EDUARDO FERREIRA FRANCO

A dynamical evaluation framework for technical debt management in software maintenance process

São Paulo / Roma
2020

EDUARDO FERREIRA FRANCO

A dynamical evaluation framework for technical debt management in software maintenance process

Programs: Computer Engineering and Operations Research

Advisors: Prof. Kechi Hirama and Prof. Paolo Dell'Olmo

São Paulo / Roma
2020

EDUARDO FERREIRA FRANCO

Original Version

A dynamical evaluation framework for technical debt management in software maintenance process

Thesis presented to the Polytechnique School of Universidade de São Paulo and the Università degli Studi di Roma “La Sapienza” as a partial requirement for obtaining the title of Doctor of Science.

Programs: Computer Engineering and Operations Research

Advisors: Prof. Kechi Hirama and Prof. Paolo Dell’Olmo

São Paulo / Roma
2020

I authorize the reproduction and total or partial dissemination of this work, by any conventional or electronic means, for the purposes of study and research, as long as the source is mentioned.

Cataloging in publication

Franco, Eduardo Ferreira
A dynamical evaluation framework for technical debt management in software maintenance process / E. F. Franco – São Paulo, 2020.
193 p.

Thesis (Doctorate) - Polytechnique School of Universidade de São Paulo. Departamento de Engenharia de Computação e Sistemas Digitais.

1.Technical debt 2.Software maintenance 3.Software evolution 4.Software sustainability 5. System dynamics I.Universidade de São Paulo. Escola Politécnica. Departamento de Engenharia de Computação e Sistemas Digitais II.t.

A sign that the Software Engineering profession has matured will be that we lose our preoccupation with the first release and focus on the long-term health of our products. Researchers and practitioners must change their perception of the problems of software development. Only then will Software Engineering deserve to be called Engineering.

(Parnas, 1994, p. 279)

I dedicate this dissertation to my daughter, Beatriz;
my wife, Paula; my sisters, Luciana and Gabriela;
my mother, Cristina; and my father, Laércio.

Acknowledgements

At last, this journey has come to an end. Without a doubt, it was more intense and challenging than I initially imagined. However, for countless reasons, and at the right moments, I had the company of people, or I had the opportunity to meet others, who were essential for defining my path and for encouraging me to pursue it. At the same time, these people inspired me not to be satisfied with the bare minimum, but always to look for more, to get out of my comfort zone.

First, I am grateful to my beloved wife, Paula, who jumped on board this journey with me. She was the first to buy into my “daydream” and started to motivate and encourage me to start this project a few years ago. She was always by my side, during both joyful moments and hard times. I am also thankful for my young little princess, Beatriz; she was not even born when this journey began, but has been my biggest inspiration from the first day of her life and the reason I have kept going and not given up.

I am forever grateful to Professor Laércio Joel Franco, my father, from whom I learned my first lessons and for whom I have deep and lasting admiration. It was him that fostered my interest in the endless pursuit of knowledge. Words are also insufficient to express my gratitude to my mother, Maria Cristina Ferreira Franco, and my two sisters, Luciana Ferreira Franco and Gabriela Ferreira Franco. They have always unconditionally supported me, and this has been no different during my studies.

I am deeply thankful to Professor Kechi Hirama, my advisor, for the trust he placed in me and for his guidance and support throughout this journey. His constant interventions and directions were crucial for the completion of this work.

I thank Professors Joaquim Santos and Selma Shin Shimizu Melnikoff for their valuable comments during the qualifying exam. I also thank Professor Joaquim Santos for sharing his experience and enthusiasm on system dynamics; he continuously challenged me and gave me valuable input that shaped this research.

During my doctorate, I had the privilege of undertaking a sandwich period in Italy. Professor Stefano Armenia played a central role in making this period so successful. He welcomed me at the Sapienza University of Rome and introduced me to the international research scenario while guiding my research and offering insightful feedback. I am also

thankful to Professor Paolo Dell'Olmo for agreeing to be my supervisor at Sapienza. He was always interested and available to listen to my ideas, and encouraged me to move forward. I thank Professor Salvatore Monaco, who initially supervised my mobility period and gave me all the support needed to establish my double degree program between Sapienza and USP.

I thank all the friends who have contributed ideas, support, and encouragement from the beginning of this project: Hamilton Carvalho, Bassiro Só, and Thyago Nepomuceno.

Finally, I thank the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), the Conselho Nacional de Pesquisa e Desenvolvimento (CNPq), and the European Erasmus Mundus Action 2 – Strand 1 (EMA2 – STRAND 1), under which the EBW+ Project was held, for providing me with research fellowships.

Abstract

FRANCO, E. F. **A dynamical evaluation framework for technical debt management in software maintenance process**. 2020. Thesis (Doctorate) – Polytechnique School of Universidade de São Paulo and the Università degli Studi di Roma "La Sapienza", 2020.

Over the years, initiatives involving software products have enabled increasing maintenance costs to keep them operating and meeting the needs of their users. During the lifetime of these software-based systems, development and maintenance activities inevitably introduce technical violations (some of which can be considered items of technical debt principal), whether intentional or not. If these violations are not adequately addressed, they can negatively impact the software product's maintainability and its capacity to adapt and evolve. In this context, there is growing motivation from the software engineering community, and from those directly involved in decision-making related to software investments, to assess and anticipate the impacts of resource allocation policies (investments) in the various maintenance activity types (perfective, corrective, and preventive). The aim is to preserve satisfactory technical quality characteristics of the software and, at the same time, maintain the cost and the tangible software asset itself at levels acceptable to organizations. Software-based systems have often been in operation for long period, which makes assessing how to allocate resources to maintenance a non-trivial and often complex activity. In line with these decision-making challenges, the modeling of the complexity, mainly with reference to the dynamic dimension, is gaining attention in terms of its use as a support tool for assessing the impact of various decisions on maintenance investments regarding the long-term effects. These effects inevitability define the evolutionary path of the software product, which goes through numerous iterations throughout its lifetime. The objective of this research is twofold. First, it aims to propose and develop a simulation model that enable us to expand knowledge in the area of software maintenance and technical debt management. Second, it aims to explore and evaluate the impact of different resource allocation policies among the different types of maintenance activities on the evolutionary behavior of software systems and their quality attributes related to functional suitability, reliability, and maintainability, together with economic aspects related to cost and tangible assets. The proposed simulation model was developed and tested using the system dynamics approach and, together with computational simulations, was used to evaluate three different resource allocation scenarios focused on (1) perfective maintenance, (2) preventive

maintenance, and (3) corrective maintenance. The data obtained from the three scenarios demonstrate counter-intuitive results. For example, focusing on preventive or corrective maintenance can cause, in the long run, the number of functional requirements in operation to be higher than when focusing exclusively on the development of functional requirements (perfective maintenance). However, the results obtained cannot be easily generalized. They depend on countless factors and variables that must be analyzed on a case-by-case basis, depending on context-specific characteristics related to each decision made regarding investments in software maintenance.

Keywords: Technical debt. Software maintenance. Software evolution. Software sustainability. System dynamics.

Resumo

FRANCO, E. F. **A dynamical evaluation framework for technical debt management in software maintenance process**. 2020. Thesis (Doctorate) – Polytechnique School of Universidade de São Paulo and the Università degli Studi di Roma "La Sapienza", 2020.

Ao longo dos anos, iniciativas envolvendo produtos de software tem apresentado custos crescentes para mantê-los operando e satisfazendo as necessidades de seus usuários. Durante o tempo de vida desses sistemas baseados em software, as atividades de desenvolvimento e manutenção inevitavelmente introduzem violações técnicas (algumas dessas podendo ser consideradas itens da dívida técnica). Essas violações podem ser geradas intencionalmente ou não e, se não forem tradadas, podem impactar negativamente a manutenibilidade e capacidade de adaptação e evolução do software com o passar do tempo. Nesse contexto, existe um interesse crescente da comunidade de engenharia de software e daqueles envolvidos diretamente nas tomadas de decisões relacionadas aos investimentos em manutenção de software. Esse interesse existe em avaliar e antecipar os impactos causados pelas políticas de alocações de recursos (investimentos) nas diversas modalidades de manutenção (perfectiva, corretiva e preventiva) de modo a preservar níveis satisfatórios de qualidade das características técnicas do software e, ao mesmo tempo, manter o custo e os ativos tangíveis de software em patamares aceitáveis para as organizações. Atualmente, os sistemas baseados em software têm operado por períodos longos e cada vez maiores, o que torna a avaliação de como alocar os recursos uma atividade não trivial e muitas vezes complexa. Alinhado a essas expectativas, a modelagem da complexidade, em especial pela dimensão dinâmica, vem ganhando atenção e sendo considerada como uma ferramenta de suporte capaz de avaliar o impacto de longo prazo de possíveis tomadas de decisão sobre investimentos em manutenção de software, que inevitavelmente definem o caminho evolucionário do produto de software que sofre inúmeras interferências ao longo do seu ciclo de vida. O objetivo deste trabalho de pesquisa foi propor e desenvolver um modelo de simulação que permitisse ampliar o conhecimento na área de manutenção de software e, ao mesmo tempo, explorar e avaliar o impacto que diferentes políticas de alocação de recursos em manutenção podem causar no comportamento evolutivo dos sistemas baseados em software e nos seus atributos de qualidade relacionados a adequação funcional, disponibilidade e manutenibilidade, juntamente com aspectos econômicos relacionados a custo e ativos tangíveis. O modelo de simulação proposto foi construído e testado

utilizando a abordagem de dinâmica de sistemas e, junto com simulações computacionais, permitiu avaliar três cenários distintos de alocação de recursos: o primeiro com foco na manutenção perfectiva, o segundo com foco na manutenção preventiva e o terceiro com foco na manutenção corretiva. Os dados obtidos a partir dos três cenários simulados demonstraram resultados contra intuitivos. Por exemplo, focar na manutenção preventiva ou corretiva pode fazer com que, no longo prazo, o número de requisitos funcionais em operação seja maior do que o obtido quando se foca exclusivamente no desenvolvimento de requisitos funcionais (manutenção perfectiva). Entretanto, os resultados obtidos não podem ser analisados de forma objetiva e conclusiva. Eles dependem de inúmeros fatores e variáveis que devem ser analisados caso a caso, dependendo do contexto único de cada tomada de decisão em investimentos em manutenção de software.

Palavras-chave: Manutenção de software. Dívida técnica. Evolução de software. Sustentabilidade de software. Dinâmica de sistemas.

Astratto

FRANCO, E. F. **A dynamical evaluation framework for technical debt management in software maintenance process**. 2020. Thesis (Doctorate) – Polytechnique School of Universidade de São Paulo and the Università degli Studi di Roma “La Sapienza”, 2020.

Nel corso degli anni, le iniziative relative ai prodotti software hanno consentito di aumentare i costi di manutenzione per mantenerli operativi e soddisfare le esigenze dei loro utenti. Durante la vita di questi sistemi basati su software, le attività di sviluppo e manutenzione introducono inevitabilmente violazioni tecniche (alcune delle quali possono essere considerate voci di debito tecnico), intenzionali o meno. Se queste violazioni non vengono adeguatamente affrontate, possono avere un impatto negativo sulla manutenibilità del prodotto software e sulla sua capacità di adattamento ed evoluzione. In questo contesto, vi è una crescente motivazione da parte della comunità dell'ingegneria del software e di coloro che sono direttamente coinvolti nel processo decisionale relativo agli investimenti nel software, per valutare e anticipare gli impatti delle politiche di allocazione delle risorse (investimenti) nei vari tipi di attività di manutenzione (perfetti, correttivo e preventivo). L'obiettivo è preservare le soddisfacenti caratteristiche di qualità tecnica del software e, allo stesso tempo, mantenere i costi e la risorsa software tangibile stessa a livelli accettabili per le organizzazioni. I sistemi basati su software sono stati spesso in funzione per un lungo periodo, il che rende la valutazione di come allocare le risorse per la manutenzione un'attività non banale e spesso complessa. In linea con queste sfide decisionali, la modellizzazione della complessità, principalmente in riferimento alla dimensione dinamica, sta attirando l'attenzione in termini di utilizzo come strumento di supporto per valutare l'impatto di varie decisioni sugli investimenti di manutenzione a lungo termine effetti. Questi inevitabili effetti definiscono il percorso evolutivo del prodotto software, che attraversa numerose iterazioni per tutta la sua vita. L'obiettivo di questa ricerca è duplice. Innanzitutto, mira a proporre e sviluppare un modello di simulazione che ci consenta di espandere le conoscenze nel settore della manutenzione del software e della gestione del debito tecnico. In secondo luogo, mira a esplorare e valutare l'impatto delle diverse politiche di allocazione delle risorse tra i diversi tipi di attività di manutenzione sul comportamento evolutivo dei sistemi software e i loro attributi di qualità relativi all'idoneità funzionale, affidabilità e manutenibilità, insieme agli aspetti economici relativi ai costi e beni materiali. Il modello di simulazione proposto è stato sviluppato e testato utilizzando l'approccio della dinamica del sistema e, insieme alle

simulazioni computazionali, è stato utilizzato per valutare tre diversi scenari di allocazione delle risorse incentrati su (1) manutenzione perfetta, (2) manutenzione preventiva e (3) manutenzione correttiva. I dati ottenuti dai tre scenari dimostrano risultati controintuitivi. Ad esempio, concentrarsi sulla manutenzione preventiva o correttiva può comportare, a lungo termine, un numero di requisiti funzionali in funzione superiore rispetto a quando si concentra esclusivamente sullo sviluppo di requisiti funzionali (manutenzione perfetta). Tuttavia, i risultati ottenuti non possono essere facilmente generalizzati. Dipendono da innumerevoli fattori e variabili che devono essere analizzati caso per caso, a seconda delle caratteristiche specifiche del contesto relative a ciascuna decisione presa in merito agli investimenti nella manutenzione del software.

Parole chiave: Manutenzione del software. Debito tecnico. Evoluzione del software. Sostenibilità del software. Dinamica del sistema.

List of figures

Page #

Figure 1. Percentage of effort expended on hardware, development, and maintenance 34

Figure 2. Interaction of a software product with the operating environment. 36

Figure 3. IBM OS/360 growth throughout releases 38

Figure 4. Types of software product maintenance 40

Figure 5. Evolution history of software product quality measurement models 41

Figure 6. Hierarchical structure of the SQuaRE quality model 42

Figure 7. Software product quality mode 43

Figure 8. GQM method’s hierarchical evaluation structure 49

Figure 9. System dynamics’ iterative modeling process 56

Figure 10. Examples of the elements of a causal loop diagram 58

Figure 11. Logical sequence of formal steps of model evaluation 62

Figure 12. Theoretical elements of the proposed framework 65

Figure 13. Overview of the proposed dynamical evaluation framework 68

Figure 14. Maintenance costs behavior over time 73

Figure 15. Technical debt (TD) principal and maintenance effort behavior overtime 74

Figure 16. Changes to software maintainability when the effort employed remains constant 76

Figure 17. Change of effort employed and the software maintainability when change rate is constant 76

Figure 18. Software system’s functionality growth over time 77

Figure 19. Expected growth in higher (left) and lower (right) technical debt scenarios 79

Figure 20. Maintenance policies variations and development investments decisions 80

Figure 21. Impact of technical debt interest due to technical debt item’s repayment 82

Figure 22. Subsystem diagram of the proposed model. 83

Figure 23. Continuing growth feedback loop 86

Figure 24. Increasing complexity feedback loop 87

Figure 25. Requirements gold plating feedback structure 88

Figure 26. Declining quality feedback structure 90

Figure 27. Continuing changes feedback structure 91

Figure 28.Work harder feedback structure 92

Figure 29. Haste makes waste feedback structure 94

Figure 30. Work smarter feedback structure 96
Figure 31. Self-regulation feedback structure 98
Figure 32. Perfective maintenance activities subsystem’s stock and flow diagram...............102
Figure 33. Corrective and preventive maintenance subsystem’s stock and flow diagram.....104
Figure 34. Resource management subsystem’s stock and flow diagram 108
Figure 35. Goal evaluation subsystem stock and flow diagram...110
Figure 36. Nominal versus current perfective maintenance productivity for Scenario #1.....117
Figure 37. Functional requirements growth pattern for Scenario #1....................................118
Figure 38. Resource allocation fractions for Scenario #1 119
Figure 39. Preventive and corrective violations density for Scenario #1..............................120
Figure 40. Tangible and perceived asset, and opportunity costs for Scenario #1 121
Figure 41. Technical debt's principal and interest for Scenario #1 122
Figure 42. Relative debt to current asset for Scenario #1...123
Figure 43. Nominal versus current perfective maintenance productivity for Scenario #2.....124
Figure 44.Functional requirements growth pattern for Scenario #2......................................125
Figure 45. Resource allocation fractions for Scenario #2 126
Figure 46. Preventive and corrective violations density for Scenario #2..............................126
Figure 47. Tangible and perceived asset, and opportunity costs for Scenario #2 127
Figure 48. Technical debt's principal and interest for Scenario #2 128
Figure 49. Relative debt to current asset for Scenario #2...129
Figure 50. Nominal versus current perfective maintenance productivity for Scenario #3.....130
Figure 51. Functional requirements growth pattern for Scenario #3....................................131
Figure 52. Resource allocation fractions for Scenario #3 132
Figure 53. Preventive and corrective violations density for Scenario #3.............................132
Figure 54. Tangible and perceived asset, and opportunity costs for Scenario #3 133
Figure 55. Technical debt's principal and interest for Scenario #3 134
Figure 56. Relative debt to current asset for Scenario #3...135
Figure 57. Comparisons of the results of the simulated scenarios 138
Figure 58. Example of software development framework 166
Figure 59. Example of a reinforcement loop for continuous software growth......................167
Figure 60. Example of balancing mesh for hiring people 168
Figure 61. Structure of delay in the assimilation of new employees.....................................169
Figure 62. Example of table function related to working hours vs. productivity..................171
Figure 63. Example of the software development process 172

Figure 64. Maintenance productivity variance due to size of base system and cyclomatic complexity 190

List of tables

Page #

Table 1. Average lifetime of software applications still in use (years) 26

Table 2. Lehman's laws of software evolution.. 37

Table 3. Software sustainability dimensions 47

Table 4. Goal/Question/Metric model example 50

Table 5. Technical sustainability goals question, and metrics structure. 69

Table 6. Economic sustainability goals question, and metrics structure 70

Table 7. Description of the proposed model's subsystems 83

Table 8. Boundary chart of the proposed model 84

Table 9. Summary of the tests performed on the model.. 111

Table 10. Model's initial conditions for Scenario #1 116

Table 11. Model's initial conditions for Scenario #2. 124

Table 12. Model's initial conditions for Scenario #3. 130

Table 13. Final conditions of the model's elements for the three simulated scenarios (120 months) 136

Table 14. Final conditions of the model's elements for the three simulated scenarios (60 months) 137

Table 15. System dynamics model's elements 160

Table 16. Polarity of relations and definitions, with examples. 163

Table 17. Common modes of behavior and their feedback structures 164

Table 18. Software defects per function point by industry segment 188

Table 19. Software defect origin percent by industry segment 188

Table 20. Approximate U.S. productivity ranges by of applications (data expressed in function points per staff month).. 189

Table 21. Approximate productivity rates by size of application (data expressed in terms of function points per staff month) 189

Table 22. U.S. average productivity in function points per staff month 190

Table 23. Application probable requirements "creep" (data expressed in percentage of original requirements) 191

Table 24. Average rate of annual enhancements (data is based on percentage change of application function points) 191

Table 25. Defect repairs time by defect origins....192
Table 26. U.S. average for delivered defects per function point....193

Table of contents

Page #

1. Introduction 22
1.1 Motivation 23
1.2 Objective 29
1.3 Justification 30
1.4 Document structure 33
2. Background 34
2.1 Software evolution 34
2.2 Software maintenance 39
2.3 Software quality models 41
2.4 Technical debt 44
2.5 Software sustainability 46
2.6 Static analysis 48
2.7 Goal, Question, Metric method 49
2.8 Chapter summary 51
3. Materials and methods 53
3.1 Research questions 53
3.2 Research objectives 53
3.3 Research process 54
3.3.1 System dynamics 55
3.4 Chapter summary 63
4. The Dynamical Evaluation Framework 65
4.1 Hierarchical software sustainability evaluation structure 69
4.1.1 Technical sustainability evaluation 69
4.1.2 Economic sustainability evaluation 70
4.2 Proposed simulation model 72
4.2.1 Problem articulation and dynamical hypothesis 72
4.2.2 Model formulation 100
4.2.3 Model testing 111
4.2.4 Policy formulation and evaluation 113
4.3 Chapter summary 113

5. Results and discussion 115
5.1 Model evaluation 115
5.1.1 Scenario #1: Perfective maintenance focus 115
5.1.2 Scenario #2: Preventive maintenance focus 123
5.1.3 Scenario #3: Corrective maintenance focus 129
5.2 Scenarios comparison 135
5.3 Chapter summary 139
6. Conclusions 140
6.1 Addressing the proposed research questions 140
6.2 Contributions 143
6.3 Areas of future research 144
References 146
Appendix A – System dynamics tools and elements 160
A.1 Elements and notations 160
A.2 Mathematical formulation 162
A.3 Common behaviors and their corresponding feedback structures 164
A.4 Basic patterns and equations 165
A.4.1 Constant flow and one stock 166
A.4.2 Variable flow and one stock 166
A.4.3 Reinforcing loop 167
A.4.4 Balancing loop 168
A.4.5 Delay 169
A.4.6 Table function 170
A.5 Example 171
Appendix B – Model documentation 173
B.1 Perfective maintenance subsystem 173
B.2 Corrective & preventive maintenance subsystem 176
B.3 Resource allocation sector 178
B.4 Goal evaluation sector 182
Appendix C – Secondary data used 188

1. Introduction

This work explores the influence, within software maintenance activities, that different resource allocation policies have on software product quality attributes throughout the phases of operation, maintenance and deactivation. To evaluate the impact of different scenarios, a model constructed according to the systems dynamics approach (Forrester, 1961, 1969, 1971) is used in conjunction with computational simulations.

The unit of analysis corresponds to the set of elements comprising the software maintenance process, the software product's quality attributes, and the influences that the operating environment has on the dynamic behaviors of those quality attributes throughout the software product's lifetime.

Among the existing different types of software systems, this research investigates those whose use is embedded in corporate environments and that "operate in or address a problem or activities of the real world" (Lehman & Ramil, 2006, p. 12). They automate human or social activities and make assumptions about the real world, and they interact with it by providing or requesting services, thereby becoming an integral part of the domains within they operate and that they address.

In general, software products of this type are known as "E-type" software and represent most current operating software systems (Lehman, 1980, 1991, 1996b; Lehman & Ramil, 2003). The "E" stands for evolutionary as they must be adapted to fit any change occurring in the real world. As its operational context is dynamic, "E-type" software must be continuously adapted to remain faithful to its domain, and its application purpose, compatible with its operating environment, and relevant to the objectives and expectations of its stakeholders (Cook et al., 2006).

Another characteristic is that such software becomes an integral part of the domain in which it operates, influencing and being influenced by the environment. In order to remain compatible with the inevitable changes in applications, domains, and properties, this software must be continually modified and updated – that is, it must evolve (Lehman & Ramil, 2006).

1.1 Motivation

It is not surprising that currently the software inventory owned by a company usually represents a significant share of its assets (Wiederhold, 2006); thus the company has a vital interest in preserving and maximizing the investments made to build its software libraries and to optimize future ones.

The dissemination and use of software products in corporate environments became a reality some decades ago. Some authors consider software products no longer as competitive advantages, but as commodities (Carr, 2003). The wide dissemination of software has resulted in organizations becoming operationally, managerially, and strategically dependent on software-based information systems (Melville et al., 2004). This phenomenon has been accelerated both by increased competitiveness (Bharadwaj et al., 2013; Drnevich & Croson, 2013), and by the unimaginable amount of data and information required for decision making within an increasingly reduced response time (Chen et al., 2012).

This increasing dependence demands large investments that are frequently associated with greater expectations of positive results and higher returns on the investments. Those who finance projects involving software products aim to be successful at the end of the software's development, deployment, and operation (McKinsey & Company, 2011), regardless of the current common understanding of what success and completion of these initiatives mean.

Despite these expectations, the literature reports failures of such software initiatives, presenting cases where the software was unable to deliver the expected benefits and recorded disappointing performance indexes. Although controversial, the data presented by the "Chaos Report" (Standish Group International, 2013), for the period from 1994 to 2012 show that, on average, 65% of the evaluated projects failed and were canceled before completion (which is termed "failed") or had some type of change in relation to the initially anticipated term, cost, or scope (which is termed "challenged").

Challenges occur in software and information system projects in various contexts, including complex information system deployments such as Enterprise Resource Planning (ERP) (Hong & Kim, 2002), international software development (Ahsan & Gunawan, 2010), military IT projects (Royal Academy of Engineering, 2004), and the British government's initiative to automate health records that extended from 2000 to 2010 and was abandoned after costs reached the order of US$5–10 billion (Sommerville et al., 2012).

Consequently, there is an increasing trend of research on complex projects (Bosch-Rekveldt et al., 2011; Johnson, 2013) and the dynamic aspect stands out among the dimensions that characterize complexity (Geraldi et al., 2010).

Although widely cited, the "Chaos Report" is also questioned and criticized (Eveleens & Verhoef, 2010; Glass, 2005, 2006; Jørgensen & Moløkken-Østvold, 2006). Until 2014, this report adopted three criteria for defining and evaluating success: time, cost, and quality (also known as the "iron triangle"). In other words, to be successful a project had to adhere to the initial forecast for these three variables.

The iron triangle, which has been widely used as the criteria for evaluating project success, has been discussed and expanded in recent years in light of various considerations. For example, it has been argued that success evaluation criteria vary from project to project due to difference in size, complexity, and uniqueness. Müller and Turner (2007), and several authors have suggested that success consists of a multi-dimensional and inter-related construct (Carvalho & Rabechini Junior, 2015; Shenhar et al., 2001). These dimensions can include project efficiency, impacts on the team, impacts on the customer, a distinction between business and direct success, and preparation for the future (Shenhar et al., 2001) and, more recently, sustainability dimensions have also been discussed (Carvalho & Rabechini Junior, 2015, 2017).

There is an understanding that software system projects are broader than just putting artifacts into operation. The introduction of a software system alters the structure and culture of an organization; in addition, it changes the way people think and work (Dwivedi et al., 2014). The adoption of a software system also has political implications, since it has the potential to allow some situations and restrict others, causing some people to win and others to lose influence and power (Orlikowski & Robey, 1991). These implications indicate that the main issues associated with the success of software projects are related to political, cultural, and personal factors (Markus et al., 2000).

Defining the success of software system initiatives is also a non-trivial activity. Consequently, there is no consensus in the research community about how to define and measure it (Cecez-Kecmanovic et al., 2014; Seddon et al., 2002).

Among the various proposed models in the scientific literature for evaluating the success of initiatives of this nature, an especially prominent and influential one was proposed by DeLone and McLean (1992). Revised 10 years later following contributions and critiques from the scientific community (DeLone & McLean, 2003), the model proposes a user-centered

approach to evaluate success, which is defined as a dependent variable of six interdependent dimensions: quality of the system, quality of information; quality of service; intention to use and use of the system; user satisfaction; and benefits generated.

In relation to the analysis of failures, on the other hand, Sauer (1993) proposed that a software system can only be considered a failure when its development or its operation is canceled. Based on this criterion of failure, software-based systems resemble natural systems, where observed behaviors are explained by their survival goals. The survival of a software system is obtained through the continuous supply of resources (e.g., finances, people) that support the continuity of its operation, and thus it cannot be considered a failure while continuing to attract those necessary resources (Yeo, 2002).

Comparing the success of software systems with the survival of natural systems is also supported by the fact that their early life cycle of design, construction, testing, and deployment constitutes a small fraction of their entire lifetime and total investments. Lehman (1980) found that of the total investments made in software in the United States, 70% of the resources were destined for maintenance (which he considered to be any type of change made in the software after it had begun operation); this figure was later revised to approximately 80% (Glass, 2001; INCOSE, 2015).

Furthermore,

Table 1 shows that the average lifetime of software products has been growing steadily over the last few decades (Jones, 2008). The longer life expectancy implies not only greater investments to ensure that the software contributes to satisfy business and user needs, but also the challenge of predicting and anticipating the long-term effects in the early stages of software development and operation.

The higher and constant demand for resources, even after the software system deployment and operation, has caught the attention of the scientific community, and since the 1970s researchers have begun to investigate the possible causes of the demand for constant investments even after software development and deployment have been completed (Belady & Lehman, 1971, 1976; Woodside, 1979). These investigations, along with the advances made in recent decades, have given rise to a new area of research in software engineering known as "software evolution", and to the consolidation of laws of evolution that describe abstractions of observed behaviors based on models (Lehman, 1980; Lehman & Ramil, 2006).

Table 1. Average lifetime of software applications still in use (years)

Type of application	**1990**	**1995**	**2005**
End-user	1.50	2.00	2.00
Web	-.-	1.50	5.00
MIS	10.00	15.00	20.00
Outsourced	5.00	7.00	9.00
Systems	5.50	8.00	12.00
Commercial	2.00	2.50	3.80
Military	12.00	16.00	23.00
Average	**5.14**	**7.43**	**10.69**

Source: Adapted from Jones (2008)

Lehman (1980) found that software, like complex natural systems, evolves as responses and reactions to pressures from the external environment, and that changes in operational, functional and structural patterns inevitably makes software systems more complex, inflexible and resistant to changes. To survive, software systems must maintain their adaptability and ability to change, and the extent to which these are achieved can make all the difference to their success or failure, profit or loss.

> Whether solution is satisfactory also depends on the circumstances during execution or, more precisely, when the results of the execution are applied. But computing systems are, in general, tightly and intimately coupled to applications and application domains that are forever changing. Hence software must evolve, undergoing continuous adaptation and change. It must be treated as an *ever be adapted* ***organism***, not as *to he produced once* ***artifact***. This fact compounds the problems to be solved in computer application, a major challenge to implementors and users alike. (Lehman, 1989, p. 5, emphasis in original)

Interest in the topic of complexity in the research literature on software-based systems has grown in recent years (Nan, 2011; Sommerville et al., 2012; Whitney & Daniels, 2013), especially with respect to their dynamic and emergent behaviors (Georgantzas & Katsamakas, 2008; Geraldi et al., 2010). As well as the recent growth of the topic's popularity among the software engineering community, the subject has been discussed for decades in other areas of science (Weaver, 1948).

Complex systems are a multidisciplinary subject; therefore, there is not a single and absolute definition of them. However, some of their characteristics are common to several existing definitions, according to which complex system are described as being composed of multiple elements, that interact in a nonlinear way (colloquially, the "whole is greater than the sum of the parts"), do not have a central control, present emerging behaviors, process information, and adapt through learning and evolution (Mitchell, 2011).

Some of these characteristics are shared with definitions of complex software-based systems. For example, Stoyenko (1995) described them as computational systems composed of multiple components, that interact with external elements (including people), operate in an uninterrupted and adaptive way, suffer progressive degradation, and produce unpredictable reactions when subjected to a sequence of unexpected events. These systems have a long lifetime (years or decades), and during the period of operation the system's complexity increases as its components evolve, and as its logical and physical interconnections, its operational interfaces and its semantics change.

Throughout its life cycle, a software product must adapt and evolve in response to several external influences. The ability to adapt and evolve depends on the intrinsic characteristics of its quality attributes, which in turn are determined by the prioritization of the different types of maintenance activities. The resource allocation in maintenance activities represents the configuration of a given organization's policy to respond to external stimuli and defines the evolutionary path of the organization's software product, a path that is influenced and regulated by several factors.

During the software product lifetime, those responsible for its development and maintenance may violate good practices related to architecture and coding. The "technical debt" metaphor was created to describe the liabilities accumulated by decisions, whether intentional or unintentional, to deliver software products with inferior quality to achieve business objectives (e.g., to shorten delivery schedules). Technical debt refers to the accumulation of violations caused by decisions that increase the cost associated with software products' maintenance and reduce the products' ability to change to meet current and future business needs (Cunningham, 1993).

An accumulation of technical debt can accelerate the delivery of features to meet immediate needs. However, this strategy also entails a growing accumulation of unresolved errors and violations, which lessen the flexibility and ability to modify the software product,

increases the cost and time required to carry out the maintenance activities, and, consequently, reduces the feasibility of keeping the software in operation and its capacity to meet future demands (Ramasubbu & Kemerer, 2014).

Dynamically complex business contexts make the evaluation of the long-term impacts of different investment policies, or even the decision about the ideal timing of deactivation of software products, a non-trivial activity, and they often lead to counterintuitive results because of the high failure rate.

It is a strategic issue for organizations to systematically assess a priori the impact of different investment scenarios on maintenance, and to evaluate the ability of the software product to remain in operation and still able to be modified and adapted to continue to meet emerging demands.

> More exploration of the impact of project variables such as developer skills, development approach, and user involvement on System Quality is needed to determine the relationships between the project management tasks and the resulting technical quality of the system. Further research exploring the impact of project variables such as IT planning, development approach, project management skills, and domain expertise on the success of resulting systems is warranted. (Petter et al., 2013, p. 43)

Despite the advances made in recent decades in processing capability, hardware cost and the application of scientific rigor and engineering to the software development process, little attention has been given to management aspects (Abdel-Hamid & Madnick, 1989).

The complexity associated with these initiatives is characterized by interactions between technological components, people, and information, and by organizational issues that create a dynamically complex context, containing cycles of feedback, accumulations and temporal delays between cause and effect, and presenting behaviors that are often not trivial and demand non-intuitive solutions (Georgantzas & Katsamakas, 2008).

An area of research that has sought to address this issue and has contributed to a better evaluation of scenarios and prediction of possible impacts of proposals for software process improvements is “Software Process Simulation and Modeling” (SPSM) (Kellner et al., 1999; Ruiz et al., 2004).

The application of modeling and simulation in software processes is a relatively recent development when compared to other areas of research (Kellner et al., 1999), but it has nevertheless attracted interest from researchers (Ali et al., 2014; Zhang et al., 2014). It has been

perceived as an approach and a tool capable of assisting in the analysis of complex business contexts, in the definition and revision of policies, and in carrying out tests and experiments to analyze scenarios that would often be economically unviable or too expensive to be explored in the real world.

The research questions that guide the present research are: Why, even after the start of their operation, do software products require continuous investments in maintenance to maintain sufficient levels of quality attributes? And how do different configurations of resource allocation in maintenance activities influence the variation of these attributes throughout their evolution?

1.2 Objective

The objective of this research is to propose and develop a simulation model (with equations, parameters, and initial conditions) that enables an increase in knowledge, as well as the exploration and evaluation of the impact that different resource allocation policies in maintenance activities have on software systems' evolutionary behavior and their quality attributes related to functionality, maintainability and cost throughout the phases of operation and maintenance.

The proposed model will support the elaboration and evaluation of different maintenance policies that optimize the compromise between technical debt accumulation, investment in different types of maintenance (preventive, perfective, and corrective), and the ability to modify the software product to meet the emerging demands of its users and business.

The construction process of the simulation model aims to broaden the current knowledge about the problem addressed in this research, through the formalization of its structure, the causal relationships between the identified elements, and the analysis of emerging dynamic behavior.

The proposed model is also intended to be a decision-support tool when planning software product maintenance, allowing decision makers both to evaluate and anticipate potential impacts of different investment scenarios and to assess desired quality attributes levels throughout the software product lifetime.

To achieve the main objective of this research, some intermediate results need to be met with the purpose of understanding:

- The cause and effect structure involved in the relationship between the accumulation of technical debt and the maintenance activities of software products.
- How different resource allocation scenarios in maintenance activities influence lifespan.
- How different resource allocation scenarios in maintenance activities influence the behavior over time of quality attributes of software products.
- How the accumulation of technical debt influences the software product's maintenance capacity.
- The main delays in information, decision making, and action in the allocation of resources in maintenance activities throughout the operation and maintenance cycle.
- The decision structure regarding the allocation of resources in maintenance activities.
- The pressures involved in decision making for resource allocation in different maintenance activities.

1.3 Justification

The interest of researchers and practitioners in process modeling and simulating has grown. It has been perceived as an approach that can be used to help the analysis of complex business contexts, to support the design and evaluation of potential intervention policies, and to explore hypothetical scenarios that would often be economically unfeasible to explore in the real world. Although modeling and simulation techniques have long been and widely employed in various disciplines, their adoption in the areas of software development and process improvement has been slow (Kellner et al., 1999).

There are several approaches for building models and performing simulations (Petri nets, agent-based, Monte Carlo, Bayesian networks etc.); however, a literature review exploring studies published between 1998 and 2012 on the application of simulation in the software industry indicated that the predominant approach (accounting for approximately 37% of the reviewed studies) is system dynamics (Ali et al., 2014).

The system dynamics approach was developed in the 1950s, by Jay Forrester (1961) to study complex business problems and was later expanded to study problems associated with the sustainability of population growth in urban centers and throughout the world (Forrester, 1969, 1971). In the mid-1980s, studies applying this approach to study the dynamics associated with software projects began to emerge (Abdel-Hamid, 1984; Abdel-Hamid & Madnick, 1982).

Simulation models of software processes proliferated in the 1990s (Abdel-Hamid & Madnick, 1991; Kellner et al., 1999; Lin et al., 1997; Waeselynck & Pfahl, 1994).

Research studies related to software system projects deal with the development, management and effects of systems on people, organizations, and markets. These projects are socio-technical systems that involve interactions between technical components, people, data, and organizational issues. These interrelationships create a dynamically complex environment containing feedback loops, accumulations and delays between causes and effects, presenting behaviors which are often not trivial, thus requiring non-intuitive solutions by making use of the system dynamics approach suitable for studying how these initiatives evolve over time (Georgantzas & Katsamakas, 2008).

The use of modeling and simulation techniques to achieve the proposed objective of this research is appropriate because it provides a viable way to build knowledge when the cost, risk or logistics of manipulating the real system of interest are prohibitive (Sterman, 2000). It is also appropriate when the complexity of the system being modeled is beyond what other techniques can usefully represent (Kellner et al., 1999).

Although several previous works have employed system dynamics in the context of software systems, there is a predominance of models that explore, as an element of analysis, parts of the software life cycle, especially the software development phase (Franco et al., 2017). Therefore, there is a lack of previous research studies that explore the behaviors associated with the long-term evolution of software systems, services, and organizations (Ali et al., 2014; Kellner et al., 1999; Zhang et al., 2008, 2010).

> Models of emerging and evolving software processes […] are still scarce in the SPS [software process simulation] community despite their popularity among practitioners. Discovering the uncharted territory of dynamic process modeling requires input and critiques not just from model developers but also from prospective simulation consumers and critics. Broader and more integrated perspectives of systems and of enterprises are also required. As software complexity increases, holistic approaches to system development can facilitate better understanding of critical software issues and thus promote better systems. (Zhang et al., 2014, p. 925)

The phenomenon of software product evolution presents a dynamically complex context, which is described by Lehman (1996b) in his formulation of the eighth law as a "feedback system, with multiple levels, multiple loops and multiple agents." Lehman also highlighted the possibility of constructing predictive models of the phenomenon:

> The resultant evolution of software appears to be driven and controlled by human decision, managerial edict, and programmer judgment. […] measures of its evolution display patterns, regularity and trends that suggest an underlying dynamic that may be modeled and used for planning, for process control, and for process improvement. (Lehman, 1980, p. 1067)

In the process of investigating the phenomenon of software evolution, the use of simulation models enables evaluation of how the set of laws dynamically behave and also consideration of the influences they exert on each other (Herraiz et al., 2013).

Two approaches predominate in the models described in previous research studies: black box and white box. The former focuses on observable external behaviors to construct statistical models that fit the empirical data, and seek to predict some metrics (Kemerer & Slaughter, 1999; Turski, 2002; Woodside, 1979). The white box approach seeks to identify and model the different factors that influence evolution, with system dynamics prevailing for the construction of these models (Kahen et al., 2001; Lehman et al., 2002; Wernick & Lehman, 1999).

Previous works applying systems dynamics investigated and replicated part of the phenomena described by the laws of software evolution, and the focus of these works was on the effect that the increasing software complexity has on the capacity of modification and maintenance of the software over time, with the key variables of interest being time, cost, and quality (Chatters et al., 2000; Kahen et al., 2001; Lehman et al., 2002; Wernick & Hall, 2003; Wernick & Lehman, 1999; Zhang et al., 2008). The technical debt effect was also explored, but only in the software development phase (Cao et al., 2010).

Mens et al. (2005) identified the lack of studies that sought to integrate the phenomenon of evolution into the software life cycle in order to increase managerial awareness about its effects and the need to build better predictive models that could help in the long-term management of software systems. It is this research gap that this thesis seeks to fill.

This research study uses the white box approach, more specifically system dynamics, to identify causal factors and relationships that explain dynamic complex emergent behaviors created during the software maintenance phase and stimulated by internal and external influences. This approach is also used to investigate how different policies of resource allocation for maintenance activities influence technical debt management and the software's lifetime.

1.4 Document structure

This thesis is organized into seven chapters, according to the following structure.

1. Introduction: introductory chapter setting out the rationale, objective, and justification of the research.
2. Background: concepts and theories related to software engineering used to formulate the simulation model, with discussion of evolution, maintenance process, quality model, technical debt, sustainability, static analysis, and the "goal, question, metric" method.
3. Material and methods: presentation of the research questions derived from the defined research purpose, the expected specific research objective to be obtained when addressing the research questions, and a brief description the research process that is based on the system dynamics approach and is used for the construction, testing, preparation, and evaluation of resource allocation policies in maintenance activities.
4. The Dynamical Evaluation Framework:
    a. Hierarchical software sustainability evaluation structures: presentation of the two hierarchical structure that were used for evaluating, from technical and economic perspectives, how difference resource allocation scenarios behaved over time.
    b. Proposed simulation model: presentation of the model, constructed according to the approach described in the previous chapter.
5. Discussion: contains the discussions about the obtained results and how they address the proposed research questions.
6. Conclusion: presentation of the conclusions of this work, in light of the results and discussions, as well as the limitations of the research, possibilities for improvement, and opportunities for future work.

2. Background

This chapter presents an overview of the literature related to software evolution, software maintenance, software quality, technical debt, static analysis, software sustainability, and the “Goal, Question, Metric” (GQM) method.

