

A Succinct Study on Infamous Cases of Requirements Engineering Failure

Diya Burman

Abstract—This report looks at the impact of inadequate requirements engineering on software development through the study of two prominent instances of software failures/bugs.

I. INTRODUCTION

Requirements Engineering is the process of identifying, defining, documenting and managing the requirements for a software based system. The purpose of any requirements engineering procedure is to create a set of requirements that is holistic, complete, relevant to and consistent with what the customer wants. While a perfect set of requirements is an unattainable feat, the effort should always be to conjure up each possible user scenario, conflict, doubt and problem instance. This is essential such that the resultant software functions ideally in all the common user scenarios, and, in the event of unforeseen conditions, is robust enough to function without crashing or failing entirely on the user.

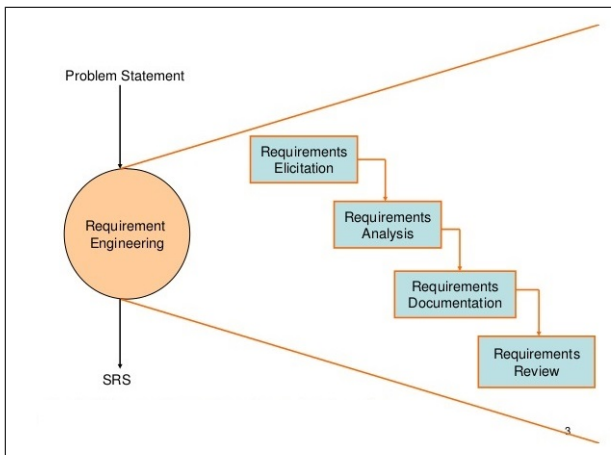


Fig. 1. Basic fundamental steps to the Requirements Engineering process

Over the course of the 90s and the early 2000s, with a rapidly increasing scale and sophistica-

tion in software, developers were seduced by a whole world of powerful programming languages and development tools which promised to make building software a pure breeze. Under this notion, programmers and management often overlook the importance of brainstorming through problems thoroughly and scrutinizing every aspect of the customer's needs and contrasting them with the developers' abilities to deliver as well as the impact of real world environment in certain cases.

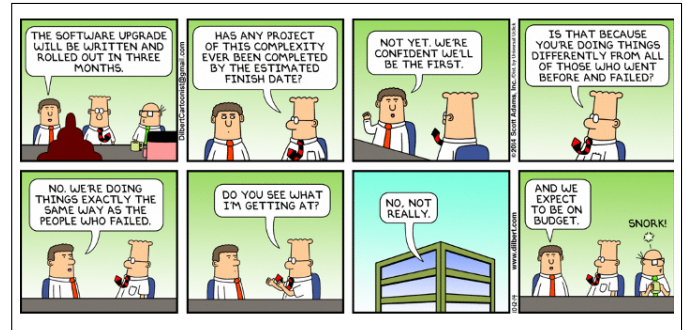


Fig. 2. Scott Adam's cartoon on how most Software Engineering Processes work

Therefore, with the advent of software, also rose the cases of software failures due to negligence towards proper requirements specification and analysis. Two such famous instances include the explosion of The Ariane 5 rocket launched by the European Space Agency in 1996 as well as the infamous Y2K millennium bug, both of which resulted from negligence towards the system's requirements analysis and design basics phase costing respective companies millions of dollars in losses.

II. THE Y2K MILLENNIUM BUG

A. Background to the Bug

During the initial days of software programming, memory space was very limited in computer

systems. Computer programmers therefore tried to conserve space wherever possible while coding. When working with dates, a little space could be saved in the databases by using the last two digits of the year component of a date instead of all four digits. When judged on a cumulative basis, such a practice led to significant savings in terms of memory space. The problem in the 60s and 70s there was no way for programmers to foresee the rapid advancements in hardware alongside software which would minimize cost of memory devices by a huge ratio in the coming decades[19]. However, with technology upgrading itself so fast, they hardly ever expected their software to be running 20 or 30 years from then. While this might be true, for instance, software for word-processing or embedded chips or game development might be built on legacy systems but tend to be upgraded rapidly with time. On the other hand, more average, every-day usage software for menial tasks like scheduling software, ticket booking, payroll processing etc., tend to remain in usage for long periods of time without drastic or frequent changes being made. Firms and governments order the latter group of software with long term usage in mind and they aren't changed or upgraded that frequently[6].



