Editor: **Neil Maiden**
City University London,
n.a.m.maiden@city.ac.uk

Listen, Then Use EARS

Alistair Mavin

WHY ARE REQUIREMENTS so hard to write? Shouldn't it be a simple thing to do? Setting aside the issues of requirements analysis and management, even the expression of a single requirement is a two-stage process, and neither stage is necessarily easy. First, you have to determine the need and then you have to find a clear way to express it. This column focuses on the second phase.

A wise man once said, "I didn't have time to write a short letter, so I wrote a long one instead" (attributed to Blaise Pascal in 1657, but also to Samuel Johnson, Abraham Lincoln, and Winston Churchill). Having read many requirements documents, I'm tempted to conclude that engineers think along similar lines. Perhaps a typical engineer might paraphrase Pascal as follows: "I didn't have the time or inclination to understand the real requirement, so I wrote down everything I knew about the solution instead."

Solution Mode

Engineers are problem solvers. Their natures, their personality types, and their training incline them to solve problems, not define them. They draw analogies between the situation at hand and previous problems and jump straight into solution mode. Engineers will think, "I have a solution to that," or "I can amend a previous solution to solve that." Their mantra seems to be "Let's cut metal!" or "Let's write code!" This worldview can often be seen as "getting on with the real engineering work." Indeed, if you label this approach as agile, then it's legitimized as a paradigm. However, many would argue that this is not good systems engineering.

Language Barrier

In my experience, software developers love to use the latest notation. However, customers, users, or stakeholders (call them what you like) will typically recoil in horror at any specialist jargon. So while developers want to use a rigorously defined leading-edge notation, the vast majority of requirements in the real world are written in natural language.

The requirements engineer's role is often to act as a translator (or dare I say *mediator*) between the stakeholder and the developer; but must it always be this way? Couldn't we find a way to express requirements in a language that both stakeholders and developers can understand? Perhaps the answer is to develop a new universal requirements notation. In reality, we don't need a new notation because a perfectly good one already exists; it's called the English language.

So how can you write clear and concise textual requirements? The English language is rich and can be applied creatively to great effect. This flexibility makes it an ideal medium (or *notation*) for expressing sentiments such as love, poetry, and humor. However, these same characteristics make it inherently unsuitable for the expression of deterministic, testable requirements.

Herein lies the dilemma: human beings want to write requirements in natural language, but it's too imprecise for the task. In my view, the solution is a gently constrained application of the English language. A lightweight notation that works well is the Easy Approach to Requirements Syntax (EARS), a simple framework based on the hypothesis that we can classify all textual requirements into five basic templates (or *classes*).[1,2]

0740-7459/12/$31.00 © 2012 IEEE

EARS Templates

The EARS notation provides the following classes.

Ubiquitous

A ubiquitous requirement is

- something that the system must always do,
- unconditional, and
- continuously active.

For example, "The Engine Control System software shall comply with DO-178B."

Event-driven

An event-driven requirement applies when

- the system response is initiated by a triggering event detected at the system boundary,
- the trigger must be something that the system itself can detect, and
- the requirement is denoted by the keyword "when."

For example, "When commanded by the aircraft, the Engine Control System shall dry crank the engine."

State-driven

A state-driven requirement is

- active while a particular state or states remain true,
- continuous as long as the state holds, and
- denoted by the keyword "while."

For example "While the aircraft is in flight and the engine is running, the Engine Control System shall maintain engine fuel flow above *x* lbs./sec."

Option

An option requirement

- applies when a system response is needed only in applications that include a particular feature,
- is used as a simple way to handle product or system variation, and
- is denoted by the keyword "where."

For example, "Where electronic components are used in the Engine Control System, they shall comply with DO-254."

Unwanted Behavior

An unwanted behavior requirement is

- the required system response to unwanted events (such as failures, disturbances, and any unexpected behavior of interacting systems or users),
- a variation of the event-driven requirement, and
- denoted by the keywords "if" and "then."

For example, "If the engine fails to start during a third ground start attempt, then the Engine Control System shall terminate the autostart sequence."

Combining Templates

We can combine the basic EARS templates to build more complex requirements. For instance, the same event detected at the system's boundary might trigger a different system response depending on the states that exist when the event is detected.

Consider this example: "While the aircraft is on-ground, when reverse thrust is commanded, the Engine Control System shall enable deployment of the thrust reverser" (thrust reverser deployment is actually commanded by the aircraft, not directly by the Engine Control System). The triggering event (reverse thrust command) must trigger the system response only while a particular state exists (the aircraft is on-ground). Were the aircraft in flight, the system must not enable deployment of the thrust reverser. This is an unwanted (and potentially catastrophic) event, handled by an unwanted behavior requirement.

We can also use the keywords "when," "while," and "where" within if-then requirements to handle unwanted behavior with more complex conditional clauses. For example, here's a requirement to handle inadvertent thrust-reverser commands during the in-flight state is "While the aircraft is in-flight, if reverse thrust is commanded, then the Engine Control System shall inhibit thrust reverser deployment." In this situation, the trigger (reverse thrust command) is unwanted while in the in-flight state and therefore the system response prevents the aircraft from entering an unsafe condition.

To quote Ian Alexander quoting an Italian peasant farmer, "Making wine is simple, but not easy." And so it is with requirements using the EARS templates. The result of applying an EARS template is a simple, clear requirement. However, it can take a lot of work to reach this concise statement of need. To write a simple statement, you must first understand what you want the system to do, which might be difficult. The simplicity of the EARS templates prevents engineers from hiding behind ambiguous statements of what the system must do.

As Hans Hoffman said, "The ability to simplify means to eliminate the unnecessary so that the necessary may speak."

References

1. A. Mavin et al., "Easy Approach to Requirements Syntax (EARS)," *Proc. 17th Int'l Requirements Eng. Conf.* (RE 09), IEEE CS, 2009, pp. 317–322.
2. A. Mavin and P. Wilkinson, "Big EARS (The Return of 'Easy Approach to Requirements Syntax')," *Proc. 18th Int'l Requirements Eng. Conf.* (RE 10), IEEE CS, 2010, pp. 277–282.

ALISTAIR MAVIN is a requirements specialist with Rolls-Royce. Contact him at alistair.mavin@rolls-royce.com.