# The Debugging Mindset

## Understanding the psychology of learning strategies leads to effective problem-solving skills.

**Devon H. O'Dell**

Software developers spend 35-50 percent of their time validating and debugging software.[1] The cost of debugging, testing, and verification is estimated to account for 50-75 percent of the total budget of software development projects, amounting to more than $100 billion annually.[11] While tools, languages, and environments have reduced the time spent on individual debugging tasks, they have not significantly reduced the total time spent debugging, nor the cost of doing so. Therefore, a hyperfocus on elimination of bugs during development is counterproductive; programmers should instead embrace debugging as an exercise in problem solving.

It is more appropriate for programmers to focus their efforts on acquiring and encouraging effective learning strategies that reduce the time spent debugging, as well as changing the way they perceive the challenge. This article describes how effective problem-solving skills can be learned, taught, and mentored through applying research on the psychology of problem solving, conducted by Stanford's Carol Dweck and others.

### Debugging is Inherently Difficult

In the 1974 classic, *The Elements of Programming Style*, Kernighan and Plauger wrote, "Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"[6] Is debugging really twice as hard as writing a program? If so, why should it be?

To answer these questions, let's first consider why bugs occur. In a 2005 paper, Andrew J. Ko and Brad A. Meyers suggest that bugs occur as a result of "chains of cognitive breakdowns... formed over the course of programming activity."[7]

At a fundamental level, all software describes changes in the state of a system over time. Because the number states and state transitions in software can have combinatorial complexity, programmers necessarily rely on approximations of system behavior (called *mental models*) during development. The intent of a mental model is to allow programmers to reason accurately about the behavior of a system.

Since mental models are approximations, they are sometimes incorrect, leading to aberrant behavior in software when it is developed on top of faulty assumptions. Neophyte programmers experience this problem to a greater degree: their models of the language and development environment are necessarily incomplete, and even a simple syntax error can be a major blocker for someone learning to program. This isn't limited to novice programmers. The C11 language

specification is just north of 700 pages, for example. How many systems programmers understand the language completely?

The most sinister bugs occur when programmers falsely believe their mental models to be complete. This is the crux of the problem: they have assumed correctness in their implementations and by definition do not know where they went wrong. The only way programmers can hope to solve such bugs is through knowledge acquisition. While it remains unclear whether Kernighan is correct about the quantitative difference in difficulty between programming and debugging, it does seem clear that debugging is the more difficult task. Since solving bugs requires learning, the debugging process can be made easier by better understanding effective learning and teaching strategies.

## Teaching Debugging Skills

*Debugging* is simply a domain-specific term for *problem solving.* Bugs are described as word problems: for example, "An out-of-bounds write to object O causes corruption of adjacent memory when conditions A and B are both true." Unfortunately, word problems are found to be some of the most difficult to teach, and in software nearly all bugs can effectively be characterized as word problems.[13]

Perhaps to compensate for this difficulty, much existing research on the pedagogy of debugging focuses first on discriminating "experts" from "novices" and assessing the techniques each use in debugging tasks. It then attempts to improve the novice's ability through teaching expert techniques. A review of the literature, however, finds that even experts differ greatly in debugging skill.[8]

Those most effective at debugging draw from extensive experience, as well as refined problem-solving skills. They also employ generalized strategies for problem solving[13] instead of treating every individual bug as a new, specific case. While teaching the techniques of expert programmers can reduce the time novices spend on programming activities, it hasn't been shown to be effective in reducing the time spent debugging. Furthermore, testing students from groups that have and have not received expert technique intervention does not yield statistically significant differences in test scores.[2]

Experience and accurate models are precisely the tools that the novice programmer lacks. Although computer science education devotes a lot of time to teaching algorithms and fundamentals, it appears that not much of this time is spent applying them to general problems. Debugging is not taught as a specific course in universities. Despite decades of literature suggesting such courses be taught, no strong models exist for teaching debugging.

The problems are what to teach and how to teach it. Carefully considering research in the field of psychology can be helpful in both understanding students' needs and designing appropriate curricula.[13] Instructors, mentors, and educators should therefore be aware of the relevant research in this area to guide what they are teaching and how they are teaching it. Individuals must be cognizant of how to approach problems, and whether they are perceived as limits of ability or as part of the learning process.

## A Note on Tools and Environments

Significant effort has been expended on developing tools, languages, and programming environments aimed at reducing or eliminating bugs. Tools (once their use is understood) undoubtedly save time in the debugging process, but they can't solve all problems. What if no tool exists for a specific problem? What if available tools don't scale to the load required to reproduce an issue? What if you're unaware of the tools, or unable to afford them? Tools aren't a panacea; when they don't exist (or are otherwise unavailable for use), you need to be willing and able either to write them or to forego their use.