2.1 Software evolution

The proliferation of software systems, the continuous growth in software size, and the need to continually change a software product throughout its operation have resulted a substantial increase in the costs of maintaining software systems in recent decades, when compared to hardware and development costs (Figure 1).

Figure 1. Percentage of effort expended on hardware, development, and maintenance

Source: Adapted from (Deißenböck, 2009)

Regarding the proportion of resources invested in maintenance activities, Lehman (1980) argued that the constant need for change is intrinsic to the nature of software use and that it is embedded in a continually changing environment; thus the investigations regarding high maintenance costs should not focus exclusively on controlling and reducing them. Software products must be constructed so that they can maintain their ability to be modified throughout their life cycle. Economic feasibility assessments should include all associated costs incurred throughout the software lifecycle, not just in the early stages of development.

There is no single, standardized definition of the concept of "software evolution", and this term is often used interchangeably with "software maintenance". Lehman and Ramil (2001) described software evolution as a process of corrections, adaptations and continuous improvements to maintain stakeholder satisfaction with the software's response to changes in domain, needs, and expectations.

Bennet and Rajlich (2000) argued that evolution occurs at a particular stage in the life cycle and only when the initial development has been successful. The goal of this phase is to adapt the software to constant changes in requirements and operating environment, to correct faults, and to respond to the learning of users and developers.

Godfrey and German (2008) proposed semantic differences that distinguish between the terms "maintenance" and "evolution". The former suggests the preservation and resolution of problems and usually represents a set of planned activities carried out in the system, whereas the latter term indicates new projects that have evolved from older ones and is associated with what happens to the software itself throughout its lifetime.

The dynamics associated with the evolution of software products is based on the recognition that any program is "*a model of a model within a theory of a model of abstraction of some portion or of the world or of some universe of discourse*" (Lehman, 1980, p. 1061, emphasis in original). Software, like any other model, contains simplifications and imperfections, and it interacts with and changes the operating environment itself.

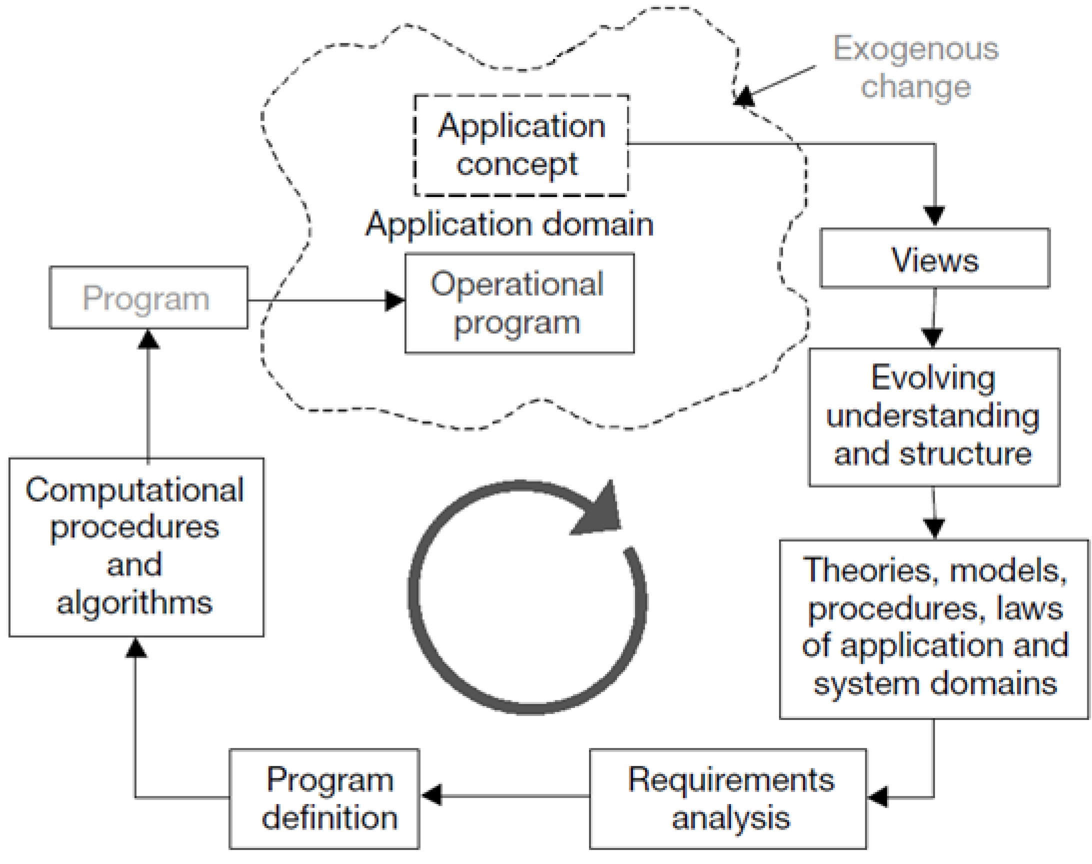
> The installation of the program together with its associated system […] change the very nature of the problem to be solved. *The program has become a part of the world it models*, it is *embedded* in it. Conceptually at least the program as a model contains elements that model itself, the consequences of its execution. (Lehman, 1980, p. 1063, emphasis in original)

Figure 2 schematically shows the interaction of a software program with the external environment, as proposed by Lehman (1980). The activities of analysis, requirements survey, design, and implementation involve extrapolations and predictions of the consequences of the introduction of the software into its operating environment and the potential for it to evolve. These predictions inevitably involve opinions and judgments. Once the software is completed and becomes operational, issues related to its accuracy, adequacy, and satisfaction emerge and inevitably lead to additional pressures for change.

At the same time, as users become familiar with the software—whose design and attributes depend in part on the users' attitudes and practices before software installation—they will modify their behaviors to minimize their efforts or maximize their efficiency, thereby creating more pressure for change.

Moreover, exogenous pressures will also cause changes in the application environment in which the software operates. Examples of such pressures are the introduction of new hardware, new data traffic patterns, changing demands, technological advances, and wider social evolution.

Figure 2. Interaction of a software product with the operating environment.



Source: Adapted from Lehman and Ramil (2006)

These investigations, along with the advances made in recent decades, have given rise to new lines of research associated with software evolution and the consolidation of laws of evolution that describe abstractions of observed behaviors based on statistical models (Lehman, 1980; Lehman & Ramil, 2006).

Table 2 presents the wording of the last revision of the eight laws proposed by Lehman (1996b). Several studies have been carried out to confirm and evaluate their applicability in

different contexts, and although some authors have found that not all of them apply to the free software domain, their application to commercial software has been confirmed by several researchers (Herraiz et al., 2013).

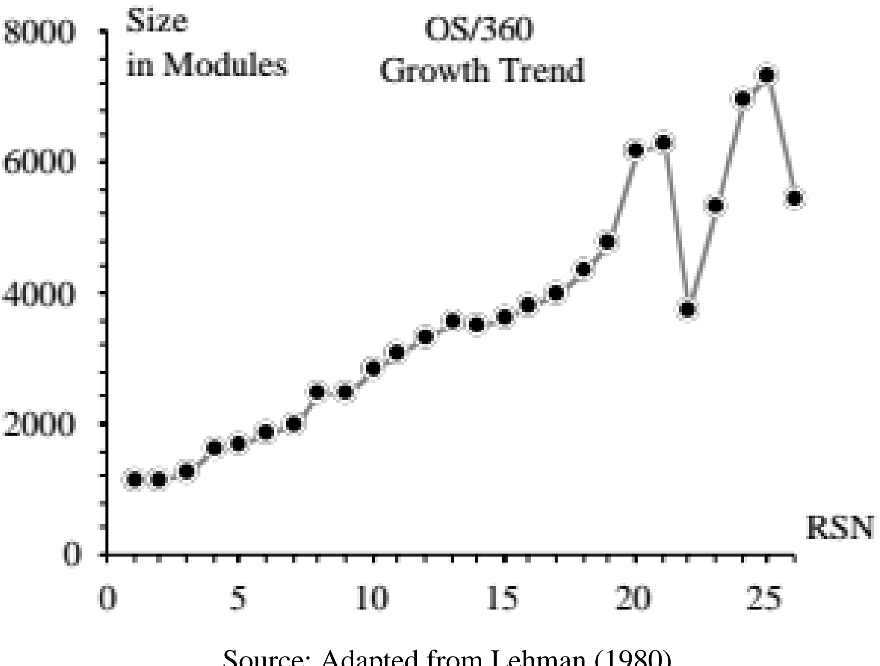
Table 2. Lehman's laws of software evolution

#	Name	Statement
1	Continuing change	An E-type system must be continually adapted, else it becomes progressively less satisfactory in use.
2	Increasing complexity	As an E-type system is changed its complexity increases and becomes more difficult to evolve unless work is done to maintain or reduce the complexity.
3	Self-regulation	Global E-type system evolution is feedback regulated. The program evolution process is self-regulating with close to normal distribution of measure of product and process attributes.
4	Conservation of organizational stability	The work rate of an organization evolving an E-type software system tends to be constant over the operational lifetime of that system or phases of that lifetime. The average effective global activity rate on an evolving system is invariant over the product lifetime.
5	Conservation of familiarity	In general, the incremental growth (growth rate trend) of E-type systems is constrained by the need to maintain familiarity. During the active life of an evolving program, the content of successive releases is statistically invariant.
6	Continuing growth	The functional capability (functional content) of E-type systems must be continually enhanced to maintain user satisfaction over system lifetime.
7	Declining quality	Unless rigorously adapted and evolved to take into account changes in the operational environment, the quality of an E-type system will appear to be declining.
8	Feedback system	E-type evolution processes are multi-level, multi-loop, multi-agent feedback systems and must be treated as such to be successfully modified or improved.

Source: Adapted from Lehman (1996b) and Lehman and Ramil (2006)

Figure 3 presents one of the first observations to capture the essence of the phenomenon of evolution and from which the eight laws described in Table 2 were derived. This figure depicts the growth trend of the IBM OS/360 and the number of modules per release sequence number (RSN), from its birth to the critical moment where it was segregated into two distinct versions of the original product.

Figure 3. IBM OS/360 growth throughout releases



Source: Adapted from Lehman (1980)

The figure captures the regular dynamic nature of the software evolution and characteristics of a feedback system, whose cyclic pattern is characteristic of self-regulated systems. As was observed at the time:

> The ripples on the data are typical of a self-stabilizing process with both positive and negative feedback loops. That is, from a long-range point of view the rate of system growth is self-regulatory, despite the fact that many different causes control the selection of work implemented in each release, with varying budgets, increasing number of users desiring new functions or reporting faults, varying management attitudes towards system enhancement, changing release intervals and improving methodology. (Belady & Lehman, 1972, p. 503)

The instability period observed in Figure 3 from the “20” release represents the split in two development branches of IBM OS/360. The oscillatory pattern indicates the loss of control of the evolution of the system, where the chaotic behavior was triggered by ambitious growth objectives resulting from excessive positive feedbacks that activated self-stabilization processes consisting of negative feedback loops (Lehman, 1996a).

The first research to address the issue of evolution and feedback in software processes originated in 1970 and gained notoriety and interest with the formulation of the hypothesis known as “Feedback, evolution and software technology” (FEAST). This hypothesis portrays the evolution of software as a global process, consisting of a complex feedback system of learning. Therefore, significant improvements can only be achieved when the process is treated

in this way, because stabilization effects restrict the outcome of efforts for this purpose (Lehman & Ramil, 1999).

The FEAST hypothesis was formulated as follows: "As complex feedback systems E-type software processes evolve strong system dynamics and the global stability tendency of other feedback systems" (Lehman et al., 1998). This hypothesis includes three assertions:

1. The software evolution process for E-type systems constitutes a complex feedback system.
2. Where present, feedback is likely to constrain the global benefits derived from forward path changes to the process, however effective they may appear locally.
3. Major improvement requires process innovation to change system dynamics by modification of feedback mechanisms.

Learning and feedback loops play an essential role in determining many of the dynamic aspects at all levels of software evolution. The characteristics and mechanisms of these feedback meshes are also responsible for the process dynamics, as Lehman observed:

> The process may therefore be expected to display the stable behaviour that is the hallmark of feedback systems in general. Externally observable system properties and behaviour remain relatively constant within specified limits over the operational range until instability sets in despite changes in the characteristics of forward path elements, the process environment and the operational environment. (Lehman, 1996a, p. 683)

2.2 Software maintenance

Throughout the operation of a software product, various maintenance interventions are performed to maintain its operational condition and meet emerging demands. ISO/IEC 14764:2006 defines software maintenance as: all the activities necessary to provide cost-effective support for a software system. The activities mentioned correspond to those performed before or after the delivery of the software product.

This norm hierarchically classifies the demands created by changes into two groups: "corrections", or modifications made to a software product after its delivery to correct existing problems; and "enhancements", or modifications made to an existing software product to meet new requirements (Figure 4).

Figure 4. Types of software product maintenance

Source: Adapted from ISO/IEC 14764:2006

The two classifications are further refined into four types of maintenance, defined according to the purpose of the intervention:

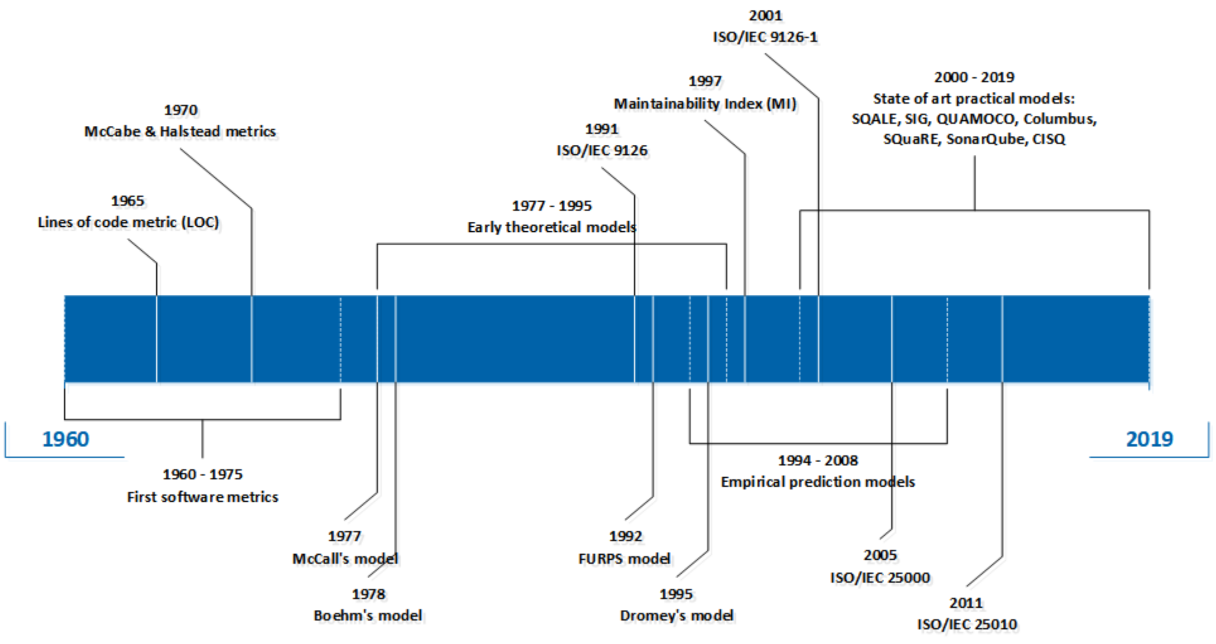
- *Corrective*: reactive changes in the software product, after delivery, to correct identified problems.
- *Preventive*: modifications made after software product delivery to detect and correct latent faults before they become operational faults.
- *Perfective*: modifications made after delivery to detect and correct latent faults in the software product before they manifest themselves as faults. Perfective maintenance improves the performance or maintainability of the software product. This category provides user enhancements, documentation improvements, recoding to improve performance, maintainability, or other attributes of the software.
- *Adaptive*: modifications made after delivery of the software product to keep the product usable in a changed or changing environment. This maintenance category is required to accommodate changes in the environment in which the software operates (for example, operating system upgrade).

Godfrey and German (2008) argued that, according to this categorization, corrective and adaptive maintenance activities do not alter external semantics, whereas perfective maintenance activities, by including a variety of possible changes, and preventive activities, generate improved projects and change external semantics.

2.3 Software quality models

Software product quality models evaluate the aspects of the software product itself. These models measure different types of source code metrics and group them in order to evaluate product quality (e.g., code lines, coupling). These models have undergone a period of intense development and evolution in recent decades (Figure 5).

Figure 5. Evolution history of software product quality measurement models



Source: Adapted from Ferenc et al. (2014)

The first tools to evaluate software product quality were simple, such as code lines, cyclomatic complexity (McCabe, 1976) and the metrics proposed by Halstead (1977). These metrics emerged in the mid-1960s and, with their proliferation, gave rise to the first theoretical models (Boehm et al., 1978; Cavano & McCall, 1978) that were able to hierarchically capture properties of quality of the software product.

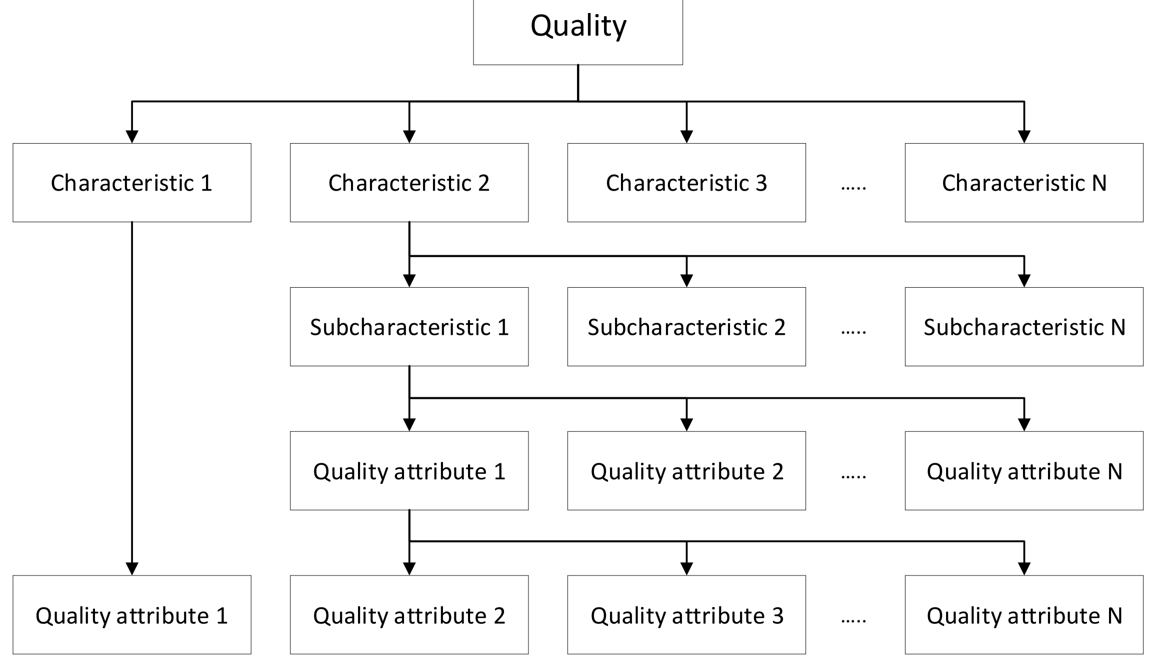
In the 1990s, the theoretical models, that had emerged in the 1970s and 1980s were compiled and gave rise to the ISO/IEC 9126:1991 standard, which was later revised to become the ISO 250nn (System and software Quality Requirements and Evaluation – SQuaRE).

Also, in the 1990's, a set of quality assessment approaches, based on predictive empirical models that were themselves based on regressions, neural networks, and Naive-Bayes classifiers, were developed. One of these widely used models is the "Maintainability Index" (Oman & Hagemeister, 1992).

According to ISO/IEC 25010:2011, system quality can be defined as "the degree to which the system meets the stated and implied needs of its various stakeholders and thus provides value".

These needs are represented in the SQuaRE series by quality models that categorize product quality into characteristics, subcharacteristics, and attributes. The lowest level is composed of attributes, which can be measured by quality element measures (Figure 6).

Figure 6. Hierarchical structure of the SQuaRE quality model



Source: Adapted from ISO/IEC 25010:2011

There are specifications for measurements of three quality dimensions associated with software products in the SQuaRE series: quality in use (ISO/IEC 25010:2011 and ISO/IEC 25022:2016), product quality (ISO/IEC 25010:2011 and ISO/IEC 25023:2016, 2016) and data quality (ISO/IEC 25012:2008 and ISO/IEC 25024:2015).

In this research, only a subset of the categories contained in the second dimension (i.e., product quality) is explored. This decision was taken because the product quality characteristics are the only ones that can be statically measured through the automated inspection of the source code using static analysis tools (described in Section "2.6").

The software product quality model's characteristics and attributes are depicted in Figure 7. The model organizes the attributes into eight categories, which in turn are composed of a set of related subcategories.

Figure 7. Software product quality mode

System/Software Product Quality

Functional Suitability	Performance Efficiency	Compatibility	Usability	Reliability	Security	Maintainability	Portability
Functional completeness Functional correctness Functional appropriateness	Time-behavior Resource utilization	Co-existence Interoperability	Appropriateness recognition Learnability Operability User error protection User interface aesthetic Accessibility	Maturity Availability Fault tolerance Recoverability	Confidentiality Integrity Non-repudiation Accountability Authenticity	Modularity Reusability Analyzability Modifiability Testability	Adaptability Installability Replaceability

Source: Adapted from ISO/IEC 25010:2011

The ISO/IEC 25010:2011 standard descriptions of the subset composed of five characteristics from the software product quality model, and which are of interest for this research, are as follows:

- *Functional suitability*: the degree to which the set of functions covers all the specified tasks and user objectives.
- *Performance efficiency*: the performance relative to the amount of resources used under stated conditions, which may include other software products, the software and hardware configuration of the system, and materials.
- *Reliability*: the degree to which a system or component performs specified functions under specified conditions for a specified period of time (limitation in reliability are due to faults in requirements, design, and implementations).
- *Security*: the degree to which information and data are protected so that unauthorized persons or systems cannot read or modify them, and authorized persons or systems are not denied access to them.
- *Maintainability*: the degree of effectiveness and efficiency with which the product can be modified. Modifications can include corrections, improvement or adaptation of the software to changes in environment, and modifications to requirements and functional specifications.

Since 2000, several practical models were proposed to address the complexity and the low level of details for the implementation of the ISO/IEC 250nn standard, as well as to overcome the lack of clarity in the interpretation and traceability of the standard (Bakota et al., 2011; Heitlager et al., 2007; Letouzey & Coq, 2010; Mordal et al., 2013; Wagner et al., 2012).

This set of models defines a set of concrete metrics at the level of the source code and algorithm and, from the results obtained through the inspection of the elements of analysis (source codes), these practical models aggregate the results obtained at higher levels following the hierarchical structure established by the ISO/IEC 250nn (see Figure 6).

2.4 Technical debt

The “technical debt” metaphor was coined by Cunningham (1993) and refers to the long-term cost associated both with shortcuts taken throughout software development and with maintenance by programmers to deliver short-term benefits to the business.

> Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unconsolidated implementation, object-oriented or otherwise. (Cunningham, 1993, p. 30)

Despite the simple and intuitive definition, the metaphor has been used indiscriminately to describe any kind of impediment, friction and obstacle affecting the sale, development, deployment, maintenance or evolution of software-based systems. This wider use has weakened and diluted the meaning of the metaphor (Kruchten et al., 2012).

The technical debt metaphor is composed of a set of comprehensive constructs that help to communicate the costs and risks relating to the low structural quality of a software program that remain in a software after it has been released. Its definition can be partitioned into the following elements (Curtis et al., 2012):

- *Should-fix violations*: Violations of good architectural or coding practices, which are known to have an unacceptable probability of contributing to severe operational problems (interruptions, security breaches, corrupting data, etc.) or to the excessive cost of acquisition, such as excessive effort to implement change.
- *Principal*: The necessary cost of remediating should-fix violations in production software code.
- *Interest*: The continuous cost attributable to should-fix violations in the production code that have not been remediated, such as extra hours of maintenance required to carry out activities and inefficient use of resources.

- *Technical debt*: The future costs attributable to known violations of the software in operation that should be fixed; technical debt should include both principal and interest.

The Consortium for IT Software Quality (CISQ)[1] later introduced a complementary concept to the technical debt metaphor that captured the business risks, which consisted of two elements (OMG, 2018):

- *Liability from debt*: The costs to the business resulting from operational problems caused by the flaws in the production code. These flaws include both should-fix problems included in the calculation of the technical debt as well as problems not listed as should-fix because their risk was underestimated.
- *Opportunity cost*: The benefits such as revenue from new features that could have been achieved had resources been committed to developing new functionalities rather than being assigned to paying technical debt. The opportunity costs represent the tradeoff that decision makers must weigh when deciding how much effort to devote to paying technical debt.

There is no consensus regarding what can be classified as technical debt, what level of impairment of software quality attributes for the violations can be classified as such, and what the limits of the metaphor's use are. However, a systematic mapping of the literature was conducted to evaluate the use of the metaphor, according to which ten categories were identified: requirements, architecture, design, code, tests, build, documentation, infrastructure, versioning and defects (Li et al., 2015).

Technical debt can be beneficial or detrimental to the management of operation and maintenance of software products. Violations that are intentionally incurred to obtain short-term benefits can be positive if the associated costs are kept visible and under control (Allman, 2012). On the other hand, they can occur unintentionally and not be perceived by those involved. If they remain invisible and unresolved, they can accumulate and pose risks to long-term maintenance and evolution activities (Li et al., 2015).

Violations of good practice and the accumulation of technical debt compromise quality attributes. Li et al. (2015) surveyed works related to technical debt and identified that the existing literature indicates that only attributes related to product quality are affected (quality

[1] http://www.cisq-it.org

attributes are only indirectly affected). Most papers claim that technical debt negatively affects the maintainability of software products, while other features and sub-features are reported only in a small number of cases.

By negatively impacting the maintainability of the software product, the accumulation of technical debt negatively influences the adaptability and evolution of the software in operation.

Technical debt management consists of a set of activities that seek to prevent potential violations from being incurred or to treat existing debt to keep it at a reasonable level. It encompasses the activities of identification, measurement, prioritization, prevention, monitoring, payment, representation, and communication.

To avoid the long-term effects of technical debt accumulation on software maintenance ability, debt needs to be paid at some point in the lifecycle. The most commonly used form of payment is the refactoring technique, which involves making changes to the source code, design or architecture of a software without changing its external behavior to improve its internal quality (Fowler et al., 1999).

Despite the growing interest among the scientific community and software engineering practitioners, technical debt management still lacks tool support for accurately managing sources of debt. The metaphor remains an abstract concept and lacks the details for delineating the source of technical debt from its causes and consequences (Ernst et al., 2015).

In addition to the introduction to the technical debt metaphor presented above, further details are elaborated in the proposed model formulation (Section 4.2).

2.5 Software sustainability

Software sustainability has recently began to attract the attention of researchers and practitioners (Becker et al., 2015; Venters et al., 2014). Although there is still no consensus about its definition, the general rationale of software sustainability is that it is related to the software's capacity to endure and to meet current needs without compromising future needs (i.e., software survival goals). The "Karlskrona Manifesto" was proposed in 2015 (Becker et al., 2015), and it lists some of the principles of software's sustainability:

- It is a systemic property and systems thinking has to be the starting point for the transdisciplinary common ground.

- It has multiple dimensions that have to be taken into account to understand its nature (the whole is more than the sum of its parts).
- It requires action on multiple levels and different systems may have different interventions leverage (counter-intuitive behaviors).
- It requires long-term thinking, and the benefits and impacts of any interventions should be evaluated at the outset (cause and effect are not closely related in time and space).
- There are different orders of effects, since the consequence of actions plays out over multiple timescales, and their cumulative impact may be irreversible (nonlinear effects).

Sustainability is seen as a systemic property that encompasses multiple dimensions, which are shown in Table 3.

Table 3. Software sustainability dimensions

Dimension	**Covers**
Individual	Individual freedom and agency (the ability to act in an environment), human dignity, and fulfillment.
Social	Relationships between individuals and groups. For example, it covers the structures of mutual trust and communication in a social system and the balance between conflicting interests.
Economic	Financial aspects and business value. It includes capital growth and liquidity, investment questions, and financial operations.
Technical	The ability to maintain and evolve artificial systems (such as software) over time. It refers to maintenance and evolution, resilience, and the ease of system transitions.
Environmental	The use and stewardship of natural resources.

Source: Adapted from Becker et al. (2015)

Venters et al. (2014) suggested that software sustainability is also an emergent property that cannot be attributed to any particular part of the system; rather, it emerges once its components have been integrated. Thus, they argued that sustainability “cannot be designed or engineered and quantified until after the software system is operational” (Venters et al., 2014, p. 5).

There is no quantifiable and objective way to measure sustainability directly. Its evaluation varies according to the context, the stakeholder perspective and the point in the life cycle at which it is assessed. One way to evaluate technical and economic sustainability is to use the software product’s quality models and source code properties as proxies to assess them indirectly.

2.6 Static analysis

Software quality metrics and technical debt estimates are usually made using static analysis tools. There are several tools for automating the identification of predefined types of anomalies by scanning and parsing the source code while looking for a fixed set of patterns. Some of these tools are commercial, and others are free or even open source.

Many tools automate the process of scanning a software source code and parsing it to detected predefined quality rules to gather quality measurement data (e.g., SonarQube[2], PMD[3], Checkstyle[4], FindBugs[5], Cobertura[6], Jacoco[7], among others). These tools provide an API for extracting time series data for the different measures, which are necessary for calibrating the proposed simulation model, as well as the list of detected violations, which is necessary for consolidating the low level metrics into the proposed CISQ's standards (OMG, 2016a, 2016c, 2016d, 2016b).

After each version of the software's source code is scanned, the data are extracted from the scanning tool, and the identified violations are assigned to one of the software product's characteristics, according to the SQuaRE model, and to one of the proposed CISQ's static analysis rules. The computation for calculating each of the base normative measures is to count the frequency of each type of violation and sum them by categories:

$$Base\ Measure = \sum_{i=1}^{n} (detected\ violation_i)$$

To obtain the functional density of violations for each type of quality characteristics, the base measure is divided by the size of the software:

$$Functional\ density\ violation = \frac{Base\ measure}{Size\ of\ software}$$

[2] http://www.sonarqube.org

[3] https://pmd.github.io

[4] http://checkstyle.sourceforge.net

[5] http://findbugs.sourceforge.net

[6] http://cobertura.github.io/cobertura/

[7] http://www.eclemma.org/jacoco/

The CISQ standards also offer some alternative weighted informative measures, one of which is useful when estimating the technical debt of a software system. For each of the quality characteristics, the standard suggests weighting each of the identified violation element by its estimated effort to fix it. Then, we sum all of them to assess information regarding total cost of ownership and to estimate future maintenance costs and effort.

$$\text{Overral maintanence effort} = \sum (detected\ violation_i \cdot estimated\ effort\ to\ fix)$$

2.7 Goal, Question, Metric method

The "Goal, Question, Metric" (GQM) method (Basili et al., 2002; Basili & Weiss, 1984) is based on a hierarchical structure capable of creating a purposeful measurement model that targets a particular set of issues and has a set of rules for interpreting the measurement data.

To achieve this, it must:

- Clearly specify goals (conceptual level)
- Establish how each goal would be evaluated by defining questions for each of them (operational level)
- Trace each of these questions to the data that must be interpreted in order to answer them individually (quantitative level).

Figure 8. GQM method's hierarchical evaluation structure

Source: Adapted from Basili et al. (2002)

The resulting measurement model has three levels, which are shown in Figure 8. Basili et al. (2002) provided the following definitions for each of these levels:

- *Conceptual level (goal)*: a goal is defined for an object, for a variety of reasons, concerning various models of quality, from various points of view, relative to a particular environment. Objects of measurement are products (artifacts, deliverables, and documents that are produced during the system life cycle), process (software-related activities), and resources (items used by a process to produce their outputs).
- *Operational level (question)*: a set of questions is used to characterize the way the assessment/achievement of a specific goal is going to be performed based on some characterizing model. Questions try to characterize the object of measurement (product, process, resource) concerning a selected quality issue and to determine its quality from the selected viewpoint.
- *Qualitative level (metric)*: a set of data is associated with every question in order to answer it in a quantitative way (objectively or subjectively).

Basili et al. (2002) argued that the process of setting goals is critical to successful application of the GQM method. Each goal has four elements:

- *Purpose*: Why is the object being examined?
- *Issue*: Which attribute of the object is being examined?
- *Object*: What is being examined?
- *Viewpoint*: From which point is the object being examined?

The development of a goal is based on three sources of information: (a) policy and strategy of the organization, where the purpose and the issue of the goal are derived; (b) the description of the process and products of the organization, where the description of the model of the object of interest is defined; and (c) the model of the organization, which provides the viewpoint on the goal.

Table 4 presents a complete example of an GQM model structure developed following the described method.

Table 4. Goal/Question/Metric model example

Goal	*Purpose*	Improve
	Issue	the timeliness of
	Object	change request processing
	Viewpoint	from the project manager’s viewpoint

Question 1	What is the current change request processing speed?
Metrics	• Average cycle time • Standard deviation • % cases outside of upper limit
Question 2	Is the performance of the process improving?
Metrics	• $\frac{Current\ average\ cycle\ time}{Baseline\ average\ cycle\ time} \cdot 100$ • Subjective rating of manager's satisfaction

Source: Adapted from Basili et al. (2002)

2.8 Chapter summary

This section summarizes the content presented in this chapter, with the purpose of organizing the main concepts adopted in the formulation of the proposed model.

The relevant concepts are:

- *Software evolution*: The laws of software evolution describe statistical models of behavior observed in software products throughout their life cycle. These behaviors are incorporated in the formulation of the proposed model. Examples of the behaviors are continuous growth, decreasing quality, increasing complexity, and continuous changes.
- *Software maintenance*: This section aimed to typify the activities involved in the process of maintaining software products. The segregation of the maintenance process into perfective, preventive, corrective and adaptive activities supports the establishment of the configuration of resource allocation. It is influence of resource allocation on quality attributes and the accumulation of technical debt that this thesis seeks to evaluate.
- *Software quality models*: The software product quality model establishes the quality attributes that serve as evaluation constructs for the different resource allocation configurations in the maintenance process. This thesis evaluates the characteristics of functionality and maintainability are evaluated.
- *Technical debt*: This concept is used to represent accumulations and delays in the proposed model. The "principal" component of technical debt represents the accumulation of violations that occur during the execution of maintenance activities. This accumulation over time reduces the maintainability of the software process and

the ability to deliver new features (i.e., interest), phenomena also described in the laws of software evolution.

- *Software sustainability*: Sustainability is a theme of increasing interest in many fields, including in the software engineering area. This thesis explores two software sustainability dimensions (economic and technical) for constructing the hierarchical evaluation policy structure.
- *Static analysis*: This is a technique and a set of tools that can be used to automate data extraction and collection based on a software source code, which can be used to support the estimation of technical debt.
- *Goal, Question, Metric method*: This is a method for creating a hierarchical measurement structure. It is used to establish the measurement structure for evaluating maintenance resource allocation policies based on the simulation model's output data.

3. Materials and methods

This chapter presents the materials and methods that were employed in order to achieve the specific purpose of this research, which was: to propose and develop a simulation model (with equations, parameters and initial conditions) that enables an increase in knowledge, as well as the exploration and evaluation of the impact that different resource allocation policies in maintenance activities have on the technical and economic sustainability of software systems' evolutionary behavior and on their quality attributes related to functionality, maintainability and cost throughout the phases of operation and maintenance.

3.1 Research questions

In order to achieve the proposed research purpose, three research questions (RQ) were defined. They are as follow:

- **RQ1**: How should the dynamical behavior of a software product's quality attributes, due to maintenance activities, be characterized throughout its evolution?
- **RQ2**: How do different resource allocation policies in software maintenance activities affect the dynamical behaviors of these quality attributes?
- **RQ3**: How should the resource allocation in maintenance activities be managed to improve the technical and economic sustainability of a software product?

3.2 Research objectives

For each research question, a specific research objective was defined. Each objective detailed the expected key research outcomes for the three research questions previously defined.

a) *Explain how the observed behavioral characteristics of software maintenance emerges as a result of the system structure*: based on an extensive literature review an initial hypothesis was proposed, which identified how key elements and their relationships influence each other, detailing the equations for describing their interaction and the initial conditions that account for the observed dynamical behaviors. This initial hypothesis was formulated with textual explanations, causal loops diagrams, and stock and flow diagrams, and time series plots (behavior over time). The outcome of this objective is "as is"

hypothetical description of the system under investigation, endogenously explaining how and why the observed behavior emerged. This description is given in Section 4.2.1 (“Problem articulation and dynamical hypothesis”).

b) *Identify and describe the impact that different resources allocation policies in software maintenance activities have on a software product’s quality attributes and how they influence the software’s long-term technical and economic sustainability*: a review was undertaken of how different hypothetical resource allocation policies affect the quality characteristics of software system along the software system’s lifetime, how they impact on technical and economic sustainability, and how different outcomes emerge from the inner elements of the system being modeled. The outcomes and discussion of this objective are presented in Section 5 (“Results and discussion”).

c) *Propose and evaluate alternative resource allocation policies*: based on specific contexts and selected evolutionary conditions, resource allocation policies that help to improve the software product’s technical and economic sustainability were specified. This objective generated suggested resource allocation policies that can be better suited to different goals of different software systems and contexts (see Section 5, “Results and discussion”).

3.3 Research process

In order to achieve the objectives proposed in this work, a model was constructed that was based on the dynamic system approach and used the Vensim Professional software (Ventana System, 2018). This enabled the model to be simulated and helped in conducting tests, the evaluation of the proposed dynamical hypothesis, and the analysis of hypothetical resource allocation policies scenarios.

The modeling process was carried out according to the methodology proposed by Sterman (2000), the system dynamics approach. This approach is described in the Section 3.3.1 (“System dynamics”), and more details are available in Appendix A (“Appendix A – System dynamics ”).

Once a model is formulated, the test stage begins to build confidence in the model. Validation or verification to establish the truth and accuracy of the model or to derive correct conclusions based on assumptions is impracticable (Sterman, 2000). Instead, the testing stage seeks to establish confidence in the model’s representativeness of the real world (Forrester &

Senge, 1980). Models should focus on solving specific problems and being useful for broadening the understanding of the problem and for efficiently intervening in the real-world context.

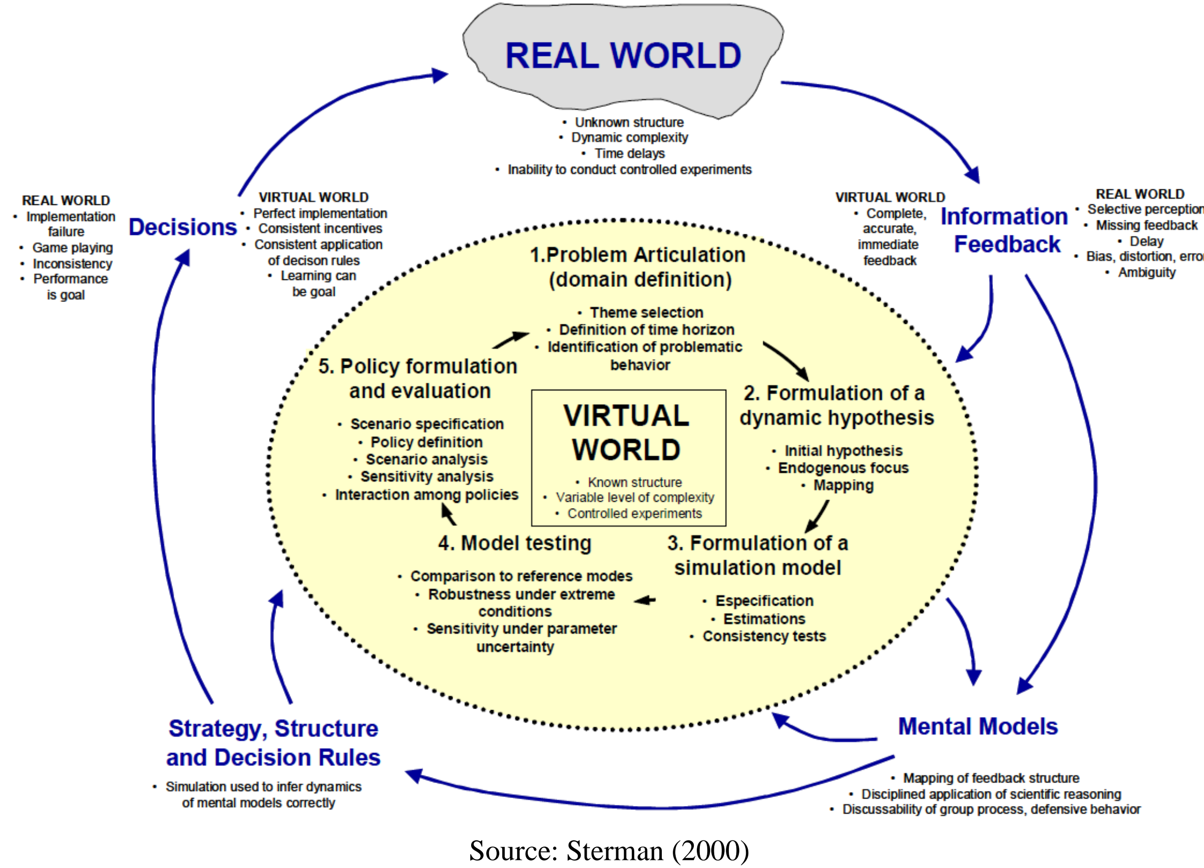
This research study took place in three distinct phases, which were as follows:

1. *Literature review*: an extensive and systematic literature review was carried out to identify previous work that could contribute to the proposed research, to identify theories and elements that could be used as starting points or incorporated into the proposed model, and to justify modeling decisions taken during the model development. Some of the results obtained from this phase have already been published (Franco et al., 2017).
2. *Model construction*: the model formulation was conducted according to the system dynamics approach. It was based on an extensive literature review, to identify elements and descriptions of dynamic behaviors, which served as the basis to the definition of the dynamical hypothesis tested in this thesis. Secondary data collection was used in the construction of the plausible preliminary version of the model that captures the defined dynamic hypothesis.
3. *Experimentation*: finally, the model was used to conduct experiments on a set of managerial practices and policies for the maintenance and operation of software products. The expected results of this phase were: to identify dysfunctional consequences of current practices adopted; to enable the formulation of new proposals for allocation of resources in software maintenance activities; and to generate new knowledge associated with the phenomenon of software evolution during operation and maintenance phases of software systems.

