Analysis on the ValuJet tragedy and its relevance to Software Engineering:

On May 11, 1996, the ValuJet Airlines McDonnell Douglas DC-9 operating the Flight 592 route crashed into the Everglades about 10 minutes after taking off from Miami as a result of a fire in the cargo compartment caused by improperly stored cargo. Unfortunately, all 110 people on board died. The article tells the story of this tragedy and helps us understand in detail the events that ended up causing this accident.

According to the article, there are three kinds of airplane accidents. The most common one is called "procedural". These are those old-fashioned accidents that resulted from single obvious mistakes that can immediately be understood in simple terms and that have simple solutions (ex: avoid flying on a thunderstorm). The second kind of accident could be called "engineered". Consists of surprising materials failures that should have been predicted by designers or discovered by test pilots but were not. These problems result in tangible solutions.

However, the ValuJet accident is different from the two previous mentioned kind of accidents. It represents the third and more elusive kind of disaster, what they call a "system accident". These accidents are science's illegitimate children, bastards born of the confusion that lies within the complex organizations with which we manage our dangerous technologies. In the case of ValuJet, the study of system accidents presents us with the possibility that we have come to depend on flight, that unless we are willing to end our affordable airline system as we know it, we cannot stop the occasional sacrifice. The people involved this airline business do not consciously trade safety for money or convenience, but they inevitably make a lot of bad little choices. They get away with those choices because, even though there might be a possibility of

something going wrong, in practice it goes right most of the times. But then one day a few of the bad little choices come together, and circumstances take an airplane down.

The article finalizes summarizing that there really is not a single person that can take the blame for the tragedy. Instead, the article suggests that the accident was actually arguable unavoidable at that point given that there were many elements involved. These elements are linked in multiple and often unpredictable ways. The article argues that the failure of one part may coincide with he failure of an entirely different part, and this unforeseeable combination may cause the failure of other parts, leading to an accident like the one described.

Even though the accident described on the article represents an accident that is more hardware and systems focused, there are many ways that software can contribute to a system accident. Take the Soviet Gas Explosion as an example. The Soviet pipeline had a level of complexity that would require advanced automated control software. The CIA was tipped off to the Soviet intentions to steal the control system's plans. Working with the Canadian firm that designed the pipeline control software, the CIA had the designers deliberately create flaws in the programming so that the Soviets would receive a compromised program. It is claimed that in June 1982, flaws in the stolen software led to a massive explosion along part of the pipeline, causing the largest non-nuclear explosion in the planet's history. Clearly, this was not a single person's fault, and even though there might have been someone to blame, the reality is that this person was only one of the factors that contributed to this highly complex series of events that lead to such an accident.

It is obvious to note that as years go by, the importance of software into our daily lives and for big companies increases exponentially. As software importance grows, so does its complexity, and new paradigms of development arise to help guide the process in a systematic way. For example, test driven development is a new age evolutionary approach to development which emphasizes test-first development. This is considered good practice and should help develop minimize risk when implementing software that is very complex. Say that you have a company and you introduce a complex IT system that interacts with software from another part of your company (or another company) in order to automatically pay your employees. Since you don't have control over the second part of the software, if they make changes to it, this might interact with your software incorrectly leading to undesired events, for example overpaying or underpaying an employee. Because of this your company's IT system should be coded with a set of tests, which during development helps make sure everything that the engineers are working for aligns with the desired plan. Also, if in the future some part of software that interacts with your software makes the tests fail, you will catch the error before it is into production.

Going back to the lessons learnt from the ValuJet accident, one important thing to learn is the fact that complexity produce errors and errors produce accidents. Hence, as developers we should strive to break up this complexity whenever possible and transform it into less complex and smaller parts that, if thought individually are easier to understand and solve. One interesting idea comes from the Agile project management methodology, under which teams are encouraged to work on sprints. Under this idea, instead of working and thinking on the end product as a whole,

developers are able to set their minds on more realistic goals that can be finished in a small period of time. This allows developers to only focus one small and less complex tasks at a time and make sure that they are implemented correctly (including testing).

The lessons of ValuJet also provide ethical dilemmas. Engineers have a way of thinking that focuses on making a product efficient and cost effective. With this way of thinking sometimes engineers don't fully realize that in order for this to happen, they might be trading something that is more important, like safety, environmental issues and political or ethical impacts among others. In the ValuJet example, the article mentions how flight engineers who worked on the plane's emergency oxygen supply focused so much on reducing the cost of the product that it led to be an unsafe product that produced the massive accident. Such engineers should have considered these ethical issues when making this decision. Likewise, as software developers, our products might negatively impact our users, even if we are trying to do the right thing. Take the Morris warm as an example. The Morris warm is known to be a program that was developed by a Robert Morris, a Cornell University student that wound up spreading wildly and crashing thousands of computers in 1988 because of a bug on its code. The student meant no harm and thought it was a simple experiment, his program produced a huge negative impact and he was convicted of criminal hacking offense. Just like ValuJet should not have grown so fast, or like Robert Morris should have not released his program without further testing, nor should we rush into development whenever we think we have a great idea for the product. Instead we should evaluate the consequences and thoroughly create a design prior to start developing that considers the ethical implications of our work.

What design methods could help prevent such disasters?

Design test driven development, collaboration accross teams, agile, crash issues

Ethical dilemas and engineering faults that might transfer accross disiplines