Virtual machine-based languages, interpreted languages, and languages with runtime environments encourage users to view the execution environment as a black box. The goal here is to make programming easier by reducing the scope of the mental model the programmer must maintain. When bugs occur in these execution environments, you're left with a complete gap in understanding. You might solve this by understanding more about the execution environments, but then what is the point of that abstraction? Such runtime environments are not a panacea; you still need to understand how they behave.

Finally, extensive research exists in the area of formally verifiable languages. These environments allow programmers to prove the correctness of their code while solving some problem, but they do not help us understand the problems they are solving in the first place. If you don't fully understand a problem and then implement a "provably correct" solution based on incomplete understanding, debugging will still be required. Such languages are not a panacea; you still need a complete understanding of the problems to be solved before writing your software.

Since bugs occur because of an incomplete mental model, the philosophy of relying entirely on tools, runtime environments, and languages to catch bugs seems self-defeating. It's almost as if we are saying that we can't possibly be smart enough to understand these bugs, so why even bother? Such a view falls squarely to the side of the "entity theorist" as described by Carol Dweck's research on "self-theories of intelligence."[5]

## Self-Theories of Intelligence

Dweck, a leading researcher in the field of motivation, is responsible for four decades of research that attempts to identify and characterize behaviors of high-achieving individuals. Her work has been repeatedly validated in many studies across individuals of varying cultures, genders, ages, and socioeconomic backgrounds.

Dweck proposes that individuals fall somewhere on a spectrum of self-theories. This spectrum varies from the *entity theorist* on one end to the *incremental theorist* on the other. An entity theorist tends to view intelligence as innate and fixed, and fundamentally believes that not much can be done to increase intelligence. An incremental theorist fundamentally believes that challenging problems are a core part of the learning process and that intelligence is malleable: it can be gained through hard work. When confronted with challenges, entity theorists interpret them as limits of their abilities and do not try hard to solve them. In literature, an entity theory is

frequently referred to as a *fixed mindset* and an incremental theory as a *growth mindset*—owing largely to the underlying motivations of the entity and incremental theorists.

Though Dweck's work initially focused on theories of intelligence, these theories of self can be applied to domain-specific beliefs. For example, individuals can hold an incremental theory toward intelligence, while still having entity-theory views on specific skills such as programming or debugging,[12] so this research does not imply global views.

Individuals with an entity theory of intelligence are more likely to be motivated by appearance than performance, are more likely to engage in maladaptive behaviors (such as giving up or cheating) in the face of problems, and are less likely to collaborate with others. In the software development field, these behaviors manifest through maladaptive strategies including hero worship, impostor syndrome, "cargo culting," and lackadaisical work ethic.

Incremental theorists by contrast are motivated by performance-oriented goals. Whereas the entity theorist takes shortcuts to appear smart (such as tacking a large number of objectively simple tasks), the incremental theorist tackles hard problems head-on, seeing hard work as a necessity to gain knowledge and deliver value. Furthermore, incremental theorists tend to work effectively in groups, value helping educate and promote the success of others, and engage in other behaviors generally positive for professional programming environments.

This incremental theory of intelligence is clearly beneficial to programmers and characteristic of the best in the field, whether professionally or academically oriented. It would be useful to figure out how to apply Dweck's research effectively, recognizing that the practice of programming (like many other fields) gains from lifelong learning. Educators, managers, and mentors would benefit from incorporating this research into courseware, management, and teaching strategies.

The key to applying Dweck's research to our field is that it is possible to "shift" on the spectrum of self-theories. By changing the way you react to, respond to, and praise folks, you can help move others (and yourself) from a "fixed" to a "growth" view.

## Self-Theories in Computer Science

How capable are you at programming compared with when you started? At debugging? Whether you've been at it for one year or 50, you are almost certainly solving problems that seemed intractable when you began your learning journey. This isn't a result of innate ability: before we started programming, none of us had this ability. We don't have to look very far back in history to understand that ability in this area is nascent among humanity.

Still, we must carefully consider psychological research before applying it to our work[13] and decide if it is relevant. In a 2008 paper, Laurie Murphy and Lynda Thomas enumerate ways in which embracing such research can help in the field of computer science:[9]

• Applying this research to introductory computer science courses may improve student learning and course retention rate.

• The gender gap in computer science education and practice may be directly attributable to cultural influences on self-theories of intelligence in women.

