FOM Hochschule für Oekonomie & Management

university location Hamburg

Bachelor Thesis

in the study course Business Information Systems

to obtain the degree of

Bachelor of Science (B.Sc.)

on the subject

Avoiding Software Decay in Military Software Development

by

Karl Wedrich

Advisor:	Prof. Dr. Ulrich Schüler
Matriculation Number:	597615
Submission:	March 20, 2025

Contents

List of Figures III

List of Tables IV

List of Abbreviations V

List of Symbols VI

1 Introduction 1

2 Literature Review 1

2.1 Problems of Software Engineering 1

2.2 Software Decay and Technical Debt 2

2.3 Mitigation Strategies in Commercial Environments 7

2.3.1 High-Level Mitigation Strategies 8

2.3.1.1 Agile Methodologies 8

2.3.1.2 Continuous Integration/Continuous Deployment 10

2.3.2 Specific Techniques and Tool-Supported Practices 11

2.3.2.1 Refactoring 11

2.3.3 Formal Debt Management and Tracking Techniques 11

2.3.4 Organizational and Cultural Factors 11

2.3.5 Conclusion 11

Appendix 12

Bibliography 13

List of Figures

List of Tables

List of Abbreviations

CI	Continuous Integration
CI/CD	Continuous Integration/Continuous Delivery
CD	Continuous Delivery
TDD	Test Driven Development
XP	Extreme Programming

List of Symbols

1 Introduction

T This is the Introduction.

2 Literature Review

2.1 Problems of Software Engineering

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

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

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

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

---

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

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

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

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

2.2 Software Decay and Technical Debt

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

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

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

This 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.'[14] 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.'[15]

The analysis of Eick et al. provide empirical validation for these theoretical laws, offering 'very strong evidence that code does decay.'[16] This conclusion is based on their findings that 'the span of changes increases over time'[17] meaning that modifications to the software tend to affect 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,'[18] measured specifically by three criteria: cost of change, time to implement change and the resulting quality of the software.[19] Therefore, the combination of theoretical insights from Lehman and Belday and empirical data from Eick et al. paints a clear picture: software decay is an inevitable consequence of ongoing evolution unless consciously and proactively managed through structured efforts such as continuous refactoring and architectural vigilance.

The concept of software decay aligns closely with earlier theoretical discussions by David Parnas (1994). In his influential paper 'Software Aging', Parnas describes software aging as a progressive deterioration of a program's internal quality primarily due to frequent, inadequately documented modifications which he termed 'Ignorant surgery'[20], as well as the failure to continuously adapt the architecture to evolving needs which he called 'Lack of movement'[21]. Without this proactive maintenance and refactoring effort, Parnas argues

---

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

that software inevitably reaches a state where changes become more risky, costly and error-prone[22].

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

The term ‘Technical Debt’ was first coined by Ward Cunningham in his paper ‘The WyCash Portfolio Management System’ (1992). This metaphor was used to describe the trade-off between a quickly implemented solution and a thought-out process. Use of using the quick solution ‘is like going into debt.’[23] 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’[24].

This term was further built upon and refined by the industry through white papers like ‘Technical Debt’ by Steve McConnell (2008) or the ‘Technical Debt Quadrant’ by Martin Fowler (2009). McConnell differentiates between two types of technical debt: Unintentional and Intentional[25]. 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.’[26] 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’[27]. The long-term debt on the other hand is more strategic and is taken on to help the team in a larger context. The difference between those two is that short-term debt ‘should be paid off quickly, perhaps as the first part of the next release cycle’[28], while long-term debt can be carried by companies for years.
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.’[29] 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

[22] *Parnas, D. L.*, Software Aging, 1994, pp. 280–281.
[23] *Cunningham, W.*, WyCash Portfolio, 1992, p. 2.
[24] Ibid., p. 2.
[25] *McConnell, S.*, Managing Technical Debt, 2017, p. 3.
[26] Ibid., p. 3.
[27] Ibid., p. 3.
[28] Ibid., p. 4.
[29] *Fowler, M.*, Technical Debt Quadrant, 2009.

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.[30]

In their article 'Technical Debt: From Metaphor to Theory and Practice' (2012) Kruchten et al. criticize the concept of technical to be 'somewhat diluted lately'[31], stating that every issue in software development was called some form of debt. Therefore they set out to define 'a theoretical foundation'[32] 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.[33] According to them, this includes architectural debt, 'documentation and testing'[34] as well as requirements and infrastructure debt. All these debt types allow engineers to better discuss the trade-offs with stakeholders and to make better decisions.

TODO: More about Kruchten and Theory?