3.3.1 System dynamics

The maintenance process was modeled using the system dynamics approach (Sterman, 2000), which consists of an iterative approach consisting of five stages: problem articulation; the definition of dynamic hypotheses; formulation; testing; and policy formulation and evaluation (see Figure 9).

Figure 9. System dynamics' iterative modeling process



Source: Sterman (2000)

The objectives and activities performed in each of these stages are briefly described in the following sections. The approach consists of an iterative and non-linear process, where results and knowledge acquired throughout the process can be used to revise assumptions and conclusions from previous steps to refine the model (Sterman, 2000).

3.3.1.1 Problem articulation

For a model to be useful, it needs to solve a specific problem and it needs to simplify rather than to reproduce the complete system under investigation in details. It should simplify the real-world to allow understanding of the research context and problem. The articulation of the problem establishes the purpose of the model, which serves as a criterion for defining the elements that must be included and for excluding those that can be ignored, so that the goal of the model is reached (Sterman, 2000).

Two of the most common tools for this purpose are:

- *Reference modes*: problems are characterized dynamically, that is, patterns of behavior unfold over time, demonstrating how the problem originated and how it can evolve over time.

- *Time horizon*: this tool involves looking back in time and describing the symptoms of the problem until the reasons for the problem's emergence can be identify. At the same time, this tool must also involve and extension into the future to capture the indirect and delayed effects of potential intervention policies.

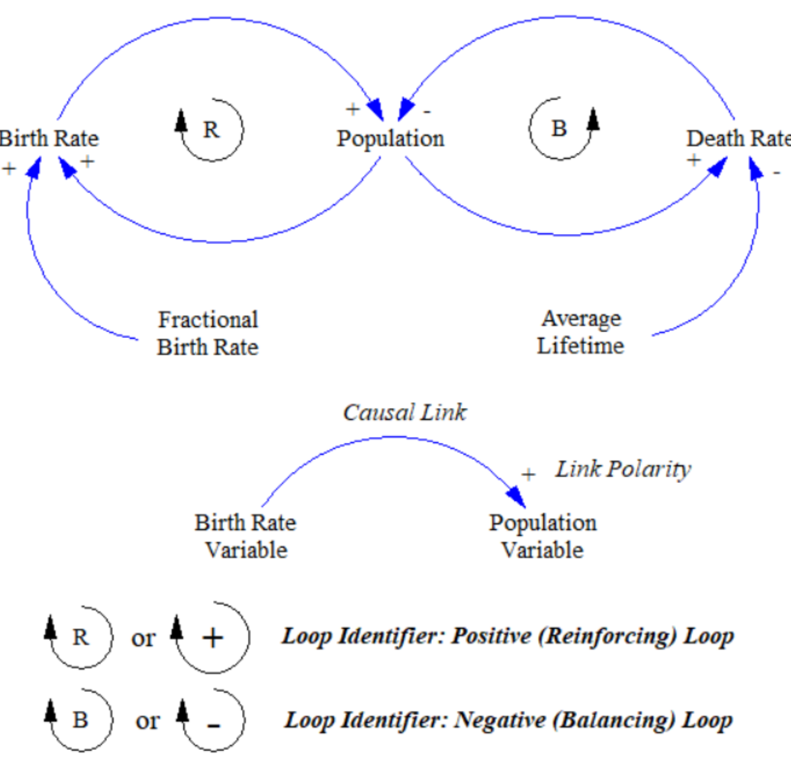
3.3.1.2 Dynamic hypothesis

The dynamic hypothesis corresponds to a developing theory that seeks to explain problematic behavior, and how it originated. It serves as a guide for modeling the inner structures of the system involved. The focus is to construct a theory that endogenously justifies and reproduces the dynamics observed through the interaction of the variables and agents represented in the model (Sterman, 2000).

The system dynamics approach provides various tools to represent the boundaries and the inner causal structures of the model. In this work three specific tools were used:

- *Model boundary chart*: summarizes the concepts included in the model endogenously (i.e., that affect the model and are affected by it) or exogenous (i.e., that affect the model but are not affected by it), and those that were excluded (i.e., relevant aspects, but which are not part of the model's focus). When listing items that have been removed, the model limitations and results exceptions are stated explicitly.
- *Subsystem diagram*: presents the general architecture of the model constituted by the main subsystems. Each subsystem is presented in conjunction with the resource, information, and request flows that coupled with the subsystems. The subsystems convey information about the boundary and aggregation level of the model, presenting the number and types of organizations and the different agents represented, and they establish a channel of communication between the mental and formal model (Morecroft, 1982).
- *Causal loop diagram*: corresponds to the mapping of the causal relations between the variables, identifying how they affect each other. This type of diagram constitutes an important tool for representing feedback loops of systems of any domain. The diagram consists of nodes (variables) and their relationships (arrows), where relationships can be positive or negative (indicated by the corresponding symbol at the end of an arrow). Figure 10 shows examples of the elements that make up the causal diagram.

Figure 10. Examples of the elements of a causal loop diagram



Source: Adapted from Sterman (2000)

- *Stock and flow maps*: emphasize the physical structure of the model. These diagrams represent the accumulations of materials, financial resources, and information as they move through the system. The flows represent the rates of increases or decreases in inventories. Inventories characterize system states and generate the information on which decisions are made. Decisions, in turn, change the rates of flows and the stocks, and they close the feedback loop structures of the system.

3.3.1.3 Model formulation

The formulation of a simulation model enables experiments to be performed that would not be possible in certain situations (due to economic or ethical issues). A simulation model also allows experiments to be conducted in a virtual environment facilitating the identification of flaws in the designed dynamic hypothesis.

This step consists of transposing the conceptual domain of the diagrams into a fully specified model, with equations, parameters, and initial conditions. In this stage the following activities are conducted:

- Specification of the structure and decision rules of the model.
- Estimation of parameters, behavioral relationships and initial conditions.

- Tests to assess the consistency of the model for the research purpose and to define the model's boundaries.

3.3.1.4 Testing

Once the model is formulated, the test stage begins, and confidence is built in the developed model. Validation or verification to establish the model as a truth, and accuracy or to derive correct conclusions based on assumptions is impracticable (Sterman, 2000). Instead, tests seek to establish confidence in the representativeness of the model, which is gradually accumulated as it undergoes new tests and as new points of correspondence with the empirical reality are identified (Forrester & Senge, 1980).

Sterman argued that:

> all models, mental or formal, are limited, and correspond to simplified representations of the real world. They differ from reality in small and large forms, but in infinite numbers. The only statements that can be validated – shown to be true – are pure analytic statements, propositions derived from the axioms of a closed logical system (Sterman, 2000, p. 846)

Forrester also recognized the impossibility of validation in the sense of establishing truth, where he wrote:

> Any "objective" model-validation procedure rests eventually at some lower level on a judgment or faith that either the procedure or its goals are acceptable without objective truth. (Forrester, 1961, p. 123)

Barlas later pointed out the reasons for the impossibility of using statistical techniques to validate the systems dynamics models:

> Validation of a system dynamics model is much more complicated […] because judging the validity of the internal structure of a model is very problematic, both *philosophically* and *technically*. It is philosophically difficult, because […] the problem is directly related to the unresolved philosophical issue of verifying the truth of a (scientific) statement. And the problem is technically difficult, because there are no established formal tests (such as statistical hypothesis tests) that one can use in deciding if the structure of a given model is close enough to the "real" structure. Furthermore, standard statistical tests cannot even be used in validating the behavior of a system dynamics model, because of problems of autocorrelations and multicollinearity. (Barlas, 1996, p. 186, emphasis in original)

Barlas further justified why most statistical tests are not practical when validating system dynamics models:

> The problems involved in using statistical significance tests in validating system dynamics (and other socio-economic) models are both *technical* and *philosophical*. The technical reasons why statistical significance has little relevance in model validation have to do with some fundamental assumptions that must hold for statistical tests to be valid. Most statistical methods presume, at least, that the data are (i) serially independent (not autocorrelated); (ii) not cross-correlated; (iii) normally distributed. The first two of these assumptions are almost never met by systems dynamics models. Data generated by systems dynamics models are autocorrelated and intercorrelated by their very nature. (Barlas, 1996, p. 196, emphasis in original)

Given this premise, authors have suggested that the focus of the tests should be on assessing the usefulness of the model and identifying its shortcomings via a systematic process of experiments and empirical tests designed to refute the proposed dynamic hypothesis. If it is empirically invalidated, it must be discarded and replaced by a new and more precise theory. Otherwise, it must be conditionally accepted until it is improved or refuted (Forrester & Senge, 1980; Sterman, 2000).

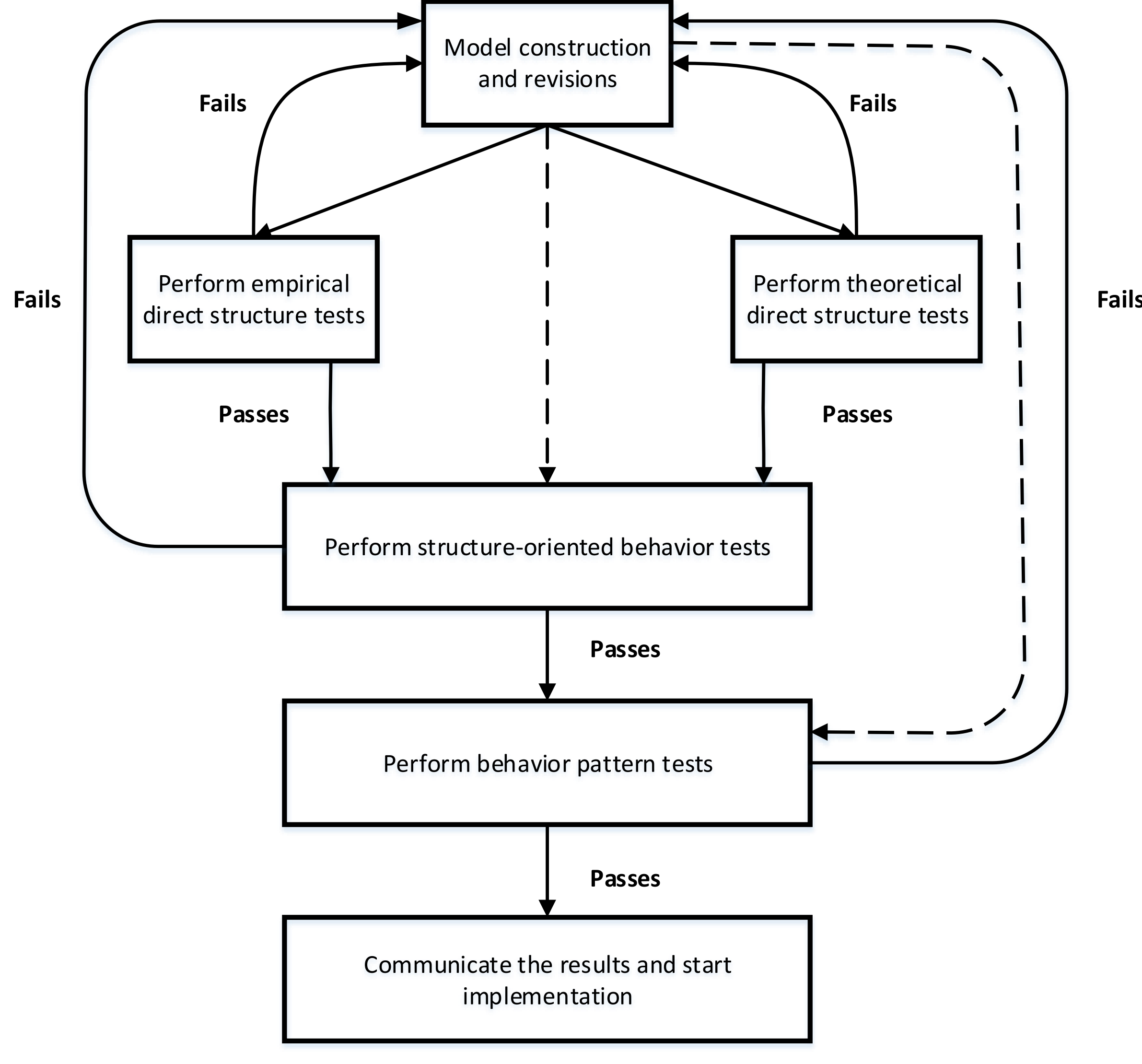
Among the tests to which the model was submitted, the following stand out:

- *Border adequacy test* assesses whether concepts important for addressing the problem are endogenous, whether the behavior is significantly altered when boundary assumptions are relaxed, and whether recommended policies are changed when the boundary is expanded;
- *Structure evaluation test* verifies whether the structure is consistent with the relevant descriptive knowledge of the system if the level of aggregation is adequate, whether the model complies with fundamental laws of physics (e.g., conservation of matter), and whether the rules of decision capture the behavior of the system actors.
- *Dimensional consistency test* verifies whether the dimensionality of each equation is consistent without having to use conversion parameters that have no meaning in the real world;
- *Parameter evaluation test* certifies that the parameter values are consistent with the relevant descriptive and numerical knowledge of the system and that each parameter has a counterpart in the real world.
- *Extreme condition test* submits each model equation to input data representing extreme conditions to verify whether the equations still make sense and whether the model responds plausibly when subjected to extreme policies, conditions and parameters;
- *Integration error test* evaluates whether the obtained results are sensitive to the time-step choices and the numerical integration method.

- *Behavior reproduction test* certifies whether the model reproduces the behavior of interest and other phenomena (qualitative and quantitative) observed in the system, and whether the relationship of phase and frequency between the variables are in agreement with the data;
- *Behavior anomaly test* changes or removes assumptions from the model to assess whether anomalous behavior occurs.
- *Familiarity test* evaluates whether the model can reproduce behaviors observed in other instances of the same system.
- *Emerging behavior test* verifies whether the model produces behaviors not observed or not previously recognized and whether it consistently anticipates a system response to the new conditions.
- *Sensitivity analysis* tests the numerical, behavioral, and political sensitivity of the model when assumptions about model parameters, boundaries, and aggregations are varied along plausible ranges of uncertainties.
- *System improvement test* evaluates whether the model has helped change the system to a better stage.

Barlas (1996) argued that the ultimate objective of system dynamics model test is to establish the appropriateness of the model's structure. He suggested that the logical order of evaluation is first to test the appropriateness of the model's structure, and then to start testing the model's output behavior accuracy (see Figure 11).

Figure 11. Logical sequence of formal steps of model evaluation



Source: Adapter from Barlas (1996)

Each of the steps proposed in Figure 11 are aimed to evaluate specific aspects of a system dynamics model (Barlas, 1996):

- *Direct structure tests* take each relationship (mathematical equation or any form of logical relationship) individually and compare it with the knowledge available without simulating the model.
    - *Empirical direct structure tests* compare the model structure with quantitative or qualitative information obtained directly from the real system being modelled (i.e., structure evaluation and parameter evaluation tests).
    - *Theoretical direct structure tests* compare the model structure with generalized knowledge about the system available in the literature (i.e.,

dimensional consistency, integration error, extreme condition, structure evaluation and parameter evaluation tests).

- *Structure-oriented behavior tests* assess the validity of the structure indirectly, by applying certain behavior tests on the generated model's behavior patterns obtained from the partial or complete model's simulation output (i.e., border adequacy, extreme conditions, sensitivity analysis, behavior reproduction, and familiarity tests). These are strong behavior tests that can provide information on potential structural flaws.
- *Behavior pattern test* are deployed once enough confidence has been built in the model structure, to measure how accurately the model can reproduce the major behavior patterns exhibited by the real system (i.e., behavior anomaly, emerging, and improvement tests). Barlas (1996) indicated that the emphasis is on pattern prediction (periods, frequency, trends, phase lags, and amplitude) rather than point (event) prediction.

3.3.1.5 Policy formulation and evaluation

According to Sterman (2000), this step is much broader than just changing the parameter values of a model. The policy formulation stage includes the creation of new strategies, structures, and decision rules. As the feedback structure of a system determines its dynamic behavior, its leverage points often involve changing dominant meshes by changing stock and flow structures, eliminating delays, changing the flow and quality of information available at decision points, or even recreating decision-making processes.

3.4 Chapter summary

This section summarizes the content presented in this chapter with the purpose of organizing the decisions taken regarding the methodological approach to achieve the formulated objective.

The relevant content of this chapter has been:

- *Research questions*: based on the research purpose of this thesis, three research questions were elaborated that guide the research to achieve the intended result.
- *Research objectives*: for each of the three research questions, a specific research objective was defined. Each objective describes the expected research outcomes of the research question.

- *Research process*: this section of the chapter presented the research process adopted for answering each of the proposed research questions and attaining the specific purpose of the current study. In addition to the research question, an overview of the system dynamics, which constitutes to a central approach in the research process used in this study, was provided (a more detailed overview can be found in "Appendix A").

4. The Dynamical Evaluation Framework

This chapter details the proposed *Dynamical Evaluation Framework* for evaluating the impact that different resource allocation policies for maintenance activities have on the software system's evolutionary patterns over time in relation to the system's technical sustainability (based on the software quality characteristics, previously discussed in Section 2.3, "Software quality models") and economic sustainability (based on the technical debt metaphor, previously discussed in Section 2.4, "Technical debt").

The proposed framework is informed by the theories presented in Chapter 2 ("Background") and the materials and methods presented in Chapter 3 ("Materials and methods"). Figure 12 shows how the previously discussed topics interrelate and how they are arranged together to build the proposed GQM method's hierarchical evaluation structure (to be presented and discussed in Section 4.1, "Hierarchical software sustainability evaluation structure").

Figure 12. Theoretical elements of the proposed framework

Economic sustainability	Technical sustainability	GOAL
Technical debt	Quality attributes (ISO/IEC25000 series)	QUESTION
Principal, asset, and interest	Maintainability, Reliability, and Functional suitability	METRIC
Static analysis & simulation output		DATA

Source: Author

Figure 12 also presents the theoretical elements of the proposed evaluation framework that were built upon the following levels of abstraction and:

- *Goal level* contains two trade-off goals dimensions that must be considered in order to assess the evolutionary outcomes resulting from different resource allocation policies for maintenance activities: economic and technical sustainability. Usually,

they represent conflicting interests. Taken to the extreme, building the perfect quality software system (without any kind of violations and thus no technical debt) will demand resources that are usually not available, or the time constraint for delivering the software increases the likelihood of defects flowing into production environment. Moreover, Lehman (1989) also argued that all software has latent defects and that it is a matter of time for them to become known to users; thus, there is no zero-defect software, but it is possible to minimize the defects by investing in quality assurance techniques.

- *Question level* is used to evaluate each of the two defined goals, a set of questions was elaborated. Economic sustainability's questions were defined based on the technical debt metaphor's elements (shown in Section 4.1.1, "Technical sustainability evaluation"), while the technical sustainability's questions were defined based on the ISO 250nn quality characteristics (shown in Section 4.1.2, "Economic sustainability evaluation").
- *Metric level* contains a set of metrics that were defined to answer each set of questions, and they are used to evaluate each of the two defined goals. These metrics were also based on the previously discussed technical debt measurements (Section 2.4) for economic sustainability, and the software quality characteristics (Section 2.3) for technical sustainability.
- *Data collection level* is used in two different contexts: the first is the data collection relating to the software static analysis technique to assess a software's past conditions; and the second is the data collection relating to the output from the proposed simulation model in order to evaluate the potential future outcomes (scenarios) from the different hypothetical resource allocation policies.

Once the hierarchical evaluation structure had been defined, the proposed framework was assembled, which was inspired by iterative nature of the system dynamics modelling approach discussed in Section 3.3.1, and illustrated in Figure 9, and the "System Dynamics-based Project-management Integrated Methodology" (SYDPIM) proposed by Rodrigues (2000). The SYDPIM framework focuses on the engineering and project management process of software product development; within this framework it was proposed to integrate the system dynamics approach with the Program Evaluation and Review Technique (PERT) and the Critical Path Method (CPM) project management tools.

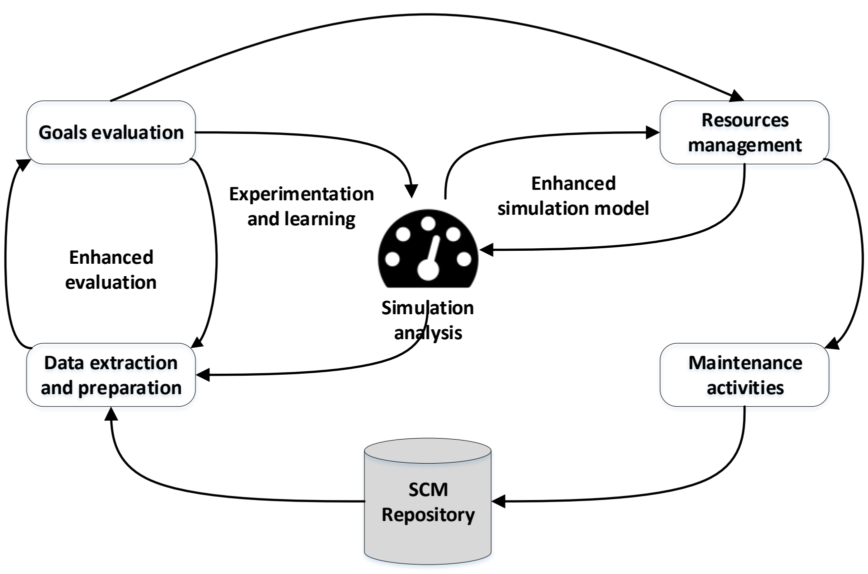
The proposed framework in this study focuses on the software maintenance process, especially in the resource allocation management of maintenance activities, which can be considered as a portfolio investment management. In the finance domain, a portfolio enables the establishment of a specific mix of investments that should generate the highest return for a given level of risk (Markowitz, 1952). The portfolio theory was later adopted by the project management community. McFarlan (1981) observed that organizations with risk-unbalanced project portfolios, could end up with operational disruptions or leave gaps for competitors to step in.

In the software maintenance context, the portfolio concept was used to represent the distinct investments (and, consequently, taking the associated risk) that are made in the different types of maintenance activities previously discussed in Section 2.2 ("Software maintenance"), namely perfective, corrective, and preventive maintenance activities. The investments are made by committing resources to these different types of activities, and these investments decisions were considered the resource allocation policies for the maintenance activities. The return on the investment strategy is measured by considering the technical and economic sustainability of the software system.

In summary, the proposed framework integrates the system dynamics approach with the software source code's quality measurement (i.e., static analysis techniques), and the GQM method's hierarchical structure for evaluating the desired goals related to software's sustainability during its evolution.

Figure 13 presents the overall framework structure. The evaluation of the software's quality characteristics measurement (m_i) at any given time (t) is performed by scanning the software system's source code with the support of an automated static analysis tool from a Software Configuration Management (SCM) tool repository. These measurements are then categorized and grouped into the ISO/IEC 25010:2011 quality model using the guidelines provided by the CISQ's standards of automated quality characteristics measures. These two steps are represented by the label *Data extraction and preparation* in Figure 13.

Figure 13. Overview of the proposed dynamical evaluation framework



Source: Adapted from Rodrigues (2000)

Next, the categorized and grouped data is evaluated using a hierarchical structure of goals, following the GQM paradigm (i.e., the *Goals evaluation* label in Figure 13), which depicts the current technical and economic sustainability state of the software system under analysis. The data extraction and preparation, as well as the goal evaluation steps, are iterative, because, in light of the results of each of these steps, details both of the data extraction and preparation process and of the hierarchical evaluation structure are prone to revisions and adjustments based on the knowledge acquired from the real software system's environment (the *Enhanced evaluation* label in Figure 13).

To optimize the investments made in maintenance activities, and thus to move closer to the desired goals, managers and organizations can evaluate the long-term impacts that the resource allocation scenarios would have by testing and evaluating them in the virtual simulation environment (the *Simulation analysis* and *Experimentation and learning* labels in Figure 13). These assessments can be performed before committing a significant amount of resources that could eventually shorten the software's lifetime.

The knowledge acquired from the experiments conducted through the execution of scenarios of interest in the virtual simulation environment can be used to improve the simulation model's structures and assumptions (the *Enhanced simulation model* label in Figure 13), and to intervene in the maintenance activities for the software system, with the software's source code then being stored in the *SCM Repository*. The proposed framework is a closed loop structure, in which the information gathered from the software is refined with knowledge acquired

through the simulation model execution, which, in turn, results in action taken in the real world that changes the current state of the software system. This cycle repeats throughout the entire life of the software, only being interrupted when a decision is taken to shut it down and finish the software's operating lifetime.

4.1 Hierarchical software sustainability evaluation structure

A hierarchical evaluation structure was developed according to the method previously described in Section 2.7 ("Goal, Question, Metric method"). The structure contains goals, sub-goals, questions, and metrics, which were later used to assess both the technical and the economic sustainability dimensions of a software system.

Section 4.1.1 presents the hierarchical structure used for evaluating technical sustainability, and Section 4.1.2 presents the hierarchical structure used for evaluating economic sustainability.

4.1.1 Technical sustainability evaluation

The technical sustainability hierarchical evaluation structure is based on a subset of the ISO 250nn quality characteristics. The structure was developed following the previously discussed GQM method (see Section 2.7, "Goal, Question, Metric method"), and the result is shown in Table 5.

Table 5. Technical sustainability goals question, and metrics structure.

Goal	Analyze	software product
	For the purpose of	characterization
	With respect to	**technical sustainability**
	From the viewpoint of	the maintenance team
Sub-goal	Analyze	software product
	For the purpose of	evaluating
	With respect to	**reliability**
	From the viewpoint of	the maintenance team
Question	How many known violations of reliability rules does it have?	
Metric	Total number of reliability rules violations (#)	
Question	What is its functional density of reliability violations?	
Metric	Total number of reliability rules violations / Size of the software (#/FP)	
Sub-goal	Analyze	software product
	For the purpose of	evaluating
	With respect to	**maintainability**

	From the viewpoint of	the maintenance team
Question	How many known violations of maintainability rules does it have?	
Metric	Total number of maintainability rules violations (#)	
Question	What is its functional density of maintainability violations?	
Metric	Total number of maintainability rules violations / Size of the software (#/FP)	
Sub-goal	Analyze	software product
	For the purpose of	Evaluating
	With respect to	**functional suitability**
	From the viewpoint of	the maintenance team
Question	What is the size of the functional requirements backlog?	
Metric	Functional requirements backlog (function points – FP)	
Question	What is its functional completeness?	
Metric	Size of the software / (Size of the software + Functional requirements backlog) (Dmnl[8])	

Source: Author

The primary evaluated goal was technical sustainability. In order for this to be evaluated, three sub-goals were defined based on the ISO 250nn quality characteristics: reliability; maintainability; and functional suitability. For each sub-goal, two pairs of questions and metrics were proposed. One pair measured the absolute number of occurrences of the quality dimension, while the other pair measured the relative number to the size of the software system, which revealed the *density* when analyzing quality violations, or the *completeness* when evaluating the software's functional suitability.

4.1.2 Economic sustainability evaluation

For assessing the economic sustainability dimension, a second hierarchical evaluation structure was defined (Table 6). It consists of the base measures, some alternative weighted measures from on the CISQ standards, and other measures extracted from the software product and the maintenance process.

Table 6. Economic sustainability goals question, and metrics structure

Goal	Analyze	software product
	For the purpose of	Characterization
	With respect to	**economic sustainability**
	From the viewpoint of	the company shareholders
Sub-goal	Analyze	software product
	For the purpose of	evaluating
	With respect to	**tangible technical asset**

[8] Dmnl is assumed as the abbreviation to dimensionless unit.

	From the viewpoint of	the company shareholders
Question	What is its size?	
Metric	Size of the software (function points – FP)	
Question	What is the estimated effort for the development of the software in the production library?	
Metric	Estimated effort (person-month)	
Sub-goal	Analyze	business risk
	For the purpose of	evaluating
	With respect to	**opportunity costs**
	From the viewpoint of	the company shareholders
Question	What is the accumulated cost associated with the lack of business functionality?	
Metric	Estimated potential effort not spent on business functionality development, due to technical debt influence when paying the principal of the debt or the maintenance productivity losses (person-month).	
Sub-goal	Analyze	software product
	For the purpose of	evaluating
	With respect to	**tangible technical debt**
	From the viewpoint of	the company shareholders
Question	What is the reliability violations remediation effort?	
Metric	Weighted sum of each reliability violation by the effort to fix it (person-month)	
Question	What is the maintainability violations remediation effort?	
Metric	Weighted sum of each maintainability violation by the effort to fix it (person-month)	
Question	What is the total remediation effort to fix all known violations?	
Metric	Sum of all weighted sums of each violation type (reliability and maintainability) (person-month)	
Question	What is the ratio of total tangible technical debt to total tangible technical asset?	
Metric	Total remediation effort / Estimated development effort (Dmnl)	
Sub-goal	Analyze	software maintenance process
	For the purpose of	Evaluating
	With respect to	**maintenance process productivity**
	From the viewpoint of	the company shareholders
Question	What is the real software maintenance productivity?	
Metric	Size of the change / Effort to make the change (FP-person/month)	
Question	What is the influence of the technical debt on the software maintenance process (i.e., the technical debt's interest)?	
Metric	(Nominal productivity − Real productivity) / Nominal productivity (Dmnl)	

Source: Author

Analogous to the first evaluation structure shown in the previous section, the primary goal for the second one is the economic sustainability dimension. Accordingly, four sub-goals were defined to evaluate the tangible technical asset, the opportunity costs, the tangible technical debt, and the maintenance process productivity.

The tangible technical asset owned by a company relates to the estimate made in the past for building the available software system functionality in the production library at a given time. This estimate is calculated by the size of the software system (measured in function points) and by the product of the sofware's size and the estimated effort for developing each unit of the metric (i.e., function points). The result of this product is a hypothetical estimate of the total effort that would be necessary to develop a new software system of an equivalent size.

On the other hand, there are the liabilities of owning and maintaining a software system in operation. The tangible technical debt is computed based on the CISQ estimation method. The *principal* is the sum of the estimated remediation effort of all the known violations, from the four different quality characteristics, presented in the software system's source code available in the production library.

In addition, another question and metric pair was used for measuring the ratio of the total technical debt to the total technical asset. This measure enables comparisons of leverage to be made across different software systems. The higher the ratio, the higher the degree of leverage and, consequently, the financial risk associated with the maintenance costs.

The maintenance productivity was also monitored to identify the *interest* component of the technical debt metaphor. The interest results from the accumulated tangible technical debt (see Section 2.4). Two pairs of question and metrics were used. One pair measured the maintenance team's productivity, and the other pair measured the impact of accumulated technical debt on maintenance team's productivity.

4.2 Proposed simulation model

The following sections detail how the iterative steps of the system dynamics approach (previously discussed in Section 3.3.1, and further detailed in the Appendix A ("System dynamics tools and elements") were used to build the proposed simulation model, and thus to address the specific purpose and research questions proposed in this thesis.

4.2.1 Problem articulation and dynamical hypothesis

Aligned with the specific purpose of the current research study, the proposed simulation model aimed to investigate why, even after the beginning of its operation, a software system continuously demands investments to remain useful, thus satisfying its users while also meeting business needs, and how different resource allocation policies for maintenance activities affect

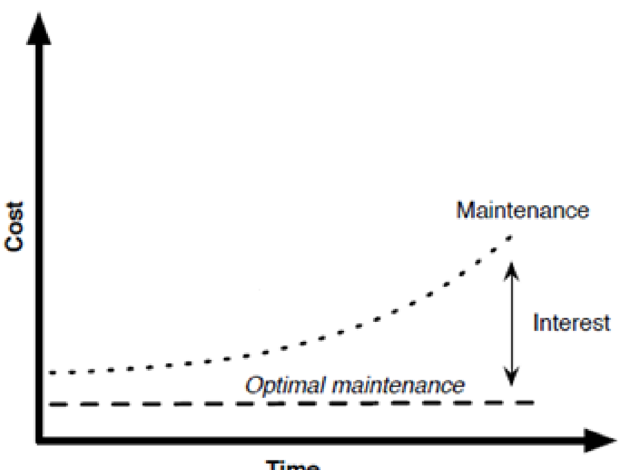
the evolutionary patterns of software system in relation to its technical and economic sustainability.

In order to understand the dynamical behaviors of the key variables related to the problem under investigation, these behaviors over time were analyzed based on theoretical and empirical evidence, and their reference modes were established.

Figure 14 shows how changes in the maintenance cost of a software system are influenced by the technical debt incurred. If no should-fix violation happened during development or maintenance activities, there would be no technical debt. Assuming that the size of the maintenance team remains constant over time, the maintenance cost would remain stable at an optimum level (*Optimal maintenance*). This residual cost occurs because, even without technical debt, perfective and corrective maintenance activities are still needed to meet new demands and correct latent defects identified during the operation phase of the software system.

However, as captured by Lehman's laws of evolution related to complexity growth and decreasing quality (the second and the seventh laws respectively; see Section 2.1), violations are an intrinsic part of the software development and maintenance activities. As violations occur during the software lifetime, the principal component of the technical debt grows, and so does its *Interest*, which is measured by the difference between the optimal *Maintenance cost* and the actual cost (*Maintenance*). The maintenance cost also grows due to degradation of the maintenance activities' productivity caused by the erosion of the software system's maintainability.

Figure 14. Maintenance costs behavior over time



Source: Adapted from Nugroho, Visser, and Kuipers (2011)

The maintenance cost is defined by the effort required to deliver the demands related to functional and non-functional requirements, which are in turn defined by the product of the maintenance team size and its productivity. The maintenance team's productivity is affected by should-fix violations accumulated (consisting of the technical debt's principal component), which reduces the software product's maintainability and, consequently, the maintenance team's productivity.

The technical debt accumulation over time is depicted in Figure 15. Assuming that the increment of the effort spent on maintenance activities at each unit of time ($Effort/\Delta t$) and the violations' potential (i.e., the approximate number of violations expected to be found during development or maintenance of the software system) remains constant over time, both the *Total effort* and the technical debt (TD) principal should show a linear growth.

Figure 15. Technical debt (TD) principal and maintenance effort behavior overtime

Source: Author

The behaviors observed in the plot shown in Figure 15, can also be described by the following analytical equations

$$Total\ effort(t) = \int_0^t \dot{Effort}(t) \cdot dt$$

$$TD\ principal(t) = violations\ pottential\ \cdot productivity \cdot \int_0^t \dot{Effort}(t) \cdot dt$$

Bakota et al. (2012) identified that there is an exponential relationship between the software maintainability (*M*) and the effort applied to the perfective maintenance activities, and their proposed model was based on two basic assumptions.

The first assumption implies that the changes to a source code do not decrease the disorder in it (except when modifications explicitly aim at maintainability improvement), which is coherent with the second law of software evolution. $S(t)$ represents the size of the software and $\lambda(t)$ the change rate of the software code at a time *t*. Their product results in the size of the change, and q is the erosion factor representing the amount of the damage inflicted when a line of code is changed.

$$\frac{dM(t)}{dt} = -\, q \cdot S(t) \cdot \lambda(t) \qquad\qquad (q \geq 0)$$

Bakota et al.'s second assumption states that the amount of change to the source code to add new functionalities (i.e., $S(t) \cdot \lambda(t)$) is proportional to the effort invested and to the software system's maintainability $M(t)$ at a given time *t*:

$$\frac{dC(t)}{dt} = \frac{S(t) \cdot \lambda(t)}{M(t)}$$

In simple terms, if one applies more effort, the code would change faster, and a more maintainable system would change even faster (and thus more cheaply), when the same effort is applied. Another interpretation is that the effort necessary for adding new functionalities is inversely proportional to the maintainability at a time *t*.

When combining the two assumptions, the authors proposed the following equation for the software maintainability estimation:

$$M(t) = e^{-\, q \cdot C(t)}$$

Thus, when applying a constant maintenance effort in each time interval over time, reproducing the linear growth behavior depicted in Figure 15, the maintainability of the software product shows the exponential decay behavior presented in Figure 16 (the *Real maintainability* line).

Figure 16. Changes to software maintainability when the effort employed remains constant

Optimal maintainability

Mainainability(scalar)

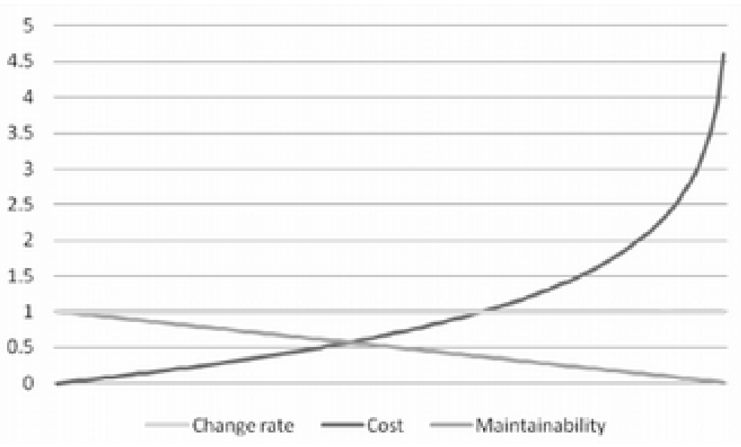
Real maintainability

Time

Source: Adapted from Bakota et al. (2012)

Another theoretical scenario can be inferred from the previous equations; in this scenario, the change rate is to be made constant over time. In order to maintain a constant change rate, and taking into consideration the previously stated assumptions, it would demand an exponential growth in the employed effort to develop new functionalities as the maintainability linearly decreases over time (Figure 17).

Figure 17. Change of effort employed and the software maintainability when change rate is constant



Source: Bakota et al. (2012)

The software's disorder represented by the maintainability measure is a proxy for estimating the impact that the technical debt incurred during the development and maintenance activities performed on a software system has on the maintenance productivity; this effect of the technical debt on the maintenance productivity is called *interest* (see Section 2.4). Thus, when performing changes to the software code, quality violations are intentionally or

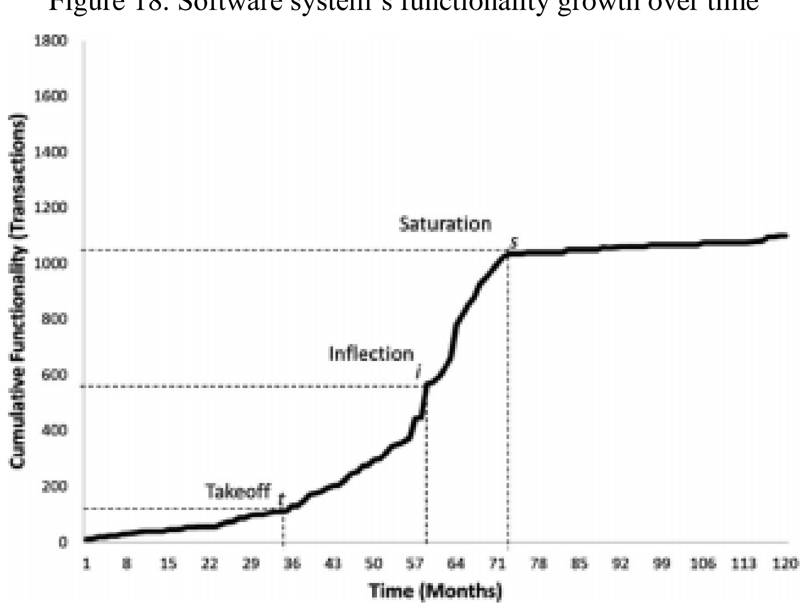
unintentionally incorporated, which reduces the software maintainability, making it costlier to change over the time.

Ramasubbu and Kemerer (2014) investigated the functionality growth patterns of a commercial software system over time, according to different development and maintenance investment policies, and they found some particularities among the observed patterns.

First, the authors found that, within the analyzed empirical data, the functionality growth of an ideal and hypothetical software system base version would follow an S-shaped pattern (also known as the logistic growth model), which they argued is in line with the product growth model in the broader population; this behavior has been established as an empirical generalization (Mahajan et al., 2000; Rogers, 2003), that is, as a regularity that repeats over a variety of circumstances. The S-shaped growth pattern arises from the different demand for new functionalities over time and how it is addressed (e.g., velocity) to be incorporated in the software system production baseline.

Moreover, Ramasubbu and Kemerer (2014) identified three distinct transition zones during the software life cycle *takeoff* (*t*) represents the point in time when, after an initial slow growth, the software's functionality growth starts to speed up to meet the users' and business demands; *inflection* (*i*), as in the logistic growth model, is a moment that lies at the half-way mark of the software's size at the end of its lifespan; and *saturation* (*s*) is defined as the moment when the software's functionality growth starts to decline. These three tipping zones are shown in Figure 18.

Figure 18. Software system's functionality growth over time



Source: Ramasubbu and Kemerer (2014)

Figure 19 illustrates the resulting behaviors from two different and antagonistic resource allocation policies. The plot on the left shows the functionality growth (a dashed line) of a policy where decisions were made toward incurring violations, and thus accumulation technical debt, causing the software maintainability to decrease resulting in the interest growing. This plot illustrates that this strategy makes the takeoff and the saturation occur much earlier than the baseline of the S-shaped curve discussed in Figure 18.

