Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

JOURNAL ARTICLE

Software Failure Analysis and Mitigation Techniques: Featuring Complexity, Safety, Quality and Resilience

Bhanu Prakash

[1]Master of Science in Computer Science Candidate, Sofia University, Palo Alto, CA, 94303 USA
[2]Software Engineering Manager and Principal Software Engineer - Independent Contractor, Prudential financial, HCL America and more, Sunnyvale, CA, 94081 USA (e-mail: bprakg@hotmail.com)

Correspondence
*Bhanu Prakash. Email: bprakg@hotmail.com

Present Address
Santa Clara, California.

Studies have found critical software malfunctions responsible for some of the worst accidents in recent times. These malfunctions are often only minor defects that snowball into large problems; a few lines of code is all it takes. Complexity, safety, quality, and resilience are among the key attributes defining a software's operational success. There are many leading factors for complexity, such as increases in the product size, the rate of requirement changes, and the number and type of stakeholders, and failure to manage these issues efficiently always has the same consequence, i.e., massive failure and sometimes technological catastrophe. This work analyzes some of the architecture, design, and implementation guidelines used as detection and mitigation techniques. It also discusses the safety considerations, as considering how the steam industry has handled safety issues could offer some guidance for ensuring safety. Complexity in such systems also causes some of the worst side effects from the quality auditor perspective. While failures in software are hard to predict, one of the most significant ways of showing preparedness is practicing software resilience. New mitigation areas, such as the fragility spectrum and failure obviation, and their usage for building a safer system are analyzed. Also discussed are various architecture styles in practice and the dramatic effect human factors can have on the success of the software being developed.

KEYWORDS:
Complexity Analysis & Mitigation, Software Architecture & Design, Safety, Quality, Fragility, Failure Obviation.

1 | INTRODUCTION

Failures in software systems can be fatal. Consider the following catastrophic accidents; there have been many more, but these examples significantly impact our lives and many precious lives have been lost/affected. These failures are directly attributed to failures in the underlying software systems. The fall of the Knight Capital Group[1] - The company's stock trading software algorithm placed erroneous bids and caused a major stock market disruption on the 1st of August 2012. This resulted in a loss of over 440 million dollars to the company in under forty-five minutes. Healthcare.gov debacle - Reference[2] stated "Healthcare.gov was a case with significant political impact, which attracted enough attention from mass media and opinion makers, who were willing to comment on or discuss such a failure on Twitter. This high-impact case is not common for most e-government project failures...the Obama administration has spent roughly $840 million on HealthCare.gov, including more than $150 million just in cost overruns for the version that failed so badly when it launched". Mt. Gox bitcoin debacle: huge heist or sloppy glitch?[3] - In February of 2014, the CEO of Mt. Gox, Mr. Mark Karpeles, filed for bankruptcy at a Tokyo district court. According to his lawyers, hackers had stolen all of the company's approximately 850,000 bitcoins, amounting to $473 million. At that time, Mt. Gox was handling over 80% of the worlds bitcoin trades. The Patriot

© 2021 by the author(s). Distributed under a Creative Commons CC BY license.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

Missile Failure[4] - A Report of the general accounting office indicated that a software problem had led to the failure of the Patriot Missile Defense system at Dhahran, Saudi Arabia. As a result of this failure, a scud struck an army barrack killing 28 and injuring more than 100 soldiers. Computer problems hit CSA payouts[5] - In 2003, a British multi-million pound computer system intended to automate child support payments missed paying thousands of single parents while overpaying others. The over payed amount was approximately $1.9 million, while 700,000 parents had uncollected child support payments of approximately $7 billion. Ariane 5 Flight 501 Failure[6] - A report by the inquiry board concluded that the reviews and tests carried out had not included adequate testing of the flight control system, which could have detected the potential failure. This failure resulted in the loss of money spent to develop the system, which was approximately $8 billion, in addition to the cost of the satellite, which was half a billion dollars. Mars Climate Orbiter Failure Board Releases Report[7] - The failure board report stated that errors within some of the ground-based computer models were undetected. Among other findings, it was also established that the systems engineering function that was suppose to double check all the integration aspects of the mission was not sufficiently robust. The system cost approximately two hundred million dollars to build this. Bashkiria train-gas pipeline disaster[8] - In 1989, a propane pipe explosion destroyed two passenger trains in the Bashkirian Republic of the then Soviet Union killing over 400 passengers and leaving another 806 with severe thermal injuries. The failure reports of these disasters show that a malfunction in critical system software can be devastating; thus, there is absolutely no room for error.

This study features low complexity, safety, excellent quality, and resilience as key characteristics of a successful and safe software system. It examines some of the architecture, design, and implementation guidelines used as complexity detection and mitigation techniques in sections two, three, and four. Section five concentrates on safety considerations, and an assessment of how the steam industry has handled safety issues could offer some guidance for ensuring safety. Furthermore, how complexity in these systems causes some of the worst side effects from the quality auditor's perspective is considered in section six. Section seven analyzes new mitigation areas such as fragility spectrum and failure obviation and their usage for building a safer system.

2 | ISSUES WITH COMPLEXITY IN SOFTWARE

Complexity in software is one of the leading causes of failure because of the nature of the problem of automation and the inherent complexity of a software system. It causes risk, safety, and security issues and displays an emergent behavior. While there have been models and mitigation techniques, there is no definitive theory or industry standard. Emergent behavior is an undesired state of a system not dependent on its individual parts but on their relationship with each other[9]. They say "Emergence is when some totally new phenomenon emerges out of the collective behavior of much simpler parts where the individual simpler parts are responding through simple rules to their local environment". This is an unexpected outcome that manifests when all the individual composing systems are brought together in production. Software systems automate some part of or the entire real or imaginary world, and hence, they are at least as complex as the domain. This is 'essential complexity', i.e., the inherent complexity of the domain plus the additional complexity from the software code owing to the structure and process formalization[10 11]. Measuring the essential complexity contained in a software system is an ambitious task since software systems can inherit an infinitely large number of problem domains and defining complexity means arriving at a universal complexity theory. There have been a few attempted measures of software complexity that can potentially lead to good progress, but no defining theory has emerged.

