**User stories**

Typically, each user story will have multiple associated tasks.

In Backlogs DON’T DO

~~User Story  
Coding  
Testing  
Check-in  
Build  
Demo~~

OR

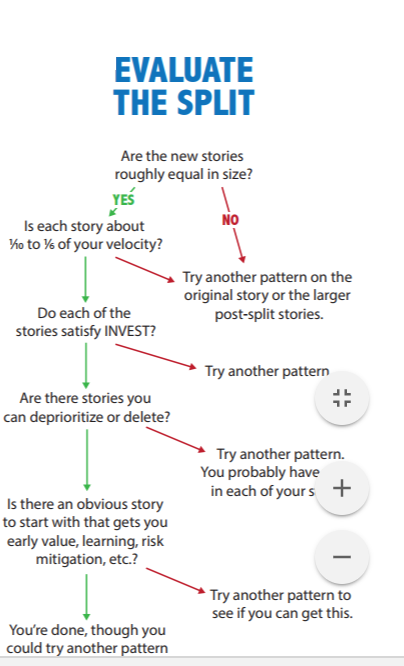
~~Explore> Develop> Test> LIVE~~

**Some guidelines for a good user story include:**

* It should be written by someone who represents the business users (usually the product owner)
* It should initially include brief descriptions of the "**who, what, and why**" but **not the "how"**
* It should produce a vertical slice of working code
* It should be small enough that it can be coded and tested in one iteration (usually a one- to four-week period)

It should includes

1. The **actor** makes sure you’re thinking about who will use this feature. If there isn’t an identifiable customer for the feature, you should reconsider whether you need it.
2. The **action** describes what will happen, but not \*how\* it will happen (so in the case above, not ‘I want to pick an option from a list of three possibilities, using a radio button display’). User stories are designed to start a conversation within the team about the best way to make this feature.
3. The **achievement** describes the ultimate purpose of the feature. If you can’t think of an achievement, that’s a signal that you should reconsider whether the feature you’re trying to describe is actually important.



**1.  Create Meaningful tasks**

Describe the tasks in such a way that they convey the actual intent. For example, instead of saying Coding, describe the tasks as  “Develop the login class”, “Develop the scripting part for login functionality”, “Develop the password encryption for the login functionality”

**2.  Use the Definition of Done as a checklist**

It includes all items that have to be completed to claim that the story is done by the development team. A simple example:

Acceptance criteria is verified during testing

Coding tasks completed.

Exploratory Testing completed and signed.

Regression test reviewed and passed.

**3.  Create tasks that are right sized**

Tasks which are very small, broken down to a minute level like, 10 min, 30 min, 5 min tasks, for example: Write Accept User Name Password, Validate Login, and Give Error Messages. Breaking the user stories with too many details is an overhead. **One guideline is to have tasks that span less than 8 hours so that each one of them can be completed in at least a day**.

**4.  Avoid explicitly outlining a unit testing task**

If possible, make unit testing not a separate task but part of the implementation task itself.  This encourages people to practice [Test Driven Development](http://www.payton-consulting.com/agile-test-planning-primer/) as an approach. However, this practice may not be ideally suitable for new Scrum teams.

There are many strategies available for breaking up large user stories.

**Breaking up large user stories**

**Strategy 1: Split by workflow steps**

If user stories involve a workflow of some kind, the item can usually be broken up into individual steps. Take a user story for an order process of a regular webshop:

**As customer I want to purchase the goods in my shopping basket so that I can receive my products at home**

*As customer I want to log in with my account so I don't have to re-enter my personal information every time;*

*As customer I want to review and confirm my order, so I can correct mistakes before I pay;*

*As customer I want to pay for my order with a wire transfer, so that I can confirm my order;*

*As customer I want to pay for my order with creditcard, so that I can confirm my order;*

*As customer I want to receive a confirmation e-mail with my order, so I have proof of my purchase;*

By breaking up a large user story like this, we have improved our understanding of the functionality and our ability to estimate. It will also be easier for a product owner to make decisions about priority. Some workflow steps may not be important right now and can be moved to future sprints.

**Strategy 2: Split by business rules**

**As customer I want to purchase the goods in my shopping basket so that I can receive my products at home**

*As shop owner I want to decline orders below 10 dollars, because I don't make any profit on them;*

*As shop owner I want to decline customers from outside the US, because the shipping expenses make these orders unprofitable;*

*As shop owner I want to reserve ordered products from stock for 48 hours, so other customers see a realistic stock;*

*As shop owner I want to automatically cancel orders for which I have not received payment within 48 hours, so I can sell them again to other customers;*

Often, the test cases imply important business rules. Once the business rules have been identified, it will have improved our understanding and ability to estimate. The product owner may decide that some business rules are not important for now, or can be implemented in a simplified form.

