User Stories and Story Splitting

Good user stories are at the heart of any high value Agile effort. It doesn't matter if you are a single developer or a large enterprise that produces software using the coordinated efforts of many teams. Success requires good user stories. This mini-book will show you how to write stories such that no matter how many teams you have coordinating together, they can ultimately put new functionality into production every two weeks or less based on stories that were created at the beginning of that two weeks.

Writing good user stories starts from a handful of basic principles. A user story is a user story if and only if it is something that:

- Can go from concept to release (to real users) in a week or less.
- Provides value that a real user will say "thank-you" for when it is released.

There are a number of basic beliefs that are important to understand before writing user stories:

- It is best to move from working and shippable software to a new version of working and shippable software a single small user story at a time.
- Producing more value over a given period of time is far more important than reducing the cost of providing a given amount of value.
- Our goal is not to produce specific functionality, our goal is to provide value to users.

Before going into the details of writing user stories, let's set up the context. The intention of this book is that everything within it applies regardless of:

- The number of people involved in the overall effort
- The complexity of the environment
- The complexity of the software
- The complexity of the domain
- The risk
- The required performance or scalability
- The interface (graphical user interface, voice, API, hardware, etc)

That said, you may also find that the discussion of writing good user stories suggests certain changes to the way your organization is currently organized or the way that your process currently works.

User Stories are Small Pieces of Value that can be Implemented Independently of Each Other

User stories are intended to be implemented one at a time in order of business value. You focus on one story until it is completely done and production ready and then work on the next one. This may seem inefficient. However, there are a number of good reasons for working this way. The first reason is that by going one story at a time your software goes from a working state to a working state on a regular basis. The second reason is that if you discover any problems when implementing a story, you can use that information when working on all subsequent stories. It also makes it easier for whoever is deciding which story to work on next to change their mind. That's useful because if the business value of the stories to be implemented changes, you want to focus on working on whatever has the most business

value next. And lastly, finishing small stories gives you confidence that you are making progress and can inform others of problems much earlier than when creating software using traditional methods. Think about it, which report of progress would you trust more: that the software is 50% of the way through the plan or that 50% of the functionality desired is working and can be shown to be finished?

User Stories are like Patches

A good way to understand user stories is to think of them in terms of patches (aka hotfixes). Why do you do a patch? Because if you don't somebody somewhere will be losing a great deal of value. That value may be revenues, customer satisfaction, or something else of value. In any case, you do a patch because it produces the most possible value to your organization compared to anything else you could do at that moment. It may not seem like value, but negating a negative value is a positive value.

Now think about the process that you go through to produce a patch. Since you want to get it out as fast as possible, you first try to reduce the scope down as much as possible and then do more as needed. As part of this you make sure that you are as clear as possible who the patch is for and exactly what it needs to do and why. You then write the code, create tests for the change and make sure everything is working well in a production-like environment. It is very likely that people are thinking about how to test the change and working on creating those tests at the same time that other people are writing the code. Lastly, you deliver the patch.

If patches are the process we use when it is most important and produces the most value, then why don't we use this approach all the time? Also, why don't we do our utmost to make this process as simple and stress-free as we can?

User stories are very similar to patches. They are small pieces of functionality that deliver value in and of themselves. Also, you work on user stories in the order that provides the most business value to your organization (which is generally aligned with providing the most value to the customer or market).

Traditionally, patches are done under stress and can lead to bad code. The thought of doing everything as patches can conjure images of "spaghetti code" or a "Frankenstein's Monster" architecture. That's a perfectly valid thought and without using Agile technical practices a likely outcome. User stories work best when used as part of a larger Agile approach that includes the concepts of Incremental Architecture and Agile Technical Practices such as Emergent Design, Test Driven Development, Continuous Integration, DevOps, and Refactoring to Patterns.

User Story Basics

A user story is simply a description of functionality that will provide value to a user. The reason for focusing on value is simple: if we don't know what the value for something is, why should we do it? We should only work on functionality that provides value because providing value is the whole point of software in the first place.

In traditional software development, we use multiple forms of documentation at each stage of development starting with conversations with end users. We then translate those conversations into business requirements. Those requirements are then translated into technical specifications which are then translated into tasks that individuals implement which eventually becomes working software. The exact steps taken vary from organization to organization, but at each of these stages the original intent can be lost in translation. Of course, we always try to clarify information by checking with the author(s)

of the documentation from the previous stage, but as time goes by, it becomes harder and harder to reconstruct any missing or mistranslated information.

To the extent that it is possible, it is best to communicate information face-to-face, implement what is needed as quickly as possible, and get feedback on the result as quickly as possible. User stories help to serve this purpose by encapsulating user intent in a simple form that everyone involved in implementing it can understand and use as a double-check during implementation and as sufficient documentation after implementation.