Information ecosystems are made of algorithms, networks and software systems[12]. As seen in the next sections, algorithms are mathematically verifiable through complexity theories and network through information in a communication channel. However, unlike these two, there is no verifiable comprehensive theory for software system complexity. Many factors influence software complexity. It arises due to increase in size, changes in requirements, inherent complexity of the business domain being automated, certain numbers or types of stakeholders, etc[13 10]. While there may be additional causes, these normally remain hidden during the initial engineering phases. Only during production or later running phases do they eventually display emergence. Software projects are like terrifying werewolves[10]. They might seem benign and under control until something goes terribly wrong; then, in no time, the situation spirals into an unmanageable nightmare. Unlike werewolves, there is no silver bullet for this beast nor are there any on the horizon. There is no agreed upon development strategy that can influence considerable improvement in the design and development of software.

Complexity in software is due to its *essence* – the inherent difficulties in building software and *accidents* – something a running software is susceptible to in addition to its inherent complexity[12 10]. Essential complexity is compounded by the complexity of the solution[14 11], which is due to the cognitive complexity i.e., the increased effort involved in understanding the problem domain, and algorithmic complexity of the solution.

While complexity is one of the main inherent properties, others include conformity, changeability and invisibility[10]. The size of software projects is much larger than the size of other engineering constructs. Unlike other engineering fields, the concept of repeating entities is relatively less. The volume of the unknown and the number of states handled are massive. Some of the real-world problems represented in algorithms involve exponential increases in states and data with respect to the inputs. Problems of complexity are not merely technical; they also affect the coordination

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

and working of teams, software delivery schedules, timelines, time required for the product to go from test bed to market, revenue, conformity to local regulatory compliance, system security, data security, research, and much more.

Enterprise architectures consider complexity as something that is difficult or impossible to address. Reference [15] stated the following: “In the context of EA, complexity involves coping with situations that are either very difficult or utterly impossible to comprehend in their entirety.” They further say that when addressing complexity issues, it is seldom possible if not impossible to address all possible scenarios. This leads to uncertainty.

3 | REVEALING COMPLEXITY

Complexity can exist at the architectural, design or code level. This section mostly covers revealing code complexity and a small part of design complexity. The next section addresses architecture and design complexity in more detail. As stated in previous sections, there is no standard way of identifying complexity. Some of the tools and techniques described in this section provide a way to identify problem areas. Each approach is specific to the enterprise, domain, technology, architecture, design and code being implemented. A combination of this along with the mitigation techniques prescribed in the following section helps identify problem areas very early in the software development life cycle. Identifying these issues early on and addressing them before they spiral out of control are the ideal way of dealing with inherent, algorithmic, and solution complexity. The objective of this research is to establish improvement techniques for reducing complexity. This study does not address their implementation details since there are many studies detailing them, for example, while documenting this research, there were 798 citations of Halstead’s complexity theory.

3.1 | Domain specific modeling

Depending on the intent and software engineering phase, a model-driven engineering approach is used for constructing, supporting, analyzing, and improving the modeling of complex system engineering projects, and a domain-specific modeling approach is used for revealing or explaining the complexity in a system [16].

3.2 | Unified Modeling Language (UML) and Systems Modeling Language (SysML)

UML is a semi-formal syntactic and semantic modeling language for software architectures and designs [17]. It describes two families of diagrams: behavioral diagrams that model what happens in the system being modeled and structural diagrams that serve as representations of the system being modeled. Reference [18] shows it is possible to measure the complexity of a system from its UML model, it states the following: "...fractal complexity measure α can be used to assess the complexity of a software system, and on the proposition that α is somehow related to the amount of intellectual energy built into the software artifacts, we tested a hypothesis that by measuring the complexity of UML model of a software system, the complexity of the system’s implementation can be foreseen". SysML is an extension of UML [19]; it is a profile of UML2 with additional modeling features for system engineering developed by OMG, INCOSE, and AP233.

3.3 | Program complexity - Algorithms and data structures

While it is possible to find tools that perform a static analysis of the code for cyclomatic complexity or other kinds of static analysis, there is another dimension to complexity that still cannot be addressed: use of the appropriate algorithm and the correct data structure. For program complexity, the average or worst case (Big O) needs to be used to identify the optimum complexity [20]. This is further categorized into runtime or memory complexity, for example, an algorithm requiring much memory could operate extremely fast compared to one that uses memory efficiently. This could be acceptable in large machines or embedded systems but not on mobile devices. Mitigating code complexity in these scenarios is specific to the type of automation being carried out.

3.4 | Halstead

Halstead established that the difficulty in programming algorithms increases as the number of operators increases and as the number of operand occurrences increases [21 12]. He proposed that the longer version of the same algorithm is more difficult and requires more time to code than its shorter version. He also proposed that the algorithmic complexity and cognitive complexity are proportional to the difficulty and size of the code.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

3.5 | McCabe

McCabe's cyclomatic complexity analyzes all the independent paths an execution takes while a particular program is running [22]. This can be laid out as a series of circuits from program start to end, and the total number of paths taken will be the number of circuits formed. The mathematical equivalent of this is $complexity = E - N + 2P$, where E is the number of edges, N is the number of nodes and P is the number of partitioned programs. There is also a graphical way of arriving at the same result, which involves writing connected graphs of the program, and the complexity is the total of all the cyclic graphs (considering the start and end of the parent as a cyclic graph).

3.6 | Henry and Kafura

Henry and Kafura suggested a structural complexity theory that measures the inter-component information flow. As per this theory, the procedure, module and interface measurements reveal potential design and implementation difficulties [23]. The total complexity of these modules $Cp = length * (fan - in * fan - out)^2$, where length is the length of the procedure in terms of the number of lines of code, fan-in is the amount of information flow into the module, and fan-out is the amount of flow out of the module.

