# The Plot Against People

In his essay “The Plot Against People” a prominent American journalist and writer Russel Baker suggests that all inanimate objects’ sole purpose is to upset, confuse and infuriate their owners, thus, making them “against people”. According to Baker, all inanimate objects fall into one of three categories: those that break down, those that get lost, and those that don’t work. His classification is based on the way in which objects resist people. He then elaborates by bringing up various examples such as cars, gloves and toy trains. Reading his essay inspired me to come up with my own classification.

Working as a software engineer I often have to deal with bugs. Bugs are mistakes in the application’s code that are causing errors, interrupting the business, leading to customer dissatisfaction and, ultimately, financial loss. This is the reason most companies treat bugs in their products very seriously. However, bugs are unavoidable as developers oftentimes cannot predict all possible ways a user might come up with in order to break their app. Sometimes users even use certain bugs and anomalies in the app’s behavior to their advantage or simply get used to working around them, and, when a bug is finally identified and patched, write letters to the customer support about how upset they are with the recent update.

If a bug goes past both the developer and quality assurance specialist and makes it into the production environment unnoticed, customers will be first to bump into those. They then reach out to customer support reporting the issue. Customer support notes down the bug and reaches out to some sort of manager for impact evaluation. Sometimes a bug can get intentionally disregarded. This happens if a bug is not causing much disruption to the business. Engineers need to work on new features as it brings in more revenue and distracting them with insignificant bugs may delay a feature release.

Impact evaluation is an important step in the process of establishing if a bug is worth fixing. Many questions have to be asked, such as “Are we losing profit?”, “How much frustration is this causing?”, and “How many users are affected by this issue?”. Having established the bug’s severity, a manager then asks engineers to evaluate the bug in terms of human-hours it would require to develop a patch. Knowing both the severity of a bug and the time estimate for developing a patch, a bug can be assigned a priority. At this stage, naturally, many bugs fall into the “Won’t fix” category. Other bugs will get worked on when the team has time.

Coming up with a time estimate can often be complicated. Before any time estimates can be derived, a realistic solution has to be suggested. And for that, one has to establish what’s causing the issue. Identifying the cause is not always straightforward. Some issues are pretty easy to understand. A typo is a typo, and if a button is the wrong color, the developer would know where to look for an error straight away. I categorize such bugs as “Obvious”. They are characterized by being easy to understand, easy to reproduce and easy to fix. Obvious bugs are cheapest to fix and are often worth fixing even if they have low impact.

“Coconuts”, on the other hand, are a different story. It is not hard to spot a coconut growing on a palm tree, but getting your hands on one and actually opening it requires a lot of labour. I reserve this category for bugs that are easy to reproduce and easy to find the cause for, but hard to fix. Consider you’re developing a video-conferencing app. Users report that their friends from abroad who they are scheduling meetings with receive invitations that contain the wrong date and time. It doesn’t take much time to notice that the time in the invitation is the same as the time set by the meeting organizer, despite the invitee living in a different timezone. One might think this is an easy fix, but it is categorized as a “Coconut” for a reason. Upon closer investigation it turns out that your application has been storing all meeting start times in a timezone-agnostic manner this entire time. As you realize it you understand that fixing such an issue (cracking the coconut) would require a lot of time. Not only do you have to update the entire system to save meeting times in a timezone-aware way, but you have to update all the previously sent invitations so that they feature a correct timestamp adjusted to the invitee’s timezone. The irony is, a solid portion of your users have gotten used to making adjustments to the time in the invitation in their heads for it to match the actual time where they are. As soon as you develop and release a fix for the timezone issue, your customer support inbox gets flooded with complaints about the meeting time in the invitation being “wrong” once again.

Another category are “Illusive” bugs. These are rare-occurring and are notorious for being hard to reproduce and understand. These usually require a very specific chain of events to occur at very strategic times complete with numerous conditions being met. Consider a taxi-ordering app. Users from various cities that are next to each other occasionally report that two cars showed up although they only requested one. This is a serious issue, because not only do such customers get charged for ordering both taxis, despite only needing one, which causes major frustration, but the taxi driver is also wasting time when they could be driving someone to the airport and earning money for the company instead. The impact of such an issue was deemed to be high and you as an engineer were asked to find the cause and propose a solution. But how? In order to understand what causes the issue you look at all the users who have reported this same issue. They seem to have nothing in common. All of them use different model phones, make their orders at different times of the day and even live in different towns. You carry out a long and thorough investigation. You test one hypothesis after another and none of them seem to work. Every time you order a taxi with the app only one car shows up. After a week of fruitless investigation you tell your manager that you cannot establish the reason for this issue at that particular moment. A week later you are reading news at the breakfast table. You come across a headline: “Issues at a local power plant cause major disruption in cellular communication”. Below you find a list of affected cities that all get their power from the same power plant. It overlaps with the list of cities that your affected users are from! And then it hit you: there must have been a connectivity issue when they were requesting a car. You test it and, indeed, if the network connection is lost right at the moment a taxi request has been sent, but before an acknowledgement response has been received, the app starts retrying the request practically asking for another car to pick up the customer at the same location. Illusive bugs are the hardest to catch during development, as they can almost feel like something mysterious, but they sure feel the most rewarding. However, they can also be frustrating at times. It’s not uncommon when after a long and tedious research into the issue it turns out that fixing it would be too expensive and the manager would not approve it. Feels like a lot of wasted time.

I remember a time when I had to deal with such an “Illusive” bug myself. A group of customers from Tajikistan reported not being able to log in to their accounts despite having no problems signing up for the service. It took me more than a week to find out why this was happening. Our app uses phone numbers as logins. Phone numbers are notoriously complicated: the same phone number can have a dozen different forms. In their international form they generally consist of a country code, an area code, which is specific to the carrier and the region the number is issued in, and a unique subscriber number which is normally the last 7 digits. In order to convert all phone numbers our users submit to the same international format we performed a process called normalization on them. The code responsible for phone number normalization was written in the early days of the project and has remained unchanged for years, no one could even suspect it was faulty, as it is very uncommon for something to work for years without any issues and then suddenly break. I was looking through all the recent changes trying to find an error, but no luck. And then I noticed something odd: all the affected customers had phone numbers issued by the same carrier in the same region thus having the same area code. And this area code started from zero! As I looked once more at the phone normalization code I noticed that for a brief moment, it was splitting the number in two parts: a country code and a nationally-significant number, which is a combination of an area code and subscriber number. And then it took the nationally-significant number and converted it into an integer, practically treating it like an actual number rather than a series of digits. It would then proceed to merge the two parts back together, but at this point the leading zero of the area code was removed as it was insignificant for a number. And a different number, one digit shorter, came out after normalization. So when users were entering their phone number to log in it would get “normalized” into a different number which could not be found in the database. Oddly enough, the correct version of their phone number was stored in the DB. This turned out to be due to those customers being referred to us by a partner that provided their phone numbers directly. Phone numbers provided by the partners went straight to the database bypassing the normalization completely. When I found this out it only took me a minute to change a single line of code and resolve the issue, but the investigation took 10 full working days of my time. What an adventure.