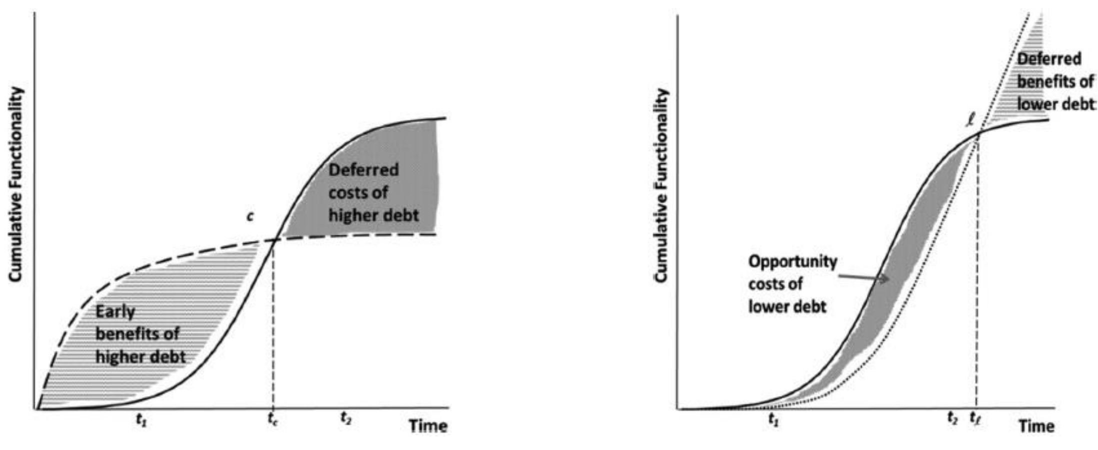
The *early benefits of debt* could be of value so that the development could be faster, and the software delivered earlier. However, as the software's lifetime extends and more maintenance investments become necessary in order for the software to remain useful and satisfy its users (in accordance with the first and sixth laws of software evolution; see Section 2.1), these initial benefits can be exceeded by the *deferred costs of higher debt*.

The baseline and the high-debt curves intersect at point *c*. If the benefits accumulated before t_c are higher than the deferred costs after t_c, then the early benefits are higher than the long-term costs. This makes it a suitable strategy for stakeholders who are mainly interested in the short-term return on investments associated with a higher rate of business functionality growth and who are not concerned with the late maintenance costs associated with the low quality software systems (i.e., low maintainability).

The plot on the right side of Figure 19 shows an opposite strategy: delaying quality violations and creating *opportunity costs of lower debt*. After point *l*, and without compromising the software maintainability, the maintenance costs remain lower than the previous policy and create *deferred benefits of lower debt*.

Analogous to the prior analysis, if the accumulated opportunity costs before t_l are lower than the deferred benefits after t_l, then the technical debt avoidance policy benefits exceed the lack of business functionalities. This strategy is therefore more suitable to contexts where stakeholders are mainly interested in long-term return on investments associated with lower late maintenance costs (i.e., high maintainability).

Figure 19. Expected growth in higher (left) and lower (right) technical debt scenarios



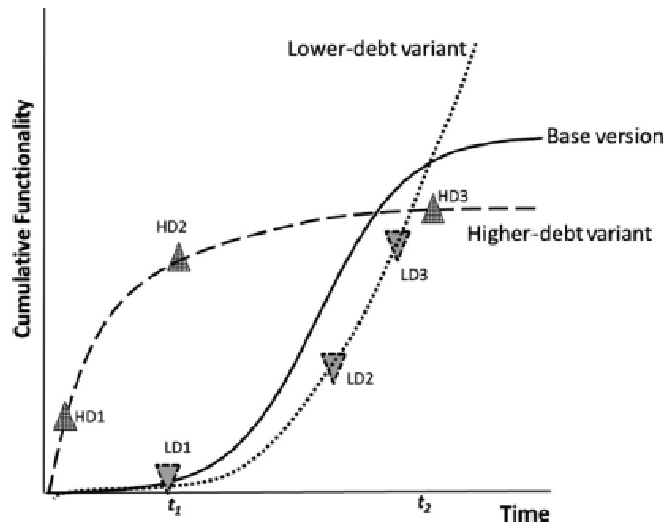
Source: Ramasubbu and Kemerer (2014)

These evolutionary patterns are in accordance with the analogy made by Sauer (1993), according to which he compared a software system's success to the survival goals of a natural system. Depending on the software system's goal, both resource allocation policies (high- and low-debt incurrence) can be valuable. A short-lived software system could benefit from the high-debt strategy and long-lived software system could benefit from the low-debt strategy.

However, problems arise when choosing a resource allocation policy unsuited to the desired goal. Examples of such unsuitable policies are the adoption of a high-debt strategy for a long-lived software system and making it excessively costly to maintain at a late state, or, on the contrary, the adoption of a low-debt strategy for a short-lived system and not realizing the business benefits associated with higher delivery rates of new business functionalities during the early stages.

Another important finding of Ramasubbu and Kemerer's (2014) study, and one of the core concepts for establishing the reference mode of the proposed model, was their identification of six key decision points throughout the software system's evolution life span. Three of these decision points are on the high-debt policy's trajectory (HD1-3), and three are on the low-debt policy's trajectory (LD1-3); these are shown in Figure 20.

Figure 20. Maintenance policies variations and development investments decisions



Source: Ramasubbu and Kemerer (2014)

According to the Ramasubbu and Kemerer, (2014) the first pair of decision points (HD1 and LD1) pertain to opportunities to alter maintenance investments policies (i.e., resource allocation policies) as they occur before the *takeoff* transition zone. Whatever the chosen policy, any associated costs should be accounted as learning investments.

Once the software system's functionality growth takes off, it should be possible to get reliable estimates of the potential technical debt obligation at decision points *HD2* and *LD2*, which will need to be paid off at some point in the future, as well as estimates of the opportunity costs due to the lack of business functionalities.

The last two decision points (HD3 and LD3) occur in the late stage of the software system's life cycle. When taking a high-debt policy variant, and if the software system is retired at this point, then the incurred technical debt could be partially written off (not entirely because the costs related to the higher costs of maintenance due to the software's lower maintainability would already be incurred – i.e., the interest amount). However, if the software system has still to be kept in operation, decision-makers face a debt obligation and resources must be allocated in preventive maintenance activities in order to reduce the accumulated technical debt.

Similarly, when following a low-debt policy variant, and if the software system is retired at a late stage, the deferred benefits are lost. However, if kept in operation, the opportunity costs can be recovered due the higher maintainability (i.e., higher maintenance productivity) and at a relatively cheaper cost when compared to the higher debt policy variant.

When paying the technical debt interest, some authors have proposed two distinct situations when the technical debt's interest can manifest itself (Ampatzoglou et al., 2016):

- Interest while performing maintenance activities (*IM*): The difference between the necessary effort for performing maintenance in software with accumulated technical debt and the effort that would be necessary with the same software if it had no technical debt. This type of interest will occur, and will be simultaneously paid, when maintenance activities are performed.
- Interest while repaying technical debt (*IR*): The difference between the necessary effort for repaying a technical debt's principal item at any time point *t* is higher than the effort that would be needed for repaying it at any time point prior to *t*. This type of interest will occur when (and if) the technical debt item is to be paid off.

This distinction adds more details about when these extra costs can occur. Thus, the interest of each technical debt item (I_{TDI}) should be estimated based on the following formula:

$$I_{TDI} = IR_{TDI} + IM_{TDI}$$

To transform the previous formula from the technical debt item level to the software system level, an aggregation function is used so that the interest at the software system level (*I*) can be calculated as:

$$I = \sum_{j=1}^{j=count(TDI)} IR_j + IM_j$$

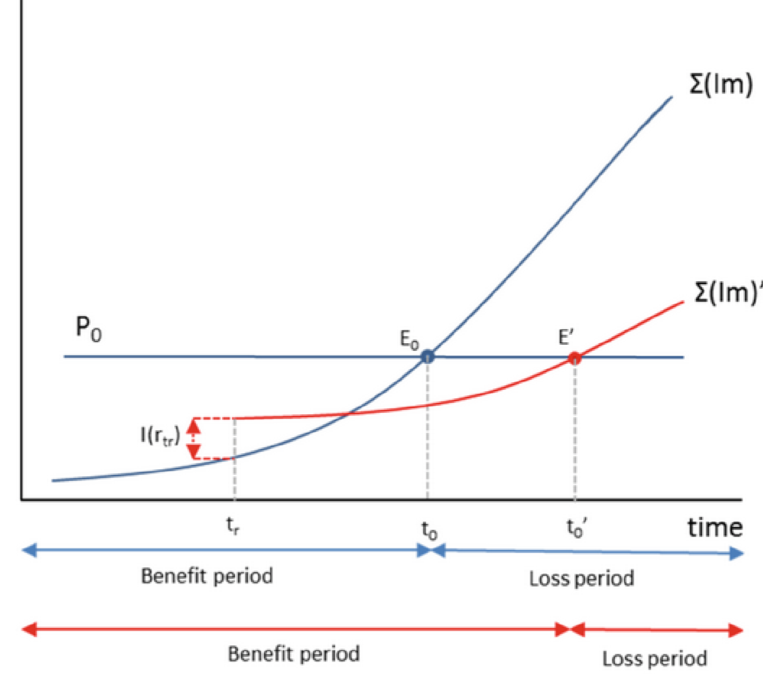
Further, Ampatzoglou et al. (2016) proposed an interest theory that is presented in Figure 21, where the x-axis represents the time and the y-axis represents the amount of effort. The P_0 line represents the initial principal (the money/time borrowed when taking development shortcuts), and this assumed to be constant over time if no extra should-fix violations occurred.

When the level of should-fix violations (i.e., the technical debt's principal) remains constant over time, so too the software maintainability will remain constant. Thus, the accumulated amount of interest incurred ($\sum(IM)$) during maintenance activities would continuously increase following an exponential slope (see Figure 17).

The intersection point E_0 represents the equilibrium at time stamp t_0, indicating that the complete amount of effort borrowed from technical debt (e.g., shortcuts to speed up the maintenance activity) has been spent on extra maintenance effort caused by the incurred technical debt. It is important to note that this equilibrium point is related only through the analysis of the effort (both saved and spent). Any other related costs or benefits were not considered in this formulation.

Therefore, if the expected lifespan of the technical debt item is shorter than the time t_0 (the equilibrium point), then undertaking the debt is a good decision; however, if not, the incurred debt becomes harmful.

Figure 21. Impact of technical debt interest due to technical debt item's repayment



Source: Ampatzoglou et al. (2016)

When preventive maintenance is performed to remove a technical debt at a time point t_r, the line representing the accumulated interest $\sum(IM)$ moves up due to the interest paid when repaying the debt ($IR(t_r)$), while the slope of the curve softens since the maintainability of the software increases and the interest is expected to lower for future maintenance activity (IM). This repayment at time t_r postpones the equilibrium point (E') to a later time point t_0', expanding the benefit period.

4.2.1.1 Model's subsystem diagram

The proposed model was organized into four subsystems: *Quality measurement*, *Goals evaluation*, Software *maintenance*, and *Resource management*. Figure 22 presents a diagram of the model's subsystems and their main interactions.

Figure 22. Subsystem diagram of the proposed model.

Source: Author

Additionally, Table 7 presents a brief description of each of the proposed model's four subsystems.

Table 7. Description of the proposed model's subsystems

Model subsystem	Description
Quality measurement	This consists of a set of proxies that are used to mimic the software quality characteristics related to functional and nonfunctional requirements and the effects caused by the interaction of the software system with the external environment throughout its life cycle. The laws of software evolution also influence the software system's quality measurements, such as continuing changes, decreasing quality, increasing complexity and continuing growth.
Goals evaluation	This contains the structures used for evaluating the defined software system's goals related to economic and technical sustainability. This subsystem also contains the structures involved in the reasoning and

	the decision-making processes about the goals' gaps and the resource allocation policies for intervening, through the maintenance activities, in the software's current operational state.
Resources management	This represents the execution of the decisions made about resource allocation policies (financial, personnel, etc.) for performing activities related to perfective, corrective, and preventive maintenance. The availability of the maintenance team is represented as a finite resource that imposes restrictions on the ability to perform the necessary interventions, and the decisions made among resource allocation policies constitutes a trade-off analysis between reducing the technical debt (preventive maintenance), removing reliability, performance, and security defects (corrective maintenance), and meeting the demands for new or modified functionalities (perfective maintenance).
Software maintenance	This includes elements related to the execution of software maintenance activities performed to meet the business demands (represented by the functional requirements backlog) of the software's users. In addition, this subsystem is also responsible for performing preventive maintenance activities to reduce the level of technical debt and to preserve the software product maintainability (i.e., repaying technical debt to reduce the interest), and the corrective maintenance activities for fixing reliability, security and performance defects.

Source: Author

4.2.1.2 Model boundary chart

Table 8 shows the model boundary chart with the variables considered endogenous, exogenous, or excluded.

Table 8. Boundary chart of the proposed model

Endogenous	**Exogenous**	**Excluded**
Functional requirements backlog	Software product growth rate	Adaptive maintenance activities
Production library (functional requirements available)	Refactoring effort necessary	Data quality
Maintainability violations (TD principal)	Refactoring overhead	Intangible assets
Maintenance team	Schedule	Liability risks
Maintenance team productivity		Software infrastructure (hardware, operation system, network, etc.).

Perfective, corrective and preventive maintenance activities	Support services (training, helpdesk etc.)
Reliability violations	User satisfaction levels
Resource allocation policies	Technology evolution
Software product maintainability	
Violation rates	
Violation potentials	
Violation density	
Tangible asset	
Interest rate	
Software overall attractiveness	

Source: Author

4.2.1.3 Model's causal loop diagrams

The following sections present and describe the causal loop diagram, which is one of the elements for formulating the dynamical hypothesis tested in the present work. The hypothesis consists of the feedback structures presented in the following sections being responsible for reproducing the behaviors over time previously discussed in Section 4.2.1 ("Problem articulation and dynamical hypothesis").

For simplicity and in order that the proposed causal loop structure is better understood, a bottom-up presentation has been adopted whereby the diagram will be shown incrementally.

4.2.1.3.1 Continuing growth – Lehman's sixth law (R1)

As previously stated by Lehman (1996b) in his sixth law of software evolution, every software system must continuously grow to remain useful and fulfill the needs of its stakeholders.

This phenomenon is represented by the reinforcing loop *R1 – Continuing growth*, shown in Figure 23. This loop is composed by the software "Production library", which, when increased (measured as the number of function points in operation), will increase the software's *Functional suitability* (the ratio of the actual software features in operation and the desired software size, both measured in lines of code). When this ratio changes, the *Overall software attractiveness* will move in the same direction (it will increase or decrease), causing the

Functional requirements backlog's increase rate to follow the same trend (when attractiveness increases, it will increase faster; when it decreases, it will increase more slowly than would be expected if nothing has changed).

Figure 23. Continuing growth feedback loop

Functional suitability
Production library
Overall software attractiveness
R1
Continuing Growth (6th law)
Software perfective maintenance rate
Resources allocated to perfective maintenance
Functional requirements backlog

Source: Author

When the *Functional requirements backlog* increases, it will demand more resources to be allocated to the software's perfective maintenance activities (increasing the effort and investments made) to transform functional requirements items into running operational software code, which will then increase the software's *Production library* and close the *R1* feedback loop.

4.2.1.3.2 Increasing complexity – Lehman's second law (B1)

If nothing limits the dynamic produced by the interacting elements of the loop "R1", the software's "Production library" would theoretically grow indefinitely, producing an observed behavior that goes against what Lehman (1996b) previously observed during his empirical studies at IBM, and later captured in his second law of software evolution, namely the law of "increasing complexity".

One of the forces that limits the continuous growth of the software system size stems from the constant changes made to the code from perfective maintenance and the fact that the software maintenance activities are error prone. Consequently, every line of code changed

increases the probability of introducing violations of reliability, performance, security, and maintainability (the latter making the software harder and more costly to modify).

This behavior is illustrated by the balancing feedback loop *B1 – Increasing complexity* (Figure 24) and captured by the second software evolution law: increasing complexity. When the *Perfective maintenance effort* is employed to develop a new software's functional requirements (i.e., resources are allocated), *Maintainability violations* are also incurred due to process failures, human errors, and so on. Violations of maintainability are the basis of the software technical debt metaphor, corresponding to its *principal* (see Section 2.4 for more details). If the violations are not removed, they will hinder the maintenance productivity team (i.e., *Current maintenance team productivity*), which will, in turn, cause the *Perfective maintenance productivity* to follow this trend and be negatively impacted.

Figure 24. Increasing complexity feedback loop

Source: Author

The closed feedback structure formed by the closed loops *R1* and *B1* follow the system archetype "limits to growth" proposed by Senge (2006). System archetypes are similar to the software engineering concept of "design patterns" (Gamma et al., 1994), where they capture the software engineering community's knowledge and good practices in relation to commonly used structures and the contexts of problems.

Senge describes the "limits to growth" archetype as follows:

> A process feeds on itself to produce a period of accelerating growth or expansion. Then the growth begins to slow (often inexplicably to the participants in the system) and eventually comes to a halt, and may even reverse itself and begin and accelerating collapse.

> The growth phase is caused by a reinforcing feedback process (or by several reinforcing feedback processes). The slowing arises due to a balancing process brought into play as a "limit" is approached. The limit can be a resource constraint, or an external or internal response to growth. The accelerating collapse (when it occurs) arises from the reinforcing process operating in reverse, to generate more and more contraction. (Senge, 2006, p. 354)

4.2.1.3.3 Requirements "gold plating" (B2)

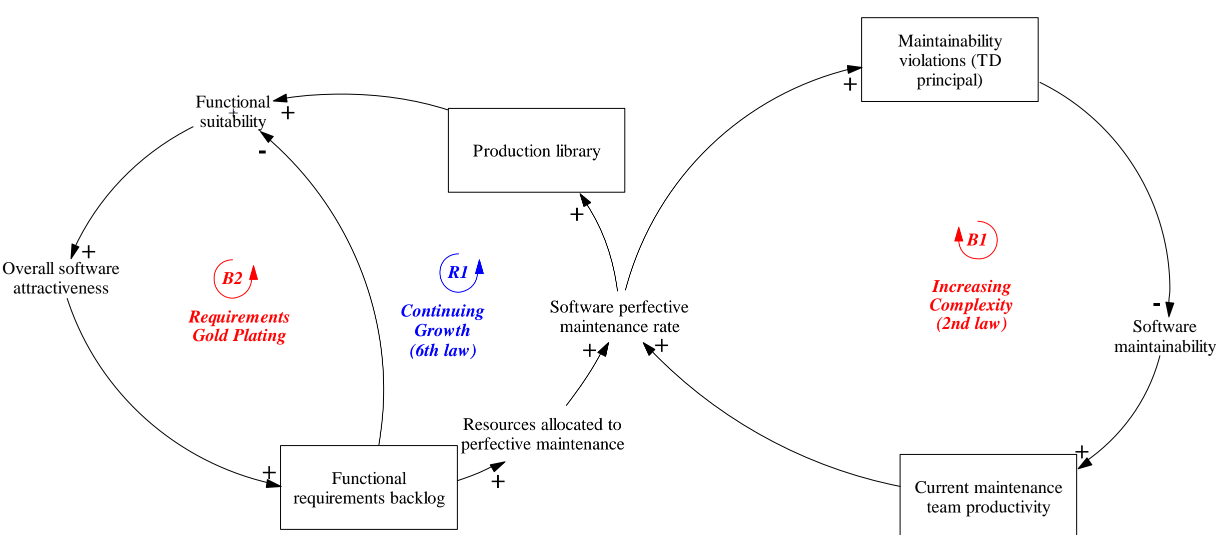
Besides the constraint to the software system's growth imposed by the *Increasing complexity* feedback structure, McConnell (1996) pointed out that software projects commonly have more requirements than needed (either by the project team or by the client) right from the beginning, which can unnecessarily lengthen the software project's development schedule.

In order to capture this in the model, there is a negative link between *Functional requirements backlog* and the *Functional suitability*. *Functional suitability* is calculated as a ratio between the overall desired software functionality and the functionality deployed in the *Production library*.

$$Functional\ suitability = \frac{Production\ library}{(Production\ library + Functional\ requirements\ backlog)}$$

Thus, when requirements start to accumulate in the backlog because they are not developed and deployed in the *Production library*, the *Functional suitability* diminishes, thus reducing the *Software overall attractiveness*, which in turn reduces the desire for new features for the operating system (Figure 25).

Figure 25. Requirements gold plating feedback structure



Source: Author

This feedback loop (B2) indicates that if the organization is unable to manage user expectation for fast delivery of new operational functionality and to keep assuming new commitments for new developments, the *Software overall attractiveness* would inevitably be harmed as the organization would not be able to deliver the desired new functionalities within the expected time frame.

4.2.1.3.4 Declining quality – Lehman's seventh law (B3)

However, it is not only from broken promises (which impact functional suitability) and increasing complexity that the maintained software system suffers along its operational lifetime.

Lehman's (1996b) seventh law states that a software system will also appear to face a continuous degradation of quality, and he observed that a software system that has previously performed satisfactorily may over time suddenly exhibit unexpected, and previously unobserved, unsatisfactory behavior.

Lehman argued that this phenomenon can be explained by the *Principle of Uncertainty*:

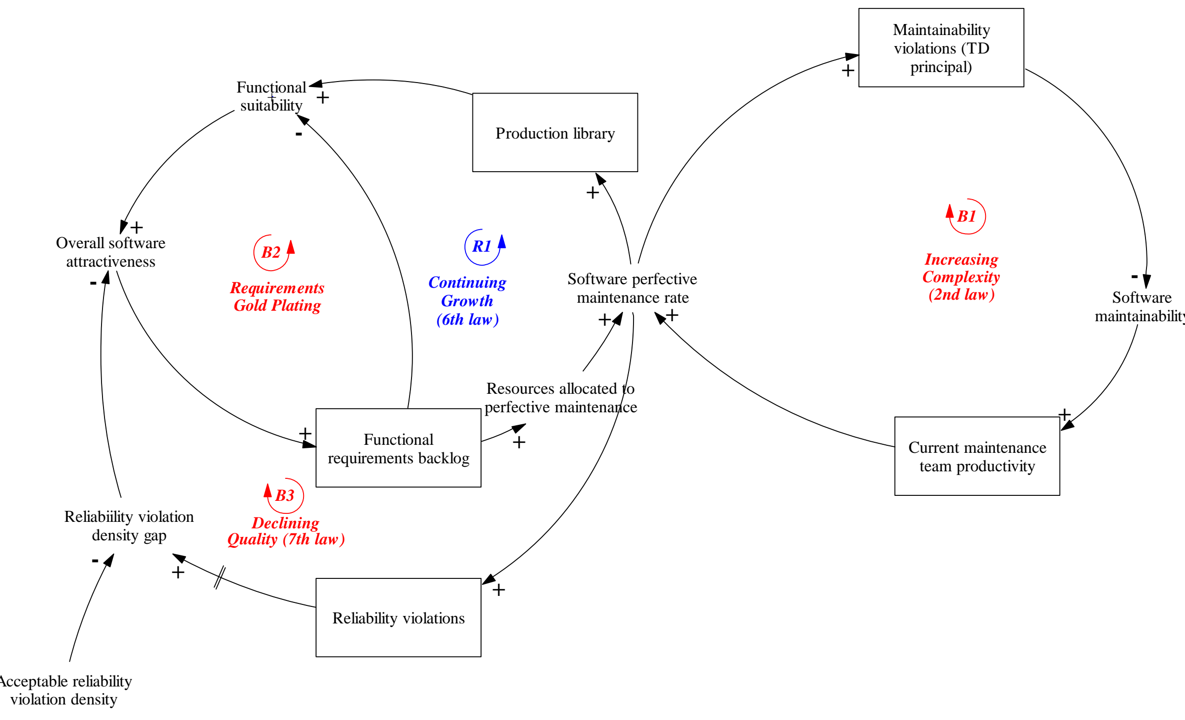
> Even if the outcome of past executions of an E-type program have been satisfactory, the outcome of further executions is inherently uncertain; that is, a program may display unsatisfactory behavior or invalid results. (Lehman & Ramil, 2002, p. 185)

Although this principle statement makes no reference to the source of the uncertainties, the proposed model incorporate them as the *Reliability violations* that are introduced by the *Software perfective maintenance rate*, which, with some time delays, increases the *Reliability violation density gap*, thus reducing the *Software overall attractiveness*.

$$\begin{aligned} &\mathit{Reliability\ violation\ density\ gap} \\ &\qquad = \frac{\sum \mathit{Reliability\ violations}}{\mathit{Production\ libray}} - \mathit{Aceptable\ reliability\ violation\ density} \end{aligned}$$

These interactions are represented by the balancing feedback loop B3, which is illustrated in Figure 26.

Figure 26. Declining quality feedback structure



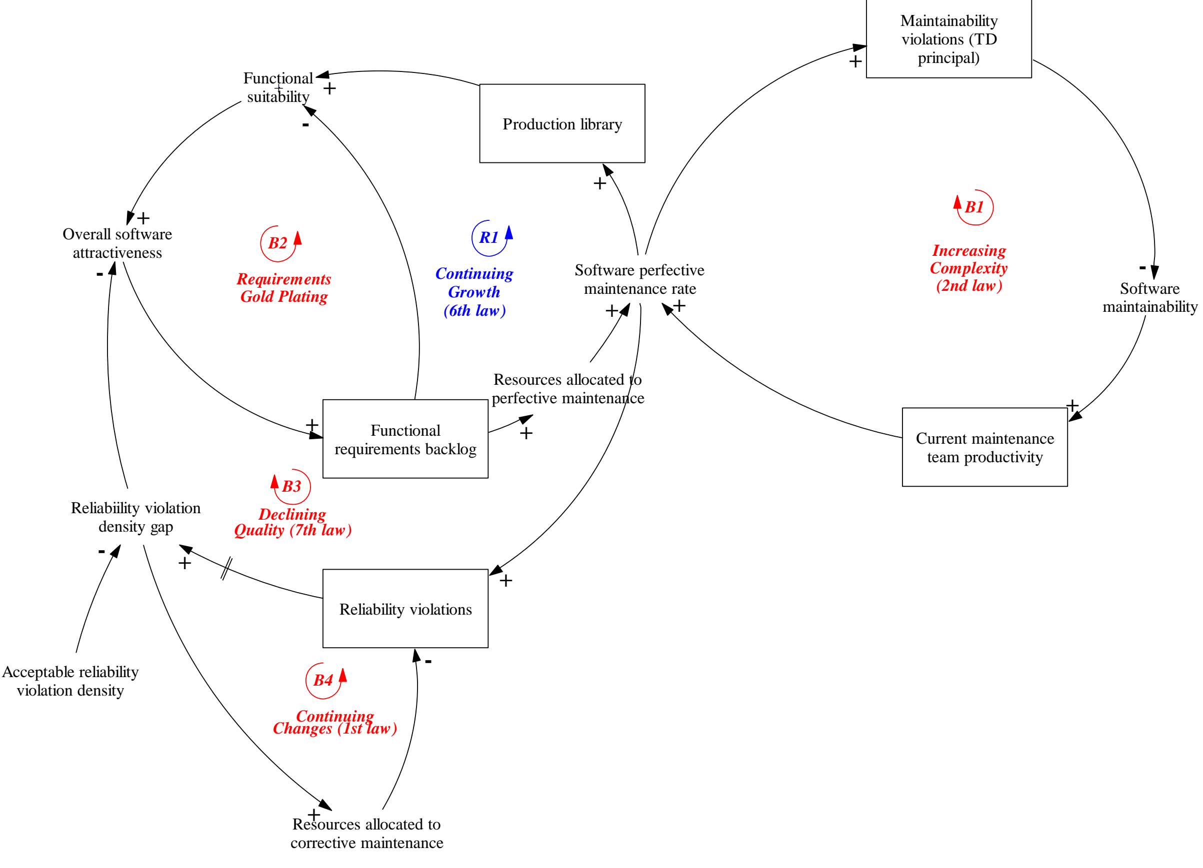
Source: Author

4.2.1.3.5 Continuing changes – Lehman's first law (B4)

In order to mitigate the effects of the declining quality captured by the seventh law of software evolution, a countermeasure is necessary so that the software system constantly adapts. During the life cycle of the software system, unexpected and unsatisfactory behaviors emerge, and they are caused by violations that were previously introduced during its development and the maintenance activities made during its lifetime, which were not properly identified and removed before deploying the modified software code in the operational environment.

This countermeasure is captured in Figure 27 by the balancing feedback loop B4, consisting of the *Reliability violations* level, which, when increased, also increases the *Reliability violation density gap* (calculated as the difference between the *Acceptable reliability violation density*) that will trigger the decision-making to move towards allocating resources in corrective maintenance activities for removing theses violations (i.e., *Resources allocated to corrective maintenance*).

Figure 27. Continuing changes feedback structure



Source: Author

4.2.1.3.6 Work harder (R2)

From the *Increasing complexity* balancing feedback loop (Section 4.2.1.3.2), there is a degenerative effect that erodes the perfective maintenance team's productivity over time as the maintainability violations level builds up and causes the software system's maintainability to decrease.

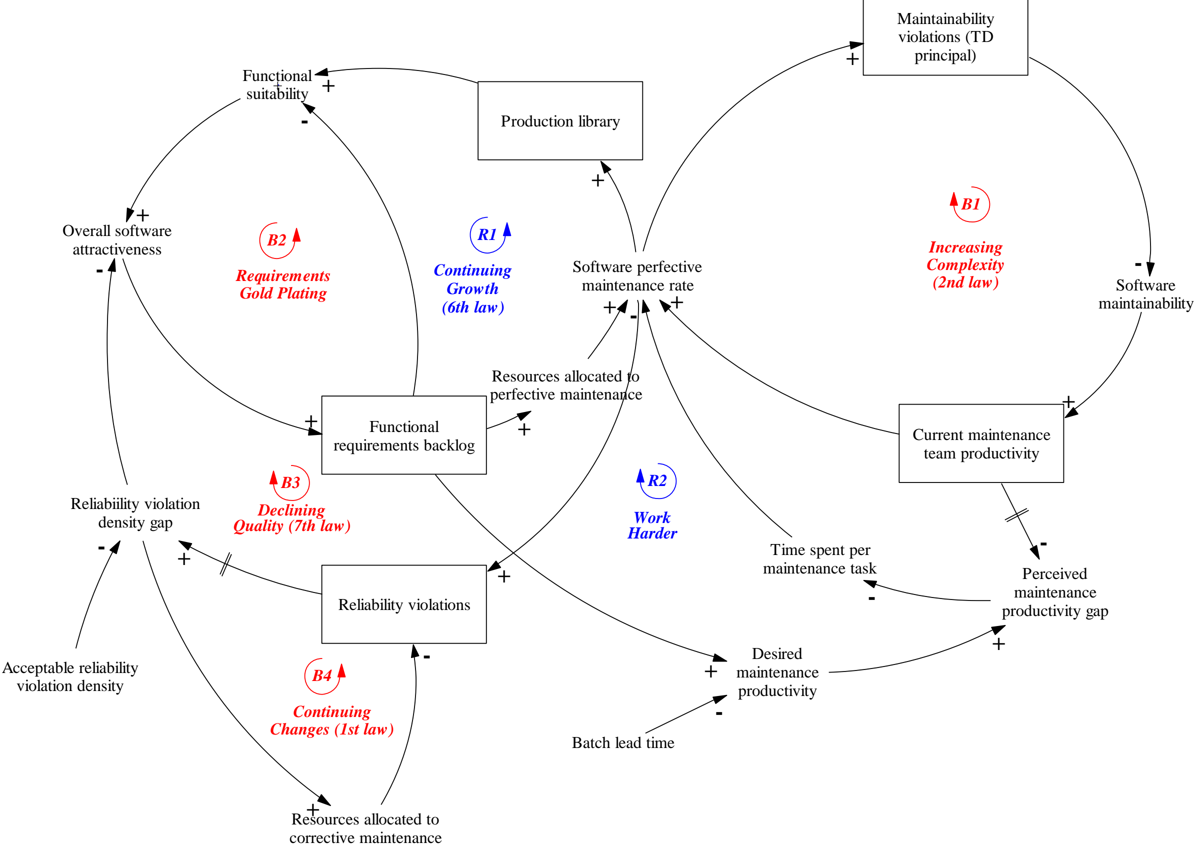
Repenning and Sterman (2001) found that it was rare to find processes performing above expectations within the studied organizations. They identified that, due to high and rising demands, team members and decision-makers usually faced performance gap (i.e., productivity shortage), and when falling behind schedule, they usually start searching for opportunities to improve and close the performance gap.

Still, organizations are usually reluctant to hire new employees and increase their fixed overhead costs. Hence, team members and decision-makers must either extend their working time (i.e., work overtime) or take shortcuts by neglecting standard routines (code quality standard, documentation, good practices, tests, and so on), capturing the idea that the throughput

gain, by cutting corners and reducing the time spent on each activity, comes at the expense of leaving behind the organization's good practices.

Figure 28 illustrates this causal relationship through the reinforcing loop *Work harder* (R2).

Figure 28.Work harder feedback structure



Source: Author

The *Desired maintenance productivity* is computed by the work still to be done in the *Functional requirements backlog* and by the company's *Batch lead time*. The *Batch lead time* defines the company interval for delivering new increments for deploying the new functionalities in the *Production library* (the software system's new releases).

$$Desired\ maintenance\ productivity = \frac{Functional\ requirements\ backlog}{Batch\ lead\ time}$$

The *Perceived maintenance productivity gap* is then calculated as the difference between the *Desired maintenance productivity* and the *Perceived maintenance productivity*, which is perceived with some time delay in the *Current maintenance productivity*.

$$\begin{aligned} \mathit{Perceived\ maintenance\ productivity\ gap} \\ &= \mathit{Perceived\ maintenance\ productivity} \\ &- \mathit{Desired\ maintenance\ productivity} \end{aligned}$$

The higher the *Perceived maintenance productivity gap*, the shorter will be the *Time spent per maintenance activity*, which will, in turn, increase the *Software perfective maintenance rate*.

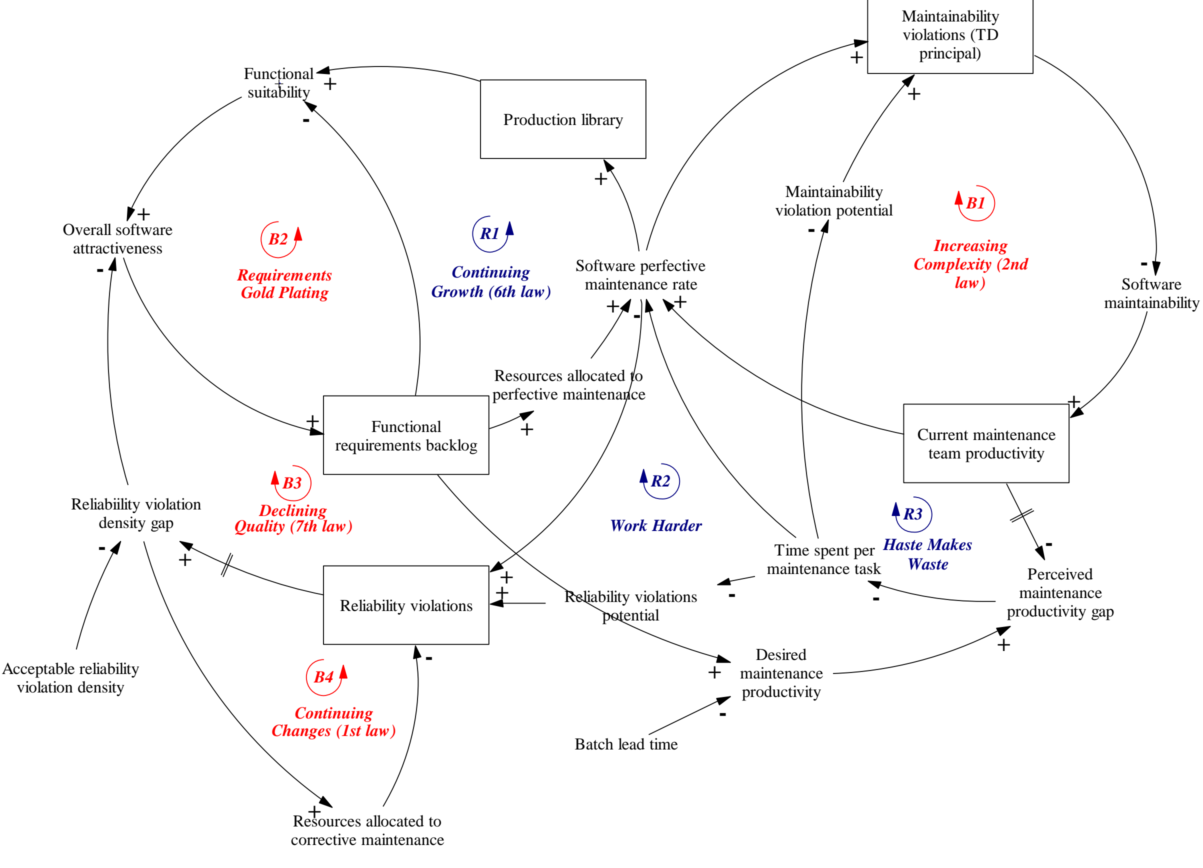
4.2.1.3.7 Haste makes waste (R3)

The delivery throughput gain achieved by the shortcuts and overtime working of the maintenance team (obtained from the *Work harder* reinforcing feedback structure, discussed in Section 4.2.1.3.6) comes at a cost.

As the maintenance team starts taking shortcuts they will, for example, start leaving behind the organization development quality norms; delivering inappropriate software quality code; begin skipping the development and running the unit tests; have insufficient time to prepare and update the software system's documentations; and so on (Abdel-Hamid, 1990; Abdel-Hamid & Madnick, 1991).

All these shortcuts, if not adequately addressed, will in the long term become manifest as they will increase the *Maintainability violation potential*. This will increase the propensity of the maintenance activities to incur more maintainability violations, which will be accumulated in the *Maintainability violations (TD principal)* level; this, in turn, will then hinder the *Software maintainability*, thus decreasing the *Current maintenance team productivity* again.

Figure 29. Haste makes waste feedback structure



Source: Author

Depending on the dynamics interactions, the effect caused by the reinforcing loop *Haste makes waste* (R3) can easily overcome the short-term throughput gains obtained from the reinforcing loop *Work harder (R2)*; hence, the software system's lifetime will be shortened, as maintaining it becomes costlier over time.

The interactions captured by these two feedback structures (R2 and R3) show that both technical and economic sustainability are intertwined and co-dependent, which is in accordance with what was previously discussed in Section 2.5 ("Software sustainability").

In addition, this phenomenon is also in accordance with Lehman's fourth law of software evolution (i.e., the "*Conservation of Organisational Stability*"; see Section 2.1):

> By and large it is still generally believed that the effort expended on system growth and evolution is determined by managerial decision. To some degree corporate and local management certainly do control activity targets and resource allocation to a project, system or activity. Their ability to do this is, however, constrained by external forces, trade unions or the availability of personnel with appropriate skills for example. But as suggested by the third law the effort usefully expended, that is to achieve satisfactory results, is also determined by system attributes, complexity for example, that are analogous to attributes such as inertia and momentum in mechanical systems. (Lehman, 1996b, p. 2)

4.2.1.3.8 Work smarter (B5)

Lehman's second law of software evolution states that "As a program is evolved its complexity increases unless work is done to maintain or reduce it", which correlates with the entropy principle of thermodynamics.

The "limits to growth" archetype is closely related to a second system thinking archetype, namely the "growth and underinvestment", with which it is usually correlated, although the final outcomes differ from each other.

Senge stated that this common system structure happens when the following dynamics occur:

> Growth approaches a limit which can be eliminated or pushed into the future if the firm, or individual, invests in additional "capacity". But the *investment must be aggressive and sufficiently rapid to forestall reduced growth or else it will never get made* [emphasis added].
>
> Oftentimes key goals or performance standards are lowered to justify underinvestment. When this happens, there is a self-fulfilling prophecy where lower goals lead to lower expectations, which are then born out by poor performance caused by underinvestment. (Senge, 2006, p. 365)

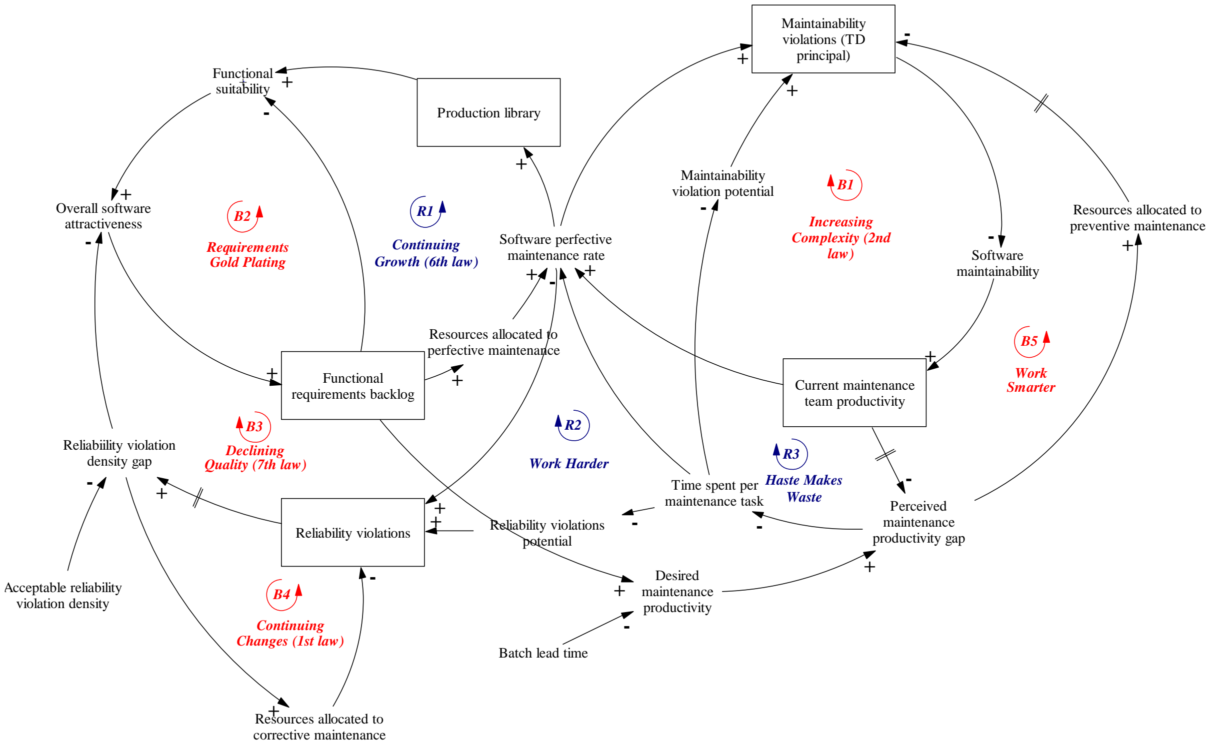
In the software maintenance context, investing in capacity is also related to paying the technical debt, although not only to this (i.e., investments can be made to improve processes, acquire new technologies, hire and train staff, etc.). This research study focuses on the technical debt influence on the maintenance productivity because, despite the process, technology and better training of teams, it will only be a matter of time for the impact caused by technical debt to become manifest, and it will eventually risk the continuing operation of the software system and its ability to adapt to new scenarios.

