A common challenge with writing user stories is how to handle a product's non-functional requirements. These are requirements that are not about specific functionality (“As a user of a word processor, I want to insert a table into my document.”), but are rather about an attribute or characteristic of the system. Examples include reliability, availability, portability, scalability, usability, maintainability. As you can see from that list, non-functional requirements are often referred to as “-ilities.” Of course, not all non-functional requirements end in “-ility.” We also have security, performance, robustness and so on.

Thinking back into the dark ages, I can remember when I first read about “non-functional requirements.” The term threw me for a loop. If it's non-functional, why do I care about it? I'm sure the author of that book clarified this for me a page later, but the term has always seemed an odd one to me. I prefer to think of non-functional requirements as “constraints” we put on the system. When a product owner says, “this system must perform adequately with 100,000 concurrent users,” the product owner is putting a constraint on the development team.

The product owner is effectively saying, “Develop this software any way you'd like as long as you achieve 100,000 concurrent users.” Each constraint we put on a system narrows the design choices a bit; calling these “constraints” rather than “non-functional requirements” helps us remember this. Fortunately constraints/non-functional requirements can be easily handled as user stories. Here are a couple of examples:

- As a customer, I want to be able to run your product on all versions of Windows from Windows 95 on.
- As the CTO, I want the system to use our existing orders database rather than create a new one, so that we don't have one more database to maintain.
- As a user, I want the site to be available 99.999 percent of the time I try to access it, so that I don't get frustrated and find another site to use.
As someone who speaks a Latin-based language, I might want to run your software someday.

As a user, I want the driving directions to be the best 90 percent of the time, and reasonable 99 percent of the time.

As you can see from these, I was easily able to stick with the, “As a , I want , so that ” template, which I prefer for most user stories. I do this for a couple of reasons, but want to comment further on only one of them here. Consider the example of the CTO constraining the team to use the existing orders database. (This was the real situation; the team was considering a second orders database that would be synchronized at night. The CTO overheard this and said, “No!”)

If we wrote this requirement as simply: “Must use existing orders database,” it would be entirely possible that a few months down the road we wouldn't remember why we should be constrained in that way. We'd ask the product owner if she cared if we used a secondary database, and she'd say she had no objections. And we'd make a mistake of using one.

Embedding the person who wants something can be very useful. But, you should be careful not to get obsessed with that template. It's a thinking tool only. Trying to put a constraint into this template is a good exercise, as it helps make sure you understand who wants what and why. If you end up with a confusingly worded statement, drop the template. If you can’t find a way to word the constraint, just write the constraint in whatever way feels natural.
agile, you can also have Mike email you a short tip each week.