Beauty in Software Engineering

Jon G. Hall and Lucia Rapanotti
The Open University

Is there, like Euler's identity, a simply expressed relationship, dense in meaning, between software engineering's most basic entities?

Euler's identity ($e^{i\pi}+1 = 0$) relates, in just seven symbols, some of the most fundamental entities and operations in mathematics. It has been described as the most beautiful theorem in mathematics and the greatest equation ever.

In this article we relate some of software engineering's most important concepts, including patterns and formality, functional and non-functional requirements, and validation and verification. Whether or not the result is beautiful is, as they say, in the eye of the beholder.

MR. EULER, MEET MR. ROGERS

G.F.C Rogers, writing as recently as 1983, defined engineering as "the practice of organizing the design and construction of any artifice which transforms the physical world around us to meet some recognized need" (*The Nature of Engineering: A Philosophy of Technology*, Macmillan, 1983).

Part of the utility and beauty of Rogers's definition is its applicability in many areas: with judicious choice of the right type of artifice—a plane, a ship, or a bridge, for instance—the definition can be contextualized to aeronautical, nautical, or civil engineering. When the artifice is a financial instrument, a business, or a course, the definition covers financial, organizational, and educational engineering. Critically, for us, when the artifice is software or an information system, we have a definition that covers software engineering.

Even if Rogers's definition needs no formalization, our intention here is to represent its components formally, to relate them to some important concepts in software engineering and capture certain relationships. In this way, we will build relationships as Euler did.

The language we choose to define our relationships is of our own invention; it's called problem-oriented engineering (POE; "Software Engineering as the Design Theoretic Transformation of Software Problems," *Innovations in Systems and Software Eng.*, vol. 8, no. 3, 2012, pp. 175-193).

TUTORIAL: PROBLEM-ORIENTED ENGINEERING

Suppose we let W represent Rogers's real-world environment, S his artifice, N his recognized need, and G the stakeholder whose need was recognized. An engineering problem—more simply, problem—is the proposition P_G:

$$P_G : W(S) \textbf{ meets}_G\, N, \qquad (1)$$

whose truth value indicates that when S is installed in environment W, their combination meets the need N to the satisfaction of stakeholder G.

From our representation of problems as propositions, we can build representations of problem-solving processes as relationships between propositions that represent problems. In this way, we can arrange that a problem is solved with respect to the stakeholder if and only if the proposition is true. We won't go into too much detail here, but as an example, if both Michael and Gordon are stakeholders in the same problem P, with Michael more critical than Gordon, then we can say

$$P_{Michael} \Rightarrow P_{Gordon}$$

to capture the fact that if Michael says a problem is solved, then Gordon will agree.

We can take such implications and make them problem-solving steps, writing, for instance,

$$\frac{P_{Michael}}{P_{Gordon}} \textit{Michael} \text{ stricter than } \textit{Gordon}$$

to mean that "as Michael is stricter than Gordon, to solve problem *P* for Gordon, it is sufficient to solve it for Michael," which relates two problems via its justification. POE is designed to represent the relationship between problems via justifications.

In software engineering, the strictest Michael can be is to demand a formal proof that a solution satisfies the need, and this will satisfy all (rational!) stakeholders. But such proofs are difficult to come by for real-world systems. In POE, requiring proof isn't our intention: Gordon might, for instance, be willing to accept rigorous testing as assurance of some software's adequacy against its spec, thus allowing an easier or cheaper, yet still fit-for-purpose, software solution.

WHAT SORT OF SOFTWARE NEEDS ARE THERE?

Within software engineering, recognized needs come in two forms: functional and nonfunctional. Functional needs express relationships between mathematical inputs and outputs. One such is "should add input base to input offset to produce output address," the solution for which is probably just a single line of machine code.

You might think a single line of machine code is trivial to get right, but it's not. The simplest code—the line of machine code that performs address = base + offset—is subject to failure; even proven-correct code will fail because the real world is a very aggressive place.

Consider, for example, Google, with its massive server farms. For Google, reliable software is mission-critical. Bianca Schroeder and colleagues describe the effects of software failures on Google's operations: "Typically a machine shutdown is caused" by "a single uncorrectable error [being] considered serious enough to replace the dual in-line memory module (DIMM) that caused it" (B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM Errors in the Wild: A Large-Scale Field Study," *Proc. 11th Int'l Joint Conf. Measurement and Modeling of Computer Systems*, ACM, 2009, pp. 193-204). For some Google platforms, the uncorrectable failure rate currently is up to 4 percent per year, leading to large (but obviously sustainable) costs. The problem isn't poor software, it's the effect of the software's environment: one bit flipped by a cosmic ray, causing one bad machine code calculation, for example.

The get-out-of-fail-free clauses are the nonfunctional requirements that characterize the solution's behavior in situ.

That software must be embedded in the real world to function makes functional needs the Platonic ideas of recognized need. Thus, for a purely functional *F*, in a nontrivial context *W*, and no matter the stakeholder *G*, the problem

$$W(S) \textbf{ meets }_G F \tag{2}$$

has no solution. This is true no matter how much time and effort developers devote to creating the software: when it's installed in its environment, it will fail. Notwithstanding the achievement of formalists in codifying the machine and in producing techniques to reason about absolute correctness, in the real world, functional requirements are simply unsatisfiable.

