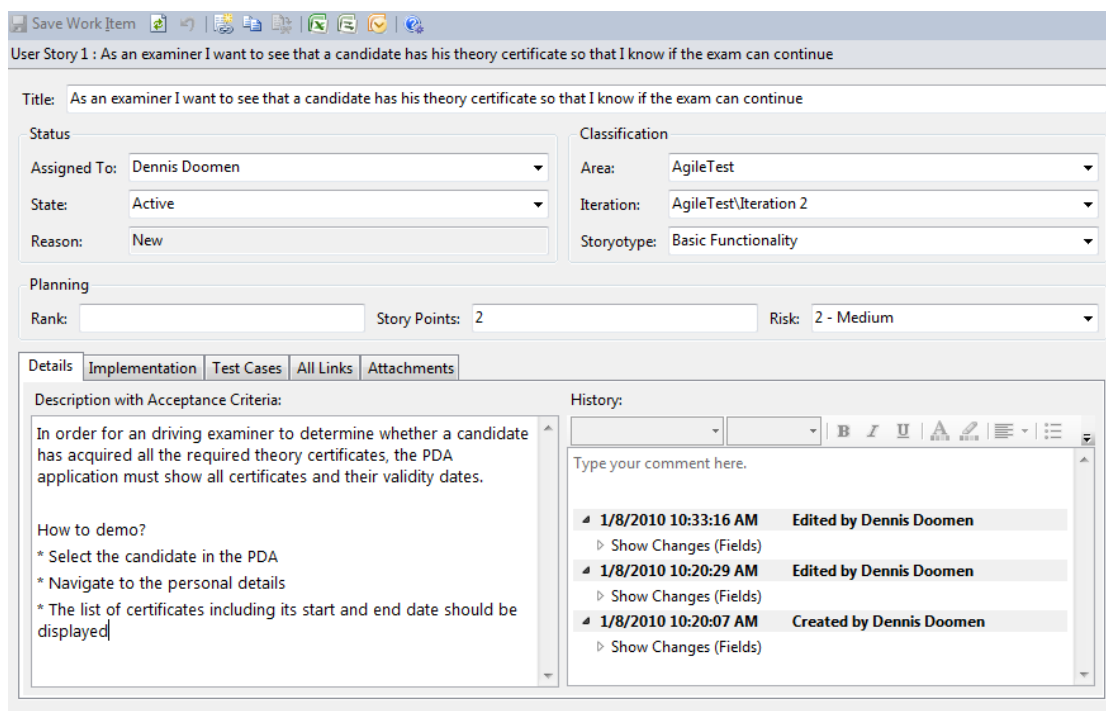


User Stories...The Story So Far

Let me start with a definition. The ideal user story represents an independent, estimable and prioritizable functional or non-functional requirement, is written in the language of the business and adds value to the project or system. That's it? Well no, it's not that easy unfortunately, but I promise you will get a whole lot closer after reading this article (or run away screaming with more questions than you started with).

Why should I read it?

With the introduction of Team Foundation Server 2010, Microsoft showed that they are finally embracing the best practices of Scrum and Extreme Programming. The MSF Agile 5.0 process template contains a dedicated User Story work item type and includes reports and Excel workbooks that keeps you in control of your entire development process. If you ask me, that's a great opportunity for delving deep into the depths of an agile user story.



The screenshot shows a 'User Story 1' work item in Team Foundation Server 2010. The title is 'As an examiner I want to see that a candidate has his theory certificate so that I know if the exam can continue'. The 'Assigned To' is Dennis Doomen, 'State' is Active, and 'Reason' is New. The 'Classification' section shows 'Area: AgileTest', 'Iteration: AgileTest\Iteration 2', and 'Storytype: Basic Functionality'. The 'Planning' section shows 'Rank' as empty, 'Story Points' as 2, and 'Risk' as 2 - Medium. The 'Details' tab is selected, showing a description with acceptance criteria: 'In order for an driving examiner to determine whether a candidate has acquired all the required theory certificates, the PDA application must show all certificates and their validity dates.' Below this, it says 'How to demo?' followed by three bullet points: '* Select the candidate in the PDA', '* Navigate to the personal details', and '* The list of certificates including its start and end date should be displayed'. The 'History' section shows three entries: '1/8/2010 10:33:16 AM Edited by Dennis Doomen', '1/8/2010 10:20:29 AM Edited by Dennis Doomen', and '1/8/2010 10:20:07 AM Created by Dennis Doomen'. Each entry has a 'Show Changes (Fields)' link.

