encyclopedia

MDPI

Entry

Fatal Software Failures in Spaceflight

Christian R. Prause [1,2,*], Ralf Gerlich [3,4] and Rainer Gerlich [4]

[1] Digitalization, Software and AI, German Space Agency at DLR, 53227 Bonn, Germany
[2] Distance Learning, Team Academic Lecturer CSc, DaSc & AI, Campus Bad Honnef, IU International University of Applied Sciences, 53604 Bad Honnef, Germany
[3] Faculty of Computer Science, Furtwangen University, 78120 Furtwangen, Germany; ralf.gerlich@hs-furtwangen.de
[4] Dr. Rainer Gerlich System and Software Engineering (GSSE), 88090 Immenstaad am Bodensee, Germany; rainer.gerlich@gsse.biz
* Correspondence: prause@acm.org

Definition: Space exploration has become an integral part of modern society, and since its early days in the 1960s, software has grown in importance, becoming indispensable for spaceflight. However, software is boon and bane: while it enables unprecedented functionality and cost reductions and can even save spacecraft, its importance and fragility also make it a potential Achilles heel for critical systems. Throughout the history of spaceflight, numerous accidents with significant detrimental impacts on mission objectives and safety have been attributed to software, although unequivocal attribution is sometimes difficult. In this Entry, we examine over two dozen software-related mishaps in spaceflight from a software engineering perspective, focusing on major incidents and not claiming completeness. This Entry article contextualizes the role of software in space exploration and aims to preserve the lessons learned from these mishaps. Such knowledge is crucial for ensuring future success in space endeavors. Finally, we explore prospects for the increasingly software-dependent future of spaceflight.

Keywords: spacecraft failures; software engineering; systems engineering; software glitch; spaceflight mishap; software quality

check for updates

Citation: Prause, C.R.; Gerlich, R.; Gerlich, R. Fatal Software Failures in Spaceflight. *Encyclopedia* **2024**, *4*, 936–965. https://doi.org/10.3390/encyclopedia4020061

Academic Editors: Ramesh Agarwal and Raffaele Barretta

Received: 29 March 2024
Revised: 14 May 2024
Accepted: 30 May 2024
Published: 12 June 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction to Spaceflight and Software

Space exploration and society: Space exploration stands as a pivotal industry for shaping the future. Its significance escalates year after year. Space technologies, serving as catalysts for innovation, permeate nearly every facet of modern life. Space exploration has become indispensable as an integral part to essential sectors like—just naming a few—digitalization, telecommunications, energy, raw materials, logistics, environmental, transportation, and security technologies. Spillover effects of advancements in space exploration extend into diverse realms, making it the foundation of innovative nations. The demand for satellites, payloads, and secure means to transport them into space is booming. Besides the traditional institutional spaceflight organizations, small and medium-sized enterprises and startups increasingly characterize the commercial nature of space exploration. Projections indicate that by 2030, the global value creation in space exploration will reach one trillion euros (for comparison, the current global value creation in the automotive industry stands just below three trillion euros). Public investments are deemed judicious, as each euro for space exploration (e.g., Germany: 22 euros per capita; France: 45; USA: 76, or 166 including military expenditures) is expected to generate four-fold direct and nine-fold indirect value creation [1].

Space mission peculiarities: Preparing a mission, engineering and manufacturing the system, and finally operating it can take decades. Many missions rely on a single spacecraft that is a one-of-a-kind (and also first-of-its-kind) expensive device comprising custom-built,

state-of-the-art components. For many missions, there is no second chance if the launch fails or if the spacecraft is damaged or lost, for whatever reason, e.g., because a second flight unit is never built, (cf. [2]) because of long flight time to a destination, or because of a rare planetary constellation that occurs only every few decades. Small series launchers, multi-satellite systems, or satellite constellations that naturally provide some redundancy can, of course, be exceptions to some degree. Yet failure costs can still be very high, and the loss of system parts can still have severe consequences.

Space supply chains: Supply chains of spaceflight are known to be large and highly complex. They encompass multiple tiers of suppliers and other forms of technology accommodation, and include international, inter-agency, and cross-institutional cooperation, stretching over governmental agencies, private corporations, research institutions, scientific principal investigators, subcontractors, and regulatory bodies. The integration of advanced technologies, stringent quality control measures, and the inherent risks associated with space exploration add further layers of intricacy to these supply chains.

Space technology: A typical spaceflight system is divided into different segments (see Figure 1). The space segment is the spacecraft, which typically consists of an instrument that provides the functional or scientific payload and of the satellite platform that provides the flight functions. The space segment is connected to its ground segment via radio links. Both segments are usually designed and built in parallel as counter-parts for one another. Services are provided to the user segment indirectly through the ground segment (e.g., Earth observation data) or directly through respective devices (e.g., navigation signals, satellite television). Finally, the launch segment, actually a spacecraft in its own right, is the rocket that takes the space segment into space.

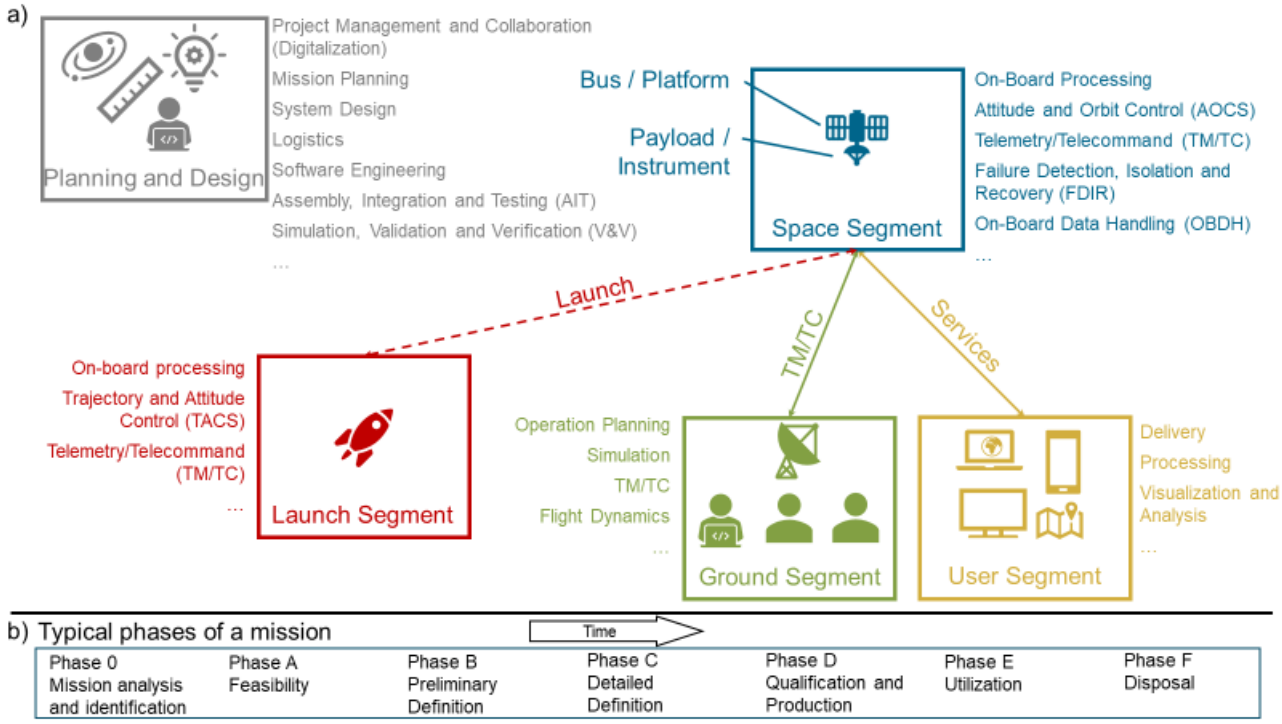


Figure 1. (**a**) Software as an indispensable part of a space system (i.e., space, launch, ground and user segments) and for space project execution. As a subsystem, it assumes vital functions in all segments, acts as "glue" for the different parts, and is essential for engineering activities (cf. [3,4]). The different segments are shown in different colors. (**b**) Typical project execution phases of a mission from left to right.

The role of software for spaceflight: Today, space exploration cannot be imagined without software across all lifecycle phases and in all of its segments and subsystems. Software is the "glue" [5] that enables the autonomous parts of systems of systems to work together, re-

gardless of whether these parts are humans, spacecraft components, tools, or other systems. The digitalization of engineering, logistics, management, development, and other processes is only possible through software. Space exploration is renowned for huge masses of data that can only be handled and analyzed through software. The agile development mindset, originating from and heavily relying on software, is the cornerstone of the New Space movement and its thriving business (cf. [6]). As the brain of each mission, software grants spacecrafts their ability to survive autonomously, automatically and instantly reacting to adverse environmental effects and subsystem failures by detecting, isolating, and recovering from failures (FDIR). Moreover, software updates are currently the only way to make changes to spacecraft after launch, e.g., in order to adapt to new objectives or hardware failures. Harland and Lorenz report numerous missions like TIMED (Thermosphere Ionosphere Mesosphere Energetics and Dynamics), TERRIERS (Tomographic Experiment using Radiative Recombinative Ionospheric EUV and Radio Sources), and ROSAT (Röntgensatellit) with problems that were rescued through inflight software patches. Occurrence of failures—in hardware or software—is the norm [7]. Eberhard Rechtin, a respected authority in aerospace systems, (and others, e.g., [8]) attested that software has great potential to save on mission and hardware costs, to add unprecedented value, and to improve reliability [9]. Since the ratio between hardware and software costs in a mission has shifted from 10:1 to 1:2 since the 1980s, Rechtin predicted in 1997 that software would very soon be at the center of spacecraft systems [9], which the chair of the NASA Aerospace Safety Advisory Panel would later confirm:

> "We are no longer building hardware into which we install a modicum of enabling software, we are actually building software systems which we wrap up in enabling hardware. Yet we have not matured to where we are uniformly applying rigorous systems engineering principles to the design of that software." (Patricia Sanders, quoted in [10])

Distinctive features of flight software: Quite heterogenous software is used in the various segments of a spaceflight system. While software for planning, design, the on-ground user segment, and data exploitation is more like a common information system, software executed onboard a spacecraft, in the launch and space segments, has its own peculiarities. Flight software exhibits the following qualities:

- Lacks direct user interfaces, requiring interaction through uplink/downlink and complicating problem diagnosis;
- Manages various hardware devices for monitoring and control autonomously;
- Runs on slower, memory-limited processors, demanding specialized expertise from engineers;
- Must meet timing constraints for real-time processing. The right action performed too late is the same as being wrong [11].

The role of software engineering: The NASA study on flight software complexity describes an exponential growth of a factor of ten every ten years in the amount of software in spacecraft, acting as a "sponge for complexity" [11]. At the same time, the cost share of software development in the space segment has remained rather constant [11]. Experiences with the reuse of already-flown software and integration of pre-made software components (or COTS, commercial off the shelf) software are "mixed", i.e., good and bad, as reuse comes with its own problems [9,11,12]. If rigorously analyzing modified software is prohibitively expensive or beyond the state of the art, then complete rewriting may be warranted [13]. Often software is not developed with reuse in mind [14], which limits its reuse. In consequence, this means that software development on a per-mission basis is important and that software engineers face a rapidly increasing density of software functionality per development cost, further emphasizing the importance of and advances in software engineering. In fact, software engineering was invented as a reaction to the software crisis, which basically says bigger programs, bigger problems (cf. [15]). Nonetheless, software engineering struggles to be accepted by traditional space engineering disciplines.

Only few project personnel, especially in management, understand its significance [16]. Some may see it as being little more than a small piece in a subsystem or a physicist's programming exercise:

> "Spaceflight technology is historically and also traditionally located in the area of mechanical engineering. This means that software engineering is seen as a small auxiliary discipline. Some older colleagues do not even use the word 'software', but electronics." (Jasminka Matevska, [17])

A side note on software product assurance: Together with project management and engineering, product assurance is one of three primary project functions in a spaceflight project. Product assurance focuses on quality; simplifying a bit, it aims at making the product reliable, available, maintainable, safe (RAMS), and, more recently, also secure. It observes and monitors the project, witnesses tests, analyzes, and recommends, but it does not develop or test the product, manage people, or set product requirements. Instead, it has organizational and budgetary independence and reports to highest management only. There are direct lines of communication between the customer's and their suppliers' product assurance personnel. For product assurance and engineering, there are software forms called software engineering and software product assurance (see [2]). However, it is important to note that these are organizational roles, whereas both their technical backgrounds are software engineering. So, when we speak of software engineering here, we mean the technical profession, not the organizational role. Both roles are essential to mission success.

Software cost: The flight software for major NASA missions like the Mars Exploration Rover (launched in 2003) or Mars Reconnaissance Orbiter (launched in 2005) had roughly 500,000 source lines of code (SLOCs) [11]. As a rule of thumb, a comprehensive classic of space system engineering [18] calculates a cost of $350 (ground) to $550 (unmanned flight) for development per new SLOC, while re-fitted code has lower cost. These costs already include efforts for software quality assurance. Another important cost factor of software is risk. As famous computer scientist C.A.R. Hoare noted:

> "The cost of removing errors discovered after a program has gone into use is often greater, particularly [. . . when] a large part of the expense is borne by the user. And finally, the cost of error in certain types of program may be almost incalculable—a lost spacecraft, a collapsed building, [. . .]." [19]

Software risks: The amount of software in space exploration systems is growing. More and more critical functions are entrusted to software, the spacecraft's "brain" [20]. Unsurprisingly, this means that sometimes software dooms large missions, causing significant delays or outright failures [16]. A single glitch can destroy equipment worth hundreds of millions of euros. According to Holzmann [21], a very good development process can achieve a defect rate as low as 0.1 residual defects per 1000 lines of code. Given the amount of code in a modern mission, there are hundreds of defects lingering in the software after delivery. More importantly, however, there are countless ways in which these defects can contribute to Perrow-class failures (cf. [21,22]). In increasingly complex safety-critical systems, where each defect is individually countered by carefully designed countermeasures, this "conspiring" of smaller defects and otherwise benign events can lead to system failures and major accidents, i.e., resulting in the loss of a spacecraft system, rendering the mission goals unreachable, or even causing human casualties [21]. But it is not only benign errors. In the spirit of Belady, the necessary software-encoded human thoughts that allow the mindless machine to act on our behalf are missing [5]. This, according to Prokop (see Section 2), appears to happen quite frequently [23]. Leveson concludes that software "allows us to build systems with a level of complexity and coupling that is beyond our ability to control" [13]. However, MacKenzie notes that software (across different domains) "has yet to experience its Tay Bridge disaster: a catastrophic accident, unequivocally attributable to a software design fault, in the full public gaze. But, of course, one can write that sentence only with the word "yet" in it" [24].

