Volume 2, Issue 8, August 2012 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Case Studies of Most Common and Severe Types of Software System Failure

Sandeep Dalal[1]
Department of Computer Science and Applications,
Maharshi Dayanand University, Rohtak

Dr. Rajender Singh Chhillar[2]
Department of Computer Science and Applications
Maharshi Dayanand University, Rohtak

Abstract: ***Today software system is an integral part of each and every business model, be it core product manufacturing, banking, healthcare, insurance, aviation, hospitality, social networking, shopping, e-commerce, education or any other domain. If any business has to be leveraged and simplified then software has to be integrated with the main stream business of any organization. Designing and development of any software system requires huge capital, a lot of time, intellectuals, domain expertise, tools and infrastructure. Though the software industry has matured quiet a lot in past decade, however percentage of software failure has also increased, which led to the loss of capital, time, good-will, loss of information and in some cases severe failures of critical applications also lead to the loss of lives. Software could fail due to faults injected in various stages of software or product development life cycle starting from project initiation until deployment. This paper describes the case study of most common and severe types of software system failures in Software Industry.***

Keywords used: Software system, software system failure

1. Introduction

Every organization starts a project with intent of deploying it successfully to perform the function specified by the client or as required by the business, however there are reasons that this goal of the organization is not achieved due to some faults which later results in failures. This could happen due to inappropriate project initiation, planning, monitoring and control, execution or deployment of software system. In bigger projects each phase of the product is considered to be a project, for example 'requirement analysis, elicitation and validation' could be considered a project which would give feed to later stages of product development. So this is not a wrong statement to say that software failure could happen at any stage of software product development **[1][3].** Software failure term is generally used when the software doesn't perform its intended function or crashes after deployment. This paper intends to study the most recent case studies pertaining to most common and severe software failures. Later in this paper we would analyze and conclude the common reason of software failures.

1.1 Software System Failure

Software system is any software product or application supporting any business. A software system could be defined as a system of intercommunicating components based on software forming part of a computer system (a combination of hardware and software). It "consists of a number of separate programs, configuration files, which are used to set up these programs, system documentation, which describes the structure of the system, and user documentation, which explains how to use the system". According to Laprie et al.[12], ``A system failure occurs when the delivered service no longer complies with the specifications, the latter being an agreed description of the system's expected function and/or service". This definition applies to both hardware and software system failures.

1.2. Focus Area of Study

A recent survey **[6,7]** of 800 IT managers says that 62% of total software fails, which is true. 49% software suffered budget overruns, 47% had higher than expected maintenance costs and 41% failed to deliver the expected business value and ROI. Few software while designing never thought of considering the requirements which cause threats and failures later in the stage at the time of utilizing the product for example- information security, hacking, virus threats, scaling up to the level of

usage, maintainability and performance. Software projects fail for various reasons from all the domains and technologies. So this paper would consider case studies showing threats, risks and failure of software systems supporting nation security, banking and financial analysis application, aviation, medical and social networking applications which are used globally.

2. COMMON & SEVERE FAILURES

Those failures are considered to be most severe which impacts the lives of people, huge loss of capital, The severity or impact of fault is determined on basis of various parameters like

a) Number of users of the application
b) Involvement of monetary transactions.
c) Type of use of application like- Home use, National security or defense, space, missile and satellite, aviation related app etc
d) Could the application impact the lives of people if fails.

Severity of failure could be amplified if any of the parameters mentioned above are touched by application. There could be other daily use application which could lead to discomfort if they don't function appropriately. Such applications could be any online shopping app, ticket booking app, emails, chat server, gaming and entertainment app and social networking sites. Such applications experience most common and frequent failures. Most frequent failures include non-functional issues with the application which adds uneasiness and embarrassment. These are generally home use applications. Such failures could be any of these:

a) Slow response from application server.
b) Pages are not downloading properly
c) Application is not compatible with the browser
d) Performance issues like slow access time, load time, run time.

"Electricity lets us heat our homes, cook our food, and enjoy security and entertainment. It also can kill you if you're not careful"

"Energy Notes" (Flyer sent with San Diego Gas & Electric utility bills.

