Harness the Power of Debugging

Kiruthika Ganesan

Kiruthika.ganesan@gmail.com

Abstract

Debugging is a superpower which will elevate anyone in tech to the next level and for a tester the benefits increase multifold. With this superpower in their arsenal any ordinary tester could transform into an extraordinary one. This talk is a reflection on how debugging has changed my life and empowered me to navigate the various challenges in my career. I started debugging as part of analyzing bugs, then used it to understand and read code. Later, I got involved in writing automated checks, which in turn gave me the tools to expand my testing and look at software implementations with a different perspective. Following the positive results, I have tried to employ the same routine in other problem areas. From my experiences, I have found that one could debug a production issue to do root cause analysis, debug a process to identify gaps, debug an idea/problem to understand the scope, debug an initiative to explore the possibilities and many more. The whole journey takes a different turn when done alone, when done with someone or a group, when different tools and techniques are used.

Biography

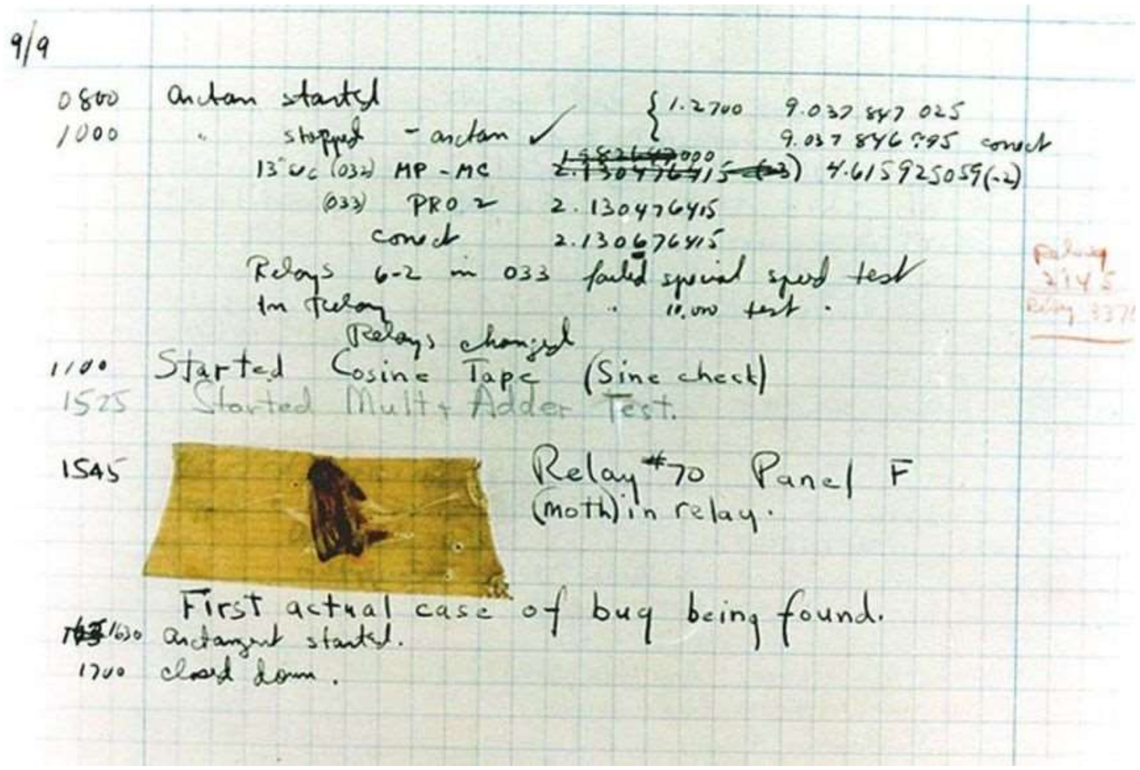
Kika is passionate about testing and approaches testing with a holistic view. She has over 17 years of experience in the IT industry, working as a tester, developer, and a trainer. Kika believes in the power of people and collaborating to build a safe environment where the teams can thrive and work on creating great software. She enjoys teaching and getting involved in community activities like speaking at conferences and delivering workshops. She is also one of the tutors at the Coders Guild and a core member of Synapse QA. Also, a keen advocate of Women in Tech initiatives and a global ambassador. In her spare time, Kika loves spending time with her family and writing short stories.