Incurring technical debt hinders the maintenance team's productivity. Repaying it means, in accordance with Lehman's second law of software evolution, that the organization is investing in order to restore the productivity to its desired level and close the trade-off decision between further investing in new business functionalities or improving the software system's maintainability.

Both Lehman and Senge, besides discussing different contexts, captured the trade-off decision of further exploiting short-term benefits and the consequences this brings to the long-term horizon.

Figure 30 shows the technical debt repaying structure. When the *Perceived perfective maintenance productivity* falls behind the *Desired perfective maintenance productivity* (with some time delay), decisions are made toward moving *Resources allocated to preventive maintenance* and repaying the incurred technical debt violations, and thus, restoring the desired maintenance team's productivity, which closes the balancing feedback structure *Work smarter* (B5).

Figure 30. Work smarter feedback structure



Source: Author

4.2.1.3.9 Self-regulation – Lehman's third law (B6)

The amount of resources that an organization can commit over time to investment in maintenance activities for a software system, to keep it operating as well as to satisfy its stakeholders, is limited.

In relation to his third law of software evolution (i.e., "*Self-regulation*"; see Section 2.5), Lehman argued that

> The evolution of industrially produced E-type software is implemented by a technical team operating within a larger organisation. The interests and goals of the latter extend far beyond completion of the system in question. Checks and balances will have been established by corporate and local management to ensure that operational rules are followed and organisational goals at all levels are met. The positive and negative feedback controls

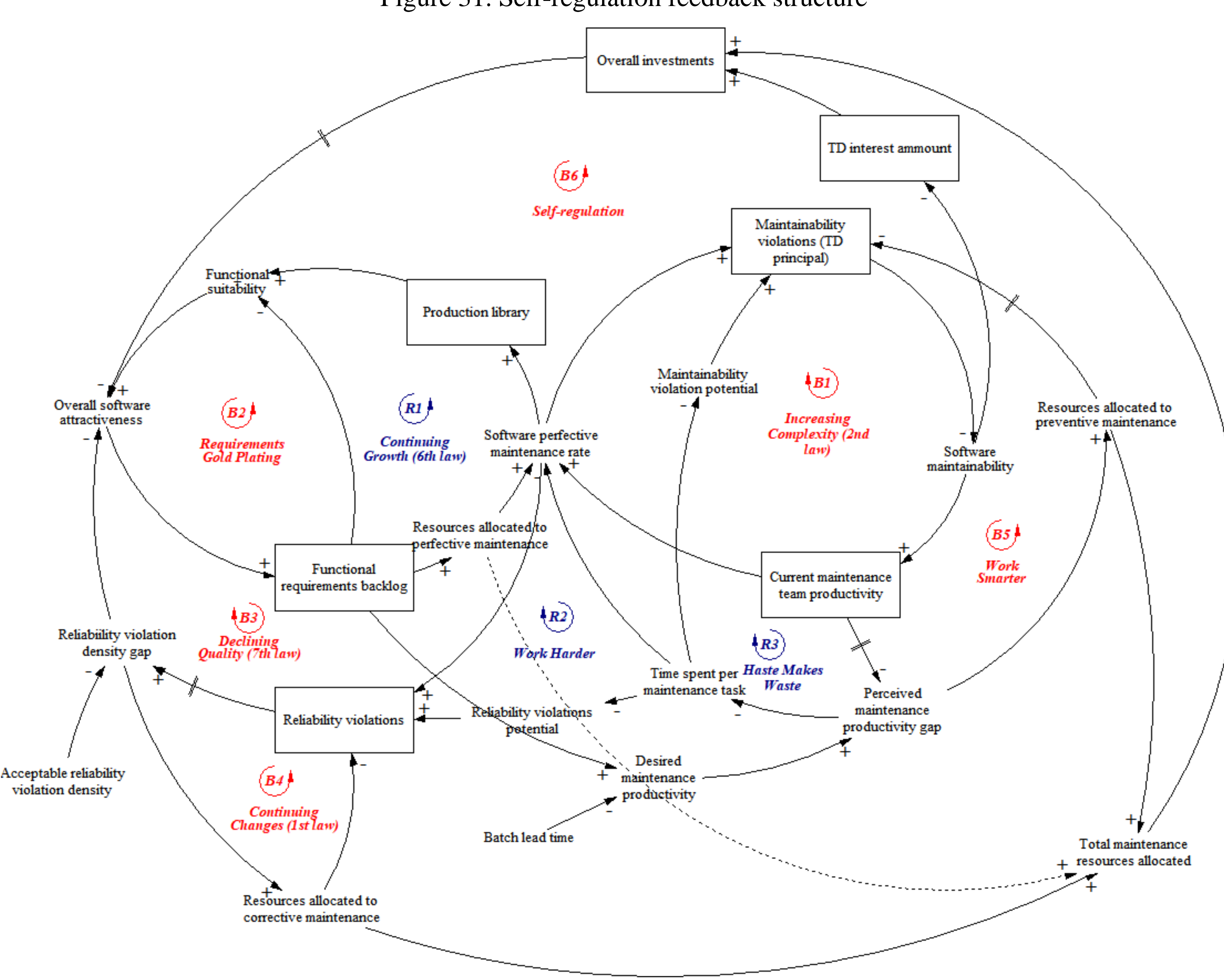
> that implement these checks and balances provide one example of feedback driven growth and stabilisation mechanisms. (Lehman, 1996b, p. 2)

The organization's goals, which extend far beyond the software system's goal, are represented by *Overall investment* made in the software maintenance activities, which consists of the sum of the *TD interest amount* and the *Total maintenance effort employed*.

$$Overall\ investment = TD\ interest\ amount + Total\ maintenance\ resources\ allocated$$

$$Total\ maintenance\ resources\ invested = \sum Resources\ allocated\ to\ perfective\ maintenance + \sum Resources\ allocated\ to\ corrective\ maintenance + \sum Resources\ allocated\ to\ preventive\ maintenance$$

Figure 31. Self-regulation feedback structure



Source: Author

The *Overall investments* made during the software's lifetime in maintenance activities negatively influence the *Software overall attractiveness*, which, in turn, closes the balancing feedback structure *Self-regulation* (B5) by moving the *Functional requirements backlog* increase rate in the same direction.

4.2.1.3.10 Feedback system – Lehman's eighth law

The eighth law proposed by Lehman, although the last to be formulated, arguably should have been the first to be stated as its concepts pervade and underlies the behaviors encapsulated by the other seven laws. The eighth law recognizes that the software evolution process, which starts as soon as the software system has been deployed, consists of a set of complex feedback loops whose effects must be carefully considered (Godfrey & German, 2014).

Lehman (1996b) argued that the role of feedback was recognized almost from the start of detailed studies on the software process and was also present in his early empirical observations of software evolution phenomena. Later, when discussing the importance of understanding the involved feedback structure, Lehman pointed out:

> The behaviour of feedback systems is not and cannot, in general, be described directly in terms of the aggregate behaviour of its forward path activities and mechanisms. Feedback constrains the ways that process constituents interact with one another and will modify their individual, local, and collective, global, behaviour. According to the eighth law the software process is such a system. This observation must, therefore, be expected to apply. Thus, the contribution of any activity to the global process may be quite different from that suggested by its open loop characteristics. If the feedback nature of the software process is not taken into account when predicting its behaviour, unexpected, even counter-intuitive, results must be expected both locally and globally. For sound software process planning, management and improvement the feedback nature of the process must be mastered. (Lehman & Ramil, 2001, p. 35)

Within the proposed causal loop diagram presented in Figure 31, the eighth law is captured by the set of the single feedback structures discussed so far and their corresponding interactions that produce emergent behaviors, which are presented and discussed in sections 4.2.1.1 ("Model's subsystem diagram") and 4.2.3 ("Model testing").

4.2.2 Model formulation

The following sections present each of the four subsystems of the proposed simulation model, consisting of their stock and flow diagrams, their underlying reasoning, and their main equations.

The simulation model was developed using version 7.3.4 of the Vensim Professional software (Ventana System, 2018). Its complete documentation, containing the model's source code, is available in Appendix B ("Model documentation") of the present thesis.

4.2.2.1 Maintenance process subsystem

The design and implementation of the maintenance process subsystem was further divided into two distinct smaller subsystems. This decision was taken due to the particularities of the inner dynamic interactions and characteristics of the different types of maintenance activities (perfective, corrective, and preventive). Separating them would enable their differences to be modeled without creating unnecessarily complicated structures.

Therefore, the maintenance process subsystem was modeled as being composed of two different smaller subsystems that were named "Perfective maintenance subsystem" and "Corrective and preventive maintenance subsystem", which are described in the following sections.

4.2.2.1.1 Perfective maintenance subsystem

This subsystem of the model was inspired by previous works published by Taylor and Ford (2006, 2008), in which the authors investigated the impact that reworking had on the performance of individual development projects. They proposed an aging chain structure to model the observed behaviors. An aging chain structure consists of a sequence of levels that are interconnected with conserved flows (i.e., there is no loss or addition from the intermediate stages), in which the levels depict the contents of the chain in different conditions (Sterman, 2000).

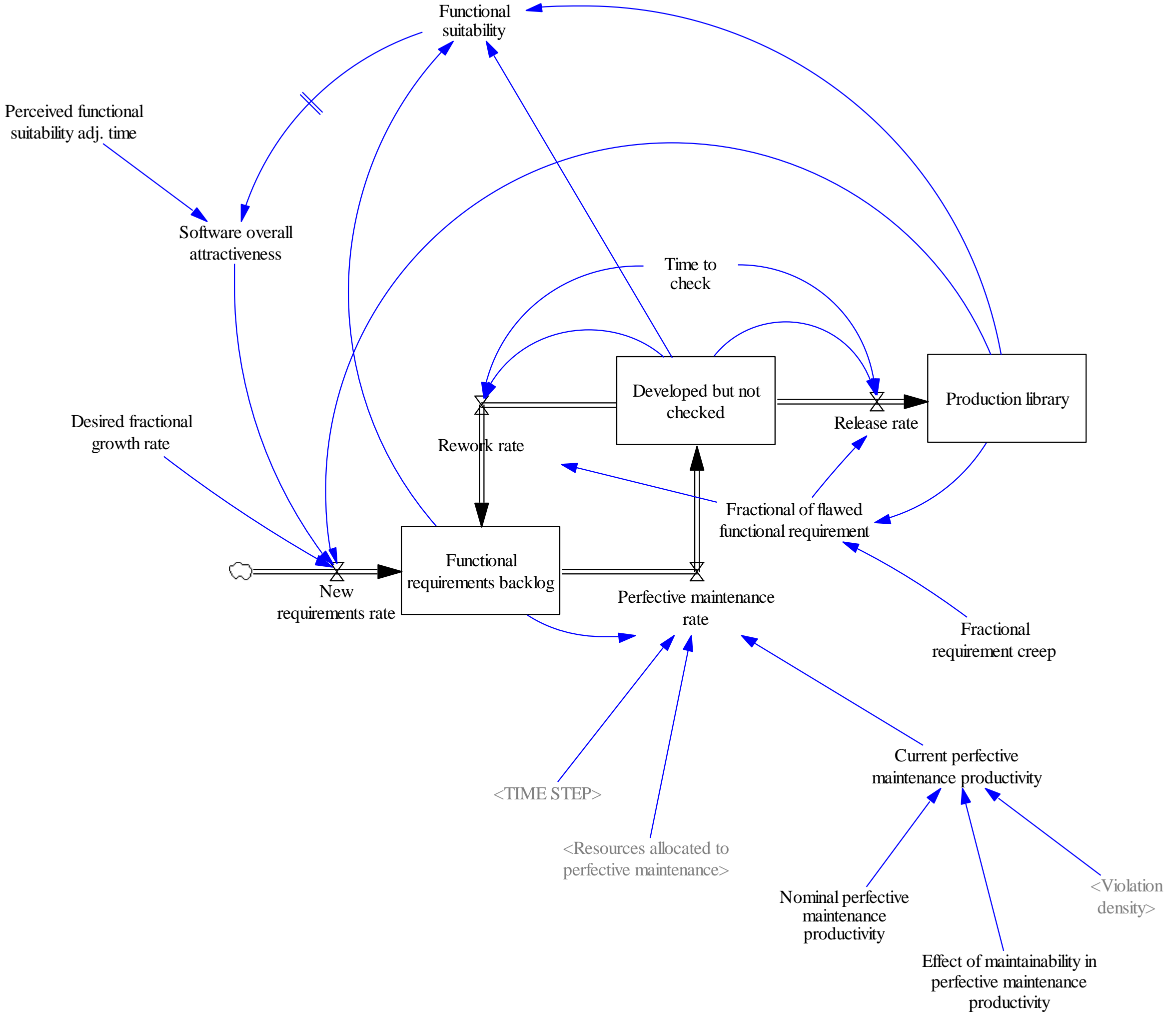
The levels contained in the structure presented in Figure 32 show that the functional requirements accumulated in *Functional requirements backlog* items are transformed over time in to *Developed but not checked* items according to a *Perfective maintenance rate*.

These developed but unchecked items go through a quality inspection process, and, after a delay represented by *Time to check*, they are released into the *Production library*. Due to process performance and maturity, however, some of the developed functional requirements items are flawed and need to be reworked. The fraction of rework items is defined by the *Fractional of flawed functional requirements*; thus, the release and rework rates are computed according to the following equations:

$$Release\ rate(t) = \frac{Developed\ but\ not\ checked(t)}{Time\ to\ check} \cdot (1 - Fractional\ of\ flawed\ functional\ requirements)$$

$$Rework\ rate(t) = \frac{Developed\ but\ not\ checked(t)}{Time\ to\ check} \cdot Fractional\ of\ flawed\ functional\ requirements$$

Figure 32. Perfective maintenance activities subsystem's stock and flow diagram



Source: Author

Simultaneously, new functional requirements are added to the *Functional requirements backlog* according to the *New requirements rate*, thereby closing the previously discussed reinforcing feedback structure representing the sixth software evolution law relating to continuing growth (see Section 4.2.1.3.1).

$$\begin{aligned} &Functional\ requirements\ backlog(t) \\ &\quad = Functional\ requirements\ backlog(t-1) \\ &\quad + New\ requiements\ rate(t) - Perfective\ maitenance\ rate(t) \end{aligned}$$

The *New requirements rate* is influenced by the current software size (*Production library*) and its current overall attractiveness, which can be defined by the following equation:

$$\begin{aligned} &\mathit{New\ requirements\ rate}(t) \\ &\quad = \mathit{Production\ library}(t) \cdot \mathit{Nominal\ growth\ rate} \\ &\quad \cdot f\big(\mathit{Software\ overall\ attractiveness}(t)\big) \end{aligned}$$

The perfective maintenance rate is calculated by the following equations:

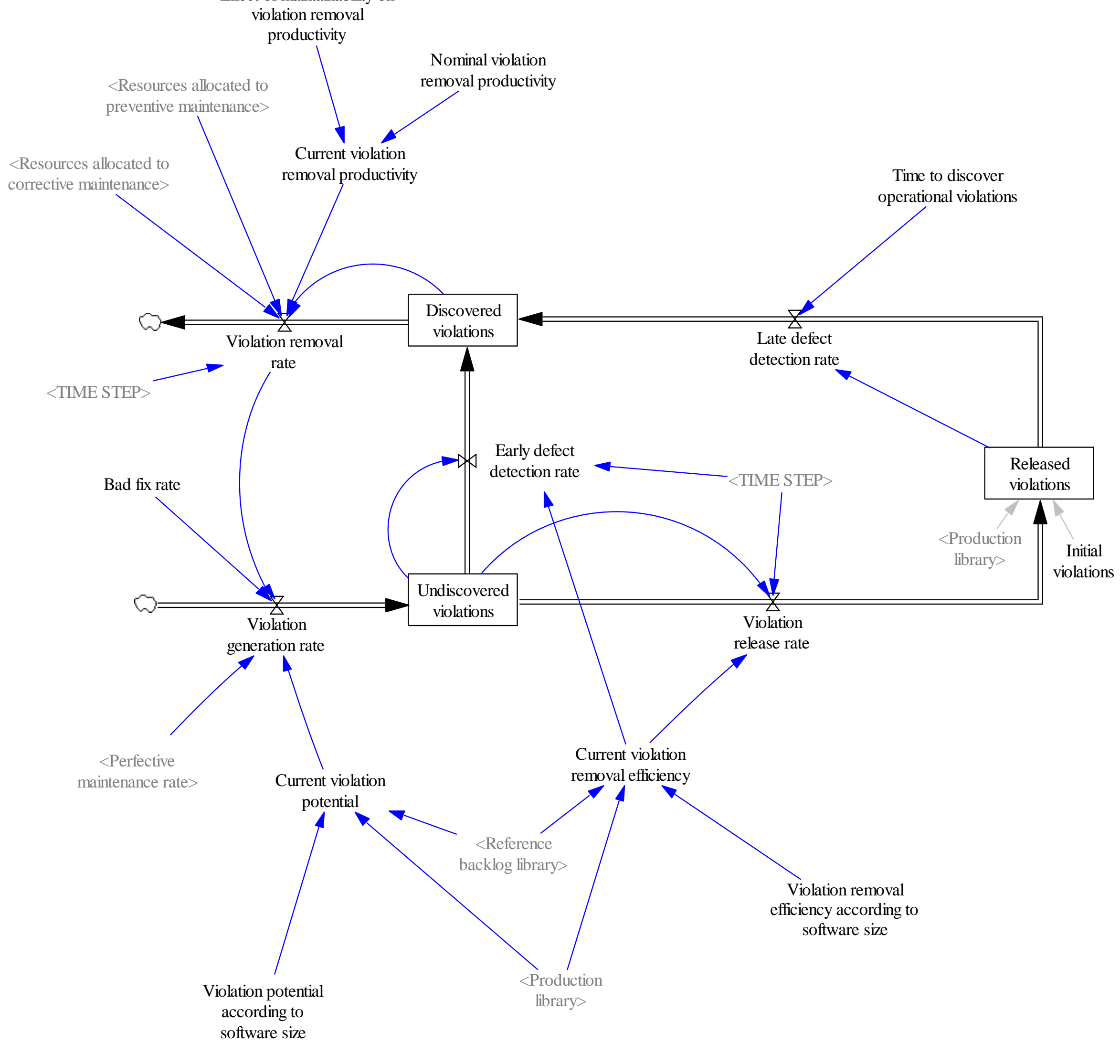
$$\begin{aligned} &\mathit{Perfective\ maintenance\ rate}(t) \\ &\quad = \mathit{Current\ perfective\ maintenance\ productivity}(t) \\ &\quad \cdot \mathit{Resources\ allocated\ to\ perfective\ maintenance}(t) \end{aligned}$$

$$\begin{aligned} &\mathit{Current\ perfective\ maintenance\ productivity}(t) \\ &\quad = \mathit{Nominal\ perfective\ maintenance\ productivity} \\ &\quad \cdot f[\mathit{maintainability}(t)] \end{aligned}$$

4.2.2.1.2 Corrective and preventive maintenance subsystem

The corrective and preventive maintenance subsystem was also implemented as an aging chain but detached from the previous perfective maintenance subsystem. The structure of this subsystem, which is presented in Figure 33, contains a set of three levels, attached to conserved flows that represent how the software system’s violations are handled.

Figure 33. Corrective and preventive maintenance subsystem's stock and flow diagram



Source: Author

The new violations are introduced into the software system according to the *Violation generation rate*, and they are initially accumulated in the *Undiscovered violations* level. Over time, some of these *Undiscovered violations* are identified (i.e., *Early defect detection rate*) and removed (i.e., *Violation removal rate*) before the maintained code is released to the production environment (i.e., the *Release violations* level).

The *Violation generation rate* is proportional to the probability of introducing violations while performing changes to the software code, which is represented by the *Current violation potential* and *Bad fix rate* auxiliary variables.

$$Violation\ generation\ rate(t) = Perfective\ maintenance\ rate(t) \cdot Current\ violation\ potential(t) + Violation\ removal\ rate(t) \cdot Bad\ fix\ rate$$

The efficiency of identifying the violations introduced by the perfective maintenance activities is defined by *Current violation removal efficiency*. Once a violation is identified, it becomes a known violation that has to be addressed, and it flows to the *Discovered violations* level.

The remaining *Undiscovered violations* that are not addressed (i.e., "*1 – Current violation removal efficiency"*) are released to the production library (i.e., *Released violations*) and become latent until the violations are discovered during the software's operation; thus, they also become *Discovered violations*.

$$Released\ violations(t) = Released\ violations(t-1) + Violation\ release\ rate(t) - Late\ defect\ detection\ rate(t)$$

where

$$Violation\ detection\ rate(t) = Undiscovered\ violations(t) \cdot Current\ violation\ removal\ efficiency(t)$$

and

$$Late\ defect\ detection\ rate(t) = \frac{Released\ violations(t)}{Time\ to\ discover\ operational\ violations}$$

The elements contained in the corrective and perfective maintenance subsystem were implemented as arrays (i.e., subscripts in Vensim software). When using arrays, each element can represent a number of different concepts (i.e., it can contain data relating to corrective or preventive violation types). The behavior over time of the three previously described violation levels can generally be described by the following equation:

$$Total\ violations(t) = Total\ violations\ (t-1) + \sum(\Delta Discovered\ violations(t) + \Delta Released\ violations(t) + \Delta Undiscovered\ violations(t))$$

The preventive and corrective maintenance subsystem is modeled as a coflow of the perfective maintenance subsystem structure. A coflow structure is used to keep track of the attributes of various items as they move through the stock and flow structure of the system (Sterman, 2000). In the proposed simulation model, the attributes of the coflow structure consist of the software system's quality attributes relating to reliability, security, and maintainability violations (i.e., rework) that go through the aging chain described in this section and shown in Figure 33.

4.2.2.2 Resource management subsystem

The resource management subsystem is illustrated in Figure 34. This subsystem is responsible for managing the resource allocation within the different maintenance activities types. The allocation policy depends on how much work has to be done on each of the different maintenance types (perfective, corrective, and preventive), the resources available for performing the maintenance activities, and the defined business priorities. The available resource is represented by the *Maintenance team* level.

The fractions of the available resources allocated to each of the different maintenance activities types are represented by the three levels named *Actual preventive maintenance fraction*, *Actual corrective maintenance fraction*, and *Actual perfective maintenance fraction*.

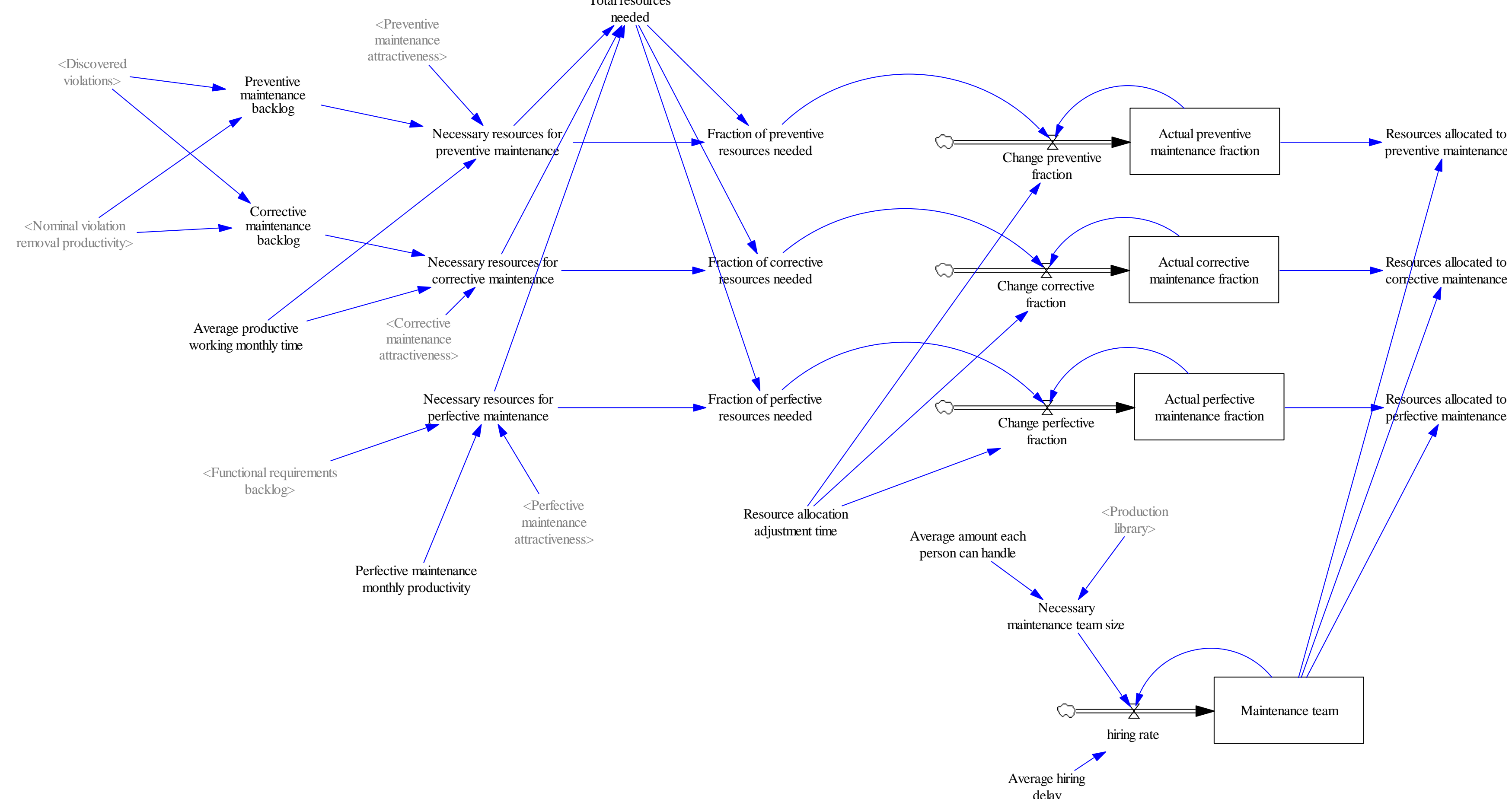
Each of these allocation fractions is computed as the ratio between the necessary effort to address each specific maintenance type (i), compared to the total sum of all the necessary maintenance effort, according to the following equation:

$$Fraction\ of\ resources\ needed_i(t) = \frac{Necessary\ maintenance\ effort_i(t)}{Total\ resources\ needed(t)}$$

The changes to the allocations of resources do not happen immediately; rather, they occur after a time delay. These delays are represented by the three levels presented in Figure

34 (*Actual preventive maintenance fraction, Actual corrective maintenance fraction, and Actual perfective maintenance fraction*), and the auxiliary variable *Resource allocation adjustment time*.

Figure 34. Resource management subsystem's stock and flow diagram



Source: Author

The resource allocation fractions for each maintenance activities type (i) are computed according to the following equation:

$$Actual\ maintenance\ fraction_i(t) = Actual\ maintenance\ fraction_i(t-1) + \frac{Fraction\ of\ resources\ needed_i(t) - Actual\ maintenance\ fraction(t-1)}{Resouce\ allocation\ adjustment\ time}$$

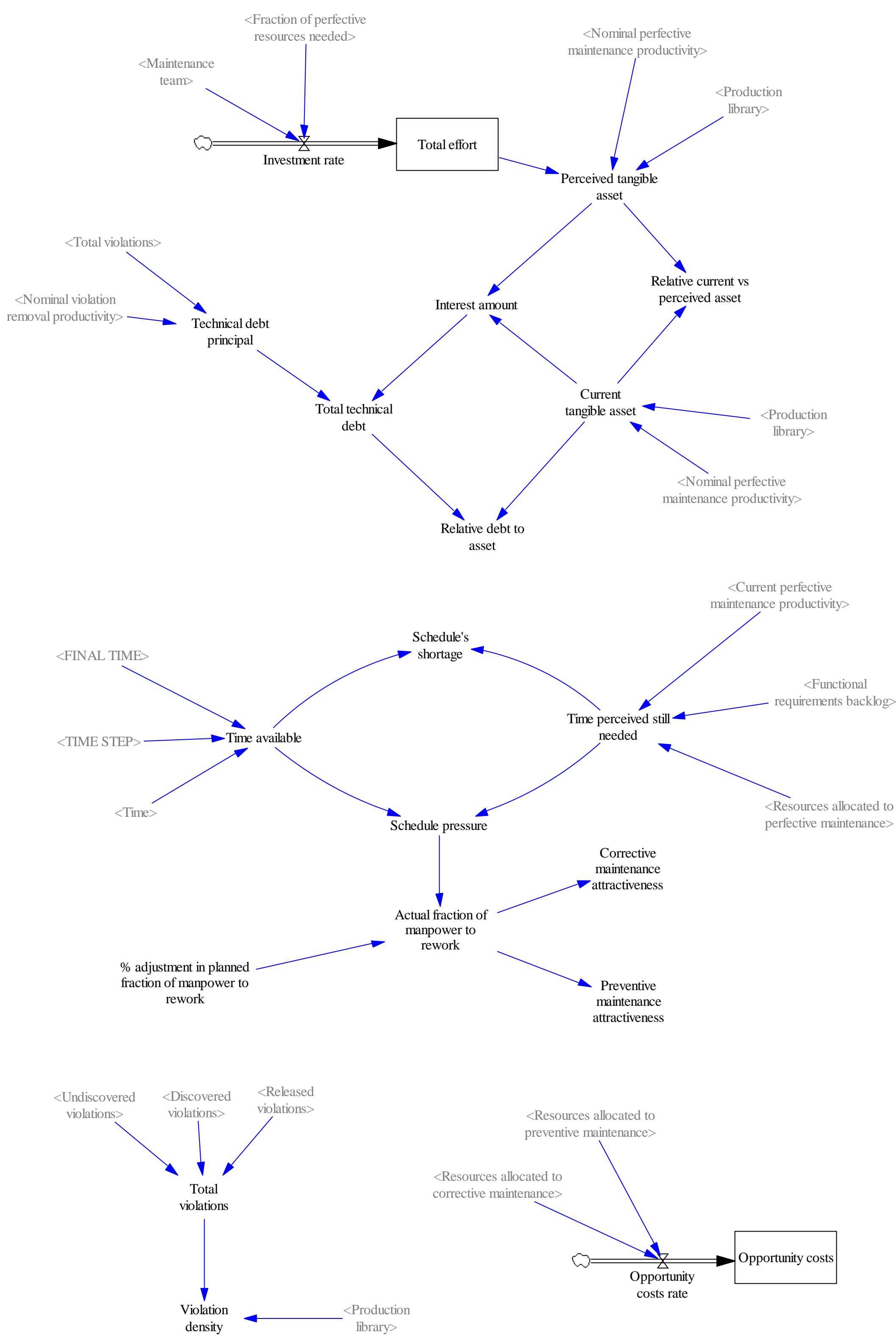
The resources fraction for each maintenance type i is the product of the necessary resources for maintenance type i and its corresponding attractiveness (a scalar number representing how much stakeholders favor each maintenance type), divided by the total necessary resources for all maintenance types (perfective, corrective, and preventive):

$$Resource\ allocated\ to\ maintenace_i(t) = \frac{Necessary\ resources\ for\ maintenance\ fraction_i(t)}{Total\ necessary\ resources(t)}$$

4.2.2.3 Goal evaluation subsystem

The third subsystem of the proposed model is responsible for evaluating the sustainability goals according to the previously proposed hierarchical evaluation structure (shown in Section 4.1, “Hierarchical software sustainability evaluation structure”). Essentially, all the structures contained in this subsystem are proxies for other elements of the model for computing the key metrics of interest. Their formulation can be seen in Appendix B – Model documentation, and their graphical notations are shown in Figure 35.

Figure 35. Goal evaluation subsystem stock and flow diagram



Source: Author

4.2.3 Model testing

After the model was designed and implemented, it was subject to a set of tests to assess its adequacy. The procedures used for each test on the model, were previously described in Section 3.3.1.4. They are listed, along with the results, in Table 9.

Table 9. Summary of the tests performed on the model

Test	Major purpose of test	Procedure conducted in this research study	Results
1. *Boundary adequacy*	Ensures that important concepts and elements are considered and included into the model	Extant literature review was thoroughly reviewed and the model's causal diagram and subsystem diagram were checked against existing published literature	Model was improved based on feedback received
2. *Structure assessment*	Ensures that the model structure is consistent with the relevant descriptive available knowledge of the system	Major relationships, input variables, and output variables were reviewed. The Vensim "Model Check" was also used for assessing the model structure against semantic errors.	Passed, no error was identified.
3. *Dimensional consistency*	Checks if each equation is dimensionally consistent	Used the Vensim "Units Check" dimensional utility and manually inspected all of the model's equations.	Passed, no error was identified.
4. *Parameter assessment*	Checks if parameters values are consistent with relevant descriptive and numeric knowledge of the system	Evaluated each variable in the model to make sure that each of them has a corresponding real-life meaning. The values of the variables are based on 1) a careful and detailed analysis of the literature, and 2) judgmental estimation.	Passed

5.	*Extreme conditions*	Ensures that each equation makes sense on extreme input values	Inspected each equation Tested the model's response to extreme values of each input	Passed, model's equations made sense when extreme values were used.
6.	*Integration error*	The model outputs are sensitive to the time step	The time step used in the proposed model was one month long and integration step to 0.0625.	Passed, the integration step "*dt*" was set as small as no furthers significant changes were observed in the model's outputs.
7.	*Behavior reproduction*	Checks if the model reproduces the behavior captured by the stated reference modes.	Compared model behavior with secondary real project behavior sets (using published data from previous published studies).	The reproduction of the reference modes shown in Section 4.2.1 are discussed in Section 5.1.
8.	*Behavior anomaly*	Establishes the significance of important relationships by examining whether anomalous behavior arises when the relationship is deleted or modified	When different maintenance closed loops are removed from the model, the output from the model's simulation exhibits anomalous behavior.	Passed
9.	*Family member*	Ascertain whether the model can generate the behavior of other instances in the same class as the system the model was built to mimic	Three different resource allocation policies were simulated, and the results are discussed in Chapter 5.	Performed well and the results are discussed in Chapter 5.
10.	*Surprise behavior*	Evaluates unexpected behavior	The model did not show behaviors that were significantly different from what was expected	Performed well as the model was able to reproduce the expected behaviors, and the

			results are discussed in Chapter 5.
11. Sensitivity analysis	Assesses the impact of changing the model's assumptions	Univariate sensitivity analysis was performed on several of the model's element (e.g., violation potential, maintenance productivity, initial conditions)	Passed. The model demonstrated behavior sensitive to reasonable variations in parameters (numerical sensitivity) and alternative structures (behavior mode sensitivity).

Source: Adapted from Sterman (2000)

4.2.4 Policy formulation and evaluation

The final step of the iterative model development process, as proposed by Sterman (2000), is policy formulation and evaluation. The results obtained from the scenario simulations, including the analysis of the model's capacity to reproduce the reference modes formulated in Section 4.2.1 ("Problem articulation and dynamical hypothesis"), are presented in Chapter 5 ("Results and discussion").

4.3 Chapter summary

This section summarizes the content presented in this chapter with the purpose of organizing the knowledge acquired with the development of the proposed dynamical evaluation framework and the simulation model for evaluating resource allocation policies in maintenance activities.

The relevant contributions of this chapter have been:

- It has presented the *dynamical evaluation framework proposed* for evaluating and supporting decision-making in maintenance investments.
- It has developed and discussed two *hierarchical software sustainability evaluation structures* that were used to evaluate, from technical and economic perspectives, how difference resource allocation scenarios behaved over time.

- It has detailed the steps taken to build the *proposed simulation model* following the system dynamics approach and it has shown the corresponding artifacts produced to document the model's decisions and assumptions.

5. Results and discussion

This chapter presents and discusses the results obtained from the simulation of the proposed model, according to pre-defined software maintenance scenarios.

5.1 Model evaluation

After the proposed model had been developed and tested, and according to the iterative development process adopted (described in Section 3.3, "Research process"), the "policy formulation and evaluation" phase began. In this phase, the model was assessed against previous preselected and predefined resource allocation scenarios.

The following sections describe three different scenarios, depicting different resource allocation policies on software maintenance activities, and then discuss the obtained outcome behaviors according to the hierarchical evaluation structure of the software's technical and economic sustainability, which was previously discussed in Section 4.1 ("Hierarchical software sustainability evaluation structure").

5.1.1 Scenario #1: Perfective maintenance focus

The first evaluated scenario was named the "Perfective maintenance focus". It represents the context in which the stakeholders' mindsets are mainly focused on the delivery of functional requirements, thereby neglecting the long-term effects of the productivity hindrances due to the increasing density of software quality violations (especially the maintainability violations), which directly impact the productivity of the different types of maintenance activities (perfective, corrective, and preventive).

To simulate the first proposed scenario, the initial conditions of several of the model's elements must be defined. The values shown in Table 10 are the defined values, which were obtained from published literature on productivity and quality of software measurement (Abdel-Hamid & Madnick, 1991; Jones, 2008), available online data repositories containing software metrics and benchmarking related to software development and maintenance from

the International Software Benchmarking Standards Group[9] (ISBSG), and the existing empirical data available online at the SonarQube[10]. These used data are available in Appendix C – Secondary data used.

Table 10. Model's initial conditions for Scenario #1

Model's element	**Initial value**
Desired fractional annual growth rate	12% per year
Initial delivered defects	0.55 per function point (Jones, 2008)
Fractional requirement creep	Non-linear function that depends on the *Production library* size and was estimated by Jones (2008), see Appendix B – Model documentation.
Violation potential	
Violation removal efficiency	
Production library	1.000 function points
Functional requirements backlog	200 function points
Initial maintenance team size	Two people, which is the production library divided by the average size a maintenance team member can handle, which is 500 function points (Jones, 2008).
Nominal perfective maintenance productivity	5.68 function points / person / month
Nominal corrective maintenance productivity	$\frac{1}{0.36\ person-day} \cdot (5\ days * 4\ weeks) \cong 55{,}56$ corrective violations / person / month (Abdel-Hamid & Madnick, 1991).
Nominal preventive maintenance productivity	$\frac{1}{0.54\ person-day} \cdot (5\ days * 4\ weeks) \cong 21.85$ preventive violations / person / month (Abdel-Hamid & Madnick, 1991).
Number of corrective violations	Based on the defect removal efficiency identified by Jones (2008) of 91% related to the software size calculated in function points, the initial value was estimated for corrective violations.
Number of preventive violations	Based on the defect removal efficiency identified by Jones (2008) of 91% related to the software size

9 http://www.isbsg.org

10 http://sonarcloud.io

	calculated in function points, the initial value was estimated for preventive violations.
Perfective maintenance attractiveness	0.8
Preventive maintenance attractiveness	0.1
Corrective maintenance attractiveness	0.1

Source: Author

The time horizon adopted for the model simulation was 120 months (i.e., 10 years), representing the average age of software applications still in use (Jones, 2008). The outputs obtained after the simulation, according to the first scenario model's setup, are shown in Figure 36.

The graph in Figure 36 compares the behavior over time of the model's two variables of nominal and current perfective maintenance. The obtained result resembles the productivity exponential decay depicted in the reference mode and shown in Figure 16. The initial gap between both productivities is due to the existing initial preventive violations that are present in the software at the beginning of its operation.

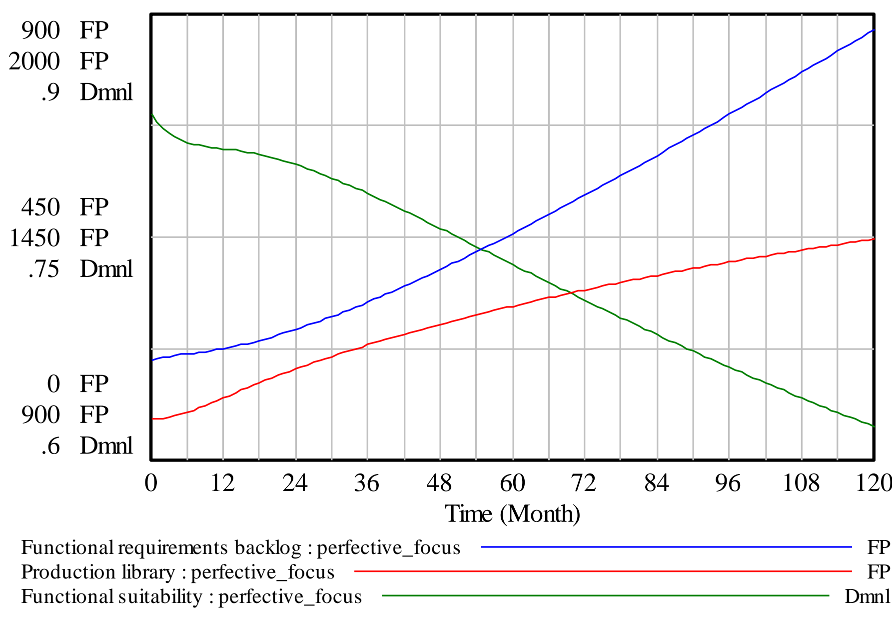
Figure 36. Nominal versus current perfective maintenance productivity for Scenario #1

Source: Author

Figure 37 relates the behavior presented on the left side of Figure 19 (i.e., the higher technical debt context), where the software size rapidly grows in its early life stage (the takeoff phase) and slows almost to a halt in the saturation phase. It is possible to see the functional requirements backlog (blue line) slowly building up until month 18, and then the functional requirement backlog starts to build up with an increased rate.

These changes in the behavior over time of the functional requirements backlog and the production library can also be seen in the functional suitability attribute. Over time the software becomes less attractive to its stakeholders as their desired functional requirements do not get into the operational production environment (represented by the production library).