• Collaborative work, such as pair programming, is found in some studies to be wildly effective—and wildly ineffective in others. Conflicting goals of entity versus incremental theorists could explain these apparent contradictions.

• Defensive classroom climates, characterized by students who ask "pseudo-questions" to demonstrate knowledge and professors who afford special status to those students, could be explained and ameliorated by adopting Dweck's work.

As bugs present themselves as problems, and because problems tend to be perceived by the entity theorist as fundamental limits on ability, the focus here should be on moving students, peers, and co-workers toward a more malleable view of intelligence.

## Organizational Adoption of Malleable Self-Theories

Whether operating as a manager, mentor, or educator, adopting and promoting malleable self-theories is important in creating successful students and co-workers. Debugging must not be an afterthought in educating; industry must stop insisting that bugs be interpreted as failures of individual programmers (especially since individual programmers are rarely responsible for the design and functioning of an entire system). Programmers should instead be praised for their efforts in solving bugs. In all cases, solving bugs is part of the learning process. How can they be presented as such?

Several studies have attempted to move individuals' views from a fixed to a malleable perspective. A familiarity with the literature is important here, especially for educators and managers. In a 2004 paper, Mantz Yorke and Peter Knight suggest that educators should "(1) appreciate the significance of self-theories for student learning; (2) be able to infer whether students are inclined toward fixedness or malleability; and (3) possess strategies for encouraging ‘fixed' students to move toward malleability."[14]

To do this, Murphy and Thomas suggest looking at psychologist Lev Vygotsky's work from the early 20th century. They state, "students learn best when pushed slightly beyond their independent capabilities." This requires an instructor or mentor capable of assisting students past their current ability; Yorke and Knight observe that this system works best when both the teacher and student possess a malleable view of intelligence.

Moving an individual from an entity to an incremental view can be as simple as framing information in a particular way. In multiple studies, Dweck notes that information presented as praise of *ability* promotes formation of an entity theory, whereas praise for *effort* promotes an incremental theory. This appears to be true whether or not the individual receiving the information is the individual being praised.

Dweck states that the temporal effect of praise-based shifts is unclear. In one set of individuals in their study, Cutts et al. consistently reinforced an incremental mindset every time feedback was

given on graded work.[3] Only students receiving this intervention (coupled with two other intervention methods) saw movement toward incremental self-theories *and* statistically significant improvement in test scores (a feat that no other study I have found has been able to show). It may be that consistent environmental feedback promoting an incremental theory is sufficient for an individual's long-term adoption. Given that it is shown to work in short-term situations, consistent feedback may simply be a functional equivalent.

Most examples of such feedback fall into framing information in a way that promotes growth-oriented goals. When a colleague solves a particularly nasty bug, people tend to say, "You're brilliant!" Instead they should say, "Great job on the hard work!" If you task engineers with work that is very easy for them to solve, you can be apologetic: "I'm sorry for assigning you a task you couldn't learn much from." Try to direct more challenging work to those colleagues in the future. Students and individuals will face frustration when solving bugs. If you, your students, or your colleagues voice frustration, try to frame it in terms of what you or they can expect to learn upon completing the project.

Unfortunately, students and job candidates are generally not able to choose teachers or managers who hold such views. Because learning is best achieved when both students and teachers hold incremental theories,[14] it is crucial that organizations hire individuals possessing such views and train existing staff on the material.

Instructors and tutors possessing an entity theory are more likely to focus on helping only those they perceive to be the brightest, writing off the struggling students as lost causes. This is particularly unfortunate since in several studies, Dweck shows that underachieving students with a malleable mindset moving into harder problems (as happens in transitions from elementary school to middle school, or from high school to college) tend to outperform high-achieving students with an entity theory. Instructors with an entity theory are less likely to help the underperforming students succeed.

## Reinforcing Incremental Theories for Individuals

Education is a lifelong process. As individuals, we should adopt malleable views of intelligence in daily life. There are specific ways of approaching problems that develop and reinforce a malleable view of intelligence.

### Active Recall

One way people may force themselves into entity-framework thinking is an overreliance on reference manuals and documentation. Are the arguments to `memcpy` in the order of source, destination, length; or are they destination, source, length? Is `strstr` haystack, needle; or is it needle, haystack? Many individuals develop a habit of consulting references (such as system manuals) as soon as the question comes up.

Active recall is a study method in which you first make a guess before looking up the solution. (This is the basis upon which study tools like flash cards are built, but it must be employed properly to be effective.) In the case of interfaces such as `memcpy` and `strstr`, write the code first:

take your best guess at the argument ordering. Once you've done this, look at the reference manual to confirm whether you've done it correctly.

