In the last reading, we learned about user stories as well as how to create them.

In this reading we are going to learn:

- How to properly break down user stories and add additional details.
- How to use User Stories for software estimation.
- How to take a big collection of user stories and use it to define the product backlog.
- How to plan a software release.

Breaking Down User Stories And Adding Additional Details

While reading this section, keep in mind that it is not expected that you go and add as much detail as you can to your user stories right now. This section is about how you can incorporate new information you are learning about your user stories as you discuss them with your team.

Many people have the misconception that the written user story is the most important thing, when in reality it is intended to be a conversational aide to get customers and developers talking about how a user story can be accomplished within the system.

“User stories get their name from how they are supposed to be used, not what you’re trying to write down… If you’re not getting together to have rich discussions about your stories, then you’re not really using stories.” - Jeff Patton, *User Story Mapping.*

Many people think that using user stories means that you write down all the things the user can do in these little sentences and that you are now finished doing the requirements for the system and can move on to more important tasks. This is not the case at all.

The written user story serves a placeholder representation of the requirements of an aspect of the system. During the planning phase, this is usually sufficient to estimate how long this aspect is going to take, and what the priority of this aspect is in comparison to the other user stories.

However, as conversations about this aspect of the system take place, as well as conversations about other user stories, more information is revealed about how this user story is going to work. The user story is sometimes broken down into smaller pieces, but when the card is small enough, this additional information is then added in the form of acceptance criteria which are tests to determine whether the system currently meets the needs of the user.

What should be in the Card?

A brief summary of the action the user is taking to get value from the system.

If it is not clear what value is being gained the card should be rewritten.

For example in a shopping app the following is clear enough:

```
The user can pay for items.
```

Whereas this is not:

```
The user may enter their name.
```

When it comes to splitting stories remember the criteria for good stories established in the last reading:

- Independent
- Negotiable
- Valuable to users or customers
- Estimable
- Small
- Testable

If a story can be split into multiple stories that match those criteria, it is probably beneficial to do so.

What should be in Acceptance Tests?

Alternate, overlapping/dependent ways of accomplishing the same action to get the same value.

`User can pay with credit cards.` might have acceptance tests for Visa, MasterCard, and Discover, but these should not be different user stories because their processing is typically all the same.

Ways in which the user story can go wrong:

`The user is informed when their payment is rejected.` is a good acceptance test but a bad user story, because the user is not performing an action to receive value.

Ways in which the story can go right:

Consider the following statements:

```
Player can turn left.
Player can move backward.
```

Both may be valid acceptance tests for some sort of player movement user story, but if there are a great number of this type of acceptance tests or achieving some are much more complicated than others, this is a good indicator that the story might need splitting.

What should not be on the Card or Acceptance Tests?

Technical and user interface details make your user stories quickly become out of sync with the actual system. Big picture technical and user interface details are established as tasks during the sprint planning phase, and further details are understood through discussion over the course of the sprint.

Technical Details

Do not include technical details such as Object names, Database Table names, or Function names in your Card or your acceptance tests.

`The data the user entered has been stored in the hamburger table in DynamoDB.` is a bad acceptance test because it provides details about technical aspects of the system.

`The data the user entered is persisted even when the application is closed.` is a much better test that accomplishes testing the same value without technical details.

User Interface Details

An example of an acceptable amount of user interface details is,
`User is able to choose a show to watch.` While an example of too much detail is,
`User can select a show to watch via the drop down menu on the home screen.`

An Example of Breaking Up a User Story and Adding Acceptance Tests

A User Story might originally consist of:

```
The customer can pay for the items in their order.
```

During the planning stage, a rough estimation might take place as to how long a robust payment system might take to implement.

The card does not have any information added until that story begins to be considered for inclusion in active development.

It is at this point that this card is really discussed by the team and the product owner, breaking down exactly what is meant. This is the Conversation phase, the most important part, and the entire point, of using User Stories.

The product owner might indicate that they need the system to accept PayPal, Visa, or MasterCard. If these options were different enough to be a lot of separate work, these three things would either become their own stories in the form of:

```
The customer can pay for the items in their order using PayPal.
The customer can pay for the items in their order using MasterCard or Visa cards.
```

At this point the priorities of these items might be re-evaluated and it turns out that while PayPal is not super important to the customer, accepting credit information is. Therefore only the MasterCard and Visa Story is going to make it into the next sprint and the PayPal Story can be discussed at a different time.

This means that the MasterCard and Visa Story still needs further discussion, and so the Conversation stage is still not over.

The product owner might indicate that not only must the system process valid cards, it also has to reject invalid card information.

This doesn't mean that there should be a User Story that says

```
A user can have their credit card rejected when trying to pay for their order.
```

The user is not getting value by being rejected, this is an exception, or something going wrong, to the case that they are trying to check out.

Therefore this information is encoded into `The customer can pay for the items in the shopping cart using MasterCard or Visa.` via an acceptance test. This means that added to this story is:

```
The customer is rejected when invalid card information is provided.
```

Each time this feature is tested, invalid card information should now be provided, and it should be indicated whether the system rejected it or not. When the system has rejected the invalid card, PASS should be written

on this acceptance test, and if accepted, FAIL should be written instead.

In fact, an additional test should be added:

```
The system is able to accept valid card information.
```

If this test fails, the feature is considerably more broken than when the other test is failing, and so this test should be given a higher priority than the previous one.

Upon further discussion, the product owner mentions that when the shopping cart is empty, the purchase option should not be available.

Again, doesn't mean that there should be a User Story that reads:

```
A customer can not checkout an empty order.
```

There would be no value to the customer in trying to do this. Instead this is another acceptance test to be added.