1. Introduction

What is debugging?

According to Wikipedia: “In computer programming and software development, debugging is the process of finding and resolving bugs (defects or problems that prevent correct operation) within computer programs, software, or systems.”

A little bit on etymology. The terms "bug" and "debugging" are popularly attributed to Admiral Grace Hopper in the 1940s. While she was working on a Mark II computer at Harvard University, her associates discovered a moth stuck in a relay and thereby impeding operation, whereupon she remarked that they were "debugging" the system. However, the term "bug", in the sense of "technical error", dates back at least to 1878 and Thomas Edison who describes the "little faults and difficulties" of mechanical engineering as "Bugs". Below is the first bug report ever written and it was an entry in Grace’s diary.



9/9

0800 Arctan started
1000 " stopped - arctan ✓ {1.2700 9.037 847 025
9.037 846 995 conect
13" UC (032) MP - MC ~~1.982647000~~ ~~2.130476415 (-3)~~ 4.615925059(-2)
(033) PRO 2 2.130476415
conect 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10,000 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 Arctangent started.
1700 closed down.

Relay 2145
Relay 3376

Figure 1 A computer log entry from the Mark II, with a moth taped to the page

However, I would describe debugging as a creative process which uses both sides of the brain. The left hemisphere is more logical & analytical while the right is intuitive and imaginative. Traits like curiosity, problem solving, and synergy come from the right brain while the left makes use of our rational processes of analysis and logic. When both work in harmony, any problem can be solved. So, debugging also means problem solving.

2. Stages of Debugging

When we talk about bugs, we can find them in any phase of the Software Development Life Cycle (SDLC). Bugs don’t just exist on the code level. We can find them in requirements, design, architectural diagrams, UI wireframes, code, documentation and even processes. If we ever have a bug or a problem, we can use these four stages to explore and fix them.

The four stages are;

1. Identify
2. Isolate
3. Fix
4. Review

The initial stage is to Identify the bug/problem and gather information on what we know. It could be from previous experience handling a similar problem, talking to people who know more context on the issue, from the logs, monitoring tools, or bug reports. This phase is primarily to identify what the is bug and understand why it is happening.

Next is to Isolate so it can be reproduced. Where is this bug - frontend, backend, database, API, message queue, etc. Form a hypothesis and experiment with it – test the hypothesis. In the case of an opaque process (something you don't have the code for), follow the same approach and try to isolate the part of the process which is the pain point and try different options.

When we are happy with the result, make the relevant change. Fix the code or fix the problem.

The last stage is to test the changes, check whether the bug is no more and, without fail, run the existing regression suite , so we don't introduce any new issues. It might also be a good idea to assess whether this scenario can be added to the regression suite to ensure the bug doesn't resurface again.

3. My Experiences with Debugging

I want to share my top three experiences with debugging in this paper and highlight how it has benefitted my career.

3.1 Debugging a production bug

Finding out what has happened and trying to reproduce a production bug in a test environment is what got me started on debugging. It is fascinating when you go down this path because you are critically thinking about why it happened, what did we miss?