3.7 | Chidamber and Kemerer

Chidamber and Kemerer identified six metrics for OO (Object Oriented) programs [24]. These were put forth to measure the efficiency of OO programs, and failure to meet them will increase the complexity of the programs. Their metric have dominated the measure of all OO features since this is the first complete design metric. Their metrics are as follows:

WMC: Weighted method per class.

DIT: Depth of the inheritance tree.

NOC: Number of children.

CBO: Coupling between object classes.

RFC: Response for a class; and

LCOM: Lack of cohesion of the methods.

4 | MITIGATING COMPLEXITY

The previous section detailed the identification and mitigation of code-level complexity. Depending on the area of existence, there are mitigating techniques in the form of best practices, patterns, and frameworks. Some of the most prominent mitigation techniques under each category are introduced in this section.

4.1 | Design level

Design patterns are standard solutions for standard problems. Reference [25] [26] describes many uses of design patterns, which they believe reduce software complexity in particular by naming and defining abstractions. Patterns form a repository of experience for building reusable software and act as building blocks for constructing more complex designs. Thus, many other domain or implementation-specific design patterns have emerged, e.g., Java [27] [28] [29], C# [30] [31] and python [32] [33] design patterns. Adhering to these implementation-specific patterns enables the management, modeling and mitigation of complexity.

Reference [34] describes four primary symptoms of a degrading design: rigidity - a tendency of a software that makes it difficult to change; fragility - a characteristic that causes even a small change to break the system; immobility - a feature that makes it difficult to move code or modules around even within the same project; and viscosity - the increase in difficulty to make design-preserving changes to the software. A good design that follows OO principles and techniques, as per [35] [36], does not degrade, and the principles are as follows (also referred to as SOLID principles):

SRP: The Single-Responsibility Principle.

OCP: The Open-Closed Principle.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

LSP: The Liskov Substitution Principle.

ISP: The Interface-Segregation Principle and

DIP: The Dependency-Inversion Principle

4.2 | Architecture level

There are a few enterprise architecture analysis and formalization techniques, such as DoDAF, TOGAF and Zachman[37]. The following section covers these approaches briefly. They address how different systems must be aligned to meet an enterprise's or organization's needs[38]. Good architecture and design have a lasting impact on the complexity of a software system. However, these are extremely complex theories. One requires considerable experience with software architecture techniques to effectively practice them because they not only define how to build architectures but also suggest how to reorganize them correctly at the organization level (TOGAF). This process requires buy-in from many different organizations within an enterprise. Coordination among organizations within the enterprise is another logistical challenge.

4.2.1 | Enterprise architecture

Enterprise architectures (EAs) align enterprises to solve collective business and technology needs[38]. The various frameworks described below have a specific definition that loosely translates to this general meaning. EA has been recommended as a discipline to manage the complex amalgamation of IT and business[39] by ensuring a coherent software structure to provide a clear view of the organizational system and pave the way for change. EA has a considerable impact on the success of a software system. While EA does not solve the problem of software complexity directly, having a planned enterprise enables the management and mitigation of complex. Managing EA is the practice of continually improving the EA to control complexity and change[15]. Reference[40] establishes another dimension to EA intended at better management: technical EA - targeted at the betterment of the software development process; operational EA - targeted at smooth and efficient operation; and strategic EA - linking strategy and execution by driving enterprise strategies.

All EA frameworks propose three broad phases of automation: As-Is architecture definition phase, To-Be architecture planning phase, and a migration phase. The following are some of the leading enterprise architecture frameworks.

TOGAF - The Open Group Architecture Framework

Reference[38] describes TOGAF as an EA framework. They define it as a tool for assisting with acceptance, production, use, and maintenance of EA. TOGAF is an iterative process model backed by best practices and many reusable reference architecture assets.

ZEAF: Zachman Enterprise Architecture Framework

This is a framework for an information systems architecture. It was established by studying fields of engineering external to software, such as architecture, construction, and manufacturing, and hypothesizing by analogy a set of architectural representations for IT systems[41]. There are also approaches in ZEAF targeted at handling complexity, such as Complex Adaptive Systems; Cynefin - a framework that models complexity domains as Obvious, Complicated, Complex, Chaotic and Disorder and offers generic solutions to each of these; High Reliable Organizations; Systems Thinking; and Open Systems Theory[15].

EAP: Enterprise Architecture Planning

EAP was first published in 1992 and had experienced widespread adoption. It is considered one of the foundational works in the area of documentation of the system of systems and was later called EA practice[42]. It is used on its own or in combination with other EA frameworks like ZEAF[43].

DoDAF: Department of Defense Architecture Framework

This is a military-based EA framework. Reference[37 44 45] describes DoDAF as a common way of documenting and establishing DoD architectures. DoDAF has selected applicability to the military framework of an organization[44 46], with fundamental differences in the type of products, architectures, and military perspective in comparison to other EA frameworks.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

Gartner EA

Reference [47] [48] describes Gartner EA as follows: "Gartner EA Process Model provides organizations with a logical approach to developing an EA. It is a multi-phase, iterative and nonlinear model, focused on EA process development, evolution and migration, and governance, organizational and management sub-processes. It represents key characteristics and a synthesis of best practices of how the most-successful organizations have developed and maintained their EA."

FEA: Federated Approach to Enterprise Architecture

FEA is primarily an EA management tool. It is a federated approach for keeping the EA models up to date by having an EA repository designated to store a copy of model data from specialized architectures for EA [49] [50].

4.2.2 | Architecture styles

Choosing the right architecture style could significantly impact complexity. Reference [51] states "architecture styles narrow the solution space: First, styles define what elements can exist in an architecture (e.g. components, connectors). Second, they define rules for how to integrate these elements in the architecture". Different kinds of architecture styles identified by [52] [51] are as follows:

- Pipe-and-filter style for data-intensive systems
- Client-server style for synchronous systems.
- Blackboard style for data centered systems.
- Layered style for a higher level of abstraction.
- Interacting processes style for event-based systems.