Is it a Use Case?

You might think that a UML use case or a task from a more traditional work-breakdown also meets the above definition. But nothing could be further from the truth. Alistair Cockburn, an authority on UML and the Rational Unified Process (RUP), has documented five levels at which use cases can be scoped. For instance, RUP usually involves writing use cases at *sea level* that describe (a part of) a business process which, by definition, covers a large part of a system. Ordina and Cap Gemini often work with use cases at *fish level* where a use case typically represents a screen or a part of it. Sea level use cases have the disadvantage that they are too large to use as a planning unit, while fish level use cases can provide only part of the desired functionality. I have a lot of experience with fish level use cases, and although they are very useful for obtaining a rough size estimate in Use Case Points (a method coined by Gustav Karner), they are often still too large to serve as a unit of work in an Agile project.

Then, what is it?

I think that one of the essential differences between stories and other requirements management techniques is the way in which large chunks of functionality are broken down into small bite-sized and estimable pieces. Imagine for example that you want to introduce the concept of a product to an E-commerce website. In many projects, building this may require breaking it down in a number of tasks or use cases. The administrator might need a screen for managing products, whereas the potential buyer might need a screen for searching for products based on certain criteria.

Every project has to prioritize its work. In a use case-oriented project it is quite possible that the screen for searching is built much later than the management screen. And because the team realizes use case by use case, and consequently, screen by screen, chances are that the team will first finish the management screen including all the bells and whistles requested by the client. The big downside is that the desired functionality is not finished until both screens have been fully built. In an Agile project it is quite possible that due to changes in priority or scope the 2nd screen is never build.

User stories try to deal with that by requiring that the above mentioned functionality only has business value if the entire process of product managing and searching works. Obviously you run the risk that this results in a big chunk work that is difficult to plan or tack. That's why you need to explicitly distinct between the core functionality needed to allow the end-user to accomplish his task, and all the extras that make it easier to use.

For instance, the first story might only involve a simple entry form without any UI-specific considerations, validation or a fancy selection list, and a screen with a list of products from which the buyer can select one. Adapting the layout to the corporate standards, the ability to categorize products, and advanced validation should all be realized as separate user stories. And yes, I know, those stories wouldn't be completely independent from the original story. But at least they force you to focus on the core functionality. After finishing the first story, the user will gain a better sense of the system being build and may be better able to provide feedback. In fact, in many projects the first story becomes a catalyst for new ideas and insights.

Is it the functional design?

It's not. A study performed by the Standish Group involving more than 40,000 projects revealed several common reasons for the failure of so many projects. One of those reasons was that projects erroneously assume that it is possible to write down functional requirements in an unambiguous way. Another one involved the equally wrong assumption that the business knows exactly what they want from the start. Hence Agile methodologies work with short cycles and strongly favor intensive involvement of the business.

That is exactly why a user story should be seen as a reminder to have a more detailed discussion with the business stakeholder at the time the functionality is going to be built. Only then you'll be able to gain the most from the increasing insights of executing a project. And only then it is possible for the stakeholders to evaluate a live version of the system, albeit in an early stage. Watching an early version of a product is often a catalyst for a lot of new requests or changes to the requirements

But stories are finite so they cannot be used as lasting functional documentation. In other words, while the product or system is still under construction, the stories are used as a placeholder for notes, discussions, estimations and to keep progress. But when the functionality has been realized, the stories won't have enough details to be used

as a future reference. Worse, the stories are usually not even updated if improvements and additions are made during a demo to the stakeholder.

Additionally, while executing our first project using stories, we noticed that we lacked some structure. First, the team needed a central place to store the concepts and business rules from the business domain. Because I already had a lot of experience with domain models such as defined by Martin Fowler, and I got more and more involved in the principles of Domain Driven Design (a concept introduced by Eric Evans), it was obvious to me that we were going to use a domain model here.

Also, the customer representative sometimes loses the overview of the many stories that were on our list of requirements. After a bit of Googling, I ran into a whole series of posts and discussions initiated by Alistair Cockburn. Alistair had the same problems and experimented with use case diagrams to illustrate the system context. I now use use case diagrams to represent the screens, external dependencies and processes. This appeared to be very helpful while talking to your customers about a new feature. And even though you can store the entire functional design in a use case, we stick to simple diagrams only. All functional details are still stored in the stories.