System view of software failures: Due to system complexity and the interplay of defects and events, failures are often difficult to attribute to specific single sources. Furthermore, spacecraft failures are viewed from a spaceflight technology perspective, which, of course, is not wrong per se. But in this view, as discussed above, software is often only seen as "a modicum of enabling software". For example, the GNC combines the ACS, the propulsion system, and software for on-orbit flight dynamics. (There are many different terms associated with this group of subsystems, and terms actually used by different authors vary. A selective definition is not attempted here, but Appendix A lists several terms and their possible relationships.) The ACS again includes sensors, actuators, and software. The GNC may fail from software, hardware, or a sub-subsystem like the ACS. But in the system view, analysis often only concludes that the GNC failed. That the reason is a software defect of the ACS is only recognized when viewed from a software perspective, or when asking "why" often enough. But there is also the opposite case: there are also subsystems that sound more like software, e.g., onboard data handling (OBDH), or seemingly obvious failure attributions like a "computer glitch" (e.g., Voyager 1, humanity's oldest and most distant space probe, was recently jeopardized by computer hardware failure [25], a stuck bit, and now is about to be software-patched). Failures in these subsystems are too easy to attribute to software upon superficial analysis, although they can have hardware or system design causes.

Types of software failures: There are many different kinds of software-related failures, and many types often leave room for interpretation. Of course, there are the classical programming errors, e.g., syntax errors, incorrect implementation of an algorithm, or a runtime failure crashing the software. In most cases, software does not fail, but it does something wrong. It functions according to its specification, which, however, is wrong in the given situation [13]. Is this a software failure, a design fault, both, none? Are validation and verification activities to blame that they did not find the problem? Or was configuration management negligent? And then, MacKenzie finds that human–computer interaction is more deadly than software design faults (90:3) [24]. Is it an operation failure if the human–computer interface is bad, or if bad configuration parameters are not protected? Is it a software failure if code or software-internal interfaces are written poorly, badly documented, or misleading to other developers? In spaceflight, the natural environment and hardware failures cause random events that software should be able to cope with, for instance, by rebooting, by isolating the failure, etc. Is it a software failure if software is not intelligent enough to handle it correctly, or if it does not try that at all? In fact, there is no commonly accepted failure taxonomy; a classic attempt at a taxonomy is, for example, that of Avizienis et al. [12]. Our collection of spaceflight mishaps shows how difficult an attribution can sometimes be.

Contribution of this Entry: Newman [22] notes that it is only human nature that a systems engineer will see the causes of failure in system engineering terms, a metallurgist will see the cause in metallurgy, etc. In this Entry, we therefore look at notable space exploration accidents from a software perspective, which is relatively underrepresented in space exploration. We focus on the following:

- Revisiting studies that investigated the role of software in a quantitative, or at least quantifiable way, in order to give context and explain why qualitative understanding of accidents is important (see Section 2);
- Reanalyzing the stories and contexts of selected software-related failures from a software background. We provide context, background information, references for further reading, and high-level technical insights to allow readers to make their own critical assessment of whether and how this incident relates to software engineering. This helps software practitioners and researchers grasp which areas of software engineering are affected (see Section 3);
- Concluding this Entry with an outlook on growing software-related concerns (see Section 4).

Understanding the causes and consequences of past accidents fosters a culture of safety and continuous improvement within the spaceflight engineering community. Anecdotal stories of accidents provide valuable insights into past failures, highlighting areas of concern, weaknesses in design or procedures, and lessons learned. They improve our knowledge and understanding of how software has contributed to space exploration accidents, which are important tools for success.

2. Context of Software Failures in Space Exploration

Several studies have quantitatively analyzed failures in spaceflight. Not all studies (e.g., [26,27]) but some of them explicitly include statistics for software-based failures (see Table 1). The studies find that software is responsible for 3 to 33 percent of failures, with most values close to 10%.

Table 1. Quantitative analyses of reasons for spacecraft failures. This table only shows the results regarding software.

Who?	What?	# of Failures Due to Software
Biswal & Annavarapu [28].	Failed missions to Mars.	4 out of 33 (~12%)
Chang [29]	US and non-US launch failures from 1984 to 1994. Software is not reported separately but as a part of avionics in 1 and, respectively, 4 failures.	1 out of 14 US launch failures (7%) 1 out of 29 non-US launch failures (3%)
Fernández et al. [30]	TACS failures (18%) in a total of 57 failed launches from 2016 to 2021.	2 out of 10 TACS failures (20%) (Reasons for another 4 failures are unknown.)
Gorbenko et al. [31]	Reasons (software, hardware, radio, gyroscope, etc.) for failures in 663 launch attempts and 1060 launched spacecraft from 2000 to 2009.	5 out of 33 fatal launch failures (15%) 1 out of 14 wrong orbit placements (7%)
		3 out of 49 fatal spacecraft failures (6%) 18 out of 54 partial spacecraft failures (33%)
Kattakuri & Panchal [32]	Analysis of 100 payloads aboard 91 space missions. Failures due to "programming errors" (vs. AD&C, design, communication, etc.) from 2009 to 2019.	8 out of 100 mission failures (8%)
Newman [22]	Analysis of 50 selected fatal space system failures from 1960 to 2000 (multiple causes regarded).	4 failures due to software design (8%) 8 failures due to software V&V (16%)
Robertson & Stoneking [33]	Failure source of 35 [(a)] GN&C anomalies (accounting for 29% of all recorded anomalies) in 764 spacecraft from 1990 to 2001.	1 out of 14 fatal GN&C failures [(b)] (7%) 3 out of 38 partial GN&C failures [(b)] (8%)
Tafazoli [34]	Analysis of 156 failures of 129 spacecraft (from over 4000 spacecraft in total) from 1980 to 2005.	6% of spacecraft failures 6%/4% [(c)] of AOCS failures 12%/7% [(c)] of CDH and TTC failures

Note: [(a)]: In this publication, GN&C (guidance, navigation, and control system) anomalies include ACS (attitude control system), propulsion, ground operations, and software involving flight dynamics. Robertson and Stoneking state on their Page 6 that 35 anomalies were analyzed. However, their Table 4 lists only 34 anomalies. All GN&C anomalies in their Figure 10 sum up to 38. The paper does not explain whether double entries are possible. [(b)]: Design, verification, operations, and unknown are listed as separate cause categories besides software. It is unclear whether software problems might also be included in these categories. [(c)]: Software failures are defined to encompass mainly erroneous commands and software flaws. Tafazoli provides two numbers for software: software as a failure type (vs. electrical, mechanical, and unknown) and software errors as reasons for component failures. Both values are presented above in this order.

However, quantitative analysis of failure causes has to be taken with a grain of salt. As an informative example, consider Jacklin's list [35] of small satellite failures. First, it contains a high proportion of satellites with an unclear fate ("presumed mission failure"), which might be a particularity of the small satellite domain. But also, commercial companies [34] and launch vehicle developers [30] are reluctant to release information to the public. More secretive nations or organizations usually do not even publish information when that they launch something, further adding to the gap in the knowledge.

Second, Jacklin's list [35] is interesting because the investigated failures are summarized in one to two sentences. Reasons mentioned are, for instance, "computer data bus hung", "failure of attitude control system", or "failure of the flight control system". Yet it remains unclear what the role of software in these accidents is. (Admittedly, the report never intended to analyze software involvement.) The example just highlights that a software perspective on space system failures is needed. Many quantitative analyses do not preserve traceable information and rationale for why an accident was or was not counted as a software failure. Sometimes traceability information is available, but rationale is missing. This gap can be filled with qualitative information on the accidents.

Prokop recently analyzed software failures in 31 spaceflight systems and 24 other industries' systems. Only 18% of failures are reported as "traditional" computer science/programming-related in nature, and none resulted from programming language, compiler, development tool, or operating system errors. But a significant 40% of failures originated from the absence of code; i.e., the software could not handle a certain situation because the respective code was not implemented. Often, respective requirements were missing [23].

Similarly, Dvorak [11]—summarizing studies from US government institutions—comes to the following top seven issues with software: (i) inadequate dealing with requirements, (ii) lack of software engineer participation in system decisions, (iii) deficient management of the software life cycle, (iv) insufficient software engineering expertise, (v) costly but ineffective software verification techniques, (vi) failing to ensure the as-desired execution of complex and distributed software, and (vii) inadequate attention to impacts of COTS and reuse. Failure mitigation strategies follow immediately from these findings: (i) enforcing effective software requirements management, (ii) participation of software engineering in system activities, (iii) a culture of quantitative planning and management, (iv) building qualified talent to meet government and industry needs, (v) guidance and training to improve product quality across the life cycle, (vi) development of assurance methods for software, and (vii) improvement of guidelines for COTS use [11].

3. History of Notable Software Failures in Space Exploration

Spacecraft failures are hard to inspect, since the hardware is usually unreachable. Furthermore, the systems are complex, requiring a multi-disciplinary investigation. The reports that follow from such assessments reflect the system's complexity. A straight logical path to any single root cause is difficult to obtain [22]. Based on public reports, we re-tell the stories of notable failures from the software perspective so that software engineers can learn from them.

3.1. Nedelin Disaster—A Not-So-Software Failure of Spaceflight? (1960)

During the Cold War, soviet leaders were pushing the development of ballistic missiles, including the R-16 intercontinental missile. A design and operating problem with a "programmed sequencer" was "the last and fateful error in a long chain of events that set the stage for the biggest catastrophe in the peacetime history of missile technology" [36], exploding a rocket prototype on the launch pad and killing over 100 persons (between 101 and 180) [36].

Boris Chertok, having witnessed the incident in a leading position, vividly describes crass safety violations in his memoirs: overtired engineers working forward and backward through partly executed launch procedures, disregarded abnormalities in several electrical systems, inspection of toxic substance filling states by listening to gurgling sounds in tubes while crawling unprotected through the interior of the rocket, 100 plus an additional (actually not needed) 150 people present at the launch pad, and safety officers silenced by the mere presence of a top-rank military leader sitting himself next to the rocket. The Chief Marshal of Artillery, Mitrofan Nedelin, could later be identified only by his partially melted Gold Star medal—awarded for heroism during WWII, a fact that Chertok alludes to when he calls this not bravery but only risking his live—that remained recognizable [36].

Several root causes, including program management, schedule, design, and testing, led to the disaster, as Newman and Chertok agree [22,36]. However, the two authors disagree with respect to whether software was involved. Newman lists software design and testing as root causes. But Chertok objects. The background is that the rockets had onboard control in the form of programmed sequencers or "programmed current distributors" (PTRs). PTRs had nothing to do with what are today considered computers. They were basically only mechanical sequencers that activated the rocket's systems in a preplanned sequence [36]. Given that open information on the accident was scarce for decades, as the original investigation reports were only declassified from top secret status by 2004, "programmed sequencer" can be misunderstood to imply software. Additional vagueness comes from the fact that the term *software* was coined by John W. Tukey just two years before the accident (cf. [37]). *Software* was probably not yet in widespread use in 1960, and would certainly not appear in soviet reports.

If one accepts a software error as a root cause, which is not clear with above considerations, then the Nedelin Disaster would be the first fatal software failure in spaceflight history. Yet it might not be the unequivocal design fault in full public gaze needed to qualify it as MacKenzie's [24] yet-to-come Tay Bridge disaster.

3.2. KOSMOS 419—Mars Probe Stay-at-Home Due to Hours–Years Mix-Up in Rocket Stage (1971)

KOSMOS 419 was a Russian Mars probe launched in 1971. The entry into the transfer orbit from Earth to Mars failed due to a human error in programming the ignition timer for the rocket stage by mis-interpreting the bare number without a unit: instead of expiring after 1.5 h, the timer was programmed to expire after 1.5 years. Consequently, the upper rocket stage did not ignite in time, so the probe re-entered the atmosphere and burned up [28].

3.3. Viking 1—Software Update Leaves Mars Probe Alone with Wrong Antenna Alignment (1982)

Viking 1 was one of two Viking probes, each consisting of an orbiter and a lander, sent to Mars by NASA to land there. The launch took place on 20 August 1975, and the Viking 1 Orbiter reached Mars' orbit on 19 June 1976. After a relocation of the landing site, the landing finally took place on 20 July 1976. The mission ended after 6 ½ years on Mars surface, when, due to an error in the reprogramming of the software controlling the battery charging cycle, the configuration of the antenna alignment was overwritten. Attempts to restore contact were discontinued at the end of December 1982 [38].

The total cost of the Viking program with both its probes was 1 billion USD, [39] or about 5 billion Y2024 USD.

3.4. Phobos 1—Bad Dead Code Not-So-Securely Locked away Accidentally Set Free (1988)

The Russian Phobos program consisted of the probes PHOBOS 1 and PHOBOS 2, launched in 1988 to explore Mars and its moons Phobos and Deimos. Both probes were lost, although PHOBOS 2 was able to at least partially complete its mission [28].

During transit to Phobos, communication with PHOBOS 1 was lost. Subsequent investigations revealed that a test sequence intended for ground tests was initiated, deactivating the thrusters of the attitude control system. The program had no use during flight. But it had not been deleted from read-only memory because that required special electronics equipment and removing the computer from the spacecraft. There was not enough time, so the program was locked in a virtual safe, sealed off by other software. However, due to malignant bad luck, a faulty command from the ground station that omitted just one letter set the program free and executed it. This sent Phobos 1 tumbling, not able to recover orientation [40]. As a result, the probe could not maintain the alignment of the solar panels toward the Sun, which quickly led to power exhaustion [41].

3.5. Phobos 2—Preparations for Energy-Intensive Imaging Break Communication (1989)

PHOBOS 2 was partially successful but also experienced a loss of communication. Taking an image required a lot of electric energy. Therefore, to conserve energy, the transmitter would be turned off during imaging. But when it was expected to restart again, no signals were received. The control group's hurried emergency commands recovered some last few pieces of telemetry data before the probe went silent forever. The probe was tumbling (and probably depleted its energy). Investigations concluded that the problems were located in the onboard computer [40], but it is not clear whether software was the reason. One report [41] mentions multiple parallel onboard computers, which might hint at a software rather than a hardware problem. Roald Kremnev, stating that coming probes would have enough power to avoid regular communication shutdowns [40], seems to imply that valuable time for rescue was lost due to the shutdown antenna.