Since styles exhibit known qualities, it is possible to estimate and address nonfunctional issues even before the system is built. However, the reason style choice can effect complexity is that there is no system or standard for style application. This is usually dependent on the architect or architecture team.

5 | ACHIEVING SAFETY

Software safety is focused on the software system and how to operate the system successfully, concentrating on the system alone, and on a mathematical analysis and a model-based approach. The correctness of the software does not guarantee the operational safety of the system [53]. Complexity makes achieving safety even more difficult since the emergent behavior might not occur under the traditional safety techniques.

5.1 | High-pressure steam engines and computer software

There have been many instances in history where humans have invented a potent technology that has significantly impacted the way things are carried out. Almost every time, the technology starts out and explodes into something with little or no control. It takes time, extensive research, awareness, regulation, and practice to gain control of and operate these technologies in a controlled environment. The high-pressure steam engine is one of these inventions. Looking at how steam power was controlled and harnessed from chaos could provide insight into software safety. After all, the computing advancements of today are synonymous with the technological breakthrough of steam engines in the 19th century.
In the 1800s, steam engines gained much use and, hence, popularity. Their widespread application very frequently resulted in disastrous accidents, killing and injuring many, and causing losses worth thousands of dollars, but they only became more widely used owing to their use and lack of a viable alternative. By the mid-1800s, in the US alone, steam engines were estimated to have caused 233 steamboat explosions killing 2,562 and injuring another 2,097, with property damage in excess of $3,000,000 [54]. The inferior materials used, low standard of workmanship, untrained professionals, and lack of quality control were believed to be the problems. Later, it was identified that boiler explosions were the main cause of most accidents. Boilers quickly became technologically advanced, but little was discovered about the steam pressure build-up in these boilers and other operational challenges of the boilers. While fixes were introduced in the form of safety valves and fusible lead plugs, they were not very successful. Lack of knowledge about steam boilers coupled with an underestimation of the working environment and quality of operators were the two main reasons for this. Only a Congress regulation in 1852 corrected the problems with steam engines, significantly reducing the number

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

of fatalities. The need to observe standards was realized, and organization and insurance companies were fully operationalized to strictly observe these protocols. Reference [54] states that "through the efforts of the American Society of Mechanical Engineers, uniform boiler codes were adopted". Drawing parallels with the lessons learned offers the following clues:

- Legislation regulating compliance in developing software that always operates safely and in a controlled manner was formed.
- It is advisable to start small and achieve small manageable wins, thereby resulting in higher-order complexities while applying lessons from previous experiences.
- A deeper understanding of the scientific foundation of our craft will significantly impact this quest for software safety.
- A better understanding of operational errors is crucial in understanding failures. The cognitive aspect is a less researched but important factor influencing safety.
- Organizations and managers need to be fully on-board and eager to offer safer software. This has been shown to be lacking in some of the major disasters of the past.
- The end users and engineers of software systems need to be involved in the safety design. Since they are the end users, they can offer a better insight into safety.
- Trained professionals and engineers with an in-depth knowledge of safety and software need to be involved in the development process.

5.2 | Systems-theoretic process analysis (STPA)

Software failures are not random, and software does not wear out like mechanical or hardware components [53]. Software can enforce or compromise system safety by putting it into a hazardous state [55]. Engineering software safety hence requires a good understanding of how the software operates in the system. The following are some of key areas of safety:

- Hazard analysis
- Safety-specific requirements elicitation
- Making safety part of the design objective
- Testing exclusively for safety
- Safety-specific protocol adherence
- Appropriate usage of models and design tools such as fault tree analysis, failure mode effective critical analysis, hazard and operability analysis, etc
- Safety in the commercial – Commercial off the shelf (COTS) - tools used
- Testing and evaluation
- Runtime monitoring and alerting

Safety is unique to each system. Safety for a piece of banking software is different from that for medical equipment. Each has its own safety protocols, and the onus is on the engineering team to identify the right needs and gain the required acumen for automating the system. While this might constitute a requirement problem, there is also an implementation consideration. If this has to be implemented reasonably well, there must be enough checks in place to ensure that the safety levels are maintained. Reference [53] through their research, proposed a safety engineering approach based on STPA, which is a systematic process for engineering safe software. According to them, this is divided into three activities: "(1) deriving software safety requirements at the system level; (2) constructing the safe behavior model of the software controller; (3) software safety verification performed with two complementary activities: (3.1) formally verifying software design and implementation against its safety requirements at the design and code levels, and generating safety-based test cases, and (3.2) testing the generated safety-based test cases using the safe behavior model of the software controller". This can be applied to newly developed software and software that is already in service.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

6 | ENSURING QUALITY

There are two broad definitions of quality: *conformance to specs* - A quality definition spec is formulated and ratified, and the software product is put through this spec for conformance; *meeting customer needs* - This has no formal or specific rules or guidelines. If a product matches customer needs, it is supposed to meet the expectations [56]. Quality is a widely discussed and researched topic. There are several materials that give us a good definition of quality; hence, analyzing them gives us good insight into quality measurement. The following are some of the most prominent, flexible, and qualitative ones:

Crosby: Quality is prescribed as an inherent feature to be delivered with zero defect rather than an external attribute.

Deming, Feigenbaum, and Ishikawa: There is a strong emphasis on customer satisfaction, with everything centered on it.

Shewhart: His view of quality is abstract, with quality being regarded as having objective and subjective qualities.

