A rough guide to elaborating stories in agile projects

A summary of different approaches

There are many different approaches to elaborating stories and each team needs to find the best approach based on its circumstances and own style. So this document is not a summary of current best practice in working with stories but rather a description of some approaches that work for me.

I have broken the approaches into the following categories for the purpose of explanation:

Code and see
- Developers build from their understanding of user needs and ask frequent questions

Decomposition
- Breaking stories into more detail, for example by converting them into acceptance tests.

Abstract to concrete
- Abstracting the meaning of the story through a structured conversation and then defining the concrete elements of the story.

Modelling
- Creating a model from the high level stories and building the system from that model.

Comparison tables
- Listing stories on one axis of a table and comparing them to technical components or "quality factors". For example, comparing stories to non-functional requirements or key client drivers.

Code and see

It is a common misunderstanding that using stories means the developer must create a solution from nothing more than a single sentence (for example, from "As a retail customer I want to be able to purchase books so that I can read them").

In fact most teams give the developer a lot more information than what is contained on a story card.

However, there are times when the best approach to development is to simply have a conversation with the developer (and tester and the rest of the gang) and then let them put something together.

Of course, this does not mean that developers only get one shot to get things right. Rather, they continually come back with questions and demonstrations of what the emerging solution looks like.

© Creative commons copyright – see Appendix for details

This approach can be very effective for a team that sits close to its stakeholders, has a good understanding of customer needs and can make frequent changes to a design at relatively low cost.

Tips and things to watch out for

In this approach, it is critical that feedback is constant and that the developer has direct contact with someone who will make decisions about the final product.

It is also important that the developer, tester, business stakeholder and any other team members see themselves as equals. If the developer is scared of the business representative or worse, contemptuous of the business representative's views then this approach will lead to trouble.

This approach can work very well when creating a new user experience, though the solution is normally demonstrated on paper or in sketch form. The key to success is that there is a low cost of being wrong (ie doing rework) while building the product.

The approach can also work well when reverse engineering a solution (eg replacing an aging system) as long as the developer and customer are challenging each other's thinking and confirming the solution along the way.

The major risk is that the developer will build an ad hoc solution that does not integrate into the broader system. There is also a risk that the developer will build "an egg" – seeming solid on the outside but fragile and easy to break ... and then very hard to put back together.

Decomposition

Acceptance tests

The approach we teach in our courses is to break stories into acceptance tests.

For example, if I have the story:

> As an Iphone user I want to SMS my friends to that I can share my experiences with them

Then the team come up with a series of tests along the lines of "given the user is on the SMS screen, when he/she presses the send button then the SMS will be sent.

Given	When	Then
The user is on the SMS screen	The user hits the cancel button	The Iphone will ask "are you sure"
The user is on the SMS screen	The user presses the content section	The keypad will be displayed
The date is international leprechaun day	The user enters any text	The iPhone will convert the text to Leprechaun Gaelic.

Tips

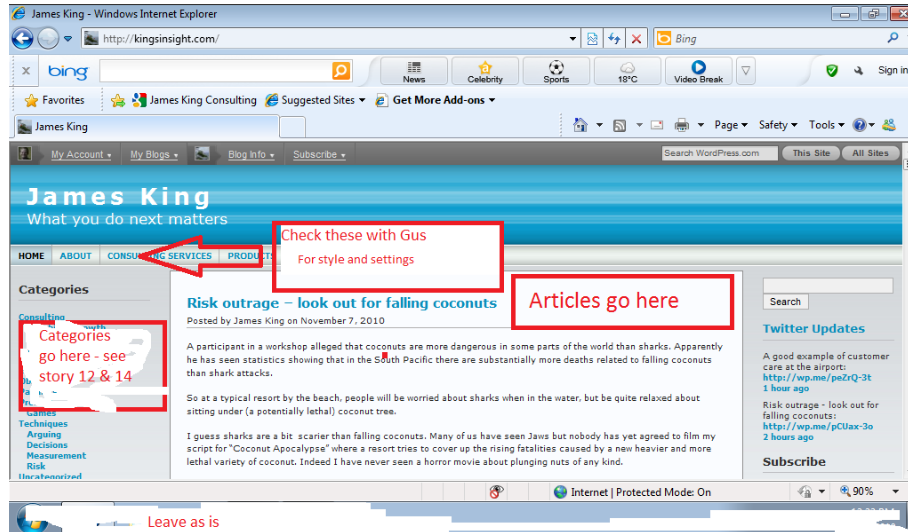
These acceptance tests are an excellent way to define the solution space for the developer and are often used in conjunction with any or the remaining approaches I will describe.

