TU WIEN Informatics

A Biomimetic View on Complex Software Systems and the Consequences for Digital Sustainability

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Wirtschaftsinformatik

eingereicht von

Vedrana Tomic, BSc
Matrikelnummer 01160907

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.-Prof. Dr. Edgar Weippl

Wien, 11. Oktober 2021

Vedrana Tomic	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

A Biomimetic View on Complex Software Systems and the Consequences for Digital Sustainability

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Business Informatics

by

Vedrana Tomic, BSc
Registration Number 01160907

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.-Prof. Dr. Edgar Weippl

Vienna, 11th October, 2021

Vedrana Tomic	Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Vedrana Tomic, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. Oktober 2021

Vedrana Tomic

Abstract

Nowadays, individual software often plays a part in large complex software systems. A system like this develops new characteristics, potentials, and risks, much like a multicellular organism, displays different properties than the unicellular organisms that it is made of. This phenomenon is called *Emergence* as it describes the emerging behaviour and features of a system due to the interaction of underlying components.

Emerging behaviour and other systemic properties are rarely considered in software systems during the development process. In a world of increasing complexity, this poses a great risk and has already led to significant issues in the past.

We are missing clear guidelines and considerations for the development stage of software in complex systems, but also for previous stages such as requirements engineering and following stages such as testing and maintenance. Developing these criteria for a new programming paradigm requires us to understand the underlying complexity phenomena, as well as the systemic behaviour of complex software systems.

The aim of this work is to research what we can learn from biology and other disciplines, and how their concepts can be operationalised for the software development process.

Interview rounds with an interdisciplinary expert panel shall guide the compilation of a new paradigm (or at least its criteria), which will be evaluated by the following content analysis and a second interview round.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement & Motivation

The first software programs developed by humans might have been complicated or simple, but the vast majority of them were certainly self-contained. Most software at that time did not exhibit a large number of relationships or interfaces with other components. It was capable of working independently without requiring much support from outside.

Nowadays, software engineers are still used to thinking of software as an autonomous unit. We consider the outcomes and ramifications as a result of the software, or developer's purpose and aim. However, modern applications are not limited to their physical or operational areas. Instead, they are highly interconnected with other applications, sources of data, physical components, and can therefore lead to unexpected but consequential effects.

A good example to illustrate this development is the evolution in the automotive industry. When motorized cars first emerged, they were far from the interconnectedness we observe in modern cars. Apart from fuel not much was required to drive the car. They were mostly independent of outside support to function. Occasionally, an engineering issue would arise, but while the underlying technology was considered complicated by the general public, a car could still be deconstructed and repaired by mechanics in a manageable time frame.

Nowadays electronics and software are deeply incorporated in motorized vehicles. This technology enabled great comfort and new features such as built-in navigation, automatic transmission, and even autonomous vehicles, that do not require a human operator and can drive autonomously. These cars cannot be considered self-contained anymore. They are highly connected to the outside world and exhibit plenty of interfaces to interact with sensors and other software. The software's area of application is not limited to the local unit and rather displays global interactions, such as cars with collision avoidance systems.

Through this work, it will become apparent that these dynamics can be observed in nearly all critical IT systems.

Together software of individual units can play a part in large complex software systems. The goals and ramifications of these systems do not follow the rules or properties of the individual software. A system like this develops new characteristics, dynamics, and risks, much like a multicellular organism displays different goals and needs than the unicellular organisms that it is made of. This phenomenon is called *emergence* as it describes the emerging behaviour and properties of a system due to the interaction of underlying components.

Emergence in software systems is rarely considered during the engineering process. Therefore, it poses a great risk in the current world, where complexity is increasing but has also led to significant issues in the past.

We have experienced software failures in critical systems such as the Boeing 737 Max disaster, where the 737's dynamic instability was tried to be fixed with software, that turned out to rely on fault-prone sensors. Output cross-checking between the sensors was lacking and the engineers did not implement the possibility for pilots to fully regain control by pulling the yoke after the software commanded to descend because of a faulty sensor [163][44][62].

The 737 Max disaster is a gruesome example of possible consequences when complex software systems are engineered without focusing on resilience.

Another issue is the growing risk of internet service outages. In June 2019 Cloudflare and its customers experienced a global blackout, while the outage in July 2020 was limited to certain geographies. The incidents lasted for less than an hour during which visitors had no access to Cloudflare sites. Both outages were not caused by external attacks but by unrelated internal updates, and in both cases, the cause could be traced back to a "one-line code" [60] [61]. Analysis of the 2019 incident constituted that a faulty regular expression has led to a cascade of failures. In principle, the IRR (Internet Route Registry) exists to prevent such incidents. This is a database where ISPs (Internet Service Providers) can enter their valid routes. Mere cross-checking would then suffice to identify faulty routes and deal with such errors. However, major ISPs often disregard contents of the database as "uncontrolled garbage" and refer to their customers also providing unreliable entries [110].

This shows that resilience requires not only technical considerations but also organisational and managerial attention. It becomes clear that this problem is not limited to a certain time or geographical area. Systemic issues like these can be found in almost all critical areas of everyone's lives, and they have far-reaching ramifications and require action. From 2011 to 2017 at least 64%, that's more than half, of all software projects in the CHAOS database were not successful, meaning they either had trouble remaining on time, budget, or on target [78]. If nothing is done to shift the focus towards resilience and sustainability in the engineering process of software systems, the probability of issues like empty shelves in supermarkets [119, 22, 161], leaks and abuse of personal

data [68, 146, 80, 30], internet or power outages [110, 151], and many others will keep increasing drastically.

Some academic and industry groups have already recognised this threat.

Alenezi and Zarour conducted an extensive literature review, where the relationship between complexity and security in software is discussed. [8] In another study by Alenezi et. al, a more practical approach to this has been taken. [77] The study aims to build more arguments around the theory that complexity indeed increases the vulnerability of a system. By analysing projects varying in size and popularity they found that the number of files and the percentage of methods to classes account for most of the software's vulnerability.

Kerr argues that the development of new software should be seen as growth rather than movement, as the software will still exist in the place from where it moved. She emphasises the interconnectedness of software systems and compares the engineering team to a forest, that is exchanging information through an invisible network of roots and fungi, and finally embedding this knowledge into the software. Engineering of complex software systems should therefore be viewed as an organic process without a distinct start- or end-point, and focus on knowledge instead of tasks [84]. Papapetrou, Hunt, Thomas, and others have started to advocate for the notion of *software gardening* instead of engineering. The development of software is often compared to constructing a building. It starts with a plan, architects draft a design, which is then implemented, people move in and the project ends. However, this is not an accurate depiction of software development. While there is still a plan and design, the outcome cannot be predicted in such detail as the construction of a building. Nowadays software systems are highly interconnected with their environment and with other software, resembling a garden more than a building. This realisation also plays into the importance of maintenance, since a garden needs to be maintained continuously, regardless if we want to adapt it or keep it the same [170, 72]. Papapetrou has drafted a "Software Gardener Manifesto", which emphasised the perception of software as a "being" rather than just an object, the importance of teamwork and collaboration, and other gardening notions [123]. Winters, Manshreck, and Wright discuss the complex emergent dynamics in today's software systems and the resulting difficulties to understand the system's behaviour. They talk about the dangers of small changes resulting in *tipping points* and endangering the whole system, the necessity of deployment automation and short release cycles to handle increasing complexity, the advantages of a least privilege model for security, and other hands-on experiences during the engineering of complex software systems. The need for understandability to manage a system's complexity and how mental models can help is particularly stressed. Many insights are related to design principles, meaning that these things need to be taken into account from the very start of the development [180]. Humphrey argues that the reason why the majority of large software projects fail is that they are hard to manage and accordingly that wrong management techniques are used. Autocratic management appears to be ineffective for software. Instead, he advocates for the "Team Software Process", which was developed by the Software Engineering Institute

(SME) and focuses on quality and disciplined and motivated teams [70, 71].

1.2 Aim of the Work

Science and therefore also computer science has evolved to deal with *tame problems* while most of the challenges to modern software systems are *wicked problems*. The latter portrays contradicting or changing requirements and is usually impossible to solve. There is no one true answer to a wicked problem but rather good or bad solutions. These are applied to the problem until resources run out or the solution meets other criteria, meaning wicked typically problems don't have a stopping rule. [133]

Current software engineering paradigms lack these considerations. If we continue to develop software systems that solve tame problems, rather than shifting to the requirements of wicked ones, we will not only continue to experience system failures like the 737 Max disaster and the Cloudflare outages, but these incidents might increase in number and the consequences in severity.
Nature is used to dealing with wicked problems, along with emergent behaviour and complexity, which makes biology an excellent research area to gain a better understanding in this regard.

The thesis aims to utilise advancements and knowledge from complex dynamics in biological systems for application to software engineering. We want to investigate biological resilience principles such as evolution and examine how these could be relevant for software engineering and where parallels or analogies can be drawn. This should aid the understanding of complexity in software.

In the first step, an interdisciplinary literature review will be conducted to formulate a common framework of concepts and terminology for the rest of the thesis. General concepts such as complexity, resilience, but also the term of software engineering have multiple levels of meaning. In the context of this work, they should be described. This research will serve as a base for qualitative methods which will be used in the next step. Since this work stresses multidisciplinarity, the intention is to involve the opinion of biologists, complexity scientists, and experts from the industry. Qualitative methods in the form of explorative interviews will be used to gain insight into the perceptions and handling of complexity in these fields, as well as, their limitations in practice.

Together with the previously conducted literature review, the data will be contextualized in two ways:

1. Deduction of criteria or principles for enhanced handling of complex software systems and thereby increasing resilience and sustainability

2. Exemplary discussion of limitations and obstacles, and implications for industrial application

Based on these insights, the aim is to identify which of the examined principles are particularly relevant for sustainability, and how the basic conditions have to change so that they are applicable during in software engineering process.

Finally, this work will leave a qualified structure for further research approaches in the area of digital sustainability and (software) complexity.

The research questions can be formulated as follows:

1. How can the terms software engineering, complexity, and resilience be described in the context of today's software systems?
2. Which biological principles could be utilised for sustainable software engineering and what challenges could arise in a practical application?
3. Which research approaches call for further attention when aiming to increase resilience and sustainability in complex software systems?

1.3 State-of-the-Art

As mentioned in section 1.1 the scientific community has not remained silent and much research has been conducted exploring the relationship of software, complexity, and biology.

Clarke et. al claim that the software development process needs to be adapted according to the development context rather than applying a generalized software process in all contexts. They argue that the relationship between the development process and its situational context is a part of a complex system itself. [38]
O'Reilly applies the principles of Residual Theory to the software design process to improve the handling of complex software systems. The properties of Residual Theory in Software Engineering were elaborated, and the benefits were described. [122]

While this is not a biological principle, it provides an interesting approach to dealing with complexity.
Francesco et. al promote the importance of bio-inspired paradigms for the software engineering of resilient systems. They draw the comparison between biological immune systems to the concept of software resilience. To illustrate this they chose the Akka[1] implementation of the Actor Model. Based on the model they finally propose a bio-inspired architecture for resilient software. [54]
Similarly, the work of Santosh revolves around the idea of methodological reductionism, a concept that is also found in biology, where he describes the attempt of explaining all biological phenomena based on underlying biochemical and molecular processes. [139]

The paper introduces the "Extracellular Software Development Ecosystem" that promotes code use and reuse philosophy. They kept the approach neutral to platforms or

[1] http://akka.io/

methodologies intending to support the engineering process in creating a sustainable environment.
Another biological point of view has been provided by Bandi and Ramsden. The paper promotes the various potentials of biological concepts for programming and provides basic "building blocks" of life. Based on that they formulate a programming paradigm. [15]
All the above-mentioned work provides great in-depth insights, however, a broader context over the laid out methodologies and concepts, and their implications, is missing.

CHAPTER 2

Theoretical Framework

2.1 Definitions and Taxonomy

While this work is not focused on a particular research niche and mostly uses terms and phrases known to all research fields, it is still necessary to clearly define certain terms. The reason being, that definitions might vary depending on their context or have a historical development.

To avoid confusion and miscomprehension, this chapter will summarize the most important terms and give general definitions as well as describe the understanding of these terms in a software engineering context.

2.1.1 Software Engineering

There is a lot of confusion about the meaning of the terms *software engineering*, *software development*, or *computer science*. Sometimes they are used as synonyms, and often their intended meaning is not defined. In this work, mostly the term software engineering will be used, and it is crucial to deliver a clear definition in the context of this work.

In 1968 the NATO SCIENCE COMMITTEE published a report concerned with the following aspects of software engineering [117]:

- relation of software to the hardware of computers
- design of software
- production, or implementation of software
- distribution of software
- service on software

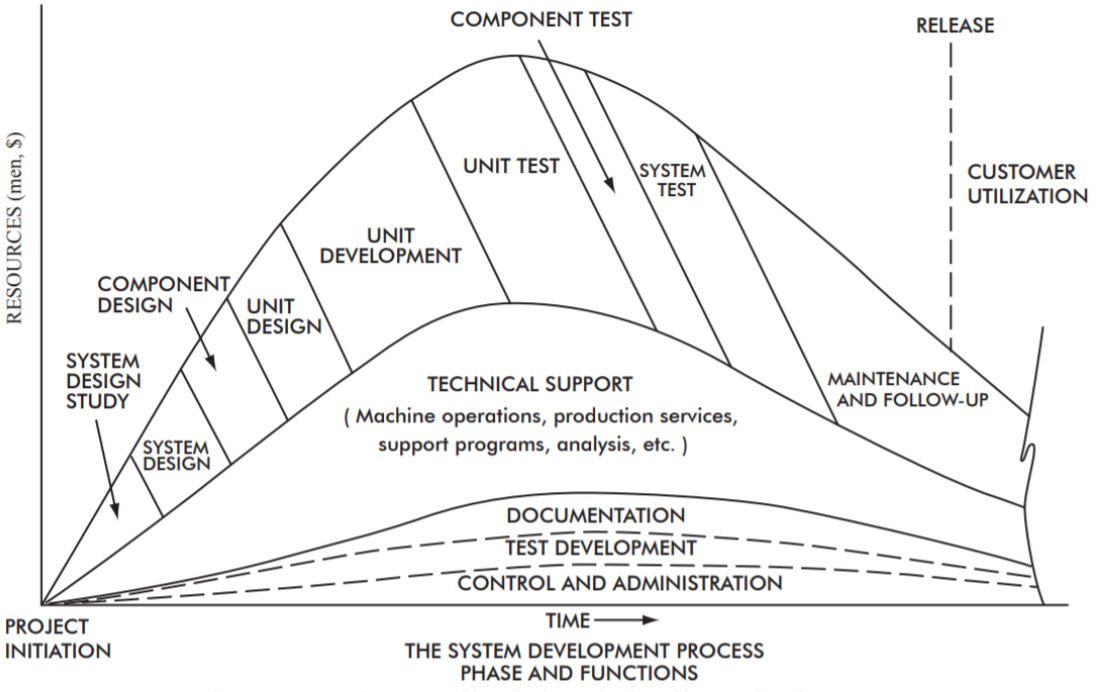

Figure 2.1: Various activities of a software project by Nash [117]

While the first association is that these aspects are only relevant to those working with computers or software, NATO made it clear that the report appeals to a wide circle of readers. It also includes those that do not work directly in the field, however, still require a found understanding of software engineering. These readers could be managers of businesses that use computers (which nowadays includes essentially all enterprises), academic researchers from all fields and other university officials. Additionally, the report also includes passages dealing with the effects of software on society. This makes it a relevant piece for civil servants, politicians, and policymakers of public and private enterprises [117].

A software engineering project starts with an analysis and design phase, where firstly the system is studied and designed, then the components, and lastly the units. Only then the units are implemented followed by testing phases. Even after the release, maintenance and customer utilization still need to be taken care of. This phase generally follows the project as long as the software exists and is used. See figure **??**.

While the software engineering process has evolved (from waterfall to agile), the components from figure **??** can still be found today. When mentioning the term software engineering these technical considerations will come to mind and they are the heart of software projects. Nowadays all these phases are preceded by requirements engineering. In a software context Chemuturi defines a requirement in the following way:

> "*A requirement is a need, expectation, constraint or interface of any stakeholders that must be fulfilled by the proposed software product during its development [35].*"

Requirement engineering is a managerial task that used to stand at the beginning of each project but shifted more into a rolling planning measure since the rise of agile programming. It presumes the possibility to describe a software's requirements precisely. This is an important issue considering that it has been becoming increasingly difficult to describe *how a software system is supposed to behave*. It is already challenging to describe *how a software system is currently behaving*. This doesn't mean that it is not possible to precisely describe the desired function of a small feature. What we don't understand are the emerging side effects that seemingly insignificant developments might trigger. The meaning and challenges of requirements engineering have changed [135].

However, this is not the only non-technical consideration that needs to be made during software engineering. It is crucial to understand that our modern software systems are hyperconnected, dealing with large amounts of personal data, and providing support for individuals, communities, businesses, and governments. The relevant facets for developing such software nowadays go beyond the considerations of managers and developers. Not only do individual humans interact with these systems, but society itself has become a user. It has always shaped technology and in return, technology has shaped society. Given the deep interactions and dependency of humans and computers, this is now more true than ever, which is exactly why the term software engineering goes beyond the scope of development, maintenance, and administration of projects [82].

Definition

Software engineering describes the process of developing software while considering technical, managerial, and societal aspects.

2.1.2 Complexity

Complexity is one of the most essential topics in this research and has been defined many times in various ways often depending on the context that it is being used in.

When the term is mentioned in software engineering, it is often a misnomer as it is associated with *computational complexity*, which deals with the question of how many computational resources an algorithm requires [11]. In this work, we will go beyond this definition.

Firstly, it is necessary to make the distinction between complicated and complex.

Gawande describes the classification of problems in three categories:

- **Simple problems** refer to well-understood and well-known problems that can be easily solved by acquiring some basic skills and following instructions. E.g baking a cake

- **Complicated problems** refer to repeatable tasks that might be broken into a series of simple problems, however, no straightforward instructions exist and usually more advanced skills need to be acquired. E.g sending a rocket to the moon

- **Complex problems** refer to highly dynamic and individual issues that are not repeatable. Once a rocket was built and sent to the moon, the process can be repeated, however, raising a child for instance is a complex issue. Every child is unique and the process cannot be easily repeated.

[57]

Similarly, Taleb provides a comparison between non-complex (mechanical) and complex (organic) systems in 2.1. The notion that complex and organic systems need perturbations to sustain, is discussed further in chapter 4.

THE MECHANICAL, NONCOMPLEX	THE ORGANIC, COMPLEX
Needs continuous repair and maintenance	Self-healing
Hates randomness	Loves randomness (small variations)
No need for recovery	Needs recovery between stressors
No or little interdependence	High degree of interdependence
Stressors cause material fatigue	Absence of stressors cause atrophy
Age with use (wear and tear)	Age with disuse *
Undercompensates from shocks	Overcompensates from shocks
Time brings only senescence	Time brings aging and senescence

* Frano Barovic reading this chapter wrote to me: "Machines: use it and lose it; organisms: use it or lose it."Also note that everything alive needs stressors, but not all machines need to be left alone — a point we will visit in our discussion of annealing.

Table 2.1: THE MECHANICAL AND THE ORGANIC (BIOLOGICAL OR NONBIOLOGICAL) [159]

These descriptions help to understand the differences between non-complex and complex, although concretely describing what constitutes complexity is a difficult task.

Waldrop tries to give a definition relevant to physicists, biologists, computer scientists and economists. He defines complexity as

"*... the name given to the emerging field of research that explores systems in which a great many independent agents are interacting with each other in many ways* [172].",

This interaction between agents can also be found in Johnson's considerations. He argues that complexity could be defined as the "phenomena which emerge from a collection of interacting objects". He refers to crowds of humans for instance and mentions the respective emerging phenomena caused by them such as traffic jams, and market crashes [79].

This general description applies to the understanding of complexity in most areas. Other emergent phenomena include examples from biology such as our immune system, or extreme weather conditions such as hurricanes or droughts. Even in technology and engineering complexity plays a great role, looking at internet outages as an example of emerging phenomena.

Emergence is also mentioned by Riedl who gives a biological angle of view to complexity. He argues that sharpening the definition of complexity would be delusive as complexity is a widespread and diverse state in our world. It is polymorph in nature, so it can only be defined by a list of attributes [131]:

One characteristic is the existence of certain forms of order within complexity. This order is independent of the physical equilibrium. Dissipative systems, which describe systems with dynamic and self-organised structures, can be observed. Complexity always arises in a certain environment, so the system conditions have an impact. However, this influence is not the sole determinant. Riedl states that the inner conditions of a system gain autonomy, and start to follow their own trends, constraints and degrees of freedom [131].

Sheppard shares this view and states for a software engineering context that

"*Complexity is a metaphysical property and thus not directly observable. What is required is a means to link the behaviour of measurable product characteristics.*" [147]

Honglei, et al. sees this link in the cost consumed by development, maintenance, and usage of software, and propose measuring these attributes for assessing software complexity [67].

Similarly, Ramamoorthy defines software complexity in the following way:

"*We define software complexity as the degree of difficulty in analysis, testing, design and implementation of software.*" [129],

Among these various definitions, it is critical to understand which kinds of complexity exist and what their nature is.

The following list displays a possible characterisation in a software engineering context [189]:

- Computational complexity (mentioned for the sake of completeness)

- Structural complexity - deals with the relationships between a system's components
- Logical complexity - deals with logical decisions or flows and branches in a system
- Textual complexity - deals with program source code
- Conceptual complexity - deals with the cognitive perception and relative difficulty to understand and adapt a system

Particularly conceptual complexity has been mentioned by Edmonds, who writes that complexity is

> "*... that property of a language expression which makes it difficult to formulate its overall behaviour, even when given almost complete information about its atomic components and their interrelations.*" [46],

This is an extremely critical insight since we have described this issue in section 2.1.1. It means that the increasing difficulties in requirements engineering can be associated with the increasing complexity of software.
Another indicator for this could be the structural complexity of current software systems. Universally it is defined by the number of objects and interrelationships between them. Luo et. al comply with this definition by further detailing it for the software context, stating that "structural complexity is reflected by the composition and structure of the system. ... Overall, the structural complexity of information systems is a measurable characteristic, which depends on the number of subsystems and their connectivity structure." [96]

One approach to measuring the structural complexity of software systems is to base it on complex networks. These are studied across various fields of science and certain properties could be observed among a majority of networks. These global features describe phenomena such as the *small world* or *scale free* property [175, 16].

Ma et. al propose three metrics to qualitatively measure the structural complexity of a software system [186]:

1. **Structure Entropy**: measures according to Shannon's information entropy the amount of uncertainty in the structural information
2. **Evaluation Coefficient**: is based on the structure entropy
3. **Weight Ratio**: measures how interactions or relationships between components of a software system influence its overall topology

High structural complexity can further increase complexity in a software system. It can have direct effects such as leading to a surge in conceptual complexity.

However, the effects can also be recursive, and increase the structural complexity itself. Many lines of code do not immediately amount to higher complexity, however, this could result in developers hesitating to make adjustments and reusing existing code due to a lack of understanding. This might lead to undesired coding actions such as code duplication, which further increase the structural complexity and decrease the system's understandability.

These are only some of the possible definitions and categorisations of complexity, but it becomes already very clear the various forms are not independent properties but depend heavily on each other. Since this is a very broad topic it is impossible to cover all aspects of complexity in the scope of this project.

Future research should deepen and improve the definitions of complexity in IT and enrich our knowledge and understanding with intelligence from other fields such as biology and complexity research.

Definition

Complexity is a phenomenon that emerges from the interactions of agents in a system and that cannot be found at the individual level of the agents.

Definition

Conceptual Complexity is defined by the psychological perception of how difficult it is to understand, maintain, and adapt a software system.

Definition

Structural Complexity is defined by the amount and relationships of a system's components and the resulting phenomena.

2.1.3 Systems

In the previous section, complexity has been discussed and some definitions were given. Simultaneously the term *system* has been used often, particularly when describing complexity within a system.

This section will briefly outline what a system is and how it can be understood in a software context.

Generally, we can summarize that a system is a set of interacting elements that has a real or fictional boundary between these elements within the system and its surrounding environment.

