User Stories have become firmly embedded in agile practice. These days most Scrum implementations seem to take them for granted. It’s easy to see why. Requirements capture can be a tricky business, and many techniques and processes have evolved over the years to manage the problem. Some of these are expensive, heavyweight, and complex. User Stories are remarkable in their apparent simplicity and the lack of overhead involved in applying them. It seems that all you have to do is write a couple of structured sentences on an index card, and you’re away.

Well, that’s true up to a point. It should be remembered that User Stories are only as good as the thought that has gone into them. For that matter, most agile methods (including Scrum) don’t mention User Stories at all...they are quite agnostic about how requirements are expressed. What can be said is that Business Analysis doesn’t evaporate just because a team has been trained to express requirements in a certain format; rather, the skill has been removed from its silo and distributed across all team members. This is a gradual process that can take months or even years.

Here are some of the key issues that a team must address before they’ve really got User Stories cracked:

* Do all business requirements actually need to be written as User Stories?
* Should we be breaking stories up into tasks?
* What about defects? Are they User Stories?
* Should we be writing Acceptance Criteria for every story?
* What about Non Functional Requirements? Are they stories too?

We’ll explore each of these matters and get a head-start on applying User Stories in practice. But first, let’s remind ourselves of what a User Story actually is.

**Reminder: What is a User Story?**

A User Story is a requirement expressed in a particular format:

**As a <role>, I need/want <goal/desire> so that <benefit>**

The goals and benefits can be broken into clauses, each of which is joined by "and". Here’s an example:

**As a user closing the application, I want to be prompted to save anything that has changed since the last save so that I can preserve useful work and discard erroneous work.**

A common mistake is to construct a User Story as a tautology. For example, the above story might be miswritten in the following way: *“As a user closing the application, I want to be prompted to save anything that has changed since the last save so that anything since the last change is saved”*. Instead we should write it in such a way that the benefit is clearly separated from the goal.

A User Story should give a clear expression of the business value, including who expects it and why. Yet for all of its clarity, a good User Story will be very thin on details...and deliberately so. A User Story isn’t meant to be a specification. It’s a placeholder for a conversation between developers and the people who understand the business value of the requirement. As long as these folks are talking and collaborating on the delivery of the value expressed in the story, that’s all we need from the story itself. It’s serving its purpose. This is why User Story cards are placed on Scrum and Kanban boards. They then become an **information radiator** showing the progress of requirements from acknowledgment through to delivery, and can be the focus for any related discussion.

Years ago, Ron Jeffries summarized the lifecycle of a User Story in terms of *Card*, *Conversation*, and *Confirmation*. Here’s the lowdown on each of these “3 C’s”:

* **Card**. The story should be concise enough to write on a simple index card along with any annotations that may be required.
* **Conversation**. This is when the story gets to earn its keep on a Scrum or Kanban board. It is discussed and clarified, re-estimated if necessary, and perhaps even re-written. The benefit is an improved understanding at the point of development, and deliveries that are consistently fit for purpose.
* **Confirmation**. A shared understanding of the User Story is confirmed by the delivery of work products that meet the requirement.

The better written a User Story is, the more effectively the “3 C’s” will be facilitated. What really makes for a good story though? There has to be more to it than the use of good grammar and the avoidance of tautology. Well, there are indeed some additional considerations, and they are referred to as the *User Story INVEST criteria*:

* **Independent**. User Stories should not overlap in terms of the value they deliver.
* **Negotiable**. It should be possible to debate and change User Stories, and to trade them in and out of scope.
* **Valuable**. Every User Story must deliver stakeholder benefit.
* **Estimable**. It should be possible to anticipate how much effort a User Story will require for implementation.
* **Small**. It is better to work on multiple small pieces of work than a larger one, since progress is more easily ascertained and at least some value can potentially be delivered earlier.
* **Testable**. It should be possible to confirm the successful completion of a User Story by objective means.

