ALMOST 15 YEARS ago, I gave a talk at a pacemaker manufacturer and was somewhat surprised to discover it using agile development. At the time, the idea of agility for safety-critical systems seemed oxymoronic. (For a brief definition of safety-critical systems, see the sidebar.) However, the company’s software engineers explained that they performed a rigorous, up-front hazard analysis from which they discovered safety goals and requirements and that they initiated the iterative sprint-based agile process only after specifying these requirements.

In contrast, at a more recent meeting with another medical-device manufacturer, its developers claimed not to follow any specific safety practices. They trusted that safety problems would be identified and addressed as a natural byproduct of their agile Scrum process. Their slightly alarming, unmitigated faith in the agile process led me to contemplate ways in which safety activities could and should be addressed in agile development.

I’m clearly not the first person to think about this. The Open Platform for Evolutionary Certification of Safety-Critical Systems (OPENCOSS; www.opencoss-project.eu) and IBM’s report Adopting Agile Methods for Safety-Critical Systems Development\textsuperscript{1} provide detailed techniques for developing safety-critical systems using agile methods. However, I focus here on a lighter-weight approach for software systems at lower levels of safety criticality.

Consider a healthcare device for
monitoring the physical exercise of someone rehabilitating after a heart attack, or a system using unmanned aerial vehicles (UAVs) to deliver commercial goods to consumers. Both systems could cause harm if they malfunctioned. Their failure’s impact, although not catastrophic, would certainly not be negligible. The healthcare monitor might fail to warn its user when vital signs reach unacceptable limits; a UAV might lose control and fly into a crowd of people. Because companies are adopting agile processes to develop such systems, we need effective ways to integrate safety practices.

Safety processes typically include preliminary hazard analyses and safety assurance activities. Here, I look at both of these and how they fit into the Scrum cycle. In particular, I show how developers can use safety stories to improve safety processes. All my examples come from the Dronology system we’re developing at the University of Notre Dame. Dronology uses UAVs to support search-and-rescue missions.

Preliminary Hazard Analysis
The first step in any safety process is a preliminary hazard analysis that systematically identifies potential hazards and their contributing faults. For example, consider a hazard in which a UAV crashed because its battery failed without warning. There were three potential contributing faults:

- The battery level detector reported an incorrect level.
- The battery level monitor failed to detect a low level.
- The battery discharged faster than anticipated, allowing insufficient time for remedial action.

Hazard identification can be supported by brainstorming, checklists, and analyzing reports of previous failures. These activities should be conducted carefully and rigorously to produce a Fault Mode Effect Analysis (FMEA) listing the hazards and their contributing faults. Likelihood and impact ratings can be attached to each hazard. Then, safety analysts, developers, and other project stakeholders can decide which faults to address and how to mitigate them. For example, you could partially mitigate an incorrectly reported battery level by raising an alarm if the reported level falls outside the expected bounds. Such mitigations can be specified as safety requirements.

For Dronology, we adopted Alistair Mavin’s EARS (Easy Requirements Specification) format to specify safety requirements, which I’ll call safety stories from now on. EARS is indeed easy to use. It provides a simple format for specifying requirements for five types of behavior: ubiquitous, event driven, state driven, optional, and unwanted. Table 1 shows an example safety story for each of these types.

It’s important to understand and document any assumptions made on external systems. In our case, we expect the UAVs to behave as described in their specifications. For example, we assume that “The Model-X drone is able to transmit messages at distances up to 1,000 meters,” as specified by its manufacturer. So, we write the safety story, “The UAV must remain within 900 meters of a ground control station at all times.” This minimizes the likelihood of accidents associated with the lack of direct UAV control.

Safety Assurance
Once you’ve identified hazards and faults, specified safety stories, and built the system to realize those stories, you need to convince yourself and potential certifiers that the system is safe for use. Many techniques exist for doing this. However, the underlying premise is that you must construct a convincing argument that the hazard analysis has been performed systematically and thoroughly and that the system as built effectively mitigates all important faults.

One solution is to use an FMEA
in which each row represents a specific fault, each fault is linked to a safety story, and each safety story is linked to specific code files and associated tests and test logs. In other words, you establish traceability links from safety stories to any other artifacts that help satisfy those stories.3 These links don’t need to add significant overhead to the project because you can leverage typical development environments to capture them. For example, you can set up a JIRA–GitHub bridge so that as code is checked in, the commit message includes a tag referencing its associated safety story. However, particularly impactful hazards and faults might require additional analysis, perhaps throughout construction of a state transition model to carefully analyze and specify behavior. You can then use this model to provide evidence that a specific fault has been adequately mitigated.