3.6. PSLV-D1/IRS-1E—Suborbital Flight Due to Overflow in Attitude Control Software (1993)

The PSLV was the first major launcher of India capable of transporting application satellites. It was developed with technology support from the French Company Société Européenne de Propulsion in exchange for 7000 transducers and 100 man-years of labor. The launcher had its maiden flight on 20 September 1993. Problems started during the separation of the second and third stage and finally sent the vehicle on a suborbital trajectory back to ground [42,43].

In aerospace engineering, gimbal nulling ensures that gimbal axes are properly aligned by nulling out residual motion errors in order to stabilize trajectory. When the second-stage engine shut down for a coasting phase, it created disturbances that prevented accurate gimbal nulling. Additionally, two of four retro rockets used for second-stage separation failed to ignite so that the second and third stage collided, further aggravating attitude disturbance. Upon its start, the third-stage control system should have corrected the disturbance. Yet the attitude control unit failed to do so due to an overflow error in the pitch control loop of the onboard software because a control command exceeded the specified maximum value [43].

3.7. Clementine—Software Lock-up Not Detected by Missing Watchdog (1994)

Clementine was an 80 million USD (168 million Y2024 USD) [44] probe jointly developed by the United States Department of Defense and NASA to study the lunar surface and the near-Earth asteroid Geographos. Clementine launched on 25 January 1994 and reached lunar orbit about one month later. On 3 May, after a successful research mission around the Moon, Clementine left lunar orbit on its way to Geographos. On 7 May, the attitude control thrusters fired for 11 min, completely depleting the fuel. This accelerated Clementine to a rotation rate of 80 revolutions per minute. The mission to Geographos was canceled, and instead, Clementine was sent into Earth's Van Allen belt to test the instruments in this high radiation environment [45].

Due to time constraints, the software was not fully developed and tested before launch. Among other missing functionality, a watchdog had not yet been implemented to protect the software against lockups. During previous operations, over 3000 floating point exceptions were detected, forcing the operations team to manually reset hardware at least sixteen times. Another such runtime error led to the mishap. The software locked up again but this time unintentionally executed code to open the attitude control thrusters. Additionally, the lock-up prevented a 100 ms safety timeout to close the attitude control thrusters from triggering. As a result, the propellant for attitude control was depleted within 20 min before another ground-commanded hardware reset could restore control [44,46].

In 1992, NASA administrator Daniel Goldin began implementing his "Faster, Better, Cheaper" (FBC) initiative. As McCurdy explains [47], it intended to economize bureaucratic management following public dissatisfaction (New Public Management) and cut on schedule and cost, which had been spiraling out of control: potential failure of an already expensive project means more risk due to higher risk cost, justifying more efforts in preven-

tion, further increasing cost, and so on. (Wertz et al. later call this effect the Space Spiral [8].) In this perspective, "better" means more capability per dollar invested, i.e., relatively more capability, not absolutely. Smaller and lighter spacecraft were a central aspect, and software has great potential to soak up complexity and therefore simplify hardware design (see also Section 1). Also, traditional systems engineering (or systems management) was replaced by more "agile" (as an allegory to the agile manifesto, see [48]) approaches like "face-to-face communications instead of paperwork reviews" [47]. Clementine was itself not an FBC project. However, it embodied the characteristics of the prototypical FBC project: small, cheap, and hailed "as if to show NASA officials how this was done" [47], and it would bolster the confidence of the new administrator, Goldin, who already had experience with small satellites and was chosen to change NASA culture [47].

3.8. Pegasus XL/STEP-1—Aerodynamic Load Mismodeled (1994)

Pegasus was a privately developed launch vehicle that air-launched by being dropped from underneath a large Lockheed L-1011 TriStar airplane. Pegasus XL was a larger variant that had its maiden flight carrying the Space Test Experiments Platform/Mission 1 (STEP-1) satellite on a commercial launch contract. The launch and failure happened on 27 June 1994. The first stage ignited 5 s after drop. Then, 27 s later, exponentially increasing divergence in roll and yaw began, causing a loss of control 39 s into flight. About two minutes later, ignition of the second stage failed, and the self-destruct command was issued [29].

Chang identifies the "autopilot software" as responsible. It lost control of the vehicle in a "coupled roll-yaw spiral divergence" due to erroneous aerodynamic load coefficients determined during design [29]. The available literature does not state explicitly whether second-stage failure is related to the earlier loss of attitude control.

3.9. Ariane 5 Flight 501 (1996)—New Ariane 5 Failing Because of a Piece of Old Software

The maiden flight of the European launch vehicle Ariane 5 took place on 4 June 1996. Approximately 40 s after liftoff, the rocket underwent a sharp change in direction, leading to its breakup and triggering the self-destruct mechanism. Onboard were four research satellites intended for the study of Earth's magnetosphere. The damage was estimated at 370 million USD (≈710 million Y2024 USD) [49].

The cause of the direction change was an arithmetic overflow in the two inertial reference systems (IRSs). These systems were inherited unchanged from the predecessor Ariane 4, which had lower horizontal velocities. Software safeguards detected the overflow, but there was no code to handle it. As a result, the active IRS unit deactivated itself. The redundant IRS unit, running in "hot standby" mode, had deactivated with the same error just 72 milliseconds earlier. The IRS units then sent diagnostic data, which the onboard computer mistakenly interpreted as attitude information, causing it to assume a deviation from the desired attitude and make a correction using full thrust of the main engine. The forces generated by the abrupt change caused the rocket to break apart [50].

The overflow occurred in a part of the IRS software that continued to run for about 40 s after liftoff due to requirements from its use on the Ariane 4, even though it served no function on Ariane 5. A handler for overflows was omitted to prevent the processor utilization from exceeding the specified maximum of 80%. Several arithmetic operations had been examined during development, and protective operations were implemented for some variables. Yet the variable for horizontal bias was not included, as it was believed that its value would be sufficiently limited due to physical constraints. The investigation report suggests that there was reluctance to modify the well-functioning code of the Ariane 4 [50], risking loss of certification and heritage status otherwise.

The decision to protect only some variables was included in the documentation but was practically hidden from external reviewers, albeit not intentionally, due to documentation size [50]. Discussions on whether it was a software or system engineering error continued. Although a software malfunction triggered the disaster, software cannot be designed in isolation; it is always part of a usage environment, i.e., the system [49,51]. The event

prompted significant efforts in software verification measures, particularly utilizing the concept of abstract interpretation to develop industrial tools, enabling the demonstration of the absence of corresponding runtime errors in the corrected software [52].

3.10. SOHO—Almost Lost and Frozen over after New Calibration Method (1998)

The Solar Heliospheric Observatory is a joint mission of ESA and NASA to study the Sun. It was launched on 2 December 1995 and has been operational since April 1996. The approximate cost in 1998 was 1 billion Y1998 USD (≈ 1.9 billion Y2024 USD). SOHO orbits around the Lagrange Point L1 in the Sun–Earth system. Its two-year primary mission was declared complete in May 1998 [53]. Since then, SOHO has entered its extended mission phase, which will continue until end of 2025, when a follow-up mission will be ready [54].

A critical situation arose when an attempt was made to optimize gyroscope calibration in June 1998. The probe carries three gyroscopes, two linked to fault-detection electronics and one to the attitude control computer. These gyroscopes are selectively activated to extend their operational lifespan. Gyro A serves exclusively as an emergency sun reacquisition tool, autonomously activated by software in crisis scenarios, although this function can be manually overridden. Gyro B detects excessive roll rates, while Gyro C aids attitude control in determining orientation during thruster-assisted maneuvers. During a recalibration operation in 1998, operational errors led to a loss of attitude control: Gyro A was deactivated, and inadvertently, the automatic software function to reactivate it was also disabled. Gyro B was misconfigured and started to report amplified rotation rates. Due to the seemingly extreme rotation, SOHO entered emergency mode. In emergency mode, Gyro B was reconfigured, fixing the amplification problem. The ground control team diagnosed a mismatch in the data reported from Gyros A and B, wrongly concluding that Gyro B was sending invalid data, while—needless to say—in fact data from deactivated Gyro A was wrong. Ground control deactivated Gyro B and let the probe use Gyro A. Now the probe actually started spinning faster and faster, but the onboard software was unable to detect this, since Gyro B was deactivated. When ground control finally diagnosed the problem, discrepancies in angles were too big for the emergency software to handle, and attitude control was lost [53].

Control of the de-powered and frozen-over probe could be regained three months later by step-by-step restoring energy, communication, and thawing of fuel, pipes, and thrusters. Two gyros were lost during recovery; the third one followed in December. For several weeks, attitude control was only possible at the cost of 7 kg of hydrazine per week until a gyro-less operations mode was developed [55]. Project officials pointed out that this almost-loss should not be seen "as an indictment of NASA's much-touted" FBC initiative. Yet plans to cut down the SOHO team by about 30% were partially retracted [56].

3.11. NEAR—Redundant Attitude Control Finally Saves Probe after Two Software Problems (1998)

The Near Earth Asteroid Rendezvous (NEAR) was developed by Johns Hopkins University and NASA to investigate the near-Earth asteroid Eros. McCurdy lists the 224 million USD [57] (ca. 450 million Y2024 USD) NEAR mission as the first one of the FBC era. NEAR was launched on 17 February 1996. On 20 December 1998, a prolonged thruster burn was planned to bring NEAR to Eros' orbital velocity for rendezvous. Following the command, communication was lost for 27 h. When communication was restored, NEAR was peacefully rotating with its solar panels facing the Sun. However, telemetry data revealed turbulent events: an aborted braking burn, temporarily very high rotation rates, a significant drop in battery voltage, and data loss due to low battery voltage. The backup attitude control unit was active. NEAR had consumed about 29 kg of fuel with thousands of individual activations of its thrusters, so an alternative route to Eros had to be chosen, extending the journey by 13 months. NEAR successfully rendezvoused with Eros on 14 February 2000 [58].

The investigation commission found no evidence of hardware failures or radiation influences. The main triggers were likely a software threshold set too low and errors in the abort sequence, initiating a chain of events where potential design issues exacerbated the situation. The suspected sequence of events involved the abort of the braking burn due to exceeding the ignition thrust limit. The ignition acceleration was greater than that of previous burns due to fuel consumption, making the spacecraft lighter. As a result of the abort, a pre-programmed sequence was executed to orient the antenna toward Earth. However, the stronger 22 Newton braking thrusters were still configured for use instead of the weaker 4.5 Newton thrusters due to an error in the control sequence. This led to excessive momentum being introduced into the system. The attitude control attempted to stabilize with reaction wheels, but they could not generate enough momentum. Consequently, a desaturation maneuver was initiated, which still used the wrong thrusters, further exacerbating the situation. As the maximum duration of 300 s was exceeded, a switch to the backup attitude control occurred. Upon activation, the backup system correctly configured the attitude control system to use the reaction wheels and the 4.5 Newton thrusters for control. Subsequent correction and compensation maneuvers occurred over the next 8.5 h, with burn durations exceeding the maximum several times, triggering multiple switches between primary and backup attitude control units. After the fifth switch, the onboard software disabled error monitoring, and the backup attitude control remained permanently active. The reason for the additional compensation maneuvers is unclear, possibly due to a stuck thruster or because rate gyros had switched to a less accurate mode due to high rotation [58].

3.12. Mars Pathfinder—Priority Inversion Deadlocks Have Watchdog Eat Science Data (1997)

Mars Pathfinder was a NASA Mars probe launched on 4 December 1996 [39] as the second mission in the FBC Discovery Program after NEAR. The probe cost approximately 175 million US dollars in total (about 350 million Y2024 USD) [59]. Following its landing on 4 July 1997, the onboard computer experienced several unexpected reboots. The cause of these reboots was a priority inversion between three software tasks, resulting in all three tasks being halted. Several times, a watchdog timer detected that the software was no longer running correctly and initiated reboots, significantly limiting system availability. Also, every time, scientific data previously collected was lost. The issue was diagnosed as a classic case of priority inversion and resolved, allowing the mission to continue successfully. Notably, the problem had already been observed during pre-flight testing but could not be reproduced or explained and had been neglected in favor of testing the more critical landing software [60].

The individual components of the Mars Pathfinder exchanged data using a so-called information bus, a shared memory area that parallel processes wrote to and read data from. Three processes had access to the information bus: Process A (bus management, high priority), Process B (communication, medium priority), and Process C (collecting meteorological information, low priority). The integrity of the data in the information bus was secured by mutual exclusion locks (mutex), meaning that if one process had access, the others had to wait until it released access. If Process C with low priority held the lock, higher-priority Process A still had to wait. The situation became problematic when the long-running Process B also started and, due to its higher priority, prevented Process C from running, thereby also hindering the high-priority Process A [60]. This situation is not a hard deadlock because Process B will eventually finish, allowing Process C to complete its task and Process A to eventually gain access, but it may be too late.

3.13. Lewis Spacecraft—Dual Satellite Mission Doomed during Computer's Shift (1997)

The Lewis spacecraft was a part of a NASA dual satellite mission aimed at advancing Earth imaging technology. It was launched on 23 August 1997. Contact was lost after attempts to stabilize the rotation axis on 26 August, followed by reentry and burnup one month later. The proximate cause of the failure was a design fault in the attitude

control software, compounded by inadequate monitoring during the early operations phase. Instead, Lewis was left unattended for several hours, and although anomalies occurred, no emergency was declared. The satellite itself cost 65 million USD, with an additional 17 million USD for the launch (equivalent to a total of 160 million Y2024 USD). After the failure, Lewis' companion satellite, Clark, originally scheduled for launch in early 1998, was canceled [61,62].

Problems with Lewis began minutes after launch with an unexpected switch to the backup computing bus and difficulties in reading out recorded data from ground. Only two days later, ground operations switched back to the nominal computing bus. Shortly after, there was a loss of contact for three hours. After contact was reestablished, Lewis was stable for several hours. The operations team left for overnight rest without declaring an emergency or requesting a replacement team. The reason was pressure to save costs. The attitude control system was expected to tend to Lewis throughout the night. The next morning, however, the operations team found that batteries were discharging and diagnosed a wrong attitude. When they attempted to correct the attitude, only one out of three commands was executed because the other two were erroneously sent to the redundant computing bus. Contact was lost and never reestablished [61].

