angesprochen hattest.

Int-3: Ja, im Endeffekt all das: Also wir sind jetzt von SVN auf Git umgezogen und wir nutzen jetzt GitLab. Und in GitLab haben wir zum einen halt deutlich schönere Möglichkeiten auch agil zu arbeiten. Wir haben dann quasi ein Backlog, wo wir Issues pflegen, woraus wir die Merge-Requests erzeugen, woraus wir die Branches automatisiert erzeugen, sodass halt auch ein Issue mit dem Merge-Request und mit dem Branch gekoppelt ist. Wenn dann mal was abgeschlossen ist in sich in dem Merge-Request, dann wird das halt auch gereviewt mit mindestens vier Augen, meistens sogar mehr und dann wird das Feedback noch eingearbeitet und das wird alles schön in GitLab festgehalten. Das ist, denke ich, ein Aspekt, der wichtig ist aktuell. Und der zweite Aspekt ist natürlich die Automatisierung, die einfach dafür sorgt, weil früher war so ein bisschen das Problem, die Entwicklung, der Source-Code lag in SVN. Es gab Tests in Jenkins, die wurden zwar automatisch durchgelaufen, aber da gab halt auch Tests, die liefen in Jenkins nicht durch, dann ist man dazu übergegangen zu sagen, die skippen wir jetzt erstmal, weil in der VM laufen sie ja und so hat die Relevanz von diesen automatisierten Tests, ja, hat es an Relevanz gefehlt und sie wurden eher stiefmütterlich behandelt, sage ich mal. Und das wandelt sich jetzt ein bisschen, wir sind jetzt von Testdurchläufen auf einer VM zu Container übergegangen, dass wir auch ganz viele Tests parallel durchlaufen lassen können, so dass man schnelle Ergebnisse bekommt, dass es in GitLab integriert ist und jeder, der halt ein Issue bearbeitet, auch sicherstellen muss, dass die Tests auch in der Pipeline in den

Containern durchlaufen. Und ich denke, das ist ein wichtiger Schritt. Und ja, ein anderer Punkt ist auch, dass wir die Entwicklung vereinfachen wollen, indem wir mehr Container nutzen. Wir wollen die VM da an der Stelle jetzt auch nicht zwingend komplett ablösen, aber es soll halt auch mal schneller möglich sein, in einer Entwicklungs-VM Code zu entwickeln und direkt testen zu können. Dafür sind solche Container halt ziemlich praktisch. Dann starte ich den Container, gucke, baut mein Code, laufen die Tests da drin durch und dann kann ich so guten Gewissens wieder ins GitLab reinpushen. Und ja, das sind alles so Maßnahmen. Und auch ein weiterer Aspekt, der glaube ich nicht zu vernachlässigen ist: Wir haben halt viel auch in Word dokumentiert und für Entwickler ist es, glaube ich, auch einfacher in der Entwicklungsumgebung heraus, da mal schnell was zu dokumentieren und da ist Markdown zum Beispiel auch deutlich angenehmer. Und da wollen wir halt auch die Hürde möglichst runterbringen, damit Leute, wenn sie was ändern, weniger Hürden haben, das auch festzuhalten. Und ja, das ist so ein bisschen das Ziel. Die Hürden möglichst tief runterschrauben, die Qualität möglichst hoch und ja, Refactoring gehört natürlich auch dazu. Also immer, wenn wir jetzt neue Dinge überarbeiten, gucken wir, können wir das an anderen Stellen auch vereinfachen, können wir Sachen zentralisieren. Auch im Rahmen vom Git-Umzug haben wir viel Ballast abgeworfen, was wir jetzt erstmal nicht brauchen. Und das haben wir da auch genutzt.

Interviewer: Und was ist das jetzt für ein Zeitrahmen? Also seit wann wird das quasi gemacht?

Int-3: Also ich habe angefangen Ende letzten Jahres im November, allerdings mit einem anderen Projekt, aber mit dem gleichen Thema, halt die Containerisierung dieser Entwicklungs-VM. Und ja, das hat relativ schnell geklappt. Und dann bin ich irgendwann Anfang dieses Jahres dazu übergegangen, das halt live in die bestehende Entwicklung einzubringen. Und jetzt wird er halt auch aktiv mit den Pipelines entwickelt. Im Endeffekt seit Anfang des Jahres sind wir da relativ aktiv dabei.