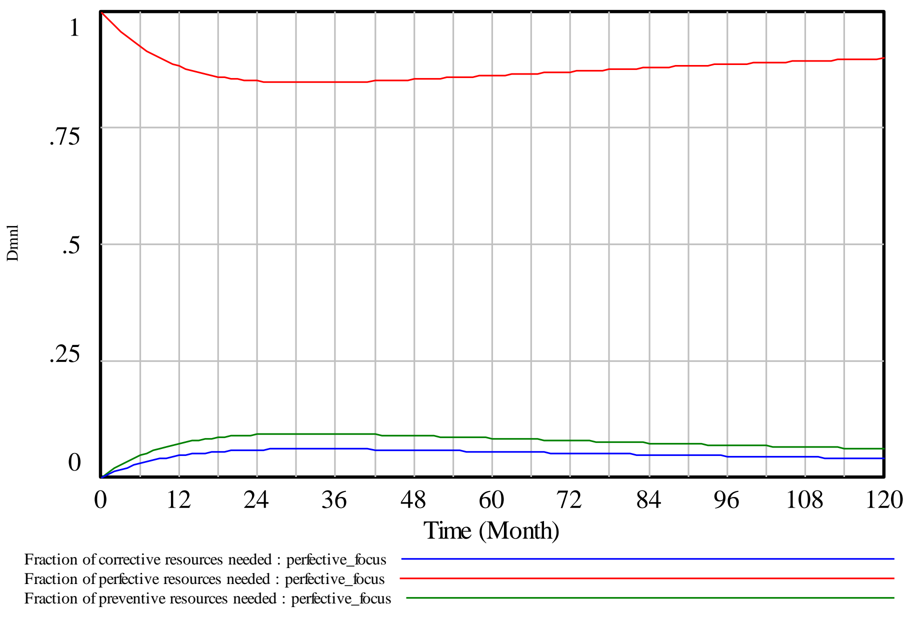
Figure 37. Functional requirements growth pattern for Scenario #1



Source: Author

Figure 38 shows how the resource allocation fractions for each of the maintenance types unfold over time.

Figure 38. Resource allocation fractions for Scenario #1



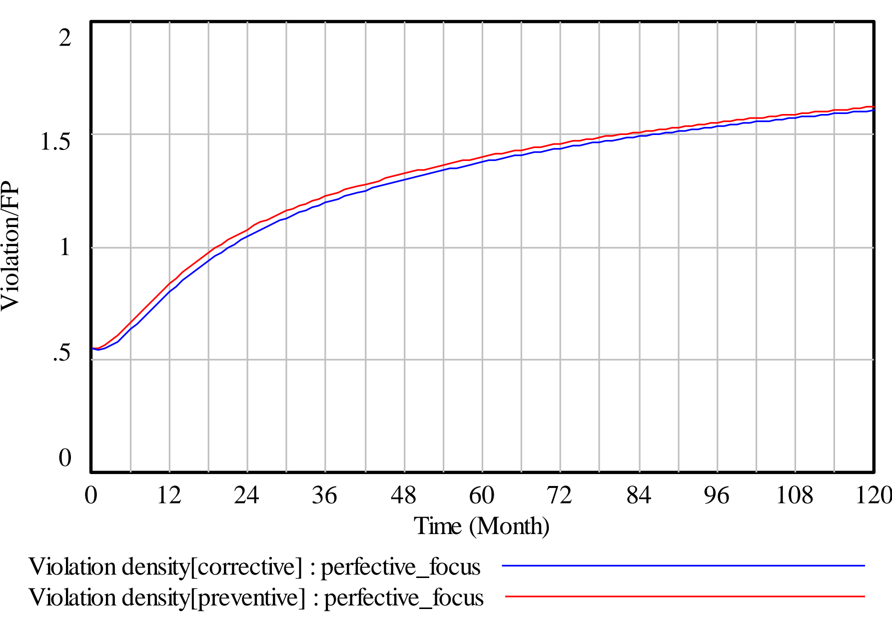
Source: Author

As expected, the majority of the resources were allocated to the perfective maintenance activities. The growth in the allocation fraction to corrective and preventive maintenance activities at the beginning of the simulation was due to the growing number of quality violations, but their growth was limited to the attractiveness shown in Table 10.

After the perfective maintenance fraction reaches its lowest value (near month 24), it started rising up again as the functional requirements backlog also started to accumulate due to the loss in the perfective maintenance productivity due to the preventive violations (technical debt) previously shown in Figure 36.

Figure 39 shows that the density (i.e., the number of violations divided by the software's production library) of both preventive and corrective violations rapidly grew until near saturation.

Figure 39. Preventive and corrective violations density for Scenario #1



Source: Author

This halt on the rapidly growing rates of the density of violations is related to the perfective maintenance productivity erosion and to the declining functional suitability, which directly impacts the software's attractiveness and thus the desire for new functionalities to be added to the functional requirements backlog. Violations are introduced into the software by the changes made to it, independently of the type of modification (perfective, preventive, or corrective – according to the auxiliary variables defect potential and bad fixes rates that are shown in the stock and flow diagram in Figure 33).

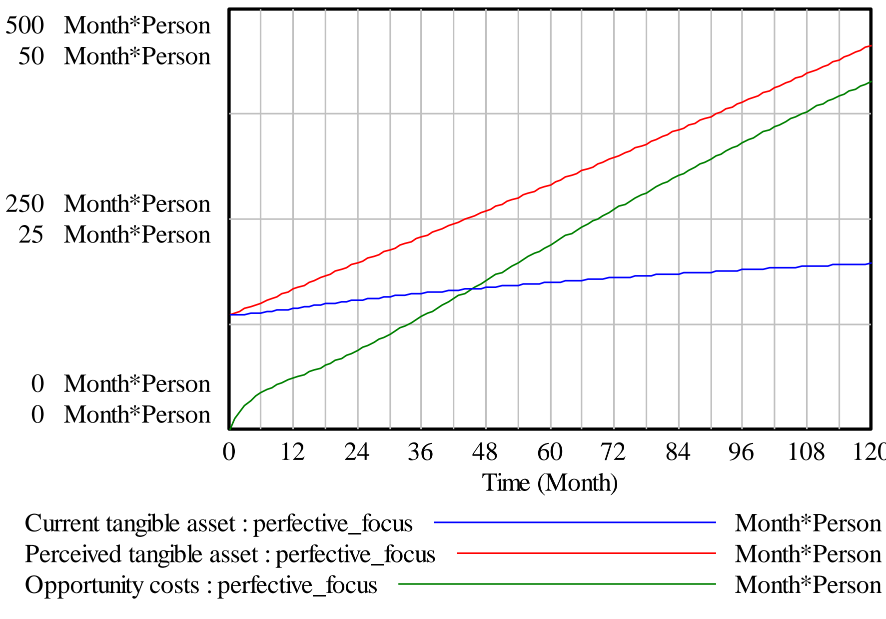
Next, the metrics related to the economic sustainability were plotted and then analyzed. The first set of variables comprehended the current tangible asset (computed as the effort to develop from scratch the available functionalities in production debt according to the nominal productivity), the perceived tangible asset (corresponding to the sum of effort spent during the software development and the effort spent in perfective maintenance activities), and the opportunity costs, which are the effort allocated to other maintenance types. The results of these variables when simulating the first scenario are illustrated in Figure 40.

It is easy to notice the gap between the current and the perceived tangible asset, where the latter reaches almost four times the real current value of the estimated effort to develop

the software running in the production library. This gap can be justified based on the previous discussions in this section. Moreover, the allocation fraction in perfective maintenance remains within a stable range over time (see Figure 38), and the productivity of the perfective maintenance activities exponentially decay very quickly (see Figure 36).

The perceived tangible asset became greater than the current tangible asset because the rising preventive violation density (see Figure 39) eroded the perfective maintenance productivity and the stakeholders no longer perceived the results of their investment in the software production library size. At the end of the simulation timeframe, the ratio between the current and the perceived tangible asset reached almost 0.431. This indicates that only 43.1% of the investments made in perfective maintenance activities (resources allocated) turned into real tangible assets (i.e., functional requirements developed and deployed in the production library).

Figure 40. Tangible and perceived asset, and opportunity costs for Scenario #1



Source: Author

The green line shown in Figure 40 represents the opportunity costs that build up over time, as resources are allocated to other activities rather than to perfective maintenance. As the simulated resource allocation policy for the first scenario prioritized the allocation to perfective maintenance, the opportunity costs remained lower than 25 person-month until the simulation reached the 120 months' time horizon.

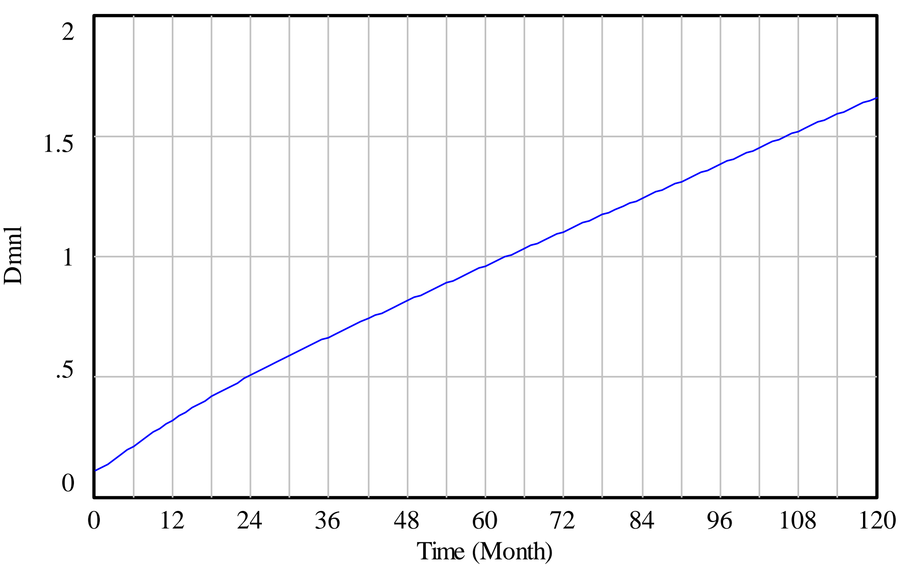
Figure 41 depicts how technical debt's principal and interest evolves over time. This graph shows when the equilibrium point was reached, which was previously illustrated in Figure 21. This point is represented by the intersection between the principal and the accumulated interest lines, and it shows the moment when the amount of effort borrowed from technical debt's principal has been spent on extra maintenance effort due to the software's decreased maintainability.

Figure 41. Technical debt's principal and interest for Scenario #1

Source: Author

Figure 42 illustrates how the ratio between technical debt (i.e., the sum of technical debt's principal and interest) and the current tangible asset evolves over time. In finance this metric is usually known as the debt-to-equity ratio and it indicates how much of the software maintenance is leverage (i.e., it measures the degree to which a company is financing its maintenance plan through debt). At month 64, the total asset owned by the company is equal to the total technical debt (liability) it has accumulated over time. The total technical debt comprises the principal plus the interest.

Figure 42. Relative debt to current asset for Scenario #1



Source: Author

The final conditions of the model’s elements, when simulated in the perfective maintenance focus scenario, are shown and discussed in Section 5.2 (“Scenarios comparison”), where it is also compared to the other simulated scenarios according to the proposed GQM hierarchical evaluation structure (described in Section 4.1, “Hierarchical software sustainability evaluation structure”).

5.1.2 Scenario #2: Preventive maintenance focus

The second evaluated scenario focuses on investment in preventive maintenance activities, that is, on improvements to the software’s maintainability and thus to improving or maintaining the software’s maintenance productivity over the software’s lifetime.

The initial conditions for simulating the second scenario were the same as those shown in Table 10, except for the maintenance types attractiveness, the adopted values of which are shown in Table 11. Within this simulation setup, the resource allocation policy favors the demand for preventive maintenance activities and thus the technical debt repayment.

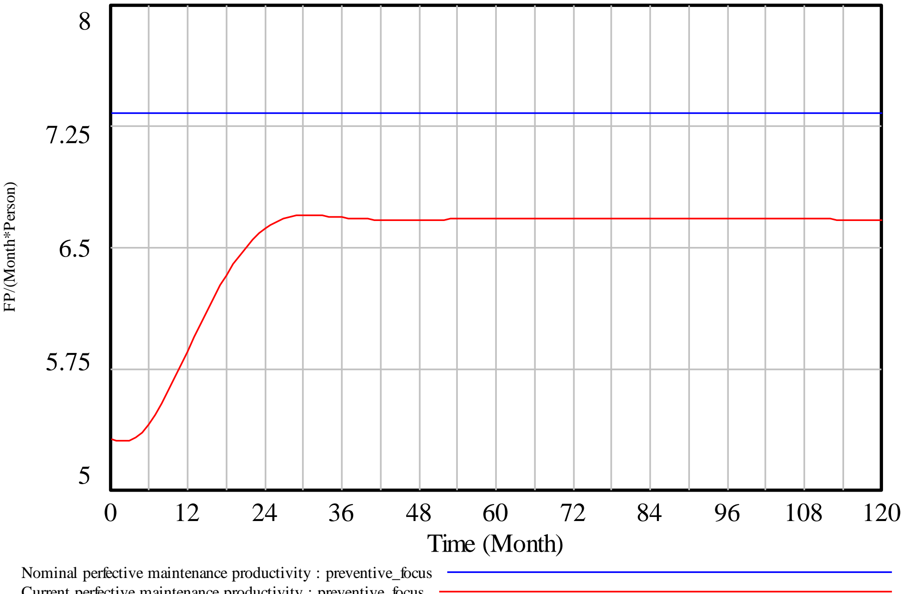
Table 11. Model's initial conditions for Scenario #2.

Model's element	**Initial value**
Perfective maintenance attractiveness	0.1
Preventive maintenance attractiveness	0.8
Corrective maintenance attractiveness	0.1

Source: Author

Figure 43 shows how the current perfective maintenance evolves over time in the simulation of the second scenario. Contrary to the behavior seen in the simulation of the first scenario (Figure 36), the productivity increases with the preventive maintenance focus until almost month 24, when the preventive violations are removed from the software, thereby increasing the software's maintainability.

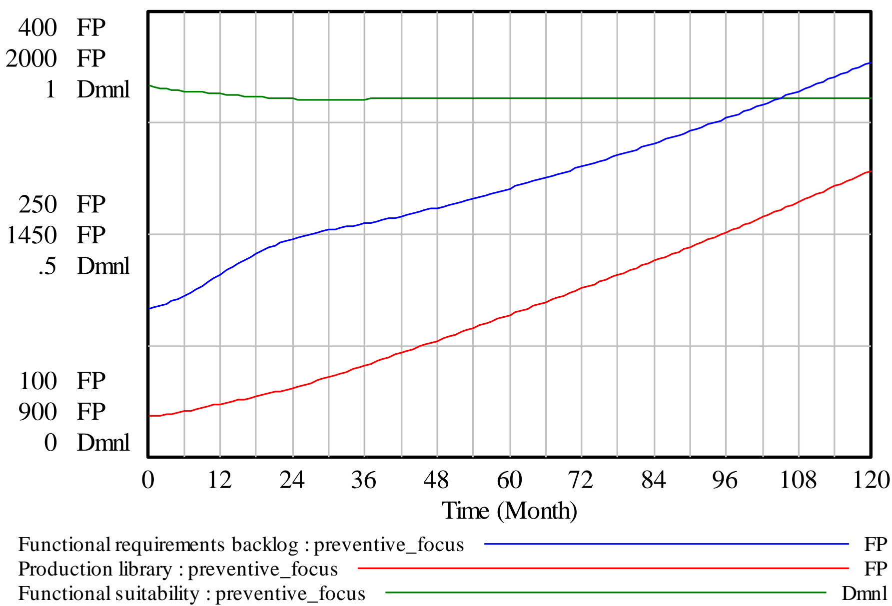
Figure 43. Nominal versus current perfective maintenance productivity for Scenario #2



Source: Author

As the behavior of the perfective maintenance's productivity unfolds differently in the two simulated scenarios, it is expected that the growth patterns of the production library, functional requirements backlog, and functional suitability behave differently over time. The time series for these three variables are shown in Figure 44.

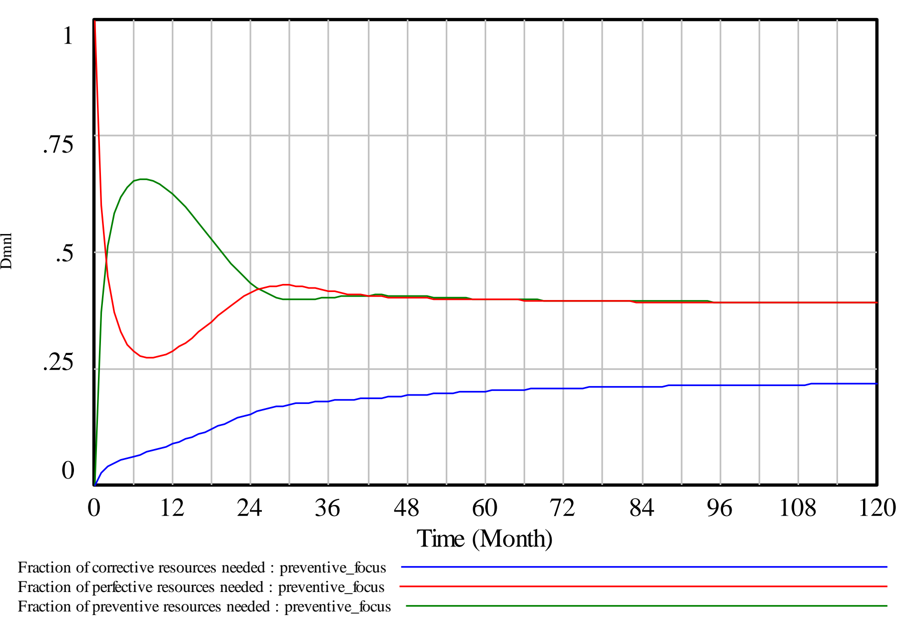
Figure 44.Functional requirements growth pattern for Scenario #2



Source: Author

The production library line show in Figure 44 presents a growth pattern that resembles the previously discussed one on the right side of Figure 19 (lower technical debt strategy). It consists of a slower growth of the software size in the early states that can eventually be paid off due to faster growth rate in later stages. The decrease in the functional suitability at the beginning can be justified by the initial focus on preventive rather than perfective maintenance activities, which are shown in Figure 45. After month 30, the allocation fraction of these two maintenance types almost reaches a near stable level, causing the software's functional suitability also to remain stable after month 24.

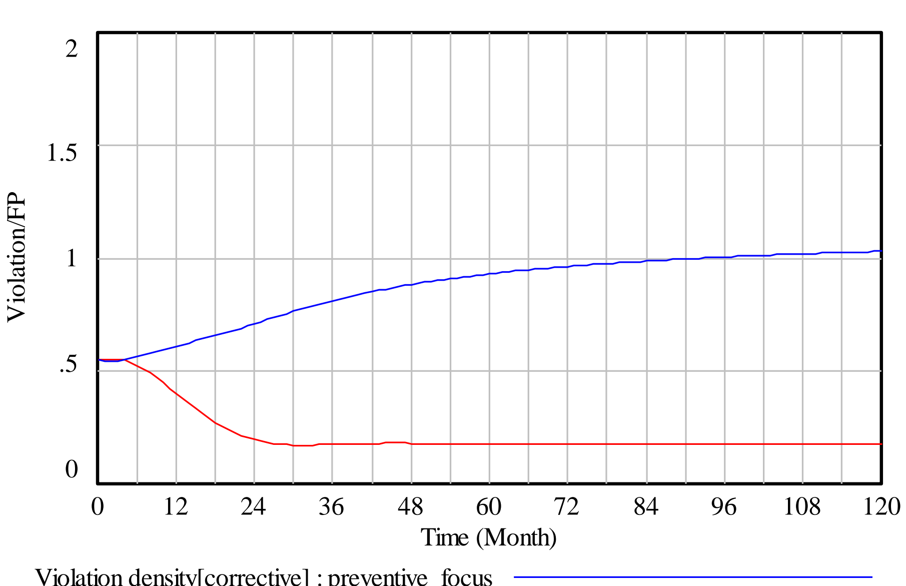
Figure 45. Resource allocation fractions for Scenario #2



Source: Author

The time behavior of the accumulated number of corrective and preventive violations are shown in Figure 46.

Figure 46. Preventive and corrective violations density for Scenario #2

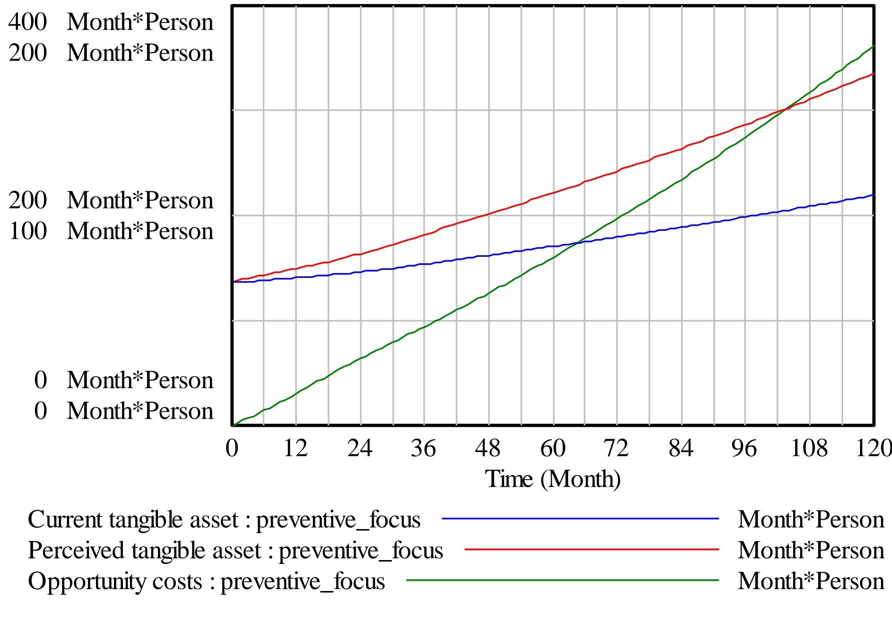


Source: Author

As was expected from a preventive maintenance policy, the removal of preventive violation was prioritized, thus making the preventive violation density decrease until reaching a stable level near month 24. This behavior supports the productivity pattern shown in Figure 43, where the perfective maintenance increases until also reaching a stable value near month 24.

The gap between the current and perceived tangible assets shown in Figure 47 is smaller when compared to the first scenario, where their ratio reaches 0.654 at the end of the time horizon simulation. This bigger ratio is again justified by the lower erosion of the perfective maintenance productivity due to the removal of preventive violations to preserve the software's maintainability.

Figure 47. Tangible and perceived asset, and opportunity costs for Scenario #2



Source: Author

However, the opportunity costs line shown in Figure 47 reaches a value more than for times greater than in Scenario #1. These higher opportunity costs are caused by a purposeful decision taken to allocated resources to preventive maintenance (to pay technical debt's, thereby reducing the preventive violation) instead of allocating them to perfective maintenance, which focuses in delivering new functional requirements to the production library.

The patterns of the technical debt's principal and interest amount for Scenario #2 are shown in Figure 48. It is possible to note the decrease in the technical debt's principal value, as the preventive maintenance does some preventive violation removals. These removals change the slope of the interest amount, thereby reducing the amount incurred at the end of the simulation time horizon, according to behavior previously shown in Figure 19.

Figure 48. Technical debt's principal and interest for Scenario #2

Source: Author

Figure 49 shows the total technical debt (principal plus interest amount) to asset ratio for Scenario #2. Conversely, the results for Scenario #2 show that the total asset owned by the company in the production library was always bigger than the liability of the technical debt, indicating that it was still cheaper to maintain operation of the current software rather than to develop an alternative software from scratch.

However, when comparing the behavior shown in Figure 21 and Figure 48, it is possible to notice a difference between the two technical debt curves depicted. In Figure 21, which represents a theoretical context, the technical debt principal remained constant over time (represented by P_0), regardless that some refactoring efforts were employed towards repaying some of the incurred technical debt. On the other hand, Figure 48 showed that after repaying part of the incurred debt, the technical debt principal level decreased.

As the technical debt principal level decreased when simulating Scenario #2, the equilibrium point was reached before then it was in Scenario #1 (Figure 41). This obtained behavior was different from the depicted in Figure 21, where after making part of the debt repayment, the equilibrium would occur after then making no repayment at all (equilibrium points E_0 and E'). Hence, the different slopes from the interest rates shown in Figure 21 comparing the impact of repaying part of the technical debt ($\sum(Im)$ and $\sum(Im)'$) were reproduced by the proposed simulation model, and it can be seen in Figure 41 and Figure 48.

Figure 49. Relative debt to current asset for Scenario #2

Source: Author

The final conditions of the model's elements, when simulated to the preventive maintenance focus scenario, are shown and discussed in Section 5.2 ("Scenarios comparison"), where this scenario is also compared to the other simulated scenarios according to the proposed GQM hierarchical evaluation structure (described in Section 4.1, "Hierarchical software sustainability evaluation structure.

5.1.3 Scenario #3: Corrective maintenance focus

The third analyzed scenario focuses on investing in corrective maintenance activities in order to reduce the reliability violations incurred during the development and maintenance activities.

The initial conditions for simulating the third scenario were the same as those shown in Table 10, except for the maintenance types attractiveness, the adopted values of which, are shown in Table 12. Within this simulation setup, the resource allocation policy favors the demand for corrective maintenance activities.

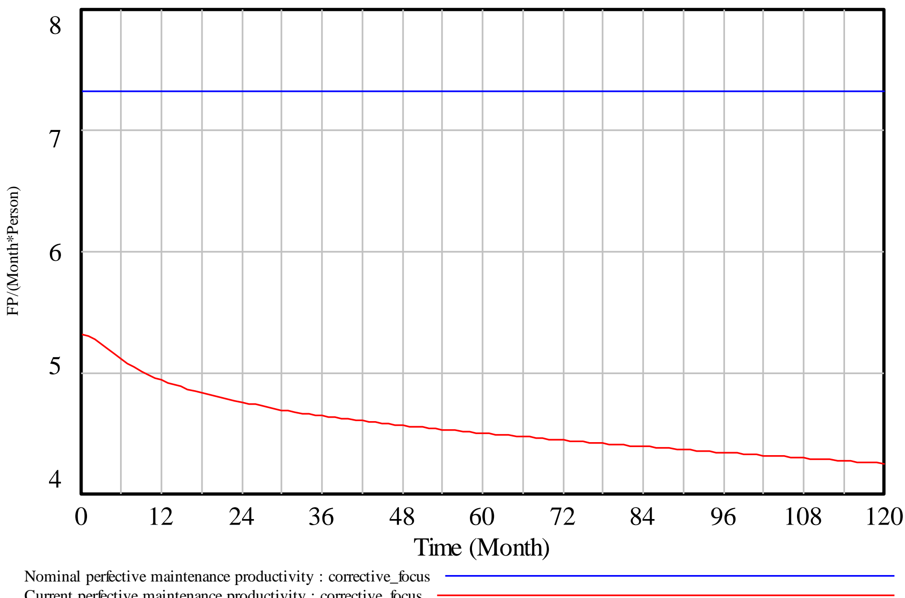
Table 12. Model's initial conditions for Scenario #3.

Model's element	**Initial value**
Perfective maintenance attractiveness	0.1
Preventive maintenance attractiveness	0.1
Corrective maintenance attractiveness	0.8

Source: Author.

As with the perfective maintenance productivity in the second scenario, the productivity erosion in Scenario #3 was smaller than in the first scenario; however, it was higher than in the second scenario, as can be seen in Figure 50.

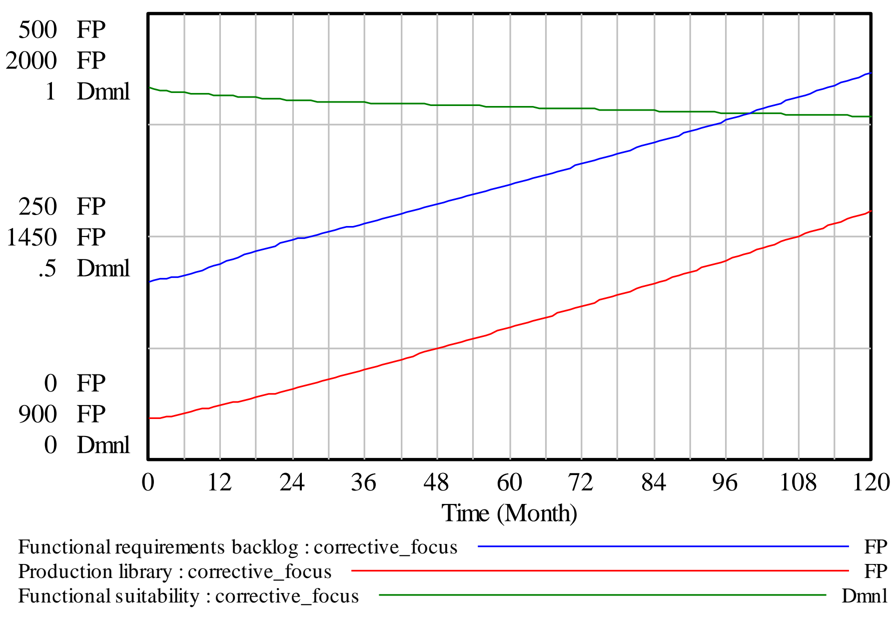
Figure 50. Nominal versus current perfective maintenance productivity for Scenario #3



Source: Author

Hence, as was expected, the values for the functional requirements backlog, production library and functional suitability were between the values previously obtained for the first two simulated scenarios as shown in Figure 51.

Figure 51. Functional requirements growth pattern for Scenario #3

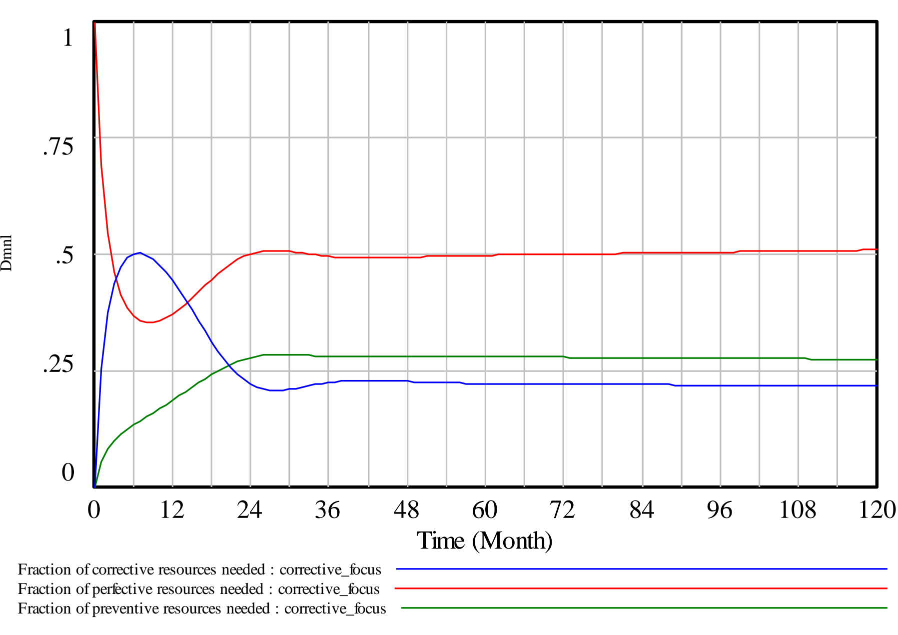


Source: Author

Unlike what happened in the first scenario, but like what happened in the second scenario, the gap between the functional requirements backlog and production library remained almost constant after month 24. This caused the functional suitability also to remain almost stable.

The equilibrium of this gap, and also the functional suitability, can be justified by the behavior seen in Figure 52. Until month 24, there were oscillations between the allocation fractions among the different maintenance types. However, after month 24, they remained, albeit with different levels, at a stable allocation for maintaining the behaviors of the time series, as shown in Figure 51.

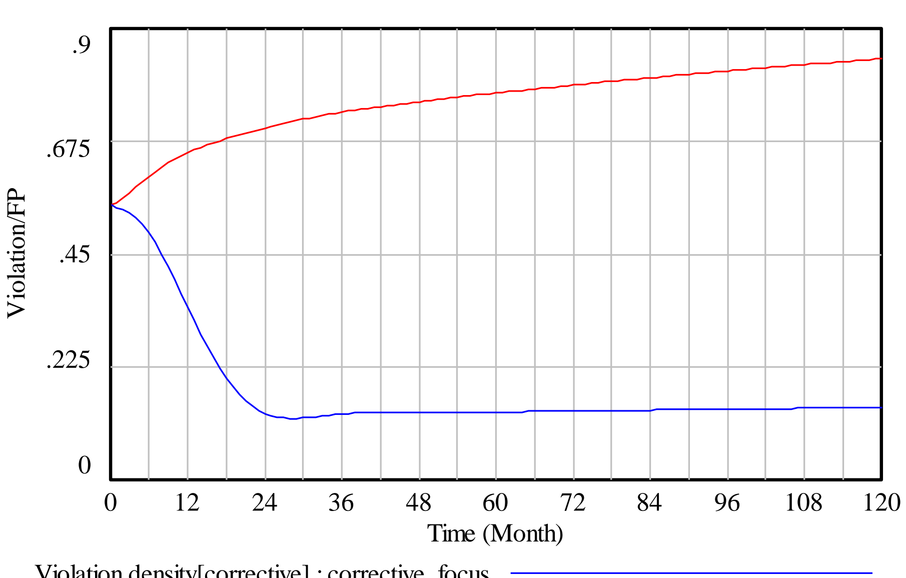
Figure 52. Resource allocation fractions for Scenario #3



Source: Author

As the third simulated scenario focused on corrective maintenance activities, it was expected to see the patterns shown in Figure 53.

Figure 53. Preventive and corrective violations density for Scenario #3

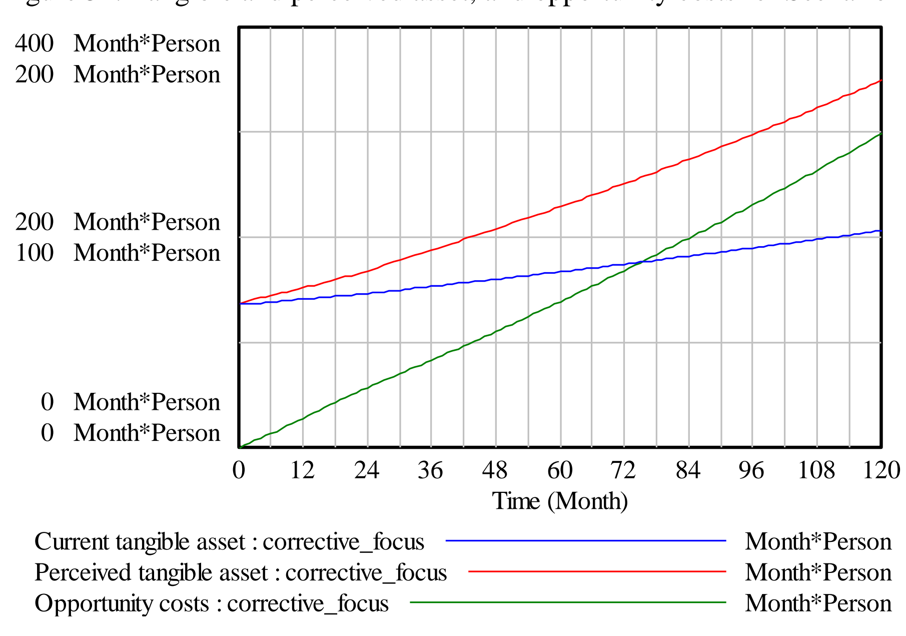


Source: Author

Figure 53 shows a fast rate of corrective violations removal, but it also shows a smaller density of preventive violations. Both perfective and corrective violations were introduced in the software through manipulations of the software's source code (maintenance activities).

As the production library grew at a slower rate at the beginning of the simulation time horizon, as shown in Figure 54, the rate of preventive violation introduction remained slower too, thereby causing the gap between current and perceived tangible assets to end up with a value of 0.591, which was between the values for the first and second simulated scenarios.

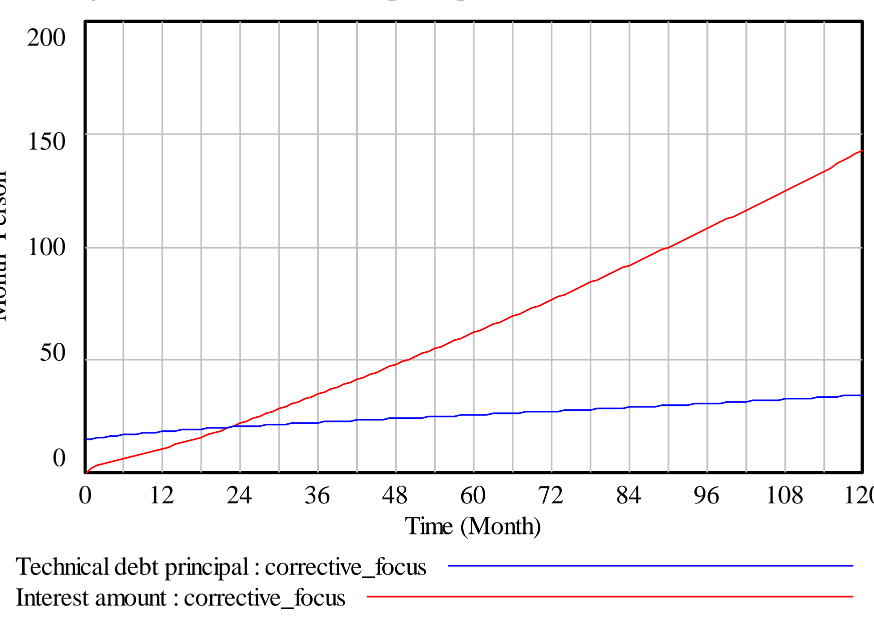
Figure 54. Tangible and perceived asset, and opportunity costs for Scenario #3



Source: Author

Figure 55 shows that the equilibrium point was reached later in the third scenario simulated, occurring near month 24.

Figure 55. Technical debt's principal and interest for Scenario #3

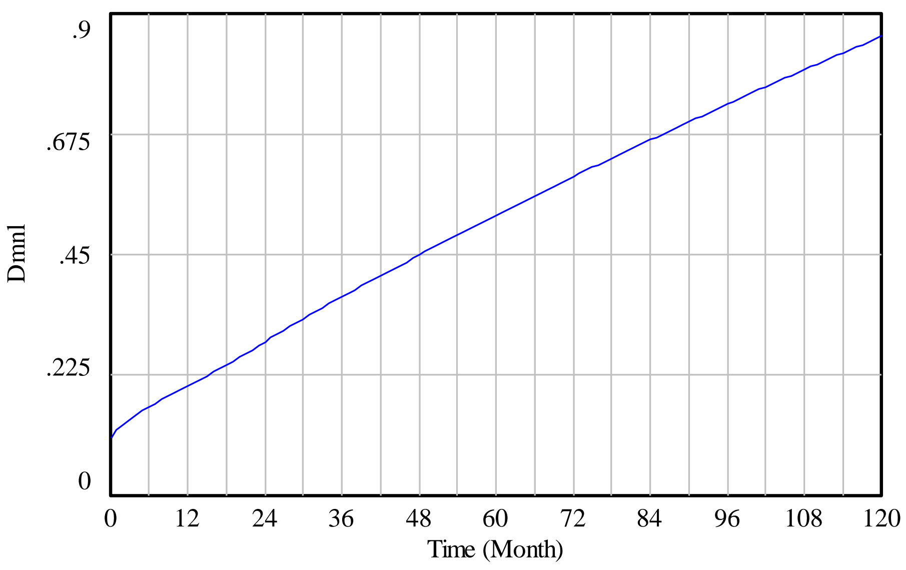


Source: Author

This was because there was no focus on repaying the technical debt, and hence on reducing its principal, and also because the rate of accumulation of interest was lower than in Scenario #1, but higher than in Scenario #2.

As with the second scenario, in Scenario #3 the total liability did not reach the total tangible asset owned by the company, as shown in Figure 56.

Figure 56. Relative debt to current asset for Scenario #3



Source: Author

The final conditions of the model's elements, when simulated to the corrective maintenance focus scenario, are shown and discussed in Section 5.2 ("Scenarios comparison"), where the scenario is also compared to the other simulated scenarios according to the proposed GQM hierarchical evaluation structure (described in Section 4.1, "Hierarchical software sustainability evaluation structure.

5.2 Scenarios comparison

The final conditions of the model's elements when simulated according to the three previously discussed maintenance resource allocation scenarios are summarized in

Table 13. The structure of the table resembles the hierarchical evaluation structure previously defined in Section 4.1 ("Hierarchical software sustainability evaluation structure"), regarding technical and economic sustainability.

The table's cells that are highlighted in bold against a gray background indicate the best values obtained at the end of the simulation for each of the model's elements.

Table 13. Final conditions of the model's elements for the three simulated scenarios (120 months)

Sustainability dimensions	**Model's elements**	**Final conditions**		
		Scenario #1	**Scenario #2**	**Scenario #3**
Technical	Production library size (FP)	1305.08	**1607.51**	1512.76
	Functional suitability (Dmnl)	0.6925	**0.804692**	0.769665
	Number of preventive violations (#)	2494.13	**288.022**	1270.44
	Density of preventive violations (Dmnl)	1.72581	**0.179173**	0.839819
	Number of corrective violations (#)	2466.97	1656.11	**218.251**
	Density of corrective violations (Dmnl)	1.70702	1.03023	**0.144274**
Economic	Technical debt's principal (person-month)	67.3415	**7.7766**	34.302
	Total interest amount (person-month)	259.909	**116.248**	142.672
	Total technical debt (person-month)	327.251	**124.025**	176.974
	Perceived tangible asset (person-month)	457.071	**335.554**	349.051
	Current tangible asset (person-month)	197.162	**219.306**	206.379
	Relative debt to asset (Dmnl)	1.65981	**0.565534**	0.85752
	Maintenance productivity (FP-person/month)	1.00492	**6.67333**	4.25206
	Opportunity costs (person-month)	**41.4081**	180.964	149.037
	Equilibrium point (month)	12	14	**23**

Source: Author

Analyzing the values from

Table 13, it can be noticed that, in general, Scenario #2 performed better than the other two simulated scenarios in the technical and economic sustainability dimensions. There were only two variables in which it was outperformed: opportunity costs, where Scenario #1 was better as fewer resources were allocated to other activities and more to the perfective maintenance; and equilibrium point, where Scenario #3 was better due to the repayment of the technical debt's principal in Scenario #2.

Long lived systems benefit from higher perfective maintenance productivity over time, but it takes some time for the investments made in preventive and corrective maintenance to pay off (these investments are represented by the opportunity costs graph). The preventive maintenance strategy displayed a higher perfective maintenance productivity

over time, indicating that the maintenance of software under these circumstances would be cheaper in the long run.