Some of the most prominent models of quality measurement are McCall's quality model, Boehm's model, FURPS, Bromey's model, ISO, IEEE, CMM, and Six Sigma. Although at a higher level it seems like two broad schools of thought exist, the real situation is actually more elusive and complex. Point of view plays a vital role in this matter, e.g., the above two schools of thought represent a transcendental and a product view of quality, but there is also an industry-specific and a value-based measure. A quantitative measure makes it easy to measure quality but might represent only a lean and narrow perspective. The transcendental view lacks a measure and hence is difficult to gauge and enforce.

Reference [57] compares excellent, average, and poor software results from the quality standpoint. It clearly shows that cost doubles between an excellent and poor software execution. Bypassing quality is counterproductive; it actually slows down the project. The main reason for schedule slippage is starting testing with many bugs. Finding and fixing bugs has been the leading cost drivers for the entire software industry. Idealistic approaches such as defect prevention, pretest inspections, and static analysis are fast, inexpensive, and most effective against these issues. Companies with excellent quality control have rigorous quality methods such as formal estimation before starting, full defect measurement and tracking. They also tend to have low levels of code cyclomatic complexity and high test coverage, i.e., >95% of the path and risk areas. Average projects do not determine the defects by origin, and hence, requirement and design defects are under-reported or unknown. Poor projects are categorized by the total omission of pretest defect removal; they have poor quality data; and maintenance requires 2 to 3 times the planned schedule. The best long-term strategy for achieving consistent excellence at high speed is to favor construction from certified reusable components. These eliminate over 80% of the construction cost and shorten schedules by more than 60%; however, they have tax implications. The following learning from [57] is a good step towards reducing complexity and increasing efficiency:

- A team software process and rational unified process have been the most effective for large (>10000 function points) projects.
- Software excellence has low defect potential (2.5/function point) and a high DRE (defect removal efficiency) of 99%.
- Bypassing quality control does not speed up a project but rather slows it down. Defect prevention, pretest, inspections, and static analysis are fast and inexpensive.
- Starting testing with many bugs is the main reason for enormous slippage. Finding and fixing bugs is the leading cost driver for the entire industry.
- A team software process and rational unified process are among the best methodologies for large projects, while agile is mediocre.
- Tracking defect by origin is the best way to track requirement and design defects.
- While it is possible to achieve these, moving toward libraries from well-formed standard reusable components seems to be the best way forward.

Continuing the discussion on the architecture and design level complexity vs the code level complexity in the previous section, design level complexity lays a foundation for analyzing quality in the next phases of the development life cycle [58] and is important for predicting the quality of the product being implemented. There is a relationship between design complexity and external quality attributes, i.e., a cognitive complexity. Reference [59] defines this as "the mental burden of the individuals who have to deal with the component, for example, the designer, developers, testers and maintainers". A complex system with a high degree of cohesion, coupling, inheritance and a massive code base increases the cognitive complexity by inducing undesirable external quality attributes. As seen in [58], fault proneness – the probability of finding a defect and reduced maintainability – and the effort needed to correct bugs, seem to be among the worse side effects of this. Other external quality attributes influencing complexity are as follows [58 60];

Defect density: Ratio of the total number of defects to the size and design complexity of the software.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

Vulnerability, Security: The number of defects due to security failure.

Testability: The ease with which a piece of software can be tested.

Volatility: Ratio of the number of enhancements to the design complexity.

Debugging cost: Effort and time required to fix a problem inherent to the class.

Development effort: Effort and time required to build a component.

Refactoring effort: Effort and time required to refactor a class.

There are different ways to ensure quality in software systems, and this is multidimensional. Verification and validation are two of the most important steps in ascertaining software quality. These are achieved through the process of software testing. Testing evaluates if the software meets the required guidelines, checks if the system under test responds to all kinds of inputs in the way it is required to respond, evaluates if the amount of time taken to process the inputs is well within the required time range, checks if the software can be installed and run in different environments, and validates if the software is usable. Through testing, software can be exposed to its live working environment, which gives us a sense of how the product reacts in production. Several use cases that were not thought through in the design phase can be exposed through testing. Making this part of the development cycle ensures that all issues, use cases, and environment scenarios are mocked and dry run as much as possible before a product is actually implemented in the real world. This is preemptive, increases confidence regarding a product, and uncovers scenarios that might have been missed.

7 | BUILDING RESILIENCE

We are heavily dependent on software; in some situations, it is necessary even for the air we breathe. Because we have back-up plans for everything, we absolutely need to have one for software failures. Fixing failures in critical systems calls for expertise in software and mathematics and an ability to remain calm under dire situations since much time may be required. Compared with any external threat, internal software failures are by far more concerning. One of the most significant ways of showing preparedness against software failures is practicing software resilience [61]. Resilience is the ability to achieve operation success in the direst situations; it is a system's ability to adapt to changing conditions while maintaining a fully functional operational state at all times to ensure habitability and engineering functionality. It is possible to plan well in advance for dire software scenarios, referred to as *resilience engineering*. This has to be practiced early in the development process and has to continue into the operations phase. It has safety as its core value and continually anticipates potential failure of the software. This not only prevents failure proactively but also prepares an organization to address failure scenarios. In addition to resilient software engineering, team-centric emergency software management is necessary to avert failure and maintain functionality in the interim.

In extremely critical missions such as those involving a spaceship, software failure is the greatest threat [62]. Software is involved in the operation of the entire ship, not only its safety systems. A software malfunction in critical systems such as those that produce breathable air can jeopardize the entire crew. External threats can be averted, e.g., an escape maneuver could be performed to avert danger, but such actions cannot be used against something internal such as corrupted software.

Software safety: This is focused on the software system and how to operate a system successfully without considering past or future learning, concentrating on the system alone, mathematical analysis, and the foundational model. Resilience and software safety are diametrically opposed to each other, not competing with each other but rather complementary aspects. Resilience is organization-centric and involves failure anticipation and foresight, while software safety is system-centric.

Reliability: This is the characteristic of a system that operates safely as desired under specified conditions for a required duration. Safety is one aspect of reliable software.