Interviewer: Also ist es quasi ein sehr neues Projekt.

Int-3: Genau, also die Änderungen, die wir jetzt machen, die sind also relativ frisch. Also wir sind jetzt eigentlich im dritten Sprint, glaube ich, und haben noch zwei vor uns.

Interviewer: Und das wirkt ja quasi eher gegen technische Schuld. Aber siehst du irgendwo, dass, wenn technische Schuld irgendwann aufgenommen wurde, ob das je getrackt wurde, wenn es bewusst war, oder ist das quasi eher, ja, wird es aufgenommen und dann passiert es einfach.

Int-3: In diesem Projekt versuchen wir natürlich es schnellstmöglich zu beheben, wo es möglich ist. Wir haben tatsächlich Dinge festgestellt, die vertraglich so erledigt wurden, wo sich im Nachhinein gezeigt hat, das war jetzt vielleicht nicht die beste Umsetzung. Auch jetzt, durch das agile Vorgehen mit Scrum im Gespräch mit Kunden. Und dann haben wir

halt das auch schnell korrigiert und verbessert. In alten Projekten war es meistens so, dass man Dinge, die man vielleicht besser hätte machen können oder vielleicht besser machen kann. Da haben wir die potenziellen Verbesserungen als Issue aufgenommen und mit einem „Wont do“-Label markiert und dann weggeschoben. So dass wir im Nachgang jederzeit gucken können, das sind Dinge, die wir besser machen könnten. Machen wir jetzt aber nicht, aber sie sind halt festgehalten für die Zukunft, falls wir da nochmal was machen wollen. So versuchen wir, das hier auch zu leben. Aber aktuell gibt es da einfach zu wenig, was wir aufschieben oder wo wir technische Schulden aufbauen.

Interviewer: Und bei den anderen Projekten wurde das dann irgendwann angegangen, wenn das auf Wont Do gestellt wurde, oder war das eher zeitlich nie wirklich gewollt?

Int-3: Nee, wir haben da halt den Auftrag, wir hatten halt ein festes Budget, wir haben die Sachen auch dokumentiert, den Kunden mitgeteilt, aber da hat sich jetzt keine Folgeauftrag ergeben. Deswegen liegen sie da jetzt erstmal. Aber wenn da was kommen sollte, könnten wir sie wieder rausholen und daran arbeiten.

Interviewer: Und du hast ja gerade angesprochen, dass ihr agile Methodiken verwendet. Und merkt man, dass sich diese Praktiken auf Umgang mit technischer Schuld und Softwareverfall auswirken?

Int-3: Ich denke schon. Dadurch, dass wir täglich miteinander reden und täglich Fragen stellen und im Review mit anderen Personen zusammensitzen, da wird sowas dann halt immer, glaube ich, gefördert, dass man solche Dinge festhält und mitnimmt. Also wir hatten tatsächlich auch schon Issues, wo wir gesagt haben, das geht jetzt ein bisschen über das hinaus, was wir geschätzt haben für das Issue. Der Scope ist zu groß geworden, und wir haben das dann abgekapselt in ein eigenes Item und dann noch den nächsten Sprint geschoben. Und so versuchen wir halt, die Dinge mitzunehmen und ich glaube, da hilft das agile Vorgehen tatsächlich. Weil wir dann halt auch in einem Sprint-Review gucken, was hat gut geklappt und beim Planning nochmal gucken, was wollen wir jetzt machen. Und ja, Dinge, die dann halt hohe Prio bekommen, die können wir auch schnell angehen und nicht liegen lassen. Ich glaube, das würde bei so einem Wasserfallvorgehen schwieriger sein.

Interviewer: Und hast du das irgendwie mal mitbekommen, wie das abläuft, wenn es kein agiles Vorgehen gibt, wie dann mit Schuldenabbau oder Refactoring umgegangen wird?