We trust our lives to technology every day. We trust older, non computer technologies every time we step into an elevator, a car, or a building. As the tools and technologies we use become more complex and more interconnected, the amount of damage that results from an individual disruption or failure increases, and we sometimes pay the costs in dramatic and tragic events. If a person, out for a walk, bumps into another person, neither is likely to be hurt. If both are driving cars at 60 miles per hour, they could be killed. If two jets collide, or one loses an engine, several hundred people could be killed.**[8][10].**
Most new technologies were not very safe when first developed. If the death rate from commercial airline accidents in the U.S. were the same now as it was 50 years ago, 8,000 people would die in plane crashes each year. In some early polio vaccines, the virus was not totally inactivated. The vaccines caused polio in some children. We learn how to make improvements. We discover and solve problems. Scientists and engineers study disasters and learn how to prevent them and how to recover from them.

2.1 Learn From Failures

American Airlines had installed GPWS (the system that warns pilots if they are headed toward a mountain) on the plane that crashed near Cali, Colombia, in 1995; they would have saved many lives. This crash triggered adoption of the GPWS. No commercial U.S. airliner has crashed into a mountain since then. Similarly, a disastrous fire led to the development of hydrants—a way to get water at the scene from the water pipes under the street. Automobile engineers used to design the front of an automobile to be extremely rigid, to protect passengers in a crash. But people died and suffered serious injuries because the car frame transmitted the force of a crash to the people. The engineers learned it was better to build cars with "crumple zones" to absorb the force of impact. Software engineering textbooks use the Cali crash as an example so that future software specialists will not repeat the mistakes in the plane's computer system.
We learn what has happened to the safety record in other technologies? The number of deaths from motor vehicle accident in the U.S. declined from 54,633 in 1970 to roughly 42,600 in 2006 (while population and the number of cars, of course,

increased) .One significant factor is increased education about responsible use (i.e., the campaign against drunk driving). Another is devices that protect people when the system fails (seat belts and airbags). Yet another is systems that help avoid accidents (many of which, like airbags, use microprocessors). Examples of the latter include rear-view cameras that help drivers avoid hitting a child when backing up and "night vision" systems that detect obstacles and project onto the windshield an image or diagram of objects in the car's path. Yet another is electronic stability systems. These systems have sensors that detect a likely rollover, before the driver is aware of the problem, and electronically slow the engine. As use of technology, automation, and computer systems has increased in virtually all work places, the risk of dying in an on-the-job accident dropped from 39 among 100,000 workers (in 1934) to four in 100,000 in 2004.

Risk is not restricted to technology and machines, it is a part of life. We are safer if we know the risks and take reasonable precautions. We are never 100% safe.

3. Case Studies

In this section we have discussed some most common and severe types of software system failure case studies.

Table 1 : List of some most common and severe types of software system failure

	Software	Failure Description	Casualties
1.	ERP project failure in Jordan	It finds sizeable gaps between the assumptions and requirements built into the ERP system design, and the actual realities of the client organisation	Huge loss of capital and unsatisfied clients
2.	Ariane 5	Ariane 5, Europe's newest unmanned rocket, was intentionally destroyed seconds after launch on its maiden flight. Also destroyed was its cargo of four scientific satellites to study how the Earth's magnetic field interacts with solar winds.	10 years hardwork and $100 million loss. Reputation of ESA(Europian Space Agency) deteriorated
3.	Therac-25	Canada's Therac-25 radiation therapy machine malfunctioned and delivered lethal radiation doses to patients	Many people dead, Many people critically injured
4.	STS-126	A software change had inadvertently shifted data in the shuttle's flight software code	"In-flight" software anomalies occurred and several automated functions failed.
5.	Automated airport baggage handling(DIA)	Failure to anticipate the number of carts correctly resulted in delays in picking up bags that would undermine the system's performance goal.	Monthly Maintenance cost exceeded the monthly manual investment

3.1 Case Study

This case study focuses on the ERP project failure in Jordan which is a developing nation. The design—reality gap model applied to a case study of partial ERP failure in a Jordanian manufacturing firm. The model analyses the situation both before and during ERP implementation[11]. It finds sizeable gaps between the assumptions and requirements built into the ERP system design, and the actual realities of the client organisation. It is these gaps and the failure to close them during implementation that underlies project failure.ERP systems are failing in developing countries. ERP (Enterprise resource planning) system integrates financial systems, HR, logistics, data systems across the organizations to save money and improve decision making and customer retention. These are increasingly being used by organizations in developing nations.