Fig. 3. [10] electronic sign displaying the year incorrectly as 1900 on 3 January 2000 in France

As a result when the new century approached, the probable discrepancies from this practice of using the last two digits for year-representation started showing itself and causing worry to companies and developers alike. The Y2K bug is perhaps

the most prominent of all numeric programming problems and also highlights why even best practices need re-evaluation with time and it might not always be wise to compromise on seemingly small areas of software development simply to conserve a bit of resources during development and consequent usage.

B. Elaboration of the Bug

"I'm one of the culprits who created this problem. I used to write those programs back in the 1960s and 1970s, and was proud of the fact that I was able to squeeze a few elements of space out of my program by not having to put a 19 before the year. Back then, it was very important. We used to spend a lot of time running through various mathematical exercises before we started to write our programs so that they could be very clearly delimited with respect to space and the use of capacity. It never entered our minds that those programs would have lasted for more than a few years. As a consequence, they are very poorly documented. If I were to go back and look at some of the programs I wrote 30 years ago, I would have one terribly difficult time working my way through step-by-step."

—Alan Greenspan, 1998

Fig. 4. Alan Greenspan's quote about the Y2K crisis[7]

The basis of the Y2K problem was pretty straightforward. For most programs, the 2-digit year format flags an issue when 00 is entered to represent the year. The software is unsure as to whether to interpret 00 as 1900 or 2000. Most programs assume the date to be 1900 by default. That is, programmers wrote most software where it either pre-pends 19 to the front of the two zeroes for processing purpose or it makes no assumption and the date is considered to be 1900 on all occasions where two zeroes are detected as last digits. This wouldn't have been a massive issue except that a lot of programs out there perform heavy date related calculations. For instance, for age calculation, a software will take the current date (say Nov 3, 2000) and subtract an individual's birth-date (Jan 4, 1971) from the current date. Since this individual has lived through the year 2000, if only last digits are being considered, the individual's age will be calculated as -71 instead of 39. Now throw in this use of the age against his investment interest calculations, service agreements in companies etc.

There was a further secondary aspect to the Y2K problem. A lot of programmers following the Gregorian calendar, followed the rule that a year

divisible by 100 would be a leap year. So naturally all century years were assumed to be leap years which isn't true. This is an incomplete premise. Years which are divisible by 400, and not just by 100 are in fact leap years. This makes years such as 1600, 2000 etc. leap years but 1900, 2100 etc. non-leap years. So in addition to the loopholes in the 2-digit format, the above assumption would cause issues with the next century as software rolled over from this century.

C. Facets of the Y2K Problem

In the very initial days of the software industry, a single megabyte of memory would cost thousands of dollars. Today that single megabyte is available for less than a dollar. There were more facets to the Y2K bug crisis than we care to notice. At the turn of the century, the use of two digits wasn't just limited to representation within code but even outside of it, e.g. Windows 98. There were larger systems with smaller software implementations such as embedded systems that were at an equal if not a higher risk of malfunctioning as traditional computers, but, those began to be noticed only quite late into the last decade of the century[17].

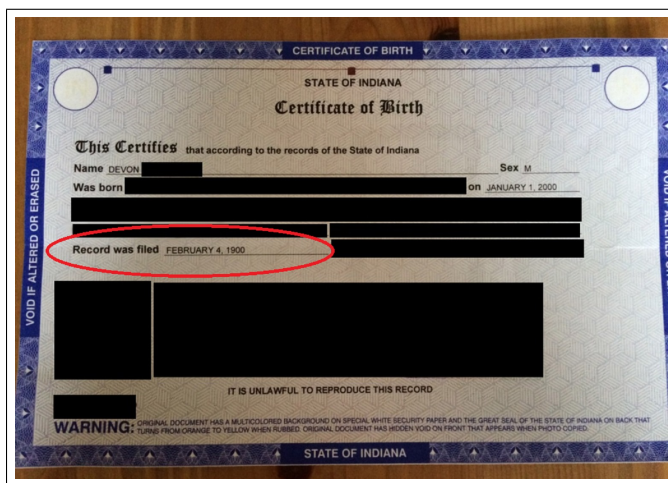


Fig. 5. The Y2K bug manifesting in important governmental and legal documents[12]