A user story removes invisible constraints by focusing on the outcome desired by the user. The technical team doing the work will see the user story, will be able to better understand what the user needs, and will be able to participate in or even own the specification and design of that story. User stories provide engineers more freedom to utilize their creativity and ability to innovate without the risk of implementing something that the user doesn't want.

This same freedom applies to all parties involved in the delivery of the user story. For instance, when a QA person looks at the story, they can focus on testing that the user's desired outcome has been achieved, rather than just testing to see that a set of requirements have been met. In general, user stories help to separate business value from implementation and focus all parties on the desired outcome.

The Headline of a User Story

The main part of a user story is the headline. This is also referred to as the title or short description. The headline is the most important part of a user story. The headline consists of 3 parts: a who, a what, and a why. You can add whatever additional details that you need in order to implement a user story, but they should not be in the headline of the story and ideally you should defer adding any details until after you have a good headline. Without a good headline, none of the additional details matter.

Here is an example user story:

> *As a **movie goer** I want to know **what movies are playing near me** so I can **decide if I want to go see one***

The exact format of a user story is not important as long as it has a who, what, and why. The "who" in the example is the person that wants to go see a movie. The "what" is determining what movies are playing near the movie goer. The "why" is so that the movie goer can make a decision based on the results. Notice that the story doesn't seem to have much to do with software other than maybe producing a list of movies that are playing nearby.

An obvious question at this point is “but you haven’t said how this should be implemented.” And that is part of the point. We want to defer the “how” of implementation for as long as we possibly can. Here are a few reasons why:

- The story may be created long before it is implemented. In the intervening time, the way to implement it may change.
- If you never implement the story any effort to elaborate it is a waste of time and money.

The headline should not contain technical jargon or refer to implementation details. It must be understandable to all parties involved in the use and delivery of the software including but not limited to: user, Product Owner, designer, developer, tester, DBA, and user documentation writer.

In summary, there is no need to discuss how you will implement a story until the time to implement it gets closer. If you are just starting to define your product and haven’t decided *what* to build, then any discussion of *how* to build it is premature. By the same token, if you haven’t decide what is valuable, then it is premature to decide what to build.

The User (The Who)

The user mentioned in a user story must be actual users of the system with very few exceptions. The users should not be “developers” or “testers” or “the architect” or anybody else other than somebody that will be using the system. Some exceptions would be when the software is a tool designed to be used by developers or you are adding new functionality to the software to make it easier for the testers to test it.

The reason for mentioning a specific kind of user is so that we can put ourselves in the user’s shoes and so that we know what kind of user to talk to if we have any questions about a particular user story. Also, if you don’t know who the story is for, how can you be sure that there is value in implementing the story?

The “What” of a Story

The “what” of a story is simply the functionality to be implemented. This is the sort of thing that we are used to. However, in a user story the description of the functionality must be in language that the user understands and the functionality must have value independent of any other user story when it is implemented.

The “Why” of a Story

In the context of a User Story, the “why” is really the goal of the story. It is the reason that the user wants the functionality. Nobody actually wants a particular piece of functionality for its own sake, they want the functionality for a particular purpose. Consider our example story.

> *As a **movie goer** I want to know **what movies are playing near me** so I can **decide if I want to go see one***

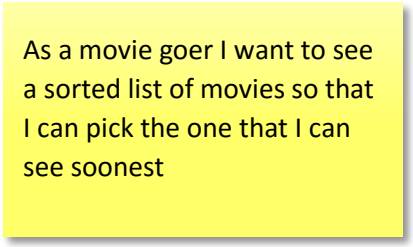
The user doesn't actually care what movies are playing near them and they don't care if the functionality provides a list or a map or anything else. Their end goal is to decide on a movie to go see. By understanding the user's end goal, we have a better idea if we are providing functionality that will help the end user achieve their goal.

Additional User Story Details

In addition to the headline of the user story you can record any other information that you feel is important. For instance, when using an Agile Project Management tool, typical information beyond the headline includes: acceptance criteria, who created the story, an estimate, a longer description or context around why the story is important, etc. During the implementation of the story it might contain additional information such as who is currently working on it, links to test cases, information on how it was implemented, and other information that is important to record for posterity. The general rule of thumb for user stories though is "less is more."

Acceptance Criteria

Acceptance criteria provide information to let you know when a story is done. They answer the question "I will know that this story is done when the user can…?" Here is an example:



Acceptance Criteria:

- Sort the movies by distance from the movie goer
- Sort the movies by minutes until show time

I.N.V.E.S.T – a Useful Acronym for Writing Good User Stories

Bill Wake coined a wonderful acronym for remembering how to write optimum user stories: "I.N.V.E.S.T." . It stands for: Independent, Negotiable, Estimatable, Small, Testable. These guidelines are primarily focused on the headline of a user story.

