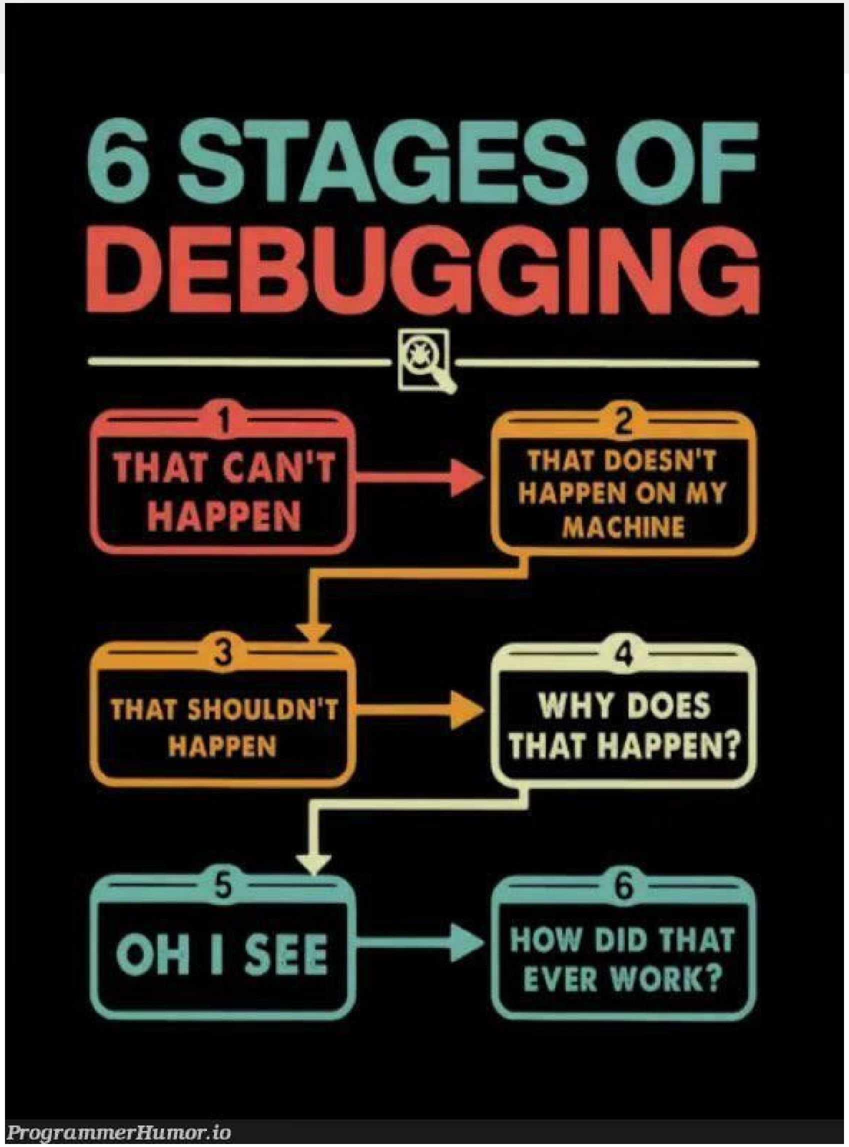
Debugging

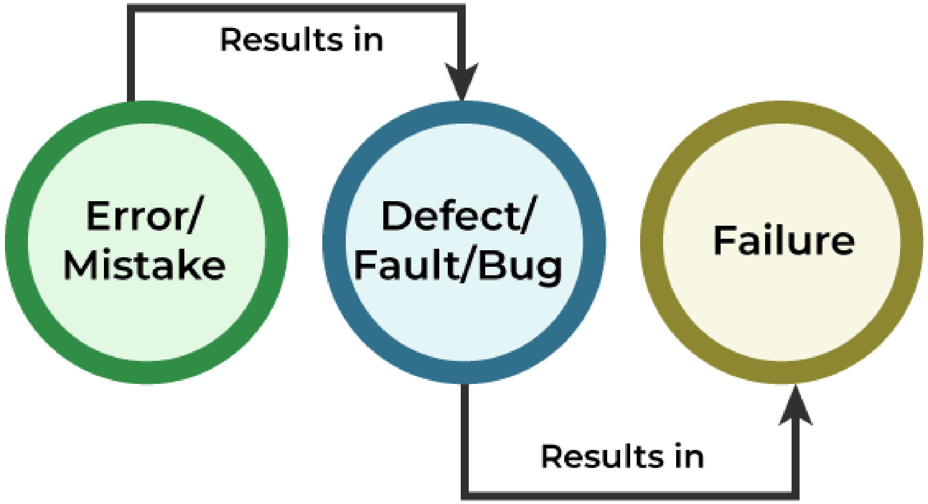
Debugging

- The process of finding bugs in code
- There are systematic ways to debug
  - Most developers “debug by poking around”
  - They don’t follow any process
  - Results in large amounts of wasted time
- Debugging has a set of best practices
  - Reduces time and effort in debugging