Failure obviation: This is the opposite of reliability. It involves studying and learning failure to increase our understanding of the development of anti-patterns that cause failure. Software safety on the other hand is related to the successful functioning of the software system, its patterns, and best practices [61].

Traditional software models take only safety and reliability into consideration. Newly introduced models include failure obviation and resilience, which are opposite yet complimentary. While one considers success and the specific system and builds around it, the other takes failure and the whole organization into consideration, builds anti-patterns, and learns from it. Safety and resilience together provide a measure, a model, importance, and guidance to a less fragile system. Reliability and failure obviation together provide the correct measure of availability and safety as well as techniques to avoid failures or, at worst, bounce back from failure quickly to attain a normal and safe stable state.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

Software safety cannot be improved if the software is not resilient; however, it is not the only thing that will improve resilience. Reliability also has to be improved. All the failure scenarios and anti-patterns need to be studied, and there should be better preparedness for failure scenarios. Organizational-level improvement is needed during the development of software and during its operation. Safety models use mathematical models and processes and employ extensive testing cycles to ensure that a system has a good degree of safety. Resiliency is what puts this procedure in the context of the bigger picture. The safety of a component is not enough to drive the whole system or an organization to have safety at its core. External forces and other failures will have a cascading effect and ultimately fail the system.

As we saw in the introduction, software failure can cause catastrophic damage and result in the loss of precious lives. The study of those failures shows that they are often only minor defects that snowball into large problems; a few lines of code is all it takes, as software is fragile. Increased fragility means increased susceptibility to failure; thus, our goal should be to produce software with a low degree of fragility. The fragility spectrum is a good measure, serving as a guideline to classify the fragility of a software system. It shows how this works toward the resilience or brittleness of a system. It has varying degrees of software imperfections along the x-axis. As we move toward the positive side of the axis, we tend toward more resilient (less fragile) software, which is what is needed. Toward the negative side of the x-axis is increasing brittleness i.e., fragile and unstable software [61].

8 | DISCUSSION

8.1 | Architecture patterns

In addition to the architecture styles described in section 4, there is another constantly evolving dimension to software architecture that offers predetermined solutions for common architecture problems and, hence, could be visualized as patterns. The SOA (service-oriented architecture), message-oriented architecture, file-/database-based integration, monolithic applications, microservice architecture, event-driven architecture, etc., are among the most widely used architecture patterns. Each pattern is a combination of previously mentioned architecture styles. A decade ago, SOA was considered a cutting-edge architectural pattern. In the last few years, it has been dethroned by microservice architecture combined with asynchronous architectures. Although experts suggest favoring a pattern that best fits the problem, a pattern in vogue normally finds the greatest applicability. The rise of the Internet of things, data science, and artificial intelligence and machine learning (AI & ML) inspired a surge of new architectures moving toward asynchronous integration.

8.2 | Software development methodology and quality

As seen in section 6, agile, while being popular, is not the best defense against quality issues in large projects; it is only reasonably successful [57]. A team software process and rational unified process are among the best methodologies for large projects. Pair programming and extreme programming are fairly more effective. This is an open list; there are many other areas of study, such as test-driven development and a microservice architecture combined with extreme programming, that have been shown to reduce bugs and produce high-quality deliverables at a fast pace.

8.3 | Human influence on quality

Humans are the main resources for software projects. Human factors can have a dramatic effect on the success of the software being developed, but studies on the human influence on quality and other aspects of software engineering have been sporadic [63]. The quality of people has been the prime driver for software quality and productivity and not the development process, which the software industry appears to be focusing on excessively. Reference [63] categorizes the following as the key sources influencing human factors.

Technological improvement and evolution: Research shows sophisticated tools coupled with adequate training could lead to a significant reduction in effort, while a basic tool would actually drive up the effort. While identifying the right tool and related training is a challenging task, more than 70% of the managers believe that a new appropriate tool would eventually become an enormous advantage.

Environment and personnel: Environment and people go hand in hand, more so in some of the recent development methodologies such as agile. Personnel competency, talent, skill, and communication are of utmost importance since working close to one another is among the methodology manifesto. Another study by the authors indicated that the expertise of the project team also affected the management's response in terms of being predictable, proactive, and increasingly supportive.

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

9 | CONCLUSIONS

Low complexity, safety, excellent quality, and resilience are among the key characteristics of a successful and safe software system. While complexity is not a desirable characteristic, it cannot be separated from software systems. The alternative is to monitor and mitigate the problems with complexity using the available techniques described in this paper. There are ways to achieve safety in this ever-changing, complex software engineering environment. Maintaining safety, quality, and resilience are of the utmost importance; otherwise, the failure of a running software could be disastrous and even catastrophic depending on the nature of the business involved.

References

1. Kapadia N, Linn M. Option Spreads and the Uncertain Cost of Equity Liquidity: Evidence from the Knight Capital Trading Glitch. *SSRN Electronic Journal* 2019. doi: 10.2139/ssrn.3517429

2. Anthopoulos L, Reddick CG, Giannakidou I, Mavridis N. Why e-government projects fail? An analysis of the Healthcare.gov website. *Government Information Quarterly* 2016; 33(1): 161–173. doi: 10.1016/j.giq.2015.07.003

3. Wagstaff J. Mt. Gox bitcoin debacle: huge heist or sloppy glitch?. http://www.reuters.com/assets/print?aid=USBREA1R0Y720140228; 2014.

4. Arnold DL. The Patriot Missile Failure. http://www-users.math.umn.edu/~arnold//disasters/patriot.html; 2000.

5. BBC News . BBC NEWS | UK | Computer problems hit CSA payouts. http://news.bbc.co.uk/2/hi/uk_news/3235394.stm; 2003.

6. Lions JL. Ariane 5 Flight 501 Failure. http://zoo.cs.yale.edu/classes/cs422/2010/bib/lions96ariane5.pdf; http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf; 1996.