• Psychological Aspect

There was a major psychological aspect to the aftermath of the Y2K crisis which a lot of major corporations and leaders failed to realize. The *guilt factor*. For instance, applications such as spreadsheets, payroll systems

and other software of the calculative and analytical nature, all needed to be checked. Hundreds of thousands of projects were being run using such legacy software from the 70s and 80s. Managers at the turn of the Y2K were working with massive global projects. The mess created by the bug could have caused cross-cultural miscommunications and terrible miscalculations which the regular software engineer would not have been able to understand and neither his manager. Employees would play the blame game and other employees would suffer from guilt and/or denial. The work environment would suffer on top of failing software. If the very people upon whom we rely to fix our technical problems begin to feel like the ones being blamed, one can imagine what it means for a company: employees would have sought to change organizations or careers altogether taking away with them, knowledge of the local situation and its problems. Managers who for the most part manage more than one project or team and are in most cases not familiar with the tools being used on the ground, would be making erroneous decisions and further affecting, negatively, the work culture as well as the projects being managed[17].

• Embedded Systems

The Y2K problem was initially heralded as an IT problem and the resolution handed over to IT staff who had very little knowledge of other processes such as embedded systems. The Y2K bug was recognized as an issue in embedded systems quite late into the 20th century, around 1997. This is because only about 10-15 percent of embedded chips would go into computers and their accessories and the rest would go into other industrial machinery and systems. E.g. systems in utility companies that distributed electricity and water, automated control equipment, nuclear reactors, chemical plants etc. If per se, the software in embedded systems was not corrected, an electricity company wouldn't be able to produce electricity, bill its customers correctly, equipment may get shut down at a wrongful date/time and the repercussions

are endless. Worse yet, tragedies could take place in oil fields and nuclear reactors due to equipment malfunction on top of utility services being disrupted[18][17].

- **Global Economy Disruptions**

The global economy functions on the backbone of massively interconnected and interdependent markets and organizations. What companies and governments failed to realize that the fate of larger corporations would affect the fate of the smaller and medium sized ones whose business is also dependent on the larger corps buying from or selling to them. Lets take an example: suppose General Motors, a large automobile manufacturing company has thousands of suppliers. Each of these suppliers can also have multiple smaller suppliers. There are marketers, retailers, wholesalers and a bunch of other intermediaries as well. Lets suppose, each one of these suppliers and other intermediaries upgraded their software to be Y2K compliant as had General Motors itself. Now, lets assume that for some reason, at the turn of the century, General Motors was unable to meet 5 percent of its demand for supplies using its regular suppliers. In order to continue manufacturing it had to import parts from a couple of small, lesser known suppliers that had never done business with an organization as big as General Motors and naturally didnt have the state-of-the-art software that is Y2K compliant or even aware of the Y2K bug crisis. As a result of using such equipment to account for just that 5 percent, the production processes at General Motors would have gone for a toss. Millions of dollars invested would have been lost, General Motors would have been forced to call back hundreds of launched vehicles, it could have had to pay millions in compensation to families or individuals who were involved in any sort of accident because of the malfunctioning software in the automobiles; in short the company could have gone bankrupt and along with the thousands of other companies reliant on it as a major customer. The ramifications on the global economy that the Y2K bug could have had

were innumerable[15][18].

- **Government Services**

Governments were far from immune to the threat of the Y2K bug either. In USA, the federal government alone spent about 8.38 billion US dollars during a 5-year period to the Y2K-bug corrections. At company-level, e.g. Nokia Inc. used about 90 million US dollars for preventive Y2K-bug corrections[13] In 1997, the Canadian Auditor General observed the pace of repairing defective code was hastened, the government would be unable to collect taxes or make necessary payments. By December 1997, the concerns grew with finally the Canadian Police as well as the military were instructed in early 1998 to assist the general population and control the anxiety which would invariably spread in case of widespread disruption of utility services. The militia in turn responded that the government and armed were dependent on the same sources of basic resources as the civilian population and that they wouldnt be able to help/control the situation for long if recovery measures were not hastened up. Canada was ahead of most countries in recognizing the threat posed by the seemingly harmless Y2K bug and was able to recover in time to avoid a national scale panic situation[18][17].

- **Last but not least**, there could have been severe cultural implications of technology failing on such a wide scale basis. With the tremendous dependency of economies, corporations and even the average mans daily activities on technology, if companies were to go bankrupt, people were to lose money and face disruption of basic utilities, the inherent trust in technology would take a hit. People would become distrusting and cynical about technology and the industrial age would suffer a significant setback.