Donella H. Meadows writes

"*A system is a set of things-people, cells, molecules, or whatever-interconnected in such a way that they produce their own pattern of behaviour over time.*" [101]

This consideration is important when analysing how systems operate. Meadows explains that understanding one component of the system does not allow for deductions of the system's behaviour when there are two components or more. Their relationship and interactions can cause entirely different patterns than we observe in just one element. An outside event can trigger certain system behaviour but how exactly it will play out is mostly decided by the system itself. Applying the same event to another system will likely lead to different patterns [101].

Further properties can be attributed. Systems may be

- open (interacting with the environment) or closed (isolated)
- abstract or concrete
- simple or complicated
- complex or chaotic [89, 130]

Simple and complicated systems refer to closed systems that commonly do not interact with their environment. They may or may not be difficult to understand but they are self-contained.

On the other side, complex systems usually interact with their environment.

Definition

A *Software System* is an encompassing combination of components that communicate with one another on a software basis and that enable parts of a computer system, e.g., software and hardware, to work together.

The cyber-physical, techno-social, and critical systems mentioned in this work generally refer to complex software ecosystems. In the following, we will define them in a bit more detail and with a particular software focus.

Techno-social systems

The term *techno-social* stems from the insight that technology and society co-evolve and shape each other. The technologies that humans develop are shaped by who they are and how they live. But these technologies in return also influence how humans live and define themselves [50].

Since the transition of society into the Information Age, technology has had an enormous impact on human interaction.

With the introduction of social media, decision support systems, and mass surveillance the way humans communicate, control, and coordinate has been greatly enhanced [50]. However, much more significant changes are happening behind the scenes, hidden from sight. For example, while the average human still has to turn on the tap for a glass of water, management, provisioning, and distribution of water might be severely dependant on the internet and digitalization.

These large software systems usually consist of many underlying components interacting with each other and the environment. They follow certain patterns so they are not chaotic but complex.

Definition

Techno-social systems refer to software systems that co-evolve with society.

Cyber-physical systems

Cyber-physical systems are technological systems that combine our traditional physical systems, such as power grids, with software. This digitalisation has not co-evolved with the physical systems but rather developed independently and was then applied to existing physical systems [86].

Again this is an open complex system that will play a relevant role in this work.

Definition

Cyber-physical systems refer to those applications areas, that are highly integrated into our lives, such as public infrastructure (smart grids), transportation (self-driving cars), or health care (medical monitoring).

Critical sytems/infrastructure (CI)

The term *critical infrastructure* is used to describe facilities and systems dealing with issues such as schools, water supply, and telecommunications [29]. Moteff and Paformak describe critical infrastructure as “the basic facilities, services, and installations needed

for the functioning of a community or society, such as transportation and communications systems, water and power lines, and public institutions including schools, post offices, and prisons. [111]"

Again the digital transformation is marching fast in these areas and in a given time it will be necessary to reconsider what is critical when software is applied in almost all critical systems nowadays.

2.1.4 Resilience

Generally, it can be said that *resilience* describes the capability to withstand failures and recover from them in an appropriate and timely manner.

Given how much software is entangled in our daily lives and how critical systems depend on it, it can be deduced that resilience is crucial to large software systems.

In this context, we can also mention the concepts of **fragility**, **robustness**, and **antifragility**.

Fragile systems are difficult to modify and expand. They cannot adapt to change and suffer greatly when experiencing perturbations. It is critical to recognize that a fragile system **will** eventually break [154].

This can be improved by designing robust systems, which are less prone to suffering harm when confronted with change and stress. An example provided by Bishop and Elliott is a software's ability to handle unexpected or purposefully wrong user input that is intended to cause a buffer overflow. Robust software would gracefully terminate, and provide the user with useful information about the encountered problem [21].

At this point, it is appropriate to mention the trade-off between **robustness** and **efficiency**. Integrating checks in your software for various possible system failures might not be efficient, but it is robust. This trade-off with efficiency appears to be an important concept and is discussed further in chapters 4 and 5.

In July 2019, Margaret Heffernan brought a more prominent example:

"*CEPI's (Coalition for Epidemic Preparedness) developing multiple vaccines for multiple diseases, knowing that they can't predict which vaccines are going to work or which diseases will break out. So some of those vaccines will never be used. That's inefficient. But it's robust because it provides more options, and it means that we don't depend on a single technological solution.*" [65]

However, even robust systems can withstand failures only to a certain point. The next step is to design resilient systems. Resilience can have multiple meanings which result in a variety of definitions in the literature depending on the context. Carpenter et. al try to provide an exhaustive definition considering three possible interpretations of resilience [32]:

- metaphor related to sustainability
- property of dynamic models
- metric in socioecological systems

By bringing the example of polluted water supply or dictatorships, both of which represent quite resilient system states, they argue that resilience can be desirable or not. On the other hand, sustainability is seen as an overall goal that covers the preferred states [32].

Regarding software engineering, the focus lies on the ability of components such as networks, operating systems, servers, and so on, to resist the effect and recover from unintended failures, intentional attacks, and other disruptive forces in a proper manner and duration [113].

We can particularise resilience further by specifying the individual capabilities [116]:

- Absorptive capability measures how much the initial negative impacts can be reduced.
- Adaptive capability measures how well the system can adapt to the negative impacts.
- Restorative Capability measures how fast and well the system can recover from the impacts.

Particularly, in systems theory, the notion of resilience carries great importance. Donella H. Meadows argues that it can be viewed as *elasticity* and that it measures if and how well a system can persist in a volatile environment. Persisting in this context does not equal being constant or static over time. Especially feedback loops are mentioned as relevant as these facilitate learning, creating, and evolving more complex restorative structures. She stresses that systems need to be managed for resilience, just as much as they are managed for productivity and stability [101].

However, it becomes increasingly difficult to design and implement resilient software. One reason for that is the increasing size of our software systems. Taleb explains that the large is doomed to fail. He argues that the larger a software programme is, the harder it becomes for it to change and adapt to stress [156].

Therefore, Taleb introduces the concept of *antifragility*. The idea behind this is that an antifragile system, when stressed at the right level, will come back even stronger in the sense that it adapts to its environment. We can find many examples of this in biology such as human muscles, or the immune system. One mechanism of achieving antifragility in software systems could be attributed to *Chaos Engineering*.

Certain factors can be considered when measuring the resilience of a system. Nowadays large software systems have many integrations and interdependencies as described in the

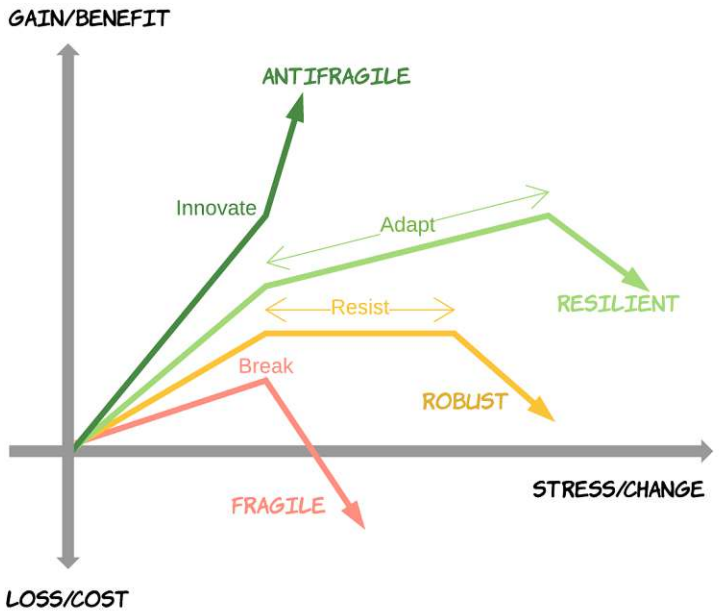

Figure 2.2: Visualizations of the described software system characteristics [74]

previous chapters, regardless of their use of microservices. For instance, an ISP (internet service provider) could offer certain services to its customers over 3rd parties and other interfaces. This results in complex networked systems that display more points of possible failures than self-contained systems do. They are highly connected and dependent so the whole service may fail if one critical component cannot deliver the necessary data [182].

While it is known that complexity is increased with the growing number of integrated and combined systems, and therefore resilience is at risk, today's applications increasingly rely on external technology platforms to provide their service [181].

This leads to another crucial issue, the diffusion of responsibility. The more platforms and people are involved to provide a certain service, the less responsibility might be taken by each individual. Taleb argues that those, who have nothing to lose (no *skin in the game*), will never learn from their mistakes [159]. Considering this, the importance of the human element in designing and implementing resilient or even antifragile systems cannot be denied. Even the above-mentioned approaches to antifragility emerge to be social-technical solutions rather than solely technical. For example, Chaos engineering forces human reactions to the injected failures and chaos, thereby making the system antifragile [74].

It is, therefore, crucial to consider these human aspects when exploring the possibilities we have in software engineering to ensure resilience, promote antifragility, and discuss the consequences should these principles be neglected.

Definition

Fragility is a property describing systems that cannot adapt to change and suffer greatly when perturbed.

Definition

Robustness is a system's ability to withstand known perturbations until a certain point.

Definition

Resilience is a system's ability to recover from unknown perturbations in an appropriate time and manner.

Definition

Antifragility is a system's ability to adapt to the environment when perturbed, thereby improving.

2.1.5 Sustainability

Sustainable development manages to satisfy the needs of the current generation without compromising the chance for future generations to satisfy their needs [120].

While this consideration has its roots in ecology, nowadays sustainability is relevant in almost all areas of our lives. Recently it has become an important researched topic in software engineering.

The reason for that lies in the current understanding of sustainability in a software context. The Karlskrona principles talk about the following dimensions that need to be considered [19].

- **technical** - deals with the perpetuation, maintenance and evolution of technical infrastructure and information, such as systems or data, in an ever-changing environment.
- **economic** - usually refers to profitability and prosperity in monetary terms. The focus lies on "assets, capital, and added value".

- **environmental** - considers the long-term effects on our natural systems. This dimension is the most well-known in terms of sustainability and includes discussions about resources, climate change, pollution, etc.

- **social** - refers to developments in society or other human organizations/groups. Aspects such as democracy, employment, or social equity are discussed here.

- **individual** - concerns itself with the individual human well-being, and including views on self-respect, freedom, education, etc.

All these dimensions have an impact on software systems and vice versa.

The difference between the last two dimensions might seem trivial but it is an important one. Particularly, in the field of software engineering, technology can be sustainable in the social dimension while extremely damaging to the individual. One example includes the Chinese Social Credit System. The idea behind the project is to create a national trust score that rates what kind of citizen you are [25]. The advantages are obvious. Such a system may likely improve society in general as it is being "nudged" in a certain direction. However, exactly that process has been criticised by many. Margaret Heffernan explains that "the technological opportunity to nudge, tempt and even mandate behaviour is a wickedly clever way to enforce predictability." [64] This can be useful for the government to manage and steer society in a certain direction. Unfortunately, it happens at the expense of individual sustainability, which is exactly why this dimension is critical

This means that sustainability should be achieved in all these dimensions, however traditional software engineering practices do not support all these aspects. Here sustainability is often only associated with resilience, and while these two are deeply intertwined, resilience often only covers the technical dimension of sustainability. Economic aspects are usually also considered, however, the environmental and social views are just as important.

For instance, it is crucial to consider electronic waste management as well as energy consumption to support environmental sustainability. One mention-worthy issue in this area is the impact of blockchain technologies on energy consumption. It is estimated to consume 12.76 TWh yearly [47]. Additionally, we know that the cyber-infrastructures supporting our diverse cloud devices and services can be held responsibly for up to 85% of the total environmental impact [120, 127].

One dimension has received the least attention regarding sustainability: social aspects. This dimension supports the well-being of software communities and developers. It deals with shifts in the human mindset and our perceptions of sustainability and software engineering [7].

Definition

Sustainability is a principle where resources are consumed in a way that permanent

satisfaction of needs can be guaranteed by preserving the regenerative capacity of the involved systems. It covers the economic, environmental, individual, social, and technical dimensions.

2.1.6 Security & Safety

The terms *safety* and *security* are often not well understood, misunderstood or even confused with each other. When we talk about safety and security, regardless of the context, we talk about system capabilities.

Safety describes a system's inability to affect the environment in an unwanted way [17]. We might want our systems to have an impact on the environment, however, we want to control which impact that is. A safe system means it won't have undesired or uncontrolled effects on its environment. For instance, in software engineering, this means that we don't want our software systems leading to plane crashes or alienation and polarisation in society [163, 50].

Security, on the other hand, describes the inability of the environment to affect the system in an undesired way [17]. Open systems interact with their environment as described in 2.1.3. Nonetheless, these inputs shall not lead to unwanted or uncontrolled behaviour in a system. Another example from software engineering displays that, security does not only regard intentional external attacks. Unintentional internal actions might also result in a software system outage or blackout [61, 60].

The concrete definitions of safety and security now depend on the type of system, its environment and the types of possible unwanted consequences. In software engineering safety is concerned with the protection of life, health, and the natural environment from possible system harm, while security is more concerned with integrity, confidentiality, and availability of information in the system [17].

These differences lead us to understand that safety and security are complementary topics, both critical to our modern software ecosystems. Methods and technologies developed for software safety have been around for a while and security experts would benefit from the established and experienced approaches. Similarly, safety experts may soon be in dire need of security mechanisms as Bartnes describes:

> *"... we will see more use of open communication networks for remote control of industrial and transportation applications. When vitally important commands are transmitted through such open networks, security techniques such as encryption and access control will become indispensable for safety. Security techniques will have to become an integral part of safety thinking".* [17]

In software engineering safety and security are deeply intertwined. It is safe to say it will be extremely difficult, if not impossible, to keep a software system secure if it is not safe, and vice-versa. This insight is extremely important considering that many other safety-critical systems, from nuclear and defence applications to everyday domains including

medical devices, traffic control, smart vehicles, and interactive virtual environments, highly depend on software to fulfil their tasks [97].

Definition

Safety and *Security* refer to a software system's capability to interact with its environment without affecting it and being affected by it in an undesirable and uncontrolled way.

2.1.7 Innovation, Progress & Maintenance

These terms are widely known and commonly used in everyday life. The words *innovation* and *progress* are often even used as synonyms, however, this might be a misconception.

We know that innovation and progress are critical for sustainability, yet recently use of the term innovation has increased, while progress seems to be mentioned less. This was different in the past, so how did this development unfold?

Klaus Kornwachs says that the term progress had become an integral part of the European worldview since the 18th and 19th century [90]. This development might have reached its peak at the beginning of the 20th century and the first World War.

We have to investigate what kind of progress can be observed since then. One particularly prominent example is the development of the internet. Information became widely accessible and the network was set up in a resilient and fail-proof nature since the dependency was distributed among various smaller players, rather than a few big players.

This has changed drastically to this day. The internet is used by a few monopolists to surveil and manipulate its users, which can be called Surveillance Capitalism [188]. Another approach is to use it for the exploitation of labour. Working standards are being undermined on a global level and the weakest in the system exploited. Technologically the internet has lost its resilience since a majority of the services is hosted and maintained by a handful of US-American companies. This centralised shift is an innovation, however, it can hardly be called progress [169].

More examples can be found in academia, where the amount of submitted patents and papers defines the success and value of a researcher. Yet it is rarely discussed if this quantitative value corresponds with the qualitative value.

Similar behaviour can be observed in medicine. When treating patients, the whole range of pharmaceutical possibilities is taken into account. Gawande argues that a more focused approach might yield better results. he proposes detailed interviews with the patients to identify their goals and adjust the therapy accordingly. This could save space in hospitals, money, and even lives [140].

The list of examples goes on, and Welzer clearly states that

"We systematically confuse innovation with progress." [140]

What is necessary for innovation to turn into progress? Steward Brand elaborates on a phenomenon that many computer science students have likely encountered. Soon after leaving university for industry, the shift from innovation to maintenance becomes very clear. Most resources flow into maintaining legacy systems which constitute the infrastructure, that we are highly dependant on. New functionality usually extends this infrastructure, rarely replaces it, and even if it does, the new functionality is heavily influenced by the properties of its predecessor [27].

Lee Vinsel states,

"The sheer reality is that about 70% of engineers maintain and oversee existing systems. Only a small minority of working engineers have jobs focused on invention and the "research" part of R&D." [171]

David Graeber summarises this phenomenon in an expressive sentence,

"Most work is keeping things the same: you buy a cup once but wash it a thousand times." [59]

Why is maintenance so important? A crucial interaction exists between innovation and progress with maintenance being right in the middle. As mentioned above, innovations such as new features commonly interact or depend on existing infrastructure. Eventually, this innovation becomes infrastructure itself, that other features and society depend on. Therefore, it needs to be set up in a way that long-term maintenance is guaranteed. We are then speaking about technical and organisational progress.

However, the most considerable issue turned out to be the significant resources, time, effort and expertise, that maintenance requires. While our digital nervous system depends on it, we tend to forget about it, let it fade into the background of our culture, as well as our financial planning as Lee Vinsel states [171]. While the term innovation has been gaining in popularity more than maintenance or progress, the rate of effective innovations has been decreasing since 1970.

For sustainable progress a holistic approach from innovation to infrastructure, maintenance, and finally progress is necessary.

Definition

Innovation describes something new or the creation of something new, such as a product or feature. It can bring added value but it can also lead to worse conditions.

Definition

Maintenance describes the process of preserving and sustaining existing infrastructure and functions. It constitutes a long-term step where innovation, and subsequent infrastructure becomes progress.

Definition

Progress describes a positively assessed advance towards a higher level of development. It always brings added value.

2.2 Expert Interviews as a Research Method

Expert interviews constitute the backbone of this work. The information gained and gathered in the previous sections was contextualised to the interviewees' fields of expertise.

Firstly, a framework for expert interviews needed to be developed. For this, a literature review of selected documents about the design of expert studies is conducted. Only those papers were considered that either researched a similar domain as this work or described expert studies and their approach in a more general way. The findings were summarised, analysed, and operationalised to create a framework including the design and strategic planning of the study.

2.2.1 Interview Process

A typical interview process consists of multiple activities. Some need to be planned and defined ahead. Others cannot be concretely fixed as they depend on the data resulting from the interviews. Hove and Anda have briefly summarised the involved activities as follows [69]:

- Scheduling
- Collecting of the interviewees' background information
- Preparing the interview guide
- Discussions/meetings
- Summary writing/transcribing

All of these tasks take up a significant amount of time, and the most time-consuming part, the analysis, is not even being considered at this point. Therefore, a proper understanding

of the activities was necessary to conduct the survey in an appropriate time frame without risking invalidation.

This section describes the required steps and possibilities in expert interviews.

Problem Statement & Interview Types

At the beginning of every survey stands a problem. Defining a clear problem statement is the most critical step in most research methods.

Thereby, the selection of the interview form heavily depends on the problem statement and aim. In the literature, various interview types are mentioned.

Eriksson gives the following categorisation [49]:

- Tutorial Interview: Experts are asked to introduce the domain.
- Focused Interview: Topics (not precise questions) are prepared in advance.
- Structured Interview: Detailed preparations are made for interviews in a depth-first manner.
- Teachback Interview: The expert describes the process to the interviewer who then describes it back to the expert.
- Observations: Interviewee is observed during a task.
- Protocol Analysis: Protocol is written and the expert is asked to think out loud during a task.

A more generalised classification is given by Bogner et al. [23]:

- Exploratory Interviews: The aim is to explore a topic. These should be as open as possible.
- Systemizing Expert Interviews: The aim is to extract specific knowledge or information from the expert. It can follow an open approach but also includes standardised design such as in the Delphi Method. [6]
- Theory-generating Interviews: The aim is to use expert opinion as a starting point in formulating a theory.

Furthermore, the questions can be formulated in an open, structured or semi-structured way to guide the interview. Various structuring techniques can be used such as repertory grids, as explained by Hart, or card sorting, where the expert simply groups a randomly distributed set of cards by his criteria of choice. The cards are labelled with a concept, and the results can then be easily compared [63, 49].

Expert Selection

When working with quantitative methods, sampled data is a representation of a probabilistic instance of a phenomenon [108]. In these cases, higher numbers are better to achieve a more accurate representation.

This is not the case with qualitative experts interviews. It is important to achieve diversity, but otherwise fewer experts are sufficient. In theory, even one expert is enough if he or she is truly credible. Ming Li and Smidts picked 10 out of 30 possible candidates [108].

Most literature mentions defining certain criteria by which the selection process is dominated.

Ming Li and Smidts have selected experts based on their credibility, knowledgeability, and dependability [108]. They reviewed the number of publications, years of experience in the industry (consulting, management, hands-on...), diversity, as well as the experts' willingness to cooperate given the methodology to be used.

A similar approach was described by Mergel et al. They argue that experts should be chosen by "virtue of their proximity to the research question". This means that the focus should lie on those who have the most valuable and relevant information. Additionally, they mention that interviewees should allow access to in-depth insights, which describes a personality trait and corresponds to Ming Li's and Smidts' "willingness to cooperate". Mergel also mentions the acquisition technique known as "snowballing" or "chain-sampling" which refers to the acquisition of experts by direct referral of already selected participants [104]. This technique, of course, is subject to numerous biases, such as Anchoring or Community bias.

Christopoulos describes an advanced snowballing technique called "Peer Esteem Snowballing" which is meant to tackle some of the weaknesses of snowballing and give a better feeling for the sampling population size [37].

Furthermore, he describes the following aspects to be the most important ones when selecting experts [37]:

- Experts should be authoritative enough to provide sought data.
- Consider the total size of the population of experts.
- Consider if population boundaries can be drawn.
- How difficult is it to reach the experts?
- How can experts be motivated to engage more actively?
- How can bias be measured and interpreted in non-response?

Finally, Chang et. al confirms the above-mentioned criteria. In that paper, the interviewees were selected based on their knowledgeability and diversity. It was considered important to cover multiple sectors [34].

Interview Conduction

This step is highly dependant on the selected interview form.

Yet, since interviews are a relatively old research method, much experience has been gained throughout the years and certain guidelines were developed.

Hove and Anda describe five types of questions that can be asked [69]:

- Behaviour/experience questions which are answered with descriptions of experiences, behaviour or actions.
- Opinion/value questions which investigate how people reckon certain issues.
- Feeling questions aim to understand the emotional responses of the interviewees to certain thoughts and experiences.
- Knowledge questions identify factual information held by the interviewee.
- Sensory questions which aim to capture the experience of the senses.
- Background/demographic questions identify the characteristics of the interviewee.

According to them, all of these questions can be asked in the present, past and future tense. Asking "what" and "how" rather than "why" or yes-no questions is recommended. Even visual artefacts might help the interviewee to remember or imagine certain concepts [69].

Often questions are thematically stacked in different parts. For instance, the first part could include general questions about the interviewee's background or evaluation of the domain, asking them to define certain terms and concepts. The consecutive questions could then focus more on the experts' opinion [165, 104, 88].

Hove and Anda also stress the importance of good interviewing skills. It is necessary to encourage the experts to talk freely, ask relevant and insightful questions, and follow up and explore interesting topics [69].

A relaxed atmosphere can be equally important to encourage experts. Particularly when interviewing managers, Trinczek argues that it is crucial to create a relaxed atmosphere to allow for more creative and open-minded thinking and to draw the managers out of their "question-answer" mentality [164].

Qualitative interviews are normally conducted with a tool that enables synchronous communication, such as online-video interviews, phone calls or face-to-face interviews. Personal meetings often include the advantage that social cues can be gauged. This information might be lost in a video or phone call. Similarly, some nuances might be lost in translation if multiple languages need to be considered [104].

Analysis & Validation

Finally, the gathered data needs to be prepared, aggregated and analysed. This step often involves hours and hours of transcribing the conducted interviews or using them as input in statistical programmes such as R or SPSS.

Afterwards, various analysis techniques can be applied to reach different goals. If the data was gathered by standardised questionnaires or other quantitative methods, simple averaging techniques can be used such as the Classical Model or Bayesian aggregation [108].

Otherwise, if the interview was conducted qualitatively, other methods need to be applied. One of the most commonly applied approaches is **Coding**. This describes the process of indexing or classifying the transcribed text to identify thematic ideas. Tunn et al. used this technique for their research about Business Models for Sustainable Consumption. The tool used for coding was Nvivo. The data was first coded according to the questions, and subsequently, coded by specific content to identify thematic ideas, as mentioned above [165].