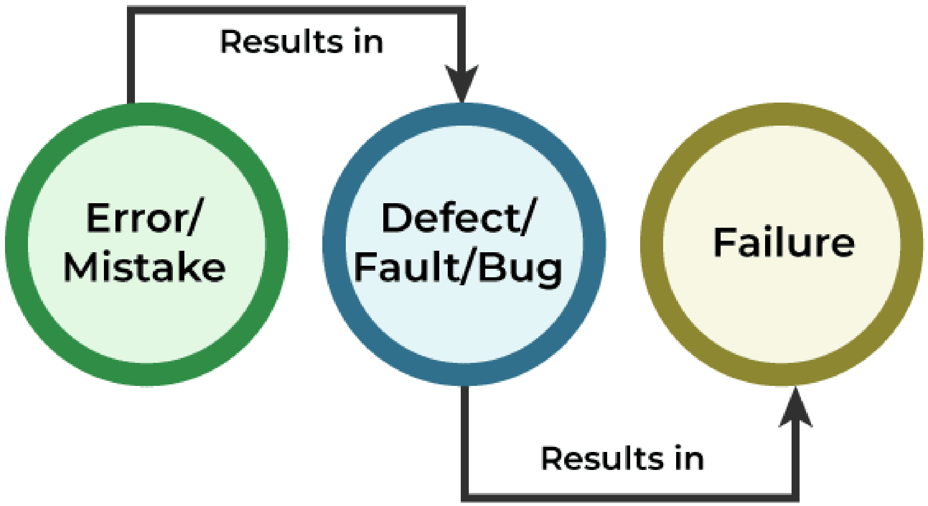
Defining Bugs

- There are three kinds of problems that are all grouped together as “bugs”
- Failures:
  - These are where the expected behavior is not the same as the actual behavior
  - Refers to any deviation from what should happen
    - *“The computed value isn’t correct”*
    - *“The file should be updated but it’s not”*
    - *“The system just hangs when we run the code”*
  - Failures are generally the start of a debugging process



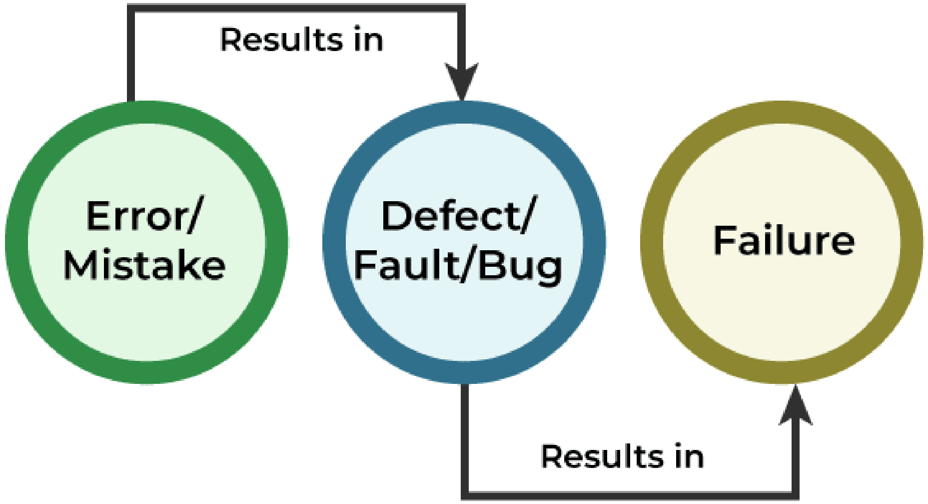
Defining Bugs

- Faults:
  - This is the code or design that was built incorrectly that resulted in a failure
  - The process of debugging is working backwards from a failure to find the underlying fault
  - This can be problematic since there is not necessarily a one-to-one relationship between faults and failures



Defining Bugs

- Error:
  - This is the action taken by the developer that resulted in a fault
  - This is often just a mistake in coding
  - But it can be the result of other factors
    - *The code was written correctly, but it was the wrong code*
  - Often the result of faults in earlier stages of the development process
    - *The specification was incorrect when it said the system should do this*
    - *We forgot to tell you in the requirements that range of inputs should be rejected*
  - Also the result of the developer misunderstanding what the code is supposed to do
    - *Test Driven Development helps resolve this one*



Testing vs Debugging

- Testing
  - Exercising the code with a set of test cases to see if any failures occur
  - The goal is to raise the possibility that faults exist in the code before it goes into production
  - Testing failures tell us that debugging to find the underlying faults should be done
  - Testing doesn’t fix anything, it just tells us something is broken
- Effective debugging is enhanced when comprehensive testing is done
  - Find bugs early often makes the debugging process simpler
  - Once the code is production, the complexity of the environment in which the failure is occurring can make debugging exponentially more difficult.