There have been many studies providing empirical evidence for the theoretical concepts of technical debt. Highly influential studies were undertaken by Potdar and Shihab (2014) as well as by Li et al. (2015)
In their study Potdar and Shihab analyzed four large open source projects to find self-admitted technical debt as well as the likelihood of debt being removed. They found that 'self-admitted technical debt exists in 2.4% to 31% of the files.'[35] 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.'[36] They also discovered that 'only between

[30] *Fowler*, *M.*, Technical Debt Quadrant, 2009.
[31] *Kruchten*, *P.*, *Nord*, *R. L.*, *Ozkaya*, *I.*, Technical Debt, 2012, p. 18.
[32] Ibid., p. 19.
[33] Ibid., p. 19.
[34] Ibid., p. 20.
[35] *Potdar*, *A.*, *Shihab*, *E.*, Self-Admitted Debt, 2014, p. 1.
[36] Ibid., p. 1.

26.3% and 63.5% of the self-admitted technical debt gets removed'[37]. 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"'[38], as well as getting 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.'[39] Additionally they found that '[m]ost studies argue that TD negatively affects the maintainability [. . .] while other QAs and sub-QAs are only mentioned in a handful of studies'[40].

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.[41] Additionally practitioners 'tend to connect any software quality issue to debt, such as code smells debt, dependency debt and usability debt.'[42] 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.'[43] 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

[37] *Potdar, A.*, *Shihab, E.*, Self-Admitted Debt, 2014, p. 1.
[38] *Li, Z.*, *Avgeriou, P.*, *Liang, P.*, Mapping of Technical Debt, 2015, p. 194.
[39] Ibid., p. 215.
[40] Ibid., p. 215.
[41] Ibid., p. 211.
[42] Ibid., p. 212.
[43] *Banker, R.* et al., Software Complexity, 1993, p. 12.

refactoring, the internal design - the architecture - of software tend to decay.'[44] To prevent this, he suggests '[r]egular refactoring [to] help[s] keep the code in shape'[45].
To prevent long-term issues, practitioners recommend to actively manage technical debt through refactoring, tracking and other strategies that integrate debt management into the software development process.

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

2.3 Mitigation Strategies in Commercial Environments

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

[44] *Fowler, M.*, Refactoring, 2019, p. 58.
[45] Ibid., p. 58.

2.3.1 High-Level Mitigation Strategies

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

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

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

2.3.1.1 Agile Methodologies

In their paper ‘Technical Debt Management in Agile Software Development: A Systematic Mapping Study’ (2024)[46] Leite et al. investigated how agile methods can be used to manage technical debt. They found that ‘...Scrum and Extreme Programming are the most utilized methodologies for managing technical debt.’[47] While this study focuses explicitly on technical debt, both Scrum and XP also inherently address the boarder issue of software decay by encouraging proactive quality management and continuous improvement practices.

Scrum was first presented by Ken Schwaber in his paper ‘SCRUM Development Process’ (1995)[48]. It has since become on of the most popular agile frameworks in the software industry. Scrum explicitly manages software quality and technical debt through iterative cycles called sprints. After each sprint, the team reflects on their work, identifies quality issues, technical debt and potential decay indicators and plans improvements in retrospectives. Leite et al. found that for identifying technical debt in Scrum the Sprint and

[46] *Leite*, *G. d. S.* et al., Technical Debt Management in Agile Software Development, 2024.
[47] Ibid., p. 318.
[48] *Schwaber*, *K.*, SCRUM Process, 1997.

Product Backlog are the most used artifacts.[49] By explicitly managing these items with their workflow, teams effectively reduce both debt and software entropy, improving overall software maintainability.

XP, first introduced by Kent Beck in his influential book 'Extreme Programming Explained' (1999)[50], explictly integrates practices to enhance software quality and prevent software decay directly. XP practices such as pair programming, Test Driven Development (TDD) and Continuous Integration (CI) and continuos refactoring help maintain high software quality, thus preventing both debt accumulation and broader software entropy. Pair programming prevents decay by ensuring higher-quality code through collaborative review and knowledge sharing between developers. Continuous Integration ensures regular, frequent code integration, significantly reducing integration complexity and associated decay risks. TDD ensures robust test coverage, catching defects early and preventing quality erosion. Refactoring, a cornerstone XP practice, has proven its effectiveness empirically; for example, Moser et al. demonstrated in their case study(2008)[51] 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—directly counteracting software decay.[53] Beck argues that XP's incremental, continuos quality practices consistently maintain software quality and adaptability throughout development, directly addressing both debt and broader software decay.

Overall Agile methodologies, particularly Scrum and XP, systematically manage technical debt and proactively prevent software decay by fostering continuous improvement, structured quality management and adaptability. Complementing these Agile practices, the adoption of automated CI/CD pipelines further enhances the proactive management of both technical debt and broader software decay through rigorous quality control and systematic automation.

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

2.3.1.2 Continuous Integration/Continuous Deployment

CI was first introduced by Beck in the context of XP and later refined by Martin Fowler in his influential article 'Continuous Integration'[54]. Fowler describes CI as not only the frequent, automated integration of code into the main repository but also the systematic automation of building and testing process. According to Fowler, 'Self-testing code is so important to Continuous Integration that it is a necessary prerequisite.'[55]. Furthermore, another critical prerequisite is 'that they can correctly build the code,'[56] thus guaranteeing that code changes consistently integrate without issues.