Another popular approach is the **Grounded Theory** method. The aim is to develop a theory based on social processes such as "causes, contexts, contingencies, consequences, covariances, and conditions". By understanding the patterns and relationships among these elements concepts can be examined based on information provided by the interviewees [104]. Leffers and Mitchell used the Grounded Truth approach to guide the theoretical direction for sustainability in global health programs. They identified codes and then included a secondary literature review. Finally, the **Qualitative Comparative Analysis (QCA)** method was used to identify how much the findings are supported by the data from the literature review. This is a well-known validation method [93].

Secondary interviews can also be conducted to validate the findings. Tunn et al. used the first interview round to derive a theory and build a business model based on these findings. Then a second interview round was conducted with the same experts to determine which combinations of the business model elements were important to experts [165].

Finally, we take a look at **Qualitative Content Analysis**. It aims to reduce the large text material to a manageable size without wasting information. The approach by Mayring contains the steps *Summarizing, Explicating and Structuring* [98].

Similarly, Meuser and Nagel advocate a six-step process consisting of the following points [105]:

- Transcribe
- Paraphrase
- Code
- Thematic Comparison
- Sociological Conceptualisation

- Theoretical Generalisation

This approach is supposed to also examine the potential influence of experience that was gained outside of the professional realm.

2.2.2 Risks & Challenges

Interviews as a research method come with a variety of challenges and risks.

Cooke and McDonald argue that the required introspection by interviewees and subjective interpretation by the interviewer poses issues in the research methods. Apart from that, it is difficult for humans to express knowledge verbally and this only increases in difficulty with the increase in the expertise of the interviewee.

They argue that expert knowledge often involves heuristics, rules, and strategies that are procedural in nature. This seems to be particularly challenging to express as some of these processes happen subconsciously. Intuition for example is considered to be a skilled pattern recognition ability that can rarely be expressed verbally [39].

This is confirmed by Eriksson who agrees that experts often have difficulties expressing their knowledge. This also leads to a decline of the interviewer's difficulties in understanding enough to use the knowledge. It is difficult to model reasoning strategies, and that a general mismatch exists between the nature in which experts express their knowledge and the way interviewers perceive that knowledge [49].

Particular attention needs to be paid to the possible biases. Ming Li and Chan have mentioned Overconfidence Bias and Location Bias. These can be tackled well by asking the experts to find reasons that would contradict their initial comments. The assessment and compensation of these biases can be called *expert calibration* [108].

Finally, personal characteristics need to be taken into consideration as well. When encountering interviewees that barely talk, it helps to ask open questions, make clear that there are no right or wrong answers, and prepare the interviewer in advance if possible [69].

When confronted with the opposite type, it is important to find out when it is necessary to let the interviewee ramble to gain relevant information, and when the interviewer should interrupt. Soft social cues can be used as well such as stopping to take notes or to nod, and to pop new questions as soon as the interviewee makes a pause. In any case, steering back should always happen gently [69].

CHAPTER 3

Methodology

3.1 Introduction & Motivation

In the previous sections, we have defined the terms complexity, emergence, resilience, and system. We have also discussed alarming examples in software engineering where these concepts might have been critical aspects to consider. In light of these findings, we cannot exclude the possibility of future threats, therefore it is necessary to acknowledge and understand the complex nature of these risks.

Due to these complex interrelationships of impact factors, it would have been extremely difficult if not impossible to model the first pillars of a sustainable software engineering paradigm by quantitative data analysis such as generalized questionnaires, observations, or with the use of machine learning.

Another challenge was posed by the target group. The aim was to gain a better understanding of behaviour in modern software systems, particularly regarding sustainability, and to discover what can be learned from biology. Due to increasing inter-connectedness in modern software systems, this is a highly complex issue, on which only experts of certain fields can elaborate.

These professionals are hard to reach. Considering the complex nature of the issues, as well as, the characteristics of the target group, expert interviews emerged as the adequate research method for this problem. Subsequently, relevant principles for sustainable software engineering were derived, compared with concepts already known from the literature and validated in a second interview round. Finally, the limitations of the results were discussed.

In the following sections, the concrete approach used in this survey will be described.

3.2 Application

For the aim of this work, it was necessary to consider the nature of the topic. We were dealing with highly interconnected and interdisciplinary areas, which all have their interpretations and understanding of complexity, resilience, sustainability, systems, etc, while it was our goal to combine these ideas and operationalise them for software engineering.

This greatly influenced the selection process, interviewing style and analysis. To support the complex nature, the interview process was conducted in a multilayered fashion and consisted of three parts.

Part 1

In the first step, the grounded theory method, as described above, was applied. Based on the experts' opinions and insights, it was our goal to understand the patterns and relationships behind the concepts and base a theory on that. Concretely, the causes, consequences, and contingencies of complex system behaviour were examined to later derive the first principles for a sustainable software engineering paradigm.

For this method, the selection of experts was crucial. In the first step, a table of possible candidates with their expertise, location, and affiliation was compiled. This first table included a high number of people, however, it was further narrowed down before proceeding with the interview steps.

Experts in the field of biology, complexity science, software engineering and other areas were identified based on the following criteria:

- Credibility (years of experience, number of publications...)
- Knowledgeability (diversity ...)
- Dependability (experts must be reachable through existing channels)

A snowball approach was applied to compile the table of possible interviewees. The exact number of experts was not fixed. It was crucial to reach a critical mass of diversity since we were dealing with an interdisciplinary issue. Therefore, the expert amount and practical selection heavily depended on the knowledgeability of the contestants. No more experts than necessary were interviewed. The form of this survey followed the theme of "quality instead of quantity".

A semi-structured interview about complex system behaviour in biology and in general was prepared. There was a strong focus on the five dimensions of sustainability, namely economic, technical, environmental, social and individual. The questionnaire served as a guideline, however, the interview style was left as open as possible to allow for creative

	Occupations	Affiliations	Core Competence	Language
Expert 1	Professor	Sigmund Freud University Vienna University of Vienna	Evolutionary Biology Ecology and Conservation Biology Issues	German
Expert 2	Author, Professor, Researcher	Virginia Polytechnic Institute and State University	Science, Technology, and Society (STS) Maintenance	English
Expert 3	Researcher	Spanish National Research Council Pompeu Fabra University in Barcelona	Complex Networks, Evolution of Technology Theoretical Ecology	English
Expert 4	Professor, Researcher	University of Vienna	Molecular Biology, Philosophy	German
Expert 5	Researcher	Complexity Science Hub Vienna	Cognitive Science, Computer Science, Physics	English
Expert 6	Lecturer, Researcher	Institut d'Études Avancées (IEA) Paris Université Paris-Saclay, Complexity Science Hub Vienna	Evolutionary Biology, Philosophy Systems Theory	German
Expert 7	Author, Mathematician, Entrepreneur	Vienna University of Technology former member of external faculty of the Santa Fe Institute (SFI)	Complexity Science, Systems Theory	English

Table 3.1: Expert Information

and explorative thinking. All results were anonymised to motivate the experts to speak more freely as well as eliminate key accountant bias.

The survey was concluded after interviewing seven experts with current and former affiliations to international universities and research institutes. The competencies of these experts mainly referred to biology and complexity research, but also included expertise on the evolutionary theory of technological complexity, the contingencies between science, technology, and society (STS), innovation, and the software aspect itself. Most of the experts already had several years of experience in their field, but a young researcher was also included to ensure diversity and a modern perspective. The table 3.1 gives a detailed expert overview.

Part 2

In the second part of the survey, previously gathered expert insights were evaluated using a content analysis approach. The exact method was selected based on the acquired data. The analysis included the following steps:

- **Transcribing** - This process has the aim to put the interviews into writing and conduct certain summarising and cleaning steps such as the removal of redundancies, interrupted sentences, noise, etc.
- **Coding and structuring** - By performing this step a conceptual basis for the thematic comparison is created.
- **Thematic comparison** - In this step, the interviews are not grouped by the interviewee but by subject area, and compared to each other. A comparative literature review was included in this process, which served as the first validation.
- **Paradigm compilation** - All previously gathered insights and information were used to derive relevant principles for sustainable software engineering in complex systems

This was a highly iterative process since the approach depended largely on the interview progress and results.

	Occupations and Competences
Expert A	IT- and Management Consultant, Researcher, Professor
Expert B	Information Security Consultant, Lecturer
Expert C	International Crisis Prevention Expert, Lecturer, Author
Expert D	Performance Management Consultant
Expert E	Insurance Consultant, Enterpreneur

Table 3.2: Industry Experts Information

Part 3

In the final step, the previously derived principles for sustainable software engineering were discussed with experts in a secondary interview round, that served as evaluation.

Two approaches were possible:

- Secondary interview with experts from the first interview
- Secondary interview with newly selected experts

Based on the results from previous steps, the second approach was selected. Another expert list for interviews was compiled and this time, the focus was set on experts with pertinent industry experience. In doing so, it was meaningful to choose experts with a broader view of the industry, such as management and consultants, rather than concentrate on specialists, such as developers. A total of five experts was interviewed. Information about their occupation and competence can be found in 3.2.

The secondary interview round served as the second and final validation of the developed theories and criteria. Firstly, the interviewees were introduced to the principles relevant to sustainable software engineering. Then they were confronted with already existing examples in the industry to find out which of these principles are acknowledged, where the obstacles lie, and how these can be surmounted.

Finally, the results were operationalised to fine-tune the principles and compile the final version. Additionally, the discussion part addresses the constraints and what is necessary for overcoming the challenges.

3.3 Expert Calibration

In expert interviewing methods, it is extremely important to consider the possible challenges and biases one might encounter. It was mentioned previously, that snowballing was used to gather a broader list of possible candidates. While snowballing is an excellent technique for interviewee acquisition, it could introduce a certain bias.

Much attention was paid to the formulation of questions and interview style to account for further possible biases.

Reasons for an expert's beliefs and a description of his understanding of the world which is being studied were obtained. Experts were also encouraged to think about arguments against their initial opinions. This helps reduce overconfidence and confirmation bias.

All experts were granted full anonymity and confidentiality, which also reduced the risk of potential key informant bias.

CHAPTER 4

Results

This chapter first summarises the results from primary expert interviews. Scientists, authors and professors from different countries and institutes with core competencies in the fields of biology and complexity research were interviewed.

During the expert interviews, various topics were addressed in regards to biological systems, their resilience mechanisms and dynamics. The findings are summarised in the following chapters. Subsequently, relevant principles and criteria for a sustainable software engineering paradigm are derived and evaluated by a comparative literature review. In a second evaluation, round managers and consultants from the industry were interviewed to validate the findings and put them in a larger context.

4.1 Principles of Biological Systems

In the primary interview round, it was discovered that most biological systems don't display linear behaviour. Rather, they are open and even *dissipative*, meaning they exhibit certain characteristics, they did not have when formed, through the exchange of energy or matter with their environment. This indicates that biological systems are complex by nature.

Complexity has already been studied by Aristotle. Other great researchers such as Galileo, Newton, and Laplace followed. It was believed that anything could be predicted. However, this was debunked quickly by the first weather models. Nowadays, various institutes are dedicated to the study of complexity with the Santa Fe Institute portraying a noteworthy mention, since it was the first institute and propelled research in this area [109].

This section discusses which robustness, resilience and antifragility principles appear to be of particular significance in biological systems, such as the ecosystem or immune system, and how these relate to complexity.

Additionally, options for analysis and predictions in such biological systems, as well as possible metrics and indicators to describe their state, are discussed.

From this perspective, the process of evolution needs to be mentioned. In the long term, all existing biological systems must be adaptive and antifragile, otherwise, they would not exist due to selection pressure.

Evolution and Fitness Stress

In the context of biological systems, the process of evolution appears to be of utmost importance. The saying "Evolution always wins." portrays that biological systems adapt to their environment as explained by expert 1. External perturbations, i.e. deviations of the system's normal state, can result in an adaption of the system, as everything that doesn't fit in these deviations, is not selected and thereby ceases to exist. Examples of perturbations could be related to temperature such as fires or the effects of climate change, but also include storms, flooding, etc.

What's also interesting to note, is that systems that are adapted to rhythmic distress, need that stress to persist. Evolution has not only created single organisms and species, but every existing ecosystem has developed in co-evolution with its environment and agents. Therefore, the system's parts also depend on the system's properties including limiting factors and perturbations.

An example in this context is the immune system which is permanently challenged by bacteria, fungi, viruses, spores, etc. The immune system can be regulated up and down, however, it is always in interaction with its environment. Therefore, it appears to need counterparts, and when these are lacking, allergies could be the result, which portray an overreaction of the immune system.

These thoughts fuelled the notion that biological systems could lose their degrees of freedom if never encountering any stress. Losing degrees of freedom means decreasing complexity in this context. In other words, some level of perturbations is necessary to have the motivation to improve. If there is no selection stress, there is no reason for the system to adapt and evolve. This can be depicted by comparison to rubber or other elastic materials. These can become brittle when permanently stored in the same condition. As they stabilise they become fragile. However, perturbations now and then could change the local bridging, and increase the system's resilience against new perturbations.

Although, it is critical to note that not all perturbations are good for biological systems. If you consider yourself an antifragile organism, it means that you have learned from your past challenges. In principle, the more obstacles you have overcome, the more antifragile are you today. However, this doesn't mean that you can master all hardships. The improvement heavily depends on the quality of the challenges, and if they can bring you into the *flow*, a state between boredom and panic [41]. Naturally, if the challenges are too big, you're constantly in panic mode and you don't learn anything. This was stressed by an evolutionary biology and philosophy expert.

Evolution is driven by the fitness landscape. In this panic mode there is only one direction, and striving too far from it, could mean that you don't get selected because the pressure is too high. However, when you're in the flow, you can move in different directions on the fitness landscape. This is called a *drift* in evolution, as you're going in a direction that is not driven by better fitness. The more time you spend in this flow, the more can you grow as an antifragile system.

A closely related concept is that of the *adjacent possible* described by Stuart Kauffman [81]. Displaying certain characteristics or having a set of skills enables an organism to perform various actions. This is the adjacent possible. However, performing those actions opens up new opportunities; The adjacent possibilities from this point on are extended. For instance, a screwdriver could be used as a harpoon to catch a fish. This again enables cooking and eating the fish, which enables further possible actions.

An ecological example could be wildfires enabling niche constructions. In ecology, the concept of *succession* (recolonisation) is very well-known. Some extreme events such as wildfires could facilitate new possibilities as they lead to various phases of succession. Then niches and niche constructions can be encountered again.

However, this is a side effect of complexity. Niches don't develop because the organisms want to be more flexible. The environment changes and perhaps opens a new niche, which suddenly enables organisms to perform functions they didn't know were a possibility, thereby increasing flexibility. These actions again influence the environment itself, which feedbacks anew to the organisms. The idea is that this cycle of co-evolution with the environment is what propels the increase of complexity, redundancy, and other phenomena.

Finally, a statement about the mutation rate needs to be made. As long as the system is adapting to something, evolutionary there is a need for mutation. If we consider viruses, we know that they tend to have a really fast mutation rate meaning errors in their copying mechanisms are not unusual. This indicates that the copying process is not critical in comparison to cell replication in humans. For an organism that is so complex and critical like a human, a slow mutation rate is necessary. Therefore, humans have developed more intricate control and repair mechanisms.

Evolution turns more sophisticated, the more complex an organism becomes. For instance, to keep an elephant alive the inner processes have to be highly fine-tuned. Peto's Paradox deals with the question of why organisms with more body mass develop less cancer or about the same rate as smaller organisms, even though they should be developing more since they have more cells that have to undergo cell division [162]. It appears that evolution has introduced more copies of the tumour suppressor gene TP53 in elephants. Redundancies were added explicitly in the control system. So it seems that this investment in a large complex organism pays off, whereas it doesn't in a gerbil since it is much more short-lived.

Niches

Many different species live together in an ecosystem and there is often competition between them. Ecological niches help ensure that they do not displace one another and that they can coexist. Birds that get their food in different ways and thus specialize in certain prey are a typical example.

Ecological niches introduce the idea that there are niches in ecology that need to be filled. However, this is not true. There are only organisms that have some possible actions. In the previous example, it could mean that some birds swim to catch their prey while others use their beaks for insects. These various actions result in different niches.

An important concept in this regard is the notion of *Exaptation*, which describes the use of a function for something that it wasn't meant for. The canonical example is the dinosaurs' ability to fly. There may have been fitness pressure to develop feathers to regulate their body temperature. Then later they also discovered that they can fly. The discovery happened by accident or out of curiosity which is intrinsic motivation and not driven by selection stress. Playing or similar activities, make it possible for organisms to acquire knowledge that might not have an immediate outcome but could be useful later.

This opens alternative pathways and introduces diversity, which are further relevant principles in biological systems.

Diversity

In a biological ecosystem, there are always species that act as predators and prey, a relationship that has often been examined and used to describe the dynamics in a biological system. When there is only one species that serves as prey, the fluctuations of the population can be seen very well. The canonical example is the relationship between the Canada lynx and its prey, the snowshoe hare. The fluctuations are well visible in graph **??**.

This is different in a species-rich system. The fluctuations aren't as visible anymore since the predator doesn't depend on solely one prey. This makes diversity one of the key principles for fostering resilience in biological systems.

Considering non-linearity in biological systems, the importance of diversity becomes even clearer. Small variations in the initial conditions might lead to vastly different final states. For instance, an ecosystem with high diversity is not particularly influenced by the introduction of new species, unless they find a new niche. When there are alternative pathways to resources, a small perturbation will not change the ecosystem's dynamics. This is different if diversity is low, and therefore alternative pathways might not exist. System variables then are very sensitive, and the ecosystem could develop in various directions.

Diversity also propels the adjacent possible. Mountain ranges for example are a reservoir of new species because every altitude represents its ecosystem due to different temperatures,

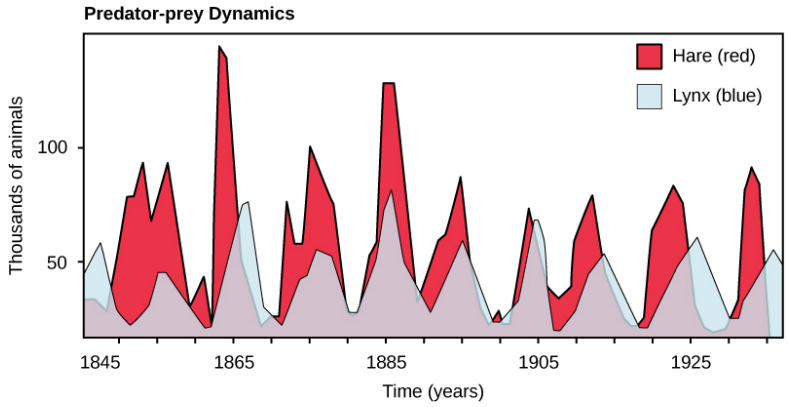

Figure 4.1: Fluctuations of predator-prey dynamic by example of Canada lynx and snowshoe hare [20]

oxygen levels, etc. However, a species or phenomenon existing in one level quickly may jump up or down. This increases the rate of new species due to the immediate possibilities that unfold, not because of additional resources.

Therefore, diversity is a prerequisite for alternative pathways.

Redundancy, Alternative Pathways and Multifunction

This principle has already been briefly mentioned in section 4.1 and describes the availability of multiple units of the same functionality.

For instance, we know nowadays that a certain percentage of stem cells doesn't turn into organ cells. Instead, every organ still has some stem cells left. They serve as reserves and don't change unless they are needed to regenerate organs or turn into cancer cells by mischance. They are highly modular, yet integrated enough that they can understand signals and react autonomously. This is explained in more detail in section 4.1.
Alternative pathways depict a form of redundancy that is founded on the notion of multiple interactions and interdependencies between the entities of a biological system.

Simplified versions of biological processes are dangerous because they often lead to mono-causal conclusions. However, there might exist alternative pathways in biological systems that involve entities, that aren't expected to be involved. Multiple ways to reach a certain final state or fulfil a function exist, and this property is shared among all biological systems. For instance, there are alternative pathways at the level of an individual organism, as well as, among different organisms or species in an ecosystem, etc.
The capability of a system's component to fulfil more than one function is called *multi-function* and is involved in alternative pathways. A banal example is the multifunction of all molecules in our body. It is clear, that the water molecule H_2O cannot be found only in blood, but also in saliva, urine, sweat, etc. The individual molecules are indifferent

towards their development, and can also alternate between components, e.g. molecules in saliva used to be in the blood. Similar arguments can be made with the ATP, TTP, GTP, and CTP molecules that don't know and don't care if they will be used as part of the DNA eventually or spend their time in muscle building and absorbing energy.

Additionally, we need to ask the question of how much redundancy should be introduced to achieve resilience in an antifragile system? Eventually, all systems will suffer from perturbations. The goal is to introduce redundancy and thereby keep the system functional as long as there is a fitness advantage or until reproduction. Beyond this, a trade-off between efficiency and fault tolerance needs to be made, i.e. it might be better to reduce redundancy to increase efficiency in an organism after reproduction.

Errors happen in nature all the time, which is why fault tolerance is so important. During the expert interviews, it became clear that one of the reasons for high robustness and resilience in a biological system are its redundant capabilities.

This requires the entities to be as independent as possible, yet still interconnected enough to be integrated, which leads to the next principle called *Modularity*.

Modularity and Self-Organisation

Each complex system has a certain topological structure and functions that it fulfils. The topology is usually outlined by the system's agents or entities, which can be connected closely or have loose relationships. In regards to possible collapses in a biological system, it was necessary to find out how these interdependencies influence the system.

Diversity was mentioned by the interviewees in the sense that resilient systems should have both, close and strong connections as well as wide and loose relationships.

On the one side, single components should be as little interconnected as possible. They should have established borders, meaning that an error in one of the parts will not propagate to the rest of the system. On the other side, there is also a need for an integrated system, since communication between the components is crucial for a complex system. This results in a trade-off between modularity and integration.

One theory about the origin of complexity is that it emerges through hierarchic systems. Herbert Simon proposes modular organisation as the main principle in the evolution of complexity [150].

Modularity plays a crucial role in *self-organisation*, i.e. the dynamic changes of the structure over time, which are defined by the agents' interaction [92, 149]. Ecologists have found that the modular structure that is formed by the systems' agents is not random but follows clear patterns. It is not yet clear why they follow these patterns [52]. One set of potential explanations focuses on the evolutionary process and selection stress, which require the system to be resilient and work as a coherent entity. Thereby, it has shaped itself, self-organised around some structure that allows the system to adapt to changes in the environment [18]. An example could be the introduction of redundancies among

species so that if one of them is removed, the system can still cope with the changes. This architecture provides the necessary degree of antifragility and resilience.

Finally, it can be asked if a relationship exists between a complex network's structure and its function in the biological context? Previously, it was stated that the network structure is not random in biological systems but follows a certain pattern. To identify what influence these patterns have on the function, it is necessary to differentiate what happens at the individual molecular level and what is dependent on the collective network structure.

An example is given by the protein structures and their relationship. Certain dynamic and topological properties can be determined about the network solely based on the structural properties. From a certain level of abstraction in such interaction networks, it is irrelevant how the single components are assembled or which properties the protein itself has biochemically. Solely the fact that the network has a specific topological structure is enough to explain phenomena. The details of specific nodes are not relevant anymore as has been explained by an expert on molecular biology.

Another example is the classification of proteins. So far they were classified by their 3D architecture and it was possible to derive correlations, which could state that a certain structural motif is involved in specific biochemical reactions. However, many interesting cellular and physiological functions go beyond biochemistry. If a kinase is going to regulate the metabolism up or down depends on the interaction partners and the network description, rather than the description of the individual players.

In systems biology, the group of Uri Alon is researching topological motifs. Even for small network structures, it is possible to describe a fair number of complex structural properties and dynamic behaviour, without any knowledge of the protein being necessary [9, 106].

This also fuels the philosophical discussion if processes can give better descriptions of fundamental things than entities. There are various perspectives if organisms can be seen as processes, and maybe it is less about which entities fulfil which functions, but rather which processes lead to these functions.

In immunology recently there have been several arguments that the differentiation between the body's substances and foreign substances is not as relevant as the fact that something is happening, i.e the process itself. Thus, even a strong reaction of the body's substances could be relevant for the immune system, while foreign substances might not call for a need of action, provided they are interacting in a normal range with the body and don't have significant influence. This would underline the theory that when something happens, the dynamics of the processes are more relevant than the entities that exist [126].

Feedback Loops

The concept of feedback loops is crucial for the regulation of complex biological dynamics. Feedback loops imply that the output of a system affects the system itself. This effect can be positive or negative, meaning it will either amplify or inhibit the system's dynamics.