D. Resolution of the Bug

The Y2K bug affected different governments and corporations to different extents. As such there was no single reformative solution. Several different and on most occasions a mixture of different approaches were used to resolve the bug in legacy systems. Three of them follow:

- **Date expansion**

The two-digit system was entirely replaced by four-digit representation of year values in programs to remove any and all ambiguity of representation. This was considered the “purest” solution resulting in a permanent solution. However, implementing this solution required large scale testing and massive conversion efforts in legacy programs, affecting entire systems, thereby considered one of the costly solutions[8][10].

- **Date re-partitioning**

IE9+, Google Chrome, Firefox, Opera, Safari, etc.					
Real year	1858	1990	1994	2000	2007
.getYear() result	-42	90	94	100	107
.getFullYear() result	1858	1990	1994	2000	2007

IE6-8					
Real year	1858	1990	1994	2000	2007
.getYear() result	1858	90	94	2000	2007
.getFullYear() result	1858	1990	1994	2000	2007

Fig. 6. The JavaScript getYear() method, a classic example of the Y2K problem[10]

In legacy databases whose size could not be economically changed, six-digit year/month/day codes were converted to three-digit years (with 1999 represented as 099 and 2001 represented as 101, etc.) and three-digit days (ordinal date in year). Only input and output instructions for the date fields had to be modified, but most other date operations and whole record operations required no change. This delays the eventual roll-over problem to the end of the year 2899[10][1][2].

- **Windowing**

Two-digit years were retained, and programs determined the century value only when needed for particular functions, such as date comparisons and calculations. (The century “window” refers to the 100-year period to which a date belongs). This technique, which required installing small patches of code into programs, was simpler to test and implement than date expansion, thus much less costly. While not a permanent solution, windowing

fixes were usually designed to work for several decades. This was thought acceptable, as older legacy systems tend to eventually get replaced by newer technology[8][10].



Fig. 7. Cartoon depicting the underestimated y2k problem[20]

E. Where did we go wrong with dealing with the Y2K problem?

More often than not, blame goes to programmers and managers for software failures and unforeseen bugs. The industry seldom thinks of looking at stages prior to code development wherein lie, most often, incorrect assumptions and lack of foresightedness. Wrong questions asked and worse, no questions asked, contribute to significant logical errors in the design of a product which later on creates hassles in code. Proceeding blindly with best practices without re-evaluating them from time to time also contributes to bugs creeping in or redundancies or misuse of resources as surrounding hardware or software evolves.

The Y2K bug could in a way be considered a case of such lack of foresightedness. As one expert rightly observes on Quora: *the programmers of yesterday are the CEOs of today*. It’s interesting to observe that while there can be speculations as to how this problem could have been solved, it’s easy to speculate, quite another to actually implement actions:

- Indeed, it’s difficult to imagine as to how none of the great minds working across decades cared to ask the question – “will our software

be in use even a few decades from now?”. The answer on so many occasions should have been yes. Or was it that the question wasn’t asked because the problem only became visible over a very extended period of time rather than in a singular instance? *Requirements Analysis*. The reason behind conserving memory by saving bit space was well founded. The reduction in the cost of memory chips was so very gradual over the decades, that for the longest time, this bit saving made sense as a best practice. Thus the emergence of a problem at the turn of the century was something that wasn’t visible immediately. The change was gradual and never a case of being *clearly visible*.

As pointed out in earlier segments, software for public services and government processes are written once and used over decades. This is because mostly, such software executes a few simple tasks like data entry, data processing, ticket counting, personal details registration etc., all of which are programs that won’t change from one generation of programming to the next, at least not the fundamentals of it. Also, when the tax payers’ money is used to develop software for public and utility services, the efforts are to make software that is robust through an extended period of time. Under such cases, when programmers thought of bit-conservation through the two-digit system, why wasn’t the question asked as to how the calculations will be affected once we move from one century to the next, or from a normal year to a leap year. Will there be any repercussions? Will we encounter any multiple such figures where last 3 or last 2 digits might be similar? Will the need for bit-conservation still make sense at a future time where large memory can be cheaply acquired due to advance in hardware technology?

It is however important to note here that pointing fingers is easy because the journey between noticing a problem to implementing a solution is governed by a lot of external factors. Any legacy software goes through such large scale modifications over its lifetime that the code becomes brittle and very delicate. In

a few decades worth of time, it becomes very sensitive to even the smallest of changes being attempted. While to the user or customer it may seem that a small patch could have been released to fix the Y2K bug, such a small patch could cause large scale alterations in the software overall affecting multiple inter-dependent and interconnected modules.