Who? What? Why?

Another aspect of user stories is the fact that traditional requirement gathering techniques usually deal with the needs or desires of the stakeholders, and if you're lucky, with the roles a user needs to have for that. However, these techniques almost always lack the reason behind such a requirement or identify the goals the user tries to reach. Consequently, user stories must be written in the form "As *who*, I want *what*, so that I *why*". The *who* describes the role or the stakeholder in whose interest the requirement defined by the *what* was introduced. The *why* should explain why the requirement is so important for the product, the project or the stakeholder. A nice example of how that can help us as developers is that because we now know what the stakeholders tries to accomplish, we might be able to come up with a better alternative. Be aware though that you don't end up with a reason that practically repeats the *what*. The trick is to keep on asking until you find the real reason.

So who writes them?

That depends. In Scrum it is the *product owner* (the term I use in the rest of this article) who represents the customer(s), and he or she has full responsibility over the list of user stories that comprise the product or system (the *product backlog*). And since he is one most knowledgeable about the particular domain, usually he is also the one who is supposed to write the user stories. In reality however, it is a combined effort with the team that realizes the functionality. Writing good user stories is not trivial, you know.

Ideally, user stories comply to a number of requirements captured in the acronym INVEST. First, they should be as *independent* as possible. After all, the more independent they are from each other, the easier it is to change the priorities between stories. No doubt this might be difficult to achieve sometimes, but quite often you can fix that by splitting a big story or combining two small stories.

The second requirement is that a story should be *negotiable*, which means that the description of a story should promote a discussion later on in the project. As I wrote earlier, stories are no substitute for the functional design, but rather a reminder to discuss the details with the product owner just-in-time. If you write stories that provide too much detail upfront, then you might give the impression that no discussion is needed anymore and that would go against the entire essence of user stories.

The third requirement, *value*, demands that a user story must have business value. Take for instance the product catalog application I've mentioned before. A screen for adding products has no business value on its own. After all, as long as a potential customer cannot see those products yet, such a screen would not make a difference. Building both screens fully with all its bells-and-whistles is not an option either. But the product owner might just accept a first version that limits the customer functionality to a list of products with a description and a price. The ability to attach a product image could be built in the next version and is a separate user story. Unfortunately, breaking down functionality like that is not trivial, so it is important to work with the product owner to see which story has sufficient business value.

The fourth requirement states that every story must be *estimable*. This may sound silly, but once you find it difficult to estimate the size of a user story, it is very likely that the scope of that story is too wide. However, it's normal that the estimates for stories that will be picked up at a later stage of the project are very rough. In fact, any attempt to provide a more detailed estimate may create the illusion that the key decisions were already made at the time of creation. As mentioned earlier in this article, it is important to estimate at the last responsible moment when you have the most up to date information available.

An obvious solution for ensuring that user stories are estimable is by keeping them *small*. A good sanity check that I've read somewhere in a blog is to ask someone in the team for an estimate for a fairly big story. If he or she answers that question with something along the line like: "Uhm, something like a month or so" then you can bet that the developer has no idea about the content and scope of the story. In that case it is not unwise to try to split this story into several smaller stories that still have business value but which scope is very limited.

And that brings us to the last letter of the acronym - the 't' for *testable* - not to be confused with automated testing or something like that. Instead, it refers to the fact that a user story must include acceptance criteria that should allow verification that the team's interpretation of the story matches the original intention of the product owner. As the product owner has not seen the end-result yet, getting those criteria might get challenging sometimes. What could be helpful is to ask him how he would demonstrate that feature other users. In typical Scrum methodology for that is the *how-to demo* part of a story.

How do you limit the scope?

From my own experiences I know that finding the right scope is very difficult, especially in the presence of the product owner or another stakeholder. Something I found very helpful is by using so-called *storytypes*. The idea is that a properly scoped story matches exactly one storytype. If you fail to find a suitable storytype than you can assume that the story is either too big, or too small, and you should combine or split them.

The concept of using stereotypes to scope user stories was first described by Gerard Meszaro in [Using Storytypes to Split Bloated XP Stories](#). In that article he identified the four storytypes listed in the table below.

