Andrey Varakin

Professor B.J. Johnson

CMSI 401

10 September 2020

Implications of Interactive Complexity

It is May, 1996, and a fisherman named Wanton Little has just made a call to the police department, reporting the crash of an airliner. Later identified as a twin-engine DC-9, the airliner is found to belong to ValuJet, a young company specialized in cutting corners to be able to offer affordable flights to its customers. A plan to investigate the wreckage begins at once, however, as this will take time, all that can be initially inferred is what was reported by the pilot to the radar controller at the Miami airport, minutes before the crash.

The pilot had called in, requesting to turn around and land back at the airport immediately due to an emergency. While the request was granted, moments later, the pilot began to report that the cabin had been filled with thick smoke due to a fire somewhere else on the plane. Shortly after, communication had been completely cut off, and the plane began barrelling down before evening out at 1000 feet, too low to safely make it back to Miami airport. Minutes later, flight 592 and its passengers and crew had completely gone off the radar.

Within the next few days, the investigation, headed by the National Transport Safety Board, unearthed several chemical oxygen generators as well as one of the black boxes used to record flight data in the event of an accident. Initially chasing an electrical fire as the cause of the crash, the NTSB inferred from these two key pieces of information that the fire had in fact been caused by a number of easily combustible oxygen generators mistakenly stowed in the airliner's cargo hold. The question quickly shifted from *what* caused the crash, to *who* was responsible. At

a surface level, the blame was of course assigned to ValuJet and the contractors it had hired, but upon further inspection, another, more invisible, suspect arose.

The air travel industry consists of many complex systems. From the initial design and engineering of an airliner, to its preparation before takeoff, there are hundreds of different processes that go into a successful flight. Oftentimes, with such complex systems, arises something which Charles Perrow, a professor of sociology at Yale, calls "interactive complexity."[1] That is to say that one kind of complex system can overlap or rely on another completely different kind of complex system, and these interactions can often be unexpected and lead to unforeseen failures. In the case of ValuJet 592, the complex system that was the oxygen generators was protected by an organizational system of checks and maintenance, carried out by technicians. This organizational structure of maintenance checks failed, however, as it was often lengthy and redundant, leading the technicians to adopt a laissez faire attitude when it came to their work. Had they performed each check properly, they would still not have had the materials needed to secure the generators, as the shipment of safety caps had not arrived that morning. Thus, even though there had been multiple complex systems in place to ensure the safety of the airliner, unexpected cracks in one system could lead the whole structure to fall down. This kind of issue naturally arises in the real world where the environment is often unpredictable, but can it take form in the vacuum-like world of software development?

In a software environment, where there can often be hundreds of moving parts under the hood, these kinds of system accidents *do* arise. One form of such system complexity comes in the number of tools which are now available for software developers. From a multitude of databases, to API's, to languages themselves, software developers rely on a toolkit created by

---

[1] Charles Perrow, *Normal Accidents: Living with High-Risk Technologies: with a New Afterword and a Postscript on the Y2K Problem* (Princeton, NJ: Princeton University Press, 1999).

wildly different people for wildly different purposes. When we rely on so many different resources, we are essentially subjecting ourselves to increased interactive complexity. Instead of adopting a few tight systems and building on them from there, we now open ourselves to a market saturated with different tools that can be pieced together in order to get the job done. It can be said that this, however, is inevitable, and the wide access we have to such tools *does* allow us to develop software in a much more streamlined fashion, yet while it can be more efficient, it often leads us to compromise flexibility and accuracy. Perhaps it is important for software developers to limit the tools they use on a project to a select few, in order to avoid opening themselves and their product to such kinds of interactive complexity, which we have seen comes with the potential for unexpected system failures. Aside from using too many different development tools, system failures can, and almost always do, result from not enough product testing.

One cause of limited testing perhaps stems not directly from software development, but from company management and resource distribution. When a company grows large enough to begin to outsource some of the software work, we begin to see another layer of interactive complexity. Now there are teams from completely different backgrounds, contributing *pieces* of a project rather than building it all from the ground up. Multiple teams require a system in place to be able to communicate effectively. Without a solid system, splitting up work can often result in confusion, bugs that go undetected, and software that is generally more disconnected than something that would come out of one organization. This, however, is yet again arguably inevitable as a company grows, and the software that is developed can only be as airtight as the company's standards for communication and chemistry between the teams working on it. Another one of the biggest forms of software failure is a lack of security.

Unlike most hardware, modern software often holds a lot of user information, and there is always the potential threat of someone trying to break into the system. As security evolves, so do cracking techniques and technology. This has led to security developers and hackers playing a game of tug of war, resulting in a system that has become increasingly complex, where major companies now have to rely on a multitude of separate security services. The fact that cybersecurity is such a quickly-growing field also means that companies have to constantly seek out and adopt new security technologies. This can often result in unexpected issues when attempting to integrate the new technologies into their own systems, which can of course have huge repercussions on the company and its user-base if something goes wrong. Again, this is something which seems inevitable in today's environment, but perhaps companies can focus on strengthening the security systems they already have in place, before relying on new ones.

While it is easy to look at the story of the ValuJet 592 crash and assign the entirety of the blame onto ValuJet as a company, in reality the root of the issue is much more complex. The company *did* in fact have systems in place to avoid such an accident. What went wrong, however, was that those systems relied upon a multitude of foreign ones that involved complex engineering, technicians, shipping, etc.. The story is not so different in the world of software development. Developers rely on an ever-increasing variety of tools, many alien to each other, just as bigger companies rely on outsourcing work to multiple teams or services. Perhaps the most important lesson to take away from the ValuJet story is that combining many of these moving parts can, and will, lead to unexpected problems that can have potent real-world consequences. While it is impossible to completely avoid pairing foreign software or services, we should be cautious of their potential interactions. Where we *must* use them, we should focus

on ensuring their success - whether that means rigorous testing, or developing structures to better connect them.

Work Cited:

Charles Perrow, *Normal Accidents: Living with High-Risk Technologies: with a New Afterword and a Postscript on the Y2K Problem* (Princeton, NJ: Princeton University Press, 1999).