- On some accounts, the blame cannot entirely be pinned on lack of foresight and/or poor design. The Y2K problem was not caused by laziness nor poor design – unless we are talking about Y2K problems in software written in the 1990’s. Most of the Y2K problem was with very old systems – mainframes and mid-range computers running COBOL or RPG code that was written as far back as the 60’s, 70’s and 80’s. Back then programmers didn’t have gigabytes of memory OR disk space, both memory and storage space were extremely precious. For software like that technological limitations were the primary cause of the Y2K problem.

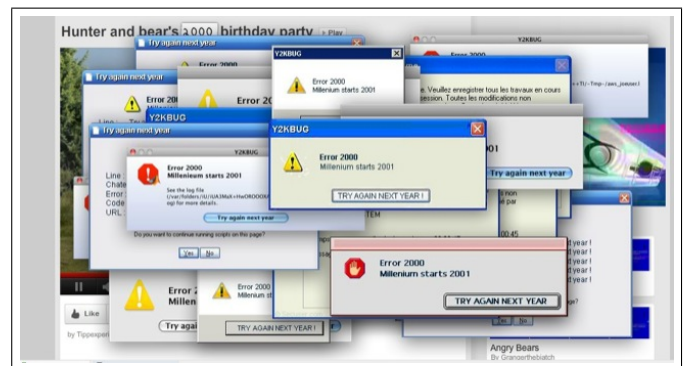


Fig. 8. y2k bug screens

Now for anyone who was writing brand new code in the 90’s and not taking into account the upcoming Y2K issue – that would be poor design. The 90s was perhaps the decade with the most drastic and furious pace of software development and advancement. Companies like Microsoft and Apple, programming languages like Java and JavaScript, all saw the light of the day during the early and mid-90s. Here, the situation begs the question – were certain practices and coding logic simply carried over from the previous generation to

the current blindly without re-evaluating best practices?

For instance,

- 1) Microsoft Excel had the Y2K problem: Excel incorrectly regarded the year 1900 as a leap year for compatibility with Lotus 1-2-3[3]. Further, years like 2100, 2200 etc., were also assumed to be leap years. The problem was consequently fixed, but since the origin of Excel's time-stamp is set to 0 January 1900 in previous versions, 1900 is still regarded as a leap year to preserve backward compatibility[10].
- 2) The Standard Library Function of C (a programming language) extracts the time-stamp of a year by subtracting 1900 from that year. Several other programming languages like Perl and Java (widely used in web development) used function from C, wrongly regarded this value as the last 2 digits of a year. Dynamic Web Pages therefore wrongly displayed 1 January 2000 as *1/1/19100*, *1/1/100*, or other variants, depending on the display format.
- 3) JavaScript was also modified following up on concerns regarding the emerging Y2K problem. Time-stamp for years changed and representation kept shuttling between two and four digit representation forcing programmers to redo existing and functioning code to make sure web pages worked for all versions[1][2].

Here, it begs the question: does it not seem plausible that programming teams were too hasty to jump to development without questioning or validating practices for yester years in relation to the approaching new century? *Requirements Validation* as well as *Requirements Analysis* both phases could be seen as loosely executed during the software engineering processes of the 90s. It is unlikely that programmers were not aware of the emerging Y2K problem. Then why couldn't new software being made in the 90s not take into account the said problem?

III. THE ARIANE 5 EXPLOSION

A. Background to The Ariane 5 and the Explosion

The Ariane 5 is part of The Ariane rocket family, a group of heavy weight launch vehicles used to deliver payloads into Geostationary Orbit or Lower Earth Orbit. These rockets are constructed under the supervision of the Centre National d'Etudes Spatiales (CNES) and the European Space Agency (ESA). On the maiden flight of The Ariane 5 as part of the *Cluster* program, on June 4, 1996, four ESA spacecrafts were launched but the rocket exploded within 40 seconds of its launch before even making it to orbit. The explosion was caused by a defect in The Ariane 5 software. There was inadequate protection from integer overflow and due to unanticipated large horizontal velocities being achieved, the numbers that flowed into the software were much larger than programming accounted for. The high air pressure caused the rocket to disintegrate gradually before triggering the self-destruction protocol leading to the explosion. The Ariane 5 rocket explosion has become one of the most infamous and costly cases of software bugs in history. An estimated 370 million US dollars was lost in the failure of this project[11].