***Success and Failure Factors*:** There are few outcome elements which decide if ERP implementation is success OR failure for example-System & Information Quality, Use & user satisfaction, Individual impact which relates to the extent to which information produced by system influences or affects the management decisions and Organization impact which measures the effect of the information produced by the system on organizational performance. The model used for deciding success or failure is DeLone & McLean's model. It provides an appropriate framework for data gathering, analysis and presentation in relation to the outcome of an ERP project; and a framework that can be integrated easily with Heeks' three-way outcome categorisation of total failure, partial failure, and success in order to provide a final classification.

Gap analysis is done on the design & real system implemented & the analysis of gap tells that it is a failure.

Loss: It involves huge loss of capital and non satisfaction of the client. This would also not at all solve the purpose of getting ERP software. The ignorant client remains in dark and never gets the business up to the mark with no appropriate decision making.

3.2 Case Study

On the 4th June 1996 at 1233 GMT (UTC) the European Space Agency launched a new rocket, Ariane 5, on its maiden unmanned flight. Ariane exploded after 40 seconds of its lift-off. Although this was an unmanned flight and therefore there were no human casualties, there is no reason to expect that the outcome would have been any different if the flight had been manned. In such an event all onflight crew and passengers would have been killed. Remember as we proceed through this case that this was a project of the very experienced European Space Agency**.[1]** The project cost was $ 7 billion. Part of the payload were four satellites, Cluster, that would engage in a scientific investigation. These satellites had taken many years to develop and cost around $ 100 million. They were irreplaceable.

In a report, James Gleick has said:

"*It took the European Space Agency(ESA) 10 years and $7 billion to produce Ariane 5, a giant rocket capable of hurling a pair of three-ton satellites into orbit with each launch and intended to give Europe overwhelming supremacy in the commercial space business. All it took to explode that rocket less than a minute into its maiden voyage last June, scattering fiery rubble across the mangrove swamps of French Guiana, was a small computer program trying to stuff a 64-bit number into a 16-bit space. One bug, one crash. Of all the careless lines of code recorded in the annals of computer science, this one may stand as the most devastatingly efficient. "*

Purpose of Ariane 5 was to deliver satellite to space. It was a improved version of Ariane4. Control System of Ariane5 was composed of :

- An inertial reference system (SRI)
- An On-Board Computer (OBC)

SRI of Ariane 5 same as one in Ariane 4.Ariane 5 failed due to SRI Software exception caused due to a data conversion. At the time of the failure, the software in the SRI was doing a data conversion from 64-bit floating point to 16-bit integer. The floating point number had a value greater than could be represented by a 16-bit signed integer; this resulted in an overflow software exception. It was actually a reuse error. The SRI horizontal bias module was reused from 10-year-old software, the software from Ariane 4. But this is not the full story: It is a reuse specification error. The truly unacceptable part is the absence of any kind of precise specification associated with a reusable module. The requirement that the horizontal bias should fit in 16 bits was in fact stated in an obscure part of a document. But in the code itself it was nowhere to be found!

The Ariane 5 disaster was a wakeup call for the software engineering community. Proper actions should be taken to ensure such a failure does not occur again.

3.3. Case Study

Canadian Cancer Therapy Machine (Therac-25, 1986) Designed by Atomic Energy of Canada, Ltd. (AECL): Therac-25 was a software controlled radiation therapy machine used to treat people with cancer. Between 1985 and 1987 Therac-25 machines in four medical centers gave massive overdoses of radiation to six patients. An extensive investigation and report revealed that is some instances operators repeated overdoses because machine display indicated no dose given. Some patients received between 13,000 - 25,000 rads when 100-200 needed. The result of the excessive radiation exposure resulted in severe injuries and three patients' deaths[5].

Causes of the errors were attributed to lapses in good safety design. Specific examples are cite failure to use safety precautions present in earlier versions, insufficient testing, and that one key resumption was possible despite an error message. The investigation also found calculation errors. For example, the set-up test used a one byte flag variable whose bit value was incremented on each run. When the routine called for the 256^{th} time, there was a flag overflow and huge electron beam was erroneously turned on.

An extensive investigation showed that although some latent error could be traced back for several years, there was an inadequate system of reporting and investigating accidents that made it hard to determine the root cause. The final investigations report indicates that during real-time operation the software recorded only certain parts of operator

input/editing. In addition, the radiation machine required careful reconstruction by a physicist at one of the cancer centres in order to determine what went wrong.