Int-3: Ja, wenn ich den Scope jetzt ein bisschen ändere, also ich glaube, intern bei uns nicht, aber es gibt halt in der Beziehung zu anderen ARGE-Partner. Es gibt halt Themen, die fallen uns auf, aber dann gibt es das Projekt nicht her, sowas im Nachgang mal einfach so zu bessern. Weil das Problem ist, alles, was schon ausgeliefert ist, musste schon durch bestimmte Tests durch, durch bestimmte Verfahren durch, hat einen Stempel bekommen. Es ist halt schwierig da noch was nachzubessern. Und dieses starre Vorgehen ist etwas schwierig. Auch wenn es dann nachher um den kaufmännischen Part geht, da kann man

dann dem Kunden solche Dinge anbieten. Man kann es transparent machen, aber das sind dann meistens die Dinge, die als erstes wegverhandelt werden. Und das ist halt das Problem, wenn man es über ein Gesamtprojekt sieht. Intern sind wir natürlich immer jederzeit bereit, wenn das halt auch finanziert wird, alles möglichst schnell zu fixen. Aber diese Strukturen darüber hinaus, die machen es halt immer so schwierig.

Interviewer: Das heißt dann quasi, technische Schulden oder sowas, wenn sie nicht von uns behoben werden und dann quasi abgegeben wurden, dann bleiben sie eher bestehen über die lange Laufzeit. Es sei denn, sie bilden große Probleme, die wirklich angegangen werden müssen.

Int-3: Ja, der Bedarf beim Kunden muss einfach so groß sein, dass er bereit ist, explizit das anzugehen. Aber oft ist das dann für den Kunden einfach attraktiver, ein neues Feature zu nehmen, statt irgendwas Bestehendes nochmal zu korrigieren. Und ja, da ist es dann eine finanzielle Sache.

Interviewer: Vielleicht ist es nicht in deinem Projekt direkt sichtbar, aber vielleicht im Generellen. Wie wird die Architektur geplant, wie wird die Architektur dokumentiert, gepflegt in so einem langlaufenden Projekt?

Int-3: Ja, also in dem Projekt ist es aktuell halt schwierig, weil wir ja ein kleines Zahnrad im ganzen Konstrukt sind. Da wird vieles halt einfach vorgegeben. Aber in kleineren Projekten, wo ich jetzt auch unterwegs war in dem Kontext, und wenn wir es halt selber machen mussten, die Konzepte gebaut haben, dann haben wir das natürlich alles mit Skizzen dokumentiert und das dann halt auch kommuniziert und besprochen und Verbesserungsvorschläge aufgezeigt. Und ja, das haben wir in dem kleinen Kontext tatsächlich mit Markdown dokumentiert. Und ja, die Bilder mit so einem Editor erstellt und als PDF ausgeliefert.

Interviewer: Aber dann in den großen Projekten arbeitet man damit nicht wirklich, sondern nimmt das quasi als Vorlage und jemand anderes kümmert sich dann um Erweiterungen und sowas.

Int-3: Also meistens gibt es ja schon eine vorgegebene Kommunikationsinfrastruktur mit festgelegten Schnittstellen, was ja auch gut und wichtig ist. Und dann, wenn wir zum Beispiel einen Prozess bekommen, den wir überarbeiten oder neu erstellen wollen, dann müssen wir uns halt einfach an die Schnittstellen halten. Da kriegen wir von der Architektur selbst wenig mit. Also in einem kleineren Projekt, das wir gemacht haben, da ging es halt auch um eine neue Kommunikationsinfrastruktur. Da haben wir für unseren kleinen Bereich die Konzepte gemacht, haben dann auf Schwachstellen hingewiesen und da können wir jetzt nur darauf hoffen, dass die das in deren Gesamtarchitektur nachher berücksichtigen oder vielleicht an irgendwelchen Stellen schrauben können.

Interviewer: Aber wie wird sichergestellt, dass die Umsetzung sich an der vorgelegten

Architektur orientiert, auch wenn es ältere Projekte sind?

Int-3: Ja, im Endeffekt gibt es ja diese Kommunikationsinfrastruktur, und wenn wir nach dem Standard halt kommunizieren, dann halten wir uns ja indirekt an die Architektur, die gegeben ist. Da können wir halt auch wenig machen. Und da wird das Datenmodell auch vorgegeben. Es kann sein, dass sich da was ändert. Da müssen wir das bei uns gegebenenfalls anpassen und wenn vielleicht irgendwas nicht richtig ist, können wir es halt zurückkommunizieren. Aber da tut sich eigentlich nicht viel.

Interviewer: Vielleicht etwas mehr in deine Richtung: Du erwähntest ja schon Automatisierung. Zu welchem Level benutzt ihr dann CI/CD in euren Projekten, und wie sieht der Prozess genau aus. Hilft dir die CI/CD bei diesen zuvor genannten Problemen?