B. A detailed look at the Causes behind the Explosion



Fig. 9. The Ariane 5 rocket launch followed by explosion[5]

The Ariane 5 used the same *inertial navigation system* as The Ariane 4. The Ariane 5's flight trajectory was significantly different from its precursors. The Ariane 5's significantly higher horizontal acceleration compared to The Ariane 4 caused failure of both the primary and backup software platforms as the flight data transmitted was misconstrued as specious position and velocity data.

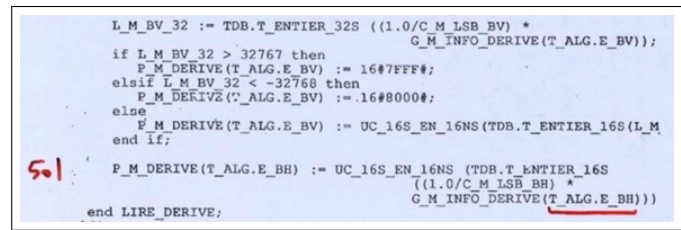
Following the explosion an Inquiry Board was setup comprising engineers from The Ariane 5 project teams as well experts from the industry, in order to investigate the failure. Extensive simulations taking into account The Ariane 5's real-time trajectory and velocities reproduced the chain of events that lead to the explosion[16].

A software bug was indeed identified as the root cause of the failure, however it was unanimously agreed upon that it was made possible due to system design failures, requirements negligence, necessary validation having been overlooked and management issues:

- 1) An internal software exception occurred i.e. a data conversion error during the conversion of a 64-bit floating point value to a 16-bit signed integer value. The floating point value that was converted was too big to be represented by a 16-bit signed integer. This resulted in an Operand Error. The Ada instructions for data conversions in this software did not provide any protection against an Operand Error, although conversions of other similar variables in the similar modules in the code were protected.

The error was caused in a component of the rocket's software that was responsible for adjusting the alignment of The Ariane 5 and it did so only until lift-off after which the component becomes inactive. The extremely large value generated was that of a variable called HB (Horizontal Bias) that stores the horizontal acceleration sensed by the rocket's launch pad. This value aids the aforementioned module to make appropriate adjustments to the rocket's horizontal alignment. The value generated for HB turned out to be much higher than expected because The Ariane 5's initial trajectory differed significantly from The Ariane 4's and the horizontal velocities achieved by The Ariane 5 were also much higher than that of its predecessor.

- 2) An unanticipated change of altitude occurred as a result and under heavy aerodynamic pressure, the rocket disintegrated at H0+39 seconds (H0 being the start in time for navigational systems).



```

L_M_BV_32 := TDB.T_ENTIER_32S ((1.0/C_M_LSB_BV) *
                                G_M_INFO_DERIVE(T_ALG.E_BV));
if L_M_BV_32 > 32767 then
  P_M_DERIVE(T_ALG.E_BV) := 16#7FFF#;
elsif L_M_BV_32 < -32768 then
  P_M_DERIVE(T_ALG.E_BV) := 16#8000#;
else
  P_M_DERIVE(T_ALG.E_BV) := UC_16S_EN_16NS(TDB.T_ENTIER_16S(L_M
end if;

51 P_M_DERIVE(T_ALG.E_BH) := UC_16S_EN_16NS (TDB.T_ENTIER_16S
      ((1.0/C_M_LSB_BH) *
      G_M_INFO_DERIVE(T_ALG.E_BH)))
end LIRE_DERIVE;

```

Fig. 10. ADA source code excerpt that led to the rapid unscheduled disassembly of The Ariane 5, Flight 501[9]

- 3) Because of defective calculations, the software caused the main engine as well as boosters to make a massive altitude correction which in fact was not needed.
- 4) The Ariane 4's navigational system was more or less retained in The Ariane 5. Like its predecessor, The Ariane 5 had a realignment component that served to make horizontal alignment adjustments and altitude corrections. Upon investigation it was found this realignment component served no actual purpose on The Ariane 5 and was retained simply for backward compatibility purposes. The realignment functions already resided within other modules in The Ariane 5's software.
- 5) During the design phase of the software navigational system for The Ariane 4 and The Ariane 5, it was decided that it was not critical to protect the system against a plausible failure scenario caused by an excessive value of the variable holding the horizontal velocity – a protection nonetheless provided for several other variables in the software. While making this decision, it wasn't analyzed thoroughly as to what possible values this variable might have to hold when the software operates post lift-off.
- 6) The specifications of the inertial navigation system and the tests performed at the equipment level were not inclusive of the trajectory data of The Ariane 5. An inertial navigation system (INS) is a computer based navigation system that aids in automated control of flight navigation through: motion and rotational sensors to continuously calculate the position, orientation, velocity etc. of a moving object without the need for