7. Isbell D, Savage D. Mars Climate Orbiter Failure Board Releases Report. https://mars.nasa.gov/msp98/news/mco991110.html; http://mars.jpl.nasa.gov/msp98/news/mco991110.html; 1999.

8. Kulyapin AV, Sakhautdinov VG, Temerbulatov VM, Becker WK, Waymack JP. Bashkiria train-gas pipeline disaster: a history of the joint USSR/USA collaboration. *Burns* 1990; 16(5): 339–342. doi: 10.1016/0305-4179(90)90005-H

9. Bondar S, Hsu JC, Pfouga A, Stjepandić J. Agile digital transformation of System-of-Systems architecture models using Zachman framework. *Journal of Industrial Information Integration* 2017; 7: 33–43. doi: 10.1016/j.jii.2017.03.001

10. Brooks FP. Essence and Accidents of Software Engineering. *Computer* 1987; 20(4): 10–19. doi: 10.1109/MC.1987.1663532

11. Antinyan V. Revealing the Complexity of Automotive Software. *In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA. ACM, New York, NY, USA* 2020(March). doi: 10.1145/3368089.3417038

12. Ghazarian A. A Theory of Software Complexity. *2015 IEEE/ACM 4th SEMAT Workshop on a General Theory of Software Engineering* 2015: 29–32. doi: 10.1109/GTSE.2015.11

13. Clarke P, O'Connor RV, Leavy B. A complexity theory viewpoint on the software development process and situational context. *Proceedings of the International Workshop on Software and Systems Process - ICSSP '16* 2016: 86–90. doi: 10.1145/2904354.2904369

14. Cardoso A, Crespo R, Kokol P. Two different views about software complexity. https://www.researchgate.net/profile/Peter_Kokol/publication/2533928_Two_Different_Views_about_Software_Complexity/links/0fcfd50b10ade42705000000/Two-Different-Views-about-Software-Complexity.pdf; 2000.

15. Lapalme J, Gerber A, Van der Merwe A, Zachman J, Vries MD, Hinkelmann K. Exploring the future of enterprise architecture: A Zachman perspective. *Computers in Industry* 2016; 79: 103–113. doi: 10.1016/j.compind.2015.06.010

16. Paige RF, Brooke PJ, Ge X, Power CDS, Burton FR, Poulding S. Revealing Complexity through Domain-Specific Modelling and Analysis. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2012; 7539: 251–265. doi: 10.1007/978-3-642-34059-8_13

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

17. Medvidovic N, Rosenblum DS, Redmiles DF, Robbins JE. Modeling software architectures in the unified modeling language. *ACM Transactions on Software Engineering and Methodology* 2002; 11(1): 2–57. doi: 10.1145/504087.504088

18. Podgorelec V, Heričko M. Estimating software complexity from UML models. *ACM SIGSOFT Software Engineering Notes* 2007; 32(2): 1–5. doi: 10.1145/1234741.1234763

19. Seemann J, Gudenberg vJW. OMG Systems Modeling Language (OMG SysML™). *Informatik-Spektrum* 2012; 21(2): 89–90. doi: 10.1007/s002870050092

20. Cormen TH, Leiserson CE, Rivest RL, Stein C. *Introduction to Algorithms, Third Edition (International Edition)*. ISBN: 978-0-262-03384-8: MIT Press. 3rd ed. 2009

21. Halstead MH. Natural Laws Controlling Algorithm Structure?. *ACM SIGPLAN Notices* 1972; 7(2): 19–26. doi: 10.1145/953363.953366

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

23. Henry S, Kafura D. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering* 1981; SE-7(5): 510–518. doi: 10.1109/TSE.1981.231113

24. Chidamber S, Kemerer C. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering* 1994; 20(6): 476–493. doi: 10.1109/32.295895

25. Gamma E, Helm R, Johnson R, Vlissides J. Design Patterns: Abstraction and Reuse of Object-Oriented Design. *Software Pioneers* 2002: 701–717. doi: 10.1007/978-3-642-59412-0_40

26. Abdelaziz T, Sedky A, Rossi B, Mostafa MSM. Identification and Assessment of Software Design Pattern Violations. *Cornell University* 2019; 1(2): 6–13. doi: 10.21608/fcihib.2019.107517

27. Sarcar V. *Java Design Patterns. A Tour of 23 Gang of Four Design Patterns in Java*. ISBN: 978-1-4842-1801-3: Springer . 2016.

28. Cooper JW. *The Design Patterns Java Companion*. ISBN: 978-0201485394: Addison-Wesley . 1998.

29. Hannemann J, Kiczales G. Design pattern implementation in Java and aspectJ. *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications - OOPSLA '02* 2002: 161. doi: 10.1145/582419.582436

30. Derezińska A, Byczkowski M. Enhancements of Detecting Gang-of-Four Design Patterns in C# Programs. *Advances in Intelligent Systems and Computing* 2019; 852: 277–286. doi: 10.1007/978-3-319-99981-4_26

31. Martin R. *Agile principles, patterns, and practices in C# (Robert C. Martin)*. ISBN: 978-0-13-185725-4: Prentice Hall . 2006.

32. Badenhorst W. *Practical Python Design Patterns*. ISBN: 978-1-4842-2679-7: Apress . 2017

33. Giridhar C. *Learning Python Design Patterns Second Edition*. ISBN: 978-1-78588-803-8: Pakt Publishing Ltd . 2013.

34. Martin R. Design principles and design patterns. http://staff.cs.utu.fi/staff/jouni.smed/doos_06/material/DesignPrinciplesAndPatterns.pdf; 2000.

35. Chebanyuk E, Markov K. An approach to class diagrams verification according to SOLID design principles. *MODELSWARD 2016 - Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development* 2016: 435–441. doi: 10.5220/0005830104350441

36. Martin R. *Agile Software Development Principles, Patterns, and Practices*. ISBN: 978-0135974445: Prentice Hall . 2002.