Independent – ideally, a user story does not depend on any other user story and user stories can be implemented in any order. This is not always possible, but should always be the goal.

Negotiable – this is to remember to keep information about how to implement a story out of the story for as long as possible, ideally until it is implemented. The headline itself should never contain information about how to implement a story, thus keeping the user intent intact regardless of any proposed implementation.

Valuable – if it isn't clear what the value to a specific user is, it isn't a good user story.

Estimatable – if you don't feel like there is enough information to estimate a story, it is probably either too big or too vague. Try splitting it and make sure there are good acceptance criteria.

Small – the best stories are small. Small user stories are more likely to be clearly understood by all involved.

Testable – if you aren't sure how to test the story, you probably don't know how the new functionality behaves. If you don't know how it behaves then it is unlikely that the user's intent will be implemented and thus it probably isn't a good user story. A good way to see if a story is testable is to make sure it has good acceptance criteria.

User Stories are not Tasks and Tasks are not User Stories

A task is simply a chunk of work to be done. Any work that anybody does can be described as a task. Creating a short description of a task and calling it a story does not make it a story. Only work that meets the INVEST criteria can be considered a user story.

User Stories in Complex Situations: Multi-Team, Multi-Tier, Multi-Component

Writing user stories for software that is written by a single team that has no dependencies has a certain level of complexity. It is generally harder to write stories for software that is written by multiple interdependent teams where different teams are responsible for different parts of the architecture. This is especially challenging if those teams integrate their work infrequently.

Consider the organization that might exist for the movie planning app. It might have a User Interface team, a business logic team, a component team responsible for location services, a component team responsible for gathering information and updates from theater information providers, and a back-end team responsible for information storage and retrieval.

A typical reason for having separate teams is that the demand for new functionality outpaced the ability of the initial team, more people were hired, and the team had to be split up somehow. Splitting into component teams is a very common pre-Agile solution.

Each team can conceivably implement valid user stories. For instance, a new UI-only feature, a new feature that uses existing UI elements, more accurate location determination, more timely movie information, and faster information storage and retrieval. But what about a story that requires new functionality from each team?

The main challenge in this environment is to write stories that are valid stories and not just tasks. There are at least five solutions to this problem: restructure the teams, implement a temporary solution to remove the dependencies, break the story into per-team tasks and coordinate implementing them in parallel, split the stories, and as a last resort do whatever you need to do to get the work done.

Often, stories can be split into smaller stories which are each valid stories but have fewer or no dependencies between teams. This is the best solution when it can be applied. There are many story splitting techniques and they will be covered in a later section. Let's take a look at the other options next.

Restructuring into User Story Teams (aka Feature Teams)

One solution to the problem of stories that require work from multiple teams is to restructure teams from architectural layer based or component based teams to true cross-functional teams that are able to work on stories with few if any dependencies on other teams. That is, not just cross functional in terms of skillsets, but also cross-functional in experience across architectural layers and components. With deeply cross-functional teams, each team can implement fully releasable functionality independent of other teams.

In the example of the movie-planning app, one approach would be to re-distribute people such that you have teams that have a combination of people that collectively have skills and domain knowledge across all of the following: User Interface, business logic, location services, working with movie and theater information providers, and information storage and retrieval. Of course, this may create a new set of challenges.

One question that naturally arises is, "how do you prevent teams from creating multiple solutions to the same problem, one solution per team?" No matter how you organize it, there will be some challenges. The question is, where do you want to experience those challenges? Do you want to take on the challenge of coordinating the work of stories that involve multiple teams or do you want to take on the challenge of keeping teams in architectural alignment? That decision depends on your exact circumstances and is a decision that only your organization can make.

Using a Temporary Solution to Remove or Reduce Dependencies

Sometimes, the time it will take for another team to implement something that you depend on is too long, and the need to deliver value now is too high to wait. In that case, it often makes sense to consider a temporary solution. Implementing a temporary solution runs the risk of creating technical debt, but sometimes it is worth it.

Perhaps a particular api returns thousands of results and it is the only way to get the necessary information. The UI team wants to filter out some information and the back-end team is too busy to create a new filtering option for the api. So, the UI team does the filtering themselves on the results returned by the existing api call.

This is an inefficient long term solution, but it provides immediate user value and removes the dependency on the other team. If you use this approach, make sure to add the temporary solution to a list of technical debts so that it isn't forgotten.