Conversely, when reducing the simulation's time horizon from 120 to 60 months, the resulting final conditions of the model's elements presented some changes, which are shown in Table 14. If the software system terminates its operational lifetime by the end of the fifth year, Scenario #1 (perfective maintenance focus) would have delivered the largest production library size, and thus the largest tangible asset.

However, the number of preventive and corrective violations remains higher at the end of the simulation in Scenario #1 when compared to Scenario #2 and Scenario #3, thereby also making Scenario #1 to have the highest violations' density.

Table 14. Final conditions of the model's elements for the three simulated scenarios (60 months)

Sustainability dimensions	**Model's elements**	**Final conditions**		
		Scenario #1	**Scenario #2**	**Scenario #3**
Technical	Production library size (FP)	**1278.99**	1250.71	1224.8
	Functional suitability (Dmnl)	**0.813745**	0.806356	0.79079
	Number of preventive violations (#)	1790.15	**221.665**	944.877
	Density of preventive violations (Dmnl)	1.39966	**0.177231**	0.771454
	Number of corrective violations (#)	1760.24	1163.29	**165.525**
	Density of corrective violations (Dmnl)	1.37627	0.930098	**0.135144**
Economic	Technical debt's principal (person-month)	48.3339	**5.98495**	25.5117
	Total interest amount (person-month)	111.072	**50.7728**	61.8997
	Total technical debt (person-month)	159.406	**56.7578**	87.4114
	Perceived tangible asset (person-month)	**285.559**	221.402	228.994
	Current tangible asset (person-month)	**174.487**	170.63	167.094
	Relative debt to asset (Dmnl)	0.913569	**0.332637**	0.523127
	Maintenance productivity (FP-person/month)	2.20026	**6.68045**	4.50262

	Opportunity costs (person-month)	**26.9907**	80.0371	69.3412
	Equilibrium point (month)	14	14	**23**

Source: Author

Software with a shorter lifetime tends to benefit from focusing on short term results (i.e., focusing on perfective maintenance activities to deliver functional requirements to its end users). The danger arises when, after incurring technical debts, the software remains in operation after the moment in time where it becomes more expensive to maintain.

Further, the behavior over time of the key variables from the three simulated scenarios are shown in Figure 57. The growth pattern of the current tangible assets shows that the perfective maintenance focus (Scenario #1) performed better until a certain point in time near month 60, ratifying what was previously discussed in Table 14 and indicating that this strategy (represented by Scenario #1) could be better suited for short-lived systems, such as end users, web, and commercial applications, as was shown in

Table 1.

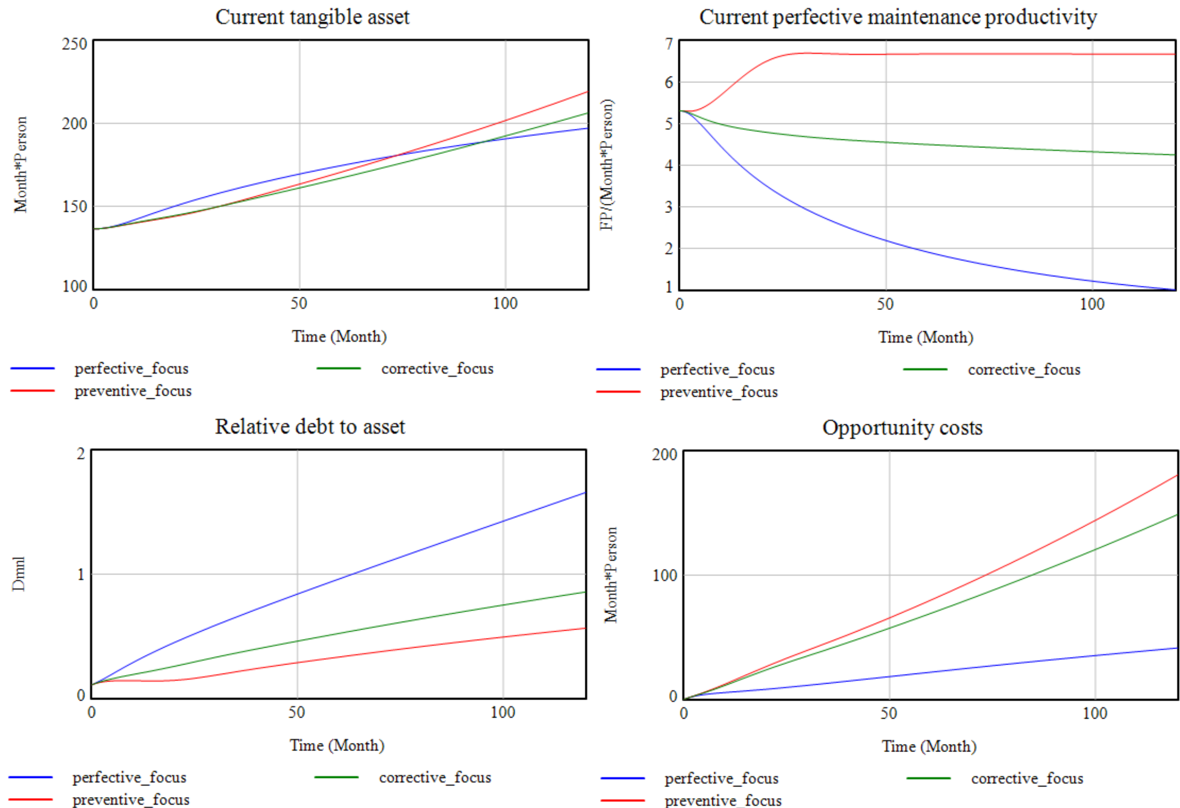


Figure 57. Comparisons of the results of the simulated scenarios
Source: Author

5.3 Chapter summary

This chapter has depicted the results obtained from the simulation of the proposed model according to four predefined scenarios. The contributions of the chapter are summarized as follows:

- *Perfective maintenance focus*: this scenario depicted a context in which the stakeholders' decisions regarding resource allocation were focused on delivering functional requirements to the end users and thus increasing the size of the production library. The end condition of the simulated scenario showed a completely different situation, since it contained the lowest number of functional requirements available in the production library among the three scenarios.
- *Preventive maintenance focus*: next, a scenario representing the focus on preventive maintenance was analyzed. Counter intuitively, this scenario, in general, performed better than the perfective maintenance scenario. In the long

run, the software's production library was bigger than it was with the strategy that focused on delivering functional requirements.

- *Corrective maintenance focus*: the final simulated scenario presented the maintenance strategy that focused on reducing the corrective violations. This scenario performed better than the perfective focus scenario, but worse than the preventive focus scenario regarding the two sustainability dimensions. This strategy could be better suited when the maintenance context involves a software with a high density of corrective violations.

All three simulated maintenance resource allocation scenarios were compared according to the previously defined technical and economic sustainability hierarchical evaluation structure.

6. Conclusions

The following sections discuss how the results presented in the previous chapter ("Results") address the research questions (RQs) formulated in Section 3.1 ("Research questions"), along with the conclusions obtained from the current thesis, the contributions obtained from this research, and proposals for areas of future research.

6.1 Addressing the proposed research questions

(RQ1) How should the dynamical behavior of a software product's quality attributes, due to maintenance activities, be characterized throughout its evolution?

In order to address the first proposed research question, this thesis followed the system dynamics iterative model development approach presented in Section 3.3 ("Research process"), with results application are presented in Section 4.2 ("Proposed simulation model").

Characterization of the software quality attributes' dynamical behavior was achieved via the following process:

- Problem articulation: The key variables and concepts responsible for the problem involving software maintenance and technical debt management were identified and summarized in Table 8 ("Boundary chart of the proposed model") and in Section 4.1 ("Hierarchical software sustainability evaluation structure"). In addition, a dynamical definition of the research problem, based on past and future patterns of behavior of the key variable and concepts, was elaborated via a set of reference modes and equations described in Section 4.2.1 ("Problem articulation and dynamical hypothesis").
- Dynamical hypothesis: A theory containing a set of causal relationships among the variables and concepts (discussed in Section 4.2.1.3, "Model's causal loop diagrams") was developed, which accounted for the problematic behavior previously captured as time series plots (reference modes). The developed theory

was built based on existing published literature as detailed in Chapter 2 ("Background").

- Model formulation: Based on the proposed theory, a fully specified simulation model was developed following the system dynamics' stock and flow notation, with the results from this discussed in Section 4.2.2 ("Model formulation"). The developed simulation model was then subjected to a set of tests, where it was able to reproduce the proposed reference modes (Section 5.1, "Model evaluation"). More importantly, the model was used to try to refute the developed dynamical hypothesis by employing the set of tests listed in Table 9 ("Summary of the tests performed on the model"). As discussed in Section 3.3.1.4, since it was not possible to refute the current proposed theory via a systematic process, the focus of the evaluation shifted to assessment of model's usefulness and shortcomings (Section 5.1, "Model evaluation").

By following the systematic process above, the first research question was addressed and the influence of maintenance activities on the dynamical behavior of software's quality attributes was characterized.

(RQ2) How do different resource allocation policies in software maintenance activities affect the dynamical behaviors of these quality attributes?

In order to address the second research question, three different scenarios were simulated; their results were discussed in Chapter 5 ("Results and discussion"). These scenarios represented three different resource allocation policies on maintenance activities, which were named: Scenario #1 – perfective maintenance focus; Scenario #2 – preventive maintenance focus; and Scenario #3 – corrective maintenance focus.

These three simulated scenarios produced different behaviors for the model's elements during the simulated time horizon. The results obtained from the simulations indicate that, based on decisions made during the software's lifetime regarding the allocation of resources among various maintenance activities (perfective, preventive, and corrective), different conditions arise when analyzing the technical and economic sustainability dimensions.

(RQ3) How should the resource allocation in maintenance activities be managed to improve the technical and economic sustainability of a software product?

The results obtained showed that for long-lived software systems (in this research represented by simulating the proposed model for a time horizon of 120 months), policies that mainly focus on delivering functional requirements (i.e., perfective maintenance focus) yield the smallest production library and the highest number of violations (both preventive and corrective).

Likewise, Scenario #1 accumulated the highest technical debt over time. Thus, it became the most expensive scenario to maintain and to continue operating due to its low productivity at the end of the simulation when compared to the other two scenarios (preventive and corrective maintenance focuses).

Scenario #2, although not focusing on delivering functional requirements, had the most extensive production library at the end of the time horizon, thereby representing the most valuable tangible asset to the organization. Moreover, the preventive focus maintained the lowest level of preventive violations (i.e., lowest technical debt and interest rate), keeping the highest perfective maintenance productivity. The policy in this scenario was to invest in preventive maintenance; thus, even though it incurred the highest opportunity costs, these costs were justified in the long term as the scenario also delivered the lowest maintenance cost compared to the other two scenarios.

However, when reducing the time horizon of the simulation from 120 to 60 months, the gains obtained by the preventive maintenance focus did not overcome the opportunity costs. Hence, the perfective maintenance focus represented by Scenario #1 ended up delivering the highest tangible asset to the organization. If the organization decided to shut down the software operation it had been operating for the past five years, the investment in preventive maintenance would not have worth it.

These results obtained from simulating different scenarios show that the analysis used to addressed RQ3 is context-dependent, relying on several inputs and variables to decide how to allocate the resources to maintenance activities. These variables include initial conditions, the organization's plans for its software assets (short- or long-lived software systems), and

the organizations' objectives related to the software's technical and economic sustainability. The resource allocation decisions must be continuously adapted to take into consideration the current software's characteristics, the maintenance process performance, and the organization's goals.

6.2 Contributions

This thesis investigated the effects of resource allocation policies on maintenance activities on the evolutionary path of a software system, in terms of its technical and economic sustainability, and also how technical debt management through the software's lifetime can determine its capacity to adapt and evolve.

The proposed dynamical evaluation framework demonstrates how a software's source code metrics, obtained from static analysis tools, can be used in a simulation model to support decision making regarding software maintenance based on the organization's goal for technical and economic sustainability.

The proposed framework contains a hierarchical structure for evaluating how the software product and the maintenance process performance evolved over time. This evaluation uses data extracted from both the software source code analysis (when defining initial conditions for use in the simulation model) and the simulation model output (when evaluating possible future scenarios on how to prioritize maintenance activities before committing any resources in the real world).

Furthermore, the formulation of the proposed model extends previous research that has used simulation techniques to explore the phenomenon of software evolution. The proposed model incorporates the dynamic interactions captured by most of the Lehman's (1996b) eight formulated laws, describing based on causal relations, and on an endogenous point of view, how problematic behaviors emerge over time. The proposed simulation model was then used to evaluate how different resource allocation policies could unintentionally produce different outcomes.

This work also supports the idea that system dynamics is a suitable approach for modeling and simulating problems related to the software maintenance process, phenomena

pertaining to the software evolution, and technical debt management practices. The thesis also contributes to the scarce literature on using system dynamics in these areas, as few previous publications have been identified on the topic, and those found have mainly focused on software development and the early stages of a software's lifetime. This work shows that the system dynamics approach is capable of incorporating both technical and economic dimensions involved in decision making regarding software maintenance investment policies.

Finally, the research methods used, which were based on the system dynamics approach, made it possible to leverage current knowledge available in the field of software maintenance and technical debt management by formulating graphical representations of several of the causal relationships that exist during software operation and maintenance, and the impact therefore on software quality characteristics and costs. This knowledge represents a step toward better for explicating, describing, and justifying how long-term dynamical results emerge as a result of past decisions made. The findings can be used as a starting point for future research interested in investigating software evolution.

6.3 Areas of future research

Alongside the contributions made by this research there are several areas related to software maintenance and technical debt management that require further investigation. These include various interesting and relevant problems and challenges that represent opportunities for extending the present research. An outline of future directions is provided below.

The current model takes into account only a subset of the SQuaRE software quality model characteristics; this could be extended to include other quality characteristics and attributes (e.g., performance efficiency, compatibility, usability, and more complex functional suitability constructs). The model also did not take into consideration the type of adaptive maintenance activity that, would inevitably have to be prioritized among the three considered (i.e., perfective, preventive, and corrective).

Further tests should also be carried out to increase the confidence in the proposed model; these tests could provide additional empirical data to verify whether the model is

capable of reproducing other real-world contexts. In addition, future research could assess whether the model's conceptualization and formulation excluded other important elements.

Different resource allocation policies should also be evaluated, including more complex decision-making strategies aimed at to improving long-term results. The software maintenance process concept used in the model's formulation was a simplification of the classical waterfall model. The model could be reviewed in order to simulate and evaluate other lifecycle process approaches (e.g., iterative and incremental, spiral, agile).

The selected constructs and metrics for composing the economic sustainability hierarchical evaluation structure could also be extended in order to consider broader and deeper investment analysis to better support decision making; for example, future work could include analyses focusing on areas such as return on investment, valuation analyses that include not only tangible but also intangible assets and liabilities, or software cost estimation models (e.g., COCOMO, COCOMO II, SLIM, PRICE).

The overall structure of the dynamical evaluation framework shown in Figure 13 can be subject to automation regarding data extraction from SCM repositories using static analysis tools, and the extracted data could be automatically imported into the simulation model, enabling it to be used as a decision support system that could be easily integrated into existing static analysis tools (as discussed in Section 2.6). This would turn the simulation model into a unified and integrated application that could be used to support decisions related to resource allocation in software maintenance, and also to technical debt management.

References

Abdel-Hamid, T. (1984). *The Dynamics of Software Development Project Management - An Integrative System Dynamics Perspective*. Massachusetts Institute of Technology.

Abdel-Hamid, T. (1990). Investigating the cost/schedule trade-off in software development. *IEEE Software*, *7*(1), 97–105. https://doi.org/10.1109/52.43055

Abdel-Hamid, T., & Madnick, S. (1982). A model of software project management dynamics. *The 6th International Computer Software and Applications Conference (COMPSAC)*.

Abdel-Hamid, T., & Madnick, S. (1989). Software productivity: Potential, actual, and perceived. *System Dynamics Review*, *5*(2), 93–113. https://doi.org/10.1002/sdr.4260050202

Abdel-Hamid, T., & Madnick, S. (1991). *Software Project Dynamics: An Integrated Approach* (First edit). Prentice Hall.

Ahsan, K., & Gunawan, I. (2010). Analysis of cost and schedule performance of international development projects. *International Journal of Project Management*, *28*(1), 68–78. https://doi.org/10.1016/j.ijproman.2009.03.005

Ali, N. Bin, Petersen, K., & Wohlin, C. (2014). A systematic literature review on the industrial use of software process simulation. *Journal of Systems and Software*, *97*, 65–85. https://doi.org/10.1016/j.jss.2014.06.059

Allman, E. (2012). Managing Technical Debt. *Queue*, *10*(3), 10. https://doi.org/10.1145/2168796.2168798

Ampatzoglou, A., Ampatzoglou, A., Avgeriou, P., & Chatzigeorgiou, A. (2016). A Financial Approach for Managing Interest in Technical Debt. In B. Shishkov (Ed.), *Business Modeling and Software Design. BMSD 2015. Lecture Notes in Business Information Processing* (pp. 117–133). Springer. https://doi.org/10.1007/978-3-319-40512-4_7

Bakota, T., Hegedus, P., Kortvelyesi, P., Ferenc, R., & Gyimothy, T. (2011). A probabilistic software quality model. *2011 27th IEEE International Conference on Software*

Maintenance (ICSM), 243–252. https://doi.org/10.1109/ICSM.2011.6080791

Bakota, T., Hegedus, P., Ladanyi, G., Kortvelyesi, P., Ferenc, R., & Gyimothy, T. (2012). A cost model based on software maintainability. *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 316–325. https://doi.org/10.1109/ICSM.2012.6405288

Barlas, Y. (1996). Formal aspects of model validity and validation in system dynamics. *System Dynamics Review*, *12*(3), 183–210. https://doi.org/10.1002/(SICI)1099-1727(199623)12:3<183::AID-SDR103>3.0.CO;2-4

Basili, V. R., Caldiera, G., & Rombach, H. D. (2002). Goal Question Metric (GQM) Approach. In *Encyclopedia of Software Engineering*. John Wiley & Sons, Inc. https://doi.org/10.1002/0471028959.sof142

Basili, V. R., & Weiss, D. M. (1984). A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering*, *SE-10*(6), 728–738. https://doi.org/10.1109/TSE.1984.5010301

Becker, C., Chitchyan, R., Duboc, L., Easterbrook, S., Penzenstadler, B., Seyff, N., & Venters, C. C. (2015). Sustainability Design and Software: The Karlskrona Manifesto. *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 467–476. https://doi.org/10.1109/ICSE.2015.179

Belady, L., & Lehman, M. (1971). Programming System Dynamics or the Metadynamics of Systems in Maintenance and Growth. *Research Report RC3546, IBM*.

Belady, L., & Lehman, M. (1972). AN INTRODUCTION TO GROWTH DYNAMICS. In *Statistical Computer Performance Evaluation* (pp. 503–511). Elsevier. https://doi.org/10.1016/B978-0-12-266950-7.50030-X

Belady, L., & Lehman, M. (1976). A model of large program development. *IBM System Journal*, *15*(3), 225–252.

Bennett, K. H., & Rajlich, V. T. (2000). Software maintenance and evolution: A roadmap. *Proceedings of the Conference on The Future of Software Engineering - ICSE '00*, 73–87. https://doi.org/10.1145/336512.336534

Bharadwaj, A., El Sawy, O. A., Pavlou, P. A., & Venkatraman, N. (2013). Digital business strategy: toward a next generation of insights. *MIS Quarterly*, *37*(2), 471–482.

Boehm, B. W., Brown, J. R., Kaspar, H., Lipow, M., Macleod, G. J., & Merrit, M. J. (1978). *Characteristics of Software Quality* (1st editio). Elsevier Science Ltd.

Bosch-Rekveldt, M., Jongkind, Y., Mooi, H., Bakker, H., & Verbraeck, A. (2011). Grasping project complexity in large engineering projects: The TOE (Technical, Organizational and Environmental) framework. *International Journal of Project Management*, *29*(6), 728–739. https://doi.org/10.1016/j.ijproman.2010.07.008

Cao, L., Ramesh, B., & Abdel-Hamid, T. (2010). Modeling dynamics in agile software development. *ACM Transactions on Management Information Systems*, *1*(1), 1–26. https://doi.org/10.1145/1877725.1877730

Carr, N. G. (2003). IT Doesn't Matter. *Harvard Business Review*, *81*(5).

Carvalho, M. M., & Rabechini Junior, R. (2015). Impact of risk management on project performance : the importance of soft skills. *International Journal of Production Research*, *53*(2), 321–340. https://doi.org/10.1080/00207543.2014.919423

Carvalho, M. M., & Rabechini Junior, R. (2017). Can project sustainability management impact project success? An empirical study applying a contingent approach. *International Journal of Project Management*, *35*(6), 1120–1132. https://doi.org/10.1016/j.ijproman.2017.02.018

Cavano, J. P., & McCall, J. A. (1978). A framework for the measurement of software quality. *Proceedings of the Software Quality Assurance Workshop on Functional and Performance Issues* -, 133–139. https://doi.org/10.1145/800283.811113

Cecez-Kecmanovic, D., Kautz, K., & Abrahall, R. (2014). Reframing Success and Failure of Information Systems: A Performative Perspective. *MIS Quarterly*, *38*(2), 561–588.

Chatters, B. W., Lehman, M., Ramil, J. F., & Wernick, P. (2000). *Modelling a software evolution process: a long-term case study.* https://doi.org/10.1002/1099-1670(200006/09)5:2/3<91::AID-SPIP123>3.0.CO;2-L

Chen, H., Chiang, R. H. L., & Storey, V. C. (2012). Business intelligence and analytics: from

big data to big impact. *MIS Quarterly*, *36*(4), 1165–1188.

Cook, S., Harrison, R., Lehman, M. M., & Wernick, P. (2006). Evolution in software systems: foundations of the SPE classification scheme. *Journal of Software Maintenance and Evolution: Research and Practice*, *18*(1), 1–35. https://doi.org/10.1002/smr.314

Cunningham, W. (1993). The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4*(2), 29–30. https://doi.org/10.1145/157710.157715

Curtis, B., Sappidi, J., & Szynkarski, A. (2012). Estimating the Principal of an Application's Technical Debt. *IEEE Software*, *29*(6), 34–42. https://doi.org/10.1109/MS.2012.156

Deißenböck, F. (2009). *Continuous Quality Control of Long-Lived Software Systems*. Technical University Munich.

DeLone, W. H., & McLean, E. R. (1992). Information Systems Success: The Quest for the Dependent Variable. *Information Systems Research*, *3*(1), 60–95. https://doi.org/10.1287/isre.3.1.60

DeLone, W. H., & McLean, E. R. (2003). The DeLone and McLean Model of Information Systems Success : A Ten-Year Update. *Journal of Management Information Systems*, *19*(4), 9–30.

Drnevich, P. L., & Croson, D. C. (2013). Information technology and business-level strategy: toward an integrated theoretical perspective. *MIS Quarterly*, *37*(2), 482–509.

Dwivedi, Y. K., Wastell, D., Laumer, S., Henriksen, H. Z., Myers, M. D., Bunker, D., Elbanna, A., Ravishankar, M. N., & Srivastava, S. C. (2014). Research on information systems failures and successes: Status update and future directions. *Information Systems Frontiers*, *17*(1), 143–157. https://doi.org/10.1007/s10796-014-9500-y

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., & Gorton, I. (2015). Measure it? Manage it? Ignore it? software practitioners and technical debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, 50–60. https://doi.org/10.1145/2786805.2786848

Eveleens, J. L., & Verhoef, C. (2010). The rise and fall of the Chaos report figures. *IEEE*

Software, *27*(1), 30–36. https://doi.org/10.1109/MS.2009.154

Ferenc, R., Hegedűs, P., & Gyimóthy, T. (2014). Software Product Quality Models. In *Evolving Software Systems* (pp. 65–100). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-45398-4_3

Forrester, J. (1961). *Industrial Dynamics* (1st ed.). The MIT Press.

Forrester, J. (1969). *Urban Dynamics* (1st ed.). The MIT Press.

Forrester, J. (1971). *World Dynamics* (1st ed.). The MIT Press.

Forrester, J., & Senge, P. (1980). Tests for building confidence in system dynamics models. *TIMS Studies in the Management Sciences*, *14*, 209–228.

Fowler, M., Beck, K., Brant, J., Opdyke, W., & Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code* (1st editio). Addison-Wesley Professional.

Franco, E. F., Hirama, K., & Carvalho, M. M. (2017). Applying system dynamics approach in software and information system projects: A mapping study. *Information and Software Technology*. https://doi.org/10.1016/j.infsof.2017.08.013

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

Georgantzas, N. C., & Katsamakas, E. G. (2008). Information systems research with system dynamics. *System Dynamics Review*, *24*(3), 247–264. https://doi.org/10.1002/sdr.420

Geraldi, J. G., Maylor, H., & Williams, T. (2010). *Now, let's make it really complex (complicated): A systematic review of the complexities of projects*. https://doi.org/10.1108/01443571111165848

Glass, R. (2001). Frequently forgotten fundamental facts about software engineering. *IEEE Software*, *18*(3), 112–111. https://doi.org/10.1109/MS.2001.922739

Glass, R. (2005). IT Failure Rates - 70% or 10-15%? *IEEE Software*, *22*(3), 112, 110–111. https://doi.org/10.1109/MS.2005.66

Glass, R. (2006). The Standish report: does it really describe a software crisis? *Communications of the ACM*, *49*(8), 15. https://doi.org/10.1145/1145287.1145301

Godfrey, M. W., & German, D. M. (2014). On the evolution of Lehman's Laws. *Journal of Software: Evolution and Process*, *26*(7), 613–619. https://doi.org/10.1002/smr.1636

Godfrey, M. W., & German, D. M. (2008). The past, present, and future of software evolution. *2008 Frontiers of Software Maintenance*, 129–138. https://doi.org/10.1109/FOSM.2008.4659256

Halstead, M. H. (1977). *Elements of Software Science*. Elsevier Science Inc.

Heitlager, I., Kuipers, T., & Visser, J. (2007). A Practical Model for Measuring Maintainability. *6th International Conference on the Quality of Information and Communications Technology (QUATIC 2007)*, 30–39. https://doi.org/10.1109/QUATIC.2007.8

Herraiz, I., Rodriguez, D., Robles, G., & Gonzalez-Barahona, J. M. (2013). The evolution of the laws of software evolution: A discussion based ona systematic literature review. *ACM Computing Surveys*, *46*(2), 1–28. https://doi.org/10.1145/2543581.2543595

Hong, K.-K., & Kim, Y.-G. (2002). The critical success factors for ERP implementation: an organizational fit perspective. *Information & Management*, *40*(1), 25–40. https://doi.org/10.1016/S0378-7206(01)00134-3

INCOSE. (2015). *Systems Engineering Handbook: A Guide for System Life Cycle Processes and Activities* (D. D. Walden, G. J. Roedler, K. J. Forsberg, R. D. Hamelin, & T. M. Shortell (eds.); 4th ed.). Wiley.

ISO/IEC 14764:2006. (2016). *Software Engineering — Software Life Cycle Processes — Maintenance* (p. 44). International Organization for Standardization.

ISO/IEC 25010:2011. (2011). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — System and software quality models* (p. 34). International Organization for Standardization.

ISO/IEC 25012:2008. (2008). *Software engineering — Software product Quality Requirements and Evaluation (SQuaRE) — Data quality model* (p. 13). International Organization for Standardization.

ISO/IEC 25022:2016. (2016). *Systems and software engineering — Systems and software*

quality requirements and evaluation (SQuaRE) — Measurement of quality in use (p. 41). International Organization for Standardization.

ISO/IEC 25023:2016. (2016). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of system and software product quality* (p. 45). International Organization for Standardization.

ISO/IEC 25024:2015. (2015). *Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) — Measurement of data quality* (p. 45). International Organization for Standardization.

ISO/IEC 9126:1991. (1991). *Software enginnering — Product quality* (p. 13). International Organization for Standardization.

Johnson, S. B. (2013). Technical and institutional factors in the emergence of project management. *International Journal of Project Management*, *31*(5), 670–681. https://doi.org/10.1016/j.ijproman.2013.01.006

Jones, C. (2008). *Applied Software Measurement: Global Analysis of Productivity and Quality* (3rd editio). McGraw-Hill Education.

Jørgensen, M., & Moløkken-Østvold, K. (2006). How large are software cost overruns? A review of the 1994 CHAOS report. *Information and Software Technology*, *48*(4), 297–301. https://doi.org/10.1016/j.infsof.2005.07.002

Kahen, G., Lehman, M. M. M., Ramil, J. F. F., & Wernick, P. (2001). System dynamics modelling of software evolution processes for policy investigation: Approach and example. *Journal of Systems and Software*, *59*(3), 271–281. https://doi.org/10.1016/S0164-1212(01)00068-1

Kellner, M. I., Madachy, R. J., & Raffo, D. M. (1999). Software process simulation modeling: Why? What? How? *Journal of Systems and Software*, *46*(2–3), 91–105. https://doi.org/10.1016/S0164-1212(99)00003-5

Kemerer, C. F., & Slaughter, S. (1999). An empirical approach to studying software evolution. *IEEE Transactions on Software Engineering*, *25*(4), 493–509. https://doi.org/10.1109/32.799945

Kruchten, P., Nord, R. L., & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, *29*(6), 18–21. https://doi.org/10.1109/MS.2012.167

Lehman, M. (1980). Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, *68*(9), 1060–1076. https://doi.org/10.1109/PROC.1980.11805

Lehman, M. (1989). Uncertainty in computer application and its control through the engineering of software. *Journal of Software Maintenance: Research and Practice*, *1*(1), 3–27. https://doi.org/10.1002/smr.4360010103

Lehman, M. (1991). Software engineering, the software process and their support. *Software Engineering Journal*, *6*(5), 243. https://doi.org/10.1049/sej.1991.0028

Lehman, M. (1996a). Feedback in the software evolution process. *Information and Software Technology*, *38*(11), 681–686. https://doi.org/10.1016/0950-5849(96)01121-4

Lehman, M. (1996b). Laws of software evolution revisited. *EWSPT '96 Proceedings of the 5th European Workshop on Software Process Technology*, *1149*, 108–124.

Lehman, M., Kahen, G., & Ramil, J. (2002). Behavioural modelling of long-lived evolution processes? Some issues and an example. *Journal of Software Maintenance and Evolution: Research and Practice*, *14*(5), 335–351. https://doi.org/10.1002/smr.259

Lehman, M., Perry, D., & Ramil, J. (1998). On evidence supporting the FEAST hypothesis and the laws of software evolution. *Proceedings Fifth International Software Metrics Symposium Metrics*, 84–88. https://doi.org/10.1109/METRIC.1998.731229

Lehman, M., & Ramil, J. (2001). Rules and Tools for Software Evolution Planning and Management. *Annals of Software Engineering*, *11*(1), 15–44. https://doi.org/10.1023/A:1012535017876

Lehman, M., & Ramil, J. (2002). Software Uncertainty. In D. Bustard, W. Liu, & R. Sterritt (Eds.), *Soft-Ware 2002: Computing in an Imperfect World* (Lecture No, pp. 174–190). Springer. https://doi.org/10.1007/3-540-46019-5_14

Lehman, M., & Ramil, J. (2003). Software evolution—Background, theory, practice. *Information Processing Letters*, *88*(1–2), 33–44. https://doi.org/10.1016/S0020-0190(03)00382-X

Lehman, M., & Ramil, J. C. (2006). Software Evolution. In *Software Evolution and Feedback* (pp. 7–40). John Wiley & Sons, Ltd. https://doi.org/10.1002/0470871822.ch1

Lehman, M., & Ramil, J. F. (1999). The impact of feedback in the global software process. *Journal of Systems and Software*, *46*(2–3), 123–134. https://doi.org/10.1016/S0164-1212(99)00006-0

Letouzey, J.-L., & Coq, T. (2010). The SQALE Analysis Model: An Analysis Model Compliant with the Representation Condition for Assessing the Quality of Software Source Code. *2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, 43–48. https://doi.org/10.1109/VALID.2010.31

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193–220. https://doi.org/10.1016/j.jss.2014.12.027

Lin, C. Y., Abdel-Hamid, T., & Sherif, J. S. (1997). Software-Engineering Process Simulation model (SEPS). *Journal of Systems and Software*, *38*(3), 263–277. https://doi.org/10.1016/S0164-1212(96)00156-2

Madachy, R. J. (2008). *Software Process Dynamics* (1 edition). Wiley-Blackwell.

Mahajan, V., Muller, E., & Wind, Y. (2000). New-Product Diffusion Models: From Theory to Practice. In V. Mahajan, E. Muller, & Y. Wind (Eds.), *New-Product Diffusion Models* (1st ed., p. 355). Springer US.

Markowitz, H. (1952). PORTFOLIO SELECTION. *The Journal of Finance*, *7*(1), 77–91. https://doi.org/10.1111/j.1540-6261.1952.tb01525.x

Markus, M. L., Tanis, C., & van Fenema, P. C. (2000). Enterprise resource planning: multisite ERP implementations. In *Communications of the ACM* (Vol. 43, Issue 4, pp. 42–46). https://doi.org/10.1145/332051.332068

McCabe, T. J. (1976). A Complexity Measure. *IEEE Transactions on Software Engineering*, *SE-2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

McConnell, S. (1996). *Rapid Development: Taming Wild Software Schedules*. Microsoft Press.

McFarlan, F. (1981). Portfolio Approach to Information Systems. *Harvard Business Review*, *59*(5), 142–150.

McKinsey & Company. (2011). *A rising role for IT: McKinsey Global Survey results*. http://www.mckinsey.com/insights/business_technology/a_rising_role_for_it_mckinsey_global_survey_results

Melville, N., Kraemer, K., & Gurbaxani, V. (2004). Information technology and organizational performance: An integrative model of IT business value. *MIS Quarterly*, *28*(2), 283–322.

Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., & Jazayeri, M. (2005). Challenges in Software Evolution. *Eighth International Workshop on Principles of Software Evolution (IWPSE'05)*, 13–22. https://doi.org/10.1109/IWPSE.2005.7

Mitchell, M. (2011). *Complexity: A Guided Tour* (1st editio). Oxford University Press.

Mordal, K., Anquetil, N., Laval, J., Serebrenik, A., Vasilescu, B., & Ducasse, S. (2013). Software quality metrics aggregation in industry. *Journal of Software: Evolution and Process*, *25*(10), 1117–1135. https://doi.org/10.1002/smr.1558

Morecroft, J. D. (1982). A critical review of diagramming tools for conceptualizing feedback system models. *Dynamica*, *8*(1), 20–29.

Müller, R., & Turner, R. (2007). The Influence of Project Managers on Project Success Criteria and Project Success by Type of Project. *European Management Journal*, *25*(4), 298–309. https://doi.org/10.1016/j.emj.2007.06.003

Nan, N. (2011). Capturing Bottom-Up Information Technology Use Processes: A Complex Adaptive Systems Model. *MIS Quarterly*, *35*(2), 505–532.

Nugroho, A., Visser, J., & Kuipers, T. (2011). An empirical model of technical debt and interest. *Proceeding of the 2nd Working on Managing Technical Debt - MTD '11*, 1. https://doi.org/10.1145/1985362.1985364

Oman, P., & Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. *Proceedings Conference on Software Maintenance 1992*, 337–344. https://doi.org/10.1109/ICSM.1992.242525

OMG. (2016a). *Automated Source Code Maintainability Measure (ASCMM)* (Object Management Group (ed.)). http://www.omg.org/spec/ASCMM/1.0/

OMG. (2016b). *Automated Source Code Performance Efficiency Measure (ASCPEM)*. http://www.omg.org/spec/ASCPEM/1.0/

OMG. (2016c). *Automated Source Code Reliability Measure (ASCRM)*. http://www.omg.org/spec/ASCRM/1.0/

OMG. (2016d). *Automated Source Code Security Measure (ASCSM)*. http://www.omg.org/spec/ASCSM/1.0/

OMG. (2018). *Automated Technical Debt Measure (ATDM)*. https://www.omg.org/spec/ATDM/

Orlikowski, W. J., & Robey, D. (1991). Information technology and the structuring of organizations. *Information Systems Research*, *2*(2), 143–169. https://doi.org/10.1287/isre.2.2.143

Parnas, D. L. (1994). Software aging. *Proceedings of 16th International Conference on Software Engineering*, 279–287. https://doi.org/10.1109/ICSE.1994.296790

Petter, S., DeLone, W., & McLean, E. R. (2013). Information Systems Success: The Quest for the Independent Variables. *Journal of Management Information Systems*, *29*(4), 7–62. https://doi.org/10.2753/MIS0742-1222290401

Ramasubbu, N., & Kemerer, C. (2014). Managing Technical Debt in Enterprise Software Packages. *IEEE Transactions on Software Engineering*, *5589*(c), 1–1. https://doi.org/10.1109/TSE.2014.2327027

Repenning, N. P., & Sterman, J. D. (2001). Nobody Ever Gets Credit for Fixing Problems That Never Happened: Creating and Sustaining Process Improvement. *California Management Review*, *43*(4), 64–88. https://doi.org/10.2307/41166101

Rodrigues, A. G. (2000). *The application of system dynamics to project management: an integrated methodology*. University of Strathclyde.

Rogers, E. (2003). *Diffusion of Innovations* (5th ed.). Free Press.

Royal Academy of Engineering. (2004). *The Challenges of Complex IT Projects: The Report*

of a Working Group from the Royal Academy of Engineering and the British Computer Society. The Royal Academy of Engineering.

Ruiz, M., Ramos, I., & Toro, M. (2004). An integrated framework for simulation-based software process improvement. *Software Process: Improvement and Practice*, *9*(2), 81–93. https://doi.org/10.1002/spip.198

Sauer, C. (1993). *Why Information Systems Fail: A Case Study Approach*. Alfred Waller.

Seddon, P. B., Graeser, V., & Willcocks, L. P. (2002). Measuring organizational IS effectiveness. *ACM SIGMIS Database*, *33*(2), 11. https://doi.org/10.1145/513264.513270

Senge, P. M. (2006). *The Fifth Discipline: The Art & Practice of The Learning Organization* (2nd ed.). Crown Business.

Shenhar, A. J., Dvir, D., Levy, O., & Maltz, A. C. (2001). Project success: A multidimensional strategic concept. *Long Range Planning*, *34*(6), 699–725. https://doi.org/10.1016/S0024-6301(01)00097-8

Sommerville, I., Cliff, D., Calinescu, R., Keen, J., Kelly, T., Kwiatkowska, M., Mcdermid, J., & Paige, R. (2012). Large-scale complex IT systems. *Communications of the ACM*, *55*(7), 71. https://doi.org/10.1145/2209249.2209268

Standish Group International. (2013). *CHAOS MANIFESTO 2013: Think Big, Act Small*. 52.

Sterman, J. (2000). *Business Dynamics: Systems Thinking and Modeling for a Complex World*. McGraw-Hill/Irwin.

Stoyenko, A. (1995). Engineering complex computer systems: a challenge for computer types everywhere. I. Let’s agree on what these systems are. *Computer*, *28*(9), 85–86. https://doi.org/10.1109/2.410170

Taylor, T., & Ford, D. (2006). Tipping point failure and robustness in single development projects. *System Dynamics Review*, *22*(1), 51–71. https://doi.org/10.1002/sdr.330

Taylor, T., & Ford, D. (2008). Managing Tipping Point Dynamics in Complex Construction Projects. *Journal of Construction Engineering and Management*, *134*(6), 421–431. https://doi.org/10.1061/(ASCE)0733-9364(2008)134:6(421)

Turski, W. M. (2002). The reference model for smooth growth of software systems revisited. *IEEE Transactions on Software Engineering*, *28*(8), 814–815. https://doi.org/10.1109/TSE.2002.1027802

Ventana System. (2018). *Vensim Professional for Windows* (7.3.4 Single Precision (x32)). Ventana Systems Inc. https://vensim.com/

Venters, C., Jay, C., Lau, L., Griffiths, M., Holmes, V., Ward, R., Austin, J., Dibsdale, C., & Xu, J. (2014). Software Sustainability: The Modern Tower of Babel. *CEUR Workshop Proceedings. RE4SuSy: Third International Workshop on Requirements Engineering for Sustainable Systems*, 7–12. http://eprints.whiterose.ac.uk/84941/

Waeselynck, H., & Pfahl, D. (1994). System Dynamics Applied to the Modeling of Software Projects. *Software-Concepts and Tools*, *15*(4), 162–176. http://www.mendeley.com/research/system-dynamics-applied-modeling-software-projects/

Wagner, S., Lochmann, K., Heinemann, L., Kläs, M., Trendowicz, A., Plösch, R., Seidl, A., Goeb, A., & Streit, J. (2012). The quamoco product quality modelling and assessment approach. *ICSE '12 Proceedings of the 34th International Conference on Software Engineering*, 1133–1142.

Weaver, W. (1948). Science and Complexity. *American Scientist*, *36*, 536.

Wernick, P., & Hall, T. (2003). *Simulating Global Software Evolution Processes by Combining Simple Models: An Initial Study*. https://doi.org/10.1002/spip.159

Wernick, P., & Lehman, M. . (1999). Software process white box modelling for FEAST/1. *Journal of Systems and Software*, *46*(2–3), 193–201. https://doi.org/10.1016/S0164-1212(99)00012-6

Whitney, K. M., & Daniels, C. B. (2013). The Root Cause of Failure in Complex IT Projects: Complexity Itself. *Procedia Computer Science*, *20*, 325–330. https://doi.org/10.1016/j.procs.2013.09.280

Wiederhold, G. (2006). What is your software worth? *Communications of the ACM*, *49*(9), 65–75. https://doi.org/10.1145/1151030.1151031

Woodside, C. M. (1979). A mathematical model for the evolution of software. *Journal of Systems and Software*, *1*, 337–345. https://doi.org/10.1016/0164-1212(79)90035-9

Yeo, K. T. (2002). Critical failure factors in information system projects. *International Journal of Project Management*, *20*(3), 241–246. https://doi.org/10.1016/S0263-7863(01)00075-8

Zhang, H., Kitchenham, B., & Pfahl, D. (2008). Reflections on 10 Years of Software Process Simulation Modeling: A Systematic Review. In Q. Wang, D. Pfahl, & D. M. Raffo (Eds.), *Making Globally Distributed Software Development a Success Story* (Vol. 5007, pp. 345–356). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-79588-9

Zhang, H., Kitchenham, B., & Pfahl, D. (2010). Software Process Simulation Modeling: An Extended Systematic Review. *New Modeling Concepts for Today's Software Processes*, *6195*, 309–320. https://doi.org/10.1007/978-3-642-14347-2

Zhang, H., Raffo, D., Birkhöltzer, T., Houston, D., Madachy, R., Münch, J., & Sutton, S. M. (2014). Software process simulation at a crossroads? *Journal of Software: Evolution and Process*, *26*(10), 923–928. https://doi.org/10.1002/smr.1694

Appendix A – System dynamics tools and elements

This appendix complements the brief description presented in Section “3.3.1” and provides some examples of the elements used by the system dynamics approach.