**Strategy 3: Split by happy / unhappy flow**

Functionality often involves a happy flow and one or more unhappy flows. The happy flow describes how functionality behaves when everything goes well. If there a deviations, exceptions or other problems, unhappy flows are invoked. Take this user story for logging in to a secure website:

**As customer I want to log in with my account so that I can access secured pages**;

If we consider a regular login procedure, we can identify a happy flow and several potential unhappy flows:

*As user I want to log in with my account, so that I can access secure pages (happy);*

*As user I want to be able to reset my password when my login fails, so I can try to log in again (unhappy);*

*As user I want to be given the option to register a new account if my login is not known, so I can gain access to secure pages (unhappy);*

*As site owner I want to block users that log in incorrectly three times in a row, so I can protect the site against hackers (unhappy);*

Unhappy flows describe exceptions. By identifying the various flows, we more clearly understand the required functionality. A product owner can also more easily decide what is important right now.

**Strategy 4: Split by input options / platform**

Many web applications have to support various input options and/or platforms, like desktops, tablets, mobile phones or touchscreens. It may be beneficial to break up large items by their input options. Take a digital Scrum Board for a team:

**As team member I want to view the Scrum Board, so I know the status of the sprint**

*As team member I want to view the Scrum Board on my desktop, so I know the status of the sprint;*

*As team member I want to view the Scrum Board on my mobile phone, so I know the status of the*sprint*;*

*As team member I want to view the Scrum Board on a touchscreen, so I know the status of the*sprint*;*

*As team member I want to view the Scrum Board on a printout, so I know the status of the*sprint*;*

By splitting up large items like this, the product owner can more easily prioritize which input options or platforms are more important. It's probable that a desktop version is sufficient for now, while a mobile version is pushed to a future sprint. Or perhaps the printout can be implemented with a simple PDF capture of the board for now, without having to create a version specifically suited for print.

**Strategy 5: Split by datatypes or parameters**

Some user stories can be split based on the datatypes they return or the parameters they are supposed to handle. Take, for example, a search function for a webshop:

**As customer I want to search for products so I can view and order them**;

There are many ways to search for a product. All these potential parameters can be considered and broken up into smaller user stories:

*As customer I want to search for a product by it's product number, so I can quickly find a product that I already know;*

*As customer I want to search for products in a pricerange, so that the search results are more relevant;*

*As customer I want to search for products by their color, so that the search results are more relevant;*

*As customer I want to search for products in a productgroup, so that the search results are more relevant;*

By breaking up the search functionality, we now more clearly understand what kind of search parameters will be used. This allows us to more accurately estimate the functionality, but it also allows a product owner to make decisions about priority. Maybe paging is not yet relevant because of the small number of products. It might become relevant when the number of products grows.

**Strategy 6: Split by operations**

User stories often involves a number of default operations, such as Create, Read, Update or Deleted (commonly abbreviated as CRUD). CRUD operations are very prevalent when functionality involves the management of entities, such as products, users or orders:

**As shop owner I want to manage products in my webshop, so I can update price and product information if it is changed**

By identifying the individual operations required for this functionality, we can derive the following smaller bits of functionality:

*As shop owner I want to add new products, so customers can purchase them;*

*As shop owner I want to update existing products, so I can adjust for changes in pricing or product information;*

*As shop owner I want to delete products, so I can remove products that I no longer stock;*

*As shop owner I want to hide products, so they cannot be sold for the time being;*

When presented with this strategy, many teams wonder if the smaller items actually deliver business value.

**Strategy 7: Split by test scenarios / test case**

This strategy is useful when it is hard to break down large user stories based on functionality alone. In that case, it helps to ask how a piece of functionality is going to be tested. Which scenarios have to be checked in order to know if the functionality works? Take a task planning system:

**As manager I want to assign tasks to employees, so the can work on tasks**

If we consider this functionality based on potential scenarios, we can break down the item into:

*Test case 1: If an employee is already assigned, he or she cannot be assigned to another task;*

*Test case 2: If an employee has reported in sick, he or she should be visually marked so they can be ignored;*

*Test case 3: If an employee has reported in sick, he or she cannot be assigned to a task;*

*Test case 4: If an employee is not yet assigned, they can be assigned to a task;*

*Test case 5: Employees can be assigned with a touchscreen monitor;*

This strategy actually helps you apply the other strategies implicitly.

**Strategy 8: Split by roles**

User stories often involves a number of roles (or groups) that performs parts of that functionality. Take a user story to publish new articles to a public newspaper website:

**As news organization I want to publish new articles on our homepage, so customers have a reason to return periodically**

By considering the various roles that are involved, we can break the functionality down into the following bits:

*As customer I want to read a new article, so I can be informed of important events;*

*As journalist I want to write an article, so it can be read by our customers;*

*As editor I want to review the article before putting it on the site, so that we can prevent typos;*

*As admin I want to be able to remove articles from the site, so that we can pull offending articles;*

*As customer I want to review and confirm my order, so I can correct mistakes before I pay;*

By breaking up functionality into the roles that have to perform bits of that functionality, we more clearly understand what functionality is needed and can more accurately estimate the work involved.