3.4. Case Study

A few minutes after the Shuttle Endeavour reached orbit for STS-126 on November 14, 2008, mission control noticed that the shuttle did not automatically transfer two communications processes from launch to orbit configuration. Primary communications continued to use S-band frequencies after they should have transferred to the more powerful Ku-band.

The link between the shuttle and its payload—the Payload Signal Processor (PSP)—remained configured for a radio link rather than switching automatically to the hardwired umbilical connection.

Fortunately, mission control was able to manually command both the S-band/Ku-band switch and the PSP port shift. While mission control was not able to re-instate automatic transfers during flight, they continued to monitor communications and manually operated necessary transfers for the remainder of the mission. STS-126 completed its mission successfully and returned to earth without further software problems.

While the software problems did not endanger the mission, they caught management's attention because "in-flight" software anomalies on the shuttle are rare. Software goes through rigorous reviews during development and testing to prevent this sort of problem, and most software anomalies are detected and fixed long before the shuttle leaves the ground.

Investigation found that a software change had inadvertently shifted data in the shuttle's flight software code. Because of this build defect, the software did not send configuration commands to the shuttle's Ground Command Interface Logic, and several automated functions failed.

3.5. Case Study

What was to be the world's largest automated airport baggage handling system, became a classic story in how technology projects can go wrong. Faced with the need for greater airport capacity, the city of Denver elected to construct a new state of the art airport that would cement Denver's position as an air transportation hub. Covering a land area of 140 Km2, the airport was to be the largest in the United States and have the capacity to handle more than 50m passengers annually **[9].**The airport's baggage handling system was a critical component in the plan. By automating baggage handling, aircraft turnaround time was to be reduced to as little as 30 minutes. Faster turnaround meant more efficient operations and was a cornerstone of the airports competitive advantage. Despite the good intentions the plan rapidly dissolved as underestimation of the project's complexity resulted in snowballing problems and public humiliation for everyone involved. Thanks mainly to problems with the baggage system, the airport's opening was delayed by a full 16 months. Expenditure to maintain the empty airport and interest charges on construction loans cost the city of Denver $1.1M per day throughout the delay.

The embarrassing missteps along the way included an impromptu demonstration of the system to the media which illustrated how the system crushed bags, disgorged content and how two carts moving at high speed reacted when they crashed into each other **[4].** When opening day finally arrived, the system was just a shadow of the original plan. Rather than automating all 3 concourses into one integrated system, the system was used in a single concourse, by a single airline and only for outbound flights. All other baggage handling was performed using simple conveyor belts plus a manual tug and trolley system that was hurriedly built when it became clear that the automated system would never achieve its goals.

Although the remnants of the system soldiered on for 10 years, the system never worked well and in August 2005, United Airlines announced that they would abandon the system completely. The $1 million per month maintenance costs exceeded the monthly cost of a manual tug and trolley system.

System at a glance:

1. 88 airport gates in 3 concourses
2. 17 miles of track and 5 miles of conveyor belts
3. 3,100 standard carts + 450 oversized carts
4. 14 million feet of wiring
5. Network of more than 100 PC's to control flow of carts
6. 5,000 electric motors
7. 2,700 photo cells, 400 radio receivers and 59 laser arrays.

As with all failures the problems can be viewed from a number of levels. In its simplest form, the Denver International Airport (DIA) project failed because those making key decision underestimated the complexity involved. As planned, the system was the most complex baggage system ever attempted.Ten times larger than any other automated system, the increased size resulted in an exponential growth in complexity. At the heart of the complexity lay an issue know as "line balancing". To optimize system performance, empty carts had to be distributed around the airport ready to pick up new bags.

With more than 100 pickup points (check in rows and arrival gates) each pickup needed to be fed with enough empty carts to meet its needs. The algorithms necessary to anticipate where empty carts should wait for new bags represented a nightmare in the mathematic modelling of queue behaviours. Failure to anticipate the number of carts correctly would result in delays in picking up bags that would undermine the system's performance goals.

Failure to recognise the complexity and the risk involved contributed to the project being initiated too late. The process of requesting bids for the design and construction of the system was not initiated until summer of 1991. Based on the original project schedule, this left a little over two years for the contracts to be signed and for the system to be designed, built, tested and commissioned. The closest analogous projects were the San Francisco system and one installed in Munich. Although much smaller and simpler, those systems took two years to implement. Given the quantum leap in terms of size and complexity, completing the Denver system in two years was an impossible task.

