Assignment 1

Synopsis

“The Lessons of ValuJet 592” is an article about the analysis of the ValuJet flight 592 plane crash in May 1996 in Florida Everglades that killed all 110 people aboard. The accident first began with the oxygen generators (safety devices that produce oxygen for passengers during emergencies). These generators were removed from the aircraft during maintenance at SabreTech, a repair company in Miami. The mechanics were supposed to put safety caps on the generators before storing them, but this crucial step was forgotten. Without these safety caps, the generators could accidentally start producing oxygen and dangerous heat.

The generators ended up in cardboard boxes marked as "empty" and were loaded onto Flight 592 as cargo. During the flight, one generator started igniting, creating intense heat and fire in the cargo area below the passenger cabin. The fire spread quickly, filling the plane with deadly smoke. This led to the pilots trying to return to Miami, but they lost control and crashed into a swamp. Langewiesche explains that this was not a simple accident caused by one person's mistake. Instead, it was a "system accident", a failure that happened because of how complex organizations work together. Mechanics didn't install safety caps because the caps were not available, supervisors signed paperwork without checking the actual work, shipping clerks didn't understand what they were handling, ValuJet workers accepted dangerous cargo without realizing it, and everyone assumed someone else was handling safety.

The author describes three types of airplane accidents. First are "procedural" accidents caused by obvious mistakes with simple solutions (e.g. flying into bad weather) which can be avoided by avoiding going up into the air during bad weather. Second are "engineered" accidents caused by equipment failures which can be fixed by better design. Third are "system accidents”

which happen because complex organizations have too many moving parts that can fail in unexpected ways (e.g. the ValuJet flight 592 plane crash).

After the crash, officials blamed specific people and companies, wrote new rules, and promised better safety, but Langewiesche argues that these responses miss the real problem. The accident happened because the airline system itself was so complex that some failures were impossible to prevent. Adding more rules and procedures might actually make things worse by creating even more complexity. The investigation showed how each person made decisions that seemed reasonable at the time. The mechanics were working under time pressure and assumed someone else would catch their mistakes. The shipping workers were trying to clean up their area and didn't understand the danger. The cargo handlers trusted that the paperwork was correct. Nobody was trying to cause harm, but it was their small shortcuts that added up into disaster. The ValuJet crash teaches important lessons for software development, where complex computer systems can fail in similar ways. Modern software projects involve many people, teams, and technologies working together, creating the same kind of system complexity that caused the plane crash.

Thoughts and Application

Communication Problems

Today's software systems are like the airline industry (they have many parts that must work together perfectly). A typical web application might use dozens of different software libraries, connect to multiple databases, rely on cloud services, and integrate with other companies' systems. Each piece might work fine by itself, but when they interact, unexpected problems can happen. For example, a small change to a database might cause a website to load slowly, which triggers automatic retry attempts, which overloads a server, which causes user logins to fail. No single programmer intended this chain reaction, but the complex connections between systems made it possible. Just as the oxygen generators seemed harmless until they were put in the wrong place, software components can become dangerous when they interact in unexpected ways.

Accepting Lower Standards

The article describes how people gradually accept shortcuts and rule-breaking because nothing bad happens most of the time. Software engineering is also similar to this. Teams might skip code reviews when they're in a hurry, ignore failing tests, or deploy code without proper testing because they need to meet a deadline. Each compromise seems justified at the time. A programmer might think, "This small change is safe, so I don't need a code review," or "The test is probably wrong, so I'll just ignore it for now." But these shortcuts add up over time, creating hidden problems that can cause major failures later. Unlike missing safety caps that you can see, software problems are invisible until something goes wrong.

False Security from Processes

The airline industry created detailed paperwork and inspection procedures to ensure safety, but these processes gave people false confidence while hiding real problems. Software development has the same issue with testing, code reviews, and quality control procedures. Teams might run automated tests that check individual pieces of code but miss problems that only happen when the whole system is running. They might measure "code coverage" to see how much of their code is tested, but high coverage percentages don't guarantee that the tests actually catch important bugs. Like the safety inspections that missed the missing caps, software quality processes can create an illusion of safety while real risks remain hidden.

Shared Responsibility Problems

The ValuJet crash happened because responsibility was spread across many people and organizations. Nobody felt fully responsible for safety because they assumed others were handling different parts of the problem. Software projects have the same challenge. A security problem might involve code written by multiple teams, using libraries created by outside companies, running on servers managed by another team, with databases maintained by a different group. When something goes wrong, it's hard to know who should have prevented it. Like the mechanics who assumed supervisors would catch their mistakes, programmers might assume that security experts will find vulnerabilities, or that operations teams will handle performance problems.

Learning from the ValuJet disaster, software engineers can take several steps to prevent system accidents:

Keep Systems Simple

Instead of building complex systems with many interconnected parts, teams should create simpler designs that are easier to understand and debug. Each new layer of complexity creates more ways for things to go wrong.

Write Clear Documentation

Technical specifications and instructions should be written in plain language and tested by people who weren't involved in creating them. If someone gets confused by the documentation, the writing needs to be improved, not the person's understanding.

Test Whole Systems

Beyond checking individual pieces of code, teams need to test how all the parts work together under realistic conditions. This includes deliberately breaking things to see how the system responds to failures.

Create Safe Communication

Organizations should encourage people to report problems and ask questions without fear of being blamed. If people are afraid to admit mistakes or raise concerns, important safety information stays hidden.

Document Assumptions

Teams should write down what they're assuming about how systems will behave and regularly check whether those assumptions are still true. Like the missing safety caps that everyone assumed someone else installed, critical assumptions should be made visible and verified.

Responsibilities

The ValuJet case shows that professional responsibility becomes complicated in complex systems. Software engineers face similar ethical challenges when business pressures conflict with good engineering practices. When managers want to cut testing time to meet a deadline, or when customers demand features that could compromise security, programmers must decide how to balance competing demands. The complexity of modern software doesn't eliminate individual responsibility. Even when failures involve many people and systems, each person is still responsible for their own work quality and for speaking up about potential problems they notice.

Applying This Lesson Into Our Own Project

Our game is a system of systems. There is a game manager that oversees the rest of the systems (player, inventory, dialogue, art, etc.) and communicates with each individually, which in turn affects the others. While each system may work on its own individually, unexpected problems can occur when they interact. For example, the inventory manager might update an item incorrectly, which causes the UI to display the wrong data, which leads to a game breaking bug when the player tries to use an item. For our project, we will not be taking any shortcuts which can snowball into bigger issues down the line and test continuously.