Types of Faults

- A fault can occur anywhere, but they can be grouped into types based on where they occur
- Syntax bugs
  - These are errors in grammar of the language, for example a missing colon in Python.
  - These generally cause the program not to compile, which is the failure
  - IDEs and linters often highlight these errors and have auto-correct tools
- Semantic bugs
  - These are the coding version of typos that pass a spellcheck
  - The code is grammatically correct but contains an incorrect reference
    - *Using the wrong variable in a computation*
    - *Calling the wrong function or the wrong operator*
  - Often shows up as a runtime failure even though it compiles

Types of Faults

- Logic bugs
  - Where the programmer has used incorrect logic
    - *Implementing the wrong algorithm or program logic*
    - *The logic might also be incomplete and will fail in some cases*
- Runtime errors
  - These occur during execution
    - *Divide by zero, file not found, etc*
  - These can usually be tracked back to a fault that didn't check for valid data or program states
- Integration bugs
  - Individual components are fault free
  - The failures occur when the modules interact
    - *The fault is often a mismatch in API or the interface*

Types of Faults

- Concurrency bugs
  - These are often the most difficult to debug because of how they arise
    - *These include race conditions, deadlocks, nondeterministic scheduling issues*
- Heisenbugs
  - Reference to the Heisenberg observer effect
  - The act of measuring or observing a system changes the behavior of the system
    - *The act of observing a system makes the observer part of the system which changes the behavior of the system*
  - A heisenbug is one that disappears when we add logging or other debugging tools
    - *Often due to the effect of the added debugging code on the timing or execution of the system*
- Nondeterministic bugs
  - These are failures that only happen sometimes
  - These often depend on the system being in a particular state
    - *Replicating that state may be difficult or something we can’t figure out*
    - *Concurrency bugs are the common type of nondeterministic bugs*

Root Cause Analysis

- Failures are often not due to a single fault
  - Root Cause Analysis is a systematic process for identifying the fundamental cause of a failure
    - *For example, a failure occurs because of a faulty calculation*
    - *The code that does the calculation is the source of the failure*
    - *The code might not be the fault, but it might be receiving bad data from another module*
    - *The data source module is the root cause of the failure, not the code that did the computation*
- Key Principles
  - Symptoms vs. Cause
    - *symptom: Observable issue (program crash, wrong output).*
    - *Root cause: The underlying defect that triggers the symptom.*
  - Multiple Layers of Cause
    - *Often, there are contributing factors (e.g., missing test cases, unclear requirements, coding mistake).*
    - *True RCA digs through layers until the first event that set off the chain is found.*
  - Fix the Process, Not Just the Error
    - *Correcting only the immediate defect often leads to recurrence.*
    - *RCA aims to fix upstream causes (design flaws, lack of validation, missing tests).*

Root Cause Analysis

- The 5 Whys Technique
  - Method for digging deeper into a problem
  - Start with an observed failure
  - Ask “Why did this happen?”.
  - The answer is the basis for the next “Why” question.
  - By the fifth “Why,” the root cause is often reached

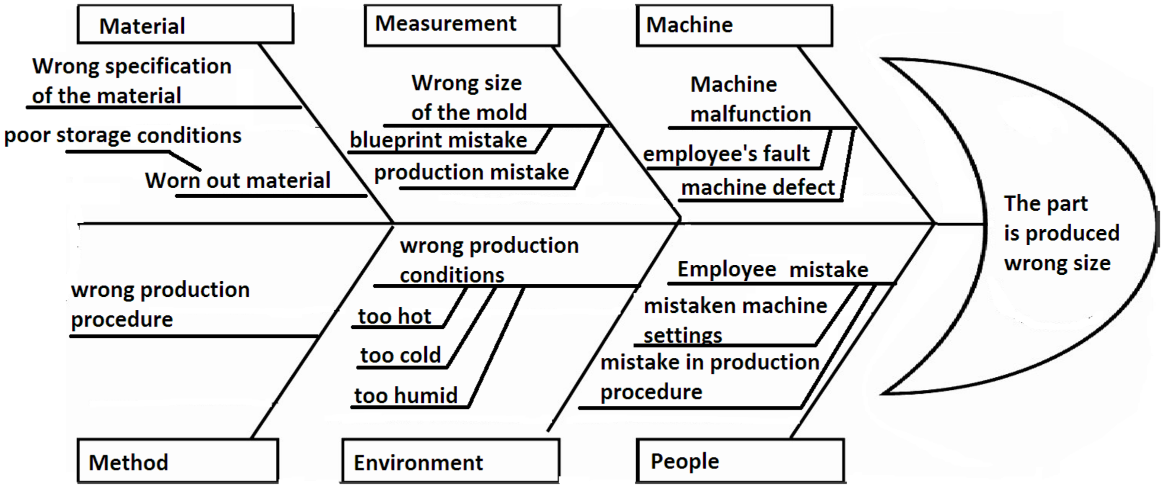
Root Cause Analysis Example

- Symptom: Program crashes with ZeroDivisionError.
- Why 1: Why did the crash occur?
  - Because the code attempted to divide by zero
- Why 2: Why was zero used as a divisor?
  - Because the variable count was set to 0
- Why 3: Why was count set to 0?
  - Because no records were loaded from the database
- Why 4: Why were no records loaded?
  - Because the database connection string was invalid
- Why 5: Why was the connection string invalid?
  - Because the deployment script was missing environment variable substitution
- Root Cause: Missing validation in deployment configuration.
  - Fix: Add environment variable checks and automated integration tests

Ishikawa Fishbone Diagram

Developed by Kaoru Ishikawa (1960s), originally for quality management.