Coordinating the Work of Multiple Teams to Complete a Story

It is generally straight-forward to break a story that requires the work of multiple teams into per-team tasks. It is perfectly valid to implement a story via per-team tasks as long as all of the tasks for the story are done within the same timeframe and integrated immediately. The goal is to implement all of the tasks as a unit as though a single team did all of the work as a single story and got it from started to done as fast as possible.

If this approach is used frequently, it may be time to consider restructuring the teams.

User Story Splitting

Sometimes stories are too big. Getting a feel for when a story is too big takes experience, but at some point you will wish that you could somehow make a story smaller. The only way to make a story smaller is to remove some of the functionality or divide the story into smaller stories. In both cases this means splitting the original story. If you can remove functionality from a story and still have something that meets the criteria of a valid story, then you have made your life simpler. However, it is safer to split a story into new user stories. After all, the functionality that you are considering removing may still have value. By splitting one story into two or more stories that are all still valid user stories, you can defer the decision about what to do about any individual story.

When you are mostly working on small user stories that have end user value, you are reducing the chance that you are putting something into the product that never gets used and you are also reducing the chance of starting work on something that never gets finished or has to be discontinued part of the way through. The smaller your stories, the smaller your risk and the less effort you've wasted when you run into problems.

Another great result of splitting user stories is that you may find that most of the value of a story is associated with functionality that doesn't require much effort to implement. In that case, you have effectively reduced the cost of the story with very little effort.

> *"Simplicity -- the art of maximizing the amount of work not done – is essential."*
> **Agile Manifesto**

User Story Integrity

Sometimes it is tempting to think of splitting a story in such a way as to reduce the technical quality of the result. That is not the intention of splitting stories. It is important to always be thinking of producing valid stories that meet the INVEST criteria. For example, a "story" that is just development but no testing is not a valid story and a "story" that is testing a development-only story is not a valid story either. A story is only a story if it adds functionality to an existing shippable version of the software and results in a new shippable version of the software that somebody agrees has new value.

Splitting by Acceptance Criteria

One of the simplest methods for splitting user stories is to split by Acceptance Criteria. Often, the Acceptance Criteria spell out individual pieces of value that guide us to new user stories. Consider the following User Story and its Acceptance Criteria:

As a movie goer I want to see a sorted list of movies so that I can pick the one that I can see soonest

Acceptance Criteria:

- Sort the movies by distance from the movie goer
- Sort the movies by minutes until show time

From this we can construct two new stories:

As a movie goer I want to see a list of movies sorted by distance so that I can pick the closest one

As a movie goer I want to see a list of movies sorted by minutes until show time so that I can see the one that is coming up sooner

Splitting by User

Different kinds of users have different kinds of needs. In general, some users have more needs than others and some users are a larger source of business value than others. Splitting stories by user allows us to focus our investment of resources on specific user or market segments and potentially start receiving a return on our investment sooner. It also allows us to start receiving feedback sooner and make any necessary changes in tactics or strategy. In any case, splitting a user story by user gives us more options. Consider the story:

As a movie goer I want to see a list of movies sorted by distance so that I can pick the closest one

There are at least three kinds of movie-goer when it comes to distances: pedestrians, public transit riders, and drivers. The time it takes a movie-goer to get to a movie depends on which kind of transportation they will be using. If we focus on the pedestrian we can narrow the scope of the story to:

As a movie goer on foot I want to see a list of movies within ¼ of a mile from me sorted by distance so that I can pick the closest one

To implement this story, you don't have to provide a set of distance options, you only need to hard-code the distance to ¼ of a mile. Once you have that working, you can add on other stories that provide more options later.

Splitting by Items in a List

There are many ways to break things down into lists and there are many ways to describe a list. Consider this story:

> As a movie goer I want to pay for my movie via credit card or reserve it and pay in person to guarantee a seat

In this user story, there is a fairly straight-forward list of payment methods: "credit card or pay in person". There are also two things to notice about this list, it isn't as specific as it could be and it can also be made more general in order to discover more potential value for the user. The words "credit card" can be made more specific, such as "Master Card, Visa, or Discover." Also, the more general version of this list is "payment method" which has many more items in its list than just credit cards and paying in person.

By being as specific as possible we can make the implementation effort required for the story smaller and get to a new working version of the software that much faster. Here is a new smaller version of the story:

> As a movie goer I want to pay for my movie using Master Card to guarantee a seat

There is at least one more list hidden in this story. If the user is running the app on their smart phone, which smart phone is it? There are many options: iPhone, Android, Blackberry, Windows, etc.

> As a movie goer I want to pay for my movie on my Windows Phone using Master Card to guarantee a seat

Splitting by Create/Read/Update/Delete or the Word "Manage"

If you need to manage something, you generally need to be able to create, view (read), update, (edit), and delete it. There are subsets of these operations that provide value. For example, being only able to create and view items can be useful all by itself. Also, updating items can be skipped initially in favor of deleting and re-creating. Certainly, skipping the ability to edit initially is annoying to the user, but even if you implement the edit functionality prior to delivering to the user, implementing piece by piece will produce a better overall design and higher quality.