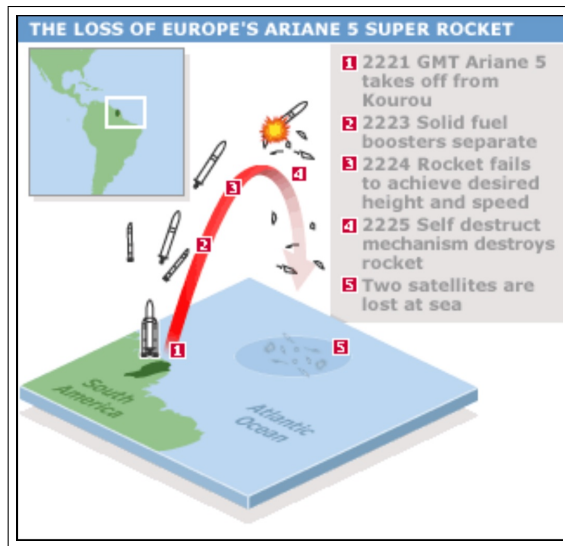


Fig. 11. The trajectory of The Ariane 5 launch and subsequent crash[4]

external references. Subsequently, various realignment functions were not tested under simulated The Ariane 5 flight conditions, and the design error was not discovered.

C. What went wrong for The Ariane 5?

One thing that can be understood from the Inquiry Board's report[16] on The Ariane 5's crash, is that, the management and project teams went ahead of themselves and were too confident from The Ariane 4's success to chart out proper specifications, design adequate testing and perform enough validations and pre-flight simulations to look out for any potential shortsightedness. No pre-flight tests were performed on the inertial navigational system under simulated The Ariane 5 flight conditions so the bug was not discovered before launch. During the Board's investigation, a simulated The Ariane 5 flight was conducted on another inertial platform. It failed in exactly the same way as the actual flight units.

One might argue that in space missions of the frequency and scale as of the Cluster space program, time and resource-conservation is of essence and a few assumptions have to be made on the basis of previous success. Nevertheless, to completely forego flight simulation for a rocket launch when there have been a few major changes in the navigational system such as higher horizontal velocities etc., was certainly not a wise decision.



Fig. 12. Diagram of The Ariane 5 with the four Cluster satellites[11]

- When physical requirements change from one project to the next, the software requirements need to be re-oriented for the same reason. The significantly higher horizontal velocities of The Ariane 5 than The Ariane 4 is something that should not have been overlooked during the specification and validation phase of the software engineering for The Ariane 5.
- Purpose of the review process, which involves

all major partners in a project, is to validate design decisions and to obtain product-launch qualification. Quoting the report of the Inquiry Board itself – “Although the source of the Operand Error has been identified, this in itself did not cause the mission to fail. The specification of the exception-handling mechanism also contributed to the failure. The reason behind this drastic action lies in the culture within the Ariane programme of only addressing random hardware failures.”[16].

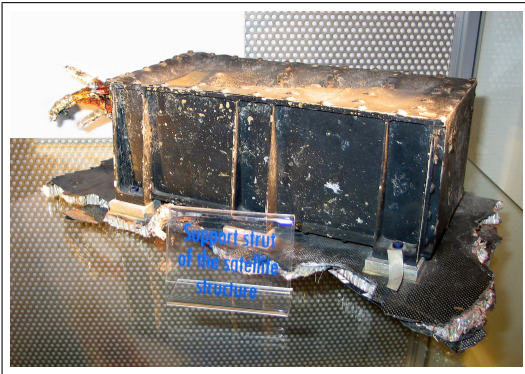


Fig. 13. Recovered support strut of the satellite structure[11]

- Component Reuse in software development is a long established best practice. It helps save time and resources and speeds up the development process. However, a previous system’s success is not always a guarantor of future success and neglecting to make space for adequate requirements analysis under the new set of environment variables can lead to certain blunders. In The Ariane 5’s case there were a few changes in the hardware specification, velocities to be obtained etc. It’s not that component reuse was an unwise decision, but, the reuse of software should be re-evaluated in light of changing hardware or external environment variables[5].