The attitude control software was reused from another spacecraft with a different mass distribution and solar array orientation. As a result, during the night, mechanical energy dissipation (cf. Dzhanibekov effect) slowly changed the rotation axis by 90 degrees until Lewis was rotating with the solar array edges toward the Sun. The attitude control design relied on a single two-axis gyro that was unable to sense the rate about the intermediate axis from which the rotation energy was transferred. Additionally, the attitude control system's autonomous attempts to maintain attitude were interpreted by the onboard software as excessive thruster firings so that it disabled it [61,63].

The root cause of the problems is today seen in NASA's Faster, Better, Cheaper (FBC) philosophy: enormous cost containment pressures led to the fatal reuse of the attitude control software and the reduced staffing. Moreover, the contracts replaced government standards for technical requirements, quality assurance, etc., with industry best practices. In fact, this meant reduced technical oversight, while the resulting oversight gaps were not filled by industry. NASA sees the mission as a reminder that it should not compromise its historical core value of systems engineering excellence and independent reviews. FBC itself was simply "tossed over the fence" without sufficient support for the personnel expected to implement it. While colocation and communication were inherently crucial for the FBC way of working (e.g., [64]), the industrial contractor's team was distributed to different locations, and project management changed frequently [61].

3.14. Delta III Flight 259/Galaxy 10—Fuel Depleted by Unexpected 4 Hz Eigenmode (1998)

Boeing designed the Delta III as a heavier successor to the highly reliable Delta II launch vehicle. However, the Delta III's maiden flight on 27 August 1998 ended abruptly about 70 s after liftoff due to loss of control, breakup, and self-destruction. The vehicle carried the costly 225 million USD Galaxy 10 communications satellite [65]. The typical launch cost for a Delta III was 90 million USD [66], amounting to a total loss of 590 million Y2024 USD for the failure.

The immediate cause of the loss of control was the exhaustion of expendable hydraulic fluid by the attitude control system, which was attempting to compensate for a 4 Hz roll oscillation that began 55 s after launch. As the hydraulic fluid ran out, the oscillation diminished. Yet some thrusters were stuck in unfavorable positions, causing forces that broke the vehicle apart [67].

The eigenmode roll oscillations were known before the flight because they also existed in the Delta II, from which software was reused. But due to the similarity between both vehicles, full vehicle dynamic testing was not repeated. In total, 56 other roll modes were known, but the 4 Hz oscillation was considered insignificant based on experiences with Delta II flights. However, simulations conducted after the incident revealed significant

differences in the oscillation patterns. More rigorous flight control analyses could have detected the new oscillation pattern, which was not included in the software specification. Boeing attributed the problem to a lack of communication between different design teams. Just as with the first Ariane 5, reuse and insufficient testing of software working on different hardware introduced new problems [65,68].

3.15. Titan-IV B-32/Milstar—Software Functions Reused for Consistency (1999)

During the launch of a Titan-IV rocket intended to deploy a Milstar satellite into geostationary orbit on 30 April 1999, a malfunction occurred, preventing the rocket from placing the satellite in its target orbit. This malfunction was attributed to the faulty configuration of a filter for sensor data within the inertial navigation unit [13]. The total cost was over 2.5 billion Y2024 USD for satellite (800M USD) and launch (400M USD) [69].

The circumstances surrounding this incident exhibit parallels to those of the Ariane 5 accident: A part of the software (the sensor data filter), although unnecessary, was retained for the sake of consistency. While modifying functional software can pose risks, executing unnecessary functions just to not have to modify a piece of reused software carries its own set of risks. Consequently, the failure was also a result of an inadequate software development, testing, and quality assurance process, which failed to avoid or detect the software faults [13].

3.16. MCO—Measurement Unit Mix-Up between Science and Industry (1999)

The Mars Climate Orbiter (MCO) was the second probe of NASA's FBC Mars Surveyor program, intended for remote sensing of the Martian surface and the investigation of its atmosphere. The system cost 180 million USD (340 million Y2024 USD). It was launched on 11 December 1998. On 23 September 1999, the MCO was supposed to enter an elliptical orbit around Mars and then decelerate into its target orbit within the atmosphere. However, the MCO approached Mars and its atmosphere too closely [70].

The reason for this mishap was minimal deviations from the approach trajectory, which arose from the repeated use of thrusters for desaturation maneuvers of the reaction wheels. During the journey to Mars, the reaction wheels had to continuously absorb asymmetrical torques from solar wind because solar panels were mounted only on one side. The forces exerted during thruster firings were recorded by MCO and transmitted via telemetry to the ground station. Based on this information, the effects of thruster firings on the probe's trajectory were determined. NASA's requirements stipulated that these so-called "Small Forces", in the form of impulse changes, be transmitted using the scientific SI unit Newton-seconds. However, the probe manufacturer Lockheed Martin had used the pound-second unit, common in the US industry, equivalent to 4.45 Newton-seconds. Thus, the operations team had assumed a much smaller impulse value [70].

At the beginning of the mission, there were issues with transmitting the Small Forces data. When the data became available, they did not align with the models of NASA's ground team. Yet, too little time was left to figure out the cause [70].

For Johnson, one cause is that the probe was built asymmetrical to save costs on the second solar panel. The resulting imbalance increased the complexity of software and operation, which served as a "band-aid" solution [71]. This might be a parallel to the later software problems of the Boeing 737 Max, where software was used to compensate for a structural imbalance arising from cost considerations (cf. [72]).

3.17. MPL—Are We There Yet? (1999)

The Mars Polar Lander (MPL) was the third probe in NASA's Faster, Better, Cheaper (FBC) Mars Surveyor Program, launched in January 1999. On 3 December 1999, the connection was unexpectedly lost upon entry into the atmosphere. The suspected cause is a software error, which shut down the braking rocket because the system assumed the probe had landed. It is believed that this erroneous decision was made due to faulty orientation

and positional information, attributed to unrecognized vibrations after deployment of the landing legs [28].

Leveson [13] sees several accidents during this period primarily rooted in a deficient safety culture and management failures characterized by unclear responsibilities, diffusion of accountability, and insufficient communication and information exchange. There was pressure to meet cost and schedule targets, leading to heightened risk through the cutting of corners and neglecting established engineering practices necessary for ensuring mission success. The FBC approach was criticized for lacking a clear definition that ensured it went beyond mere budget cuts while still expecting that the same amount of work be accomplished by overworked personnel [13].

While software plays a crucial role in FBC and we focus exclusively on software-related failures in FBC missions here, it is worth noting that several more failures were not related to software. Missions such as WIRE or the German Abrixas encountered issues related to "low-tech items" like heat protection covers or batteries, respectively, sparking debates on how to balance speed with quality control [73]. Before the backdrop of these back-to-back failures, "FBC faded into history [...] leading to a shift back to a balanced government role in managing space program development and implementation" [61]. As a reaction to Abrixas, Germany's space agency firmly embedded its product assurance department and its responsibility for missions and other major projects [2]. But Leveson also criticizes technical aspects of the accident report because it recommends employing techniques developed for hardware, such as redundancy, failure mode and effects analysis, and fault tree analysis, which are not suitable for addressing software-related issues. These techniques are designed to handle random isolated component failures rather than design errors and dysfunctional interactions among components, which are typical for software [13].

3.18. Cassini–Huygens—Software Could Have Been Rescue for Doppler Problem (2000)

Cassini–Huygens was a collaborative mission between NASA and ESA, with the involvement of the Italian space agency ASI, aimed at exploring the planet Saturn and its moons. After reaching moon Titan, the Cassini orbiter was supposed to serve as a communication relay for the Huygens lander. The duet was launched on 15 October 1997, and was largely successful.

However, during a routine test of communication between Huygens and Cassini in February 2000, it was noticed that 90% of the test data were lost. The reason for this was frequency shifts due to the Doppler effect. Extensive ground tests, which could have detected the design flaw, had been omitted due to high costs. A modification of the data stream bit detector software could have solved the problem, but software could only be changed on the ground. Instead, the trajectory of Cassini was later altered to mitigate the Doppler effect during Huygens' descent. The investigation report concludes that adequate design margins and operational flexibility should become mandatory requirements for long-duration missions. The Hubble Space Telescope and the SOHO spacecraft are named as positive examples that demonstrate how problems occurring in orbit can be resolved by software patches [74].

During the flyby of Titan on 26 October 2004, one of twelve instruments was not operational due to an unspecified software failure, which was expected to be fixed for later flybys [75].

3.19. Zenit-2 and Zenit-3SL—Unspecified Software Bug Only Lives Twice (2000)

Zenit-2 was a Ukrainian carrier rocket. During a launch on 9 September 1998, intended to deploy 12 Globalstar satellites, the system was lost. The cause was attributed to failure to close an electro-pneumatic valve of the second stage before launch. A software error was identified as the cause. In consequence, the second stage was unable to achieve its full power due to the loss of pneumatic pressure [76]. Further information about the cause of the failure was initially not disclosed, and it might not have been found at that time.

Zenit-3SL was the successor to Zenit-2, operated by multinational launch service provider Sea Launch. On 12 March 2000, the communication satellite ICO F-1 was supposed to be transported into a geostationary orbit. Once again, a valve for the pneumatic system of the second stage was erroneously not closed. Again, the target orbit could not be reached [77]. Subsequent investigation revealed that the underlying cause was a software error, the same one that had led to the previous loss. A line of code with a conditional for closing the valve just before launch was deleted by a developer during an update of the ground control software. Again, testing had not found the problem [31,76].

3.20. Spirit—File System Clutters Mars Rover RAM (2003)

Spirit, alongside Opportunity, was one of NASA's two Mars Exploration Rovers. Both rovers were launched on 10 June 2003, with Spirit touching down on the Martian surface on 4 January 2004. However, on 21 January (Sol 18), communication failures occurred [78].

The VxWorks operating system included a file management system based on DOS. Each stored file was represented by an entry. Entries for deleted files were marked as deleted by a special character in their file names but retained, leading to an ever-growing number of files. For improved performance, the file system driver kept a copy of all directory structures from the drive in the RAM [78].

The file system driver was erroneously allowed to request additional memory for the ever-growing number of files instead of receiving an error. This led to an increasing portion of the system RAM being used for file system management until none was left. A configuration error in the memory management library further resulted in the calling process just being suspended without receiving an error message. So, when the file system driver requested exclusive access to the file system structure, it was then suspended, failing to release access. Consequently, other processes were blocked from access to the file system. Detecting these suspensions, the system rebooted, entering a reboot loop, attempting to read the file system structure at startup. Luckily, a safety function allowed a new reboot no earlier than 15 min after the last reboot, opening a short access window. Furthermore, Spirit was supposed to shut down during Martian nights to conserve battery capacity, but the function needed to access the file system. Instead, when batteries were drained, hardware safeguards shut down electronics until the next Martian morning [78].

To rescue the rover, the ground crew had to manually command the system (after the hard shutdown each night) into a "crippled" mode, in which the file system would not be used. After one month of intensive work, a software update was uploaded to the rover. Ground testing had not detected the problem because the scenario extended only over 10 Martian days and, because at the time of testing not all instruments were available, fewer files were generated than during later operations. Spirit's operational life was finally extended significantly beyond the planned mission duration [78].

3.21. Rockot/CryoSat-1—Software Forgets to Send Separation Signal (2005)

CryoSat-1 was a 135 million EUR [79] (220 million Y2024 USD) research satellite of the ESA designed to measure the Earth's ice sheets. It was launched on 8 October 2005 using a Russian Rockot launcher with an additional third stage. However, it did not reach its intended orbit. The separation of the second and third stages did not occur because the third stage mistakenly failed to send the command to shut down the second stage. A software error in the flight control system is suspected to be the cause [80]. An investigation report detailing the exact causes based on the findings of the Russian State Commission was announced [81], but it was not publicly available.

3.22. MGS—Parameter Upload Not Sufficiently Verified by Onboard Computer (2006)

The Mars Global Surveyor (MGS) was a NASA spacecraft launched on 7 November 1996. The MGS belongs to the FBC era projects and was the first one in the Mars Surveyor Program [47]. On 2 November 2006, the spacecraft reported that a solar panel had become jammed. The next scheduled contact about 2 h later did not occur. The spacecraft was lost.

The main cause dated back five months, when routine alignment parameters were updated. The update used a wrong address in the onboard computers' memory. It inadvertently overwrote the parameters of two independent functions with corrupt data. This caused two errors in different areas of the system: First, an adjustment mechanism of a solar panel moved against an end position. The onboard computer mistakenly interpreted it as jammed and entered an operating mode that involved periodically rotating the spacecraft toward the Sun to ensure power supply for charging the batteries, even if this led to thermally unfavorable attitudes. In fact, direct sun heated a battery, causing the computer to limit charging to mitigate overheating. However, the reduced battery charge was insufficient to last through the subsequent lower-sunlight operating mode, leading to critical discharge of both batteries. The second error affected the alignment of the main antenna in the aforementioned mode, preventing communication with the ground station so that the operations team was unable to notice the discharge early enough [82].

The investigation report identified problems including (i) insufficiencies of operating procedures to detect errors, (ii) inadequate error protection and detection in the onboard software, and (iii) increased risks due to reduced staffing of the operations team [82].

3.23. TacSat-2—Successful Experiments Regardless of a Wealth of Software Problems (2006)

TacSat-2 was an experimental military satellite developed by the US Air Force Research Laboratory. It carried out various tactical experiments, including direct control of the spacecraft by untrained in-theater operators. The human–machine interface abstracted away spacecraft intricacies, enabling the system to provide information much more quickly than traditional sources. The mission had a total cost of over 17.5 million USD [83]. While the experiments were successful, the teams encountered software issues during preparation and commissioning. The mission is probably not notable for the amount of software issues—which might be quite normal for such mission—but for the openness with respect to them.

The project's internal database documented 340 unique problems identified during the integration and testing of the satellite. Of these, 114 were related to hardware, while nearly twice as many (226) were related to software. About 40% of the problems were related to attitude control (consisting of 30% for the IAU, which probably means inertial attitude unit but is not stated in the publication, and 10% for the attitude determination and control subsystem).