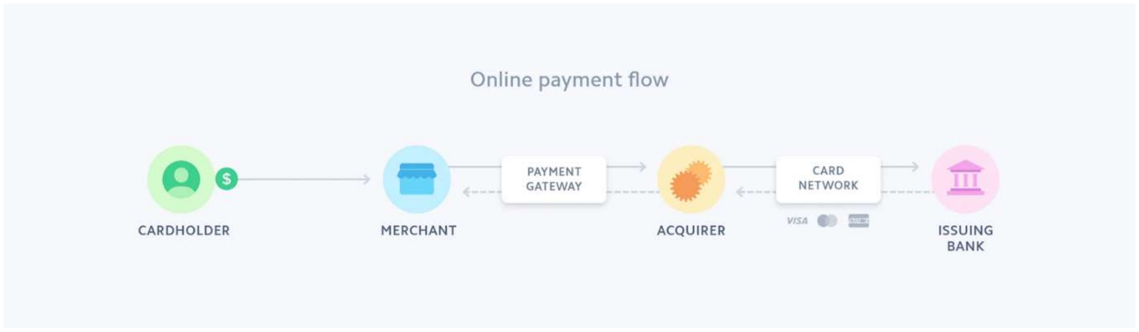


Figure 2 Online payment flow showcasing the different parties

Once upon a time, I was working on a payment gateway team, who was the middleman between the merchant and the banks. As you can see in the diagram, a user would go onto a merchant website like John Lewis and order a sofa. They will pay for the purchase on the John Lewis site. This transaction would go through the payment gateway (our team) to the acquirer who is the merchant's bank and then to the issuing bank which is the card holder's bank. Lots of checks are done and when both banks approve, the payment is taken by the merchant. As the payment gateway team, we capture all the transactions for the merchant and at the end of the day we would process settlements for them, too. Settlement is a very crucial step in the process as that is when the merchant gets paid. One day, this process was broken. We got angry calls from merchants saying their account doesn't have the money in it. Some merchants need this money to open shops for the week. It was serious and our team was tasked to find out and fix the problem.

3.1.1 Fishbone Diagram

We used a Fishbone diagram (also known as an Ishikawa diagram) for the analysis. This has three parts to it. We have a head depicting the problem. Next are the categories which are the blue boxes representing the fish bones. These are the main contributors to the problem. Finally, the categories are further subdivided into causes.

The whole approach is quite visual and organizes the cause-and-effect relationships.

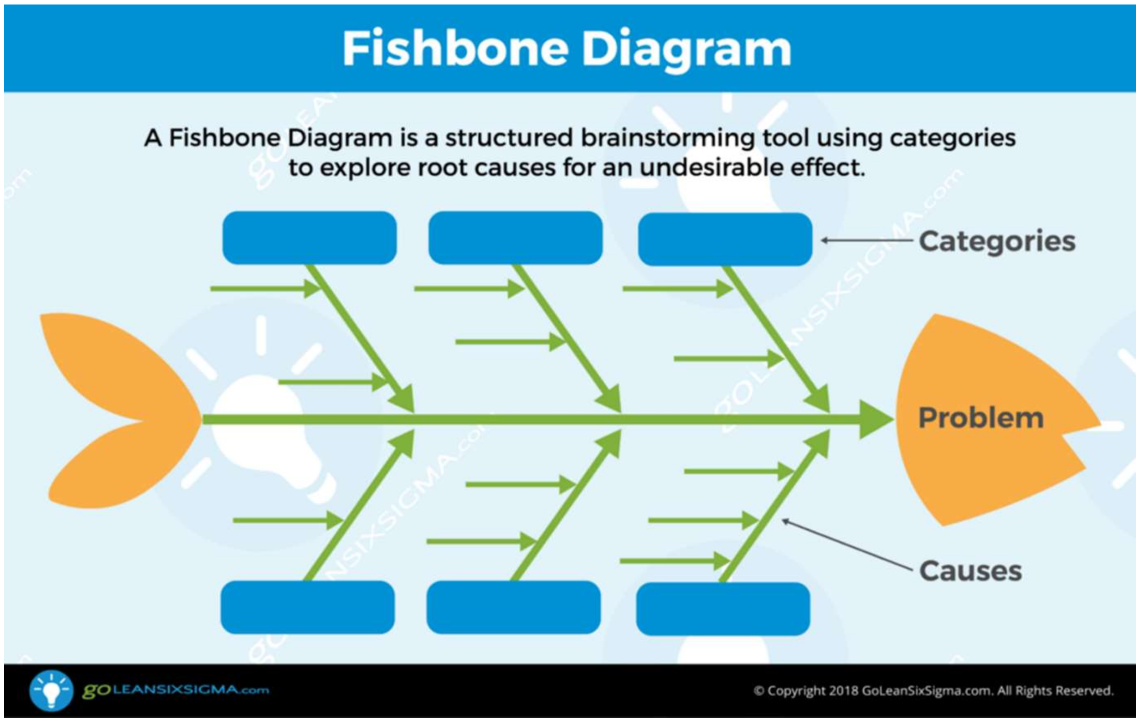

Fishbone Diagram
A Fishbone Diagram is a structured brainstorming tool using categories to explore root causes for an undesirable effect.
Categories
Problem
Causes
goLEANSIXSIGMA.com
© Copyright 2018 GoLeanSixSigma.com. All Rights Reserved.