- Visual tool that organizes possible causes of a problem into categories, shaped like the bones of a fish.
- Helps teams brainstorm and classify causes, especially when there are many possible factors.
- The “fish head” = the failure
- The “bones” = major categories of causes.
- Sub-branches = specific factors within each category.



Structured Decomposition Debugging

- Fault Tree Analysis (FTA)
  - Top-down, deductive method used to analyze the causes of system failures
  - Helpful when a fault may due to a combination of faults
- Starts the failure at the top
  - Then breaks it down into all possible causes using a tree structure
  - Uses logic gates (AND, OR) to model how combinations of lower-level faults lead to higher-level failures
  - Especially useful in safety-critical systems like aviation, telecoms, medical devices
  - Can also be applied to software debugging.

Structured Decomposition Debugging

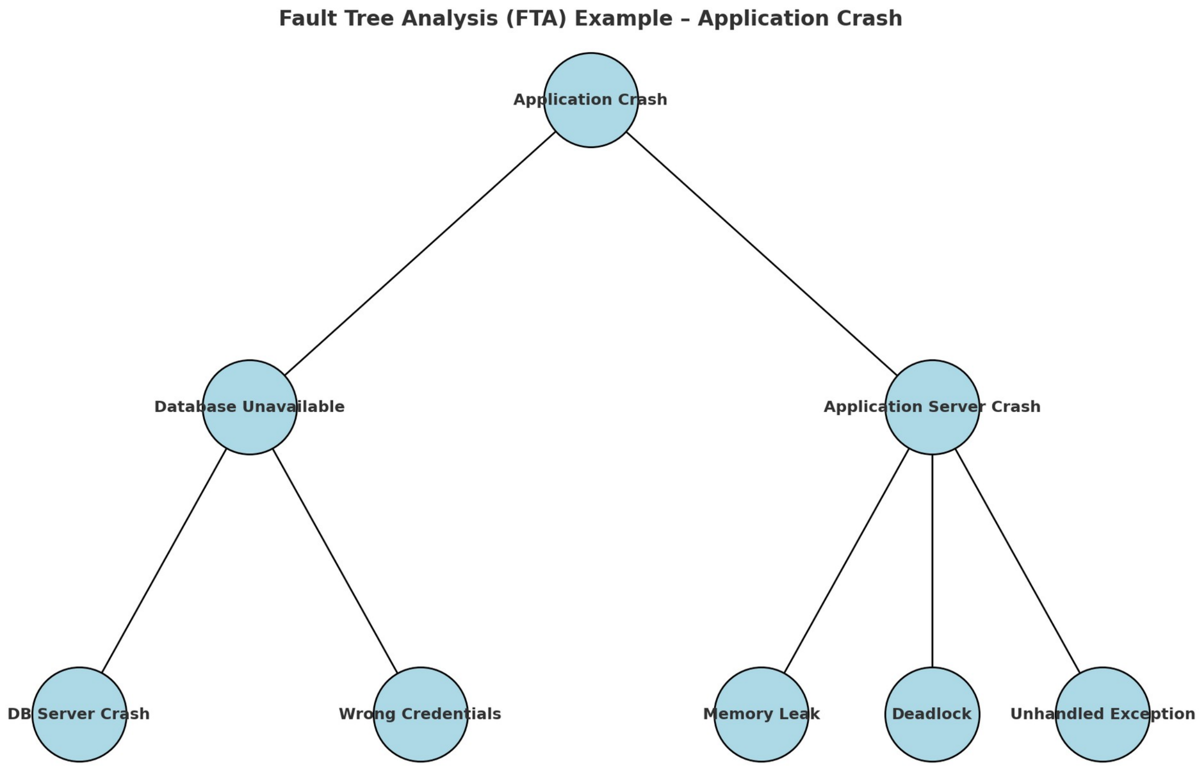
- Define the top event
  - The failure you’re analyzing (e.g., “Web application crashed”)
- Identify immediate causes
  - These are the next level down (e.g., “Memory exhaustion” OR “Database connection failure”).
- Decompose further
  - Keep asking: What could cause this? until you reach basic causes (coding error, misconfigured environment, faulty input)
- Use logic gates
  - AND gate: failure occurs only if all sub-causes happen together
  - OR gate: failure occurs if any sub-cause happens
- Analyze minimal cut sets
  - The smallest combinations of failures that can cause the top event

