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 18, 2025

Contents

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 are made regarding the quality of software in terms of reliability, security and efficiency. These expectations come with challenges for the software developers across all industries. To efficiently deal with these challenges, software engineers have tried for decades to develop methods and guides to overcome these issues. However in 1986 Frederick Brooks published his paper 'No Silver Bullet'[1] in which he argues that: '...building software will always be hard. There is inherently no silver bullet.'[2]. He based this statement of the fact that there two types of difficulty 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'.[3] According to him this complexity makes 'conceiving, describing, and testing them hard'[4]. The second essential Brooks names is conformity. To explain this he compares software development to physics. Even though they are similar complex, physics has the advantage on relying on a single set of laws or 'creator'. The same cannot be said for software engineers. Brooks claims that the complexity is of 'arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform'[5]. This is due to software being perceived as 'the most comfortable'[6] thing to change in a system. Brooks explains the changeability issue 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.'[7] Another big part of the changeability issue the fact that software often 'survives beyond the normal life of the

[1] *Brooks, F.*, 1987.
[2] Ibid., p. 3.
[3] Ibid., p. 3.
[4] Ibid., p. 3.
[5] Ibid., p. 4.
[6] Ibid., p. 4.
[7] Ibid., p. 4.

machine vehicle for which it is first written'[8]. 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 makes the not only the creation difficult but also 'severely hinders communication among minds'[9]. According to Brooks these issues are in 'the very nature of software'[10]. For him these difficulty are unlikely to be solved unlike the accidental difficulties.

In contrast the accidental difficulties arise from limitations of current languages, tools and methodologies. Brooks notes that these issues—such as inefficient programming environments, suboptimal development processes and integration challenges can be overcome as the industry improves its practices and technologies. 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 Brooks presented have been since been substantiated by later empirical research. For instance, Lehman and Ramil (2003)[11] discussed in their paper that software systems that are left unchecked will experience a decline in quality over time. This phenomenon is encapsulated in Lehman's laws of software evolution, which he formulated in multiple papers. In their paper 'Software evolution - Background, theory, practice' Lehman and Ramil present empirical observations that support the notion that software quality tends to deteriorate over time - a phenomenon often described als 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)[12]. 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 mainly two things change constantly: the hard- and software environments and the requirements of the software.

[8] *Brooks*, *F.*, 1987, p. 4.
[9] Ibid., p. 4.
[10] Ibid., p. 2.
[11] *Lehman*, *M. M.*, *Ramil*, *J. F.*, 2003.
[12] *Eick*, *S. G.* et al., 2001.
[13] Ibid., p. 1.

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 suggests: ‘The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.’[15]

Eick et al. analysis provides empirical validation for these theoretical laws, offering ‘very strong evidence that code does decay.’[16] They base this conclusion 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. 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’[19], as well as the failure to continuously adapt the architecture to evolving needs which he called ‘Lack of movement’[20]. Without this proactive maintenance and refactoring effort, Parnas argues that software inevitably reaches a state where changes become more riskier, more costly and error-prone[21].

//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)[22] This metaphor was used to describe the trade-off

[14] *Belady*, *L.*, *Lehman*, *M.*, 1976, p. 228.
[15] Ibid., p. 228.
[16] *Eick*, *S. G.* et al., 2001, p. 7.
[17] Ibid., p. 7.
[18] Ibid., p. 3.
[19] *Parnas*, *D.*, 1994, p. 280.
[20] Ibid., p. 280.
[21] Ibid., pp. 280–281.
[22] *Cunningham*, *W.*, 1992.

between a quickly implemented solution and a thought out process. When ones uses the quick solution it ‘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)[25] or the ‘Technical Debt Quadrant’ by Martin Fowler (2009)[26]. McConnell differentiates between two types of technical debt: Unintentional and Intentional[27]. 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.’[28] As the first is not planned, it is difficult to avoid it. 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 short-term or long-term. The short-term debt is taken on to meet a deadline or to deliver a feature. Therefore it is ‘taken on tactically or reactively’[29]. The long-term debt on the other hand is more strategic and is taken on to help the team in a bigger picture. The difference between those two is that short-term debt ‘should be paid of quickly, perhaps as the first part of the next release cycle’[30], while long-term debt is something companies can be carried for years.
Martin Fowler on the other hand warned in 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.’[31] He created a quadrant between reckless and prudent and deliberate and inadvertent. The difference between reckless and prudent for Fowler is the way the debt is taken on. Reckless debt happens without the right evaluation of the consequences, risking difficulties in the future. Prudent on the other hand is taken on with the trade-offs in mind and the knowledge of the future costs. Fowler differentiates between deliberate and inadvertent is similar to McConnell’s differentiation between intentional and unintentional debt. The combination of those four result in either quick solutions without considering the long-term impact (reckless and deliberate), flawed