Figure 3 Fishbone diagram (also known as an Ishikawa diagram)

We used the fishbone method to debug the problem of broken settlements. Our categories were Requirements, Deployment, Development, and Testing as we felt these would be the main contributors. Then we started breaking down the categories further and asking questions: Did we build it against latest specs? Was the release deployed? Did we use feature toggles? Did it work in the test environment?

After answering these questions, we were able to discount the categories Requirement and Deployment as they seemed to be in order. We used a different technique to deep dive and brainstorm the other problem areas.

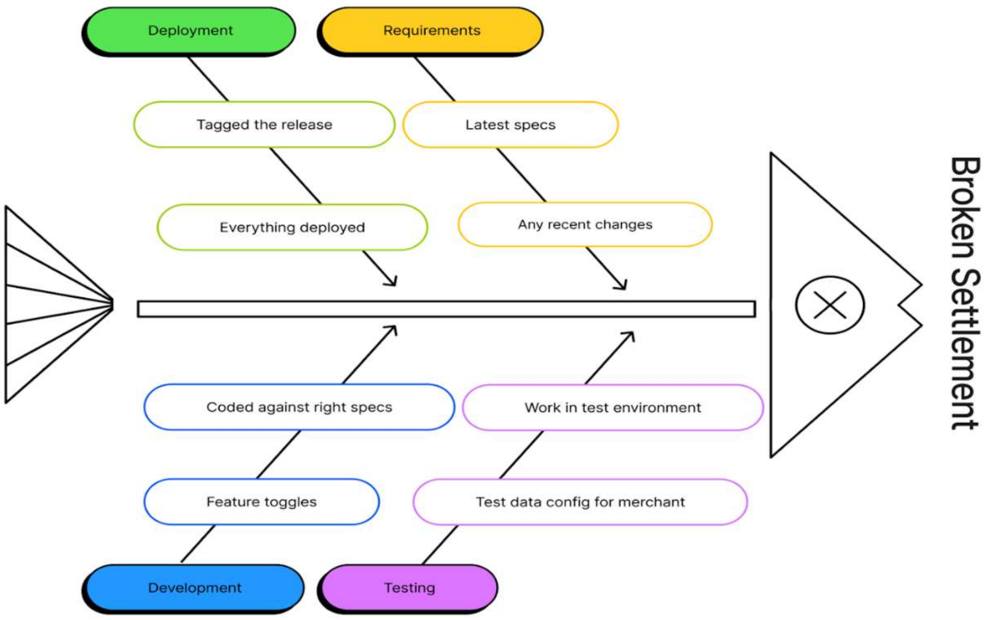


Figure 4 Debugging settlement bug using Fishbone diagram

3.1.2 5 Whys Technique

This technique is all about asking the ‘Why’ question and, for each answer, ask ‘Why’ a total of five times. With every ‘Why’ you move from one symptom to its underlying cause. We were still trying to reproduce the problem in our test environment, so we started with that as our first question: Why can’t we reproduce the issue in our test environment? The answer was, “Prod Merchant config was different.” Following which we changed our test setup to match production, tried again, and still couldn’t reproduce. Then we asked the next question: “Why can’t we still reproduce it?” The answer was, “Maybe it is the merchant’s setup causing the issue.” We needed to know what was happening on the bank’s side and they were eager to solve this issue, too. We set up a joint call to investigate. Still, we couldn’t reproduce the issue and asked another “Why?” as the problem was not us, but the bank. So, who else was in the equation? The answer was: Maybe a 3rd party. Following which we started comparing their test and prod environments. They had a 3rd party API integration which they were simulating, so we switched it off and tried again. Bingo! That was the issue. The 3rd party did some code changes, which started adding an extra character in the responses to the bank. The bank was sending us the same response and it somehow changed the order of the transactions in our settlement file. This was the issue causing the settlement process to fail. We got the 3rd party to fix their changes, the bank updated their mocks, and we updated our tests to additionally validate the order of the transactions.