This can be illustrated by the example given in section 4.1. When there are too many predators, the population of the prey sinks as an output of the system. However, this decreasing population also means fewer resources for the predator, which in turn inhibits its population. This is a typical example of a negative feedback loop.

It is often believed that nature is self-regulating and that there are negative feedback loops everywhere. However, it is more likely that systems that did not portray such properties simply collapsed and went out of existence. Only those systems with negative feedback loops persist. Positive feedback loops are very rare and usually only occur together with negative feedback loops. An evolutionary biology and ecology expert references a peacock's tail in this regard. The longer the tail grows, the more females it attracts, which means that more peacocks will have a longer tail. However, this positive feedback loop is checked by the peacock's predators that have an easier game catching them the longer their tail gets. The negative feedback loops regulate the dynamics. Without them, the system would eventually reach a tipping point. An example from organisms is the principle of apoptosis, which is programmed cell death. This mechanism is relevant so that our organs know when to stop growing. The cell knows when to activate this mechanism because it is in communication with neighbouring cells.

Mutations could turn the mechanism off by stopping the communication. An example is the development of cancer, which then can grow uncontrollably since apoptosis is not being activated anymore.

Other Resilience and Robustness Mechanisms

During the interviews, it became apparent that control- and repair mechanisms play an integral role in ensuring resilience and robustness in organisms. There are various examples of this, one being apoptosis that was mentioned in section 4.1. It is always activated by a signal or cascading signals, and the question of how it is decided at what point there are enough signals is an exciting topic for itself. Finally, the cell releases a final piece of information signalling that it can be metabolised now.

If this happens without the cell being in control, but for instance by mechanical stimulations such as heat, it is called necrosis and the cell is shredded. Signalling plays an important role here, as the cell innards come in contact with other cells. This serves as a warning signal for the immune system.

Another variant is called senescence and describes the process when a cell doesn't replicate anymore and is metabolically immobilised, however still alive.

All these are resilience and robustness mechanisms, however, they could also cause issues. One of the theories about ageing is related to senescence [148]. The more cells go into this senescent state, the poorer the functionality of the tissue. In this case, it would be better if the cells died and were replaced by new ones, instead of turning into cells, that don't work anymore.

Until a certain point, the tissue can compensate for the senescent cells as there are enough healthy cells. Afterwards, the tissue collapses for which the tissue system might still be able to compensate. However, after that, the tissue system collapses as well, and eventually the whole organism. Again the various levels of the system appear to be relevant for its resilience. When the organism can't compensate anymore, the *tipping point* is reached, which is further described in section 4.1.

The higher the evolutionary pressure for a process, the greater the importance of resilience and robustness mechanisms for it. The copying accuracy of DNA replication is extremely high due to many control- and repair mechanisms. If any errors happen during the copying process, it is possible to do a recovery of the last couple of steps, and perform repairs locally. The whole replication doesn't need to be repeated as this would be very costly. Of course, it is possible that even after all these local improvement steps something still goes wrong and the whole chromosome structure comes out faulty. As a last resort, a stop signal or a danger signal can be released to stop the whole process at the very end. These little local steps and control mechanisms enable such a high copying accuracy. For the process of protein folding, that was described before, the evolutionary pressure is not as high, and therefore there is much more fault tolerance.

Requisite Variety and Complexity Mismatch

Requisite Variety was described as a cybernetic principle stating that to cope with external perturbations a system must possess an internal variety at least as great as the variety of the perturbations. In other words, the system has to be at least as complex as the interactions that it is dealing with. These interactions can be seen as fluctuations and a certain minimum complexity is required to produce the correct response to these fluctuations [24].

This principle seems to be closely related to another concept named *Complexity Mismatch*. To describe it, a measure for complexity needs to be defined first. One possibility is to put action and interaction in the centre and pose the question: "At a given moment, how many different kinds of things will I have available to do that I can choose from?" If there is only one possible action, the degrees of freedom are low and subsequently, complexity is not very high.

Complexity Mismatch describes the state of two systems in interaction, where one of them exhibits higher complexity, in the sense that its agents have more possible actions to choose from [33].

If that gap grows and there is nothing done voluntarily to close the gap, eventually nature will step in to reduce that tension. Either the low complexity system has to increase in complexity, or usually, it's the higher complexity system that collapses. This collapse isn't a smooth break and commonly results in extreme events such as stock market crashes or earthquakes. An example is the crash of the stock market in 1929. The agents of the stock market, such as traders and investors, displayed much higher degrees of freedom in their actions than the agents in the regulating government agencies. The complexity

mismatch was so high, that eventually, the system with higher complexity (in this case the stock market) collapsed.

Future research could deepen and improve the understanding of complexity mismatch or requisite variety in connection with software engineering.

Tipping Points and Regime Shifts

In the previous sections, many processes and mechanisms have been mentioned that help to regulate, control, and repair a system. However, all these helpers support the system only until a certain point. After this point is reached, we are speaking about a system disintegration or crash.

Reaching the *tipping point* is one of the names for this phenomenon. Another is *regime shift*, a term coming from system theory.

When a system disintegrates, and many species go extinct, we speak about a system's collapse. This term has a negative connotation to it, however, it is important to ask the question "Negative for who?". The Yucatan meteorite has intervened massively in the existing ecosystems 65 million years ago. Yet if it wasn't for this meteorite and the following long-term climate change, the large dinosaurs wouldn't have gone extinct, and there would be a lot less diversity among mammals than we see today. Thus, experts 1 and 4 argue that a biological system never "improves" but rather just adapts to its environment. From a human perspective, this can be an improvement or debasement.

All biological systems are antifragile, yet this doesn't necessarily mean that they are sustainable. It is possible for a system to be extremely antifragile in certain aspects, but not in those where it matters.

Regime shifts can happen due to external events or internal changes in the components. If the system is still able to compensate or the shift is imminent, often depends on the gravity of the event but also on the time scale in which the system has to deal with the perturbations. Even small errors can accumulate over time and lead to a regime shift if there is not enough time left for the system to recover. It is important to note that these transitions from one state to another are not necessarily smooth, but often occur quickly and without warning. It is possible that our ecosystem is perturbed and we don't see any effects on the surface, while the system is getting close to a catastrophic shift and then from one day to the other disintegration of the system becomes visible [141, 142].

Another important notion in regards to the time scale is that systems are never in perfect equilibrium, but if they move out of equilibrium too quickly, memory is lost and predictions aren't possible anymore. The changes then are so quick, that this knowledge and memory doesn't have the time to integrate into our biological and cultural history.

Analysis, Indicators and Predictions

Complex systems, including biological ones, are difficult to fathom and visualise properly, which makes analysis and prediction even more challenging. So how can this problem be

tackled?

One key approach appears to be models. Complex systems can be modelled in mathematical models, agent-based simulations, networks, and hypergraphs. Particularly networks allow for visualisations, which is often the initial analysis step for many experts.

Complex systems have a hierarchy consisting of different layers. Their interaction, as well as topology, is important, and one idea is to visualise these layers in a network map, where each region can be zoomed into and then displays further subregions and more details. It could be analysed how strong the subregions are interconnected or how it changes for the regions above.

It is not possible to obtain a complete description of the complex system, which is why the question that should be answered plays an important role in the modelling process. To match the hierarchy level with the question that the model is supposed to answer, it needs to be determined which elements are required to be incorporated in the model. Ideally, this is done by an expert who understands the real-world system and facilitates an educated trial-error approach. If model predictions don't agree with what is observed in the real world, it is an indicator that a significant element or process was left out.

The modelling process is not linear but can be rather compared to a helix. Question and model are being revisited iteratively, each time at a different level. That's how models and theories evolve. Robustness plays a role in identifying a stop criterion, i.e. when the model is good enough. If components are varied or irrelevant elements are included, no change is expected in metrics and dynamics. If the phenomenon doesn't stay robust and randomness can be observed, it's a sign that more fine-tuning is necessary.

While it is true that the complete description of a complex system can never be obtained, we need these simplified representations to make it understandable. The risk here is mono-causal thinking like we're used to in physics, but it is wrong for biology as biological systems can exhibit alternative pathways and other complex phenomena. Working in cycles rather than linear displays might help in this regard.

System theorists, who serve as an intermediary between the expert and the real world, play a critical role as well. He or she should know enough about modelling to not be intimidated by it, but also know enough about the real world to be aware of the fact, that a model can only be adequate to answer a certain question. Models can only make predictions in a specific situation, and even then surprises are not unusual, as they rely on experiences and cannot consider events, that did not occur in the past.

This makes general predictions extremely difficult. Multiple models for various perspectives are necessary, and none of these can capture the full description of the system.

The accuracy of the predictions is also strongly dependent on the system's temporal dynamics. In geology, processes take so long, that it is not possible to give more precise predictions than in a time frame between tomorrow and the next tens of thousands of years. This is different for a small mouse, where you can predict the change of pulse in the next second exactly after giving it adrenaline for instance as was described by the

evolutionary biology and ecology expert. It might be an interesting insight to consider that processes are harder to predict the longer they take, and easier to predict the shorter they are.

Explicit indicators or metrics for systemic phenomena and impending events could not be specified by the experts. Instead, it seems that these vary depending on the context of the system. For example, critical nodes in a complex network could be identified by randomly removing nodes to examine how significant the impact would be on the whole system. Thereby, systemic risk could be assessed by quantifying the amount of risk of each node to which they expose the system. However, how the computed number should be interpreted and what it could indicate, immensely depends on the context of the network.

Similarly, it might be possible to attach a number to the complexity of a system by determining how many different actions each agent can choose from at a given time, indicating the agents' independent degrees of freedom in the system. The more actions are possible for the agents, the higher the complexity of the system. Although, the interpretation of this again depends on the context. Higher complexity does not necessarily mean more robustness.

Diagnostics or tests can check if the system's functions and processes produce the expected outcome. A typical example is medical diagnostics, such as physical exams, to investigate a human body's regular processes. If the diagnostic leads to unexpected results, it could represent a warning. However, it is extremely difficult for humans to correctly interpret premature markers. For instance, diagnostics could uncover that someone's blood sugar is fluctuating in a specific range throughout the day and at a certain point this fluctuation range increases. This could be interpreted as an indicator for diabetes, but given the many alternative pathways and multifunctions in the human body, it could also indicate cardiac insufficiency. In this regard, it is important to remember that correlation does not equal causation. Another example is given by the visibly spongy nerve tissue due to holes in the nerve cells in Alzheimer patients. While this would be a clear indicator of the limited cognitive capabilities of a patient, there are studies with nuns that displayed such nerve tissue, even though they performed fitly in the cognitive tests. This displays the brain's strong resilience capacities and the difficulties that arise when trying to interpret indicators in complex systems.

There might be something to learn from the discontinuity theory of immunity [126]. Instead of reacting to foreign substances, the system becomes alert when there is unusual behaviour. This could be something foreign, but also familiar to the system, just not in this magnitude. According to this theory, rapid changes in any direction would trigger the system. An analogy with psychological awareness can be made. Constant side noise is faded out, however, when the noise disappears the change, in contrast, is noticeable.

Another intriguing notion is based on the interconnectedness we nowadays experience in our everyday lives. Society is facing various crises. However, the more areas of society are involved, and the more crises appear to be interlinked, the more we have to consider this

as an indication of a meta crisis and the real risk of systemic collapse. Such non-linear side effects have been discussed for decades. One way to tackle this is to look at the complexity gap of the interactions between these systems. The attention should be focused to reduce some stress in those interactions that are not in harmony.

4.2 Placement in the context of Software Engineering

This chapter outlines factors and strategies that are assumed to be relevant for resilience and antifragility in large and complex software systems. They were derived from the first expert interviews with biologists and complexity scientists as described in section 4.1. The findings are summarised and translated to a software engineering context to understand how these principles could affect the dynamics and risks in software systems. Subsequently, the results are compared to the current state of the art in literature, as well as examples included of how these principles could be realised in practice. Finally, a few more words in regards to modelling and measuring the system are added.

4.2.1 Fault Tolerance

The theme of fault tolerance runs through entire chapters and makes its importance clear in regards to biological systems.

So far, the attempt has been made to adapt humans to the technical systems that we have built, although this is not very successful. Technical systems tend to work straightforwardly, requiring unambiguous processes, that do not tolerate errors. However, humans are organisms that regularly make mistakes. In nature, errors appear frequently, so it is not surprising that nature has introduced a way to deal with these. Instead of the attempt to adapt humans to our rigid technical systems, it might make sense to adapt the systems to error-making humans and nature.

Diversity and redundancy in all its forms are fault tolerance principles at heart.

Diversity

In software engineering, the notion of diversity is not new and its potential benefits for resilience have fuelled the research in this area since the 70s cite div94, div70. Various forms of diversity have been studied. Examples include managed software diversity, automated software diversity, integrated software diversity, etc. It can be exploited or introduced for many goals such as fault tolerance, security, performance, or software testing as the dependency on a few crucial resources is reduced cite diversity. Different implementations of the same function help to alleviate the consequences that could result from excessive dependencies on a certain resource, thereby increasing the system's fault tolerance.

More interestingly, expert 3 draws a parallel between the diversity of species in ecosystems and the diversity of ideas in software ecosystems.

A good example of this is Stack Overflow. If a problem is encountered during the development process, it can be tricky to find the correct solution in that specific context. However, it is not unlikely that someone else has already encountered this exact problem and there is information about it available on Stack Overflow. There might be code that has already solved the problem, been debugged, tested, and also discussed and fine-tuned among other developers. This huge knowledge base of diverse ideas is what propels development. It has been a dream for many years to be able to automatically include snippets in the code when encountering issues that might have been solved somewhere else already. Stack Overflow is the closest thing we have that incorporates this diversity of people and knowledge. At this point, the risks need to be addressed as well. Stack Overflow's success has also led to the dilemma that platform diversity is being reduced. As the majority of developers are using the platform a strong dependency is created. Additionally, it has been demonstrated that developers who only depend on the platform will produce significantly less secure code than those that additionally consider official documentation and books. Coupled with the strong dependency on the platform, ultimately diversity is reduced leading to the propagation of errors and less secure code among developers [3].

Further positive examples are open source projects. Platforms such as GitHub have enabled developers to contribute to existing projects instead of starting from scratch which enhances productivity.

Increasing collaboration and openness will also enhance diversity in software engineering and thereby increase resilience as was emphasized by expert 3. However, it is critical to remember the risks as described in the Stack Overflow example.

Below further applications of diversity in software engineering are listed, which were addressed during the final evaluation round.

- *N-version programming* describes the process of introducing diversity by creating multiple programs for the same objective independently to "reduce the probability of identical software faults occurring in two or more versions of the program" [36]. Individual outputs will then be taken into account by an algorithm that can be simple or sophisticated to decide the final output. Application areas are modern airlines in regards to flight control or automatic rail track switching systems.

- *Multiple firewalls* from different vendors help mitigating threats and risks in IoT as well as increase availability [118].

- *Plugin-based Software Architecture* enables software to keep a central range of functionalities across all versions while being open for the large open source communities. Successful examples include Eclipse, WordPress, and Firefox, all of which have adopted plugin-based software architecture and thereby significantly enhanced functional software diversity.

- *SQLrand* prevents SQL injections by introducing diversity in the SQL queries themselves. Keywords are simply prefixed with a specific token for each database. This creates an unpredictable language for external attackers [26].

- *A diversity in knowledge and ideas* has been shown to have a positive effect on innovation efficiency and teamwork during various phases of software engineering [102]. Additionally, this is necessary to understand and handle complexity in large software systems. Diverse personalities and skill sets represent strong software teams that help to solve the complex problems related to management, development, and maintenance [31].

Redundancy, Alternative Pathways and Multifunction

In the context of this work alternative pathways are the direct result of exploiting diversity in the context of software. In a network, diversity could mean the presence of wired and wireless links. If this is exploited so that the wireless link can be used if the wired link is cut, or the wired link is used should the wireless link be jammed, then for this specific function an alternative pathway was introduced [152]. Another option to increase flexibility and thereby resilience is the introduction of multifunction, which enables systems that are structurally different to still perform the same function i.e. lead to the same output [13].

Adding alternative pathways or multifunction components introduces redundancies in the system, which is often avoided in software engineering. To have code that is duplicated and repeated is considered inelegant as it reduces code reuse and makes maintenance difficult. This leads to another dilemma, as this approach also results in high dependencies on certain pieces of code. Expert 5 argues that if a critical path is identified in the software system, it is necessary to focus attention on that area since there is a risk of cascading failures rippling across the system. Robustness could be increased by introducing redundancies to support that path or offer an alternative should it not be available [91]. This might result in decreased maintainability and efficiency, however, it also helps to limit the impact of failures [4]. In this context, the term *graceful degradation* can be mentioned, which describes a fault tolerance capability in which limited functionality is retained in the event of a system breakdown.

Another consideration is made regarding the different levels in software engineering. Redundancies in the code itself are useful, however other levels should be able to compensate in case that the code crashes. Previously, the example was mentioned about cells that go senescent when encountering problems with themselves. Other cells can compensate until a certain point. Similarly, we could imagine such mechanisms on the different levels of software systems. For instance, redundancy could be implemented on the hardware level (e.g error correction algorithm in ECC RAM), the software level, and the organisational dimension by having more than one programmer for critical tasks, for example. Having these mechanisms at various levels increases fault tolerance.

So far, many of these principles are applied in those situations where the software system is connected to high risk. If safety and security are not considered to be critical for the application, it is common to have only one critical path and not many alternatives. The reason for this is rational, as having only one path that everyone uses increases maintainability and efficiency, while redundancies increase complexity. Ultimately, we need to remember the trade-off between redundancy and efficiency, continuously reflecting from which point on it pays off to introduce redundancies.

The points listed below serve as practical examples of diversity in software engineering and serve as a baseline for the evaluation round with industry experts.

- *N-version programming* has been described above and is also a form of redundancy. One example is the triple redundancy and actor-judge system application in SpaceX, where each decision is computed by three independent processors and a voting system decides which decision to follow [87].
- *Safety instrumented systems* refer to implementations of functions that take action to hedge critical processes when predetermined conditions indicate risk or injury. They are commonly used in power plants of the petroleum industry, and other industries [183].
- *Data backups* are one of the most well-known types of deliberately introduced redundancy where identical information is duplicated to mitigate the risk of data loss.
- *Error correction code* (ECC) is a technique for detecting and correcting errors in data by adding redundant information. Some applications include photographs from spacecraft or random access memory (RAM). It is used in critical systems where no data corruption can be tolerated [85].
- *Designing multiple response options for breakdown scenarios* is one way to control degradation. For example, less important or high-cost features, functions, and services could be disabled to save resources [180].
- *Personal Diversification* describes a form of redundancy intending to prevent the loss of complete person groups. An example would be separately travelling board members or lead developers [73].

4.2.2 Modularity

To be robust, the system should be modular. This means its components should have well-defined borders, and be able to work autonomously, while being integrated with the rest of the system, in a way that larger systems can be built.

When designing and developing large programs and complex systems, the code should be modularised in different parts. A bug causing failures in one part shouldn't propagate to other parts of the system.

However, there is also a need to integrate the system. We want it to be coherent and its modules to exchange information since complexity and its emergent properties arise from the interaction. If the systems' components don't communicate, they are isolated. So there is no interaction and the system is not functioning.

Expert 3 discussed the tension between integration and modularity, which is causing issues in software engineering because it is not yet fully understood how to decompose a system so that after reassembling the components, functions and properties, that did not exist in its single decompositions, can emerge again. In other words, we still haven't found a way to design a system that is well-integrated by modular at the same time. Instead, we rely on best practices that have emerged in a cultural evolution, which is detailed further in 4.2.3.

This relationship between modularity and evolution has been discussed in 4.1 and is mainly related to the principle of self-organisation. In the context of software systems, it could mean that the complex structures observed in today's computer networks evolved historically in a self-organising manner to stay resilient, provided there was pressure for it. In other words, the structures were formed in response to perturbations. In software engineering examples could be malicious attacks, changing user requirements, or other effects on the system's normal state. In the past two decades, the interest in self-organisation in the form of self-adaptive systems has resulted in extensive knowledge and promising approaches to the challenges of uncertainty in complex software systems [178, 177]. Future research could investigate the effectiveness of these approaches, and how they can be improved.

As software systems grow and evolve, modularity deteriorates over time [145]. Therefore, it is crucial to focus the development team on maintaining and improving modularity. Aside from technical implementations, however, the organisational design of the software team also needs to be adjusted regularly. For example, *agile software development* is a form of modular organisation. Traditional agile methods such as SCRUM are focused on the team level, which consists of a manageable number of developers. However, as the organisation grows, the number of developers also implicitly increases. Assigning these to individual agile teams results in dependencies and leads to a deterioration of modularity once again. Tackling this requires continuous adjustments of the organisational structure. One approach is the *SAFe* Framework, which helps to design the coordination without losing agility and enabling strategic management [100]. Illustration **??** shows the exploding complexity of the interactions of hundreds of developers that work on large and integrated solutions.

Finally, some practical examples of modularity in software systems are listed, which also serve as a discussion baseline in the secondary evaluation round with industry experts.

- *Low coupling, high cohesion* is a programming approach that carries the concept of modularity at its core. The code should be divided into modules that are as independent as possible and that do not influence each other when changes are

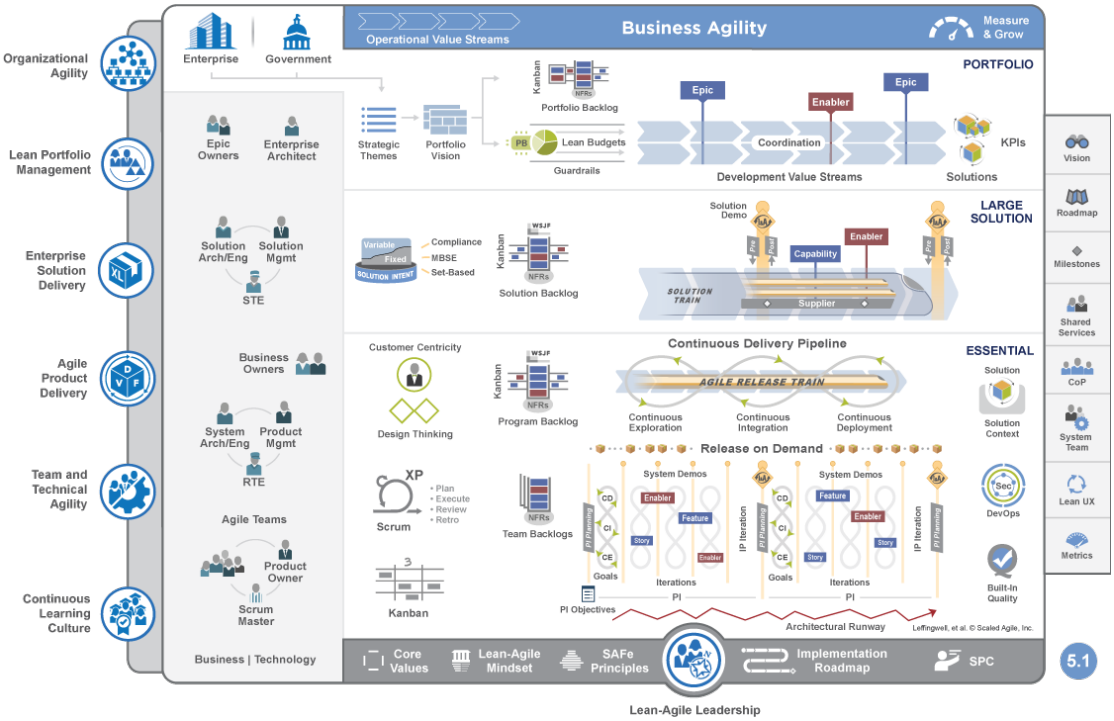

Figure 4.2: Most comprehensive configuration of the SAFe framework [75]

made (low coupling). At the same time, it should be ensured that the functions within a module are related, i.e. follow the same line (high cohesion) [184].

- *Libraries or packages* as well-defined modules or dynamic linking, where each module is stored as its own file like in Java, enable code reuse and are a canonical example of modularity [95]. If the system should be maintainable for a long period, Winters proposes that libraries should be self-written instead of imported [180]. This is an important insight that is further discussed in section 5.1.