[23] *Cunningham*, *W.*, 1992, p. 2.
[24] Ibid., p. 2.
[25] *McConnell*, *S.*, 2017.
[26] *Fowler*, *M.*, 2009.
[27] *McConnell*, *S.*, 2017, p. 3.
[28] Ibid., p. 3.
[29] Ibid., p. 3.
[30] Ibid., p. 4.
[31] *Fowler*, *M.*, 2009.

design or implementation, either carelessly or unknowingly (reckless and inadvertent), purposefully taking on debt to gain a short-term advantage with plans of repayment (prudent and deliberate) or taking on debt due to lack of knowledge or experience (prudent and inadvertent).

In their article 'Technical Debt: From Metaphor to Theory and Practice' (2012) Kruchten et al.[32] criticize the concept of technical to be 'somewhat diluted lately'[33], stating that every issue in software development was called some form of debt. Therefore they set out do define 'a theoretical foundation'[34] 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.[35] According to them this includes architectural debt, 'documentation and testing'[36] 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.
Additionally, Kruchten et al. build upon the metaphor of Cunnigham
TODO: More about Kruchten and Theory?

There have been many studies providing empirical evidence to the theoretical concepts of technical debt. Two very influential ones are the study by Potdar and Shihab (2014)[37] as well as the study by Li et al. 2015[38].
To do this Potdar and Shihab analyzed four large open source projects to find self admitted technical debt as well as how likely it is that the debt will be removed. They found that 'self-admitted technical debt exists in 2.4% to 31% of the files.'[39] 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 teh amount of the self-admitted technical debt.'[40] They also discovered, that 'only between 26.3% and 63.5% of the self-admitted technical debt gets removed'[41]. 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.

[32] *Kruchten*, *P.*, *Nord*, *R. L.*, *Ozkaya*, *I.*, 2012.
[33] Ibid., p. 18.
[34] Ibid., p. 19.
[35] Ibid., p. 19.
[36] Ibid., p. 20.
[37] *Potdar*, *A.*, *Shihab*, *E.*, 2014.
[38] *Li*, *Z.*, *Avgeriou*, *P.*, *Liang*, *P.*, 2015.
[39] *Potdar*, *A.*, *Shihab*, *E.*, 2014, p. 1.
[40] Ibid., p. 1.
[41] Ibid., p. 1.

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"’[42], as well as getting an overview of the current research in the field. Different questions where asked like what types of technical debt exists, how does technical debt affect software quality and what is the limit of the technical debt metaphor.
They found 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.’[43] Additionally they found that ‘Most studies argue that TD negatively affects the maintainability [. . .] while other QAs and sub-QAs are only mentioned in a handful of studies’[44].
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.[45] Additionally practitioners ‘tend to connect any software quality issue to debt, such as code smells debt, dependency debt and usability debt.’[46] This indicates a inflationary use of the term, which one has to be aware of when speaking about technical debt.

The implications these studies have on 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’[47] (1993) Banker et al. empirically demonstrated that ‘software maintenance costs are significantly affected by the levels of existing software complexity.’[48] 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 ‘Without refactoring, the internal design - the architecture - of software tend to decay.’[49] To prevent this he suggests ‘Regular refactoring [to] help[s] keep the code in shape’[50].
To prevent long-term issues, practitioners recommend to actively manage technical debt through refactoring, tracking and other strategies that integrate debt management into the

[42] *Li*, *Z.*, *Avgeriou*, *P.*, *Liang*, *P.*, 2015, p. 194.
[43] Ibid., p. 215.
[44] Ibid., p. 215.
[45] Ibid., p. 211.
[46] Ibid., p. 212.
[47] **dSoftwareComplexityMaintenance1993**.
[48] **dSoftwareComplexityMaintenance1993**.
[49] *Fowler*, *M.*, 2019, p. 58.
[50] Ibid., p. 58.

software development process.