We were feeling like Sherlock Holmes at this point, and I have been using the Fishbone diagram and 5 Whys for any sort of root cause analysis ever since. I have used this technique & framework to debug problems in requirements, designs and even processes. Understanding the problem well is a job half done. So, do spend the time to investigate and isolate before fixing anything.

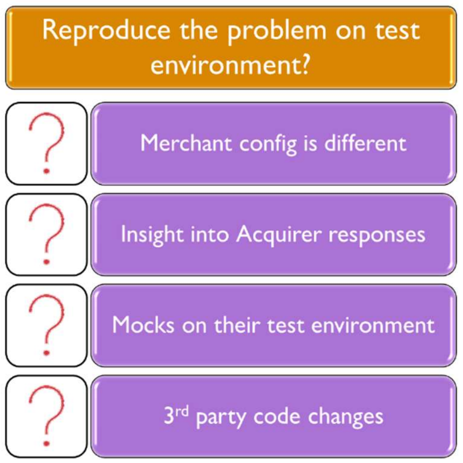

Reproduce the problem on test environment?
Merchant config is different
Insight into Acquirer responses
Mocks on their test environment
3rd party code changes

Figure 5 Debugging settlement bug using 5 Whys technique

4. Becoming technically competent

The first experience helped me understand how important those investigation skills were. The next experience got me curious to look beyond the black box. As a tester I became very good at asking questions, exploring the product, understanding logs, and analyzing bugs. Never really ventured inside the box or looked though the code. This posed a gap which held me at bay during technical conversations with the developers. So, I took baby steps to read code and find out how the software works.

4.1 Debugging failed builds

Debugging failed builds was the next steppingstone in my journey. A failing test is an opportunity to improve your understanding and your code. This is also when I fell in love with the debuggers because it gives you an opportunity to be transported into the code. Let’s look at Chrome Dev Tools which I frequently use.

Demo_ Get Started
Debugging JavaScript

Figure 6 A video of using Chrome Dev Tools to debug javascript

I found a Twitter post from Julia Evans explaining the amazing debugger features (Figure 7 . Most debuggers have conditional breakpoints, the ability to edit and continue, set watchpoints, backtraces, do time travel, and establish dependencies between breakpoints. It is worth investing some time to get to know the debuggers for your languages.

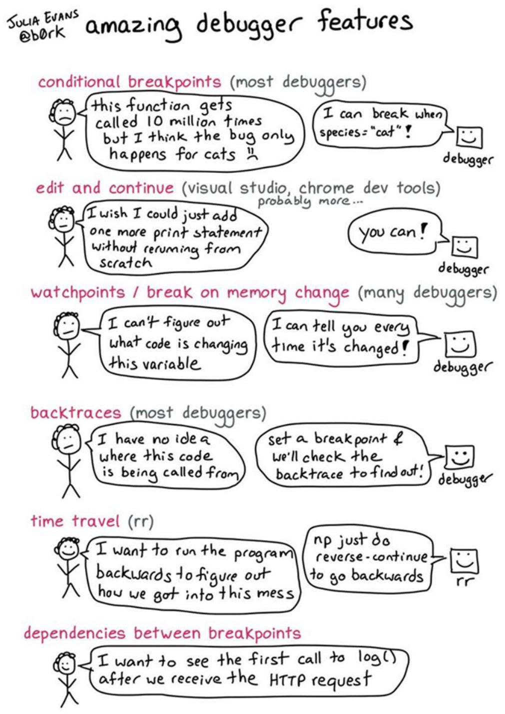

JULIA EVANS @b0rk amazing debugger features
conditional breakpoints (most debuggers)
this function gets called 10 million times but I think the bug only happens for cats !!
I can break when species="cat"!
debugger
edit and continue (visual studio, chrome dev tools) probably more...
I wish I could just add one more print statement without rerunning from scratch
you can!
debugger
watchpoints / break on memory change (many debuggers)
I can't figure out what code is changing this variable
I can tell you every time it's changed!
debugger
backtraces (most debuggers)
I have no idea where this code is being called from
set a breakpoint & we'll check the backtrace to find out!
debugger
time travel (rr)
I want to run the program backwards to figure out how we got into this mess
np just do reverse-continue to go backwards
rr
dependencies between breakpoints
I want to see the first call to log() after we receive the HTTP request