- *Modular Organisation* refers to the use of modularity in a managerial way. The notion focuses on "flexible learning organisations that continuously change and solve problems through interconnected self-organising processes" [56]. Characteristics include flat hierarchies, decentralised planning and decision-making, as well as empowerment of employees. A prominent development method in this regard is agile software development as mentioned above.

- *Development of Open Source Software Systems* follow certain patterns in self-organisation to adapt to newly emerging requirements. Large scale software engineering is a social activity just as much as it is technical, and this social self-organisation can be observed in the assignment of tasks, prioritization, and

scheduling of work [40]. One example is the self-organisation process in the evolution of Linux, which is a leading open-source software project with no central hierarchy for planning and management [185].

4.2.3 Adaptivity

As evolution and selection stress appear to play a crucial in the context of biological systems, it is reasonable to discuss their interpretation in a software engineering context.

In general, we want software systems to adapt to their environment in a way that they can deal with perturbations without breaking. According to expert 7, two qualities are required in this sense:

1. Robustness, which means to be able to survive the perturbation or event
2. Resilience, which means to be prepared for adaption to the new landscape after the event

That's one of the things that separates robustness from resilience. Robustness means to survive the shock and then go back to the same dynamics as before. Resilience means to adapt or at least be ready to do so. In this work, the difference between robustness and resilience is further distinguished in that robust systems typically withstand known disruptions, while resilient systems can also adapt to previously unknown perturbations.

Systems that don't adapt will not be selected and eventually disappear. Therefore, adaption to perturbations over time plays a crucial role in the persistence of these systems.

Self-organisation was identified as an important feature of biological systems. They possibly display certain structures to stay resilient and survive as described in 4.1. Something similar might be achieved in software engineering by letting the system undergo an evolutionary process similar to those known from biology. In keeping the objective function fixed while varying the conditions, the system might adapt to withstand new perturbations in a similar way since redundancies and other resilience mechanisms have been introduced in this evolutionary process.

In this scenario, the system adapts because it is actively provoked and thereby the developers learn, how the system reacts to failure. The gained knowledge can be used to implement appropriate mechanisms and adapt the system accordingly. This exact approach can be seen in the Chaos Monkey resilience tool, which purposefully disables some components on the live environment of a software system to analyse the effects and learn from them [76]. Another approach can be found in the DevOps culture, which advocates for shorter and faster deployment cycles, even when there is a risk to encounter issues. These regular deployments can be seen as perturbations to the normal state of the system. Feedback is received more quickly, and the system can be adapted faster [137].

It's important to note at this point that not all perturbations help the system to improve, which is why the quality of the challenges is of utmost importance. The systems should be in a flow state as described in section 4.1. This keeps them from stabilising and becoming fragile while fostering learning and adaptation.

Another evolutionary aspect can be found in the social dimension. In 4.1 it was described how evolution in a social context results in best practices. While there is no formal way for some implementations, there are accepted best practices. Collectives of people have decided what works best and arrived at a consensus through cultural evolution. This social dimension plays an incredibly critical role in software engineering as stressed by expert 3.

There are many advantages of such a consensus, especially if a particular framework is being selected. If it is highly used, we prefer people to continue using it, as the advantages are clear: more and better support, easier to hire new developers and to maintain the system.

However, this also poses a risk since it creates a high dependency on that one framework, which then becomes critical, and leads to an overall decrease in diversity.

This is countered again in DevOps culture where there is this notion that a product is never finished, and therefore always needs adapting. Kevin Kelly writes:

"*Existence, it seems, is chiefly maintenance [83].*"

There will be dependencies, that need to be updated, maintenance to be done, and change requests to adapt the system to customer needs. Innovations can be seen as mutations in the evolution process. Not all of them will succeed, but those that are selected, become critical.

The biggest takeaway here is that it is necessary to change the way that we think about software projects and systems. The belief that these projects have a clear start and endpoint, as well as well-defined scope and budget, needs to be discarded. Expert 1 argues that the term "project" even encourages managers not to recognise potential risks and threats or rather not to take them into account at all. Project management courses teach that a software project has clear start- and endpoints, and a defined budget, which means that managers tend to pull their projects through regardless of the overall condition of the system and possible side effects. Instead, we should try to look at these systems more naturally and organically, particularly concerning their dynamics, i.e behaviour over time. In this notion, software resembles a biological system such as a garden more than a technical system such as a building, which has been mentioned in chapter 1.1. It needs to be maintained continuously all the while considering possible side effects of interconnected systems today and in the future. Many developers write code without these considerations often only focusing on the speed and elegance of the code. They can be proud and professional programmers that autonomously educate themselves on new

technologies and tools, continuously improving and writing responsible code. All these are necessary for developers to be good craftsman [99]. However, as Papapetrou states, programming is more than a craftsmanship:

> "*You can't add a new level between the 23rd and the 24th floor of a skyscraper unless engineering has evolved a lot the last few years. Last time I checked it was impossible. You can't just replace a pair of old or broken bridge pillars. You need to replace the whole bridge. But in software, you can add a new layer or replace a module using old technology with a modern one. Because software is like gardening. By treating software as something organic, we expect change. We learn how to care for it so that when that change comes, we're ready for it [123].*"

This expectation towards change translates to the importance of attitude towards developed code, teamwork, and mentoring young developers.

Find below some examples of possible evolutionary principles in software engineering, that befit the software gardening notion:

- *DevOps culture* is a cultural shift more than it is about tools or technologies. In this culture there is the notion that a software product is never finished as maintenance, dependency updates, change requests to adapt the system to user's needs, etc. will always be necessary. They advocate for shorter deployment cycles which increase the speed of feedback loops. This eases the process of making, and also reverting changes, thereby accelerating the adaption of the systems to their environment [137]. Often it is difficult to introduce this culture successfully since it is mostly a management problem that requires focus on the change of processes, habits, and incentives [1].
- *Chaos Monkey* is a resilience tool whose job is to randomly disable components and services on the live system. It helps to ensure that the individual modules work autonomously even if other connected modules experience failures, thereby increasing independence in the system [76].
- *Frequent Rewrites* is a software engineering practice from Google where a parallel can be made to biological systems that need regular perturbations to adapt to the environment. Google rewrites most of its software every few years which leads to leaner software by removing all complexity that is no longer relevant, and serves as a great knowledge transfer to newer team members. Additionally, these gain a feeling of ownership which in turn increases their motivation to work [66].

4.2.4 Analysis and Metrics

Many metrics and warning indicators that exist for software systems are largely quantitative measures, such as the number of code lines and bugs, or performance measures such

as CPU usage or execution time. Analysis based on these metrics is often insufficient to describe the state of a complex software system or to detect impending systemic failures such as tipping points.

An interesting idea concerning systemic crashes is the principle of requisite variety or complexity mismatch, which has been described in section 4.1. Complex software systems include various networks, such as the collaboration and contribution network, but also the involved package/dependency network, which is not always directly visible, and one of the things that makes software systems complex. The system is in interaction with all these networks, and the principle of requisite variety could be considered in this regard. If the system's complexity is measured in its agents' degrees of freedom, then they need to have at least as many, as the agents of the systems that they are interacting with. In a software context, this could mean that more inputs (more sensors, additional ways to interact with the environment) or more dependencies necessarily lead to an increase of the software system's complexity to deal with those inputs and dependencies.

However, there is no general rule that increasing complexity in software systems due to an increase of complexity in the environment will lead to more harmony. For instance, if a system has an exposed interface, that is complex and offers lots of functionality, and the code behind is not deep enough to handle this, then the growing tension might lead to a crash as the complexity is not distributing properly. The solution is not necessarily to increase complexity in the code behind. In general, complex systems consist of various levels and each level implements some mechanisms for resilience or robustness and is, therefore, able to compensate for the failure of other levels until a certain point.

What we want is to increase complexity at levels where it makes sense. This would improve resilience, however, it is also necessary to keep a certain balance. Having too complex structures makes the system exceedingly difficult to understand, maintain, and therefore decreases resilience as described by expert 5.

Based on the analysis approaches in biology, networks or hypergraphs could be considered for the analysis of complex software systems [168]. One concrete idea is to develop a network of the system and then manipulate it. For instance, critical modules or agents in the network could be identified by removing random nodes or causing them to fail. This would help to determine the level of propagation and the ramifications of this failure. Another idea introduced in the interview with expert 5 is to remove dependencies or other external data sources to examine the impact on the system's test suite. This analysis might show which functions are critical, regardless of the number of their occurrences. It is possible to have a function occur maybe just once, but its failure would ripple across the whole system.

The tongue in cheek metric *bus (truck) factor* was also mentioned in this context. It determines the minimal number of developers that must be eliminated (run over by a truck or bus) before the project is completely lost. Having a truck factor of one means that most of the project's critical knowledge is gathered by one person. Some articles are proposing methods to quantify this, however, it is usually at the level of one project,

which is neglecting the complex network of collaborations between projects, as functions, dependencies, and libraries are imported. A meaningful bus factor should consider all of the network's agents and dynamics.

Another interesting approach is the shift from quantitative metrics such as budget, effort, and deadlines of software projects to qualitative measures, that evaluate the whole process and include social values such as the well-being of the developers, sustainability, etc. From these social values, further quantitative measures can be derived. A scientific approach to build such metrics has been proposed by Forsgren et. al [53]. It is trying to correlate the cognitive level of people with the team's performance, and there is a need to do this from their perspective.

A more practical idea was discovered based on the discontinuity theory of immunity. The implementation of a control mechanism that gauges if individual system variables are changing rapidly and far beyond the common frame, could help in the early detection of issues that require intervention. The development of the concrete metrics is specific to the context.

Finally, warning signals before approaching tipping points can be considered. These are particularly interesting as they appear to be a universal rule applicable to all complex systems rather than depend on the specific details of the underlying agents. It has been discovered that the system's response to perturbations slows down when approaching a tipping point due to gradually changing conditions. This phenomenon is called *critical slowing* and has been described by Sugihara [153]. The effects can be seen in the form of alternation between two stable states in temperature, population numbers, or a lake (clean vs. algae-ridden). Another example where this behaviour was observed are various climate shifts as described by Vasilis et al [42]. Sugihara mentions another early indicator which is called *spatial resonance*, which describes the phenomenon of coinciding impulses of adjacent systems in the network. He mentions the example of a global financial meltdown where institutions displayed homogenous revenue-generating and risk-management strategies when the crash was imminent [153]. These metrics are essential for survival as they can indicate an imminent potentially catastrophic event. Future research should focus on discovering additional warning indicators in this regard.

For the sake of completeness, the area of complexity metrics needs to be mentioned as well. One approach could be the measurement of structural complexity with the metrics of structure entropy, evaluation coefficients, and the weight ratio, which are described in section 2.1.2.

To determine how much effort should be put into analysing and measuring the system's state, an analogy can be drawn to biology with viruses. As mentioned in section 4.1 commonly there are not many mechanisms built into viruses that control their replication process, which results in a high mutation rate. It was explained why this is less critical for viruses than for complex organisms. In software testing, there are approaches where a high mutation rate, which could lead to errors, is desired. For instance, in fuzzy testing, an automated software testing approach, various input parameters are input to the system

to analyse how the behaviour changes. In this case, the high mutation rate (various inputs) are not an issue as the goal is to be pointed towards the critical side effects that haven't been dealt with yet. Testing highly increases resilience and inspires confidence in the system as it evolves. However, such behaviour might not be desired for highly critical and complex functions. Therefore, it is necessary to consider the fitness advantage when performing analysis and tests and keep the effort in an appropriate frame.

It can be concluded, that the dynamics in complex systems make it extremely difficult if not impossible to predict extreme events. The models reach their limits when one takes into account that they cannot predict extreme events and risks if they have not yet occurred in the past. A landmark study commenced and evaluated by Tetlock indicates that even expert predictions are on average hardly more accurate than random guessing by an amateur [160]. Predicting crises is therefore not a suitable tool for politics and management. However, we can build in protection and anticipate them by paying attention to those interactions with a high complexity mismatch. Nassim Taleb mentions existing methods and procedures to anticipate and react to such events. One example is the non-naive cautionary principle [155]. Taleb argues that we have to be particularly careful if a problem is potentially ruinous.

Additionally, we can use certain metrics and models to analyse the behaviour of a system and better understand the risks.

- *Metrics derived from social values* such as generative organisations should be in the focus as they influence software delivery performance which in turn influences the whole organisational performance [53]. Examples include measuring employee's satisfaction, how much they would recommend the workplace to someone else, if they consider their environment appropriate to do their work, or if they reckon to have options to use their skills well at work [167]. Most of these were researched by Westrum and core values in DevOps culture[176].
- *Truck/Bus factor* metrics can refer to the social dimension (removing developers or managers), but also incorporate the removal of technical artefacts, such as libraries, data sources and other dependencies [103]. A similar approach is given by Google, which calculates a criticality score that is supposed to detect security risks in open source projects [2, 115].
- *Discontinuity theory of immunity* could be applied to software systems so that the system reacts to unexpected and unusual provocations while tolerating slow and steady stimulation [126]. This would result in warning indicators for potentially threatening behaviour.
- *Fuzz Testing* is an automated software testing approach that checks the system's reactions to random input parameters that can be valid and invalid. Nowadays, it is an integral part of Chaos Engineering and is regularly performed by large software companies such as Netflix or Google. It is (partly-)integrated with various tools such as the continuous delivery platform Spinnaker [4].

- *Risk Assessment of Cascading Failures* is critical for complex software systems to examine how complexity is distributed among the agents (developers, libraries...). A network is designed and the removal of nodes is simulated to identify the critical nodes by observing cascading effects. This information can then be used to focus attention on these nodes by adding support, for instance [166, 174].

CHAPTER 5

Discussion

This chapter summarises the insights gained from the secondary interview round with industry experts which also serves as an evaluation of the findings. Additionally, the results' limitations are discussed. On the one side, software systems are not equal to biological ones, therefore it is necessary to outline where the differences lie. On the other side, even the most sustainable and resilient principles have difficulties succeeding in a system that is troubled by a harmful environment. Therefore, the contextual system challenges are addressed in the final step as well.

5.1 Evaluation

In the Methodology chapter 3 it was stated that the derived principles and criteria for sustainable software engineering will be evaluated in a final interview round with experts from the industry. A total of five consultants/experts was selected with core competencies in Management, Information Technology, Information Security, and Crisis Prevention.

This section discusses direct expert feedback for the derived principles addressed in chapter 4.2. Experts' interpretation of advantages, disadvantages, acceptance, and limitations of these principles in practice are outlined, as well as, investigated how basic conditions should change to promote the application of these principles. There were contradicting opinions in some areas, however, most issues converged to organisational and social aspects.

In the following, the results of this evaluation are presented.

5.1.1 Fault Tolerance

All experts stressed the trade-off with efficiency. Even though fault tolerance principles increase resilience and robustness, considerations about fitness benefits and pay-off appear

to play a great role in the industry. Three of two experts stressed that fault tolerance is mainly necessary where the product is associated with high risk or has relevant security aspects, such as software in critical infrastructure. Aviation control software has different requirements than the resource planning system of a local baker. The other experts emphasised the threats given high dependencies between software systems, as the product risk is often overseen nowadays and the mechanisms not introduced, even when the fitness benefit could be significant.

For example, experts B and D underlined the technical and organisational challenges that are encountered as diversity is introduced. It can be summarised, that currently there is not much incentive to foster diversity in most sectors. The trade-off with maintenance and efficiency is so high that many view it as unmanageable in a technical and organisational view.

Additionally, there is a clear trend towards monocultures which depicts a direct opposition to diversity. Commonly, it exists when new software is introduced and various providers can be found offering different implementations of the same software, fostering diversity. However, this quickly converges to a few providers that end up controlling most of the market share. This is a development not exclusively reserved for the software industry. It can be found in other industries as well, which leads to the suspicion that the cause for this lies in the incentives set by the economic environment. For example, managers tend to use tools that represent the industry standard to have a better line of argumentation in case of an incorrect decision. This view was shared by all experts.

In other cases, diversity cannot be introduced as it doesn't exist. Expert B states that hospitals, for instance, are forced to rely on the few software providers there are, regardless of their quality.

Regarding organisational diversity in the form of diverse ideas and knowledge, issues often lie in the know-how distribution. Expert A and E explain that communication in companies of 10-15 people is possible on a personal level. Therefore, diversity is facilitated since experts talk to generalists and exchange ideas. With an increasing number of people, certain structures are required to enable that kind of communication. Intermediaries are needed. These usually come from the field and have much experience, which is necessary to develop a big picture. However, they are rare and tend to hinder innovation due to their long-lasting experience, which was mentioned by expert C as well.

Another cause for insufficient know-how distribution in large companies is the lack of software competence (in development and experience) of employees in higher positions. These are often unaware of the consequences that their decisions can have. Expert A argues that problematic incentives lead to a higher priority in company politics rather than expertise. Smaller companies, on the other hand, often struggle with the issue that founders have difficulties enabling structures and hierarchies to pass on responsibility to employees, albeit having a high professional competence.

Expert E distinguishes between the idea finding phase, where diversity should be fostered, and the execution phase, where it is objectionable. However, they also mention that even

in this phase, the efficiency pressure is so high, that any ideas without short-term output are disregarded, regardless of their long-term benefits.

Similar issues occur with the idea of redundancy. Expert C mentions that the introduction of alternative ways or reserving resources for backups contradict the notion of efficiency. All experts agree that management is often limited to a specific period in a company, which in turn leads to short-term thinking.

Expert B brings another counterargument for redundancy which is its weakness to common cause failures. Having multiple resources of the same implementation, for example, two same hard drives do not increase fault tolerance, as the same errors and bugs affect both hard drives. Therefore, the interplay between diversity and redundancy is crucial.

5.1.2 Modularity

Initially, interconnectedness renders the system more stable. This can be seen in families or communities that support and help each other out by being connected. However, as the number of agents grows in the network, stability starts to dwindle and unwanted side effects can occur as described by expert C. This risk can be tackled with a modular approach, which is characterised by decentralised organisation, borders and hierarchical nesting, i.e modules can be decomposed into more granular modules.

Frozen accidents, a notion coined by Murray Gell-Mann, describe the phenomenon of an accident with far-reaching ramifications that can be traced back to one random event [28]. Expert A mentions the width of railway tracks as an example. The initial choice for the width was more or less random, however, after installing thousands of kilometres of railway tracks, the choice cannot be reverted anymore and society is stuck with this technology. The development can be seen in software systems as well regarding architectural decisions or protocol definitions. Expert A describes examples such as the introduction of a specific enterprise system (e.g SAP) that cannot be reverted afterwards or only with enormous effort. Many programmes are written on top of old legacy systems or technology.

While it is extremely difficult to tackle this issue, modularity can at least help in this regard as it eases the exchange of components in a system. Particularly critical infrastructure would benefit greatly from modular units to ease the process of adaption as explained by expert C.

Standards, which would enable such an exchange of modules, are often insufficient or non-existing as mentioned by experts A and D. Providers are not interested in conforming to standards as they don't have any direct benefits from the exchangeability of their software. Expert D argues that exceptions only occur when there is high customer pressure. Developers and users on the other hand, often don't abide by standards as these can hinder innovations. Overall, software is not a commodity, that is simple to exchange once in place as even default software is configured specifically to the company and highly connected with other systems. This was stressed by experts A and B.

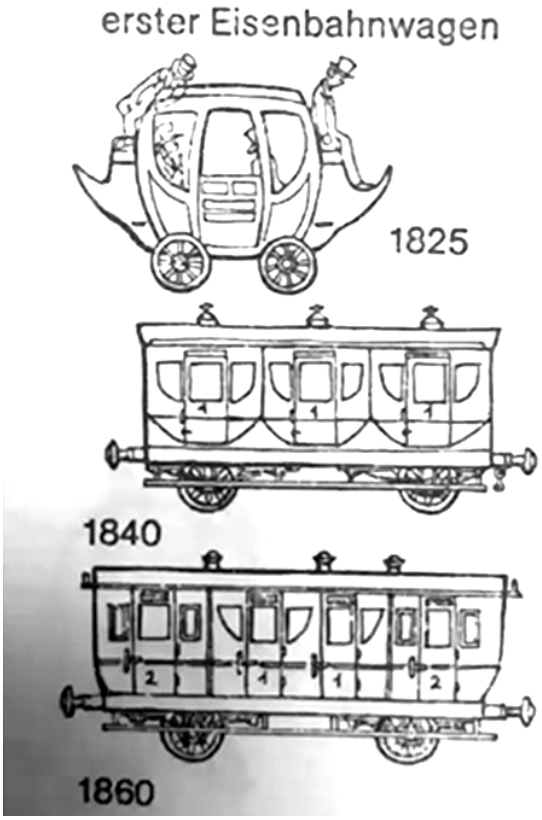

Figure 5.1: Evolution of the railway carriage [132]

In this context, another issue was described by the same experts. Not only is it technologically challenging to replace or exchange established components, but psychologically as well. The illustration in **??** portrays how our thinking is shaped considerably by what we already know and are familiar with. This thinking also shapes the novel, often in a superfluous manner because we cannot imagine the novel in any other way. The first railroad car ever designed looked a lot like a horse carriage. Fifteen years later the design looked like multiple horse carriages next to each other. This portrays how hard it is for humans to drop what is familiar. Expert A mentions technological examples such as developers that come from a certain tradition and then switch to a different paradigm, e.g. Java developers who write Python code, but also the effect of repeating the same mistakes even when building new systems. Expert A and expert 2 both state that "we just don't learn". A fitting addition is given by Edgerton, who claims that:

"*It is becoming ever more obvious that the characteristic feature of our age has not been innovation but imitation. Never before in world history have so many shared exactly the same things in such quantities, nor used them in exactly the same way [45].*"

Expert B mentions the phenomenon of *software bloat* in this context. It describes the undeniable growth of software in size disproportional to its functionality. For example, in only three years apps on Android phones have doubled in size without apparent reason. No additional functionality that could justify this growth was added. Instead, it appears that there is a trend to include unnecessary dependencies or packages for simple problems, without considering that these include further dependencies, jeopardising the metastability of the system [128]. Prokopov emphasises the necessity to periodically re-evaluate existing software, rewrite it and remove unnecessary dependencies, an idea that is further described in section 5.1.3. One reason for this development could be this inability to break away from the familiar, as expert B explains. Another is the anticipation that the application will scale and needs to be prepared for that. Further issues related to scaling are discussed in section 5.2.2.

However, one issue is obvious and that is efficiency, a notion that was stressed by experts D and E. It is crucial to consider the fitness benefit and efficiency trade-off same as with diversity and redundancy. This was illustrated by experts A and B using the example of microservice architecture. While it increases modularity and thereby introduces many other operational benefits, there is also an increase in complexity rendering the development of even simple features and maintenance of the system more difficult. Uber describes in one of their engineering blogs how the adoption of microservice architecture can greatly enhance operational benefits, but also warns that it highly depends on organisation size. They compare the implementation to gardening with "hedge trimming" at the right moment to give nudges in the right direction. This is an iterative and dynamic process, rather than a "top-down or one-time architecture (or re-architecture) effort" and requires redundancies such as dedicated engineering resources [58]. It becomes apparent that this undertaking is costly and greatly increases complexity, therefore the fitness benefit always needs to be considered.

Modularity is often put on par with decentralised organisation, and while this has undoubtedly certain benefits, expert A stresses the consideration of borders and communication between the modules. A counterexample is given by the rise of blockchain technology in power trading systems [94]. While a distributed approach is more flexible and addresses security and privacy concerns, it could also lead to great vulnerabilities as complexity increases beyond our cognitive capacities and the same technological implementation is used over large parts reducing diversity [5].

Decentralised modularity can also be found in the organisational aspects, for instance in the form of agile development. Expert B argues that some applications are not suitable for an agile process such as the development of monitoring tools, authentication solutions, or other security components. However, the reluctance among experts is not limited to these areas. There appears to be a lack of benefit recognition and hesitation to undergo long-term cultural change.

Expert A explains that managers commonly have difficulties to relay responsibilities or trust the team with decisions, while expert D argues that they often don't see direct benefits for themselves. Large companies usually want to map all processes in a formal

system, which contradicts the notion of self-organised and decentralised organisation, while smaller companies are often under too high fitness stress to undergo cultural change. Another issue lies within the employees themselves. Self-organisation requires a certain level of competence, experience, commitment to the company (as opposed to job hoppers) and maturity which cannot be found among all employees. Expert A argues that one of the reasons for this is the current circumstance, where the demand for IT personnel significantly outgrows the supply. Expert B adds that the lack of gatekeeping for the software industry (formal training and degrees are non-essential) worsens the situation. [125] Motivation is another factor. The development of open-source software is commonly performed in a decentralised manner and results in the successful development of functional, safe, and performant software. However, this intrinsic motivation is not necessarily given among employees in conventional companies, where the motivation is often extrinsic, i.e. money, short travel time, etc.