Structured Decomposition Debugging

Event: Application Crash

Main branches (causes):

- Database Unavailable
  - DB Server Crash OR
  - Wrong Credentials
- Application Server Crash
  - Memory Leak OR Deadlock OR
  - Unhandled Exception



Structured Debugging Process

- Debugging should not be “trial and error”
- Structured debugging applies the scientific method to debugging
  - Systematic in the same way scientists investigate natural phenomena
  - Scientific method provides a structured way to diagnose, experiment, and confirm solutions
- Step 1: Define the problem (observation & problem statement)
  - Goal: Clearly describe what went wrong
  - Document:
    - *Intended outcome: What should have happened?*
    - *Actual outcome: What did happen?*
    - *Symptoms observed: error messages, incorrect values, performance issues.*
    - *Example: “Program should return the sum of numbers but instead throws a TypeError when inputs are mixed types.”*

Structured Debugging Process

- Step 2: Gather data (background research)
  - Collect logs, stack traces, system metrics, user reports
  - Check environment factors: OS, versions, dependencies
  - Ask: Has this program ever worked? When did it last run successfully?
- Step 3: Form hypotheses (possible Ccuses)
  - Brainstorm potential causes (“list of suspects”)
  - Consider:
    - *Recent code changes*
    - *Data/input anomalies*
    - *External systems (APIs, DB connections)*
    - *Environment (hardware, libraries, OS differences)*
  - Example: “The error might be due to improper type conversion in function X.”

Structured Debugging Process

- Step 4: Experiment (test hypotheses)
  - Divide and conquer: isolate code sections until the faulty part is narrowed down
  - Change one thing at a time to keep experiments valid
  - Use controlled inputs, mock data, or stubs
  - Example: Replace real DB with a test DB to see if the bug persists
  - Logbook: Record every experiment: what was changed, the result, and the conclusion
- Step 5: Analyze results
  - Compare expected vs. observed behavior
  - For every debugging experiment, you should already have defined:
    - *Expected outcome → based on your hypothesis (“If the bug is caused by X, then doing Y should fix or reproduce it”).*
    - *Observed outcome → what actually happened when you ran the test*

Structured Debugging Process

- Example:
  - *Hypothesis: “Changing the input encoding will fix the parsing error”*
  - *Expected: Program processes file successfully*
  - *Observed: Program still fails, but with a different error. (didn’t align)*
- Refine or discard hypotheses
  - *If results align then the hypothesis likely correct, then apply a fix*
  - *If results don’t align, then refine (adjust theory) or discard (move on).*
  - *This prevents wasted time chasing the wrong explanation*
- Don’t ignore anomalies
  - *Critical rule: Any result that doesn’t fit your current theory might be the real clue*
  - *Debuggers often fall into confirmation bias where they are only noticing evidence that supports their idea, while dismissing outliers*
  - *Those “weird cases” often reveal hidden dependencies or concurrency issues*
- Example:
  - *A bug appears only on Mondays but not any other day*
  - *Easy to dismiss, but digging deeper reveals it’s tied to a monthly batch job that interferes with the database lock*

Structured Debugging Process

  - Look for patterns
    - *Analyze whether results are consistent, intermittent, or random*
    - *Consistent: usually indicates a logic error (e.g., always off by one)*
    - *Intermittent: often points to environmental or concurrency bugs*
    - *Random/nondeterministic: might be race conditions or memory corruption*
  - Iterative loop
    - *Each analysis either validates the cause or loops back to forming a new hypothesi.*
- Step 6: Identify root cause
  - Drill down to find the underlying issue, not just the surface symptom
  - Techniques:
    - *5 Whys*
    - *Structured decomposition breakdown*
    - *Fishbone Diagram (categorize causes: people, process, tools, environment, etc.).*
- Step 7: Apply the fix
  - Implement a solution targeted at the root cause
  - Validate that the fix eliminates the problem without side effects
  - Use regression tests to ensure nothing else is broken

Structured Debugging Process

- Step 8: Verify and document
  - Retest with original failing inputs and additional test cases
  - Document:
    - *The problem, root cause, and fix*
    - *Any process improvements (e.g., better tests, coding standards)*
    - *Knowledge sharing prevents recurrence across the team*

Psychology of Debugging

- Debugging is deeply influenced by human psychology.
  - How we think, what we assume, and where we focus influence the process
  - This is true in most sorts of cognitive activity, not just debugging
- Confirmation bias
  - Tendency to look for evidence that supports our existing beliefs and ignore evidence that contradicts them
  - In debugging:
    - *“I know this function works — it can’t be the problem”*
    - *“It passed the unit tests, so it must be fine”*
  - Result: Time wasted chasing the wrong cause
  - Better approach: Doubt everything. Even tested, “proven” code can break under new conditions