Help is at hand in the form of "get-out-of-fail-free" clauses that explain the situations in which a solution's failure is acceptable. For Google's production environment, failures of up to 4 percent per year are tolerated. The get-out-of-fail-free clauses are the nonfunctional requirements, or qualities, that characterize the solution's behavior in situ. Qualities come in many forms—usability, learnability, and other -ilities—and are generally expressed as properties of populations of real-world behaviors.

Associating functional requirements with one (or more) qualities can cover the fallibility of embedding a system in the real world, thus making them implementable. So, even if Problem 2 has no solution, with a judicious choice of quality *Q*, the following problem might have one:

$$W(S) \textbf{ meets }_G Q(F), \tag{3}$$

that is, find *S* that, when installed in *W*, meets *F*, subject to *Q*, to the satisfaction of *G*.

This association of functional requirements with qualities results in sometimes solvable software engineering problems.

YOU'RE LATE, MR. ALEXANDER!

A single quality can mitigate failure for many functional requirements in numerous contexts; that is, for particular *W* and *Q*, Problem 3 is a recurring problem.

This brings us to Christopher Alexander and colleagues' ideas on patterns: "A pattern is a careful description of a perennial solution to a recurring problem within a building, describing one of the configurations which brings life to that building" (C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*, vol. 2, Oxford Univ. Press, 1977). Of course, Alexander was an architect, so his recurrent problems resided in buildings. However, software engineering professionals realized as early as the late 1980s that these concepts applied to software, too, as design, architectural, and other patterns.

For Google, correctable error rates run many times higher than 4 percent: Schroeder and colleagues stated that "an average DIMM experiences nearly 4,000 correctable errors per year." Google's "recurring reliability problem," as we might term it, is solved by memory chips that employ error-correcting codes (ECCs) that can negate the bad effects of these errors, thus reducing failure to Google's 4 percent.

As for ECCs, in other contexts engineers look for patterned solutions for other qualities. Suppose, in context W and with quality Q, we've been successful in finding one solution, let's call it A_Q, that provides a buffer against W's real-world aggression. As a problem, this is stated as follows:

$$W(A_Q(S))\ \mathbf{meets}_G\ Q(F). \qquad (4)$$

Of course, no matter how good A_Q is, another context might be too aggressive even for A_Q. Hold that thought!

AND THEN THERE WERE FOUR

In his 1986 paper, "And No Philosophers' Stone, Either," Wladyslaw M. Turski observed that "there are two fundamental difficulties involved in dealing with nonformal domains (also known as 'the real world'): 1. Properties they enjoy are not necessarily expressible in any single linguistic system; 2. The notion of mathematical (logical) proof does not apply to them" (*Information Processing*, vol. 86, 1986, pp 1077-1080).

Because the real world is aggressive and not amenable to formalization, Turski advised careful validation of the properties of any solution. Don Knuth's comment to Peter van Emde Boas springs to mind: "Beware of bugs in the above code; I have only proved it correct, not tried it" (http://www-cs-faculty.stanford.edu/~knuth/faq.html).

To complete the picture, then, we should, for each stakeholder G, design a validation scheme, VG, that varies with F, Q, and W, whose role is, in Turski's words, to "isolate the experimental steps from formal ones":

$$\frac{S\ \mathbf{meets}_G\ F}{W\ (A_Q\ (S))\ \mathbf{meets}_G\ Q(F)}\, V_G \qquad (5)$$

or, using an appropriate validation scheme, reduce a situated engineering problem in an aggressive environment through the application of well-chosen qualities and associated patterns to that of developing software S to satisfy a simple functional specification F.

Euler managed to relate in just seven symbols some of the most fundamental entities and operations in mathematics. Equation 5 uses nine symbols to characterize some of the most important relationships in software engineering.

We return to Turski, who stated, "the realization that an experimental validation (testing) is a necessary step in application software construction, and that at the same time it is the least conclusive and most influential stage of the process, should lead to developments of two kinds. We should see a development of software design paradigms which isolate the experimental steps from formal ones, and we should see an emergence of sound principles of software validation."

Perhaps Turski hoped that there would be a single theory to bind the developments together; then again, perhaps not. Even so, perhaps the relationship we have forged in POE between functional and nonfunctional requirements, between experimental and formal, contributes something to the investigation of such a software design paradigm.

Because POE has no a priori dependence on the nature of software, we hope that software engineering can, by expansion of the language of the context, the need, and the solution, be melded more closely with its sister engineering disciplines. **C**

Acknowledgments

Thanks to Chris Tiernan, Managing Partner at Grosvenor Consultancy Services, for his insightful comments on earlier versions and to Michael Jackson for stimulating discussions.

Jon G. Hall *is a senior lecturer and researcher in the Computing Department at the Open University. Contact him at jon.hall@open.ac.uk.*

Lucia Rapanotti *is senior lecturer and researcher in the Computing Department at the Open University. Contact her at lucia.rapanotti@open.ac.uk.*

Editor: Mike Hinchey, Lero—the Irish Software Engineering Research Centre; mike.hinchey@lero.ie