Now, this last criterion - objective testability - raises an important question. How can objective tests be determined at all, given that the User Story is just a “conversation placeholder”? The answer lies in another technique that has become ensconced in agile practice...the formulation of *Acceptance Criteria*. Like User Stories, Acceptance Criteria are brief sentences that conform to a certain format. The difference is that they place the focus not on statements of value but on tangible outputs:

**Given <initial context>, when/where <event or condition>, then <expected outcome>**

There can be multiple Acceptance Criteria for each User Story, since many conditions may need to hold before the delivery of value can be confirmed. Here’s an example of one possible Acceptance Criterion for the User Story we saw earlier:

**Given that a running application has open work, when the application is closed and changes to the work have been made since the last save, then the user will be prompted to save the work.**

A common mistake made by teams is to write Acceptance Criteria that simply rephrase the User Story, and thereby lack focus on tangible outputs. Here’s an example of how the above criterion might have been miswritten: *“Given that a user is closing the application, when they are prompted to save anything that has changed since the last save then the system will preserve useful work and discard erroneous work”*. This is a restatement of value rather than a test of specific circumstance that would be required for objectivity. The recent surge in *Business Driven Development* has resulted in tools, such as SpecFlow and Cucumber, which encourage the formulation of objective and testable Acceptance Criteria.

Now we’re up to speed on the basics of User Stories and how they are meant to be written, let’s look at each of the key issues we identified earlier.

**Do all business requirements actually need to be written as User Stories?**

Remember that most agile methods, including Scrum, don’t make any mention of User Stories at all. But it’s fair to say that if you are going to use them to capture business requirements then you should do so consistently. If you cherry-pick certain ones for this treatment then it begs the question as to why others are being excluded. Perhaps it is thought “too difficult” to write some requirements as stories...perhaps the benefits are unclear, or the users receiving the value are proving a bit tricky to identify. Yet if that is the case, then surely that is all the more reason to try and formulate the requirements in such a way. We hardly need to spell out the potential waste in progressing requirements where such basics aren’t clear.

Having a policy that mandates User Stories for capturing business requirements is therefore recommended. But even with that policy, it doesn’t mean that every business requirement in a Product Backlog has to be written as a User Story. I know that sounds like a contradiction, so allow me to explain.

In an agile way of working such as Scrum, requirements are fleshed out in greater detail the closer they get to the point of being worked on. It’s a *Just In Time* approach to requirements elicitation. There’s no point in producing a fully-formed requirement until it is needed. If you produce fully-formed requirements early, you are merely increasing the risk of rework due to scope changes occurring in the intervening time. That’s waste. Instead, a requirement should capture *just enough* to satisfy the demands placed upon it at a given moment. This means that for as long as a requirement sits on the back burner in a Product Backlog, it doesn’t need to be written as a full User Story with Acceptance Criteria. It might just be a two or three word title summarizing what is needed. As long as there is enough information to allow it to be prioritized and given a tentative estimate, that’s fine.

However, the closer that requirement gets to leaving the peanut gallery and into Sprint Planning, the more detail will have to be added. Once it is in Sprint Planning a full-blown User Story and Acceptance Criteria will have to exist for it, so a better estimate can be provided and the “conversation” has a proper context in which it can take place. So the policy of “every business requirement must be written as a User Story” does apply, but that doesn’t - and shouldn’t - mean that every requirement has to be formulated that way from the moment it is conceived.

**Should we be breaking stories up into tasks?**

As we have seen, User Stories are deliberately short on detail. They don’t say anything about implementation and are placeholders for a future conversation. But eventually it will “get to the getting”, and one or more team members will have to transform that story into a work product that delivers actual value.

Development is complex stuff, and just “talking about things” may not be enough for team members to get their heads around what needs to be done. If that is the case then there can indeed be benefit in breaking a User Story up into component tasks. This process should start in Sprint Planning and continue throughout the Sprint. Task discovery is effectively part of the ongoing conversation.