More often than not, teams use acceptance tests as a base and add more information if needed.

© Creative commons copyright – see Appendix for details

Generally speaking I would add a simple "narrative" if the subject matter expert is not sitting with the developer. This is a short summary of the context of the story (one or two paragraphs of text).

I also find a simple sketch or model (with comments) to be of real value to capture some of the understanding, rather than listing everything in acceptance tests.



Complex business rules can sometimes be summarised in a decision table or a decision tree more effectively than in multiple acceptance tests. For example:

User setting	Refuse to send SMS	Attach location settings	Create audit record	Confirm send
Child emergency use only and number entered is not parents	Yes	Yes	Yes	No
Novice	No	Yes	Yes	Yes
Undercover cop or criminal	No	No	No	No
Otherwise	No	Yes	No	No

It is not the role of any one person to create the acceptance tests. Rather, they are created through the collaboration of the tester, business analyst, subject matter expert and developer.

If the same test keeps appearing then convert it to a standard, so that it is taken into account by default when coding or testing.

When doing acceptance tests in conjunction with other information, do the tests first and then see if any can be added when other approaches to elaboration have been completed. Doing the tests first often clarifies understanding quickly and saves work in narratives, etc.

Watch out for:

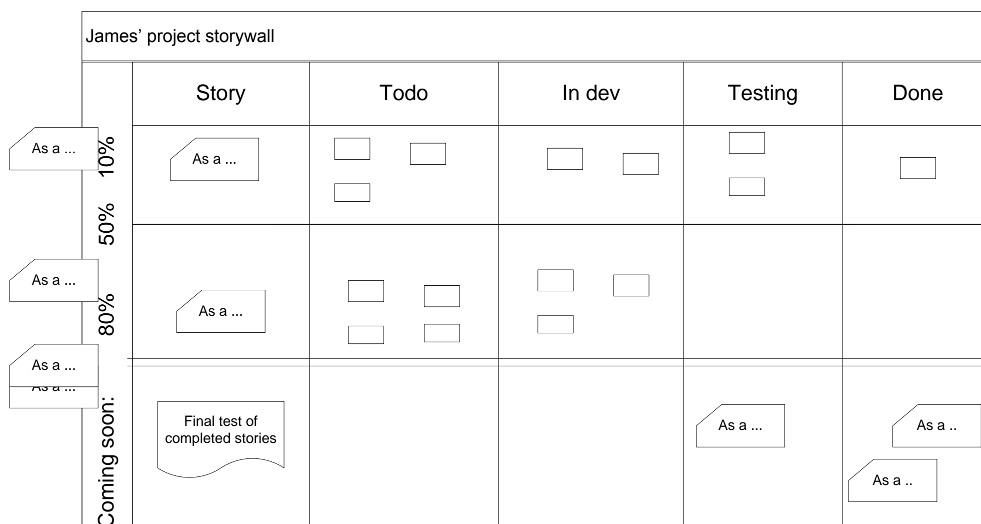
- Acceptance tests are not a contract. The developer should question and add to the tests during development.

© Creative commons copyright – see Appendix for details

- The items mentioned above are not a finite list of information to provide. Teams should feel free to create data models, lists of constraints and risks, references to development standards etc. The acceptance tests are the framework for understanding not the complete understanding.
- Some teams start to fall into what I call "panic waterfall" where developers expect every nuance to be documented. But the rule of sum should be to only document what cannot be reliably communicated through a conversation while the developer is working. That's why I like the acceptance test approach – it allows the team to think about outcomes before writing code, while leaving room for exploration by the developer while creating the code.

Tasks

In some teams, the iteration or sprint begins by discussing each story and breaking it into the specific tasks that each team member needs to perform to code, test and deliver it. The team may then track tasks on their story wall rather than complete stories.