At the launch pad, the mission's start was delayed by five days until December 16 due to a software error discovered in the attitude control system. This error could have caused serious issues with power generation due to incorrect sun pointing [83,84]. The first seven days in orbit were tumultuous: The spacecraft was tumbling rapidly on the first pass, with the reaction wheels fully saturated, the magnetic torquers exerting maximum force, and the star tracker providing inaccurate attitude information. Ground control commands were ineffective initially due to using the wrong command configuration. Once communication was established, it was discovered that a sign error in the momentum control system caused the tumbling. Additionally, the inertial measurement was not properly configured in semi-manual attitude control. After patching the code, the spacecraft entered sun tracking mode. However, the patches were erased the following day, causing the spacecraft to tumble again. The re-uploaded patches were burned into EEPROM on 23 December 2006 [83].

3.24. Fobos-Grunt—Software Failure Possibly Incorrectly Blamed on Radiation (2011)

Fobos-Grunt was a Russian space probe intended to collect and return samples from the Martian moon. The mission's cost was 165 million USD (225 million Y2024 USD). After its launch on 8 November 2011, the probe failed to enter the transition orbit to Mars, resulting in its reentry into Earth's atmosphere on 15 January 2012. The Chinese Mars probe Yinghuo-1 was also lost as a piggyback payload [28,85].

A computer reboot caused the probe to enter safe mode, which was intended to ensure the alignment of the solar panels toward the Sun. In safe mode, the probe awaited

instructions from the ground. However, due to an "incredible design oversight" it could only receive such instructions after successful departure from parking orbit [86].

Yet official statements are contradictory: News agency RIA Novosti stated that a programming error led to a simultaneous reboot of two working channels of an onboard computer. However, the official report attributes the issue to radiation effects on electronics components not designed to withstand them [85,86]. Even if the hardware was unsuitable, two failures only within seconds would be highly unlikely. At the time of the incident, Fobos-Grunt was in an orbit below Earth's radiation belts, in an area of low radiation. NASA expert Steven McClure speculates: "Most of the times when I support anomaly investigations, it turns out to be a flight-software problem, [. . .] It very often looks like a radiation problem, [but] then they find out that there are just handling exceptions or other conditions they didn't account for, and that initiates the problem" [86]. Yet a software error has not been officially confirmed, also meaning that nothing is known about the nature of the software error.

3.25. STEREO-B—Computer Confused about Orientation after Reboot (2014)

The Solar and Terrestrial Relations Observatory (STEREO) of NASA comprises two probes, STEREO-A (Ahead) and STEREO-B (Behind), which orbit the Sun on an Earth-like orbit but faster and slower than the Earth, respectively, allowing for new perspectives of the Sun. The 550 million USD (1 billion Y2024 USD) mission was launched in October 2006 and was originally planned for a duration of two years. However, both probes remained operational for much longer. While STEREO-A continues to function today, the connection to STEREO-B was lost on 1 October 2014. Contact was unexpectedly made in August 2016 and intermittently afterwards. NASA temporarily suspended contact efforts in October 2018. When STEREO-B presumably passed Earth in summer 2023, an optical search and an attempt to make contact were unsuccessful. This ended all hope to find it [87,88].

The causes of the connection loss are not fully understood. The STEREO probes are equipped with a watchdog that resets the probes' avionics if no radio signal has been received for 72 h. The restart function is actually intended to rectify configuration errors. But since the probes would be out of sight of Earth signals for three months while moving behind the Sun, preparations had to be made. STEREO-A had already passed the test. After the ground station had not sent a signal to STEREO-B for 72 h, the avionics restarted as expected and contacted the ground station. However, the signal was weaker and quickly disappeared. The transmitted data could be partially reconstructed and indicated that the attitude control computer received erroneous data that made it believe the probe was rotating. It possibly attempted to counteract this would-be rotation but began to actually rotate as a result, now losing contact with the ground station. The faulty data came from the inertial measurement unit. In case of failure, it should be deactivated and sun sensors used instead. However, since it remained active, it is evident that the attitude control computer did not detect the failure because it failed in an unexpected way [87]. Another explanation discusses frozen propellant that made thrusters perform abnormally and aggravated the situation [88].

3.26. Falcon 9/Dragon CRS-7—Emergency Parachute Not Configured on Older Variants (2015)

SpaceX's cargo vehicle Dragon CRS-7 was launched on 28 June 2015, bound for an ISS resupply mission. However, the Falcon 9 launch vehicle suffered an overpressure in the second stage after 139 s and disintegrated, propelling Dragon further ahead. The capsule survived this event but later crashed into the ocean [89].

Dragon is equipped with parachutes for landing. However, Dragon is inactive during launch, and software for initiating parachute deployment was not installed in Dragon. A later variant of Dragon, Dragon 2, is by default equipped with the necessary contingency abort software to save the spacecraft in case of failures or unexpected events like an off-nominal launch. Elon Musk noted that had "just a bit of different software" been installed

in CRS-7, it would have landed safely. After the event, other Dragon cargo spacecraft received the new abort software to be able to deploy their parachutes [89].

3.27. Hitomi—Attitude Control Events Conspire for Disintegrating Spin (2016)

Hitomi (also known as ASTRO-H) was an X-ray satellite operated by JAXA, intended for studying the spectrum of hard X-ray radiation from the universe. Hitomi was launched on 17 February 2016. Three days into its nominal mission operation, on 26 March 2016, JAXA lost communication with the 400 million USD (500 million Y2024 USD) device. Investigations revealed that a software error had had spun the satellite so rapidly that parts, including the solar panels, tore off [90,91].

Onboard, a combination of gyroscopes and star trackers was used for attitude determination. The gyroscopes were regularly calibrated using the star trackers. This involved searching for a star constellation (acquisition mode) and then tracking it (tracking mode). By comparing the data with the gyroscopes, the measurement accuracy of the latter could be estimated [92].

Prior to the accident, the star sensor transitioned normally from acquisition to tracking mode but unexpectedly reverted to acquisition mode before completing calibration. This unexpected switch was attributed to the low brightness of the captured stars. The threshold value was intended to be optimized later. However, due to the premature mode change, the measurement accuracy was compromised. The attitude control system assumed that the satellite was rotating, although this may not have been the case, and attempted stabilization using the reaction wheels, resulting in actual rotation. Due to the current orientation relative to the Earth's magnetic field, the magnetic torquers could not be used to desaturate the reaction wheels, so the control thrusters were employed instead. However, their control command had been changed to a faulty configuration a few weeks earlier. Instead of reducing the rotation, they inadvertently accelerated it [92].

3.28. Schiaparelli—Discarding Parachute Deep below Mars Surface (2016)

As part of the ESA mission "ExoMars 2016", the 230 million USD (300 million Y2024 USD) Schiaparelli lander was launched on 14 March 2016. Its objective was to validate an approach to landing on the Martian surface and demonstrate it. Shortly before the intended landing, contact with Schiaparelli was lost. Investigations revealed that Schiaparelli crashed into the Martian surface due to erroneous attitude information [93,94], quickly earning it the unofficial internet nickname "Shrapnelli".

To determine its position and velocity upon entering the Martian atmosphere, data from gyroscope and accelerometer sensors were integrated. Upon and after parachute deployment, significant rotational rates occurred, causing the gyroscopic sensors to partially saturate. This saturation was expected and factored in for a brief period. However, Schiaparelli exhibited an oscillatory motion, causing the estimated pitch angle along the transverse axis to deviate significantly from the actual orientation. Upon jettisoning the front heat shield, the radar altimeter was activated. As Schiaparelli was not falling vertically but at an angle, its longitudinal inclination had to be considered in trigonometric calculations. Assuming an inclination of 165°, which would correspond to nearly upside down, the cosine and thus, ultimately, the estimated altitude yielded a negative value, indicating a negative altitude above the Martian surface. Consistency checks of the various estimated attitude data that the onboard software conducted failed for over 5 s. After this time, the software eventually decided to trust the radar data because landing without it was deemed impossible anyways. Since a negative altitude was, of course, below the minimum altitude for the next landing phase, the attitude control system jettisoned the parachute and ignited the braking thrusters. However, they were deactivated after only 3 s more, as the negative altitude led the attitude control system to assume that this landing phase had ended as well. At this point, the lander was at approximately 3.7 km altitude and descended unimpeded toward the ground at a speed of 150 m/s, crashing within 34 s. According to the investigation report, the modeling of dynamics in the parachute

phase, treatment of sensor saturation, considerations of error tolerance and robustness, and subcontract management were inadequate [94].

3.29. Eu:CROPIS—Software Update Leads to Loss of Communication (2019)

Eu:CROPIS (Euglena and Combined Regenerative Organic-Food Production in Space) is a small satellite developed by the German Aerospace Center (DLR). The purpose of the mission was to test the possibilities of a bioregenerative life support system under conditions of gravity such as those on the Moon or Mars. On board were four experiments, including the eponymous Eu:CROPIS, consisting of two greenhouses. During a software update for this experiment in January 2019, the experimental system entered a safe mode, leading to communication issues. Several attempts to restore communication, including restarts of various modules, were unsuccessful, resulting in the experiment being ultimately abandoned. The other three experiments on board were successful [95].

Neither the application nor the adoption of ECSS (European Cooperation for Space Standardization) space standards was deemed necessary or beneficial. Therefore, verification activities focused on early end-to-end testing and the application of the Pareto principle to identify the most critical malfunctions with a less representative test setup [96].

3.30. Beresheet—New Space Lunar Lander Impacts Surface (2019)

Beresheet was a 100 million USD (120 million Y2024 USD) lunar landing mission by the Israeli company SpaceIL. Landing was attempted about two months after launch on 11 April 2019. Initially, the lander's operation appeared "flawless". Yet, after eleven minutes, the main engine erroneously shut down, and Beresheet crashed onto the Moon's surface another four minutes later [97] at a vertical speed of about 500 km/h. The immediate cause for the failure was a malfunction in an inertial measurement unit. A "command uplinked by mission control [. . .] inadvertently triggered a chain reaction that led to the shutdown of the probe's main engine" and "prevented it from activating further" [98]. While the investigation report is not published, Nevo [99] gives more details:

The Beresheet mission was conducted as a New Space project, i.e., it relied on private funding to develop a small device for a low-cost mission. As part of this strategy, on the one hand, emphasis was put on inexpensive but less reliable components not tested for space. For instance, to compensate this lower reliability, Beresheet had two redundant inertial measurement units. Yet, on the other hand, it had only one computer for cost-saving reasons. Therefore, inflight software patches could only be stored in non-permanent memory, meaning they had to be uploaded after every reboot [99].

Beresheet hitchhiked as a secondary payload to save on costs, utilizing excess capability from a different launch. But it also meant that the schedule was dictated by the primary payload and could not be changed. A manager later admitted that Beresheet was not sufficiently prepared, and usually a launch would have been postponed. Hence, work before the launch was surely intensive, and software development continued afterwards to fix or work around newly detected problems. For example, dust from the launch contaminated the star tracker, complicating attitude determination and increasing maneuver complexity to avoid sun blinding. Engineers cut vacations to deal with the immense amount of work and even slept on-site, causing fatigue and squeezing out important training [99].

When one of the two redundant inertial measurement units failed during landing, the operations team decided to recover the first unit instead of relying on the second one alone. However, the recovery attempt jammed communication with the functioning unit for about a second, which was enough for the computer to suspect a malfunction and reboot. In turn, it lost all software patches and did five consecutive reboots until it could finally load them. During these reboots, the main engine had shut down, and Beresheet was already falling. The computer tried to restart the engine immediately but to no avail because a necessary power source had also shut down during the multiple reboots. The problem was known but not yet fixed due to a lack of time [99]. The incident evokes reminiscences of FBC program failures, although there are also notable differences.

3.31. Boeing CST-100 Starliner—Insufficient Processes and Software Quality Assurance (2019)

On 20 December 2019, a NASA-contracted Boeing CST-100 Starliner capsule launched on an unmanned test flight to the ISS. After separation from the rocket, a burn was intended to propel the capsule into the correct orbit. But instead, it consumed fuel for overcorrections. While ground interventions enabled the capsule to be controlled and an orbit correction was made, the capsule had lost too much fuel to reach its destination [100]. Boeing had to set aside 410 million USD for a second unmanned test flight [101], which was completed successfully in May, 2022 [102].

The incident was caused by a software error in setting the mission elapsed time, which controls crucial capsule operations. The capsule was supposed to synchronize its clock with the end of the rocket's countdown. However, this synchronization occurred 11 h too early, resulting in a significant deviation [103].

During the flight, further software errors were identified. One error could have led to issues with nozzle control during the separation of the service and crew modules before reentry. A collision and damage to the crew module's heat shield (reminiscent of the Space Shuttle Columbia Disaster in 2003) were possible. The error was corrected before the reentry maneuver so that Starliner landed safely [103].

Closer examination of the capsule's design and processes by NASA revealed that the software problems were not the root cause but only symptoms. Fundamental process errors were discovered in the design, coding, verification, and testing phases. Numerous process escapes led to the introduction of and failure to detect software errors. The process-related issues should have been addressed by software quality assurance at various stages and also been detected by customer oversight. This raised the question of how to assure that no more problems than just the two above issues were in the software. Officials admitted that they wished they had done better with software, and that the flight test taught them a lot [103,104]. Investigation continued, concluding with a total of 80 corrective actions identified by July 2020. Most of the recommendations addressed process and operational improvements (35), testing and simulation (21), and software (17). [105]

4. Conclusions and Prospects

Space exploration requires increasingly sophisticated functionality and therefore will not be possible without some form of computing or software power. Software enables mission preparation, facilitates collaboration, acts as a spacecraft's brain (but is not necessarily artificial intelligence) to enable autonomy, allows rapid reacting to problems, and assists astronauts on far away missions, e.g., with knowledge, psychological issues, robotic support, or in life support systems. But its immense utility also turns software into a risk. An analyst from Project Icarus (cf. [106]) even considers software the greatest risk of future interstellar travel [107]. Software failures in spaceflight have not yet killed people (not counting the Nedelin disaster, Section 3.1), but given the steady occurrence of failures, it seems indeed to be only a matter of time until this happens.