Int-3: Ja, also ich hoffe, dass es hilft. Das ist ja auch meine Aufgabe. Es ist halt eher auch so, dass wir eine gewisse Qualität nachweisen müssen. Und da gibt es halt verschiedene Parameter, die dann ja gemessen werden, teilweise wie Code-Coverage oder dass man statische Code-Analysen macht. Und es gibt da halt Vorgaben, an die wir uns halten müssen. Und sowas lässt sich halt gut darin abbilden. Und man kann dann durch die CI/CD, also dann halt eine Pipeline, und wenn an einem Prozess sich was ändert, dann wird der geänderte Prozess quasi in den Container geschmissen. Es wird geguckt, ob immer noch erfolgreich gebaut wird, wenn nicht, was sind die Fehler, dann gibt es ein Report. Wenn das erfolgt ist, dann gibt es die Tests, die dann durchlaufen werden. Und wenn da irgendwas nicht passt, dann gibt es auch da einen Report, dann kann ich in GitLab einsehen, was fehlgeschlagen ist. Ich kriege teilweise Screenshots, wenn ich ein Fehlerbild habe. Und so kann ich dann halt auch so Messwerte wie Test-Coverage mir ausgeben lassen, um halt jederzeit nachweisen zu können, dass sie gut ist oder vielleicht zu niedrig ist und kann da halt gegebenenfalls nachbessern. Und ich glaube halt, der große Mehrwert ist einfach, dass das einfach zu jeder Zeit passiert. Und auch gerade in der agile Softwareentwicklung ist ja der Grundgedanke, dass man am Sprintende ein lieferfähiges Produkt hat. Und wenn du das immer alles händisch testen musst, dann kriegst du das einfach nicht hin. Dann ist man ja zu 90% der Zeit einfach nur mit Tests beschäftigt, und deswegen hilft uns das da an der Stelle enorm.

Interviewer: Du hattest das schon angesprochen. Was für Werkzeuge und Metriken verwendet ihr für die Qualitätssicherung?

Int-3: Im Endeffekt ist es so: Wir haben die Testreports, die Units-Reports, die quasi die Ergebnisse der Unittests wiedergeben. Wir haben die Codecoverage als Kennzahl und wir haben die statische Codeanalyse, die uns dann Findings quasi hochwirft, wenn wir hier und dort am Code etwas besser machen können. Und darauf können wir dann reagieren. Und ansonsten muss ich überlegen, was wären Kennzahlen? Noch mehr Kennzahlen? Ich glaube erst mal nicht. Also wir haben grundsätzlich, gerade im Ada-Part, die Möglichkeit,

alle möglichen Kennzahlen noch zu messen. Und zwar wie tief zum Beispiel Schleifen gehen können, zu welcher Tiefe, wie viele Verzweigungen etwas nehmen kann. Und über solche Kennzahlen kann man dann halt auch gucken, dass man den Code vielleicht dahingehend optimiert, um ihn einfacher zu halten und wartbarer zu machen.

Interviewer: Und habt ihr irgendwelche Stärken oder Schwächen in diesem Bereich herausgefunden, wenn es um Codequalität und Wartbarkeit geht.

Int-3: Jetzt im Bezug auf die Automatisierung oder auf den Code selbst?

Interviewer: Auf die Automatisierung.

Int-3: Ja, also die Stärke habe ich ja gerade so ein bisschen genannt. Im Endeffekt dieses jederzeit lieferfähig sein, jederzeit funktionierenden oder kompilierbaren Code zu haben. Und einen getesteten Code haben, den man theoretisch ausliefern kann. Ich denke, das ist eine große Stärke. Wenn man jetzt eine Schwäche nennen müsste, ist natürlich der Aufwand dahinter, das Ganze so weit zu automatisieren. Aber gerade bei Langlaufprojekten, da rentiert sich das ja sehr schnell. Also wenn das jetzt so ein einmaliges Ding wäre, dann würde man diesen Aufwand vielleicht nicht in der Form betreiben. Aber wir wollen die Software über Jahrzehnte noch pflegen. Das lohnt sich.

Interviewer: Also wäre das quasi deine Empfehlung bei neuen Projekten, die voraussichtlich eine lange Laufzeit haben, dass man immer so eine Infrastruktur aufbaut, die so eine Automatisierung erlaubt?