Tips

There is a common rule of thumb that no task should be more than 1-3 days work, so that the team can work rapidly in small chunks. This allows the team to have momentum and to focus on what needs to be done next.

Most teams break the whole iteration worth of stories down into tasks on the first day of the iteration. This is a good approach because it allows the team to see where some tasks might be combined for multiple stories.

However I like to have the team start on a subset of the stories to break down first. Then as those are completed or likely to be completed, to grab more stories and break them down. I usually get the developer, tester etc working on the particular story to do this rather than the whole team. However this carries the risk that we will miss integration issues or synergies. So the team will need to decide the best approach in their specific situation.

Some teams also add "unplanned" tasks on a different colour card to record work that was not anticipated in the original planning. This can be useful information for the team to reflect on at the end of the week.

© Creative commons copyright – see Appendix for details

Watch out for

Some teams will leave the task breakdown to one team member who then directs the rest of the team. This puts a lot of pressure on one person and discourages joint accountability and input.

I like to display "Stories coming soon" to show which stories we haven't started on. I also like to show them as "100% - we are sure they will be in this iteration", "80%", "50%" and "10% likely to make this iteration".

I find this helps the team start on the most critical stories and push them through before moving to the less critical ones. However I have been warned that this can let the team off the hook in committing to complete the whole lot. It also confuses the team if they are using a burn down chart during the iteration and are not sure whether to include all the stories in the target.

Technical stories or detailed specs

Rather than breaking the story into tasks, some teams keep breaking stories into multiple, smaller stories until they are small enough to build from.

So, for example the story:

> As an Iphone user I want to SMS my friends to that I can share my experiences with them

Might be broken into:

> As ... I want to select a contact to SMS from the phone book so I can send an SMS
>
> As ... I want to be able to type a phone number so that I can send and SMS
>
> As ... I want to be able to type text to include in an SMS
>
> Etc

I often tell people to break their stories down until they are less than half an iteration's work because that is my rule of thumb for when stories will become unwieldy. But others advise that no story should be more than 1-3 days work.

I also generally say to split a story into acceptance tests rather than many detailed stories but in some cases teams do find it useful to break stories into either technical components or smaller stories.

So some teams use a different format for their stories when they are building them and they abandon the "As a ... I want to ... so that ..." format. They then call these stories "technical stories" and might even use a different colour card to highlight them on a story wall.

Tips

Many teams simply write technical stories as comments or reminders, along the lines of "look up contacts", or "add lookup".

But I think it is worth keeping to a standard format agreed by the team. For example:

© Creative commons copyright – see Appendix for details

- "Trigger (who or what triggers an event); event; condition, input needed; output created; constraints; errors.
  - When the user selects contacts, the system displays the contacts if there are any. No input is needed and the system will display the phone book;
- "System" must "condition"
  - The screen must comply with branding standards; or
  - Information displayed must be accessible to reader programs.
- Use Cases. I will not describe these here but this approach can integrate into an agile project by treating the stories as use case goals and then elaborating the stories by expanding the use case as required.

Watch out for

There is a high risk with this approach that you will trap the team in "panic driven waterfall". This means that the BA's will not have enough time to write proper requirements and at the same time the developers will expect ever more detailed requirements before building.

So I suggest that you

- Make sure the creation of the detailed stories is the role of the whole team and not just the BA;
- Keep reminding the team that these detailed stories should only be added where they provide more clarity or better tracking than a conversation; and
- Keep defaulting to acceptance tests and pictures where possible rather than creating new detailed stories if the story will represent less than a day's work.

There is also a substantial risk with this approach that it will lead to scope creep.

The team might be quite disciplined up until they elaborate their stories, even carefully prioritising and estimating stories diligently.

But then as they start to write their technical stories the stories become an un-prioritised list of possible things the story could relate to and the team feel trapped into delivering all of those components, even though none of them is prioritised.

So you should use MOSCOW or another prioritisation approach to handle the technical stories if you need to, but a better approach is to write less of these stories. Keep reminding the team that the technical stories are not new requirements, they are simply a clarification of the stories, and use conversation instead of "detailed requirements" where-ever possible.