Storytype	Description
New Functionality	A new piece of basic functionality that is (reasonably) independent of other user stories.
Variation	An addition or extension of a previous user story, such as additional search fields or extra columns.

New Business Rule	Additional (complex) business rules to an existing feature.
User Interface Enhancement	Improving the look and feel of the system such as applying a corporate style, improving the layout, or anything else that gives the application a more fancy look.

Now imagine a screen where you can search for products from a catalog. In an approach based on *fish level* use cases, the entire screen, including all business rules, available search fields and the entire look and feel of the product list, is treated as a single use case. But if you want to break down this particular screen into a smaller number of user stories that still comply with INVEST, those four storytypes can be very helpful.

As a first story you could limit the searching functionality to only the name of a product and present the search results in the simple layout. That still provides enough business value and hence complies with the *New Functionality* storytype. You can add any additional search options later as one or more *Variations*, and pimp up the look and feel using an *UI Enhancement* storytype. If you have to give up some of those extra features due to lack of time, then your basic functionality is still valuable enough.

If you've already looked at the User Story work item in Visual Studio 2010 you might have noticed that it doesn't have a storytype field. But don't worry. Adjusting the MSF Agile 5.0 process template is fairly easy to do, especially with the [Team Foundation Server 2010 Power Tools](#). However, for your convenience, you can [download](#) a customized version of the User Story work item type from my blog.



Classification

Area: AgileTest

Iteration: AgileTest

Storytype: Basic Functionality

What's it worth?

In reality, those storytypes have really helped me to get the scope right in discussions with the product owner. Yet, I still noticed several types of activities that do not quite suite any of the four storytypes nor any of the requirements defined by INVEST. I regularly run into requirements that do not originate from the product owner, but which are still important for getting and keeping the system into production. For instance, the IT department may have specific demands in order for the system to be deployed in their infrastructure. Or maybe they require real-time logging so that they can monitor the system. As an architect you probably want the system to be built in a certain way so that the maintainability is guaranteed. And as a team leader you may want to set-up an automatic nightly build to track the quality of the development activities.

You could embed these tasks within one of the normal user stories. But you could also reserve a fixed percentage of your capacity for those. However, I believe that it is better to keep all of those non-functional issues explicitly visible in your project. It also helps in the discussion with the product owner when you have to explain why you will take up only a limited number of user stories in a particular sprint. For these reasons, my user stories must have *project value* and not necessarily *business value*.

Where do you start?

Suppose that after intensive discussions and tough scoping sessions you ended up with a list of user stories and are about to start building the system. The first story not only needs to realize some particular feature, but also involves building a skeleton implementation of the system's architecture. How do you avoid spending way too much time on plumbing and other general purpose stuff you need for the rest of the stories?

The article [Managing the Bootstrap Story](#) by Jennitta Andrea addressed this challenge in more detail and offers some alternative solutions. One of these solutions is to find and define a user story with the product owner that offers minimal functionality yet still has project value. Such a story is often referred to as the *backbone* story because you realize the backbone of your system in it. It's quite common to use the backbone story to realize a proof-of-concept (PoC) that verifies the chosen architecture. Since a working PoC can give the product owner confidence that the team is able to build such a product, that fact alone may be enough project value for the product owner.

More storytypes?

You might have suspected it already, but that *backbone* story is just an example of another storytype. In fact, after I started looking for an approach to capture the non-functional requirements of a project or system, I ran into a slide deck that mentioned a whole set of additional storytypes. [Dan Rawsthorne](#), the author, tried to define a storytype for virtually every possible thing you might need to do in a project. Personally I think he went a bit too far, but a small set of additional storytypes proved to be very useful anyway.

Storytype	Description
Compound Epic	A composite user story that groups a number of stories in a logical sense.
Complex Epic	A user story whose content and impact must be determined later in the project, but for which it is clear that it involves a significant amount of work.
Setup	A story that is used to setup the project environment, including a source control environment, a project website, a build server.
Technical	A story that involves making a technical improvement or adjustment. Examples include introducing a coding standard, refactoring a poor design, executing a performance test.
Documentation	A story for writing a user manual, installation manual, etc.
Training	A story for developing and/or hosting a training, or having a workshop with end users.
Quality Improvements	A story which objective is to fix a collection of related bugs, or spent a fixed amount of time to improve the quality of the code base.
Spike	A story that aims to do a technical investigation to determine the usability of a specific technology, or for trying an alternative technical solution.