The underestimation of complexity led to a corresponding underestimation of the effort involved. That underestimation meant that without realising it, the Project Management team had allowed the baggage system to become the airport's critical path. In order to meet the airport's planned opening date, the project needed to be completed in just two years. This clearly was insufficient time and that misjudgement resulted in the project being exposed to massive levels of schedule pressure. Many of the project's subsequent problems were likely a result of (or exacerbated by) shortcuts the team took and the mistakes they made as they tried to meet an impossible schedule.

4. COMMON CAUSES OF FAILURES

- Lack of clear, well-thought-out goals and specifications
- Poor management and poor communication among customers, designers, programmers[13]
- Incorrect steps to reproduce and improper fault assignment[13]
- Institutional or opinionated pressures that encourage unrealistically low bids, unrealistically low budget requests, and underestimates of time requirements
- Use of very new technology, with unknown reliability and problems, perhaps for which software developers have insufficient experience and expertise
- Refusal to recognize or admit that a project is in trouble.

There is a common misconception that increasing reliability will increase safety. Many software-related accidents have occurred despite the software being compliant with the requirements specification. Semantic mismatch is characterized by errors that can be traced to errors in the requirements – what the computer should do is not necessarily consistent with safety and reliability.

5. CONCLUSION

The analysis of case studies pertaining to common and severe failures depicts that a software failure at any stage could lead to the loss of lives, financial losses, wastage of time, effort and other intangible losses like discomfort, stress, good will, reputation, confidence, peace etc. In current information age the application of software has penetrated in each and every industry unlike traditional approach where software was altogether a separate entity. As software has become integral part of every product and process so there is a need to make a full proof system so that the software failures could be avoided. There is further requirement of root cause analysis of these software failures to understand the problematic area and suggest the areas of improvement in the current process as several corrective & preventive actions needs to be taken while developing products and software systems.

6. REFERENCES

[1] . Gerard Le Lann, " Analysis of the Ariane 5 Flight 501 Failure - A System Engineering Perspective" Proceedings of IEEE Workshop on *Engineering of Computer-Based Systems (ECBS '97),pp 339-346*

[2] . Michael Fagan, "Advances in Software Inspections", *IEEE Transactions on Soft-ware Engineering, Vol 12, No 7, July, 1986*

[3] . J. Gray and D. Siewiorek. High-Availability Computer Systems. *IEEE Computer, pages 39-48, Sept. 1991.*

[4] . Case Study – Denver International Airport Baggage Handling System – An illustration of ineffectual decision making., *Calleam Consulting Ltd – Why Technology Projects Fail,2008*

[5] . Delores R. Wallace and D. Richard Kuhn, "Failure Modes in Medical Device Software: An Analysis of 15 years of Recall Data", *International Journal of Reliability, Quality and Safety Engineering, Vol. 8, No. 4, 2001*

[6] . Andreas Zeller and Ralf Hildebrandt, "Simplifying and Isolating Failure–Inducing Input", *IEEE Transactions on Software Engineering, Vol 28, No 2, February 2002*

[7] . Shull, et al, 2002. "What We Have Learned About Fighting Defects," *Proceed-ings, Metrics 2002*. IEEE; pp. 249-258.

[8] . Jim Shore, "Fail Fast", *IEEE Software, September/October 2004, http://martinfowler.com/ieeeSoftware/failFast.pdf*

[9] . Dr. R. de Neufville , The Baggage System at Denver: Prospects and Lessons — *Journal of Air Transport Management, Vol. 1, No. 4, Dec., pp. 229-236, 1994*

[10] . Kurt R. Linberg, Software developer's perceptions about software project failure: a case study ,*The Journal of Systems and Software 49 (1999) pp177-192*

[11] . *Ala'a Hawari & Richard Heeks "Explaining ERP Failure in Developing Countries: A Jordanian Case Study" Manchester Centre for Development Informatics Working Paper 45,2010.*

[12] . J.C.Laprie(Ed.). *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Wein, New York, 1992.

[13]. SandeepDalal & Rajender Chhillar"*Role of fault Reporting in Existing Software Industry*"CiiT International Journal of Software Engineering ,Vol 4, No 7,July 2012.