Also, I recommend tracking the technical stories as sub-components of the original user stories rather than losing sight of which user need (story) the technical story relates to.

Abstract to concrete

"Abstract to concrete" is a term I am going to use to describe the idea of elaborating stories by:

- Using a structured format to interrogate the story;

© Creative commons copyright – see Appendix for details

- "Abstracting" the context and meaning behind the story rather than just decomposing the story into smaller pieces; and
- Making the abstract conversation more concrete by discussing what needs to be done in order to build the story.

It may seem strange at first glance to take the story up a level and discuss why you need it as well as what else could be accomplished. In fact doing so creates the risk of scope creep.

But I find that quite often a structured discussion of both the context with which we are building in the story and the purpose behind the story leads to a better understanding of real needs and even the ability to deliver less because what is delivered is based on a sound understanding of real needs.

The standard Abstract to Concrete approach is to:

- Use a structured conversation to understand the purpose and context of the story (I have listed three tools below for doing this);
- Allow the scope and meaning of the story to mature before becoming "concrete";
- Stop and create concrete tasks, stories, models and/or tests using whatever method suits the team; and
- If desired, prioritise the "concrete" elements of the story using MOSCOW or another prioritisation process.

Some approaches to the structure conversation are:

- Using the standard story format to make the story more abstract or more concrete;
- The "Question Compass" or a similar questioning model (This could include "6 Thinking Hats", FMEA, or any techniques the team feels are suitable);
- Telling a story based on the format "that was then ... this is now".

Using the story format

People don't always notice how useful the standard story format can be, so lets look at how we can use it to make the story more abstract or more concrete:

The standard format is:

- As a [user] I want [something] so that [reason].

To make this more abstract I simply

1. Remove the [something] and replace it with [reason], leaving a blank space where [reason] was:
    - As a [user] I want to [reason] so that [blank]
2. Ask the group to come up with possible replacements for [blank] to make the story whole again.

For example, "As an iphone user I want to sms my friends so that I can share my experiences" becomes "As an iphone user I want to share my experiences so that [blank]". Then the group might replace [blank] with "so I will be popular" or "so my life is made public".

© Creative commons copyright – see Appendix for details

If you do this multiple times then it is really just a variation of the 5-why's technique, and it can lead to merging other stories, realising gaps in your understanding and, most importantly, discussing the real drivers behind a user's (assumed) need for the story.

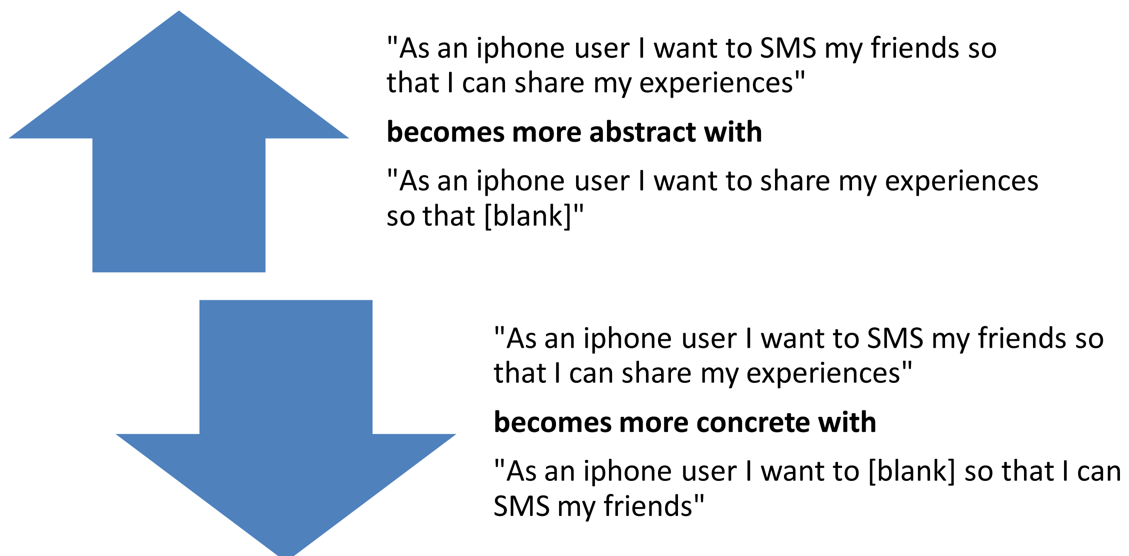
To take things a little further you can also take the [user] component and explore this with questions such as:

- Is this user part of a wider community? Who else is in that community?
- What do we know about this user?
- Who else might benefit from this story? Who else has a vested interest?
- What personal or business goals does this story help the user achieve?