Some engineers in the safety-critical domain use safety assurance cases to systematically construct arguments comprising claims, arguments, and evidence that the system is safe for use. I don’t have space to elaborate on these techniques here; the NASA System Safety Handbook provides a great introduction.4

### Fitting Safety into Scrum

Figure 1 illustrates how all this fits into a typical Scrum cycle. First, stakeholders express their needs in terms of user stories, which are placed into the product backlog. Then, a subset of qualified and informed stakeholders brainstorm potential safety hazards and their contributing faults, record them in an FMEA, specify safety stories that mitigate the faults, and add the stories to the backlog. At this point, the backlog contains a mix of traditional user stories and safety stories.

During each iteration’s planning stage, the set of faults can be continually reviewed and augmented. For example, if new features are combined in the product, new faults triggered by individual features or by their interactions with other features can be identified. This will lead to the creation of new mitigation strategies and safety stories.

At the start of each sprint, the customer can select a combination of user stories and safety stories for development. However, this means that the executable, working code produced at the end of each sprint might or might not be safe for use. Dependencies exist between user stories and safety stories, and some features (represented by user stories) might well be delivered before the associated safety stories are implemented and released. In these circumstances, the system might function under normal happy-day scenarios but could be unsafe when malfunctions occur. Stakeholders need to be gatekeepers, determining at the end of each sprint whether the system can be safely deployed. Ideally, safety stories associated with specific user stories will be released in tandem with those stories; however, for scheduling purposes, this won’t always be the case.

It’s widely acknowledged that the cost and effort invested in safety should be commensurate with the criticality of the system under development.5 As I mentioned before,

---

### Table 1

**Safety stories specified using the Easy Requirements Specification (EARS) format.**

<table>
<thead>
<tr>
<th>Behavior type</th>
<th>Basic safety story format</th>
<th>Example safety story</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ubiquitous</td>
<td>The <code>&lt;component name&gt;</code> shall <code>&lt;response&gt;</code></td>
<td>The drone shall maintain a minimum separation distance from other drones at all times.</td>
</tr>
<tr>
<td>Event driven</td>
<td>When <code>&lt;trigger&gt;</code>, the <code>&lt;system name&gt;</code> shall <code>&lt;system response&gt;</code></td>
<td>When the drone is within x centimeters of the minimum distance from another drone, the collision avoidance system shall provide directives to all drones in the vicinity.</td>
</tr>
<tr>
<td>State driven</td>
<td>While <code>&lt;in a specific state&gt;</code>, the <code>&lt;system name&gt;</code> shall <code>&lt;system response&gt;</code></td>
<td>While in landing mode, the drone shall descend vertically until it reaches the ground.</td>
</tr>
<tr>
<td>Optional</td>
<td>Where <code>&lt;feature is included&gt;</code>, the <code>&lt;system name&gt;</code> shall <code>&lt;system response&gt;</code></td>
<td>Where parachute mode is enabled and a drop is initiated, the drone shall scan the drop zone for obstacles.</td>
</tr>
<tr>
<td>Unwanted</td>
<td>If <code>&lt;optional preconditions&gt; &lt;trigger&gt;</code>, the <code>&lt;system name&gt;</code> shall <code>&lt;system response&gt;</code></td>
<td>If wind gusts exceed the desired velocity but are below the maximum velocity, the drone shall return to the base.</td>
</tr>
</tbody>
</table>
the process I described here targets software systems that have safety implications at lower levels of the criticality spectrum. Many projects that might not be traditionally classified as safety critical actually have safety implications. Integrating analysis into the Scrum process provides a lightweight, yet robust approach for exploring and mitigating such safety concerns.

This is my last Requirements column for IEEE Software. I’ve enjoyed being a columnist over the past several years and the opportunity it has given me to think about requirements practices in a rapidly evolving IT environment. I’m thrilled to hand the reins over to Sarah Gregory from Intel.

Sarah has been an active participant in the requirements-engineering research community over the past decade and is a popular teacher and trainer and an experienced practitioner. I look forward to learning from her in what I’m sure will be exciting, insightful future columns.

References

JANE CLELAND-HUANG is a professor of software engineering at the University of Notre Dame. Contact her at janeclelandhuang@nd.edu.

FIGURE 1. Safety activities injected into the Scrum cycle. This process targets software systems at lower levels of safety criticality.