The history of spaceflight is filled with accidents related to software. However, attributing these accidents solely to software is not always straightforward, and identifying the specific software or software engineering fault involved is often challenging. Software engineering was born from the software crisis, and quality is the cross-cutting theme of software engineering (e.g., 500+ hits for "quality" are packed across the 335 pages of the SWEBOK [108]). Many of our software methods were invented to prevent failures, and many of them are invaluable. For example, bug or software problem tracking is vital to the knowledge management of projects, even in small ones (e.g., [109]). In this section, we do not want to discuss or evaluate all those methods. Looking back at the list of spaceflight accidents, we see the following broader prospects for software-heavy spaceflight of the future:

Software accountability: In the future, more accidents that started out as adverse natural events or hardware failures might be seen also as software failures if a smarter software or FDIR could have saved the spacecraft (e.g., Dragon CRS-7). Just as more and more

functionality is transferred from hardware to software, in the future, software could be more frequently held responsible for system failures originating from hardware or natural events. We think that Prokop [23], with her 40% missing functionality, also argues in this direction

Software poka-yoke: Shigeo Shingō's principle attempts to reduce errors by designing connecting interfaces in such a way they can only be connected in the right way. It is traditionally used in hardware engineering. But poka-yoke could have avoided several accidents (e.g., MCO) had it been applied to user (e.g., human–machine interaction, usability; cf. MacKenzie [24]) and data interfaces (e.g., program structure signatures, application programming interfaces, network protocols, hardware–software interaction, file formats). In the spirit of poka-yoke, it may mean intuitive design, validation of inputs and feedback, consistency and standardization, or automation and assistance. For some of these features (e.g., validation of inputs, providing feedback), additional software functionality is obviously needed.

Software margins: Functional or technical margins are widely used in engineering, and they are particularly associated with spaceflight due to the high stakes and complexity of missions, where they apply, for instance, to structure, power systems, or computing (e.g., memory margin, CPU margin). As opposed to these classical margins, a software margin is a "reserve" of software-implemented additional functionality beyond the bare necessary minimum so that it can tolerate more of the inevitable failures [7] and handle (more) exceptional situations. In order to improve the resilience of spaceflight systems, software margins will be necessary. But, of course, error handling and fail-safe behavior add to complexity, cost, and development time.

Increasing complexity: Complexity is a problem in reliable systems because it implies risks and leads to fragility (cf. [110]). For example, Paxton points out that failure review boards have a remarkable ability to locate software and design issues. The obvious reason why they have this ability is that they know where to start looking [64]. But Perrow-class failures convene on the basis of complexity. Complexity can increase inadvertently, which should be avoided by proven means, i.e., software engineering. But complexity can also be necessary to provide additional functionality, e.g., new functions, FDIR, taking over functionality from hardware, software margins, and fault-deterring poka-yoke interfaces. As such, complexity is also needed to improve resilience, i.e., robustness, but also the ability to tolerate transient phenomena (cf. [64,110]). Where complexity cannot be avoided, tools for better dealing with complexity are needed; these may include more reuse of software "components" to limit complexity for developers [14], benefit from previous validation (which still has its own problems), and extend further to open source or software product lines with their inherently high level of reuse (e.g., [111]). AI assistants (e.g., for requirement quality [112]), software analysis tools (e.g., more technical ones like static and dynamic analysis but also software reliability analysis), simulators, software metrics for abstracting from complexity and creating transparency (e.g., [113]), and many other methods can help to cope with inevitable technical and project complexity. The ever-increasing complexity will surely continue to push the development of new methods. And new methods will allow for even more functionality and complexity.

Inflight updates: The ability to update software inflight is valuable. Where updates were not available (e.g., Cassini–Huygens), it was often regretted. Spacecraft can also be intently launched with reduced functionality and receive planned post-launch updates later. This can save valuable onboard computing resources (e.g., Mars rover Perseverance [114]) or reduce mission preparation times. Yet inflight updating can also add risks (e.g., hardware cannot be reached anymore in case something goes wrong), ongoing pressure on developers (e.g., Beresheet), and increased complexity of software.

Software as a first-class citizen: Software and software engineering, in particular, were and sometimes still are underestimated. For a long time, they played only a minor role in the perception of aerospace engineering. While there are indications that this perception is changing (cf. Section 1), it is in our experience not entirely certain that public (and also

private) decision-makers have truly understood its importance. Generally, a lack of software engineers in key leadership positions is problematic [11]. However, it is clear that with the increasing impact of software, the knowledge and application of software engineering methods are vital and must evolve to keep up with the ever-growing complexity of software. Software engineering must learn from the past and critically question and improve methods for the future. Yet this can only happen if software engineering is given the necessary recognition and importance. Furthermore, some failures are too easily blamed on software because the final wrong decision was made inside the software. However, there is a chain of events originating from system-level decisions that makes it all but impossible to develop smarter software. Software engineering aspects of missions must be considered in the early phases of mission analysis and definition.

New methods: The human body of knowledge is constantly expanding. New methods appear continuously and challenge the old ways. Examples are FBC, and more recently New Space (e.g., Beresheet) or agile software development (the latter is a cornerstone of the New Space mindset, cf. [6]). Yet famous computer scientist Barry Boehm already noted that new methods like agile development must be approached with care (cf. [115]). Similarly, when asked about agile development, ESA's former lead software product assurance engineer Lothar Winzer noted that agile development sounded sexy, but "hope" was not a new product assurance method (cf. [116]). Another example is auto-coding, which may help to reduce complexity and failures. For instance, model-driven software development generates code from formal models (e.g., [117]), and tools like Matlab/Simulink are mainstream. Auto-coding based on AI large language models might revolutionize development, and software reliability growth models might allow researchers to better predict where reliability is. But the new ways are not necessarily always superior. They might be of or evolve to higher efficiency based on consequent analysis, and this needs to be approached.

Limited transparency: Many space activities, and mishaps in particular, are not made public. Very often, actors get away with a black eye, so they make no big deal of it. Sometimes actors just do not have the resources to investigate what happened, or it is not deemed worthwhile (e.g., small satellites). Commercialization and militarization drive the "secretization" of spaceflight. Therefore, the limitation of transparency is no surprise. Yet, for software engineering of space systems, this situation is unfavorable because much can be learned from accidents.

Cybersecurity: Finally, cyberattacks have not been discussed here at all. But, in fact, the topic is a growing concern and may have a serious impact on mission success in the future or cause accidents. Fritz [118], for example, discusses 20 jamming and countless eavesdropping incidents, along with 12 hijacking incidents where satellites were made to transfer an illegitimate hacker's signal and, perhaps most critically, 5 incidents where hackers took control of the spacecraft. For instance, hackers caused permanent damage to the imager of the US–German ROSAT satellite in 1998. The attack is attributed to an allegedly Russia-borne cyber-intrusion at the NASA Goddard Space Flight Center [118]. A more recent incident was the hacking of the Viasat company's communication network shortly before Russia's invasion of Ukraine that also shut down thousands of wind power plants across Europe [119]. Willbold et al. [3] highlight that historically, satellite developers have often relied on the principle of security through obscurity. For instance, the developers of the Iridium network argued that its complexity was too challenging for potential attackers, which has been proven wrong.

Competent space actors: Actors of spaceflight include industry, research institutes, and agencies. A fundamental idea of FBC was limited trust in government organizations' own capabilities, leading to outsourcing to the industry (e.g., Lewis spacecraft). While it is true that government organizations have their own issues, it is essential to remember that the industry also faces limitations. Knowledgeable and experienced personnel are crucial to all actors (cf. [64]). Standards like ECSS represent codified knowledge of best practices, although rationale may sometimes be lost, which is an issue. Since standards are often perceived as, and sometimes prove to be, inflexible and present significant challenges

for startups (e.g., [6]), they have to be worked on continuously, with competence (cf. [2]). Competent actors also have the right processes established and a positive attitude toward quality. The latter will manifest in separate software engineering and software product assurance units, each with their particular philosophy (cf. [2]). The German Space Agency at DLR, for example, has reacted to the growing importance of software and software technologies by founding the group Digitalization, Software and AI (DiSoKI). In addition to coordinating technology development, it provides expertise to the agency and supports the development of ECSS standards.

Author Contributions: Conceptualization, R.G. (Ralf Gerlich) and R.G. (Rainer Gerlich); methodology, R.G. (Ralf Gerlich) and R.G. (Rainer Gerlich); validation, R.G. (Ralf Gerlich), R.G. (Rainer Gerlich) and C.R.P.; formal analysis, R.G. (Ralf Gerlich); investigation, R.G. (Ralf Gerlich) and C.R.P.; resources, R.G. (Rainer Gerlich) and C.R.P.; data curation, R.G. (Ralf Gerlich) and C.R.P.; writing—original draft preparation, C.R.P.; writing—review and editing, R.G. (Rainer Gerlich), R.G. (Ralf Gerlich) and C.R.P.; visualization, C.R.P.; supervision, C.R.P.; project administration, R.G. (Ralf Gerlich); funding acquisition, C.R.P. All authors have read and agreed to the published version of the manuscript.

Funding: This research was subcontracted to GSSE by AUDENS Telecommunications Consulting GmbH for the German Space Agency at DLR on behalf of the Federal Ministry of Economic Affairs and Climate Action (BMWK) as part of the Heinrich Hertz Mission.

Acknowledgments: The authors are grateful to the Heinrich Hertz Mission project manager, the project team at DLR, and AUDENS Telecommunications Consulting GmbH for making the ATHH project possible. The content of the manuscript was created without the help of AI. However, ChatGPT 3.5 was used for an initial translation of preexisting passages of non-English text from internal project reports that covered several spacecraft failure descriptions (Section 3) and for intermediate proofreading of paragraphs of text, i.e., for correcting language that would sound poor to native speakers (across all sections). All such text was subsequently checked manually and corrected manually where necessary. The final compilation and proofreading of the manuscript were performed manually.

Conflicts of Interest: Rainer Gerlich is the owner of GSSE. GSSE was a subcontractor in the context of the DLR Space Agency's Heinrich Hertz mission, directly reporting to DLR. The contract ended before the Heinrich Hertz satellite's launch in 2023. Ralf Gerlich was working for GSSE during initial investigations for this Entry. He is now a professor at Furtwangen University.

Appendix A

This appendix lists some of the abbreviations that are commonly used among space exploration engineers but which might not be known to readers from other domains. Note that usage depends on the concrete mission and local spaceflight culture. The terms are not defined unambiguously.

Table A1. Some of the abbreviations that are commonly used among space exploration engineers.

ACS	Attitude control system	The ACS subsystem includes sensors, actuators, and software for onboard attitude and orbit control [33].
AOCS	Attitude and orbit control System	The AOCS combines the ADCS and GNC [34].
ADCS	Attitude determination and control system	The ADCS subsystem keeps the spacecraft in the desired pointing attitudes, stabilizing it against external disturbance torques [34,83].
CDH	Command and data handling	The CDH subsystem listens to commands from the ground segment to distribute them to respective other subsystems, and it gathers and delivers status information about the spacecraft for onboard processing and to the ground segment [34]. See also OBDH.

Table A1. *Cont.*

COTS	Commercial off the shelf	Software (but also hardware) that is not custom made or bespoke for the concrete project but is a commercially available product.
GNC/GN&C	Guidance, navigation, and control	The GNC combines the ACS, the propulsion system, and software for on-orbit flight dynamics (like trajectory planning and determination, navigation, and attitude determination). This may include ground segment software [33,34].
FDIR	Failure detection, isolation, and recovery	A set of techniques and procedures to detect when a fault or anomaly occurs, isolate its cause or location, and implement recovery actions to mitigate the issue. FDIR ensures the reliability, safety, and continued operation of spacecraft, especially in autonomous or remote environments where human intervention may be limited.
OBDH	Onboard data handling	The OBDH transfers data between different subsystems, manages it, and includes or interfaces with TM/TC, TTC, etc. See also CDH.
SLOC	Source lines of code	A line of code that is neither empty nor comment. Provides rough estimates for software size.
TACS	Trajectory and attitude control system	Not precisely defined by [30] but presumably the ACS of a launch vehicle.
TM/TC	Telemetry/telecommand	Telemetry and telecommand provide downlink and uplink communication, respectively, between the spacecraft's CDH and the ground station. See also TTC.
TTC, TT&C	Telemetry, tracking, and command	The TT&C subsystem is the interface between space and the ground segment. It delivers housekeeping data about the status of the system, tracks the ground station to keep up the radio link, and receives control commands [34].
V&V	Verification and validation	*Verification* checks that the product is built right, e.g., designed and produced according to its specifications and is free of defects (cf. [120]). *Validation* checks that the right product is built, e.g., that it is able to accomplish its intended use in the intended operational environment (cf. [120]).

References

1. Deutscher Bundestag. Raumfahrtstrategie der Bundesregierung: Plenarprotokoll 20/127, 11. Oktober 2023, Berlin. 2023. Available online: https://dserver.bundestag.de/btp/20/20127.pdf (accessed on 29 March 2024).
2. Prause, C.R.; Bibus, M.; Dietrich, C.; Jobi, W. Software product assurance at the German space agency. *J. Softw. Evol. Process* **2016**, *28*, 744–761. [CrossRef]
3. Willbold, J.; Schloegel, M.; Vögele, M.; Gerhardt, M.; Holz, T.; Abbasi, A. Space Odyssey: An Experimental Software Security Analysis of Satellites. In Proceedings of the IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–25 May 2023; IEEE Computer Society: Washington, DC, USA, 2023; pp. 1–19, ISBN 978-1-6654-9336-9.
4. Everett, D.F. Overview of Spacecraft Design (Chapter 14). In *Space Mission Engineering: The New SMAD*; Wertz, J.R., Everett, D.F., Puschell, J.J., Eds.; Microcosm Publishing: Portland, OR, USA, 2011; pp. 397–438. ISBN 978-881-883-15-9.
5. Belady, L.A. Software is the glue in large systems. *IEEE Commun. Mag.* **1989**, *27*, 33–36. [CrossRef]
6. Horn, R.; Frischauf, N.; Baumann, I.; Heinrich, O. NewSpace—Geschäftsmodelle an der Schnittstelle von Raumfahrt und digitaler Wirtschaft: Chancen für Deutschland in einer vernetzten Welt. 2016. Available online: https://www.bmwk.de/Redaktion/DE/Publikationen/Studien/bmwi-new-space-geschaeftsmodelle-an-der-schnittstelle-von-raumfahrt-und-digitaler-wirtschaft.pdf?__blob=publicationFile&v=1 (accessed on 29 March 2024).
7. Harland, D.M.; Lorenz, R.D. *Space Systems Failures: Disasters and Rescues of Satellites, Rocket and Space Probes*; Springer Praxis: Dordrecht, The Netherland, 2005; ISBN 0387215190.
8. Wertz, J.R.; Conger, R.C.; Rufer, M.; Sarzi-Amadé, N.; van Allen, R.E. Methods for Achieving Dramatic Reductions in Space Mission Cost Reductions in Space Mission Cost. In Proceedings of the AIAA Reinventing Space Conference, Los Angeles, CA, USA, 2–5 March 2011; pp. 1–18.