2.3 Mitigation Strategies in Commercial Environments

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

2.3.1 High-Level Mitigation Strategies

To efficiently mitigate software decay and technical debt, proactive management strategies are essential. These strategies aim to prevent the accumulation of debt and decay by integrating debt management directly into everyday development process. These include Agile methodologies like Scrum or Extreme Programming but also technical practices like Continuous Integration/Continuous Delivery (CI/CD). These practices are designed to embed ongoing maintenance and quality assurance into routine workflows.
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.
In a similar matter CI/CD introduces rigorous automation, rapid feedback loops and early detection of defects to proactively control technical debt accumulation. These methodologies create a culture of continuous improvement and quality insurance, 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)[51] Leite et al. investigated how agile methods can be used to

[51] *Leite*, *G. d. S.* et al., 2024.

manage technical debt. They found that '. . . Scrum and Extreme Programming are the most utilized methodologies for managing technical debt.'[52] Therefore, this section will focus on these two methodologies and their impact on technical debt management.
Scrum was first presented by Ken Schwaber in his paper 'SCRUM Development Process' (1995)[53]. It has since become on of the most popular agile frameworks in the software industry. It manages the the technical debt explicitly through iterative cycles called sprints. After each sprint the team has to opportunity to reflect on their work, identify technical debt and discuss improvements in so called retrospectives. Leite et al. found that for identifying technical debt in Scrum the Sprint and Product Backlog are the most used artifacts.[54] Making the debt visible in the backlogs creates an overview of the debt and helps the team to prioritize it, depending on the impact on the project.
Extreme Programming (XP) on the other hand was first presented by Kent Beck in his book 'Extreme Programming Explained' (1999)[55]. This framework introduces practices to increase software quality like pair programming, Test Driven Development (TDD) and Continuous Integration (CI) and refactoring. Pair programming has shown to prevent technical debt through collaboration of two developers. This way the code quality is increased and the knowledge is shared between the developers. Through CI new code is integrated into the main code base consistently. This way the code is always up to date and the risk of integration issues is reduced. Through TDD and a higher test coverage is introduced, preventing unintentional technical debt. Refactoring, a key component of XP, has proven to increase productivity and code quality by Moser et al. in their paper 'A Case Study on the Impact of Refactoring on Quality and Productivity in an Agile Team' (2008)[56]. They found that 'prevents an explosion of complexity' as well as 'driving developers to simpler design and as a consequence less complex and coupled and easier to maintain code.'[57] Due to all these benefits Beck argues that XP is able to have a more consistent software quality throughout the development process, due to the strong focus on incremental improvements.
Overall Agile methodologies, particularly Scrum and XP have proven to be effective in managing technical debt systematically. Complementing these methods, CI/CD further enhances technical debt control though automation and contionous quality assurance.

[52] *Leite*, *G. d. S.* et al., 2024, p. 318.
[53] *Schwaber*, *K.*, 1997.
[54] *Leite*, *G. d. S.* et al., 2024, p. 315.
[55] *Beck*, *K.*, 1999.
[56] *Moser*, *R.* et al., 2008.
[57] Ibid., p. 262.

2.3.1.2 Continuous Integration/Continuous Deployment

CI was first introduced by Beck in his book about XP and was later refined by Martin Fowler in his article ‘Continuous Integration’[58]. In this article Fowler describes CI not only as the automated integration of the code into the main database but also the automated testing and building of the code. According to Fowler ‘Self-testing code is so important to Continuous Integration that it is a necessary prerequisite.’[59]. In addition to the successful test another prerequisite is ‘that they can correctly build the code’[60]. To further prevent accidental technical debt a quality analysis tool can be used.
Continuous Delivery (CD) first introduced by Humble and Farley in their 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 in a releasable state at any time. According to Humble and Farley having a functional CD pipeline ‘creates a release process that is repeatable, reliable, and predictable’[62]. However, this is not the only benefit. Next to team empowerment and deployment flexibility the authors state that CD can reduce errors. Specially ‘those introduced into production by poor configuration management.’[63] This includes information like ‘configuration files, scripts to create databases and their schemas, build scripts, test harnesses, even development environments and operating system configurations’[64].
Combining these two techniques together creates an automatic process to catch integration issues early.