When is the story complete?

So how do you know that a user story has been successfully realized? Well, if all is good, all stories will conform with INVEST and are associated with a number of acceptance criteria (typically written down as the *how-to demo*) specified by the product owner. That should be enough to determine if it is functionally sound. But what you still miss is a way of explaining the stakeholders, including the product owner, when the team treats the story as

finished. That may differ by team, but usually includes some or more of the following criteria.

- The code compiles and there are no warnings or errors.
- The code meets the coding standards setup by the project or the organization.
- The code is reviewed by a peer developer.
- All automated unit and integration tests have completed successfully.
- Visual Studio's static code analysis tool does not report any violations.
- ReSharper reports no potential errors (a.k.a. everything is 'green').
- The daily integration build has completed successfully.
- The functionality was tested by another member of the team (anybody but the developer).
- The feature or functionality has been signed off using the project checklist.
- The system functionality is tested by a tester.
- The visual look and feel is has been approved by an employee of the communications department.

Together with the story's *how-to* demo these criteria are commonly referred to as the *definition-of-done*. Usually, a team or project will have a default definition-of-done that applies to all stories and only mentions the particulars of that story if necessary.

Then what about the planning?

User stories are an excellent unit for tracking progress within your project. However, purists within the Agile community will tell you that an Agile project will have no long term plan. Instead, the functionality is realized iteratively according to the priority defined by the product owner. I agree with the latter and believe that its iterative nature is essential for dealing with the changing requirements that are common in all projects. It allows deferring decisions to the last responsible moment, and that's always a good thing. But in reality you often can't escape from providing at least a rough schedule to your management. How should you deal with that?

What I often do to get all stakeholders to join me in a number of workshops. Using use case diagrams to illustrate the context of the discussions, I try to get enough stories on paper to represent the entire scope of the project. You need to beware though that you don't write down too much details or have too much in-depth discussions. That would give the stakeholders a false sense of precision, and consequently, will cause them to see the stories as a formal functional design. Also, if you run into some high-level chunk of functionality for which nobody really knows what it will look like, add an *epic* story for it and include a *spike* to elaborate on the *epic* later on in the project.

Then organize a number of shorter meetings with the team or, if the team hasn't been formed yet, with a few experienced developers. Let them discuss every story one by one and then try to estimate the size of each story in so called *story points*. Some people from the Agile community say you should estimate using relative sizes only. In other words, a story that seems to require twice as much work as another story should also have twice as many story points. The story point as a unit does not have value. It's the relative differences that are important.

What works for me is that every story point corresponds to the ideal day of an experienced senior software developer. In other words, one story point means that an experienced developer familiar with the chosen architecture, technology and project methodology needs to work for 8 hours without being disturbed by telephone, email, coffee breaks, or any other distractions. [Mike Cohn](#), author of [User Stories Applied](#), has dedicated many

chapters to this estimation technique. Ideally, each story is between 1 and 8 story points, but at the beginning of the project you still may have some epics to break up.

After finishing those meetings you should have an estimate of the total size of the project. Now, in order to get from those story points to a total number of hours you need to estimate the expected productivity of the team. Mike Cohn does this by creating a table with the expected roles, their availability (to deal with part time employees), and the expected productivity compared to the ideal senior developer (as a percentage). By calculating the average productivity and multiplying it with the number of story points you'll end up with the total number of estimated man-hours. It's only an estimate and both the productivity can be disappointing as well as the estimate in story points may appear to be wrong. But it still gives you an initial estimate that can be used for global planning and budget discussions. Obviously it is important to ensure that you keep on continuously measuring the actual productivity.

Wow, now what?

By now, it should be clear that a user story is not an independent concept but something that closely resonates with many of the aspects of our work in the software industry. In this multi-part post I have tried to explain a number of those aspects and to clarify the relationship between them. But even though I've not touched everything as detailed as possible, I still hope I've managed to convince you about the power and potential of user stories. Last but not least, if you have any questions or comments, please do not hesitate to email me at dennis.doomen@avivasolutions.nl or tweet me at my Twitter ID [ddoomen](https://twitter.com/ddoomen).