A.1 Elements and notations

Table 15 presents the elements and notations used for the formalization of models based on the system dynamics approach. These notations refer to the Vensim Professional tool (Ventana System, 2018) used in this work for the construction and simulation of the model.

Table 15. System dynamics model's elements

Element	**Notation**	**Description**
Level/Stock	Stock	A stock represents an accumulation over time, also called “level” or “state variable”. It can serve as a storage for material, energy, or information. Its content moves through the stocks by incoming or outcoming flows (or “rates”). Stocks represent the state variables (i.e., the system’s memory) and are a function of previous accumulations of rates. Some examples are: • Software tasks (measured in function points, lines of code, use cases, modules, components, etc.) • Number of defects • Team size • Effort spent
Source/Drain		Sources and drains indicate that flows originate or terminate somewhere outside the process. Their presence means that accumulation in the real world occurs outside the boundaries of the modeled system. They represent infinite sources or repositories that are not specified in the model. Examples include: • Sources of requirements;

		• Software delivered to the consumer; • Sources for hiring or dismissing employees.
Rate/Flow	rate	Flows are also called "rates", and they represent the "actions" in the system. They make changes in stocks and can represent decisions or policy statements. Flows are computed as a function of the stocks, constants, and auxiliary variables. Examples include: • Development productivity rate • Defect generation rate • Team members hiring rat • Learning rate
Auxiliary variable	<<Variable name>>	Auxiliary variables are input-to-output converters and help to incorporate details of the stocks and flow structure explicitly. An auxiliary variable must be associated with an information link, which connects stocks and flows. They often represent goal maintenance variables (i.e., they are goal-seeking). Examples include: • Percentage of completion of a job • Quantitative goals or planned values • Constants, such as average delay times • Defect density
Information link		Information links are used to represent information flows (as opposed to material flow). Flows, such as control mechanisms, often require connectors of other variables (stocks or auxiliaries) for decision making. Links can represent feedback loops between the elements. Examples include: • Progress information for decision making • Knowledge of defect levels to allocate resources for rework • A link between process parameters, flows, stocks, and other variables

Source: Adapter from Madachy (2008)

A.2 Mathematical formulation

System dynamics modeling tools allow, in large part, systems to be described in a visual way, without having to formulate and calculate differential equations. The tools themselves make numerical calculations of integrations, and it is up to the modeler to define the rate equations and auxiliary variables.

The mathematical structure of a simulation model constructed according to the system dynamics approach corresponds to a coupled nonlinear set of first order differential equations. This structure can be described by a vector of levels (**x**), a set of parameters (**p**) and a nonlinear vector function (**f**), according to the representation:

$$x'(t) = f(x, p)$$

The use of numerical integrations occurs because, in general, it is not possible to solve even small models analytically, since they correspond to high-order and non-linear systems. The order of a dynamic system, or a mesh, is defined by the number of state variables, or levels, that it has. A first-order system has only one level. Linear systems are systems in which the rate equations correspond to linear combinations of state variables and other exogenous inputs (Sterman, 2000).

The rate of change of a level is the sum of all its entries subtracted from all its outputs. The stocks, in turn, accrue flows. Mathematically, the levels integrate their rates and the rates are the one-level derivatives.

In the simulations performed in the modeling tools, as the simulation time progresses, all model rates are calculated and integrated to define the levels using numerical integration methods (e.g., Runge-Kutta, Euler).

To determine the level of a given stock at a given time *t* based on input and output rates, the following numerical integration is used:

$$Stock(t) = Stock_0 + \int_{t_0}^{t} [input(t) - output(t)] \cdot dt$$

The parameter dt corresponds to the time increment defined for execution and can be solved by the equation (assuming $Stock_0$ equals zero):

$$Stock(t) = Stock(t - dt) + [input(t) - output(t)] \cdot dt$$

Equivalently, the rate of change of a given level, its derivative, corresponds to the difference between the input and output flows, defining the differential equation:

$$\frac{dStock(t)}{dt} = input(t) - output(t)$$

Table 16 summarizes, with examples, the symbolic representation of causality between elements, their polarity, their interpretation, and the mathematical concepts involved.

Table 16. Polarity of relations and definitions, with examples.

Symbol	Interpretation	Mathematics	Examples
X —(+)→ Y	All else equal, if X increases (decreases), then Y increases (decreases) above (below) what it would have been. In the case of accumulations, X adds to Y.	$\partial Y / \partial X > 0$ In the case of accumulations, $Y = \int_{t_0}^{t} (X+..)ds + Y_{t_0}$	Product quality —(+)→ Sales; Effort —(+)→ Results; Births —(+)→ Population
X —(-)→ Y	All else equal, if X increase (decreases), then Y decreases (increases) below (above) what tf would have been. In the case of accumulations, X subtracts from Y.	$\partial Y / \partial X < 0$ In the case of accumulations, $Y = \int_{t_0}^{t} (-X+..)ds + Y_{t_0}$	Product price —(-)→ Sales; Frustration —(-)→ Results; Deaths —(-)→ Population

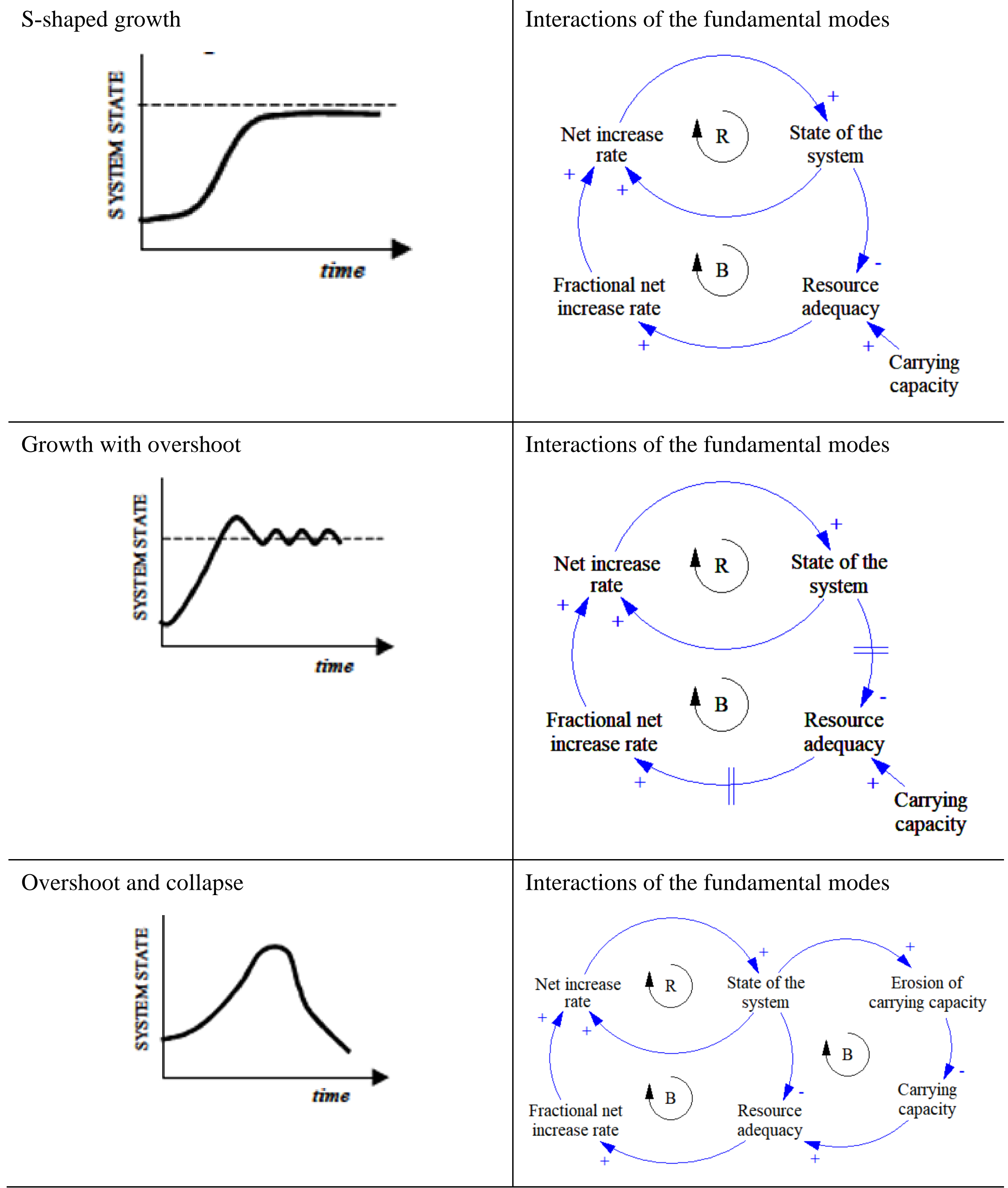
Source: Adapted from Sterman (2000)

A.3 Common behaviors and their corresponding feedback structures

Table 17 shows examples of common modes of behavior (as time series data) and the corresponding feedback structure responsible for producing them.

Table 17. Common modes of behavior and their feedback structures

Common mode	**Feedback structure**
Exponential growth	Positive feedback
Goal-seeking	Negative feedback
Oscillation	Negative feedbacks with time delays



Source: Adapted from Sterman (2000)

A.4 Basic patterns and equations

The following sections present some examples of basic patterns and equations used when formulating a system dynamics model.

A.4.1 Constant flow and one stock

One of the most straightforward structures used in modeling consists of the equation that defines a given rate by multiplying a constant and a level.

Figure 58 shows a simple software production structure, where the *development rate* is defined by the multiplication of the constant *productivity* and the *team* level.

Figure 58. Example of software development framework

Source: Adapted from Madachy (2008)

For the example shown in Figure 58, the *productivity* constant was set at 4.65 function points per person per month, and the *team* variable was set to 5 people. With this configuration, the level *Tasks developed* presents a linear growth that can be described by the equation:

$$Taks\ developed(t+1) = Task\ developed(t) + development\ rate$$

And the development rate can be defined as:

$$development\ rate = productivity\ \cdot team$$

A.4.2 Variable flow and one stock

This structure is similar to the one presented in the previous section, but it replaces the constant (in the case of *productivity*) with an auxiliary variable. This structure can model

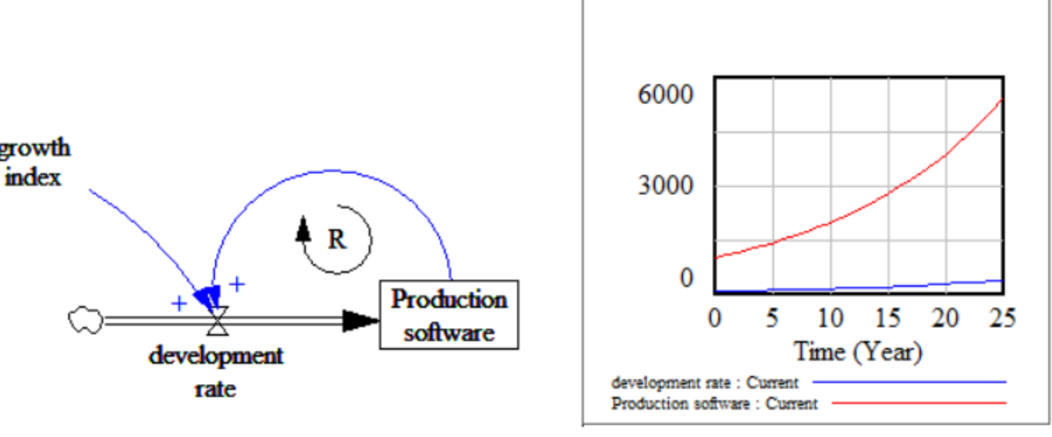
situations closer to reality when, for example, productivity is not constant over time due to the team's learning curve, training, etc.

A.4.3 Reinforcing loop

The reinforcement (or positive) loop amplifies the dynamic pattern of a system and can produce growth or decline behavior. By changing the structure presented in the Section A.4.1 ("Constant flow and one stock") by adding the feedback of the state of the stock as an input to the flow rate, the behavior presented by the stock ceases to be linear and becomes exponential growth.

The structure presented in Figure 59 represents the phenomenon described by the sixth law of software evolution, which concerns "*Continuous growth*". The *Production software* level refers to the number of functionalities available in the software product in operation; its unit of measure is function points and it has the initial value of 1,000. The rate of addition of new functionalities in operation is defined by the multiplication of the *growth index* (7% per year) and the status of the *Production software* level.

Figure 59. Example of a reinforcement loop for continuous software growth



Source: Author

The equation of the flow development *rate* of this structure is defined by the equation:

$$development\ rate(t) = Production\ software(t) \cdot growthindex$$

The graph on the right of Figure 59 shows the exponential growth of the functionalities of the software product in operation over the years and the reinforcement mesh is represented by *R*. The behavior of the *Production software* stock is described by the following equation:

$$Production\ software(t+1) = Production\ software(t) + \ development\ rate(t)$$

A.4.4 Balancing loop

This type of structure tries to bring the system to a state close to the desired goal. It exhibits goal-seeking behavior, where the change is faster at the beginning and decreases as the discrepancy between the perceived state of the system and the desired state decreases.

Figure 60 presents an example of this type of structure. Starting from a team size (*Project team* stock), initialized with 10 people, the model seeks to reach the *desired team size* of 20 people.

The hiring of people does not occur instantly, and it takes time to complete the processes of recruitment, selection and effective hiring of new employees. In this example, this period was established as 4 months (auxiliary variable *hiring delay*).

Figure 60. Example of balancing mesh for hiring people

Source: Author

The current state of the system (*Project team* stock) is negatively feedbacked in the *hiring rate*, which is defined by the equation:

$$hiring\ rate(t) = \frac{(desired\ team\ size - Project\ team(t))}{hiring\ delay}$$

The graph on the right of Figure 60 shows the behavior over time of the *Project team* stock, which represents the current state of the system, and the balancing mesh is represented by *B*. The behavior of the *Project team* stock is described by the following equation:

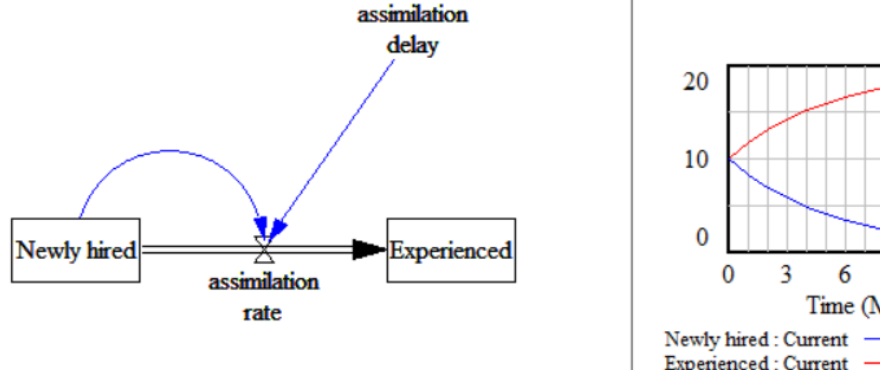
$$Project\ team(t+1) = Project\ team(t) + hiring\ rate(t)$$

A.4.5 Delay

Delays are present in most processes, and they may involve delays in materials and information. They constitute an essential structural component of feedback systems.

Figure 61 shows a delay structure regarding the time of assimilation of new employees hired by a company. These employees, represented by the level *Newly hired*, need to acquire the productivity of the oldest employees, the *Experienced* stock (e.g., they need to undergo training). The mean time of assimilation is represented by the auxiliary variable *assimilation delay*, which in this example was defined as 6 months.

Figure 61. Structure of delay in the assimilation of new employees



Source: Author

The two stocks in the structure (*Newly hired* and *Experienced*) have the initial condition of 10 employees each. Figure 61 corresponds to a first-order delay, and the equations of the stocks and flows involved can be defined by:

$$assimilation\ rate(t+1) = \frac{Newly\ hired(t)}{assimilation\ delay}$$

$$Experienced(t+1) = Experienced(t) + assimilation\ rate(t)$$

$$Newly\ hired(t+1) = Newly\ hired(t) - assimilation\ rate(t)$$

These equations generate the behavior observed in the graph on the right of Figure 61, where the two stocks start with 10 employees, with employees flowing from *Newly hired* to *Experienced* over time.

A.4.6 Table function

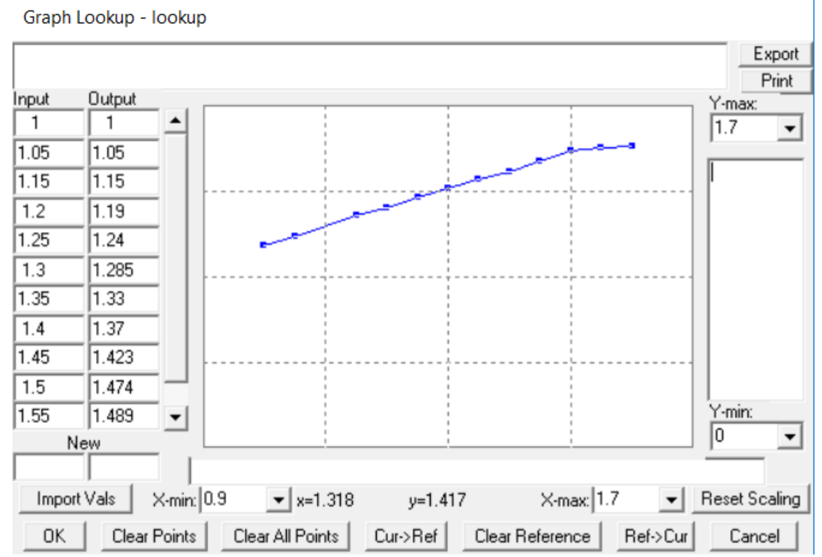
A table function represents a convenient way of specifying relationships between variables, and a plot is an easy way to visualize the relationships. These tools are useful for formulating nonlinear relationship effects in a model.

The construction of a table function involves defining the slope, its general configuration, a set of reference lines and some guidelines. It is recommended to normalize the data as a function of the input ratio by the reference value.

Figure 62 shows an example of a table function that represents the effect of work productivity on working hours. This effect is observed when the work team is asked to extend their regular workday to, for example, reduce or eliminate the schedule delays.

The plot shows the relation between the multiplier coefficient of over-day workout (input) and the independent variable referring to the ratio between the desired productivity and the nominal (output). This means that when a worker begins to extend his working time, it may gain productivity. However, if he extends for long periods, the effect becomes less efficient.

Figure 62. Example of table function related to working hours vs. productivity



Source: Adapted from Madachy (2008)

The limits of the graph are established when the desired productivity is equal to nominal, and the coefficient multiplier has no effect (i.e., is equal to 1). Assuming that people can at most extend the working week from 40 to 60 hours, the right limit is established where the multiplier coefficient saturates to 1.5, even when the pressure to extend the workday increases (desired productivity). It is important to note that if the working period is extended even further, the productivity of the team would certainly drop due to exhaustion and fatigue.

A.5 Example

The model presented in Figure 63 represents a simplified view of the software development process, where a set of requirements (represented by the *Requirements backlog* stock) is transformed into developed functionalities (represented by the *Developed requirements* stock) according to the *development rate*, which is defined by the *team size* and its *productivity*.

Figure 63. Example of the software development process

Source: Author

Appendix B – Model documentation

The following sections contain the model's documentation extracted from the Vensim software (Ventana System, 2018) for each of the four of the model subsystem discussed in Section 4.2.1.1 ("Model's subsystem diagram").

B.1 Perfective maintenance subsystem

```
Current perfective maintenance productivity=
            Nominal perfective maintenance productivity*Effect of
maintainability in perfective maintenance productivity
      (Violation density
            [preventive]/Reference perfective violation density)
      Units: FP/Month/Person

Developed but not checked= INTEG (
      Perfective maintenance rate-Rework rate-Release rate,
            0)
Units: FP

Effect of maintainability in perfective maintenance productivity(
      [(0,0)-(1,1)],(0,1),(1,0))
Units: Dmnl

Fractional of flawed functional requirement=
      Fractional requirement creep(Production library/Reference backlog
library)
Units: Dmnl

Fractional requirement creep(
      [(0,0)-
(100000,1)],(0,0.063),(100,0.063),(1000,0.168),(10000,0.336),(100000
,0.504))
Units: Dmnl
Estimates from Jones (2008), page 293.
```

```
Functional requirements backlog= INTEG (
      Rework rate+New requirements rate-Perfective maintenance rate,
            200)
Units: FP

Functional suitability=
      Production library/(Developed but not checked+Functional
requirements backlog
+Production library)
Units: Dmnl

New requirements rate=
      Production library*Nominal growth rate*Software overall
attractiveness
Units: FP/Month

Nominal growth rate=
      0.07/12
Units: 1/Month

Nominal perfective maintenance productivity=
      4.65
Units: FP/Month/Person

"Perceived functional suitability adj. time"=
      12
Units: Month

Perfective maintenance rate=
      MIN(Current perfective maintenance productivity*Resources allocated
to perfective maintenance
,Functional requirements backlog
      /TIME STEP
      )
Units: FP/Month

Production library= INTEG (
```

```
      Release rate,
            1000)
Units: FP
The size of the software product in operation.

Reference backlog library=
      1
Units: FP

Reference perfective violation density=
      1
Units: Violation/FP

Release rate=
      Developed but not checked/Time to check*(1-Fractional of flawed
functional requirement
)
Units: FP/Month

Resources allocated to perfective maintenance=
      Actual perfective maintenance fraction*Maintenance team
Units: Person

Rework rate=
      Developed but not checked/Time to check*Fractional of flawed
functional requirement
Units: FP/Month

Software overall attractiveness=
      DELAY1( Functional suitability , "Perceived functional suitability
adj. time"
 )
Units: Dmnl

TIME STEP  = 0.0625
Units: Month

Time to check=
```

```
      3
Units: Month
Defines how long it takes to a developed functional requirements
            to goes through the quality assurance process.


Violation density[violation]=
      Total violations[violation]/Production library
Units: Violation/FP
```

B.2 Corrective & preventive maintenance subsystem

```
Current violation potential[violation]=
            Violation potential according to software size(Production
library/Reference backlog library
      )
      Units: Violation/FP

Current violation removal efficiency[violation]=
      Violation removal efficiency according to software
size[violation](Production library
/Reference backlog library)
Units: Dmnl

Current violation removal productivity[violation]=
      Nominal violation removal productivity[violation]*Effect of
maintainability on violation removal productivity
[violation]
Units: Violation/(Month*Person)

Discovered violations[violation]= INTEG (
      Early defect detection rate[violation]-Violation removal
rate[violation]+Late defect detection rate
[violation],
            0)
Units: Violation

Early defect detection rate[violation]=
      (Undiscovered violations[violation]*Current violation removal
efficiency[violation
])/TIME STEP
Units: Violation/Month

Effect of maintainability on violation removal productivity[corrective]=
      1
Effect of maintainability on violation removal productivity[preventive]=
            1
      Units: Dmnl

Initial violations[preventive]=
      0
```

```
Initial violations[corrective]=
            0
      Units: Violation

Late defect detection rate[violation]=
      Released violations[violation]/Time to discover operational
violations
Units: Violation/Month

Nominal violation removal productivity[preventive]=
      12
Nominal violation removal productivity[corrective]=
            20
      Units: Violation/(Month*Person)

Perfective maintenance rate=
      MIN(Current perfective maintenance productivity*Resources allocated
to perfective maintenance
,Functional requirements backlog
      /TIME STEP
      )
Units: FP/Month

Production library= INTEG (
      Release rate,
            1000)
Units: FP
The size of the software product in operation.

Reference backlog library=
      1
Units: FP

Released violations[violation]= INTEG (
      Violation detection rate[violation]-Late defect detection
rate[violation],
            0)
Units: Violation

Resources allocated to corrective maintenance=
      Actual corrective maintenance fraction*Maintenance team
Units: Person

Resources allocated to preventive maintenance=
      Actual preventive maintenance fraction*Maintenance team
Units: Person

TIME STEP  = 0.0625
Units: Month

Time to discover operational violations=
      12
Units: Month

Undiscovered violations[preventive]= INTEG (
      Violation generation rate[preventive]-Early defect detection
rate[preventive
```

```
]-Violation detection rate[preventive],
            Initial violations[preventive])
Undiscovered violations[corrective]= INTEG (
            Violation generation rate[corrective]-Early defect detection
rate[corrective
      ]-Violation detection rate[corrective],
                  Initial violations[corrective])
      Units: Violation

Violation detection rate[violation]=
      (Undiscovered violations[violation]*(1-Current violation removal
efficiency
[violation]))/TIME STEP
Units: Violation/Month

Violation generation rate[violation]=
      Current violation potential[violation] * Perfective maintenance
rate
Units: Violation/Month

Violation potential according to software size(
      [(0,0)-(100000,10)],(100,4),(1000,5),(10000,6),(100000,7.25))
Units: Violation/FP
According to Jones (2008)

Violation removal efficiency according to software size[violation](
      [(0,0)-
(100000,10)],(100,0.92),(1000,0.85),(10000,0.81),(100000,0.65))
Units: Dmnl [0,1]
Quantity of violations found during test / Total number of bugs

Violation removal rate[preventive]=
      MIN( Discovered violations[preventive]/TIME STEP , Current
violation removal productivity
[preventive] * Resources allocated to preventive maintenance
      )
Violation removal rate[corrective]=
            MIN( Discovered violations[corrective]/TIME STEP , Current
violation removal productivity
      [corrective] * Resources allocated to corrective maintenance
            )
      Units: Violation/Month
```

B.3 Resource allocation sector

```
Actual corrective maintenance fraction= INTEG (
            Change corrective fraction,
                  1/3)
      Units: Dmnl


Actual perfective maintenance fraction= INTEG (
```

```
      Change perfective fraction,
            1/3)
Units: Dmnl

Actual preventive maintenance fraction= INTEG (
      Change preventive fraction,
            1/3)
Units: Dmnl

Average effort to fix violation[preventive]=
      50
Average effort to fix violation[corrective]=
            25
      Units: Violation/Month/Person

Average productive working monthly time=
      0.6
Units: Dmnl

Change corrective fraction=
      (Fraction of corrective resources needed-Actual corrective
maintenance fraction
)/Resource allocation adjustment time
Units: 1/Month

Change perfective fraction=
      (Fraction of perfective resources needed-Actual perfective
maintenance fraction
)/Resource allocation adjustment time
Units: 1/Month

Change preventive fraction=
      (Fraction of preventive resources needed-Actual preventive
maintenance fraction
)/Resource allocation adjustment time
Units: 1/Month

Corrective maintenance attractiveness=
```

```
      Actual fraction of manpower to rework*0+0.1
Units: Dmnl

Corrective maintenance backlog=
      Total violations[corrective]/Average effort to fix
violation[corrective]
Units: Person*Month

Fraction of corrective resources needed=
      Necessary resources for corrective maintenance/Total resources
needed
Units: Dmnl

Fraction of perfective resources needed=
      Necessary resources for perfective maintenance/Total resources
needed
Units: Dmnl

Fraction of preventive resources needed=
      Necessary resources for preventive maintenance/Total resources
needed
Units: Dmnl

Functional requirements backlog= INTEG (
      Rework rate+New requirements rate-Perfective maintenance rate,
            200)
Units: FP

hiring rate=
      0
Units: Person/Month

Maintenance team= INTEG (
      hiring rate,
            5)
Units: Person

Necessary resources for corrective maintenance=
```

(Corrective maintenance backlog/Average productive working monthly time)*Corrective maintenance attractiveness
Units: Person*Month

Necessary resources for perfective maintenance=
(Functional requirements backlog/Perfective maintenance monthly productivity
)*Perfective maintenance attractiveness
Units: Person*Month

Necessary resources for preventive maintenance=
(Preventive maintenance backlog/Average productive working monthly time)*Preventive maintenance attractiveness
Units: Person*Month

Perfective maintenance attractiveness=
0.8
Units: Dmnl

Perfective maintenance monthly productivity=
4.2
Units: FP/Month/Person

Preventive maintenance attractiveness=
Actual fraction of manpower to rework*0+0.1
Units: Dmnl

Preventive maintenance backlog=
Total violations[preventive]/Average effort to fix violation[preventive]
Units: Person*Month

Resource allocation adjustment time=
4
Units: Month
The amount of time needed to change the resources allocation from their current activity to the new one.

```
Resources allocated to corrective maintenance=
      Actual corrective maintenance fraction*Maintenance team
Units: Person

Resources allocated to perfective maintenance=
      Actual perfective maintenance fraction*Maintenance team
Units: Person

Resources allocated to preventive maintenance=
      Actual preventive maintenance fraction*Maintenance team
Units: Person

Total resources needed=
      Necessary resources for preventive maintenance+Necessary resources
for corrective maintenance
+Necessary resources for perfective maintenance
Units: Person*Month

Total violations[violation]=
      Discovered violations[violation]+Released
violations[violation]+Undiscovered violations
[violation]
Units: Violation
```

B.4 Goal evaluation sector

```
"% adjustment in planned fraction of manpower to rework"(
            [(0,-0.5)-(0.5,0)],(0,0),(0.1,-0.025),(0.2,-0.15),(0.3,-
0.35),(0.4,-0.475
      ),(0.5,-0.5))
      Units: Dmnl

Actual fraction of manpower to rework=
      1+"% adjustment in planned fraction of manpower to rework"(Schedule
pressure
)
Units: Dmnl
```

```
Corrective maintenance attractiveness=
      Actual fraction of manpower to rework
Units: Dmnl

Current perfective maintenance productivity=
      Nominal perfective maintenance productivity*Effect of
maintainability in perfective maintenance productivity
(Violation density
      [preventive]/Reference perfective violation density)
Units: FP/Month/Person

Current tangible asset=
      Production library/Nominal perfective maintenance productivity
Units: Person*Month

Discovered violations[violation]= INTEG (
      Early defect detection rate[violation]-Violation removal
rate[violation]+Late defect detection rate
[violation],
            0)
Units: Violation

FINAL TIME  = 120
Units: Month

Functional requirements backlog= INTEG (
      Rework rate+New requirements rate-Perfective maintenance rate,
            200)
Units: FP

Interest amount=
      Perceived tangible asset-Current tangible asset
Units: Month*Person

Investment rate=
      Maintenance team
Units: Person
```

```
Maintenance team= INTEG (
      hiring rate,
            5)
Units: Person

Nominal perfective maintenance productivity=
      4.65
Units: FP/Month/Person

Opportunity costs= INTEG (
      Opportunity costs rate,
            0)
Units: Month*Person

Opportunity costs rate=
      Resources allocated to corrective maintenance+Resources allocated
to preventive maintenance
Units: Person

Perceived tangible asset= INTEG (
      Investment rate,
            Production library/Nominal perfective maintenance
productivity)
Units: Month*Person

Perfective maintenance attractiveness=
      1
Units: Dmnl

Preventive maintenance attractiveness=
      Actual fraction of manpower to rework
Units: Dmnl

Preventive maintenance backlog=
      Total violations[preventive]/Average effort to fix
violation[preventive]
Units: Person*Month
```

```
Production library= INTEG (
      Release rate,
            1000)
Units: FP
The size of the software product in operation.

Relative debt to asset=
      Total technical debt/Current tangible asset
Units: Dmnl

Released violations[violation]= INTEG (
      Violation detection rate[violation]-Late defect detection
rate[violation],
            0)
Units: Violation

Resources allocated to corrective maintenance=
      Actual corrective maintenance fraction*Maintenance team
Units: Person

Resources allocated to perfective maintenance=
      Actual perfective maintenance fraction*Maintenance team
Units: Person

Resources allocated to preventive maintenance=
      Actual preventive maintenance fraction*Maintenance team
Units: Person

Schedule pressure=
      MAX(MIN(ZIDZ((Time perceived still needed-Time available),Time
available),
0.5),0)
Units: Dmnl

Schedule's shortage=
      MAX(Time perceived still needed-Time available,0)
Units: Month
```

```
Technical debt principal=
      Preventive maintenance backlog
Units: Person*Month

Time available= DELAY FIXED (
      FINAL TIME-Time,TIME STEP,0)
Units: Month

Time perceived still needed=
      Functional requirements backlog/(Current perfective maintenance
productivity
*Resources allocated to perfective maintenance)
Units: Month

TIME STEP  = 0.0625
Units: Month

Total technical debt=
      Interest amount+Technical debt principal
Units: Month*Person

Total violations[violation]=
      Discovered violations[violation]+Released
violations[violation]+Undiscovered violations
[violation]
Units: Violation

Undiscovered violations[preventive]= INTEG (
      Violation generation rate[preventive]-Early defect detection
rate[preventive
]-Violation detection rate[preventive],
            Initial violations[preventive])
Undiscovered violations[corrective]= INTEG (
            Violation generation rate[corrective]-Early defect detection
rate[corrective
      ]-Violation detection rate[corrective],
                  Initial violations[corrective])
      Units: Violation
```

```
Violation density[violation]=
      Total violations[violation]/Production library
Units: Violation/FP
```

Appendix C – Secondary data used

Table 18. Software defects per function point by industry segment

	Require. Bugs	Design Bugs	Code Bugs	Document Bugs	Bad Fix Bugs	Total Bugs
MIS	0.75	1.50	1.75	0.50	0.50	5.00
Web	1.68	0.63	1.05	0.21	0.63	4.20
U.S. outsource	0.95	1.19	1.66	0.48	0.48	4.75
Offshore outsource	1.38	1.38	1.38	0.66	0.72	5.50
Commercial	0.60	1.80	1.80	1.20	0.60	6.00
Systems	0.65	1.63	2.60	0.98	0.65	6.50
Military	1.40	1.40	2.45	1.05	0.70	7.00
End-user	–	0.45	1.65	0.30	0.60	3.00
Average	**0.93**	**1.25**	**1.79**	**0.67**	**0.61**	**5.24**

Source: Jones (2008)

Table 19. Software defect origin percent by industry segment

	Require. Bugs	Design Bugs	Code Bugs	Document Bugs	Bad Fix Bugs	Total
MIS	15%	30%	35%	10%	10%	100%
Web	40%	15%	25%	5%	15%	100%
U.S. outsource	20%	25%	35%	10%	10%	100%
Offshore outsource	25%	25%	25%	12%	13%	100%
Commercial	10%	30%	30%	20%	10%	100%
Systems	10%	25%	40%	15%	10%	100%
Military	20%	20%	35%	15%	10%	100%
End-user	0%	15%	55%	10%	20%	100%
Average	**18%**	**23%**	**35%**	**12%**	**12%**	**100%**

Source: Jones (2008)

Table 20. Approximate U.S. productivity ranges by of applications (data expressed in function points per staff month)

Software Types	1990	1995	2000	2005
Web	–	12.00	15.50	23.00
MIS	7.20	7.80	9.90	9.40
Outsourced	8.00	8.40	7.80	9.70
Systems	4.00	4.20	5.80	6.80
Commercial	5.10	5.30	9.20	7.20
Military	2.00	1.80	4.20	3.75
Average	**4.38**	**6.58**	**8.73**	**9.98**

Source: Jones (2008)

Table 21. Approximate productivity rates by size of application (data expressed in terms of function points per staff month)

	1,000 Function Points	10,000 Function Points	100,000 Function Points
SOA	8.00	26.00	20.00
TSP/PSP + Scrum	15.75	10.00	9.00
CMM 5 + Six-Sigma	13.00	9.75	9.25
TSP/PSP	14.25	9.50	8.00
CMM Level 5	12.50	9.25	7.75
Six-Sigma for software	9.00	9.00	7.20
CMM 3 + OO	16.00	8.75	6.80
CMMI	11.25	8.25	7.50
Object oriented (OO)	15.50	8.25	6.75
Agile/Scrum + OO	24.00	7.75	7.00
RUP	14.00	7.75	4.20
CMM Level 4	11.00	7.50	5.50
CMM Level 3	9.50	7.25	4.50
Agile/Scrum	23.00	7.00	5.25
Spiral	15.00	6.75	4.25
Extreme XP	22.00	6.50	5.00
TickIT	8.75	6.25	3.90
Iterative	13.50	6.00	3.75
RAD	14.50	5.75	3.60
Waterfall + inspections	10.00	5.50	4.75
CMM Level 2	7.50	4.00	2.25
Waterfall	8.00	2.50	1.50
CMM Level 1	6.50	1.50	1.25
Average	**13.64**	**7.03**	**5.23**

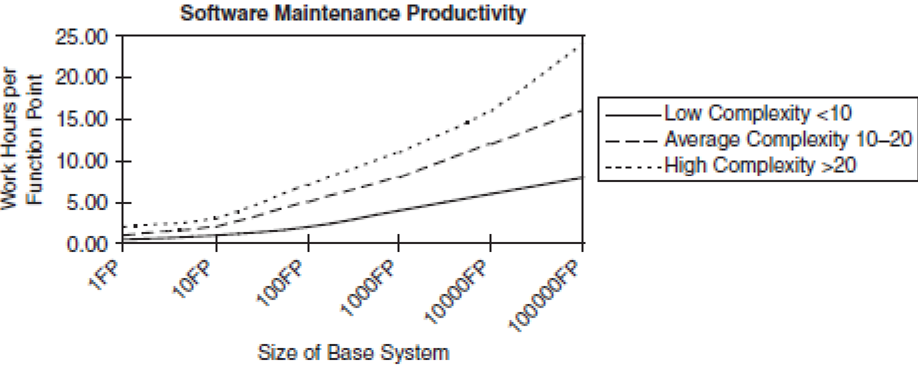
Source: Jones (2008)

Table 22. U.S. average productivity in function points per staff month

Form of Software	100	1,000	10,000	100,000	Average
End-user	52.60	–	–	–	52.60
Web	47.30	25.60	12.00	–	28.30
MIS	21.00	16.40	4.65	2.70	11.19
U.S. outsource	23.00	17.20	4.90	3.20	12.08
Offshore outsource	19.00	15.80	4.70	3.00	10.63
Commercial	11.00	9.30	4.60	4.00	7.23
Systems	9.00	6.90	5.10	4.15	6.29
Military	5.60	4.80	3.80	2.10	4.08
Average	**23.56**	**13.71**	**5.68**	**3.19**	**11.54**

Source: Jones (2008)

Figure 64. Maintenance productivity variance due to size of base system and cyclomatic complexity



Source: Jones (2008)

Table 23. Application probable requirements "creep" (data expressed in percentage of original requirements)

Form of Software	100	1,000	10,000	100,000
End-user	1.75%	0.00%	0.00%	0.00%
Web	3.00%	12.00%	22.00%	0.00%
MIS	6.30%	16.80%	33.60%	50.40%
U.S. outsource	5.70%	13.20%	28.40%	40.80%
Offshore outsource	6.80%	16.00%	35.00%	52.00%
Commercial	8.80%	19.20%	36.80%	52.80%
Systems	6.00%	14.00%	23.50%	39.00%
Military	11.25%	28.50%	48.00%	63.75%
Average	**6.20%**	**14.96%**	**28.41%**	**37.34%**

Source: Jones (2008)

Table 24. Average rate of annual enhancements (data is based on percentage change of application function points)

Form of Software	100	1,000	10,000	100,000	Average
End-user	12.00%	0.00%	0.00%	0.00%	12.00%
Web	15.00%	11.00%	8.00%	0.00%	11.33%
MIS	7.00%	6.50%	7.00%	4.00%	6.13%
U.S. outsource	7.00%	6.50%	6.00%	3.90%	5.85%
Offshore outsource	7.00%	7.00%	6.50%	4.75%	6.31%
Commercial	10.00%	9.00%	7.00%	5.00%	7.75%
Systems	7.00%	6.50%	8.00%	4.00%	6.38%
Military	7.00%	8.00%	9.00%	6.50%	7.63%
Average	**9.00%**	**7.79%**	**7.36%**	**4.69%**	**7.92%**

Source: Jones (2008)

Table 25. Defect repairs time by defect origins

#	Defect origins	Find hours	Repair hours	Total hours
1	Security defects	11.00	24.00	35.00
2	Errors of omission	8.00	24.00	32.00
3	Hardware errors	3.50	28.00	31.50
4	Abeyant defects	5.00	23.00	28.00
5	Data errors	1.00	26.00	27.00
6	Architecture defects	6.00	18.00	24.00
7	Toxic requirements	2.00	20.00	22.00
8	Requirements defects	5.00	16.50	21.50
9	Supply chain defects	6.00	11.00	17.00
10	Design defects	4.50	12.00	16.50
11	Structural defects	2.00	13.00	15.00
12	Performance defects	3.50	10.00	13.50
13	Bad test cases	5.00	7.50	12.50
14	Bad fix defects	3.00	9.00	12.00
15	Poor test coverage	4.50	2.00	6.50
16	Invalid defects	3.00	3.00	6.00
17	Code defects	1.00	4.00	5.00
18	Document defects	1.00	3.00	4.00
19	User errors	0.40	2.00	2.40
20	Duplicate defects	0.25	1.00	1.25
	Average	**3.78**	**12.85**	**16.63**

Source: Jones (2008)

Table 26. U.S. average for delivered defects per function point

Form of Software	100	1,000	10,000	100,000	Average
End-user	1.05	–	–	–	1.05
Web	0.52	0.60	1.01	–	0.71
MIS	0.32	0.75	1.14	2.54	1.19
U.S. outsource	0.19	0.59	0.90	1.76	0.86
Offshore outsource	0.41	0.81	1.13	2.22	1.14
Commercial	0.24	0.40	0.64	0.92	0.55
Systems	0.15	0.24	0.35	0.56	0.33
Military	0.22	0.47	0.62	0.77	0.52
Average	**0.39**	**0.55**	**0.83**	**1.46**	**0.81**

Source: Jones (2008)