**Other strategies**

Of course, there are many other strategies that may be helpful when breaking up large user stories. I have listed some of them below, but please feel free to add more in the comments:

Split items based on identified acceptance criteria. This may seem very obvious, but it's often the easiest and most natural way to split up a story. Mapping out acceptance criteria for a user stories requires similar strategies as the ones described in this post;

Split items based on the parts that are hard to implement and the parts that are easier. It may be difficult to set up a piece of functionality in a heavily designed UI, but getting it to work with a simple UI may be easy and sufficient for now. Again, it's all about being pragmatic and delivering business value;

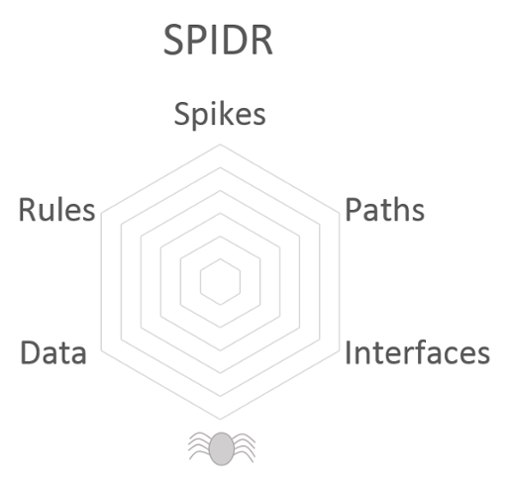
Split items based on external dependencies. Sometimes functionality is dependent on external factors, such as implementing a consumer for a remote webservice (e.g. for electronic payment or connecting to Twitter). This may be difficult, but may not have the highest priority. Or the dependencies can be mocked for the time being;

Split items based on usability requirements. This includes functionality for paging through a list of records, making a website readable for blind people or people with color blindness or implementing breadcrumbs;

Split items based on SEO requirements, such as setting up dedicated landing pages for specific keywords, setting up goals for Google Analytics or setting up XML sitemaps;

Split items based on browser compatibility. Getting a website to work in Internet Explorer 6 is time-consuming, but is sometimes are requirement (sadly). It may be beneficial to at least consider if support for older browsers can be postponed in favor of newer browsers. In that case, a strategy has to be identified to deal with browsers with older browsers though (like a warning message);

TIP Most teams do not immediately 'get' how to break down functionality. It is not uncommon to run into some resistance as a result. This is understandable; trying out new things is difficult because it makes people feel vulnerable. The best way to deal with this, is to persist and gently coach the team by helping them break up their user stories.



**Spikes**

Spike is a term used in agile software development. Spikes are small, prototypical implementations of a functionality that is typically used for the evaluation and feasibility of new technologies.

This method involves investigations and building knowledge. It should be used if other SPIDR methods have not worked well. With the help of such newly acquired knowledge, some stories can then be better understood and possibly split more easily. This method, however, is relatively abstract and therefore harder to apply than the remaining methods.

**Paths**

If there are several possible alternative paths in a user story, one option is to create separate user stories from some of these paths. It is not absolutely necessary to write a story for each individual path, just where it makes sense. For example, let's take a user story in which the user wants to be able to pay for purchases in an online store. There are now two possible paths: payment with a credit card or payment with Paypal. Payment with a credit card can theoretically be further subdivided, but you need to weigh up whether it makes sense for each type of credit card to have its own story. The overriding task of paying for purchases is, however, divided well into the two alternatives mentioned. Thus the newly created stories are smaller and more easily estimated.

**Interface**

Interfaces in this context can for example be different devices or device types, such as smartphones powered by iOS or Android. User stories can also be split in terms of this diversity. Let's stick with the example of different operating systems: in a project, for example, there may be user stories that relate exclusively to the use of Android devices, or others that focus on web browsers. To avoid making stories too large and comprehensive, you should ask yourself which devices or interfaces you want to develop. Perhaps the first development result should only refer to iOS devices, because of the probably larger target group.

**Data**

Another technique for splitting user stories can be used when the initial stories refer only to a sub-range of the relevant data. Take the example of a website intended to attract tourists to a particular city. If it is a city known for its museums, for example, the first story could include information about the different museums in this area. A subsequent story could include various tourist tours through the city, and another deal with outdoor activities.

**Rules**

Business rules or technological standards can be another splitting factor. Take the example of online purchase of cinema tickets. There are often constraints that are for example based on business requirements of the respective cinema, such as an online purchase limit of a maximum of five tickets per e-mail address.

With this story it would be conceivable that the development team omit this restriction, allowing every visitor to buy as many tickets as they wish. The restriction could then be added in a second iteration step. Incremental delivery such as this means that initial stories are not immediately implemented completely, but instead are delivered in several smaller steps. Sometimes it makes sense to neglect technical specifications or business rules, if by doing so you can more quickly achieve a presentable result that satisfies the user or client. Omitted stories can be retrieved at a later date.