Finally, it can be summarised that on the one hand, the technical implementation of modularity is complicated in practice and a constant fight against entropy, so to speak. On the other hand, the advantages of organisational modularity are often not recognised, or cannot be implemented due to economic pressure. Expert C stresses that the development of large and complex software systems requires both, decentralised organisation as well as a top-down component, which defines an overall goal. In the technical dimension, modularity should combine decentralised organisation with separate functional units, that have well-defined borders and interfaces for restricted communication to prevent manipulation and cascading errors. Damping elements at these interfaces to slow down the processes can also increase resilience. They stress that endurance of this ambiguity is a critical skill often lacking among management and employees.

5.1.3 Adaptivity

Since complex software systems cannot adapt by themselves (yet), humans need to be kept in the loop to keep these systems antifragile. The idea of purposeful perturbations in the form of frequent rewrites or high deployment frequency has been introduced and discussed with industry experts. The evaluation has shown that there are various arguments against this approach.

On the one hand, rejection of continuous adaptions and frequent rewrites can often be traced back to economic aspects such as short-term thinking and fitness pressure. Expert A and D agree that small companies need to stay competitive and since the market appears to be not concerned with quality as much as with short-term output, wild promises are made and cheap software developed with too little regard to quality losses. The procurement and legal departments follow this trend as they work with metrics solely aiming to maximise their financial gains. In doing so, their software suppliers are put under such pressure that the short-term output is lucrative, while the quality suffers greatly in the middle- and long-term thereby eliminating all gains. Expert B and D stress that each new release brings costs in the form of tests or training for adaptions, that appear unnecessary on the surface. Long-term thinking is undervalued due to uncertainty

in the future (e.g. rapidly changing technologies, changes in customer expectation or context...), as well as the stress of competition and profit maximisation.

On the other hand, they also argue that software with high product risk and security focus (e.g. pacemakers, flight communication, control software for nuclear power plants...) is usually developed once, then certified and accepted by the purchaser. Continuous updates are undesirable or unrealistic even in the face of severe security or safety threats. One example is posed by 500.000 recalled pacemakers of various providers that displayed deadly and highly critical vulnerabilities in an independent security evaluation [187].

In the past, this issue was usually curtailed as the software was not highly connected and interdependent as we observe it today.

It is not unusual to have different development speeds depending on the kind of product that is developed. The first phase of software engineering is often characterised by a high speed of adaptions. Expert E states that this can be observed in start-ups that develop their MVP (minimum viable product) quickly to save resources. High deployment frequencies and a permanent feedback loop with the customer is typical for this step.

If it is successful, it will eventually turn from innovation into critical infrastructure, at which point the development speed decreases drastically. Tasks such as tests, validations and maintenance should become the primary focus, while adaptions happen slowly or not at all. This increases the software's stability and resilience.

Expert B explains that in the past, established software used to be the fittest as it was proven and tested. Due to fast technological advancements and the growing connectedness, there are now new risks and dangers entailed in regards to established software systems [121]. This development eventually requires a rewrite of the system, however, it is not yet clear how to deal with these issues. Expert E brings the example of smart metering[1], which exhibits such high dependencies that adaptions and even tests are hardly possible without risking a system failure. They argue that it has become increasingly difficult if not impossible to adapt and test software in critical infrastructure and that this is introducing many vulnerabilities and generally poses a great risk in terms of security and reliability. However, it is not likely that this will change in existing software argues expert C. The scenario of a crash is more probable, therefore we need to concentrate on future development and include "seeds" from which the software can be rebuilt after a systemic crash. Future development of critical software should be focused on security by design and incorporate crashes in a way that they don't represent errors but make place for adaptions. Modularity plays a big part in this regard.

Expert B adds another reason for insufficient security in many applications. Usability is visible to many stakeholders, whereas security and reliability are not. This means that software is usually designed as most usable by default and not most secure. Configuring standard software for security can be tedious and overwhelming for most users. On the

[1]Smart metering refers to electronic devices that measure, determine and control energy consumption and supply.

other hand, regular updates tend to irritate older users as they are seen as product recalls of the previous version, admitting it contained errors. Users that grew up in the digital age are more tolerant in this regard since they are more familiar with patches as has been explained by expert E.

Additionally, experts A, C, and E agree that a shift from linear rationality to Taleb's non-naive cautionary principle is required in our thinking. We tend to plan and predict the future based on our past experiences. The expectation that what has happened in the past, will continue to happen in the future can lead to a significantly distorted picture of security. This is also called *security paradox*, which describes the effect that a lack of vulnerabilities and attacks on the system leads to the belief that security measures are redundant and in turn further decrease security. Such developments result in potentially ruinous events. Therefore, focus on the unexpected and the random must be fostered by building systems in a manner, that such catastrophic consequences cannot occur by design instead of trying to prevent them. Taleb writes:

> "*This idea that in order to make a decision you need to focus on the consequences (which you can know) rather than the probability (which you can't know) is the central idea of uncertainty [158].*"

However, expert E explains that it is hard to set appropriate regulations in place as they are usually based on the past and hinder innovations since they are characterised by difficult and slow changes.

Ultimately, these necessary changes in our approach to software engineering will not happen easily, as long as managers and stakeholders are forced to short-term thinking. Some issues have a time frame that exceeds the five year period of management, therefore longer company lifetime is necessary to set the incentive for management to tackle these issues in a sustainable long-term manner.

Expert E proposes the introduction of software sustainability as a business area in the form of credible auditing schemes that allow assessing the sustainability of existing software systems. They argue that if an agreement over objective criteria and indicators for sustainability and resilience in software can be reached and represented in a comparison with other software, then it could become increasingly hard for management to ignore these metrics. Expert A adds that it would be especially difficult for them if a regulatory body expresses the requirement, but even if this is not the case, significant pressure might arise if such audits would be used for due diligence purposes in case of mergers, buyouts, initial public offerings or for evaluating the quality of a software company. Expert E mentions investors overtaking a software company and wanting to check product quality, people in the field of compliance, or auditors as possible customers.

5.2 Limitations

During all interview rounds, as well as, literature review, it became clear that not all principles can be applied completely. On the one side, there are some significant differences between biological and software systems, that need to be taken into account when adopting the derived principles. On the other side, the environmental and contextual aspects of the system play a big role in the development and prosperity of all systems. Even the most resilient and sustainable system cannot prosper in a toxic environment, which is why it is necessary to discuss the posed challenges.

The second chapter

All expert interviews have spiralled back to social issues, indicating that ultimately even our technical problems are of social matter.

5.2.1 Differences between Biological and Software Systems

There was a drastic change in the dynamics of software systems historically. Computers used to be isolated, and information exchange was much more limited. Nowadays, we have the means to potentially send large amounts of information across very long distances reaching millions of people. However, the information is not only meant for people anymore. Other computers also process this information, which becomes particularly apparent in economic transactions, that are performed automatically without any human interaction. This is one of the similarities between information technology and biology as the system is co-evolving not just with humans, but with itself. Information is produced by computers and while some of this information is processed by humans, some of it is used by the same computers that produced it, further influencing the output in a feedback loop. Expert 3 argues that this makes software systems much more similar to organisms than any previous technology.

However, there are still significant differences, one of them being the complete lack of intrinsic motivation or will in software as of yet. Humans are deciding the software's will, while living organisms, even bacteria, can decide for themselves for example if they want to move in a certain direction to eat. An organism indeed reacts to its environment, however, the behaviour cannot be 100% explained by the physics of the environment, since the organism has an inner life deciding the future development for itself. It reacts to the environment, but it has also certain degrees of freedom to at least influence the future direction. Expert 6 argues that the concept of *agency* plays a big role here, as it can provoke actions from inside, while software components do not exhibit these properties. Therefore, they argue that an algorithm can never lead to a complex system, regardless of how complicated and sophisticated it is. However, this is a much-debated topic and it is not yet clear how complexity emerges. Future research could further examine existing complex systems, and analyse if or how agency and the concept of free will are relevant.

In this thesis humans are considered as a part of the software system, therefore we can argue that they have the capability to self-repair by humans. In the traditional

definition of software, these systems are unable to self-repair. Expert 3 mentions some recent developments in the field of artificial life, where people have been developing self-autonomous robots, basically programming organisms. However, this specific action of sustaining oneself is not possible yet, and humans are needed to perform maintenance and repairs. To make the comparison between software ad biology, it is necessary to keep humans in the loop. Thus, the aggregation of computers plus humans is what forms a complex system, that is capable to evolve and repair. This is also the reason why software management aspects have to consider software, hardware, and humans (technicians as well as other stakeholders). However, this view is not uncontroversial. There is a discussion about complexity emerging due to the high degree of networking between the agents, even if these are purely technical. This is related to the continuative research idea mentioned above.

In regards to evolution, two limitations need to be kept in mind when making the parallel. Firstly, modularity is lower in software systems than in nature, meaning that there is a much higher degree of dependency among software systems than among biological systems. Specific individuals can have a global effect in software systems, where most developments in biology are restricted to local changes. This is related to the fact that mutations in a species in an ecosystem take time and symmetrically distribute among the individuals of the species. For example, a single tiger will never become 1000 times more powerful than all other tigers or all other gazelles, jeopardising the metastability of the system. This asymmetry of power is exactly what we are observing in software systems where single players grow overwhelmingly [144]. Secondly, evolution needs time. The software industry is still rather new and we cannot make the statement that the oldest software systems that are around are the most adapted, i.e. the fittest. While newly developed software includes more common bugs and errors than established software, it also holds that technology is advancing. This puts established software at a risk to be outdated and leads to more complex errors and side effects as well as higher vulnerability, while the new software is at risk due to a lack of tests and experience [121]. Finally, we're referring to a distinction, which is based on the argument that software code can be executed an infinite number of times, without any change in the course. Executions of biological processes such as protein folding, however, are dealt with in a biological and physical environment, that refer to atoms at the lowest level. There are so many interactions that components always react a bit differently for each execution. However, this distinction is turning into a similarity since software process executions nowadays have started to portray a similar behaviour. Software is running on computers that are highly interconnected with other systems. These increasing interactions might cause the code execution to react a bit differently depending on factors such as bandwidth, power stability, memory, etc.

5.2.2 Contextual System Challenges

Economy and Politics

The current economic system is oriented towards short-term profit maximisation. This appears to be an issue that pulls through all systemic challenges including those in software engineering. Our agency is not limited by the technological possibilities but rather by the economic frame. Companies must maximise their profit in a short-term manner to stay competitive. All experts agreed on this.

Even technological advancements such as the open-source movement are affected by this. Expert B argues that in the past mostly individuals would contribute to open source projects, whereas nowadays it is not unusual to have hyper-scale platform vendors such as Microsoft or Google pay for these contributions and thereby influencing the development's direction in their strategic interest [12].

It is crucial to consider the notion of *short-termism* and the underlying social structures and incentives that lead to it. In business and industry, short-termism is formalised in the form of shareholder value, quarterly reports, targets, etc. The unhealthy effects of this focus such as the high volatility of stocks due to unmet targets influence the long-term health of companies including their workers. This also affects the communities around the firms, the environment, etc [171]. It also threatens the maintainability of critical infrastructure, of which IT is a significant part today. This is discussed further in 5.2.2. Another example was given by expert A, who emphasised the negative side effects of assertive software project managers who are bent to meet their targets regardless of how the rest of the firm is performing. In this context, the idea of software gardening was discussed. It describes the notion that due to the high interconnectedness, complexity and side effects in software systems, development and maintenance should be performed like in a garden rather than a technical system as described in section 4.2.3. Such an organic approach would foster a broader view of software projects. Expert A, B and C agreed that targets should shift from individual metrics to the overall well-being of the system.

This approach could allow more long-term thinking and propel creativity. Expert 5 made a parallel with the evolution of species in biological ecosystems. If there is enormous selection pressure, then there are not many directions that can be pursued on the fitness landscape. In such an environment any movement that is not driven by better fitness could lead to extinction. However, releasing that pressure could enable moving in directions that are not only driven by better fitness, which increases diversity and the occupation of more niches. In chapter 4.1 the importance of such a flow state and the possibility of resulting drifts have been underlined.

Usually, software engineers and managers are so overwhelmed with pressure nowadays that they knowingly continue with bad habits in their area. Managers are interested in modern management techniques such as agile project management, as they hope for more active and self-motivated employees. Unfortunately, this often fails. On the one hand, we still lack much knowledge regarding modular organisation and how it can be

designed and introduced correctly. On the other hand, these mechanistic and hierarchical structures have been developed initially because they provide clear control. Decentralising always also means losing control to a certain extent, which is is a hard sell for traditional management but also requires skilled and motivated people to take on the responsibilities. This is a deadlock situation as managers and politicians are in the role to advocate for change, yet naturally, they don't want to give up on the control and power that they have gained through these hierarchical structures.

Short-term thinking is relevant again, as management and politicians tend to stay in power only for a limited time and need to be re-elected again. This incentive will move them to focus on short-term profit maximisation rather than long-term success, as they usually do not have any skin in the game – with exceptions such as privately-owned companies. This concept is also stressed by Taleb [159]. Similar issues arise with common software engineers as they are being rewarded for job-hopping nowadays.

To overcome these issues, all experts agree that it is crucial to reduce pressure and change incentive structures to enable more long-term thinking and reward commitment.

When it comes to the question of how this change can be achieved, it takes a look at politics. If we believe in complex systems in the form in which they have been discussed so far, then it is possible to trigger global effects by local changes. Expert 6 and expert C claim that education and a sense of community appear to be particularly important in this regard. However, a top-down component, i.e. political change is necessary as well. Politicians have various control instruments at their disposal. They can introduce restrictive regulations or emancipative, e.g. subsidies. However, these pose certain challenges partly due to the bureaucratic system that is involved. Regulators tend to look at the past and disregard possible future developments when designing regulations. To quote Taleb, et. al.,

> "*By definition, evidence follows and never precedes rare impactful events [157].*"

Aside from that, it is bureaucratically challenging to set them in place and an even bigger challenge to adapt or remove them again in the future. This was discussed by experts A and E. Subsidies in particular often lead to unwanted side effects where lobbying absorbs most of the governmental funds, was added by expert A [173]. Restrictive regulations tend to hinder innovation, however, they can be very effective as they create pressure for change.

Finally, expert B and expert 6 addressed the *tyranny of metrics*. As soon as a metric is introduced for performance measurement and it is known how this metric is calculated, there is a risk of people abusing this. The focus shifts from the actual performance to fine-tuning of the metrics even if the means lead to a worse outcome. Jerry Z. Muller lists various examples including education, health care, government, but also industry [112]. In chapter 5.2.2 it is explained how the H-index metric for instance influenced academic research. Currently used metrics for performance measurement need to be revised keeping this dynamic in mind.

Cultural and Social Aspects

In chapter 5.2.1 the issues of asymmetric power distribution have been raised briefly. In Biology cooperation and symbiosis, but also competition can be observed everywhere. The latter can lead to issues as we believe that rivalrous dynamics can propel not only innovation (mutation) but progress as only the strongest, fastest, fittest will survive. Humans have applied these principles to markets in the sense that innovations can be seen as mutations and only those companies that provide the best product for the best price will survive. Therefore, it is often believed that it is competition and rivalrous dynamics that propel the development of new technology and higher quality. However, as mentioned in chapter 5.2.1 the power distribution in biology is symmetric while this doesn't hold for markets. A single individual holding so much power relative to the entire system causes the system to be fragile as opposed to the antifragility that can be observed in biological ecosystems. Philosopher Daniel Schmachtenberger sees this as one of the two *generator functions*, which lead to a *self-terminating civilisation*. He states that "rivalrous dynamics, multiplied by exponential technology, self-terminate." To enable technological advancement without these self-terminating properties he advocates for anti-rivalrous methods for human interaction, the development of technology in relationship with nature in a metastable way, and problem definitions in a wider sense so that solutions don't create worse side effects [143, 144]. Particularly, the last point about problem definitions is strongly related to the difficulties of sense-making that will be described in more detail in chapter 5.2.2. Future research could deepen and improve the understanding of software evolution in connection with Schmachterberger's generator functions.

Another societal issue that further sharpens the set of problems regarding technology is the idea of secular faiths or religions with technology being one of them [134]. Expert 2 explains that it is often expected or envisioned how technology is going to solve our organisational problems. Examples of this fantasy can be found even in academic literature where the question is posed whether people can be replaced with machines and "jobs automated out of existence". This can give the impression that interpersonal management and organisational issues are negligible for the production of high-quality software and avoidance of software failures. Another effect they mention is hype. Given the economic frame, many actors such as journalists or venture capitalists benefit from the appearance of dramatic technological development. Particularly management that is lacking a deeper understanding of the technology tends to get carried away by this hype and be attracted to new tools. Seduced by the promised benefits unrealistic technological decisions are made often and the focus on real issues is neglected. To tackle this, the myth of a magic bullet needs to be debunked. Rushkoff agrees with this and adds that it is necessary to recognise that we already know what the solutions to our problems are. This work portrays a great example of already known management and technological approaches that would greatly increase the resilience and sustainability of software systems, but aren't performed for various reasons. In section 5.2.2 it is further described why it is important to focus our attention on what already exists. The introduction of new technologies will not necessarily affect the environmental challenges of the system in

a deterministic way. To make the planet more worth living, he argues, we need to get local. The focus should shift away from designing solutions at scale since "operating at scale is itself the main problem" [136]. A fitting addition is given by Edgerton:

> "*History reveals that technological futurism is largely unchanging over time. Present visions of the future display a startling, unselfconscious lack of originality. Take the extraordinary litany of technologies that promised peace to the world. Communications technologies, from railways and steamships, to radio and the aeroplane, and now the internet, seemed to make the world smaller and bring people together, ensuring a perpetual peace [45].*"

Is it possible that new technologies such as the internet would have succeeded in making the planet more livable and not developed negative side effects (e.g. polarisation, alienation...) if it had been kept from operating on a large scale? Future research could further examine the effects that scaling has on the co-evolution of technology and society.

Another insight by expert 2 that shifts the focus from technological magic bullets to social matters is that real change is often achieved by social movements. People come together and find solidarity. They grow as a movement and can create change with that pressure. Examples include labour unions in the technology industry, anti-monopoly and right to repair movements. In history, many of the important social and legal changes have been created that way. In the end, even technical problems are often of social matter [124]. By studying cultural evolution it becomes apparent how social learning and communication propels technological and cultural complexity.

Finally, it is necessary to discuss the *tyrann of merit*. The concept describes the issues that arise in today's society with the belief that someone's achievements and success are their own merit. Michael Sandel argues that this belief is what polarises politics. It leads to arrogance in those that are successful and harsh judgements towards those that are not, even if their success or failure was largely thanks to the role of luck. An example could be someone who is born into a rich family and is yet seen as successful, while those less lucky are seen as lazy. He advocates for more humility and more respect for those that support the development and maintenance of the common good [138]. This concept might be able to tackle the issue that some of the best software engineers work for entertainment companies, such as YouTube, Facebook, or Netflix as these are offering "high end" jobs. While this might increase the qualitative change in entertainment and consumption, it does not result in deep economic changes. Raging demand and not enough developers have resulted in a state where critical infrastructure is being developed and maintained by mediocre and worse engineers, while the best sit in the entertainment space, as claimed by expert A and expert 2. Shifting the focus away from individualism to the common good and more complex forms of governance might help in this regard.

Education and Academics

The focus on short-term thinking, productivity, and profit maximisation has not only led to issues in the economic sector. Unfortunately, this approach has been extended into areas that do not necessarily profit directly from this philosophy, such as education, media, public health care, politics, or research. The results of this short-termism are well-known and visible, such as the exploitation of ecological resources. However, it has also led to a communication crisis. The education system is positioned to train us to become a useful part of the economic system without sufficiently teaching and promoting a complex way of thinking. Expert 6 explains how society is confronted with a sense-making crisis as incentives are amiss and the tempo in which changes and complexity are advancing makes it impossible to keep pace. Many issues such as climate change are not isolated phenomena but rather a result of the interplay between biology and various anthropogenic effects including the development of software systems.

Another educational issue is the ever stronger growing tendency towards experts [48]. While these are indispensable for the development of software systems, they tend to work in a very narrow focus, overlooking the overall context in which the system is embedded. The majority of experts agree that this approach of focusing on small technical details is not suitable to handle complex problems. Instead, as expert A emphasises, sociology aspects should be promoted in technical studies and the other way around. Additionally, know-how exchange and cooperation between experts from different areas are not being enforced even though product development in complex systems heavily depends on information exchange, cooperation and communication. Expert 3 argues that as the systems grow more complex, software engineering will increasingly become a social activity. Generalists play an important role too since today's challenges spread in different areas, and they are capable of connecting various expertises. However, these are rare since there is no talent pressure in this direction.

Tackling these issues would involve an education system that trains to understand and recognise complex problems, as well as, differentiate them from complicated problems. This would increase the share of the population that can process these issues. Instead of optimising the cost-value ratio, expert 3 states that we should try to "understand how these systems co-evolve, how they interact, how cultural evolution affects ecosystems and how ecosystems at the same time constrain cultural evolution". Thereby, prevention and preparations for uncertain and risky developments would be improved, increasing overall societal sustainability. This could be implemented as part of a degree programme, as well as, continuous education for executives and management, as proposed by expert 2.

Productivity obsession causes a problematic development in the area of research as well. Constant focus on short-term productivity cages research in a local maximum. Physicist, Safi Bahcall, describes crazy projects that can free scientists from this local maximum [14]. Many of these projects may fail, however, those that work are essential to allow jumping from one local maximum to the next summit, as stressed by expert 6. The academic research system seems to no longer offer a frame for this approach.

Instead, those scientists are successful, that know which criteria lead to financial funding and are capable of recognising these patterns. It is well-known in science that the optimisation of one's H-index means an achieving career. Literature of the past 20 years has unveiled the publication patterns that lead to more success, e.g. more authors per paper, shorter paper length, etc. Scientists have adapted to the system. Goodhart's Law states that "when a measure becomes a target, it ceases to be a good measure." and this development can be observed in scientific research [51].

Further side effects of this development are lurid and unrealistic claims in scientific findings as these tend to lure in more funding, which in turn gives credibility to these scientists. Once again, we are dealing with a problematic incentive system that leads to trust and funding, where sensational but unreal claims are made, instead of focusing on incremental and steady changes in existing systems, which make up most of the real improvements [107].

To summarise, the current academic research frame neither fosters long-term views, which could lead to loonshots, nor "unexciting" incremental changes that make up most of the improvements in today's systems. Instead, the focus lies on short-term projects with ridiculous claims and scientists, who understand how to make use of the patterns that lead to success in career terms.

Holding researchers accountable for their past claims could be a good start in the right direction. One example in this regard is the landmark study Tetlock, which was referenced in section 4.1.

Maintenance and Maintainability

Maintenance and maintainability of software are essential aspects and challenges in complex software systems.

