Writing better Agile 'Stories'

The growth in popularity of agile methods of software development, such as XP, Scrum and Kanban, has resulted in user stories becoming a popular method for capturing requirements.

Using requirements development good practice for writing of User Story Cards provides developers and product owners with a simple means to improve the understandability of each story. Increasing the understandability will reduce re-work which will concomitantly increase productivity and velocity and reduce defect density, etc.

A well-constructed user story describes a feature that represents value in the eyes of the customer. Requirements at different levels of abstraction can be expressed as user stories (one of my clients has defined 5 distinct levels of abstraction, from customer requirements down through product and system requirements to sub-system and individual software component requirements). At the lowest useful level, each story is intended to be the smallest unit of scope that delivers value.

A user story is said to be 'a promise for a conversation' – it represents the developing understanding of needs as customer and development staff work together. It combines both functional and non-functional requirements. Associated with each story are acceptance criteria that define when it is 'done'.

Stories focus on the who, what and why of a feature, not how it is implemented. They often are written on 5′ x 3′ (127 x 76mm) index cards, to limit verbosity and to provide a convenient medium. This allows stories to be considered individually, sorted, displayed on a notice board, etc.

The structure of each story is often based on a template as follows:

> As a … [stakeholder role] …
> I want to … [perform an action / record some information] …
> [With some frequency and/or quality characteristic] … So that
> … [description of value or benefit is achieved].

For example:

> As a library_user, I want to search for books by title, with speed and ease-of-use, so that I can find all books with similar titles.

Operational definitions for the 'fuzzy' terms 'speed' and 'ease-of-use' can also be written on the card, to capture the non-functional requirements.

Unfortunately this template can be a major contributing factor to the known limitations[1] of the agile method which include:-

- Without the accompanying acceptance tests, they are open to interpretation which makes them difficult to use as the basis for agreement
- They require close customer contact throughout the project which in some cases may be difficult or may be unnecessary overhead
- Can have difficulty scaling to large projects.
- Rely more on competent developers
- User stories are regarded as conversation starters. Unfortunately they may not capture where the conversation ended and fail to serve as a form of reliable documentation of the system.

Many of these limitations are shared by other development methods and so are not limitations of the Agile development paradigm per say, rather they are limitations brought about by the lack of adherence to the known good practices for requirements expression.

Effective natural language requirements[2] generally consist of four basic structural elements:- entities, actions, events, and conditions.

These elements can be used or modified by various cases such as the following:

• Owner.	• Actor.	• Target.	• Constraint.
• Owned.	• Action.	• Object.	• Trigger.

[1] Taken from Wikipaedia
[2] "Writing Effective Natural Language Requirements Specifications" by William M. Wilson

Using these concepts we can re-state the user story template like this:

As a [Actor or Owner – who/what does the action]
I shall [Action – what happens e.g. store, update, send data]
for; [Object – what is acted upon]
on the [Target or Owned – where the output is sent ; recipient or end state]
with [Performance - frequency and/or quality characteristic]
when [Trigger – causes of action; data receipt/user interaction]
unless / even if [Constraint – business rule or limiting factor]
So that [Rationale - description of value or benefit is achieved].

In the example, 'library_user' is the actor, 'search' is an action, 'books in the catalogue' are the object, the 'computer screen' is the target, the performance requirements are 'with speed and ease of use', the 'provision of the book title' is the trigger, 'incompleteness' is the constraint / qualifier and 'finding books with similar titles' is the value achieved. The user story becomes:-

> As a library_user I want to search for books in the library catalogue
> on the computer screen with speed and ease-of-use
> when I provide a book title even if the book title is incomplete
> so that I can find all books with similar titles.

The order:

<ACTOR><ACTION><OBJECT><TARGET><PERFORMANCE>

<TRIGGER><CONSTRAINT><RATIONALE>

is recommended but there are other orders that are popular, for example Use Case descriptions often use the order:

<TRIGGER><ACTOR><ACTION><OBJECT><TARGET>
<CONSTRAINT>

> I provide a book title and begin a search for books in the library catalogue on the computer screen even if the book title is incomplete.

Many user stories are more complicated than this example and no one single order can suffice but without including these elements the possibility for misunderstanding is significantly increased.

Another benefit of this template is that it enables an estimate of functional size to be made. Please see the corresponding white paper by Grant Rule on Sizing User Stories with COSMIC.

Standardized structured requirement expression enables comparisons across time, across teams, across projects, and across organisations. Developers, product owners, and their customers thus have a common way of communicating so that end user value can be delivered in an effective way.

Simon Wright – SMS Community of Practice Member

Managing Director

SymTech Ltd

April 2010

s.wright@measuresw.com