[58] *Fowler, M.*, 2006.
[59] Ibid.
[60] Ibid.
[61] *Humble, J., Farley, D.*, 2010.
[62] Ibid., p. 17.
[63] Ibid., p. 18.
[64] Ibid., p. 19.

Appendix

Appendix 1: Beispielanhang

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

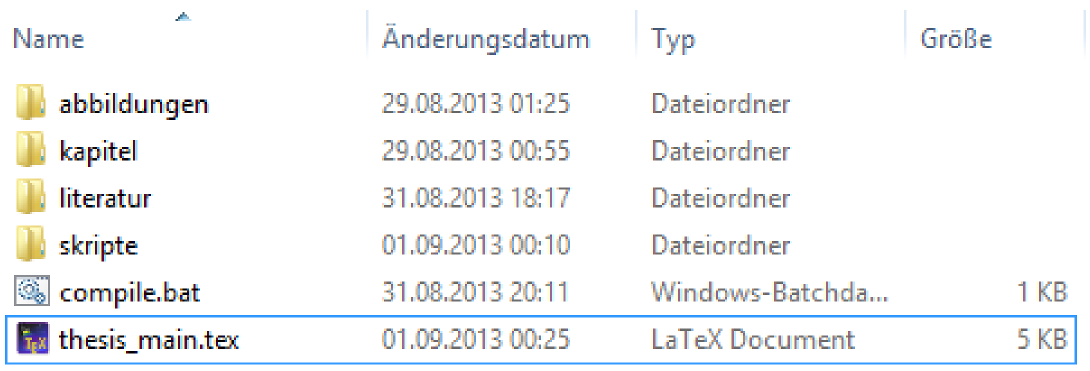
Appendix 1.1: Weitere Gliederungsebene

Auch eine zweite Gliederungsebene ist möglich.

Appendix 2: Bilder

Auch mit Bildern. Diese tauchen nicht im Abbildungsverzeichnis auf.

Figure 1: Beispielbild



Name	Änderungsdatum	Typ	Größe
abbildungen	29.08.2013 01:25	Dateiordner	
kapitel	29.08.2013 00:55	Dateiordner	
literatur	31.08.2013 18:17	Dateiordner	
skripte	01.09.2013 00:10	Dateiordner	
compile.bat	31.08.2013 20:11	Windows-Batchda...	1 KB
thesis_main.tex	01.09.2013 00:25	LaTeX Document	5 KB

Bibliography

Beck, *Kent* (1999): Extreme Programming Explained: Embrace Change, s.l.: Addison-Wesley Professional, 1999, 228 pp.

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

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

Cunningham, *Ward* (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* (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* (2019): Refactoring: Improving the Design of Existing Code, s.l.: Addison-Wesley, 2019, 418 pp.

Humble, *Jez*, *Farley*, *David* (2010): Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, s.l.: Addison-Wesley, 2010, 463 pp.

Kruchten, *Philippe*, *Nord*, *Robert L.*, *Ozkaya*, *Ipek* (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.* (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* (2024): Technical Debt Management in Agile Software Development: A Systematic Mapping Study, in: Proceedings of the XXIII Brazilian Symposium on Software Quality, SBQS '24, New York, NY, USA: Association for Computing Machinery, 2024-12-21, pp. 309–320

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

McConnell, *Steve* (2017): Managing Technical Debt, s.l., 2017-02-24, URL: https://www.construx.com/resources/whitepaper-managing-technical-debt/

Moser, *Raimund*, *Abrahamsson*, *Pekka*, *Pedrycz*, *Witold*, *Sillitti*, *Alberto*, *Succi*, *Giancarlo* (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, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 252–266

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

Potdar, *Aniket*, *Shihab*, *Emad* (2014): An Exploratory Study on Self-Admitted Technical Debt, in: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, ICSME '14, USA: IEEE Computer Society, 2014-09-29, pp. 91–100

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

Internet sources

Fowler, *Martin* (2006): Continuous Integration, <https : / / martinfowler . com / articles / continuousIntegration.html> (2006) [Access: 2025-03-18]

Fowler, *Martin* (2009): Technical Debt Quadrant, <https : / / martinfowler . com / bliki / TechnicalDebtQuadrant.html> (2009-10-14) [Access: 2025-03-10]

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.

[illegible]

Hamburg, 18.3.2025

(Location, Date)

(handwritten signature)