9. Rechtin, E. *Remarks on Reducing Space Science Mission Costs. Reducing the Costs of Space Science Research Missions*; National Academies Press: Washington, DC, USA, 1997; ISBN 978-0-309-05829-2.
10. Smith, M. NASA Safety Panel: Second Starliner OFT Software Error could have been "Catastrophic". *SpacePolicyOnline.com*. 6 February 2020. Available online: https://spacepolicyonline.com/news/nasa-safety-panel-second-starliner-oft-software-error-could-have-been-catastrophic/ (accessed on 29 March 2024).
11. Dvorak, D.L. (Ed.) *NASA Study on Flight Software Complexity: Final Report*; Jet Propulsion Laboratory: Pasadena, CA, USA, 2009.
12. Avizienis, A.; Laprie, J.-C.; Randell, B.; Landwehr, C. Basic concepts and taxonomy of dependable and secure computing. *IEEE Trans. Dependable Secur. Comput.* **2004**, *1*, 11–33. [CrossRef]
13. Leveson, N.G. Role of Software in Spacecraft Accidents. *J. Spacecr. Rocket.* **2004**, *41*, 564–575. [CrossRef]
14. Orrego, A.S.; Mundy, G.E. A study of software reuse in NASA legacy systems. *Innov. Syst. Softw. Eng.* **2007**, *3*, 167–180. [CrossRef]
15. Dijkstra, E.W. The humble programmer. *Commun. ACM* **1972**, *15*, 859–866. [CrossRef]
16. Blanchette, S. Giant Slayer: Will You Let Software be David to Your Goliath System? *J. Aerosp. Inf. Syst.* **2016**, *13*, 407–417. [CrossRef]
17. Matevska, J. Software Engenierung ist (k)eine Raumfahrtdisziplin [orally, in German, translated by the authors]. In Proceedings of the Digitalisierung der Raumfahrt, Düsseldorf, Germany, 4 May 2023.
18. Apgar, H. Cost Estimating (Chapter 11). In *Space Mission Engineering: The New SMAD*; Wertz, J.R., Everett, D.F., Puschell, J.J., Eds.; Space Technology Library: Omaha, NE, USA, 2011; ISBN 978-881-883-15-9.
19. Hoare, C.A.R. An Axiomatic Basis for Computer Programming. In *Program Verification: Fundamental Issues in Computer Science*; Colburn, T.R., Ed.; Springer: Dordrecht, Germany, 1993; pp. 83–96, ISBN 978-94-010-4789-0.
20. Prause, C.; Soltau, U. Brains of Missions: Without Software Space Technology Could not Run Successfully. 2016. Available online: https://elib.dlr.de/104622/1/Prause%20-%20Gehirne%20von%20Raumfahrtmissionen.pdf (accessed on 29 March 2024).
21. Holzmann, G.J. Conquering Complexity. *Computer* **2007**, *40*, 111–113. [CrossRef]
22. Newman, J.S. Failure-Space: A Systems Engineering Look At 50 Space System Failures. *Acta Astronaut.* **2001**, *48*, 517–527. [CrossRef]
23. Prokop, L.E. Historical Aerospace Software Errors Categorized to Influence Fault Tolerance. In Proceedings of the 45th International IEEE Aerospace Conference, Big Sky, MT, USA, 2–9 March 2024; IEEE: Piscataway, NJ, USA, 2024.
24. MacKenzie, D. A View from the Sonnenbichl: On the Historical Sociology of Software and System Dependability. In Proceedings of the History of computing: Software issues; International Conference on the History of Computing, ICHC 2000, Paderborn, Germany, 5–7 April 2000; Heinz-Nixdorf-MuseumsForum. Hashagen, U., Ed.; Springer: Berlin/Heidelberg, Germany, 2002; ISBN 978-3-540-42664-6.
25. Clark, S. Humanity's Most Distant Space Probe Jeopardized by Computer Glitch. Available online: https://arstechnica.com/space/2024/02/humanitys-most-distant-space-probe-jeopardized-by-computer-glitch/ (accessed on 16 March 2024).
26. Swartwout, M.; Jayne, C. University-Class Spacecraft by the Numbers: Success, Failure, Debris. (But Mostly Success.). In Proceedings of the 30th Annual AIAA/USU Conference on Small Satellites, Logan, UT, USA, 6–11 August 2016; pp. 1–18.
27. Tomei, E.J.; Chang, I.-S. 51 Years of Space Launches and Failures: IAC-09-D1.5.1. In Proceedings of the 60th International Astronautical Congress, Daejeon, Republic of Korea, 12–16 October 2009.
28. Biswal, M.M.K.; Annavarapu, R.N. Mars Missions Failure Report Assortment: Review and Conspectus. In Proceedings of the AIAA Propulsion and Energy 2020 Forum, Virtual Event, 24–26 August 2020; American Institute of Aeronautics and Astronautics: Reston, VA, USA, 2020; ISBN 978-1-62410-602-6.
29. Chang, I.-S. Investigation of space launch vehicle catastrophic failures. *J. Spacecr. Rocket.* **1996**, *33*, 198–205. [CrossRef]
30. Fernández, L.A.; Wiedemann, C.; Braun, V. Analysis of Space Launch Vehicle Failures and Post-Mission Disposal Statistics. *Aerotec. Missili Spaz.* **2022**, *101*, 243–256. [CrossRef]
31. Gorbenko, A.; Kharchenko, V.; Tarasyuk, O.; Zasukha, S. A Study of Orbital Carrier Rocket and Spacecraft Failures: 2000-2009. *Inf. Secur. Int. J.* **2012**, *28*, 179–198. [CrossRef]
32. Kattakuri, V.; Panchal, J.H. Spacecraft Failure Analysis from the Perspective of Design Decision-Making. In Proceedings of the 39th Computers and Information in Engineering Conference, ASME 2019 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference, Anaheim, CA, USA, 8–21 August 2019; American Society of Mechanical Engineers: New York, NY, USA, 2020. ISBN 978-0-7918-5917-9.
33. Robertson, B.; Stoneking, E. Satellite GN&C Anomaly Trends. In Proceedings of the 26th Annual AAS Guidance and Control Conference, Breckenridge, CO, USA, 5–9 February 2003.
34. Tafazoli, M. A study ofon-orbit spacecraft failures. *Acta Astronaut.* **2009**, *35*, 195–205. [CrossRef]
35. Jacklin, S.A. Small-Satellite Mission Failure Rates, 2019. The National Aeronautics and Space Administration Website. Available online: https://ntrs.nasa.gov/api/citations/20190002705/downloads/20190002705.pdf (accessed on 29 March 2024).
36. Chertok, B. *Rockets and People: Creating a Rocket Industry*; NASA History Division: Washington, DC, USA, 2006.
37. Tukey, J.W. The Teaching of Concrete Mathematics. *Am. Math. Mon.* **1958**, *65*, 1. [CrossRef]
38. Mudgway, D.J. Telecommunications and Data Acquisition Systems Support for the Viking 1975 Mission to Mars: The Viking Lander Monitor Mission May 1980 to March 1983. 1983. Available online: https://atmos.uw.edu/~mars/viking/lander_documents/meteorology/Pdf/JPL_Publication_82-107.pdf (accessed on 29 March 2024).

39. NASA Space Science Data Coordinated Archive. Viking 1 Orbiter: NSSDCA/COSPAR ID: 1975-075A. Available online: https://nssdc.gsfc.nasa.gov/nmc/spacecraft/display.action?id=1975-075A (accessed on 29 March 2024).
40. Waldrop, M.M. Phobos at Mars: A Dramatic View--and Then Failure. *Science* **1989**, *245*, 1044–1045. [CrossRef]
41. Sagdeev, R.Z.; Zakharov, A.V. Brief history of the Phobos mission. *Nature* **1989**, *341*, 581–585. [CrossRef]
42. Rao, U.R.; Gupta, S.C.; Madhavan Nair, G.; Narayana Moorthi, D. PSLV-D1 mission. *Curr. Sci.* **1993**, *7*, 522–528.
43. Nagappa, R. Development of Space Launch Vehicles in India. *Astropolitics* **2016**, *14*, 158–176. [CrossRef]
44. National Research Council. *Lessons Learned from the Clementine Mission*; National Academies Press: Washington, DC, USA, 1997; ISBN 978-0-309-05839-1.
45. NASA APPEL News Staff. This Month in NASA History: A Software Error Took Clementine for a Spin. Available online: https://appel.nasa.gov/2017/05/12/this-month-in-nasa-history-a-software-error-took-clementine-for-a-spin/ (accessed on 29 March 2024).
46. Schilling, W.; Alam, M. A methodology for quantitative evaluation of software reliability using static analysis. In Proceedings of the 2008 Annual Reliability and Maintainability Symposium, Las Vegas, NV, USA, 28–31 January 2008; IEEE: Piscataway, NJ, USA, 2008; pp. 399–404, ISBN 978-1-4244-1460-4.
47. McCurdy, H.E. *Faster, Better, Cheaper: Low-Cost Innovation in the U.S. Space Program*; Johns Hopkins University Press: Baltimore, MD, USA, 2001; ISBN 0-8018-6720-7.
48. Beck, K.; Beedle, M.; van Bennekum, A.; Cockburn, A.; Cunningham, W.; Fowler, M.; Grenning, J.; Highsmith, J.; Hunt, A.; Jeffries, R.; et al. Manifesto for Agile Software Development. 2001. Available online: https://agilemanifesto.org/ (accessed on 29 March 2024).
49. Dowson, M. The Ariane 5 software failure. *SIGSOFT Softw. Eng. Notes* **1997**, *22*, 84. [CrossRef]
50. Lions, J.-L.; Lübeck, L.; Fauquembergue, J.-L.; Kahn, G.; Kubbat, W.; Levedag, S.; Mazzini, L.; Merle, D.; O'Halloran, C. *Ariane 501 Inquiry Board Report*; Inquiry Board: Paris, France, 1996.
51. Le Lann, G. An analysis of the Ariane 5 flight 501 failure-a system engineering perspective. In Proceedings of the International Conference and Workshop on Engineering of Computer-Based Systems, Monterey, CA, USA, 24–28 March 1997; Rozenblit, J., Ed.; IEEE Computer Society Press: Los Alamitos, CA, USA, 1997; pp. 339–346, ISBN 0-8186-7889-5.
52. Lacan, P.; Monfort, J.N.; Le Ribal, V.Q.; Deutsch, A.; Gonthier, G. ARIANE 5—The Software Reliability Verification Process. In Proceedings of the Conference on Data Systems in Aerospace (DASIA 1998), Athens, Greece, 25–28 May 1998; Kaldeich-Schürmann, B., Ed.; pp. 201–205.
53. Trella, M.; Greenfield, M.; Herring, E.L.; Credland, J.; Freeman, H.R.; Laine, R.; Kilpatrick, W.; Machi, D.; Reth, A.; Smith, A. SOHO Mission Interruption: Joint ESA/NASA Investigation Board Report. 1998. Available online: https://umbra.nascom.nasa.gov/soho/SOHO_final_report.html (accessed on 29 March 2024).
54. Kissler Patig, M. Extended life for ESA's Science Missions. Available online: https://sci.esa.int/web/director-desk/-/extended-life-for-esa-s-science-missions (accessed on 29 March 2024).
55. Vandenbussche, F.C. SOHO's Recovery: An Unprecedented Success Story. *ESA Bulltin* **1999**, *97*, 39.
56. Reichhardt, T. Rescued satellite to get more managers. *Nature* **1998**, *396*, 399. [CrossRef]
57. Johns Hopkins University Applied Physics Laboratory. Frequently Asked Questions: Near Earth Asteroid Rendezvous. Available online: https://near.jhuapl.edu/intro/faq.html (accessed on 10 May 2024).
58. Hoffman, E.J.; Gay, C.J.; Ebert, W.L.; Jones, C.P.; Femiano, M.D.; Luers, P.J.; Freeman, H.R.; Palmer, J.G. *The NEAR Rendezvous Burn Anomaly of December 1998: Final Report of the NEAR (Near Earth Asteroid Rendezvous) Anomaly Review Board*; Johns Hopkins University Applied Physics Laboratory: Laurel, MD, USA, 1999.
59. NASA Space Science Data Coordinated Archive. Mars Pathfinder Rover. Available online: https://nssdc.gsfc.nasa.gov/nmc/spacecraft/display.action?id=MESURPR (accessed on 11 March 2024).
60. Jones, M. What really happened on Mars Rover Pathfinder. Available online: http://www.cs.cornell.edu/courses/cs614/1999sp/papers/pathfinder.html (accessed on 11 March 2024).
61. Wander, S.M. Lewis Spins Out of Control. System Failures Case Studies. 2007. Available online: https://sma.nasa.gov/docs/default-source/safety-messages/safetymessage-2007-11-01-lossofthelewisspacecraft.pdf?sfvrsn=89a91ef8_4 (accessed on 29 March 2024).
62. Wade, M. Lewis Satellite. Available online: http://www.astronautix.com/l/lewissatellite.html (accessed on 2 May 2024).
63. Anderson, C.; Vanek, C.S.; Freeman, H.R.; Furlong, D.; Kirschbaum, A.; Roy, R.; Wilhelm, P.; Wander, S. Lewis Spacecraft Mission Failure Investigation Board. 1998. Available online: https://spacese.spacegrant.org/Failure%20Reports/Lewis_MIB_2-98.pdf (accessed on 29 March 2024).
64. Paxton, L.J. "Faster, better, and cheaper" at NASA: Lessons learned in managing and accepting risk. *Acta Astronaut.* **2007**, *61*, 954–963. [CrossRef]
65. Kyle, E. Thunder Lost—The Delta 3 Story: Thirteenth in a Series Reviewing Thor Family History. Available online: https://web.archive.org/web/20220321061514/https://www.spacelaunchreport.com/thorh13.html (accessed on 7 May 2024).
66. Meissinger, H.F.; Dawson, S. Reducing planetary mission cost by a modified launch mode. *Acta Astronaut.* **1999**, *45*, 533–540. [CrossRef]