Figure 7 Twitter post from Julia Evans explaining amazing debugger features

One of my personal favorites is Intellij for its versatility. With Postman for APIs, you can use the API client, mock server, and console. I use Datadog for logging, monitoring options and session replay to view real time user journeys. And then there's Charles, a reverse proxy tool which records all the traffic between your machine and the Internet. Now-a -days there is an abundance of tools based on your tech stack and needs.

It is easier to learn or upskill on the job instead of through books or courses since you can practice and implement it straightaway. It is easier to relate to and it will stick with you when you use it a lot. So, find a problem worth solving and debug it. It is a rewarding experience. If possible, find an accountability partner. Once a fellow tester and I spent a couple of hours every week mapping out all the backend hops for our main customer journeys by referencing the UI wireframes designs and the architectural diagrams. For example, when a user searches something on the website, what is the microservice or API being called, what DB populates the data, what does the logging look like, and what is the ID which ties all the hops together. This was super handy for all of us for future debugging and to understand the traceability.

4.2 Wolf Fence Algorithm

What is this algorithm? We will answer it through a question: There's one wolf in Alaska, how do you find it?

You would first build a fence down the middle of the state, wait for the wolf to howl, and determine which side of the fence it is on. In debugging, this means you add some logging/print statements.

Repeat the process on that side only, until you get to the point where you can see the wolf. This is kind of a divide and conquer solution and it works well when you are trying to find a needle in a haystack.

4.3 Bug Magic

Bugs are very good at hiding. While debugging, if you have a complex piece of code, try the different paths and test out different combinations as bugs change form and shape. Heisenbug is the name given to those bugs that only happens when you're not debugging / disappears when you are. Sometimes these are time sensitive bugs such as race conditions. Have you noticed the same test works fine locally on your machine but fails on Jenkins? Be conscious of those bugs and switch your debugging tool and technique for it. Regression testing is very important. We don't want to debug and fix one problem then introduce more problems.

4.4 Rubber Ducking

Debugging is very intense. Sometimes when you hit a wall and want a fresh perspective, Rubber ducking can help. It is about explaining your approach to another person as the process of explaining gives you clarity on the problem in turn allowing you to fix it. If you can't find a person nearby, you can also explain it to an object like a toy duck. Testers can provide valuable insights in this process as we are good at asking the question Why? I personally have benefitted from this process by being on both sides.

4.5 Over Thinking

While debugging the production bug, we understood the power of analysis and thinking. But sometimes we might go into overthinking mode. For those times, we need to quit thinking, dive into the problem and start looking.

4.6 Remote Debugging

Our team used to do a lot of pairing and mobbing sessions. This all came to stand still and not so effective sessions during the COVID pandemic due to remote working. So, we had to find something to help us work collaboratively.

The "code with me" option in some IDEs was a game changer to bring back pairing and mobbing with a remote setup. On Intellij, we could invite people for a session by giving them full/restricted access. Full access means they will be able to do whatever you can on your IDE: write code, debug, and run tests. This is very handy for mobbing as in a traditional setup where a single machine would be used to code and the drivers and navigators would take turns. You could also combine the session with a video/audio call. There are options to share screen, chat, raise hand, and add emojis. Once done with the session you can disconnect and turn off the access. This option is available on the free version too but there is a restriction on the session limits. We have a licensed version of the IDE and it is super useful.

5. Writing code and preventing bugs early

The final part of the experience was attempting to code. After learning all these new skills and taking part in these pairing and mobbing sessions, I was charged up to try it out myself.

5.1 Test Driven Development (TDD)

My respect for coding went to another level when I got the opportunity to pair with an awesome dev who was obsessed with clean code practices from Uncle Bob. If you are serious about coding, I would suggest taking Uncle Bob's courses or reading his books. He introduced me to the concept of Test Driven Development (TDD). It essentially means you start by writing a failing test. Then, you write just enough code to pass the test. Next, refactor/improve the code with tests intact. TDD is a type of preemptive debugging. This approach is far more effective than just coding as it will reduce bugs because the code is simpler to understand.

