



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 10, 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
Appendix	5
Bibliography	6

List of Figures

List of Tables

List of Abbreviations

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 on the fact that there are 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 inherent 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, 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 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 is the fact that software often 'survives beyond the normal life of the

¹ Brooks, 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*, 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 ‘Techical 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 one 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.

²³ *Cunningham, W.*, 1992, p. 2.

²⁴ *Ibid.*, p. 2.

Appendix

Appendix 1: Beispielanhang

Dieser Abschnitt dient nur dazu zu demonstrieren, wie ein Anhang aufgebaut sein 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

- Belady, L., Lehman, M.* (1976): A Model of Large Program Development, in: IBM Systems Journal (1976)
- Brooks* (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
- 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
- Parnas, D.L.* (1994): Software Aging, in: Proceedings of 16th International Conference on Software Engineering, s.l., 1994, pp. 279–287

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, 10.3.2025

(Location, Date)



(handwritten signature)