Alex Groce (agroce@gmail.com), Oregon State University

George Polya's *How to Solve it: a New Aspect of Mathematical Method* is a math book, technically, not a computer science book at all. The word "computer" does not appear in the text, nor does "algorithm" (a word that magically transforms a math book into a computer science book). Nonetheless, it is a book of great interest to the software engineer (or, really anyone who meets the requirements of **the intelligent reader**). The blurb on the back of my edition (the newer one from Princeton with an introduction by John H. Conway) says it "will show anyone in any field how to think straight." I don't know if that's true, but given the strong resemblance between many software engineering tasks and mathematical problems (broadly conceived), it certainly can help software engineers think straight. For example, here's a passage where Polya discusses **diagnosis** of student problems, in the long, dictionary-format, core of the book (the phrases in *italics* indicate references to other entries in what Polya calls "The Short Dictionary of **Heuristic**"; in this review I will use **bold** everywhere to indicate entries in Polya's dictionary):

"Some students rush into their calculations and constructions without any plan or general idea; others wait clumsily for some idea to come and cannot do anything that would accelerate its coming. In ***carrying out the plan*** the most frequent fault is carelessness, lack of patience in checking each step. Failure to ***check the result*** at all is very frequent. . ."

Does this sound at all like any (all-to-common) failure mode in software engineering, whether by "students" or by supposedly grown-and-graduated developers? Does it sound like ourselves on our worst days? Yes, and yes. Is advocating that you **check the result** similar to proposing the use of assertions? Yes, I say.

Polya's book focuses on mathematical problems, but at times dips into other areas (solving a crossword puzzle clue is one), in order to focus on its general topic: how to solve a problem, where you have some **condition**s, some given data, and need some resulting "unknown." This is what a great many pieces of code do: take some available data and produce something new. Polya often uses problems from geometric construction to illustrate his ideas, which at first glance seems quite remote from computer science (I think programmers are more likely to use statistics or combinatorics, and when we use geometry it is not construction with a straightedge and compass!).

However, once you fall into the state of mind that Polya's book requires (and creates), it becomes obvious that construction with straightedge and compass is a fine **analogy** for using a limited API to perform some task. My only tools are these five function calls, and I want to get from object A to object B, how do I do that? Building a software system, whether small or large, involves solving many *problems* of some sort; many of these problems, especially those at a level more detailed than large-scale architecture, are "puzzles" or "constructions" of just the kind Polya examines. It is likely that this explains the popularity of complex games and puzzles with programmers (many leading designers of modern board games are, in their "day job" software engineers, for example). Furthermore, the habits of mind relevant to solving puzzles are useful in even large-scale design, since a key feature of any architecture (or design pattern) is how well it serves in solving the puzzles of the domain!

How to Solve It is divided, like Gaul, into three primary parts (there is also a fourth part, consisting of problems, hints, and solutions). The first part of the book is titled "In the Classroom" and talks about how to help students, based on Polya's long experience as one of the great mathematics teachers of the last century (he taught courses at Stanford until the age of ninety). This section's interest is not limited to those who teach classes. First, "the student" here may be the reader, who can use this section to slip into the habit of mind Polya's book requires, and the way of thinking about problems that is needed. Second, almost all software engineers teach, and teach often. We teach new members of a team about the project at hand, we teach junior software engineers how to become senior engineers, we teach random strangers on *stackoverflow.com* how to solve their little problems, and we hope to teach ourselves how to avoid bugs we have recently introduced into our code or our design.

The second part of the book is a dialogue titled "How to Solve It." The dialogue is only about three pages, but is an excellent gateway from the book's first section to the core of *How to Solve It*, the dictionary of heuristic. Some dictionary entries are questions, but the dialogue is the primary source of the *questions* Polya proposes as central to problem solving. They are: *where should I start, what can I do, what could I perceive, how can an idea be helpful, what can I do with an incomplete idea,* and *what can I gain by doing so?*

Finally, the majority of the book (pages 37-232 in my edition) consists of the aforementioned "Short Dictionary of **Heuristic**." This section is an alphabetical sequence of entries on various "heuristic" ideas that are helpful in solving problems in general. The entries include the ones I have highlighted throughout this column, and additional gems such as: **Bolzano, contradictory, decomposing and recombining, definition, draw a figure, generalization, have you seen it before?, inventor's paradox, Leibniz, look at the unknown, notation, progress and achievement, signs of progress, specialization, symmetry, test by dimension, variation of the problem, wisdom of proverbs,** and **working backwards.** There is no easy way to summarize these entries, which range from short suggestions to lengthy and worked out analysis of how to use some aspect of a problem in finding its solution. The names themselves should be suggestive of the connections between problems as understood by Polya and the problems of software engineering.

The structure of *How to Solve It* should remind software engineers of another book: the dictionary resembles the list of patterns in *Design Patterns*. The key difference is that the patterns are often consulted in isolation, by the reader actively seeking a solution to some particular and probably well-understood problem. Design patterns are more formalized **heuristic** methods for solving problems in (object-oriented) design, while Polya's dictionary is probably best read either in sequence to learn about problem solving in general or by randomly dipping into its pages to inspire the flailing brain faced with a dread problem. This book is for the software engineer in need of a **bright idea** or currently lacking the requisite trinity of **determination, hope, success**. This book prepares us to solve problems, which is, after all, most of what we do.