37. Bankauskaite J. Comparative analysis of enterprise architecture frameworks. *CEUR Workshop Proceedings* 2019; 2470: 61–64. doi: http://ceur-ws.org/Vol-2470/p19.pdf

38. The Open Group . *TOGAF ® VERSION 9.1-A POCKET GUIDE*. ISBN: 978 90 8753 678 7: The Open Group. 5th ed. 2016.

39. Korhonen JJ, Halen M. Enterprise Architecture for Digital Transformation. *2017 IEEE 19th Conference on Business Informatics (CBI)* 2017: 349–358. doi: 10.1109/CBI.2017.45

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

40. Bui QN. Evaluating enterprise architecture frameworks using essential elements. *Communications of the Association for Information Systems* 2017; 41: 121–149. doi: 10.17705/1cais.04106

41. Zachman JA. The Framework for Enterprise Architecture: Background, Description and Utility by: John A. Zachman. *Retrieved from: https://www.zachman.com/resources/ea-articles-reference/327-the-framework-for-enterprise-architecture-background-description-and-utility-by-john-a-zachman* 2016: 1–5.

42. Spewak S, Tiemann M. Updating the Enterprise Architecture Planning Model. https://eapad.dk/wp-content/uploads/2012/02/spewak-tiemann2006.pdf; 2006.

43. Hari Supriadi ST, Endang Amalia ST. University's enterprise architecture design using enterprise architecture planning (EAP) based on the Zachman's framework approach. *International Journal of Higher Education* 2019; 8(3): 13–28. doi: 10.5430/ijhe.v8n3p13

44. Amissah M, Handley HA. A process for DoDAF based systems architecting. *10th Annual International Systems Conference, SysCon 2016 - Proceedings* 2016. doi: 10.1109/SYSCON.2016.7490649

45. Dandashi F, Siegers R, Jones J, Blevins T. The Open Group Architecture Framework (TOGAF) and the US Department of Defense Architecture Framework (DoDAF). https://apps.dtic.mil/sti/pdfs/AD1107103.pdf; 2006.

46. Shirazi H. A Uniform Method for Evaluating the Products of DoDAF Architecture Framework Social Network Analysis View project Smart Organization View project. https://www.researchgate.net/publication/313880976; 2009.

47. Ansyori R, Qodarsih N, Soewito B. A systematic literature review: Critical Success Factors to Implement Enterprise Architecture. *Procedia Computer Science* 2018; 135: 43–51. doi: 10.1016/j.procs.2018.08.148

48. Bittler RS, Kreizman G. Gartner Enterprise Architecture Process: Evolution 2005. https://www.gartner.com/doc/486246/gartner-enterprise-architecture-process-evolution; 2005.

49. Sabau AR, Hacks S, Steffens A. Implementation of a continuous delivery pipeline for enterprise architecture model evolution. *Software and Systems Modeling* 2020: 1–29. doi: 10.1007/s10270-020-00828-z

50. Fischer R, Aier S, Winter R. A federated approach to enterprise architecture model maintenance. *Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures - Concepts and Applications, EMISA 2007* 2007; 2(2): 9–22. doi: 10.18417/emisa.2.2.2

51. Galster M, Eberlein A, Moussavi M. Systematic selection of software architecture styles. *IET Software* 2010; 4(5): 349–360. doi: 10.1049/iet-sen.2009.0004

52. Joanna MR, Bipin I. A Blackboard System for Generating Poetry. *Computer Science* 2016; 17(2): 265. doi: 10.7494/csci.2016.17.2.265

53. Abdulkhaleq A, Wagner S, Leveson N. A Comprehensive Safety Engineering Approach for Software-Intensive Systems Based on STPA. *Procedia Engineering* 2015. doi: 10.1016/j.proeng.2015.11.498

54. Leveson NG. High-Pressure Steam Engines and Computer Software. *Computer* 1994; 27(10): 65–73. doi: 10.1109/2.318597

55. Lutz RR. Software engineering for safety. *Proceedings of the conference on The future of Software engineering - ICSE '00* 2000: 213–226. doi: 10.1145/336512.336556

56. Berander P, Damm LO, Eriksson J, et al. Software quality attributes and trade-offs. https://www.researchgate.net/profile/Jeanette_-Eriksson/publication/238700270_Software_quality_attributes_and_trade-offs_Authors/links/543fa4550cf2f3e82851eef5/Software-quality-attributes-and-trade-offs-Authors.pdf; 2005.

57. Jones C. Achieving software excellence. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.434.1541&rep=rep1&type=pdf; 2014.

58. Nguyen-Duc A. The Impact of Software Complexity on Cost and Quality - A Comparative Analysis Between Open Source and Proprietary Software. *International Journal of Software Engineering & Applications* 2017. doi: 10.5121/ijsea.2017.8202

59. Briand LC, Wuest J, Ikonomovski SV, Lounis H. Investigating quality factors in object-oriented designs: An industrial case study. *Proceedings - International Conference on Software Engineering* 1999: 345–354. doi: 10.1145/302405.302654

Preprints (www.preprints.org) | NOT PEER-REVIEWED | Posted: 6 January 2021 doi:10.20944/preprints202101.0113.v1

60. Edgren R. The Little Black Book on Test Design. http://www.thetesteye.com/papers/TheLittleBlackBookOnTestDesign.pdf; 2012.

61. Dulo DA. Resilience engineering in critical long term aerospace software systems: A new approach to spacecraft software safety. https://ui.adsabs.harvard.edu/abs/2014JBIS...67..150D; 2014.

62. Dulo DA. Software or the Borg : A Starship ' s Greatest Threat?. https://www.space.com/29509-software-borg-starship-greatest-threat.html; 2015.

63. Fernández-Sanz L, Misra S. Influence of Human Factors in Software Quality and Productivity. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 2011; 6786(PART 5): 257–269. doi: 10.1007/978-3-642-21934-4_22

CONFLICT OF INTEREST

None