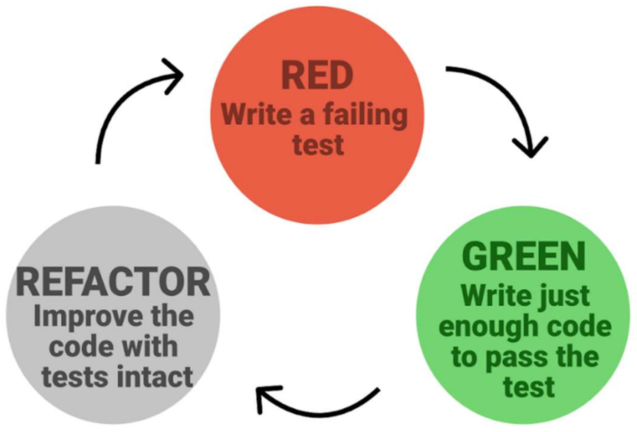

RED
Write a failing test
GREEN
Write just enough code to pass the test
REFACTOR
Improve the code with tests intact

Figure 8 A picture showing the TDD concept

5.2 Preemptive debugging & Defensive Programming

The debugging type we used for production bug and failed bugs are reactive debugging. We didn't anticipate the problem while building it, but we dealt with it after it happened. Preemptive debugging on the other hand, involves writing code that doesn't impact the functionality of the program but helps developers either catch bugs sooner or debug the source code more easily when a bug occurs. Preemptive debugging means adding better logs, retry mechanisms, and exception handling. Defensive Programming is another aspect of it where you don't let an error happen. It is about making the software behave in a predictable manner despite unexpected inputs or user actions. It is about writing foolproof code. For example, have you sent a mail saying there is an attachment and forgot to attach it? Defensive programming in this case would be to have a pop-up message saying "You have mentioned attachment in the mail, but there are no files attached. Do you want to send anyway?"

5.3 Code refactoring

Code refactoring is an integral part of TDD as it improves the overall code quality by reducing the complexity and maintenance. Anyone can easily read, understand, and work on it. In the long run, it also saves time and cost. My favorite refactoring option on IntelliJ is renaming. No need to find and replace, just use the renaming option. The Extract option can be used on multiple items like method, number, field, variables, parameter. If you have a big method, you can use this option to split that method into smaller methods. The Inline option is the reverse mechanism of Extract. If you want to combine two or more methods into one, this option can be used. Instead of reinventing the wheel, use the options these IDEs provide.

To sum up my experiences, first was the Sherlock Holmes mode. Next was being technically competent and understanding code mode. And the 3rd one was writing code mode.

6. Summary and Takeaways

1. The four steps in debugging are: Identify, Isolate, Fix and Review. You can debug all sorts of problems, not just code related ones.

2. Bugs and debugging make you better. Have the right attitude and take it as an opportunity to improve.
3. Know the problem well before solving it. Take the time to investigate and understand.
4. Sometimes you need to quit thinking and start looking at the logs or just get into it.
5. The best way to learn something new is to find the problem you are passionate about solving and then your learning becomes easy and focused.
6. Debugging is intense. It’s mentally challenging. Take breaks, talk to people, and practice rubber ducking.
7. Don’t panic. Ask for help and have fun! You are not alone.
8. Celebrate the wins and share stories. Do this to motivate yourself and others.

7. References

Wikipedia reference: https://en.wikipedia.org/wiki/Debugging

Website:

Figure1: Debugging: https://en.wikipedia.org/wiki/Debugging

Figure2: Online payment flow: https://stripe.com/in/guides/introduction-to-online-payments

Figure3: Fishbone diagram: https://goleansixsigma.com/cause-and-effect-diagram/

Figure7: Julia Evan's Twitter post: https://twitter.com/b0rk/status/1144011000208863239

Figure8: Test Driven Development: https://goodfangsm.life/product_details/70761037.html