Original User Story:

As a movie goer I want to keep my credit cards on file so I can pay right away

New User Stories:

As a movie goer I want to create and view a list of my credit cards so that I can pay right away

As a movie goer I want to delete a credit card so that I can remove cards I no longer use

As a movie goer I want to edit information about a credit card to correct mistakes

Splitting by Keyword

An easy way to spot a list in a user story is to look for the following keywords or symbols: “and”, “or”, commas, semicolons, “using”, or “with” .

Original User Story:

As a movie goer I want to see movie details, previews, and reviews so that I can decide which movie to see

New User Stories:

As a movie goer I want to see movie details so that I can decide which movie to see

As a movie goer I want to see previews so that I can decide which movie to see

As a movie goer I want to see reviews so that I can decide which movie to see

Splitting with Lists

Splitting by keyword is actually a simple version of splitting by lists. In the case of the keywords, the list is already provided in the user story. In general, you can think of categories related to functionality that can be broken down into a list. For the movie goer app, here are some example categories that can be broken down into lists.

- Transportation
    - Pedestrian
    - Transit rider
    - Driver
- Device
    - iPhone
    - Android
    - Windows Phone
    - Web (HTML5)
- Interaction / display
    - Simple listing
    - map-based
    - sophisticated interactive interface
    - voice-based
- Location determination
    - User provided
    - GPS
    - Interactive map
- Distance determination
    - Built-in default
    - Stored user preference
    - User provided

Here are two examples of using the lists above to split a story.

Original User Story:

> As a movie goer I want to know what movies are near me so that I can decide what to see

New User Stories:

> As a an iPhone using movie goer on foot I want to see a simple listing of movies that are a default distance away from a location I provide so that I can decide what to see

> As an in-car Windows Phone user I want to see a map-based display of movies within my preferred distance from my current GPS position so I can decide what to see

These stories are getting a bit long. It is good to be able to have shorter headlines so that you can see a listing of stories in a backlog and easily re-order them. It is perfectly ok to abbreviate them as long as there is a clear who, what, and why.

Shortened Story headline:

> Windows Phone in car, map, pref distance, cur GPS pos, decide movie

Within the body of the story you can provide the full details.

Splitting by Test Scenario

One of the keys to creating small stories that still have value to the user is to be more specific. What could be more specific than a test scenario? In order for a test to be useful there has to be a specific set of steps to execute and a specific expected result from the test. By thinking of test scenarios, you can find new ways to split stories.

Let's say you are testing a new greeting card app.

Original User Story:

> As a user I want to send a greeting card to show that I care

This is pretty much the same as saying "I want to create a greeting card app." By thinking of a specific test case, you can create more user stories. Usually, to create a test case you already have requirements or user stories, or an existing application that you want to test and a given user interface. In this case, you can imagine any user interface you want, the desired end result is user stories, not an actual test case. That said, you may also end up with some useful test cases.

Test case

Open app

Log in

Go to "send a card" section

Select the Valentine's Day category

Select the basic "Will you be my Valentine" card (card #1234)

Customize the card with "Can't wait to see you on Friday"

Enter our test recipient name

Select Credit Card as payment method

Enter details from test Master Card #1

Select FedEx overnight delivery

Enter test address #1 (mailbox in our building)

Expected result

Card #1234 with test name shows up in test mailbox #1 within 1 day

User Story:

> User wants to send a Valentine's day card via FedEx overnight and pay using Master Card to show that they care

This is a much smaller chunk of functionality than implementing the whole greeting card application and we can now also see categories we can use to generate lists:

- Holidays
- Various Valentine's day cards
- Ways to customize a card
- Shipping method
- Payment method

Hard Coding

Hard coding is a tried and true technique for getting a small piece of functionality working on a path towards getting a larger piece of functionality ready. Hard coding can also be applied to user stories with the same result. Consider the previous greeting card example. Implementing that story is less work than implementing the whole greeting card application, but it still requires functionality for selecting a card, customizing it, selecting a payment method, and selecting a delivery method. Hard coding as much as possible creates a much smaller story which can be used as the starting point for adding the rest of the functionality.