Once you have abstracted the story you can create some acceptance tests if desired, and then make the story more concrete again by reversing the process and seeing where you end up (which is usually with multiple stories):

To make the story more concrete:

1. Remove the [reason] and replace it with [something], leaving a blank space where [something] was:
    - As a [user] I want to [blank] so that [something]
2. Ask the group to come up with possible replacements for [blank] to make the story whole again.



For example, "As an iphone user I want to sms my friends so that I can share my experiences" would become "As an iphone user I want to [blank] so that [I can SMS my friends]". Then the group might replace [blank] with "select a friend to SMS" and "Attach photos of my experiences to an SMS".

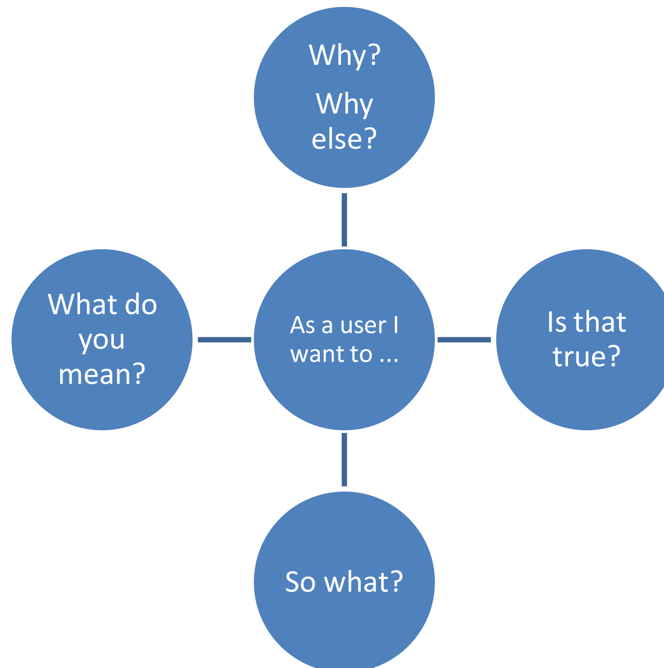
I sometimes use this process at the beginning of the project to get the team to explore and understand the real user needs and to either add additional stories to our list (by creating more concrete stories to meet the high level needs) or to cut down the number of stories that the team is managing (by leaving them at a more abstracted level). But it can also work well at the point where you are elaborating stories.

© Creative commons copyright – see Appendix for details

The question compass

Many BA's have their own format for structured interviews and mine is the poorly named "question compass". This is really just a series of questions I ask to better understand the context around any topic.

The compass looks like this:



Essentially I run through a series of questions for the story:

- What do you mean?
  - What do you mean by "any noun in the story"; for example "what do you mean by iphone?
  - How do you see people "verb", or how do people currently "verb". For example "how will users SMS?"
  - Keep going through nouns and verbs, not just in the story itself but also in the explanations to your questions.
- Why?
  - Why do people want to "verb"? Why else?
  - Why is this story aimed at that user in particular?
  - Continue asking why with respect to the answers you receive.
- Is that true?
  - When is that not the case?
  - Is that always the case?
  - What information are we basing that on?
- So what?
  - What is the impact of not having this story?
  - What will be different if we deliver this?
  - What else may need to change?

© Creative commons copyright – see Appendix for details

- What is the long term impact of not having this?
- What are we expecting will not change?

That was then – this is now

Rather than exploring the user story in detail, this approach focuses on the user.