Int-3: Also ich denke, grundsätzlich ist es immer, wenn man sich einen Moment aussuchen kann, in dem man diese ganzen Werkzeuge einführt, dann ist es am schönsten, wenn man es direkt zu Beginn macht. Dann profitiert man halt während der Entwicklung am meisten und hat halt auch den geringeren Aufwand als nachher erst eins zu bringen. Das sollte man das am Anfang machen. Und dann muss man natürlich halt immer gucken. Man muss ja auch nicht direkt die ganze Palette machen. Man kann ja mit kleinen Sachen anfangen und dann halt bei Bedarf immer weiter aufstocken. Und halt gucken, was für den Umfang oder für die Pflegbarkeit notwendig ist mit einem Augenmaß.

Interviewer: Benutzt ihr eigentlich eine Art von Quality-Gate oder sowas? Oder achtet ihr quasi einfach nur auf die Reports, die dann daraus kommen?

Int-3: Also wir machen halt die Code-Reviews. Die werden ja auch festgehalten und auch in den Code-Reviews wird dann immer jede Anmerkung notiert und festgehalten. Und ja, das ist im Endeffekt unsere Qualitätssicherung an der Stelle, wenn es darum geht, eine Lieferung fertig zu machen und mehr halt zum Thema Dokumentation und Vollständigkeit, aber nicht jetzt code-spezifisch, ob das fachlich alles richtig ist. Dann haben wir natürlich auch einen Qualitätssicherer, der da immer nochmal guckt, dass halt die Anforderungen, die wir auch vom Kunden haben, erfüllt sind und dass alles, was im Rahmen dieses ISO-Standards erforderlich ist, auch abgedeckt ist. Also jemanden, der uns halt auf die

Prozesse hinweist, haben wir da an der Stelle. Aber es ist halt eher organisatorischer Natur. Dass die entsprechende Dokumentation da ist und so weiter und der Rahmen eingehalten ist. Inhaltlich haben wir halt die Code Reviews.

Interviewer: Ja, okay. Aber nichts quasi in der Pipeline selber drin, dass ein Standard eingehalten werden muss, sonst failed die Pipeline oder sowas in die Richtung, sondern man guckt dann einfach quasi auf den Report.

Int-3: Also die Pipeline wird rot, wenn ein Test fehlschlägt.

Interviewer: Ja gut, aber wenn man jetzt zum Beispiel auf die Metriken, auf die Code Qualität oder auf die statische Analyse oder sowas guckt?

Int-3: Ja, nee, da haben wir jetzt aktuell, dass die Metriken ausgegeben werden. Also nur bei den Unit Tests, wie gesagt. Wenn die Tests nicht alle durchlaufen, dann wäre sie rot. Und das ist jetzt für uns ein Zustand, der muss behoben sein, bevor es gemerged wird.

Interviewer: Du hast ja schon die Code Reviews angesprochen. Wie laufen die ab? Wer beteiligt sich daran? Und wie wird das generell organisiert? Also du meinst ja, mindestens eine andere Person. Aber wie sieht das genau aus?

Int-3: Ja, also bei uns ist es so, dass ein Sprint-Team sagt, eine Person, ich bin jetzt mit meinem Item durch, es wäre Ready zum Review. Und dann meldet sich jemand und dann wird zum Review eingeladen. Und wenn Leute Lust haben, können rein theoretisch auch alle daran teilnehmen, je nachdem wie Zeit ist. Und ja, dann wird das halt mit mindestens vier Augen gereviewt. Aber es wird halt auch ein bisschen darauf geachtet, dass die weiteren zwei Augen auch ja, die Kompetenz besitzen, um das beurteilen zu können.

Interviewer: Und das hattest du ja auch schon sehr stark angesprochen. Aber hast du deiner Erfahrung nach den Eindruck, dass die Tests gut gegen technische Schulden helfen? Oder ist es quasi nur eine Absicherung, dass der Code so funktioniert, wie er soll?

