Knowing wehre to tap: the magical art of problem solving

Koen Hufkens

2022-05-06

1 Knowing where to tap: the magical art of problem solving

To some solving technical problems seems like a dark art. Whereby some people posses a certain sixth sense for finding out how or where things go wrong. Everyone has encountered such "magical" events to the extent that the bill that often follows when asking people for such services has been at the origin of myth.

> A town's electricity generator failed and various engineers were unable to fix it, so an elderly professor was summoned. He examined the generator carefully, then tapped it lightly once with a hammer, and power was instantly restored. He submitted his bill for $1,000.02 and itemized it: "Tapping — $.02. Knowing where to tap — $1,000."

This impression that part of it is somewhat magical or at least rooted in an extensive body of knowledge can be very daunting to students. Although it is true that lived experiences can't be easily substituted, and for sure this helps in quickly solving problems by recognizing certain patterns, it is not the only way to acquire elderly professor skills.

This set of chapters / lectures tries to put a finger on how to best solve technical problems using various systems. It explores how to approach technical problems, in both software and hardware, which will allow you to speed up your work whenever you are stuck or things break on you. Due to the nature of my experience and the questions by students I will first and foremost focus on software rather than hardware (although some comments will be made where I see fit).

It is important to note that what is described in these chapters are approaches that are valid within the technical business sphere only. Problem solving is highly context specific. Approach social and political problems using these techniques will invariably result in failure a slap in the face, or a lost job. People are not car engines, chunks of code or pieces of furniture to be rearranged at will. I will even go further to say that many of the world's problems originate in the application of an engineering problem solving mindset to what is inherently a social and political problem.

Furthermore, the routines as described are meant for non life-threatening situations only. When dealing with extreme danger, e.g. accidents in a machine shop and medical emergencies the routines described in this manual are generally too slow. Emergency problem solving relies mostly on a vast body of readily available knowledge and only in part on iterative deduction. Although being aware of some of these techniques might help you, seek a medical professional before intervening yourself.

With these warnings in place, please enjoy learning some shortcuts to the magical art of problem solving.

2 Knowing where to tap

2.1 Many parts to the problem

Identifying the cause of a problem

Efficiently solving a problem

Avoiding a problem

2.2 Root cause analysis

Faults in any technical system happen. This is a fact of life. Even well engineered hardware systems are prone to wear and tear and therefore can fail. Tracking these faults either reactively (when things break) or proactively (before things fail) can be approached using a root cause analysis.

Root cause analysis tries to pinpoint where a system breaks by using four simple rules:

1. identify the problem
2. establish a timeline until the problem (if applicable)
3. correlate circumstances with the problem (event correlation)
4. draw up or sketch when the fault occurs (visualize the process)

Key in root cause analysis is that when you remove the root cause the problem will not exist anymore. It is important to note that this is different from removing the causal factors which triggers the root cause. However in some cases they are related (see boundary conditions below).

2.2.1 Finding a root - theory

A root cause can easily be found by taking into consideration the rules above and the five “why” approach. Continue to ask why something does not work until you reach a root cause (depending on the system this might take more than five steps).

2.2.1.1 Identifying a problem

Often when things break it produces a set of symptoms, in short things that shouldn’t happen. However, these symptoms aren’t necessarily the root cause and can be linked in various ways to the root cause however observing that things break is the first step towards a solution.

2.2.2 Defining the problem

Although the next steps (below) will bring you closer to the root cause it is always wise to define the true nature of the problem. Throwing your hands up and stating that it “just doesn’t work” is not a helpful step towards a solution. One has to accurately define what does not

work. This will need you to gather data on the processes involved and the timeline it developed on, but also put into words what (process) isn't working.

2.2.2.1 Timelines

Timelines are critical in finding many root causes of software and hardware problems. Timelines give you a before (working) state and and after (failing) state of a system. Logic dictates that something in the mean time has changed which alters the outcome of your process.

Defining when something happens is therefore key in finding many software and hardware issues. Within the context of software development it can not be stressed enough how important version control is within this context. Version control allows you to track the temporal progression of code and revert to a previous working state. Maintenance logs on physical hardware serve more or less the same function.

2.2.2.2 Boundary conditions

Not all faults should be resolved. Some faults can be circumvented in a lawful way, by failing gracefully. However, be sure to investigate the conditions of failure well. When writing code you create a system which might have inherent assumptions which are known to you but not necessarily someone else. This inbalance in information can create situations where a piece of software (or hardware for that matter) is used outside its original scope. This can cause faults which are not

critical, yet failing gracefully (with clear warnings or messages of out of scope use) should prevent any confusion on this part.

Black swan events (improbable high impact but not impossible situations) or silent risk might also trip you up. This is not necessarily out of scope use but rather a combination of a set of parameters which results in failure due to their combined effect. Since the likelihood that all these parameters converge is low such (logical) errors might sneak into systems but can cause faults if used long enough in various circumstances. The latter is particularly

2.2.3 Example 1. The holiday blues

For example, your code breaks after you return from holiday. You've identified the problem (1), and you have a timeline (2) since you know that things worked before you left for your holiday. This sets you up to investigate what changed between you leaving and returning from your holiday.

You can now narrow down the source of your problem to this time window and all actions you (automatically) took between you left and returned. Say, you returned from your holiday and updated your computer. This would suggest that the state of your computer has changed and seems to be critical to your piece of software running correctly.

2.2.4 Example 2. The unexpected

Out of bound example

2.2.5 Example 2. Carnival

Type issues

3 Cheat sheet

3.0.1 Finding a root - practices

- breakpoints
- print outputs
-

3.0.2 Avoiding problems

- setup unit tests
- tests mechanistic models using a wide range of parameters
-

4 References

1977 May 19, The Corpus Christi Times, Welfare reform: Knowing where to ‘tap’ is important by George Will, Quote Page 14A, Column 5, Corpus Christi, Texas. (Newspapers_com)