Ask the group to tell a story (a tale or narrative) about the user and his or her life today. Have them:

1. Come up with a fictional character who is a typical user (this is sometimes called a persona).
2. Create a short tale describing what the user is doing without the new story being rolled out. Call this "That was then".
3. Imagine that the new story has been delivered successfully and create a new tale about how the user's life is different (and hopefully improved). Call this "This is how it should be".
4. Finally, imagine that the new story is not fully implemented or that the user is not taking full full advantage of of it. Create a tale where the user encounters frustrations and call it "This is how it should not be".
5. Discuss the gaps between "That was then" and each of the possible futures.
6. Finally – create acceptance tests to describe "this is how it should be" and to avoid or clarify constraints in "This is how it should not be" and to take into account the gaps discussed in step 5. The acceptance tests are then called "This is now" and are used to build the story.

Your own creative approach

"Abstract to concrete" approaches might not be needed for simple stories and might seem like overkill if you are working on a large number of stories on the project. But I find they can be very useful in creating a better understanding of the user's real needs.

The approaches I have listed here work for me but you should feel free to explore any approach the team would like to apply. The key is to spend time "abstracting" the meaning behind the story before listing the "concrete" components of the story (acceptance tests, tasks or technical stories).

Comparison tables

Comparison tables basically all work the same way.

- Stories are listed as either column headings or row headings ;
- Story components or characteristics are listed for each story.

Typical comparison tables include:

- The "things that matter" table; and
- The "non-functional requirements" table.

Things that matter table

The "Things That Matter" (TTM) matrix is a simple tool for understanding a story. The components of a system that be involved in a story are shown on the left of the matrix and stories are listed on the top, as shown in the table below:

		Stories				
		1	2	3	4	5
Possible components	HTML	L	H		H	L
	Wireframes	M		M		
	Java		H			
	CMS		H			I
	User research		H	M		
	Integration testing		M		M	
	Regression testing		M		H	
	SEO changes		I			
	Changes to contract		I		H	
	3rd party involvement		H		H	
	Training for sales team		H	M		

In this case, I have shown 4 possible connections between a story and a component (not counting the possibility of "no relationship"):

- L means that there will be some work on that component but it will not be complex;
- M means that there will be a moderate level of work involved;
- H means a high level of work and complexity; and
- I Means no direct work but this component is indirectly impacted and may need testing of collaboration with the team building that component.

This comparison table might be used to clarify which systems are involved to assist the team in defining which tasks each team member must do, or it might be used to generate acceptance tests.

Non-functional requirements comparison table

A similar approach to the "things that matter table" is to list the non-functional or qualities that are important in a system. In the table below I have taken the quality definitions from ISO 9126, which is an international standard for defining software quality:

Quality	Stories				Standard or guide
	1	2	3	4	
Accuracy					
Reliability		L		L	

© Creative commons copyright – see Appendix for details

Usability	H	H			
Accessibility	H	M		M	Organisational accessibility stds
Efficiency			M		
Maintainability			M		
Portability			M		Tgt architecture v9.3

I have rated the importance and complexity of the non-functional requirement as high, medium or low. However you might choose to include specific targets, constraints or statements in the table.

The non-functional comparison table might be used to generate acceptance tests or to assist the team in the overall conversation about each story.

Using comparison tables

Comparison tables are used to generate a conversation about each story and to give the team some assurance that they are not missing important elements of the story they are discussing.

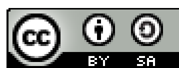
Using models

While I will not describe any models in detail here, it is possible to use stories with UML and similar modelling languages. It is also possible to create simple user interface sketches or even wire-frames as a way to elaborate the story.

These models can then be used to generate acceptance tests or they can be used as the basis for development. Do not hesitate to use models where they are useful as long as

- They are useful to the developer, rather than being more an obligation than a useful tool;
- They are consistent;
- They support, rather than replace conversations about development needs; and
- They can be understood by business users, testers, developers and anyone expected to be relying on them.

Appendix - Copyright



Notes on elaborating user stories by James King is licensed under a Creative Commons Attribution-ShareAlike 3.0 Unported License.

To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/3.0/ or send a letter to Creative Commons, 171 Second Street, Suite 300, San Francisco, California, 94105, USA.