Psychology of Debugging

- Assumptions
  - We often assume external components are reliable
    - *Example:“The database library is from a trusted vendor, it can’t fail”*
    - *“The environment is the same as last time”*
  - Reality: Libraries have bugs, and environments change (OS patches, configuration drift)
  - Better approach: Verify external dependencies — check logs, versions, and environment differences
- Tunnel vision
  - Locking onto a single hypothesis and ignoring alternative explanations
    - *Example: Spending hours rewriting a function because you believe it’s wrong, when the real issue was a bad test file.*
  - Tunnel vision often happens under time pressure
  - Better approach: Use a structured debugging approach

Psychology of Debugging

- Selective perception
  - We often see only what we think is there, not what is actually there
    - *This is why we can't proofread our own writing*
    - *And spot the bug in our code, we are not seeing what is actually there*
    - *When it's pointed out, the reaction is often "How could I have missed that?"*
  - Better approach: Get another set of eyes to examine the code

Habits of Effective Debuggers

- Reproduce the error independently
  - Don’t rely only on reports, make the bug happen in your controlled environment
  - This ensures you’re solving the right problem
- Keep an open mind
  - Any part of the system (even the “obvious” parts) could be at fault
  - The bug may not be where you first expect it
- Take breaks
  - Stepping away clears mental bias, many developers report finding solutions after a break or sleep.
- Peer conversations
  - Explaining your code to a teammate (or even a “rubber duck”) forces clarity of thought, often revealing flawed assumptions
- Document your thinking
  - Writing down hypotheses and results prevents cycling back into the same wrong assumptions

Manual Debugging Techniques

- Rubber duck debugging
  - Definition: Explaining your code out loud to a “listener”
  - Term originated from a developer claiming they debugged by explaining their code to a rubber duck while taking a bath
  - Now it just means the process of explaining your code out loud
  - Why it works:
    - *Forces you to articulate your logic step-by-step*
    - *Breaks the habit of skipping over “obvious” parts*
    - *Often reveals hidden assumptions or logic gaps*
  - Example:
    - *Developer explains: “This function returns the number of users… oh wait, I’m counting inactive users too!”*

Manual Debugging Techniques

- Logging
  - Definition: Adding statements (e.g., print(), console.log()) to track variable values and program flow
  - Advantages:
    - *Creates a permanent record for later analysis*
    - *Helps trace issues that happen intermittently or in production*
    - *Easy to add in almost any language*
  - Disadvantages:
    - *Can slow performance (especially if logging in tight loops)*
    - *Excessive logs can clutter output, making patterns harder to see*
  - Best practices:
    - *Use log levels (DEBUG, INFO, WARN, ERROR)*
    - *Log context, not just values (e.g., “Order ID=123 failed payment: NullReferenceError”)*
    - *Ensure logs are easy to search/filter*

Manual Debugging Techniques

- Tracing (step-by-step inspection)
  - Definition: Following the execution path manually, often with print/log statements or breakpoints
  - When it is useful to use:
    - *To confirm control flow (e.g., which branch of an if is taken)*
    - *To check loops, recursion, or function calls*
  - Example: Adding logs inside a loop to check index values, or before/after function calls to confirm execution order

Manual Debugging Techniques

- Code Walkthroughs
  - Definition: A structured peer review where the author walks others through the code.
    - *The step-by-step execution of the code is described*
    - *Example test cases are used*
    - *Essentially running the code manually*
  - Benefits:
    - *Fresh eyes often catch errors the author overlooks*
    - *Encourages knowledge sharing within the team*
  - Difference from Rubber Ducking:
    - *Walkthroughs are collaborative and can include feedback, while rubber ducking is one-way*
  - There are a number of formal code walkthrough methodologies
    - *Very common in software engineering*
    - *Often a routine part of a code quality and correctness process*

Manual Debugging Techniques

- Code Inspections
  - Definition: A formal review process where a team inspects code systematically.
    - *Typically the code is compared against a coding standard or set of best practices*
    - *Looks for places where the code deviates from the best practices*
    - *There is no manual execution of the code*
  - Benefits:
    - *Has a high rate of catching semantic and logical errors*
    - *Also identifies places where the code structure is not effective*
    - *Identifies areas where the code is non-compliant with standards such as security practices*
  - Like code walkthroughs, code inspections are
    - *Very common in software engineering*
    - *Often a routine part of a code quality and correctness process*

Using Debuggers Effectively

- A debugger allows you to pause execution in an executing program
  - Step through code line by line
  - Inspect the internal state of the execution environment
  - Often requires the code to be compiled with extra information so the debugger can find the code to be stepped through or examined
    - *For example, the names of variables in the source code might not be in the compiled code*
    - *Unless the compiler is instructed to remember them for debugging purposes.*

Key Debugger Features

- Breakpoints
  - A marker placed on a specific line of code where execution will pause
- Use cases:
  - To pause the program just before the suspected faulty section
  - To skip irrelevant parts of the code and go straight to the area of interest
  - Example:
    - *Set a breakpoint on the line that processes a user’s login credentials*
    - *When the program halts there, inspect variable values to confirm correctness*