> User wants to send card #1234 with no message to test recipient #1 using test Master Card #1 using FedEx overnight delivery to show that they care

The entire user interface for this can simply be a big button labeled "Send Card". Now you have implemented a fair amount of underlying functionality, can test it end-to-end, and have many options for what to add next. For instance, you could do this story next:

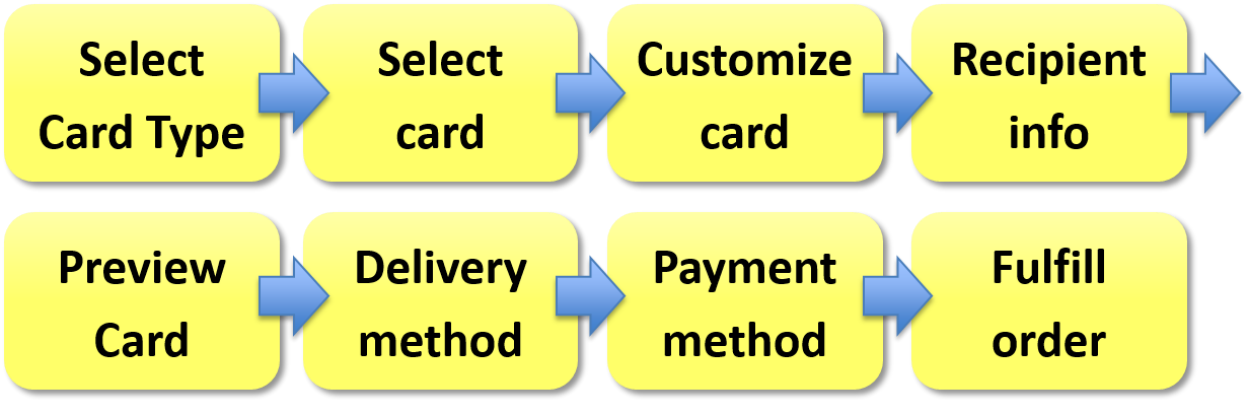
> User wants to send card #1234 with no message using Master Card using FedEx overnight delivery to show that they care

Now you only need to add the functionality to take the credit card, billing information, and recipient name and address and anybody can send a specific Valentine's Day card at the last minute!

Splitting by Workflow – an Anti-Pattern that can be exploited

One technique that is recommended in some places is splitting by workflow. While this technique is actually an anti-pattern, it can be exploited to produce a positive result. First, let's look at the technique itself.

In the greeting card example, there is a workflow which is easy to map out. Since there are many steps, it is tempting to think of each step as a story. By breaking the work down into individual steps, perhaps this is a good way to work in small chunks.



There are many steps here, so perhaps this breaks down conveniently into iterations. It looks like a good idea on the surface. The problem is that none of these "stories" are useful until the "fulfill order" story is complete. Essentially, we still have one huge story, "Greeting card app" that breaks down into 8 sub-tasks which are each fairly large on their own.

Although the split-by-workflow-steps technique is not useful on its own, it does set the stage for another technique called "cake slicing." Wherever you see a workflow, feel free to break it down into workflow steps, but then apply the cake slicing technique.

Cake Slicing

The cake slicing technique applies whenever you have something that can be described in "layers" or stages which are only valuable when you have all of the layers or stages. Cake slicing is similar to the split-by-test-scenario technique. An example of stages is the workflow steps from the split-by-workflow anti-pattern. An example of layers is a multi-layer architecture. This section will refer to both stages and layers as layers.

This technique is called "cake slicing" because it is just like slicing a cake. Each slice of cake (or pie if you prefer) is valuable to the user (aka cake-eater) whereas most people would not really want a layer of cake.

The first step is to take each of the layers to be implemented and to create as many sub-tasks as you can. As an example, the greeting card workflow might break down into the following 24 tasks:

Select card type	UI for card type selection	Get card types from db	Create card type artwork	
Select specific card	UI for card selection	Get card data from db	Create card artwork	
Customize card	Textbox for card message	Choose color scheme	Add additional images	
Recipient info	Textbox for name	Collect recipient e-mail address	Textbox for address	
Create / preview card	UI to show card preview	Generate card		
Delivery options	UI for delivery options	Integrate with UPS	Integrate with Fedex	Integrate with USPO
Payment options	UI for payment options	Integrate with MC	Integrate with Visa	Integrate with Amex
Fulfill order	Place card order with partner	Send e-mail version of card		

The next step is to find a slice of functionality that provides value to some user. For each layer, see if you can do any of the following:

1. Skip it
2. Hardcode it
3. Choose a small number of tasks to implement

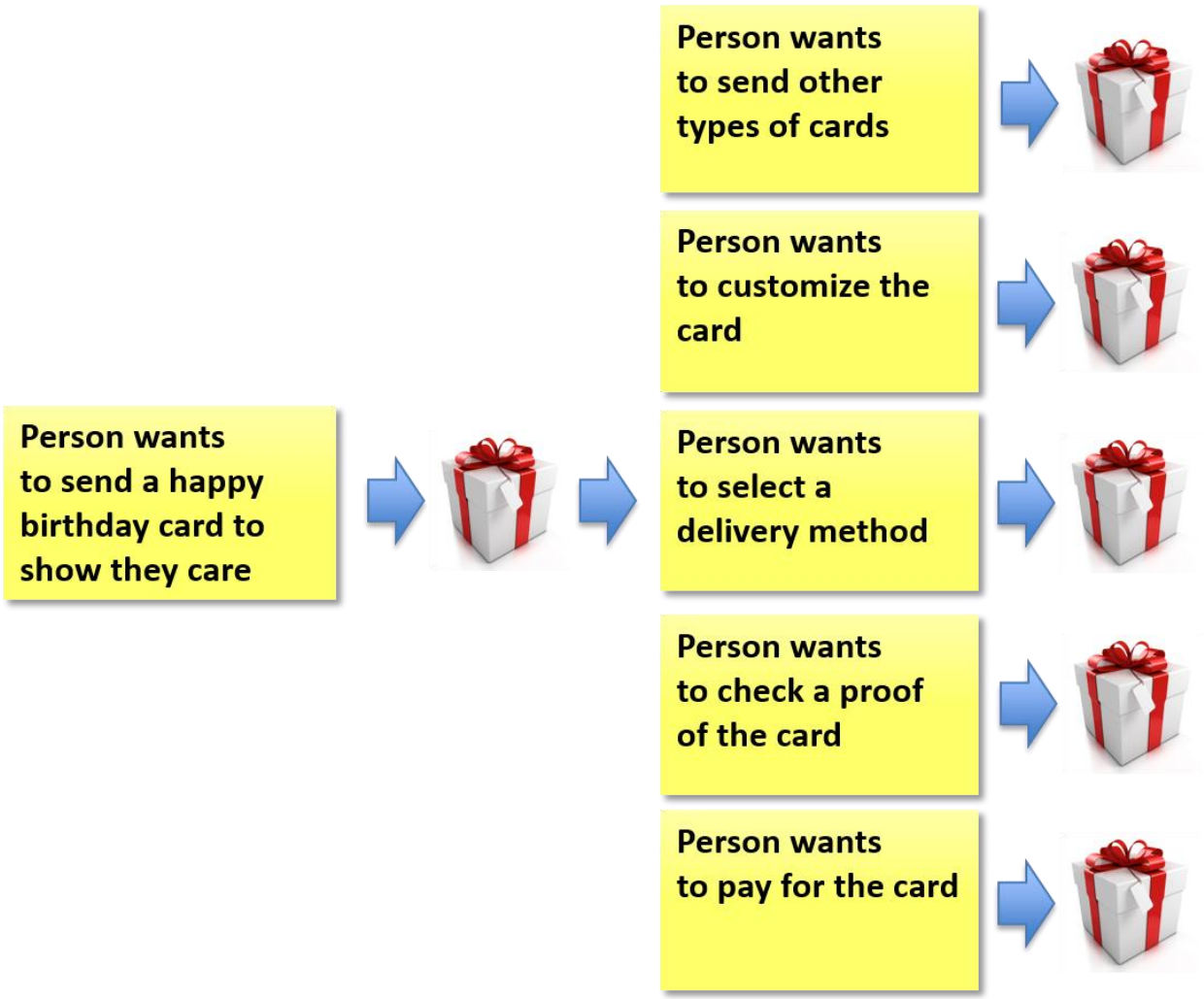
Select card type	UI for card type selection	Get card types from db	Create card type artwork	
Select specific card	UI for card selection	Get card data from db	Create card artwork	Set card as HAPPY_BDAY_1
Customize card	Textbox for card message	Choose color scheme	Add additional images	
Customize card	Textbox for name	Collect recipient e-mail address	Textbox for address	
Create / preview card	UI to show card preview	Generate card		
Delivery options	UI for delivery options	Integrate with UPS	Integrate with Fedex	Integrate with USPO
Payment options	UI for payment options	Integrate with MC	Integrate with Visa	Integrate with Amex
Fulfill order	Place card order with partner	Send e-mail version of card		

In this example, 4 layers are skipped, one layer is hard-coded, and 3 layers have a sub-set of tasks selected. The result is a small new story which other stories can be built on, similar to but even smaller than the Valentine's day card example:

Person wants to send a happy birthday card to show they care

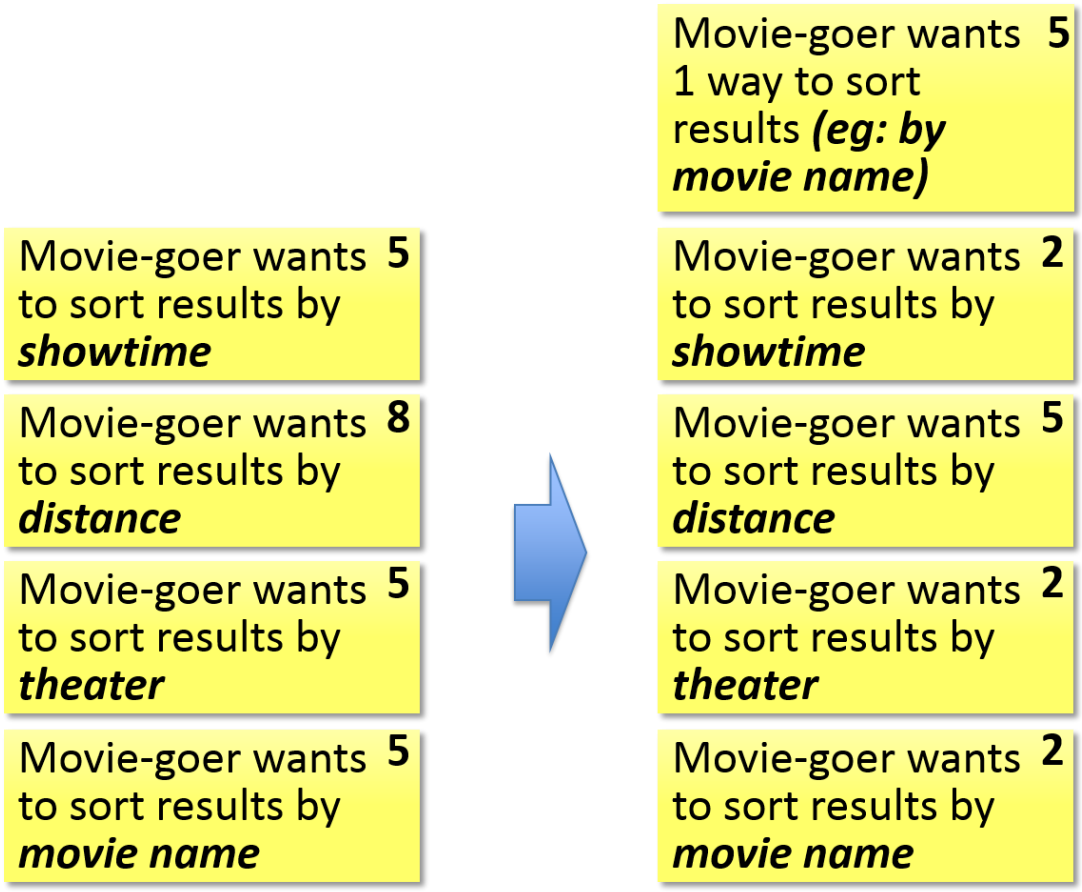
Set card as HAPPY_BDAY_1	Collect recipient e-mail address
Textbox for name	Generate card
Send e-mail version of card	

The birthday card story serves several purposes: it allows you to create some infrastructure for future stories while still being end-to-end testable, it provides immediate value, and it allows for early feedback. It also allows you to create many new user stories suggested by the workflow steps which are now independent stories that can be built on top of the birthday card story. In the illustration below, the "stories" on the right are really just place-holder Epics that can be further broken down into smaller user stories.



Creating Independent Stories

The cake slicing example provided one way to create independent stories. Here's one more example. When first developing the movie app, there is a point at which there is not yet any support for sorting. Let's say there is a desire to sort by showtime, distance, theater, and movie name. Because there is not yet any support for sorting, these stories all depend on sorting. Estimating the stories is difficult because whichever one is done first will include initial support for sorting and all of the rest can then build on that which means they will require less work.



One way to isolate the dependency on sorting is to create a placeholder story that represents doing one of the stories with a suggestion of which one it will be. That way, you can assume for the rest of the stories that basic sorting exists and estimate accordingly. If you change which kind of sorting is done first, no big deal, just change the placeholder story and nothing else has to change.

About the Author

Damon Poole is Chief Agilist of Eliassen Group's Agile practice. His 23 years of software experience spans from small co-located teams all the way up to global development organizations with hundreds of teams. Damon is a past President and Vice President of Agile New England. He writes frequently on the topic of Agile development, is the author of the web book "Do It Yourself Agile," and a pioneer in the area of Multistage Continuous Integration and mixing Scrum and Kanban. Damon has spoken at numerous conferences including Agile and Beyond 2010-2012, Agile Business Conference, Q-Con, Agile 2008-2013, and Agile Development Practices and has trained thousands of people on Agile techniques. He is also a founder and past CEO and CTO of AccuRev where he created multiple Jolt Award winning products including AccuRev and AccuWorkflow.