Int-3: Ja, das ist halt so eine indirekte Hilfe. Die technischen Schulden, die lassen sich in Reviews leichter identifizieren und darauf aufmerksam machen. Aber das Tolle ist ja, wenn sie behoben sind, hast du die Tests, die dann sagen, die Tests laufen immer noch durch. Und es kann natürlich passieren, dass man auch technische Schulden durch die Tests aufdeckt. Also es gab zum Beispiel den Fall, dass nach einem Test nicht alles richtig abgeräumt wurde und beim zweiten Test der Test beeinflusst wurde. Und das ist dann mit dem lokalen Test nicht aufgefallen aber mit den automatisierten Tests schon. Und solche Fehler lasen sich auch dadurch gut auffinden. Und gerade auch wenn Prozesse dazwischenkommen, die dann vielleicht irgendwas nicht richtig abräumen. Ja, aber genauso richtige, bewusste technische Schulden, die werden dann eher auf persönlicher Ebene diskutiert.

Interviewer: Kannst du beurteilen, ob die Anforderungen an Sicherheit, Zuverlässigkeit oder regulatorische Vorgaben sich auf die Arbeit von dir auswirken?

Int-3: Also nicht aktuell, aber es gibt andere Projekte, da sind natürlich die Anforderungen zum Beispiel an die Code Coverage sehr hoch. Um so eine hohe Coverage zu erreichen von, ich weiß nicht, teilweise sind irgendwie 95% im Raum, die sind halt natürlich schwierig zu erreichen und das hätte natürlich Einflüsse. Ansonsten versuchen wir auch intrinsisch die Qualität so hoch wie möglich zu halten, auch zum Selbstzweck, damit wir auch selbst ein gutes Gewissen haben bei den Sachen, die wir ausliefern. Von daher würde ich sagen, sind wir eher auch intrinsisch getrieben. Und die Regulatorien sind oft tendenziell eher ein bisschen lascher. Ja, wobei, eine Sache ist zum Beispiel, wenn wir eine Vorgabe bekommen, dass wir mit bestimmten Tools testen müssen, dann ist das natürlich wieder mit extra Aufwand verbunden. Das wäre vielleicht doch dann etwas, was Auswirkungen hat.

Interviewer: Inwiefern beeinflussen die Beschaffungsprozesse oder lange Freigabeschleifen die Projekte in Bezug auf technische Schulden oder Softwareverfall?

Int-3: Da wären wir wieder bei dem Thema was wir zu Beginn ein bisschen hatten. Wir können zwar technische Schulden oder Fehler melden und kommunizieren. Aber wir sind natürlich darauf angewiesen, dass das auch beschafft wird, dass das behoben wird. Und ohne dass wir einen Auftrag bekommen, dürfen wir halt auch nichts machen. Wir können nicht einfach an Codebestandteilen irgendwas ändern, nur weil wir etwas schöner oder besser machen und korrigieren wollen. So toll es auch ist, wir dürfen es nicht, weil sie sind halt abgenommene Bestandteile. Daher dürfen wir nicht einfach beliebig an irgendwelche Sachen ändern. Darum sind wir in diesem Beschaffungsprozess natürlich gebunden. Und ja, der ist dann heute mal sehr langwierig. Und leider kann man auch nicht ausschließen, dass manche Fehler nie behoben werden. Und das ist halt ein Problem. Deswegen machen wir auch viel Werbung für agiles Vorgehen, wo sowas halt leichter unterzubringen ist.

Interviewer: Auf das gesamte Projekt das agile Vorgehen zu verwenden ist eher unwahrscheinlich derzeit? Gibt es da eine Bewegung, die quasi daran interessiert wäre?

Int-3: Ja, wir hatten zu den Sprint-Reviews ja auch schon das Beschaffungsamt hier und die sind grundsätzlich schon interessiert und sie sehen auch die Vorteile, aber es ist natürlich auch mit Nachteilen für das Beschaffungsamt dahingehend verbunden, dass sie natürlich mehr Präsenz zeigen müssen, mehr Aufwand unter Umständen haben, das enger begleiten müssen. Was natürlich der Qualität zugutekommt und ich glaube, das Interesse ist da, es tut sich ein bisschen was. Aus meiner Sicht müsste auch nicht alles direkt komplett agil passieren, wenn man halt in den Verträgen schon leichte Spielräume schaffen würde, mal Dinge zu streichen, neue Dinge reinzunehmen, die vielleicht im Laufe des Projekts einfach in der Priorität so eine hohe Gewichtung bekommen, wäre dem Problem schon viel geholfen. Und das ist halt das größte Problem, dass es diese Spielräume aktuell zu wenig gibt oder sie zu wenig genutzt werden.

Interviewer: Kannst du ein Beispiel nennen, in dem eine Maßnahme zur Reduktion von technischer Schuld oder zur Vermeidung von Softwareverfall besonders erfolgreich war?