### Segmented Study

Think of the last time you spent an entire day on a marathon debugging session. Did you solve the problem? Or did you need to take a break and work on something else for a bit, perhaps going home, sleeping, then solving the problem the next day?

Engineers frequently engage in sunk-cost fallacies while debugging, expending additional effort solving a problem despite diminishing returns for the time spent. Our brains are not tuned to focus on specific tasks for hours on end. Segmented study is the idea of having one or two additional and unrelated tasks to switch to over the course of an activity. Switching gears and taking breaks are (perhaps counterintuitively) effective methods for making progress when you're stuck.

For managers, mentors, and educators, this is also true. Explicitly allow colleagues and students time to work on other problems. Students and employees will not usually ask to work on something else, so it is crucial to grant folks the time they need to process information.

### Persevere

What *segmented study* doesn't mean is giving up when things get difficult. You must persevere: tenacity and passion are at least as important as intelligence to both success and skill development. The most successful individuals in a field practice their craft for years at the edge of their ability. This "grit" is highly correlated with individual success factors.[4]

### Be Curious

With sufficient experience, it becomes appropriate to mentor others. When teaching and mentoring others, it is vital to be curious. Many paths exist to arrive at some knowledge, and your own intrinsic views aren't always the best for everybody. When learning, people want to relate new knowledge to existing knowledge; this gives them a strong foundation upon which to build. Curiosity is about being open to ideas, solutions, and methods of thinking that do not necessarily reflect your own views. You must teach from the perspective of those you wish to educate.

## A General Approach to Debugging

Through continued learning, malleable views of problems, and effective use of tools, you can become successful in debugging. Still, some insist that debugging is more of an art than a science. I think we can dispatch this idea entirely. It is clear that debugging requires learning, and the scientific method is specifically designed to yield new knowledge. The method, summarized: (1) Develop a general theory of the problem. (2) Ask questions leading to a hypothesis. (3) Form a hypothesis. (4) Gather and test data against the hypothesis. (5) Repeat.

In my experience, very little attention is paid to the formation of a hypothesis, resulting in a wasted effort, testing without any theory pertaining to the cause of the bug. Forming a good hypothesis is rather more difficult than it would seem. In practice, hypotheses are poorly formed, and many rely more on intuition than information gathering. Intuition can be an effective strategy for debugging but requires extensive experience and, when used as the only strategy, leaves the programmer unprepared to handle new, unfamiliar bugs. Lacking a framework for solving these bugs is particularly damaging for entity theorists.

A good hypothesis describes a problem and is both testable and falsifiable. In fact, forming a proper hypothesis almost always implies that a bug is fully understood. Consider the following three statements:

• A bug exists in the logging module.

• A race condition exists in the logging module when concurrent log producers enqueue the same item.

• A race condition between concurrent log producers consuming pooled work objects is caused by an improperly handled return code when the pool is empty, and later results in a double-free when the consumer attempts to reenqueue the same object multiple times into a pool but fails because the pool is then full.

These statements were part of a thought process around solving a real bug. The first hypothesis is indicative of very little planning or research, and is the result of the "hunch" programmers have about what could constitute a bug. This is a testable hypothesis, but it is poor: if this hypothesis is confirmed through testing, the testing is unable to provide any more data on how to solve the problem.

The second statement is marginally better. It's clear that it is operating on more information, and so it seems like the bug has been reproduced at this point. This hypothesis is still incomplete, because it does not make any predictions as to why concurrent log producers would produce the same item. Furthermore, though it sounds like it describes what the failure is (a race condition), this isn't actually the terminal flaw, as described in the third hypothesis.

This third hypothesis is clearly the best. It describes both why the bug happens and what the failure is. Importantly, it identifies that the cause of failure occurs separately from where and when the program actually fails. This hypothesis is great because it can be very specifically tested. If regression tests are part of your development framework, only this hypothesis provides a description of how such a test should behave.

Falsifiability is an important and crucial property of a real hypothesis. If a hypothesis cannot be proven false, any test will confirm it. This cannot possibly give you confidence that you understand the issue.

Forming a sound hypothesis is important for other reasons as well. Mental models can be used to intuit the causes of some bugs, but for the more difficult problems, relying on the mental model

to describe the problem is exactly the wrong thing to do: the mental model is incorrect, which is why the bug happened in the first place. Throwing away the mental model is crucial to forming a sound hypothesis.