To further prevent technical debt and broader software decay, a quality analysis tools (e.g. static code analyzers such as SonarQube or DeepSource) are frequently integrated into CI pipelines. These tools often provide a metric to evaluate technical debt which is calculated based on the effort in minutes to fix the found maintainability issues.[57] In their paper 'Technical Debt Measurement during Software Development using Sonarqube: Literature Review and a Case Study' (2021)[58] Murillo et al. found that SonarQube is a useful tool for early debt detection. The estimated remediation effort metric allows for a good debt management prioritization.[59] However during their research they noticed if they changed SonarQubes default rules by just 26 rules, the technical debt effort would increase from 1 hours and 50 minutes to 11 hours.[60] This and the fact that SonarQube can only detect code related debt and not for example infrastructure or requirements debt, makes these tools useful but not a complete solution.

Continuous Delivery (CD), introduced by Jez Humble and David Farley in their foundational book 'Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation' (2010)[61] 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

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

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].

Empirical evidence from academic studies clearly demonstrates the effectiveness of CI/CD in reducing both technical debt and software decay. For instance, TODO: Find emperical evidence for CI/CD

2.3.2 Specific Techniques and Tool-Supported Practices

In addition to high-level strategies, there are a variety of specific techniques and tool-supported practices which can be used to managed technical debt and prevent software decay. These practices are often integrated into the development process to provide targeted, effective quality assurance and debt management.

2.3.2.1 Refactoring

As mentioned earlier, refactoring is a key practice to prevent the code base from decaying. However, not only the complexity can be reduced by refactoring. In their study 'An Empirical Study of Refactoring Challenges and Benefits at Microsoft' (2014)[64] Kim et al. investigated benefits and challenges of refactoring at Microsoft, due to the fact that other studies showed very different results when it comes to the benefits of refactoring.

2.3.3 Formal Debt Management and Tracking Techniques

2.3.4 Organizational and Cultural Factors

2.3.5 Conclusion

[63] *Humble*, *J.*, *Farley*, *D.*, Continuous Delivery, 2010, p. 19.
[64] *Kim*, *M.*, *Zimmermann*, *T.*, *Nagappan*, *N.*, Refactoring Challenges, 2014.

Appendix

Appendix 1: Beispielanhang

Dieser Abschnitt dient nur dazu zu demonstrieren, wie ein Anhang aufgebaut seien kann.

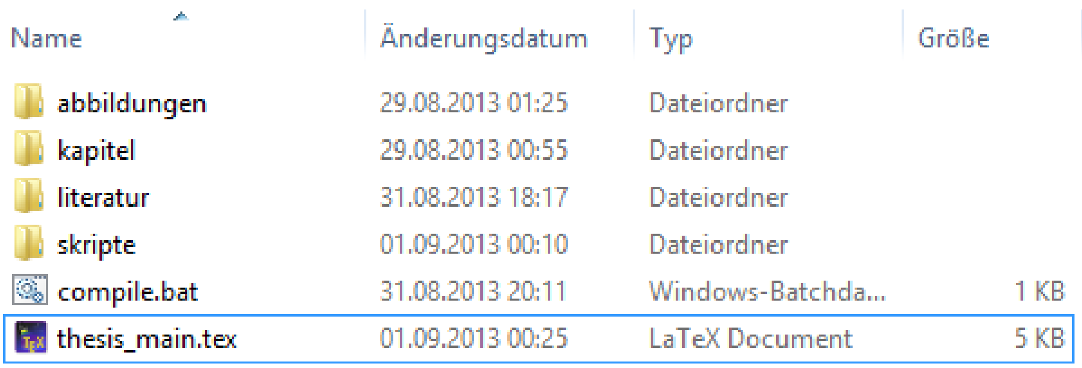
Appendix 1.1: Weitere Gliederungsebene

Auch eine zweite Gliederungsebene ist möglich.

Appendix 2: Bilder

Auch mit Bildern. Diese tauchen nicht im Abbildungsverzeichnis auf.

Figure 1: Beispielbild



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

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

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

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

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

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

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

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

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

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

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

Li, *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

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

Moser, *Raimund*, *Abrahamsson*, *Pekka*, *Pedrycz*, *Witold*, *Sillitti*, *Alberto*, *Succi*, *Giancarlo* (Case Study Refactoring, 2008): A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team, in: *Meyer*, *Bertrand*, *Nawrocki*, *Jerzy R.*, *Walter*, *Bartosz* (eds.), Balancing Agility and Formalism in Software Engineering, vol. 5082, s.l.: Springer Berlin Heidelberg, 2008, pp. 252–266

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

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

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

Internet sources

Fowler, *Martin* (Continuous Integration, 2006): Continuous Integration, (2006) [Access: 2025-03-18]

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

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

Declaration of authorship

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

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

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

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

Karl AC. Vedrich

Hamburg, 20.3.2025

(Location, Date)

(handwritten signature)