Key Debugger Features

- Watchpoints
  - Execution pauses when a specific variable’s value changes
    - *We might not know where to set a breakpoint if there are multiple places a variable might change*
- Use cases:
  - To detect where/when a variable is unexpectedly modified
  - To track down bugs involving “mysterious value changes” or shared state
  - Example:
    - *If a global variable balance changes unexpectedly, add a watchpoint*
    - *The debugger halts at the exact instruction where the change occurs*

Key Debugger Features

- Step Into / Step Over / Step Out
  - Control how you advance through the code:
  - Step Into: Moves execution inside the function being called, allowing you to debug line by line within it
  - Step Over: Executes the function call as a whole and moves to the next line, skipping the internal details
  - Step Out: Finishes the current function and returns to the caller
  - Choosing the right step option helps control the granularity of your investigation

Key Debugger Features

- Inspecting Stack Frames
  - Definition: A stack frame is the local execution context of a function
    - *Its parameters and local variables*
  - Debugger ability: Switch between active and previous stack frames
  - Why this is useful:
    - *To trace the sequence of function calls that led to the failure*
    - *Inspect variables not just in the current function, but also in the caller*
  - Example:
    - *A crash occurs deep inside a library function*
    - *Inspect the caller’s stack frame to see what parameters were passed in*

Key Debugger Features

- Call Stack Navigation
  - Definition: The debugger shows the ordered list of functions that have been called up to the current point (the call stack)
  - Use cases:
    - *Trace the execution path that led to the error*
    - *Identify unintended recursion or unexpected call sequences*
  - Example:
    - *Stack trace shows main() → processOrder() → validateCard() → nullReference()*
    - *Following this path pinpoints where the error originated*

Key Debugger Features

- Variable Watches
  - Definition: The debugger continuously displays the values of selected variables while stepping through code
  - Use cases:
    - *Monitor how state evolves over time*
    - *Detect logical errors (e.g., variable updated incorrectly in a loop)*
  - Example:
    - *Watching total in a shopping cart loop reveals it’s being reset instead of incremented*

Best Practices for Using Debuggers

- Combine breakpoints + watches
  - Stop execution at the right moment and immediately check variable states
- Use conditional breakpoints
  - Pause only when a condition is met (e.g., i == 1000). Saves time in large loops
- Don’t just step blindly
  - Have a clear hypothesis before you start debugging
  - Use other debugging techniques to narrow the code you want to use the debugger on
- Retest
  - After fixing the bug, recompile for production without debugging symbols
  - Then run tests to ensure the failure has been eliminated in the production version

Well-Designed Code and Debugging

- The quality of the codebase impacts how effective debugging can be
- Good design
  - Reduces the likelihood of bugs
  - Makes them easier to isolate when they do appear.
  - Poor design spreads problems across the system and obscures the root cause
- This section refers back to our sections on engineering principles and clean code

Well-Designed Code and Debugging

- Readable code means easier debugging
- Clear names
  - Variables, functions, and classes with descriptive names make the code self-explanatory
- Consistent style
  - Indentation, formatting, and naming conventions reduce cognitive load when reading
- Comments where necessary
  - Explain why something is done, not just what is happening
  - This helps describe what the code should be doing which makes it easier to see where it is doing what it should
- Readable code shortens the time it takes to understand what’s wrong

Well-Designed Code and Debugging

- Loose coupling and high cohesion
  - Loose coupling means components/modules have minimal dependencies.
    - *A bug in one module is less likely to cascade into others*
  - High cohesion means each component has a clear, focused responsibility.
    - *Makes it easier to localize faults by narrowing the focus of where to look*
  - In a tightly coupled system, changing one class may break five others
  - In a loosely coupled, cohesive system, the bug can be isolated in the module that owns the responsibility

Well-Designed Code and Debugging

- Error Handling & Exceptions
  - Structured error handling (try/catch, exceptions) helps the program fail gracefully and provide useful debugging info
  - Consistent and standardized ways of handling errors help spot bugs
  - Without structured handling
    - *Errors propagate chaotically and symptoms appear far from the cause.*
    - *The root cause of the error is much more difficult to ascertain*