In The Ariane 4, flights using the same type of inertial reference system there had been no such failure because the trajectory during the first 40 seconds of flight is such that the particular variable related to horizontal velocity cannot reach (including an adequate operational margin) a value beyond the limit present in the software. The Ariane 5 had a high initial acceleration and trajectory differ-

ent from The Ariane 4, which led to a build-up of horizontal velocity five times more than that for The Ariane 4. The excessive value of the horizontal velocity of The Ariane 5, within the 40-second time frame, caused the inertial system computers to cease operation[16].

In numerous tests with assertions checking turned on, exception did not occur for The Ariane 5’s software test runs. Hence, in the interest of speed it was decided that the base code would do no operand checking[14]. This however, was not cross checked under real-time flight simulations. While it is difficult to argue whether simulations would have thrown up the integer overflow problem because of the higher horizontal velocities, one cannot completely over rule the importance of having performed flight simulations for The Ariane 5.

IV. CONCLUSION

Software Development is no doubt a complicated process. While it is true that it is not always possible to foresee consequences of design and requirements decisions decades from now, software stakeholders nevertheless need to ensure periodic re-evaluation of best practices, weighing the application of best practices in light of changing environment and purpose, thorough testing and validation under real-time scenarios, and thoroughly analyze a programming decision being made to save development resources so that it doesn’t cause future issues. A lot of these often connect back to the requirements and design analysis and validation phases which is why it becomes imperative to take into account the smallest changes and constraints.

REFERENCES

- [1] Javascript reference javascript 1.2, sun microsystems, retrieved june 2009.
- [2] Javascript reference javascript 1.3, sun microsystems, retrieved june 2009.
- [3] Microsoft support (17 december 2015), “Microsoft Knowledge Base article 214326”, retrieved 16 october 2016.
- [4] BBC News Article: “Engine Glitch Brought Rocket Down”.
- [5] NY Times Article by James Gleick. “Sometimes a Bug Is More Than a Nuisance”.
- [6] A. Terry Bahill and Steven J. Henderson. Requirements development, verification, and validation exhibited in famous failures. *Systems and Industrial Engineering, University of Arizona*.
- [7] Alan Greenspan Ex-Chairman of the US Federal Reserve’s statement before the Senate Banking Committee. Fig. 4:: Testimony by alan greenspan. ISBN 978-0-16-057997-4, 25 February 1998.
- [8] Mindi Gile. Preparing for the millennium bug’s bite: Legislators, lawyers and potential litigants race to define duty and limit liability. *UCLA Journal of Law and Technology archives*, Fall, 1999.
- [9] [HTTP://accu.org/index.php/journals/1898](http://accu.org/index.php/journals/1898). Ada source code.
- [10] [HTTP://en.wikipedia.org/wiki/Year_2000_problem](http://en.wikipedia.org/wiki/Year_2000_problem). Year 2000 problem.
- [11] [https://en.wikipedia.org/wiki/Cluster_\(spacecraft\)](https://en.wikipedia.org/wiki/Cluster_(spacecraft)). Ariane 5.
- [12] https://www.reddit.com/r/mildlyinteresting/comments/3z2586/my_birth_certificate_was_one_of_the_minor/. Reddit - picture of incorrect birth certificate.
- [13] Jussi Koskinen. Software maintenance costs. *Jyväskylä: University of Jyväskylä*, 2010.
- [14] Grard Le Lann. An analysis of the ariane 5 flight 501 failure a system engineering perspective. *Proceedings of the 1997 international conference on Engineering of computer-based systems (ECBS’97)*. IEEE Computer Society. pp. 339346. ISBN 0-8186-7889-5.
- [15] Bashar Nuseibeh. Article: “A Bug and A Crash”. May 1996.
- [16] The Chairman of the Board : Prof. J. L. LIONS. “ARIANE 5, Flight 501 Failure, Report by the Inquiry Board”. originally appeared at “<http://www.esrin.esa.it/htdocs/tidc/Press/Press96/ariane5rep.html>”, July, 1996.
- [17] Laszlo Ropolyi. Social and ethical aspects of the y2k problem.
- [18] Stuart A. Umpleby The George Washington University. Before and after studies of the year 2000 computer crisis. *Prepared for the Fourth European Congress on Systems Science Valencia, Spain, September 20-25, 1999*.
- [19] Rob Thomsett. The year 2000 bug: a forgotten lesson. *IEEE Software*, 15(4):91, 1998.
- [20] Hakim Weatherspoon. Y2k: The millennium bug system failures. *Berkeley CS294 Class: Principles of Fault Tolerant Computing, Fall 2000, Homework 1: Anatomy of Failure*.