On the one side, the significance of maintenance in terms of sustaining the infrastructure that we have built is undisputed. On the other side, there is this call for radical change and attraction towards innovations and new technology. Expert 2 stresses that throughout history, there were moments of disruptive innovations such as new technology in the energy industry, however, most growth comes from little adjustments in existing systems. Looking at the advancements made by AI until today, it becomes clear that many promises and predictions weren't fulfilled [43, 179, 55]. The graphic **??** illustrates AI booms in the past, which were full of unkept promises. Today's hype around AI is again filled with many unrealistic fantasies. AI could have a significant economic impact, however, these changes will happen slowly and gradually. Most of the innovations follow a rather boring process of little adjustments. Stacking all these incremental changes together results in what we call the modern world, and according to Kevin Kelly this trend will continue:

"*Technological life in the future will be a series of endless upgrades. And the rate of graduations is accelerating [83].*"

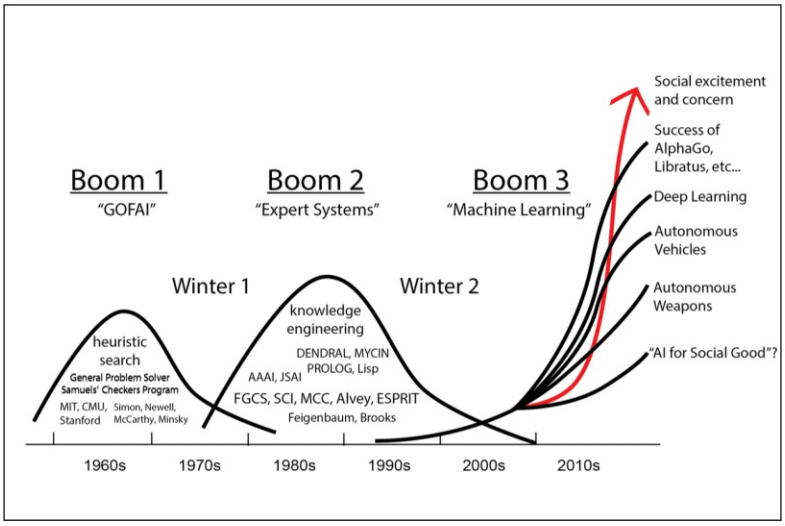

Figure 5.2: Booms and promises of AI [55]

This focus on hypes and new technology, rather than improving existing infrastructure, is generally dangerous as future predictions of technological developments are highly inaccurate. It results in a genuine risk of jeopardizing the current infrastructure, e.g. creating massive blackouts during the energy transition. Such an event could have catastrophic ramifications, which further underline the importance to maintain and sustain what we have already built instead of focusing on innovations. Unfortunately, this idea is vastly underrepresented.

The age of the software industry itself possibly plays a role in this regard. Many other industries had a long period of learning to develop best practices involving maintenance. Some developments in software systems are so new that failures are bound to happen. However, it is important to note that other factors are relevant too. Pre-emptive maintenance practices have been created in the railroad industry in the early 20^{th} century and since then have spread to other industries. Yet many plants and utilities still don't practice healthy pre-emptive maintenance even 100 years later, meaning factors such as leadership and culture are highly relevant as well.

Particularly regarding short-termism, maintenance itself is characterized by problematic traits. It requires dedicated resources which lead to redundancies. Additionally, the introduction of maintenance protocols and processes initially leads to a hit in productivity. Both contradict the notion of efficiency. However, to do something meaningful, an initial decrease in terms of output is inevitable.

More challenges are posed by both, the employee and management sides. Often employees are not interested in the benefit of maintenance and are unwilling to add more tasks to their responsibilities. Managers on the other hand, while being aware of the benefits, are constrained by competing priorities, which results in exhaustiveness and cognitive limits.

Another crucial aspect is the maintainability of the software itself. Designing and developing maintainable software should be in our interest as deterioration will happen and maintainability promotes quality in this regard. However, it also increases costs and since contracts are often designed in a way that exempt providers from liability for damages caused by their product or at least strongly limit the liability, there is no incentive for providers to focus on this aspect [114]. The nature of software has made such contracts possible, however, getting some of that liability back could significantly change the situation regarding software maintainability and quality. There is evidence that bearing the liability for defects drives engineers to develop software that is of higher quality and more secure [10].

CHAPTER 6

Conclusion

This thesis aimed to identify which biological and complexity principles are particularly relevant for sustainability and antifragility in general, and particularly what can be learned from these principles for engineering and managing complex software systems.

To answer this question, a theoretical framework to provide definitions was developed in the first step. Especially the terms *software engineering*, *complexity*, and *resilience* were important in this regard. In the frame of this work, these terms were rethought and it was concretely defined what they mean in the context of today's complex software systems. Software engineering includes steps such as development, testing, etc, but systems are increasing in complexity so much that it is not only difficult to describe how the system currently behaves, but already difficult to define how the system is supposed to behave. Therefore, the definition in this work takes social and managerial aspects, such as requirements engineering, into account as well. There are various definitions of complexity depending on the area and perspective. In this work it was described as a phenomenon that emerges from the interactions of the agents in a system, meaning that it doesn't exist on the individual level of each agent. Two more aspects were defined, namely structural complexity, which is defined by the number of relationships and interactions of a system's components, and conceptual complexity, which describes the psychological perception of how difficult it is to understand, maintain, and adapt a system. Similarly, the term resilience can be interpreted in various ways. This work describes resilience as the desired reaction of complex software systems towards internal and external perturbations in a long-term perspective.

Based on these definitions questionnaires were prepared, and seven experts from the areas of complexity science and biology were selected following a snowball approach. Expert calibration was performed as well. Then the first interview round was performed in person and virtually (where required), and lasted for two hours on average. The interview style was explorative and organised in a semi-structured manner. Based on these interviews a rough draft was developed containing the principles and mechanisms

that enable and promote resilience and antifragility in biological systems. These principles were evaluated in a subsequent literature review, and three pillars were identified to be of notable importance for antifragility and sustainability of complex systems:

- *Fault Tolerance* in the form of diversity, redundancies, and alternative pathways
- *Modularity* in the form of decentralised and confined modules, that eventually form complex interactions with other modules
- *Adaptivity* in the form of antifragility and continuous adaptions of systems

Each of these pillars increases sustainability and resilience in its way. For example, excessive dependencies on certain resources make the system fragile. Small perturbations could then have strong effects on the system's dynamics. Diversity or alternative pathways diminish such dependencies and are therefore one of the key principles in fostering resilience. Diversity even increases the adjacent possible, which could lead to more degrees of freedom in the system. However, it is important to take the trade-off with efficiency into account, and introduce fault tolerance mechanisms only where the investment pays off.

Similarly, we want a system that is well-integrated but also does not propagate errors from one module to another. The trade-off between modularity and integration needs to be considered, as well as, the granularity at which modularity is introduced. Additionally, it can be noted that modularity deteriorates over time, which means it requires continuous attention.

This is related to the last pillar, adaptivity, which states that to persist software systems need to adapt continuously to internal and external changes, such as evolving customer requirements, the development of innovative technologies, and the rise of new cyber threats. Without stress and perturbations, there is no motivation for a system to keep the capacity to adapt and evolve. This poses a problem in software engineering as sometimes perturbations or the risk of such are not visible to managers and developers or are actively avoided where possible (e.g. believing to protect critical systems), thereby leaving a false feeling of security. As systems stabilise, they ossify in the long run and become increasingly fragile, which is the opposite of what is expected usually. Periodic perturbations could help to avoid this, however, the type and quality of the perturbations are relevant. A system can be antifragile and yet not sustainable. It may be antifragile in certain aspects, but not in those where it matters.

System adaptions also affect the dynamics of its environment, which in turn requires more adaptions, as they form some kind of meta-feedback loop with other systems, where (even circular) co-dependencies exist. This cycle of co-evolution with the environment might be what propels the increase of complexity. A better understanding of this dynamic appears to be crucial to handle complex software systems in the future.

It can be concluded that all of the three mentioned principles are necessary for complex software systems to persist and avoid breakdowns. However, these principles are not limited to technical implementations only. On the contrary, the organisational dimension plays an equally significant role in this regard. Ultimately, our problems, even the technical ones, are of social matter. This was revealed in the secondary interview round with five experts from the industry, which served as an evaluation of the identified pillars. It became evident that most resistance towards the introduction of presented principles did not come from a technical perspective, but rather originated from economic and social aspects. Ultimately, our problems, even the technical ones, are of social matter. This further underlines why the definition of software engineering includes human actors explicitly.

At the heart of the problem is short-term thinking, which is a result of our economic incentives that are laid out to focus on short-term profit and the maximisation thereof. Such incentives lead to multipolar traps, e.g. the tragedy of the commons. This race has led developers and managers to knowingly practice bad habits. The sentence "Don't look at my code" can be heard from every IT office in every corner of the world. Another issue is posed by the comparatively young age of the software industry. As software is increasingly becoming an integral part of other traditionally more mature industries, these industries now inherit the problems derived from software complexity, meaning that we are still lacking many best practices and experience on how to deal with software accordingly.

One example could be the insufficient liability regulations regarding software. Historically, the very nature of software products has made it easy for lawyers to almost completely dismiss liability. Now we are starting to notice the effects of this development and initiating counter-movements.

The education system has not been left untroubled by short-termism either, training students to become a valuable part of the economic system without teaching how to tackle complex problems. People are being taught in ever so narrow areas focusing on expert knowledge and detailed know-how. This has led to a lack of generalists that can connect the wider consequences of many issues. Software engineering is particularly impaired by this development as there are rarely legal hindrances in becoming a software developer, as they can be found for lawyers, construction engineers, or physicians. It is a task that is easy to learn, but hard to master. Paired with the drastically rising demand for software developers, it is one of the main reasons for overall low software quality.

These issues combined with the increasingly complex nature of software systems makes it extremely difficult, if not impossible, to predict future developments. However, we can build in protection and anticipate catastrophes by paying attention to those interactions with a high complexity mismatch. The focus shift from short-termism to sustainability seems to require several steps including political regulation. One idea is the introduction of software sustainability as a business field with according policies in place, following the steps of other established sustainability practices with reporting standards and best

practices such as the *GRI* (Global Reporting Initiative). This would make it increasingly difficult for management to ignore.

In conclusion, it can be said that while principles for sustainable software engineering exist, they are mostly neglected and ignored due to organisational restrictions such as short-termism. Future work could try to deepen the understanding of how these restrictions can be lifted to focus on more resilient, antifragile, and sustainable software systems. It is crucial, however, to be aware that this development is a social matter and not solvable by a magic bullet.

Lastly, one aim of this thesis was to outline additional research approaches that could further help to improve software sustainability and the understanding of complex software system dynamics. The list below contains these prospective ideas on further research:

1. How can the term *software complexity* be characterised and defined more precisely in the context of the modern world? What could a standardised taxonomy look like and what would be its nature?
2. How can the concept of *generator functions* by Schmachtenberger be utilised to better understand and adjust the evolution of software systems?
3. What can the concepts of *requisite variety* and *complexity mismatch* teach us about collapses in complex software systems? How can these concepts be operationalised to prevent systemic failures?
4. How effective are self-organising technologies and methods in software systems and how can they be improved? What are the risks?
5. Which further early warning indicators similar to critical slowing and spatial resonance can be discovered? How can these indicators be utilised in the real world?
6. Are the concepts of agency and free will relevant for the emergence of complexity and if so, in what form?
7. How does scaling from a local community to a global network affect the co-evolution of technology and society? Could a focus shift from the global good to more localism aid the sustainability of digital technologies?

Appendix

Interview Guide – Biology

MAIN OBJECTIVE: How are the dynamics in a biological system similar to that of a software system?

1. What is your role in your organization? Which tasks are you responsible for?

Introduction, position und responsibilities of interview partner

2. During my research I have defined the following terms. Do you agree with these definitions and if not, how do they differ in your field?
    - **Complexity** characterises phenomena that emerge from a system of interacting agents.
    - **Robustness** is a system's ability to withstand failures and stress until a certain point.
    - **Resilience** is a system's ability to recover from failures in an appropriate time and manner.
    - **Antifragility** is a system's ability to adapt and even improve under stress.
    - **Sustainability** is a principle where resources are consumed in a way that permanent satisfaction of needs can be guaranteed by preserving the regenerative capacity of the involved systems. It covers the economic, environmental, individual, social, and technical dimension.

Clarification of terminology (for the ongoing interview and the master thesis overall)

3. Are there further systemic terms, that one should be aware of in the context of biological systems?
    - ecosystem collapse, regime shift, ecological collapse, tipping points, redundancy, modularity.

„Redundancy", „Collapse",… have been mentioned in previous analysis; Here I want to complete the terminology and based on the answers, dive deeper into the topic of characterisation, monitoring and controlling of eco systems.

4. Please elaborate which "resilient systems" exist in your area, and which properties are displayed by them.

Additional information, how concrete systems in biology are characterised (immune system, cell system).

5. What can cause structural and behavioural changes in a biological system?

I want to learn; how different events and properties can influence a system's structure or behaviour.

Ad 1. Sub question: e.g. How does high dependency of the system variables on each other influence the system?

Ad 2. Sub question: How does high variable sensitivity influence the system?

Ad 3. Sub question: When and how can a Regime Shift happen? What role play tipping points? (Transition of one equilibrium to another e.g., Transition from a stem cell to a liver cell)

Ad 4. Sub question: Could you name examples to explain the concept of a collapse. Are there differences/commonalities in the collapse of different systems, and if so, what are they?

6. A system has structural features that might impede the risk of systemic collapse or endow a system with the ability to recover from a disturbance. How can these features be identified?

Systems display various structural and behavioural properties. I want to learn which are system-benevolent and which are system-malevolent and how these can be identified. The principle of antifragility is used in the field of molecular biology. I want to learn if the expert is familiar with the concept and which process act in an antifragile manner.

Ad 1. Sub question: How are redundancy (niche overlap between species) and modularity (the interconnectedness of a system's components) relevant in this regard?

Ad 2. Sub question: Do stressors always affect the system in a bad way? Are there processes that make an entity stronger (sturdier) through the effect of ageing, shock, or stress?

7. Which early warning indicators regarding a system's state are there? Which principles form their basis?

Indicators can give hints about the state of a system and forthcoming threats. I want to learn which indicators exist in this regard and on what they are based.

8. How can a system's structure and state be modelled, and monitored and measured?

So far, we have examined which principles are relevant for a system's success/collapse. Now I want to learn, how a system's state or structure can be described/measured in general.

Ad 1. Sub question: How can a high-level representation be developed to model the environment?

Ad 2. Sub question: Which methods or tools can be used to gain cross-disciplinary insights into complex systems?

Interview Guide – Complexity

MAIN OBJECTIVE: Which principles and information from Complexity Science can be applied to the Complexity of software systems?

1. What is your role in your organization? Which tasks are you responsible for?

Introduction, position und responsibilities of interview partner

2. During my research I have defined the following terms. Do you agree with these definitions and if not, how do they differ in your field?
    - **Complexity** characterises phenomena that emerge from a system of interacting agents.
    - **Robustness** is a system's ability to withstand failures and stress until a certain point.
    - **Resilience** is a system's ability to recover from failures in an appropriate time and manner.
    - **Antifragility** is a system's ability to adapt and even improve under stress.
    - **Sustainability** is a principle where resources are consumed in a way that permanent satisfaction of needs can be guaranteed by preserving the regenerative capacity of the involved systems. It covers the economic, environmental, individual, social, and technical dimension.

Clarification of terminology (for the ongoing interview and the master thesis overall)

3. Are there further systemic terms, that one should be aware of in the context complexity science?
    - e.g., chaos, tipping points, redundancy, modularity, long tails

„Redundancy", „Collapse",… have been mentioned in previous analysis; Here I want to complete the terminology and based on the answers, dive deeper into the topic of characterisation, monitoring and controlling of complex systems.

4. Please elaborate if there are different types of complexity (organised, disorganised, structural, perceptive …) and how they differ from each other?

Additional information which types of complexity exist and which properties they share or how they differ.

Ad 1. Sub question: How can be determined if we are dealing with disorganised or organised complexity?

5. What can cause structural and behavioural changes in a complex system?

I want to learn; how different events and properties can influence a system's structure or behaviour.

> *Ad 1. Sub question: How does dependency of the system variables on each other or variable sensitivity influence the system?*
>
> *Ad 2. Sub question: When and how can a Regime Shift happen? What role is played by tipping points? (e.g., Transition of one equilibrium to another)*
>
> *Ad 3. Sub question: What exactly does a „collapse" in a system mean and how can it come to that?*

6. What kind of emergence can be observed in complex systems? How can emergent phenomena be categorised/classified?

Emergent phenomena are a critical and essential part of complex systems. I want to learn, how regularities or emergent behaviour in a complex system can be identified.

> *Ad 1. Sub question: How can this behaviour be identified?*

7. Do complex/chaotic systems exhibit properties that are universal and predictable? And if so, can you tell me more about this "universality"?

In my previous analysis and research, the concept of "universality" was described as properties, that remain the same (=universal) over many different functions. I want to gain a better understanding of this concept, the resulting properties, and their use.

8. A system has structural or behavioural features that might impede the risk of systemic collapse or endow a system with the ability to recover from a disturbance. How can these features be identified?

Systems display various structural and behavioural properties. I want to learn which are system-benevolent and which are system-malevolent and how these can be identified. I am particularly interested in the expert's view on antifragility. Example might be needed for sub question 1.

> *Ad 1. Sub question: How are redundancy (e.g., niche overlap between species) and modularity (the interconnectedness of a system's components) relevant in this regard?*
>
> *Ad 2. Sub question: Do stressors always affect the system in a bad way, or can we identify processes that make an entity stronger (sturdier) through the effect of ageing, shock, or stress?*

9. Which early warning indicators were developed/can be used and what principles form the basis of them?

Indicators can give hints about the state of a system and forthcoming threats. I want to learn which indicators exist in this regard and on what they are based.

10. How can a system's structure and wellbeing be modelled and monitored or measured?

So far, we have examined which principles are relevant for a system's success/collapse. Now I want to learn, how a system's state or structure can be described/measured in general.

Ad 1. Sub question: How can a high-level representation be developed to model the environment?

Ad 2. Sub question: Which methods or tools can be used to gain cross-disciplinary insights into complex systems?

11. Which methods or tools can be used to gain cross-disciplinary insights into complex systems?

We find cross-disciplinarity (interdisciplinarity) in biological systems as well as software systems. I want to learn if there are general tools to examine cross-disciplinarity in any complex systems, and how these are used.

12. Look to the future: Can you name concrete goals of complexity research? Which tangible added value do you expect in real systems? When do you expect it?

In my analysis complexity research is described as a relatively new and young research field; I hope to learn something about the "vision" of this research.

Bibliography

[1] Gartner: Devops funktioniert nicht. https://www.heise.de/ix/meldung/Gartner-DevOps-funktioniert-nicht-4072904.html. Accessed: 2021-08-30.

[2] Arya Abhishek, Kim Lewandowski, Dan Lorenc, and Julia Ferraioli. Finding critical open source projects. https://opensource.googleblog.com/2020/12/finding-critical-open-source-projects.html/, 12 2020. Accessed: 2021-09-28.

[3] Yasemin Acar, Michael Backes, Sascha Fahl, Doowon Kim, Michelle L. Mazurek, and Christian Stransky. You get where you're looking for: The impact of information sources on code security. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 289–305, 2016.

[4] H. Adkins, B. Beyer, A. Oprea, P. Blankinship, P. Lewandowski, and A. Stubblefield. *Building Secure and Reliable Systems: Best Practices for Designing, Implementing, and Maintaining Systems.* O'Reilly Media, 2020.

[5] A. Ahl, M. Yarime, M. Goto, Shauhrat S. Chopra, Nallapaneni Manoj. Kumar, K. Tanaka, and D. Sagawa. Exploring blockchain for the energy transition: Opportunities and challenges based on a case study in japan. *Renewable and Sustainable Energy Reviews*, 117:109488, 2020.

[6] Georg Aichholzer. *The Delphi Method: Eliciting Experts' Knowledge in Technology Foresight*, pages 252–274. 01 2009.

[7] Maryam Al Hinai and Ruzanna Chitchyan. Engineering requirements for social sustainability. 01 2016.

[8] Mamdouh Alenezi and Mohammad Zarour. On the relationship between software complexity and security. *International Journal of Software Engineering & Applications (IJSEA*, 11, 01 2020.

[9] Uri Alon. Alon, u.: Network motifs: theory and experimental approaches. nat. rev. genet. 8, 450. *Nature reviews. Genetics*, 8:450–61, 07 2007.

[10] Ross Anderson and Tyler Moore. Information security economics – and beyond. volume 4622, pages 68–91, 08 2007.

[11] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, USA, 1st edition, 2009.

[12] Matt Asay. Open source is written by big greedy capitalists. what else is new? https://www.techrepublic.com/article/open-source-is-written-by-big-greedy-capitalists-what-else-is-new/, 10 2018. Accessed: 2021-09-23.

[13] Vivek Asokan, Masaru Yarime, and Miguel Esteban. Introducing flexibility to complex, resilient socio-ecological systems: A comparative analysis of economics, flexible manufacturing systems, evolutionary biology, and supply chain management. *Sustainability*, 2017, 06 2017.

[14] S. Bahcall. *Loonshots: How to Nurture the Crazy Ideas That Win Wars, Cure Diseases, and Transform Industries*. St. Martin's Publishing Group, 2019.

[15] Gergely Bandi and Jeremy J Ramsden. Biological programming. *Journal of Biological Physics and Chemistry*, 10:39–43, 03 2010.

[16] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286(5439):509–512, 1999.

[17] Maria Bartnes. Safety vs. security? 05 2006.

[18] David Batten, Salthe Stanley, and Fabio Boschetti. Visions of evolution: Self-organization proposes what natural selection disposes. *Biological Theory*, 3, 03 2008.

[19] Christoph Becker, Ruzanna Chitchyan, Leticia Duboc, Steve Easterbrook, Martin Mahaux, Birgit Penzenstadler, Guillermo Rodriguez-Navas, Camille Salinesi, Norbert Seyff, Colin Venters, Coral Calero, Sedef Akinli Kocak, and Stefanie Betz. Sustainability design and software: The karlskrona manifesto, 10 2014.

[20] Giovanni Giuseppe Bellani. Chapter 6 - ecology and predation. In Giovanni Giuseppe Bellani, editor, *Felines of the World*, pages 309–342. Academic Press, 2020.

[21] Matt Bishop and Chip Elliott. Robust programming by example. volume 406, 01 2013.

[22] Imogen Blake. Co-op supermarkets left with rows of empty shelves after it issue causes delivery problems. *MailOnline*.

[23] Alexander Bogner and Wolfgang Menz. *The Theory-Generating Expert Interview: Epistemological Interest, Forms of Knowledge, Interaction*, pages 43–80. Palgrave Macmillan UK, London, 2009.

[24] Max Boisot and Bill McKelvey. Complexity and organization-environment relations: Revisiting ashby's law of requisite variety'. In Peter Allen, Steve Maguire, and Bill McKelvey, editors, *The Sage Handbook of Complexity and Management*, pages 279–298. Sage Publications, 2011.

[25] Rachel Botsman. Big data meets big brother as china moves to rate its citizens. https://www.wired.co.uk/article/chinese-government-social-credit-score-privacy-invasion, 2017. Accessed: 2020-11-04.

[26] Stephen W. Boyd and Angelos D. Keromytis. Sqlrand: Preventing sql injection attacks. In Markus Jakobsson, Moti Yung, and Jianying Zhou, editors, *Applied Cryptography and Network Security*, pages 292–302, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[27] S. Brand and Viking Press. *How Buildings Learn: What Happens After They're Built*. Viking, 1994.

[28] J. Brockman. *Third Culture: Beyond the Scientific Revolution*. A Touchstone book. Simon & Schuster, 1996.

[29] Rebekah Brown. Impediments to integrated urban stormwater management: The need for institutional reform. *Environmental management*, 36:455–68, 10 2005.

[30] Carole Cadwalladr and Emma Graham-Harrison. Revealed: 50 million facebook profiles harvested for cambridge analytica in major data breach. *The Guardian*.

[31] Luiz Fernando Capretz and Faheem Ahmed. Why do we need personality diversity in software engineering? *SIGSOFT Softw. Eng. Notes*, 35(2):1–11, March 2010.

[32] Steve Carpenter, Brian Walker, John Anderies, and Nick Abel. From metaphor to measurement: Resilience of what to what? *Ecosystems*, 4:765–781, 12 2001.

[33] J.L. Casti. *X-Events: The Collapse of Everything*. William Morrow, 2012.

[34] Stephanie E Chang, Tim McDaniels, Jana Fox, Rajan Dhariwal, and Holly Longstaff. Toward disaster-resilient cities: Characterizing resilience of infrastructure systems with expert judgments. *Risk Analysis*, 34:416–434, 2014.

[35] Murali Chemuturi. *Requirements Engineering and Management for Software Development Projects*, pages 33–54. 01 2013.

[36] Liming Chen and A. Avizienis. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Twenty-Fifth International Symposium on Fault-Tolerant Computing, 1995, ' Highlights from Twenty-Five Years'.*, pages 113–, 1995.

[37] Dimitrios Christopoulos. Peer esteem snowballing: A methodology for expert surveys. 01 2007.

[38] Paul Clarke, Rory O'Connor, and Brian Leavy. A complexity theory viewpoint on the software development process and situational context. pages 86–90, 05 2016.

[39] N. M. Cooke and J. E. McDonald. A formal methodology for acquiring and representing expert knowledge. *Proceedings of the IEEE*, 74(10):1422–1430, 1986.