Int-3: Eine Maßnahme, wo sie besonders erfolgreich war, also da wir jetzt das, woran ich arbeite, noch nicht ausgeliefert haben? Ja, wenn man das agile Vorgehen an sich nimmt und diese leichte Testbarkeit unsererseits, dann haben wir zumindest einmal schon sehr schnell auf einen Kundenwunsch reagiert und Dinge, die halt sonst einfach vielleicht nicht genutzt worden wären, weil sie zu umständlich umgesetzt wurden, korrigiert und mit wenig Aufwand jetzt umgesetzt haben. Das denke ich jetzt für das agile Vorgehen auf jeden Fall ein Gewinn und hat den Endkunden gezeigt, dass es möglich ist. Und ich glaube halt, es war vielleicht in dem Sinne weniger eine technische Schuld, sondern eine Usability-Schuld, die wir da beglichen haben. Wenn man diesen Mechanismus auch für andere technische Schulden noch umsetzen könnte, wäre es gut.

Interviewer: Und was macht die Agilität aus deiner Sicht besonders erfolgreich beim Vermeiden von technischer Schuld?

Int-3: Ja, das sind halt diese Spielräume, Dinge anzugehen. Also man hat zum einen eine engere Kommunikation mit dem Endkunden oder Endnutzer und man kriegt halt viel besser mit, was will er eigentlich wirklich. Oft entstehen ja auch Dinge, die er gar nicht so wollte, die aber in den Anforderungen so drin standen und das sind halt so Themen, bei denen finde ich es schwierig, zwischen technischen Schulden und vielleicht organisatorischen Schulden oder irgendwelchen Schulden zu unterscheiden. Es sind halt Dinge, die machen das System komplex, sie führen in die falsche Richtung, verschlechtern die Wartbarkeit und die Nutzbarkeit. Da müssen wir halt mehr ran und ich finde, mit agilem Vorgehen lässt sich das besser machen, halt durch die bessere Kommunikation, durch dieses Sprint-Vorgehen, wo man halt zum Beginn eines Sprints Dinge ganz nach oben nehmen kann, weil sie gerade wichtig sind und ja, das macht es einfacher. Beim Wasserfall kriegt man halt eine Eingabe von ganz vielen Anforderungen, setzt sie nach eigenem Wissen und Gewissen durch, bei manchen Dingen waren die Anforderungen vielleicht schlecht, bei manchen hat man es vielleicht nicht richtig interpretiert, so entsteht halt ein Riesenwust an, ja, falschen Umsetzungen und da wäre vielleicht das Thema technische Schulden drin. Wenn dann auf dem letzten Meter noch Dinge schnell hingebogen werden sollen, dann ist das natürlich ein Riesenpotenzial für solche technischen Schulden, dass man da schnell mal was einbaut, nur um irgendjemanden zu befriedigen. Sowas passiert bei einem agilen Vorgehen aus meiner Sicht ein bisschen weniger.

Interviewer: Gibt es weitere Erfahrungen oder Empfehlungen, die du in Zusammenhang mit Softwareverfall oder technischen Schulden gerne teilen möchtest?

Int-3: Es gab natürlich schon ein Projekt, an dem du auch mitgearbeitet hast, wo man quasi ein Projekt übernommen hat, wo schon etwas ziemlich unsauber war, wo aber

aus Zeitgründen auf dieser technischen Schuld aufgebaut hat. Und dann hat man irgendwann eine ganze Stadt draufgebaut und dann kriegt man sie nichtmehr so leicht weg. Grundsätzlich ist der Wille da, es zu beheben, aber das Geld nicht. Und das ist natürlich immer so ein wichtiger Aspekt. Wenn man eigentlich es schöner machen will, aber die Zeit oder das Geld es nicht hergeben, dann wachsen solche technischen Schulden. Automatisierung schenkt einem Zeit, wenn sie erstmal da ist. Dann kann man sich vielleicht Zeit für sowas nehmen und deswegen bin ich ein großer Verfechter der Automatisierung und auch vom agilen Vorgehen, um technische Schulden nicht zu groß werden zu lassen.

Usage of AI

These interviews were transcribed through a locally run version of OpenAI's Whisper with the model large-v3-turbo.

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, 16.5.2025

(Location, Date)



(handwritten signature)