This may be harder than it seems. For example, comments in code suspected to contain bugs may reinforce existing mental models. This may cause you to paper over buggy code, thinking it is obviously correct. Consider, for example:

```
/* Flush all log entries */
for (i = 0; i <= n_entries; i++) { flush_entry(&entry[i]); }
```

This code (maybe obviously) illustrates an example of an off-by-one error. The comment above it is correct but incomplete. This code *will* flush all entries. It will also flush one more. When debugging, treat comments as merely informative, not normative.

## Conclusion

Debugging is one of the most difficult aspects of applied computer science. Individuals' views and motivations in the area of problem solving are becoming better understood through far-reaching research conducted by Carol Dweck and others. This research provides a means to promote continued growth in students, colleagues, and yourself.

Debugging is a science, not an art. To that end, it should be embraced as such in institutions of higher learning. It is time for these institutions to introduce entire courses devoted to debugging. This need was suggested as far back as 1989.[10] In 2004, Ryan Chmiel and Michael C. Loui observed that "[t]he computing curriculum proposed by the Association for Computing Machinery and the IEEE Computer Society makes little reference to the importance of debugging."[2] This appears to still be true.

Learning solely through experience (suggested by Oman et al. as the primary way debugging skills are learned[10]) is frustrating and expensive. At a time when the software engineering industry is understaffed, it appears that individuals with certain self-theories resulting from social and cultural influences are left behind. Understanding Dweck's work and changing the way we approach education, mentorship, and individual study habits can have a profound long-term effect on the progress of the software development industry. While research into tools to ease the task of debugging continues to be important, we must also embrace and continue research asking and showing how better to help students, colleagues, and peers toward success in computer science.

### References

1. Britton, T., Jeng, L., Carver, G., Cheak, P., Katzenellenbogen, T. 2013. Reversible debugging software. Cambridge Judge Business School; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.444.9094&rep=rep1&type=pdf.

2. Chmiel, R., Loui, M. C. 2004. Debugging: from novice to expert. *SIGCSE Bulletin* 36(1): 17-21.

3. Cutts, Q., Cutts, E., Draper, S., O'Donnell, P., Saffrey, P. 2010. Manipulating mindset to positively influence introductory programming performance. Proceedings of the 41st ACM Technical Symposium on Computer Science Education: 431-435.

4. Duckworth, A. L., Peterson, C., Matthews, M. D., Kelly, D. R. 2007. Grit: perseverance and passion for long-term goals. *Journal of Personality and Social Psychology* 92(6): 1087-1101.

5. Dweck, C. 1999. *Self-theories: Their Role in Motivation, Personality, and Development.* Psychology Press.

6. Kernighan, B. W., Plauger, P. J. 1974. *The Elements of Programming Style.* McGraw-Hill.

7. Ko, A. J., Meyers, B. A. 2005. A framework and methodology for studying the causes of software errors in programming systems. *Journal of Visual Languages and Computing* 16(1-2): 41-84.

8. McCauley, R., Fitzgerald, S., Lewandowski, G., Murphy, L., Simon, B., Thomas, L., Zander, C. 2008. Debugging: a review of the literature from an educational perspective. *Computer Science Education* 18(2): 67-92.

9. Murphy, L., Thomas, L. 2008. Dangers of a fixed mindset: implications of self-theories research for computer science education. *SIGCSE Bulletin* 40(3): 271-275.

10. Oman, P. W., Cook, C. R., Nanja, M. 1989. Effects of programming experience in debugging semantic errors. *Journal of Systems and Software* 9(3): 197-207.

11. RTI. 2002. The economic impacts of inadequate infrastructure for software testing; http://www.nist.gov/director/planning/upload/report02-3.pdf.

12. Scott, M, Ghinea, G. 2014. On the domain-specificity of mindsets: the relationship between aptitude beliefs and programming practice. *IEEE Transactions on Education* 57(3): 169-174.

13. Winslow, L. 1996. Programming pedagogy—a psychological overview. *SIGCSE Bulletin* 28(3): 17-22.

14. Yorke, M., Knight, P. 2004. Self-theories: some implications for teaching and learning in higher education. *Studies in Higher Education* 29(1): 25-37.

**Devon H. O'Dell** is a tech lead at Fastly, where his primary focus includes mentorship of team members and the scalability, functionality, and stability of Fastly's core caching infrastructure. Prior to Fastly, O'Dell was lead software architect at Message Systems and contributed heavily across the Momentum high-performance messaging platform. His experience over the past 15 years ranges from web applications to embedded systems firmware (and most areas in between). These days, you can usually find him debugging and being confused about something.