[40] Kevin Crowston, Qing Li, Kangning Wei, U. Eseryel, and James Howison. Self-organization of teams for free/libre open source software development," inf. *Information and Software Technology*, 49:564–575, 06 2007.

[41] M. Csikszentmihalyi. *Flow: the Psychology of Optimal Experience by Mihaly Csikszentmihalyi*. CreateSpace Independent Publishing Platform, 2018.

[42] Vasilis Dakos, Marten Scheffer, Egbert Nes, Victor Brovkin, V. Petoukhov, and Hermann Held. Slowing down as an early warning signal for abrupt climate change. *Proceedings of the National Academy of Sciences of the United States of America*, 105:14308–12, 10 2008.

[43] Angel N. Desai. Artificial Intelligence: Promise, Pitfalls, and Perspective. *JAMA*, 323(24):2448–2449, 06 2020.

[44] Murillo Dias and Bruno Cruz. Crashed boeing 737-max : Fatalities or malpractice? (published version available), 01 2020.

[45] D. Edgerton. *The Shock of the Old: Technology and Global History Since 1900*. Profile Books, 2008.

[46] Bruce Edmonds. What is complexity? - the philosophy of complexity per se with application to some examples in evolution. 07 1995.

[47] Abeer Elbahrawy, Laura Alessandretti, Anne Kandler, Romualdo Pastor-Satorras, and Andrea Baronchelli. Evolutionary dynamics of the cryptocurrency market. *Royal Society Open Science*, 4:170623, 05 2017.

[48] D. Epstein. *Range: How Generalists Triumph in a Specialized World*. Pan Macmillan, 2019.

[49] H. Eriksson. A survey of knowledge acquisition techniques and tools and their relationship to software engineering. *J. Syst. Softw.*, 19:97–107, 1992.

[50] Doyne Farmer, Fotini Markopoulou, Eric Beinhocker, and Steen Rasmussen. Collaborators in creation. https://aeon.co/essays/how-social-and-physical-technologies-collaborate-to-create, 2020. Accessed: 2020-10-20.

[51] Michael Fire and Carlos Guestrin. Over-optimization of academic publishing metrics: observing Goodhart's Law in action. *GigaScience*, 8(6), 05 2019.

[52] David N Fisher and Jonathan N Pruitt. Insights from the study of complex systems for the ecology and evolution of animal populations. *Current Zoology*, 66(1):1–14, 04 2019.

[53] Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps Building and Scaling High Performing Technology Organizations*. IT Revolution Press, 1st edition, 2018.

[54] Gallo Francesco, Marco Autili, Massimo Tivoli, Alexander Perucci, and Amleto Di Salle. Biological immunity and software resilience: Two faces of the same coin? volume 9274, 09 2015.

[55] Shunryu Garvey. Broken promises and empty threats: The evolution of ai in the usa, 1956-1996. *Technology's Stories*, 03 2018.

[56] Giampaolo Garzarelli and Riccardo Fontanella. Open source software production, spontaneous input, and organizational learning. *Organization Science*, 4, 11 2011.

[57] A. Gawande. *The Checklist Manifesto: How to Get Things Right*. Profile, 2011.

[58] Adam Gluck. Introducing domain-oriented microservice architecture. `https://eng.uber.com/microservice-architecture/`, 7 2020. Accessed: 2021-09-19.

[59] D. Graeber. *Bullshit Jobs: A Theory*. Penguin Books Limited, 2018.

[60] John Graham-Cumming. Details of the cloudflare outage on july 2, 2019, Jul 2019.

[61] John Graham-Cumming. Cloudflare outage on july 17, 2020, Jul 2020.

[62] R. Harris and P. Johnston. The boeing max saga: Lessons for software organizations. *Software Quality Professional*, 21:410–424, 2019.

[63] A Hart. Knowledge acquisition for expert systems. 1 1986.

[64] M. Heffernan. *Uncharted: How to Map the Future*. Simon & Schuster UK, 2020.

[65] Margaret Heffernan. The human skills we need in an unpredictable world. https://www.ted.com/talks/margaret_heffernan_the_human_skills_we_need_in_an_unpredictable_wo 2019. Accessed: 2020-12-06.

[66] Fergus Henderson. Software engineering at google, 2020.

[67] Tu Honglei, Sun Wei, and Zhang Yanan. The research on software metrics and software complexity metrics. In *Proceedings of the 2009 International Forum on Computer Science-Technology and Applications - Volume 01*, IFCSTA '09, page 131–136, USA, 2009. IEEE Computer Society.

[68] Nick Hopkins. Deloitte hit by cyber-attack revealing clients' secret emails. *The Guardian*.

[69] S. E. Hove and B. Anda. Experiences from conducting semi-structured interviews in empirical software engineering research. In *11th IEEE International Software Metrics Symposium (METRICS'05)*, pages 10 pp.–23, 2005.

[70] Watts Humphrey. Why big software projects fail: The 12 key questions. *Software Management, Seventh Edition*, 03 2005.

[71] W.S. Humphrey. *Winning with Software: An Executive Strategy*. SEI Series in Software Engineering. Pearson Education, 2001.

[72] A. Hunt and D. Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Pearson Education, 1999.

[73] Reinhold Hölscher and Ralph Elfgen. *Herausforderung Risikomanagement: Identifikation, Bewertung und Steuerung industrieller Risiken*. Gabler Verlag, 2002.

[74] Bilgin Ibryam. Software: From fragile to antifragile. https://dzone.com/articles/software-from-fragile-to-antifragile, 2016. Accessed: 2020-12-06.

[75] Scaled Agile Inc. Safe 5 for lean enterprises. `https://www.scaledagileframework.com/`, 2021. Accessed: 2021-10-03.

[76] Yury Izrailevsky and Ariel Tseitlin. The netflix simian army. `https://netflixtechblog.com/the-netflix-simian-army-16e57fbab116/`, 2011. Accessed: 2021-10-03.

[77] Yasir Javed, Mamdouh Alenezi, Mohammed Akour, and Ahmad Alzyod. Discovering the relationship between software complexity and software vulnerabilities. *Journal of Theoretical and Applied Information Technology*, 96, 07 2018.

[78] J. Johnson. *CHAOS Report: Decision Latency Theory: It Is All About the Interval*. CHAOS report series. Standish Group, 2018.

[79] N. F. Johnson. Simply complexity: A clear guide to complexity theory. 2007.

[80] Marcus Jung. Sammelklage gegen mariott in grossbritannien. *Frankfurter Allgemeine*.

[81] S. Kauffman, P.B.B.S. Kauffman, and S.A. Kauffman. *At Home in the Universe: The Search for Laws of Self-organization and Complexity*. Oxford paperbacks. Oxford University Press, 1995.

[82] R. Kazman and L. Pasquale. Software engineering in society. *IEEE Software*, 37(1):7–9, 2020.

[83] Kevin Kelly. *The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future*. Viking Press, USA, 2016.

[84] Jessica Joy Kerr. Grow to where we're going. `https://jessitron.com/2020/08/03/grow-to-where-were-going/`, 8 2020. Accessed: 2021-09-29.

[85] Jennifer Key. Some error-correcting codes and their applications. 05 2003.

[86] S.K. Khaitan and James Mccalley. Design techniques and applications of cyber-physical systems: A survey. *IEEE Systems Journal*, 9:1–16, 07 2014.

[87] Yasoob Khalid. Software engineering within spacex. `https://yasoob.me/posts/software_engineering_within_spacex_launch/`. Accessed: 2021-08-31.

[88] Daniel Kiel, Julian Müller, Christian Arnold, and Kai-Ingo Voigt. Sustainable industrial value creation: Benefits and challenges of industry 4.0 [rewarded with ispim best student paper award]. 06 2017.

[89] Witold Kinsner. *System Complexity and Its Measures How Complex Is Complex*, volume 323, pages 265–295. 12 2010.

[90] K. Kornwachs. *Philosophie der Technik: Eine Einführung*. Beck'sche Reihe. C.H.Beck, 2013.

[91] Alexander Kott and Tarek Abdelzaher. Resiliency and robustness of complex, multi-genre networks, 2016.

[92] J. Ladyman and K. Wiesner. *What Is a Complex System?* Yale University Press, 2020.

[93] Jeanne Leffers and Emma Mitchell. Conceptual model for partnership and sustainability in global health. *Public health nursing (Boston, Mass.)*, 28:91–102, 01 2011.

[94] Shenggang Li, Shuguo Chen, Weibin Ding, Zhongzheng Xiang, and Yuanyuan Liu. Distributed power trading system based on blockchain technology. *Complexity*, 05 2021.

[95] Chris Lüer, David S. Rosenblum, and André van der Hoek. The evolution of software evolvability. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, page 134–137, New York, NY, USA, 2001. Association for Computing Machinery.

[96] A. Luo, Zhang M., Yi. Mao, Y. Kou, and X. Zhang. A structural complexity metric method for complex information systems. *Journal of Software*, 14(7):332–339, 2019.

[97] Robyn R. Lutz. Software engineering for safety: A roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, page 213–226, New York, NY, USA, 2000. Association for Computing Machinery.

[98] J. Magenheim, W. Nelles, T. Rhode, N. Schaper, S. Schubert, and P. Stechert. Competencies for informatics systems and modeling: Results of qualitative content analysis of expert interviews. In *IEEE EDUCON 2010 Conference*, pages 513–521, 2010.

[99] S. Mancuso. *The Software Craftsman: Professionalism, Pragmatism, Pride*. Robert C. Martin Series. Pearson Education, 2014.

[100] Marcelo Marinho, Rafael Camara, and Suzana Sampaio. Toward unveiling how safe framework supports agile in global software development. *IEEE Access*, 9:109671–109692, 2021.

[101] D. Meadows and D. Wright. *Thinking in Systems: A Primer*. Chelsea Green Publishing, 2008.

[102] Álvaro Menezes and Rafael Prikladnicki. Diversity in software engineering. pages 45–48, 05 2018.

[103] Tom Mens. An ecosystemic and socio-technical view on software maintenance and evolution. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–8, 2016.

[104] Ines Mergel, Noella Edelmann, and Nathalie Haug. Defining digital transformation: Results from expert interviews. *Gov. Inf. Q.*, 36(4), 2019.

[105] Michael Meuser and Ulrike Nagel. *The Expert Interview and Changes in Knowledge Production*, pages 17–42. 01 2009.

[106] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298(5594):824–827, 2002.

[107] Christopher Mims. New research busts popular myths about innovation. `https://www.wsj.com/articles/new-research-busts-popular-myths-about-innovation-11631937693/`, 9 2021. Accessed: 2021-09-20.

[108] Ming Li and C. S. Smidts. A ranking of software engineering measures based on expert opinion. *IEEE Transactions on Software Engineering*, 29(9):811–824, 2003.

[109] M. Mitchell. *Complexity: A Guided Tour*. Oxford University Press, 2009.

[110] Erich Moechel. *Internet-Großausfälle werden immer wahrscheinlicher*, Jul 2019.

[111] John Moteff and Paul Parfomak. Critical infrastructure and key assets: Definition and identification. page 20, 10 2004.

[112] J.Z. Muller. *The Tyranny of Metrics*. Princeton University Press, 2019.

[113] A. Murray, M. Mejias, and Peter Keiller. Resilience methods within the software development cycle. 2017.

[114] Daniel Métayer, Manuel Maarek, Eduardo Mazza, Marie-Laure Potet, Stephane Frenot, Valérie Viet Triem Tong, Nicolas Craipeau, and Ronan Hardouin. Liability issues in software engineering the use of formal methods to reduce legal uncertainties. *Commun. ACM*, 54:99–106, 04 2011.

[115] Maika Möbus. Scorecards 2.0: Sicherheitsrisiken in open-source-software aufdecken. `https://www.heise.de/news/Scorecards-2-0-Sicherheitsrisiken-in-Open-Source-Software-aufdecken-6127588.html/`, 7 2021. Accessed: 2021-09-28.

[116] Cen Nan, Giovanni Sansavini, Wolfgang Kröger, and Hans Heinimann. A quantitative method for assessing the resilience of infrastructure systems. *PSAM 2014 - Probabilistic Safety Assessment and Management*, 01 2014.

[117] Peter Naur and Brian Randell, editors. *Proceedings, NATO Conference on Software Engineering*, Garmisch, Germany, 1968.

[118] Jean Pierre Nzabahimana. Analysis of security and privacy challenges in internet of things. In *2018 IEEE 9th International Conference on Dependable Systems, Services and Technologies (DESSERT)*, pages 175–178, 2018.

[119] Phelim O'Neill. Cyber security an issue for everyone. *Irish Farmers Journal*.

[120] Shola Oyedeji, Ahmed Seffah, and Birgit Penzenstadler. A catalogue supporting software sustainability design. *Sustainability*, Volume 10, 07 2018.

[121] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, USA, 2006. USENIX Association.

[122] Barry O'Reilly. An introduction to residuality theory: Software design heuristics for complex systems. *Procedia Computer Science*, 170:875–880, 01 2020.

[123] Patroklos Papapetrou. Software gardening: Yet another crappy analogy or a reality? *Methods & Tools*, 23:4–9, 2015.

[124] C. Perrow and Princeton University Press. *Normal Accidents: Living with High Risk Technologies*. Princeton paperbacks. Princeton University Press, 1999.

[125] Rajiv Prabhakar. Why is there so much crap software in the world. https://software.rajivprab.com/2020/12/18/why-is-there-so-much-crap-software-in-the-world/, 2020. Accessed: 2021-10-04.

[126] Thomas Pradeu and Eric Vivier. The discontinuity theory of immunity. *Science immunology*, 07 2016.

[127] Chris Preist, Daniel Schien, and Eli Blevis. Understanding and mitigating the effects of device and cloud service design decisions on the environmental footprint of digital infrastructure. pages 1324–1337, 05 2016.

[128] Nikita Prokopov. Software disenchantment. https://tonsky.me/blog/disenchantment/, 2017. Accessed: 2021-10-05.

[129] C. V. Ramamoorthy, Wei Tek Tsai, Tsuneo Yamaura, and Anupam Bhide. Metrics guided methodology. In *Proceedings - IEEE Computer Society's International Computer Software & Applications Conference*, Proceedings - IEEE Computer Society's International Computer Software &; Applications Conference, pages 111–120. IEEE, December 1985.

[130] Andreas Rausch, Christian Bartelt, Sebastian Herold, Holger Klus, and Dirk Niebuhr. *From Software Systems to Complex Software Ecosystems: Model- and Constraint-Based Engineering of Ecosystems*, pages 61–80. 12 2013.

[131] R. Riedl. *Strukturen der Komplexität: Eine Morphologie des Erkennens und Erklärens*. Springer Berlin Heidelberg, 2013.

[132] Rupert Riedl. *Die Strategie der Genesis*. Serie Piper ; 290. Piper, 3. aufl. edition, 1984.

[133] Horst W. J. Rittel and Melvin M. Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4(2):155–169, 1973.

[134] Brian Rogers. Religiously we dwell: Heidegger's later contribution to philosophy of religion: Religiously we dwell. *New Blackfriars*, 96, 07 2015.

[135] C. Rosenthal and N. Jones. *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media, 2020.

[136] Douglas Rushkoff. Exponential tech doesn't serve social good. https://onezero.medium.com/exponential-tech-doesnt-serve-social-good-c6263dc5ab66, 2021. Accessed: 2021-10-05.

[137] Mary Sánchez-Gordón and Ricardo Colomo-Palacios. Characterizing devops culture: A systematic literature review. In Ioannis Stamelos, Rory V. O'Connor, Terry Rout, and Alec Dorling, editors, *Software Process Improvement and Capability Determination*, pages 3–15, Cham, 2018. Springer International Publishing.

[138] M.J. Sandel. *The Tyranny of Merit: What's Become of the Common Good?* Penguin Books Limited, 2020.

[139] Dubey Santosh J. Extracellular software development ecosystem with organic supervenience for methodology neutrality and process continuity. Jul 2018.

[140] Alexander Schatten. Zukunft denken - innovation oder fortschritt? https://open.spotify.com/episode/4zZkpCvuO7AL2n9Yin3Fj8, 2019. Accessed: 2021-01-15.

[141] M. Scheffer, Stephen Carpenter, J.A. Foley, Carl Folke, and Brian Walker. Catastrophic shifts in ecosystems. *nat*, 413:591–, 01 2001.

[142] Marten Scheffer and Stephen Carpenter. Catastrophic regime shifts in ecosystems: Linking theory to observation. *Trends in Ecology & Evolution*, 18:648–656, 12 2003.

[143] Daniel Schmachtenberger. Solving the generator functions of existential risk. `https://civilizationemerging.com/solving-generator-function/`, 10 2017. Accessed: 2021-09-23.

[144] Daniel Schmachtenberger. Utopia or bust designing a non self terminating civilization daniel schmachtenberger. `https://www.youtube.com/watch?v=J9f5tuzzFxY&ab_channel=ThePoetryofPredicament`, 9 2020. Accessed: 2021-09-23.

[145] Marcelo Serrano Zanetti. *A complex systems approach to software engineering.* PhD thesis, ETH Zurich, Zürich, 2013.

[146] Yadarisa Shabong. British airways settles with 2018 data breach victims. *Reuters.*

[147] Sylvia B. Sheppard, Elizabeth Kruesi, and Bill Curtis. The effects of symbology and spatial arrangement on the comprehension of software specifications. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, page 207–214. IEEE Press, 1981.

[148] Charles J Sherr and Norman E Sharpless. Forging a signature of in vivo senescence. *Nature reviews. Cancer*, 07 2015.

[149] Bram A. Siebert, Cameron L. Hall, James P. Gleeson, and Malbor Asllani. Role of modularity in self-organization dynamics in biological networks. *Physical Review E*, 102(5), Nov 2020.

[150] Herbert A. Simon. *The Sciences of the Artificial (3rd Ed.).* MIT Press, Cambridge, MA, USA, 1996.

[151] Daniel AJ Sokolov. Botnetze können das stromnetz sabotieren. *heise online.*

[152] James Sterbenz, David Hutchison, Egemen Cetinkaya, Abdul Jabbar, Justin Rohrer, Marcus Schöller, and Paul Smith. Redundancy, diversity, and connectivity to achieve multilevel network resilience, survivability, and disruption tolerance (invited paper). *Springer Telecommunication Systems Journal*, 56, 01 2012.

[153] George Sugihara. On early warning signs. *SEED Magazine*, pages 60–65, 12 2010.

[154] Nassim Taleb. Nassim taleb looks at what will break, and what won't. Nov 2010.

[155] Nassim Taleb, Rupert Read, Raphael Douady, Joseph Norman, and Yaneer Bar-Yam. The precautionary principle (with application to the genetic modification of organisms). 10 2014.

[156] Nassim Nicholas Taleb. *Antifragile*. Random House, 2012.

[157] Nassim Nicholas Taleb, Yaneer Bar-Yam, and Pasquale Cirillo. On single point forecasts for fat-tailed variables. *International Journal of Forecasting*, 2020.

[158] N.N. Taleb. *The Black Swan: The Impact of the Highly Improbable*. Penguin Books Limited, 2008.

[159] N.N. Taleb. *Skin in the Game: Hidden Asymmetries in Daily Life*. Random House Publishing Group, 2018.

[160] P. Tetlock and D. Gardner. *Superforecasting: The Art and Science of Prediction*. Random House, 2015.

[161] Beate Thomashausen. Computerproblem sorgt für leere regale in eisleber discounter. *Mitteldeutsche Zeitung*.

[162] Marc Tollis, Amy M. Boddy, and Carlo C. Maley. Peto's paradox: how has evolution solved the problem of cancer prevention? *BMC Biology*, 07 2017.

[163] Gregory Travis. How the boeing 737 max disaster looks to a software developer. Apr 2019.

[164] Rainer Trinczek. *How to Interview Managers? Methodical and Methodological Aspects of Expert Interviews as a Qualitative Method in Empirical Social Research*, pages 203–216. Palgrave Macmillan UK, London, 2009.

[165] Vivian Tunn, Nancy Bocken, E.A. Hende, and Jan Schoormans. Business models for sustainable consumption in the circular economy: An expert study. *Journal of Cleaner Production*, 212, 11 2018.

[166] Vaiman, Bell, Chen, Chowdhury, Dobson, Hines, Papic, Miller, and Zhang. Risk assessment of cascading outages: Methodologies and challenges. *IEEE Transactions on Power Systems*, 27(2):631–641, 2012.

[167] Uwe Valentini. Das funktioniert bei uns nicht! https://blog.hood-group.com/blog/2016/07/13/das-funktioniert-bei-uns-nicht/. Accessed: 2021-08-31.

[168] Sergi Valverde, Blai Vidiella Rocamora, Raúl Montañez Martínez, Aurora Fraile, Soledad Sacristán, and Fernando García-Arenal. Coexistence of nestedness and modularity in host–pathogen infection networks. *Nature Ecology & Evolution*, 4:1–10, 04 2020.

[169] Niels van Doorn. Platform labor: on the gendered and racialized exploitation of low-income service work in the 'on-demand' economy. *Information, Communication & Society*, 20(6):898–914, 2017.

[170] Bill Venners. Programming is gardening, not engineering - a conversation with andy hunt and dave thomas, part vii. https://www.artima.com/articles/programming-is-gardening-not-engineering/, 4 2013. Accessed: 2021-09-29.

[171] L. Vinsel and A.L. Russell. *The Innovation Delusion: How Our Obsession with the New Has Disrupted the Work That Matters Most*. Crown, 2020.

[172] M. Mitchell Waldrop. *Complexity: the emerging science at the edge of order and chaos*. Simon & Schuster, New York, 1992.

[173] Kortmann Walter. Subventionen: die verkannten nebenwirkungen. *Wirtschaftsdienst*, 84(7), 2004.

[174] Ying Wang, Zhiliang Zhu, Hai Yu, and Bo Yang. Risk analysis on multi-granular flow network for software integration testing. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(8):1059–1063, 2018.

[175] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–442, June 1998.

[176] Ron Westrum. A typology of organisational cultures. *Quality & safety in health care*, 13 Suppl 2:ii22–7, 01 2005.

[177] Danny Weyns. Engineering self-adaptive software systems – an organized tour. pages 1–2, 09 2018.

[178] Danny Weyns. *Software Engineering of Self-adaptive Systems*, pages 399–443. Springer International Publishing, Cham, 2019.

[179] Jack Wilkinson, Kellyn Arnold, Eleanor Murray, Maarten van Smeden, Kareem Carr, Rachel Sippy, Marc de Kamps, Andrew Beam, Stefan Konigorski, Christoph Lippert, Mark Gilthorpe, and Peter Tennant. Time to reality check the promises of machine learning- powered precision medicine. *The Lancet Digital Health*, 2, 09 2020.

[180] T. Winters, T. Manshreck, and H. Wright. *Software Engineering at Google: Lessons Learned from Programming Over Time*. O'Reilly Media, 2020.

[181] Katinka Wolter, Alberto Avritzer, Marco Vieira, and Aad van Moorsel. *Resilience Assessment and Evaluation of Computing Systems*. 01 2012.

[182] Rozanski Woods. The availability and resilience perspective. https://www.viewpoints-and-perspectives.info/home/perspectives/availability-and-resilience/, 2015. Accessed: 2020-10-21.

[183] Lin Xie, Solfrid Habrekke, Yiliu Liu, and Mary Ann Lundteigen. Operational data-driven prediction for failure rates of equipment in safety instrumented systems: A case study from the oil and gas industry. *Journal of Loss Prevention in the Process Industries*, 60:96–105, 2019.

[184] E. Yourdon, Y. EDWARD, ward Yourdon, L. Constantine, L.L. Constantine, and Y. Press. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Yourdon Press computing series. Prentice Hall, 1979.

[185] Liguo Yu. Self-organization process in open-source software: An empirical study. *Information and Software Technology*, 50(5):361–374, 2008.

[186] Yutao Ma, Keqing He, and Dehui Du. A qualitative method for measuring the structural complexity of software systems based on complex networks. In *12th Asia-Pacific Software Engineering Conference (APSEC'05)*, pages 7 pp.–, 2005.

[187] Tobias Zillner. 500.000 recalled pacemakers, 2 billion $ stock value loss. sec4dev conference, 2019.

[188] Shoshana Zuboff. *The Age of Surveillance Capitalism: The Fight for a Human Future at the New Frontier of Power*. 1st edition, 2018.

[189] Horst Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter & Co., USA, 1991.