Sam Gibson

Professor Johnson

CMSI 401

10/7/2020

“Do what I mean, not what I say” -Sam’s Mom 2020

Synopsis:

This article is a story of miscommunication. It tells the tale of a disastrous plane crash and what caused it.From the mishandling of parts weeks before the crash all the way up to the moments before impact it seems like everything that caused the crash was some form of miscommunication. Langewiesche walks through exactly how the accident happened and it all starts with some oxygen generators, a safety measure used to give oxygen to passengers in the cabin if they need it. The generators were stored in a hangar without their safety caps. The mechanics who handled the canisters didn’t have any of the caps on hand and had no idea how dangerous the canisters could be so they thought it would be ok if they ignored the caps. Eventually they were moved to the shipping department because that was where ValuJet property was stored and a shipping clerk was then told to get rid of the boxes, so he did what he usually did and had the canisters shipped to Atlanta. Not knowing much about oxygen canisters the clerk also mistakenly marked them as empty. At the airport to have them shipped the ramp agent loaded the canisters onto the plane even though he should have known they could be dangerous. However the canisters were just marked as “company property” and when he discussed them with the co-pilot the co-pilot made no mention of

them being dangerous. They were then loaded next to tires which likely caused the canisters to ignite and set the plane on fire.

My Thoughts:

The point that Langewiesche makes in this story is that, while all of the mistakes could have been avoided if those involved handled the canisters properly. There is no way for all the people involved to know so much about the air canisters, and as a result there is no way for them to be able to handle them properly. For example the mechanics who initially stored them without their caps would have caused no accident if the canisters were never put on a plane after, and the mechanics had no way of knowing that the canisters would be on a plane. Even if the mechanics did know the canisters would be on a plane they can't be expected to have full knowledge of every part they work with due to the high volume of parts they work with, and as Langewiesche points out how difficult the manual can be to understand. Furthermore the shipping clerk likely assumes that the mechanics stored the canisters properly so he feels just in having them be shipped off. Finally the ramp agent may have concern over putting the canisters on a plane but if the co-pilot raises no concern about them then he may think that there is a reason that he does not understand that makes them safe, especially since he was already told to ship them off.

How it relates to software:

Although software developers don't have to worry about oxygen canisters they certainly have to work in complex systems, and with many different tools that they may not have very much knowledge of. Software development is also a ripe environment for

the miscommunication we saw in the canister example as developer work with code that other developers wrote and maybe misunderstand what it does, or use tests that other developers wrote without understanding what the tests cover. I have seen first hand how something similar to the oxygen canisters could happen at a software company, but instead of oxygen canisters the mishandled and miscommunication was centered around code.

My Example:

At my internship I was assigned to write unit tests for various angular components, however I had never used angular before and this was the first time I was seeing any of these components. I managed to get all my tests to pass and create a merge request to have the tests merged into the master branch however the reviewer who was assigned to review my code was very busy so he just merged it without reading all of it because he “trusted me”. As far as I know those tests are perfectly working, good tests, however it is possible that the tests pass even when they shouldn’t, as I have seen in previous projects (insert traumatic flashbacks to CMSI 488 here) and if that was the case it could spell disaster as a future developer might think their code works even when it does not.

If we were to analogize my unit test example to the canister example we could say that in the analogy my unit tests are the canisters, I am the mechanics that initially stored the canisters, and the reviewer of my code is the shipping clerk. I like the mechanics, was working with a tool I was unfamiliar with (Angular, and the components I needed to test) and as a result there is a chance I made a mistake. My reviewer, like

the shipping clerk just assumed I did everything right and merged the unit tests, just like the clerk thought it was safe to ship off the canister. Finally its possible that one day somebody will make a change to a component and my test will pass, even though there is something wrong with the component. The new developer working on the component will see all the tests pass and think they are all clear to merge their changes. Their reviewer might not see anything wrong and also see the tests passing and merge the code, accidentally putting dangerous code into production. Then like the canisters exploding the dangerous code might cause some sort of accident with the developers having no way to predict such an accident.

My solution:

I think the best way to avoid these system accidents is to put a large amount of effort into avoiding serious system accidents such as revealing sensitive user info or endangering the livelihood of your users, and to put little effort into avoiding minor system accidents that wouldn't effect you users lives too negatively, and instead fix those minor accidents after the fact. For example take a social media application like Facebook. It is very important that Facebook does not leak sensitive user info so they should spend a lot of energy to prevent that. However if someone posts a particularly gruesome or inappropriate photo, although undesirable minimal real harm is being done to the users, so Facebook should not work to prevent these photos from being posted, but rather just take them down after the fact. My reasoning behind this solution is that as Langewiesche mentions adding more complexity to the system may just cause even more system errors, so we do not want to clog up our system by preventing minor

system errors, instead we just want to spend effort preventing the accidents that would be devastating if they were to occur and handle the minor accidents as they happen.