Well-Designed Code and Debugging

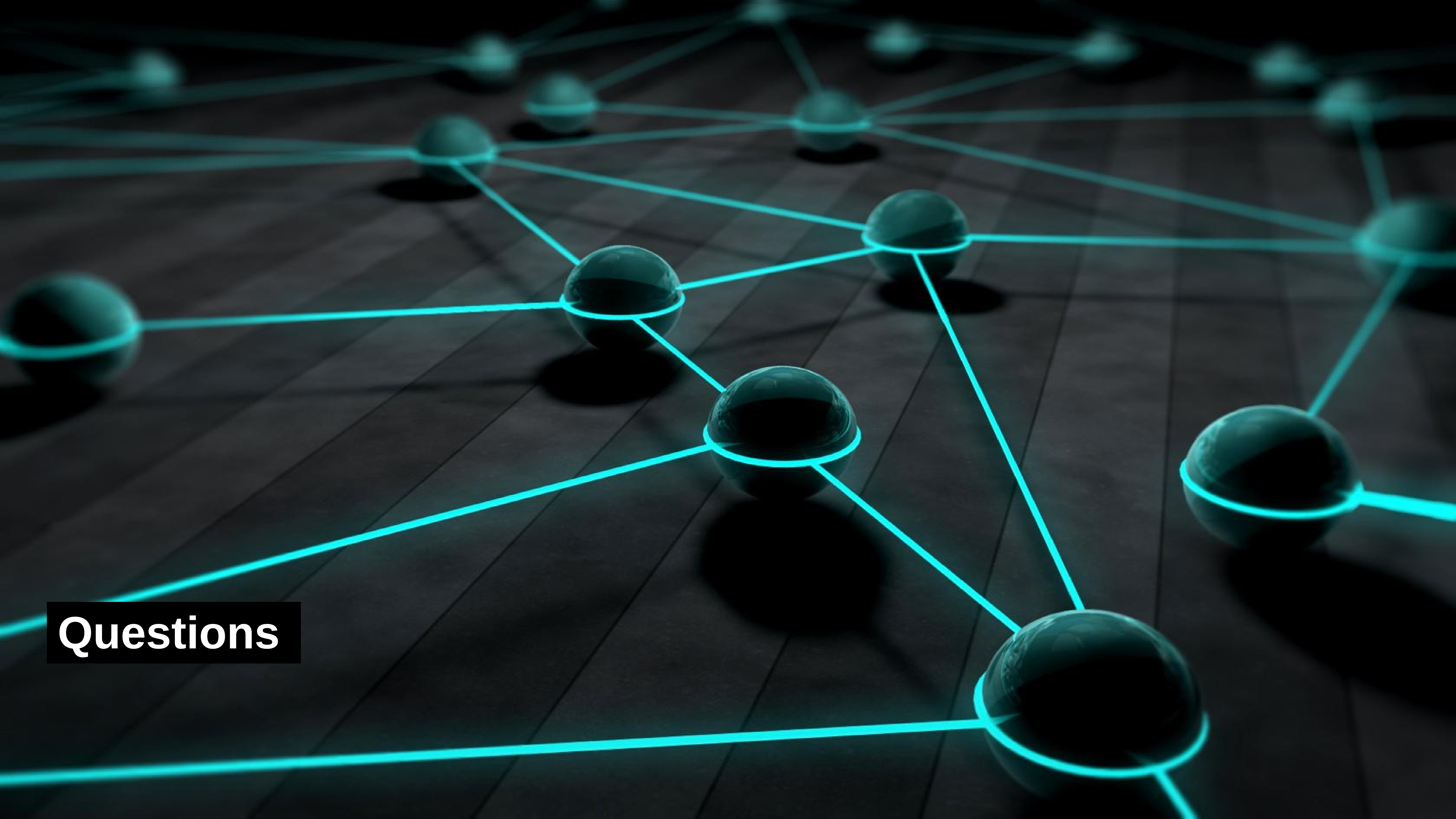
- Error Handling & Exceptions
  - Structured error handling (try/catch, exceptions) helps the program fail gracefully and provide useful debugging info
  - Consistent and standardized ways of handling errors help spot bugs
  - Without structured handling
    - *Errors propagate chaotically and symptoms appear far from the cause*
    - *The root cause of the error is much more difficult to ascertain*
- Good Test Coverage
  - Tests act as early warning systems: bugs are caught closer to where they originate
  - Unit tests make it easier to reproduce bugs in isolation early in development
  - Regression tests ensure that fixes don’t reintroduce old bugs
  - Debugging is faster when a failing test points directly to the faulty function

Well-Designed Code and Debugging

- Poorly structured code makes debugging more complex
- Code smells for poor code from a debugging perspective
  - Spaghetti code: Tangled logic, long functions, unclear flow
  - Global state abuse: Any part of the code can change shared data, making bugs unpredictable
  - No separation of concerns: Business logic, UI, and data access mixed together
- Example
  - In a “God Class” design, a bug could be anywhere in thousands of lines of unrelated code
  - Debugging becomes guesswork instead of systematic problem-solving

Well-Designed Code and Debugging

- Poorly structured code makes debugging more complex
- Code smells for poor code from a debugging perspective
  - Spaghetti code: Tangled logic, long functions, unclear flow
  - Global state abuse: Any part of the code can change shared data, making bugs unpredictable
  - No separation of concerns: Business logic, UI, and data access mixed together
- Example
  - In a “God Class” design, a bug could be anywhere in thousands of lines of unrelated code
  - Debugging becomes guesswork instead of systematic problem-solving



Questions