67. Go, S.; Lawrence, S.L.; Mathias, D.L.; Powell, R. Mission Success of U.S. Launch Vehicle Flights from a Propulsion Stage-Based Perspective: 1980-2015 (NASA/TM-2017-219497). 2017. Available online: https://ntrs.nasa.gov/api/citations/20170009844/downloads/20170009844.pdf (accessed on 7 May 2024).
68. Wunderlich-Pfeiffer, F. In den Neunzigern stürzte alles ab: Softwarefehler in der Raumfahrt. Available online: https://www.golem.de/news/softwarefehler-in-der-raumfahrt-in-den-neunzigern-stuerzte-alles-ab-1511-117537.html (accessed on 7 May 2024).
69. Harwood, W. Military Satellite in Wrong Orbit: Failure is Third Straight for Air Force's Titan IV Rocket. *The Washington Post*. 30 April. Available online: https://www.washingtonpost.com/archive/politics/1999/05/01/military-satellite-in-wrong-orbit/99803c3b-03b3-4758-bab0-4522e6ee0961/ (accessed on 7 May 2024).
70. Stephenson, A.G.; Mulville, D.R.; Bauer, F.H.; Dukeman, G.A.; Norvig, P.; LaPiana, L.S.; Rutledge, P.J.; Folta, D.; Sackheim, R. Mars Climate Orbiter Mishap Investigation Board Phase 1 Report. 1999. Available online: https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf (accessed on 4 March 2024).
71. Johnson, C.W. The Natural History of Bugs: Using Formal Methods to Analyse Software Related Failures in Space Missions. In *Formal Methods: FM 2005, Proceedings of the International Symposium of Formal Methods Europe, Newcastle, UK, 18–22 July 2005*; Hutchison, D., Kanade, T., Kittler, J., Kleinberg, J.M., Mattern, F., Mitchell, J.C., Naor, M., Nierstrasz, O., Pandu Rangan, C., Steffen, B., et al., Eds.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 9–25. ISBN 978-3-540-27882-5.
72. Leopold, G. Software Won't Fix Boeing's 'Faulty' Airframe. Available online: https://www.eetimes.com/software-wont-fix-boeings-faulty-airframe/ (accessed on 11 March 2024).
73. Abbott, A. Battery fault ends X-ray satellite mission. *Nature* **1999**, *399*, 93. [CrossRef]
74. Link, D.C.R.; Anne, J.C.; Beretta, A.; Dechezelles, J.J.; Gluitz, K.J.; Jablonski, A.; Draper, R.F.; Horttor, R.L.; Bonnefoy, R. Huygens Communications Link Enquiry Board Report: Findings, Recommendations and Conclusions. 2000. Available online: https://sci.esa.int/web/cassini-huygens/-/25652-huygens-communications-link-enquiry-board-report (accessed on 29 March 2024).
75. Martinez, C.; Savage, D. Cassini Peeks Below Cloud Shroud Around Titan. Available online: https://www.jpl.nasa.gov/news/cassini-peeks-below-cloud-shroud-around-titan (accessed on 11 March 2024).
76. Belous, A.; Saladukha, V.; Shvedau, S. *Modern Spacecraft Classification, Failure, and Electrical Component Requirements*; Artech House: Boston, UK, 2017; ISBN 978-1630812577.
77. Ray, J. Sea Launch Malfunction Blamed on Software Glitch. Available online: https://spaceflightnow.com/sealaunch/ico1/000330software.html (accessed on 8 March 2024).
78. Reeves, G.; Neilson, T. The Mars Rover Spirit FLASH anomaly. In Proceedings of the IEEE Aerospace Conference, Big Sky, MT, USA, 5–12 March 2005; pp. 4186–4199.
79. Briggs, H. Cryosat Rocket Fault Laid Bare. Available online: http://news.bbc.co.uk/2/hi/science/nature/4381840.stm (accessed on 13 March 2024).
80. Huckle, T.; Neckel, T. *Bits and Bugs: A Scientific and Historical Review of Software Failures in Computational Science*; SIAM Society for Industrial and Applied Mathematics: Philadelphia, PA, USA, 2019; ISBN 9781611975550.
81. ESA. CryoSat Mission Lost Due to Launch Failure. Available online: https://www.esa.int/Applications/Observing_the_Earth/FutureEO/CryoSat/CryoSat_Mission_lost_due_to_launch_failure (accessed on 13 March 2024).
82. Bell, M. Lesson 1805: Mars Global Surveyor (MGS) Spacecraft Loss of Contact. 2007. Available online: https://llis.nasa.gov/lesson/1805 (accessed on 29 March 2024).
83. Finley, C.; Peck, N. TacSat-2: A Story of Survival. In Proceedings of the 21st Annual AIAA/USU Conference on Small Satellites, Logan, UT, USA, 13–16 August 2007; pp. 1–16.
84. Barisic, S. Software Glitch Delays Commercial Spaceport's First Launch. Available online: https://www.space.com/3226-software-glitch-delays-commercial-spaceport-launch.html (accessed on 8 March 2024).
85. Clark, S. Russia: Computer crash doomed Phobos-Grunt. Available online: https://spaceflightnow.com/news/n1202/06phobosgrunt/ (accessed on 13 March 2024).
86. Oberg, J. Did Bad Memory Chips Down Russia's Mars Probe?: Moscow Blames Radiation Wreckage on an SRAM Chip, but Does It Add Up? Available online: https://spectrum.ieee.org/did-bad-memory-chips-down-russias-mars-probe (accessed on 13 March 2024).
87. Frazier, S. Saving NASA's STEREO-B—The 189-Million-Mile Road to Recovery. Available online: https://phys.org/news/2015-12-nasa-stereo-bthe-million-mile-road-recovery.html (accessed on 13 March 2024).
88. Kucera, T.A. STEREO-B Status Update. Available online: https://stereo-ssc.nascom.nasa.gov/behind_status.shtml (accessed on 13 March 2024).
89. Bergin, C. Saving Spaceship Dragon—Software to Provide Contingency Chute Deploy. Available online: https://www.nasaspaceflight.com/2015/07/saving-spaceship-dragon-contingency-chute/ (accessed on 29 January 2024).
90. Clark, S. Japan's Hitomi Observatory Made Cosmic Discovery before Failing. Available online: https://spaceflightnow.com/2016/07/07/japans-hitomi-observatory-made-cosmic-discovery-before-failing/ (accessed on 15 March 2024).
91. Witze, A. Software error doomed Japanese Hitomi spacecraft. *Nature* **2016**, *533*, 18–19. [CrossRef] [PubMed]
92. JAXA. Hitomi Experience Report: Investigation of Anomalies Affecting the X-ray Astronomy Satellite "Hitomi" (ASTRO-H). 2016. Available online: https://global.jaxa.jp/projects/sat/astro_h/files/topics_20160524.pdf (accessed on 29 March 2024).

93. AFP. European craft crashed on Mars, possibly exploded: ESA. Available online: https://phys.org/news/2016-10-european-craft-mars-possibly-esa.html (accessed on 15 March 2024).
94. Tolker-Nielsen, T. EXOMARS 2016—Schiaparelli Anomaly Inquiry. 2017. Available online: https://sci.esa.int/documents/33431/35950/1567260317467-ESA_ExoMars_2016_Schiaparelli_Anomaly_Inquiry.pdf (accessed on 15 March 2024).
95. Dambowsky, F.; Eßmann, O.; Hauslage, J.; Berger, T. Abschied von Mission Eu:CROPIS. Available online: https://www.dlr.de/de/aktuelles/nachrichten/2020/01/20200113_abschied-von-mission-eucropis (accessed on 13 March 2024).
96. Kottmeier, S.; Hobbie, C.F.; Orlowski-Feldhusen, F.; Nohka, F.; Delovski, T.; Morfill, G.; Grillmayer, L.; Philpot, C.; Müller, H. The Eu:Cropis Assembly, Integration and Verification Campaigns: Building the first DLR Compact Satellite. In Proceedings of the 69th International Astronautical Congress, Bremen, Germany, 1–5 October 2018.
97. Shyldkrot, H.; Shmidt, E.; Geron, D.; Kronenfeld, J.; Loucks, M.; Carrico, J.; Policastri, L.; Taylor, J. The First Commercial Lunar Lander Mission: Beresheet (AAS 19-747). In Proceedings of the AAS/AIAA Astrodynamics Specialist Conference, Portland, ME, USA, 11–15 August 2019.
98. Clark, S. Errant Command Doomed Israeli Moon Lander, Officials Vow to Try Again. Available online: https://spaceflightnow.com/2019/04/18/errant-command-doomed-israeli-moon-lander-officials-vow-to-try-again/ (accessed on 30 April 2024).
99. Nevo, E. What Happened to Beresheet? 2020. Weizmann Institute of Science. Available online: https://davidson.weizmann.ac.il/en/online/sciencepanorama/what-happened-beresheet (accessed on 29 March 2024).
100. Gohd, C. Boeing's Starliner Won't Reach Space Station After Launch Anomaly, NASA Chief Says. Available online: https://www.space.com/boeing-starliner-oft-fails-to-reach-correct-orbit.html (accessed on 15 March 2024).
101. Chang, K. Boeing Starliner Flight's Flaws Show 'Fundamental Problem', NASA Says. Available online: https://www.nytimes.com/2020/02/07/science/boeing-starliner-nasa.html (accessed on 15 March 2024).
102. Foust, J. Starliner concludes OFT-2 test flight with landing in New Mexico. Available online: https://spacenews.com/starliner-concludes-oft-2-test-flight-with-landing-in-new-mexico/ (accessed on 19 March 2024).
103. Weitering, H. Boeing's 2nd Starliner Software Glitch Could Have Led to an in-Space Collision, 2020. *Space*. Available online: https://www.space.com/boeing-starliner-2nd-software-glitch-potential-collision.html (accessed on 15 March 2024).
104. Fernholz, T. Boeing's Spacecraft Test Failure Points to Broader Problems. Available online: https://qz.com/1799365/how-boeings-starliner-test-failed (accessed on 15 March 2024).
105. Weitering, H. NASA Completes Investigation on Flawed Boeing Starliner Capsule Test Flight. Available online: https://www.space.com/nasa-boeing-starliner-test-flight-investigation-complete.html (accessed on 19 March 2024).
106. Swinney, R.W.; Freeland, R.M., II; Lamontagne, M. Project Icarus: Designing a Fusion Powered Interstellar Probe. *Acta Futura* **2020**, *12*, 47–59. [CrossRef]
107. Dulo, D.A. Software or the Borg: A Starship's Greatest Threat? Available online: https://www.space.com/29509-software-borg-starship-greatest-threat.html (accessed on 20 March 2024).
108. Bourque, P.; Fairley, R.E. *Guide to the Software Engineering Body of Knowledge: Swebok, Version 3.0*; IEEE Computer Society: Los Alamitos, CA, USA, 2014; ISBN 9780769551661.
109. Bertram, D.; Voida, A.; Greenberg, S.; Walker, R. Communication, collaboration, and bugs. In Proceedings of the CSCW '10: Computer Supported Cooperative Work, Savannah, GA, USA, 6–10 February 2010; Inkpen, K., Ed.; ACM: New York, NY, USA, 2010; pp. 291–300, ISBN 9781605587950.
110. Carlson, J.M.; Doyle, J. Complexity and robustness. *Proc. Natl. Acad. Sci. USA* **2002**, *99* (Suppl. 1), 2538–2545. [CrossRef]
111. Ganesan, D.; Lindvall, M.; McComas, D.; Bartholomew, M.; Slegel, S.; Medina, B. Architecture-Based Unit Testing of the Flight Software Product Line. In *Software Product Lines: Going Beyond, Proceedings of the14th International Conference, SPLC 2010, Jeju Island, Republic of Korea, 13–17 September 2010*; Bosch, J., Lee, J., Eds.; Springer: Berlin/Heidelberg, Germany, 2010; pp. 256–270. ISBN 978-3-642-15578-9.
112. Gerlich, R.; Schoolmann, I.; Brüggmann, J.; Ehresmann, M.; Prause, C. AI-based Formalization of Textual Requirements. *Data System in Aerospace 2024, Opatija, Croatia*. 2024. Available online: https://www.researchgate.net/publication/381252037_AI-based_Formalization_of_Textual_Requirements (accessed on 29 March 2024).
113. Prause, C.R.; Gerlich, R. Finest Magic Cloth or a Naked Emperor? The SKQuest Data Set on Software Metrics for Improving Transparency and Quality. *Standards* **2023**, *3*, 136–168. [CrossRef]
114. Wall, M. What's next for NASA's Perseverance Mars Rover after Its Landing Success? Available online: https://www.space.com/perseverance-mars-rover-landing-next-steps (accessed on 29 March 2024).
115. Boehm, B. Get ready for agile methods, with care. *Computer* **2002**, *35*, 64–69. [CrossRef]
116. Brüggemann, S.; Prause, C.R. Status Quo agiler Software-Entwicklung in der europäischen institutionellen Raumfahrt. 2018. Available online: https://publikationen.dglr.de/?tx_dglrpublications_pi1[document_id]=480192 (accessed on 29 March 2024).
117. Ed Benowitz, K.C. Auto-Coding UML Statecharts for Flight Software. In Proceedings of the 2nd IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT'06), Pasadena, CA, USA, 17–20 July 2006; IEEE Computer Society: Los Alamitos, CA, USA, 2006; pp. 413–417, ISBN 0-7695-2644-6.
118. Fritz, J. Satellite hacking: A guide for the perplexed. Culture Mandala: The Bulletin of the Centre for East-West Cultural and Economic Studies. *Cult. Mandala* **2013**, *10*, 21–50.

119. O'Neill, P.H. Russia hacked an American satellite company one hour before the Ukraine invasion: The attack on Viasat showcases cyber's emerging role in modern warfare. *MIT Technology Review*. 2022. Available online: https://www.technologyreview.com/2022/05/10/1051973/russia-hack-viasat-satellite-ukraine-invasion/ (accessed on 29 March 2024).
120. ECSS-S-ST-00-01C. ECSS System—Glossary of Terms. 2012. Available online: https://ecss.nl/standard/ecss-s-st-00-01c-glossary-of-terms-1-october-2012/ (accessed on 29 March 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.