Now, not all User Stories are so complex that task breakdowns are required. For example, a Lean Kanban team does not typically need any of its “tickets” - which can be written as User Stories - broken down any further. This is because a Lean Kanban typically deals in small and repeatable changes, and the context of each is usually quite clear. Scrum teams on the other hand often need task breakdowns, because the scope of project work tends to be more uncertain. Task boards are therefore more common in Scrum than in Kanban. A Kanban board will typically have tickets representing each discrete item, and with no further breakdown shown.

**What about defects? Are they User Stories?**

In my experience the decision of whether or not to capture defects as User Stories is fraught with politics.

For example, it can be argued that every defect that is worth fixing must impact business value in some way. Assuming that is the case, the logic goes, then it must surely be worthwhile to capture defects as User Stories. This is often the argument of old-hand developers who want to push lame or poorly written defect reports back to the raisers so they are encouraged to think matters through more clearly.

Not so, say the raisers. We just want you to fix something that is broken.

How to resolve this argument? Well, if every business requirement has been captured as a User Story before it is delivered, then every defect that is raised must match an existing story. In other words there must be at least one story representing business value that has been compromised by the bug. This observation can get you closer to a policy on how to capture defects. For example you might decide to raise a defect as a task against an existing User Story. If the story doesn’t exist, then the associated requirement must be elicited and placed on the backlog for future implementation.

**Should we be writing Acceptance Criteria for every story?**

Every User Story must be *confirmable* (the third of the “3 C’s”) and *testable* (as per the last item in the INVEST model). This implies that every User Story should indeed be complemented with sufficient, well-formed acceptance criteria to assert its quality.

However, we should also bear in mind that a requirement does not have to be fully formed until the point where such information is needed. You might therefore decide that it is acceptable to defer the writing of Acceptance Criteria until Sprint Planning or shortly beforehand, such as in a Product Backlog grooming session.

The elicitation of Acceptance Criteria can be an involved process involving research with multiple stakeholders, and as such the Product Owner should take a leading role in making sure that appropriate and timely information is provided.

**What about Non Functional Requirements? Are they stories too?**

NFR’s are requirements of underlying system quality. Response times, reliability, and scalability are some of those qualities. They can be thought of as being non-transactional in nature. They do not represent discrete functional behaviors that produce a valuable result, but they are nonetheless critical for system acceptance, and the risk associated with getting them wrong is usually very high.

Mike Cohn has pointed out that since NFR’s still yield business value, they can still be written up as User Stories. This is undoubtedly true. However we can take issue with the quality of NFR’s if they are captured in such a way. Also, we can take issue with certain commonly associated practices, as they may not represent the best transport mechanism for taking Non Functional Requirements through to implementation.

For example, it is normal for a User Story to be written on an index card and placed on a Scrum or Kanban board where work on it can be progressed. If an NFR is placed on such a card, the associated requirement can indeed be implemented the same way as any other story. Performance, uptime, and any number of other characteristics can be delivered.

The problem lies in the process of acceptance and completion. User Stories, after completion, can still be stomped on. If this happens to a Functional Requirement then a defect against it can be raised. But suppose an NFR is violated after completion? Which User Story is impeded then? That’s the problem. NFR’s are rarely independent. They cut across a whole swathe of stories...conceivably even the entire product. And even if you sort this out, the point is the stomping still happened in the first place...and most likely over some of the highest risk scope in the system.

What we need is to stop the evil genie from getting out of the bottle. One way is to place NFR’s under the governance of the Definition of Done, such that every User Story must satisfy the NFR before it can be accepted. Alternatively, you might decide to have “standing” Acceptance Criteria for certain types of story which are thought to place an NFR most at risk. Note that in either case, the NFR itself can still be written up as a User Story. There’s nothing wrong with the method of construction. Just don’t expect an NFR to behave the same way as other User Stories, and make sure you have an appropriate and consistent means of enforcing them in place.

**Conclusion**

User stories are simple in construction and potentially complex in application. In this article we’ve looked at some of the issues that can arise, and we’ve found a number of things to consider should you choose to use them. Above all, don’t be deceived into thinking that the apparent ease with which they can be written obviates the need for careful thought and the ongoing mitigation of risk.