However, this test seems to apply to both the stories we just created:

```
The customer can pay for the items in the cart using PayPal.
The customer can pay for the items in the cart using MasterCard or Visa cards.
```

Adding it to both stories is an option, or if similar things that have to do with the checkout process rather than the particular payment method an additional story that encapsulates only beginning the checkout process can be added. For instance,

```
A customer with items can begin checkout.
```

And then add the empty order acceptance test here.

In Summary

Keep this process in mind as you discuss your user stories throughout the estimation, release planning, and sprint planning phases. Think of how stories might be broken up, write down any new acceptance tests that come up, and be ready to reorganize the structure of your stories. User Stories are meant to constantly evolve as your understanding of your project grows.

We are now ready to talk about using these User Stories for further discussion and planning of your software project.

Using User Stories For Estimation

In order to plan products to fit within certain timeframes, estimate are needed for how long the parts of the system are going to take to be able to make.

As a whole, everyone is very bad at estimating when a software project is going to be done. It is extremely difficult to look at the vision for a project, or even all of its requirements or user stories, and provide an accurate estimate of when it will be completed and ready for release.

However user stories can be used to provide an initial general idea of how long the system will take, and as more insight is gained, further refine that update to be more and more accurate as the system progresses.

According to Cohn, the best approach to estimating stories would be one that:

- Allows us to change our mind whenever we have new information about a story.
- Works for both epics (big stories) and smaller stories.
- Doesn't take a lot of time.
- Provides useful information about our progress and the amount of work remaining.
- Is tolerant of imprecision in the estimates.
- Can be used to plan releases.

Story points are an approach to User Story estimation that tries to satisfy all of these goals.

Story points represent a unit of time, but the amount of time it represents is up to the team. Some teams have 1 story point represent 1 week of work, others have it represent a day of work.

Days of work are best suited to this class, because most of your app stories will be relatively simple and not require more than 1 week of work. Using hours of work would make you try to be more precise than is intended by this method. Days of work is a nice middle ground of not too precise and not too imprecise.

In addition, notice the term "days of work" rather than just "days". This is because when one of you are assigned to a story on a Monday that is expected to take "1 day of work", it is unlikely that means you will have it done on Tuesday or even Wednesday. You have classes and jobs and social lives. This unit of "day of work" is meant to capture the idea that you spent all of your day working on that story. In reality, you might be able to put in a quarter of a day's work per school day (sometimes) or a half or full day's work on a weekend (sometimes). We are estimating only how many full days of work it would take to complete the story, not giving it a due date. This allows the estimate of a task to stay the same regardless of the day of the work or the persons schedule to which it is assigned.

Story estimates should not be assigned to be done by individual team members. Stories will be broken down into tasks later which will be estimated by individual members, but at the story level estimates should be done

as a group. This will ensure that group members get more familiarity with all of the stories and that the story estimates benefit from having a variety of opinions.

How To Make The Estimates As A Group

Cohn suggests the following approach:

1. Have everyone sit together with the story cards in front of you. Have with you a stack of additional physical cards. (Even though you are using Trello) Each member should get a handful of blank cards.
2. Select an existing story at random. As a group, figure out how the story is going to work just enough to be able to come up with a rough estimate. If there are too many details unknown to come up with an estimate then defer estimating the story until more information is obtained, following the suggestions in the "Estimable" section in the previous reading.
3. When the team agrees they have enough information, each group member writes their estimate in story points on the card and does not show their estimate to anyone else. When estimating a story, include everything needed to complete the story, including research, testing, design or anything else that needs to be done to complete the whole story.
4. When everyone has finished writing an estimate, the members turn over their cards and hold them up so everyone can see.
5. It is very likely at this point that the estimates will significantly differ. This is a good thing, as it means people have different understandings of the story. If this is the case, have the high and low estimators explain their estimates. This should not come off as attacks, but rather everyone attempting to understand the differing perspectives.
6. At this point, everyone discusses for a few minutes. Anyone might have an opinion on the high or low estimates.
7. Next, the voting process is repeated, with the previous steps occurring again if there is still a significant difference between estimates.
8. This process continues as long as estimates are moving closer together. This should not take much more than three tries. The numbers on everyone's estimates do not have to be exactly the same. For example, 5,5,5,6 would probably be fine to enter an estimate of 5. If the group cannot reach a consensus, that story might need to be estimated at a later time.
9. Make sure you write the agreed upon estimate into the tool you are using for tracking stories.
10. Repeat this process for as long as you have time to estimate or until you run out of stories.

Some Tips For This Process

As you estimate more stories, it will be easier to estimate others by comparing them relative to the stories you have estimated. You can imagine it is about equal to another task, or double, or half.

It is a natural part of this task to realize that stories need to be broken up or even that some are unnecessary. The stories are always intended to be able to changed, you do not ever need to “lock them in” or accept them as they are.

When dealing with a large story, there are many more things that need to be taken into account for an estimate. This causes large items to be far more accurate than small items. This might be a good time to break up the really big stories into just large ones. To simplify providing estimates on large values, the team may want to agree on predefined possible story point values such as:

0.5,1,2,3,5,8,12,20,30

I wouldn’t recommend having stories much larger than 10 days of work. These should be broken up.

Once estimates exist for all of the important user stories for the project, we are ready to use the User Stories along with their estimates in order to plan development.

Planning A Release:

A release is a version of the project that could be considered “finished”, rather than being a development build or an unpolished version. This release would be able to accomplish some subset of the user stories, and each feature and screen would be relatively polished.

The first question when planning releases is, how often should they happen?

A large software project will typically do best with a release every two to six months, with web or mobile applications possibly releasing even more frequently.

In this class, we have 4 sprints, including the 0th one, that take place over a roughly 3 month period. Therefore you should plan for one major “release" at the end of the course. Of course, you do not actually have to publicly release your project in this class, and you can simply think of the final demo as showing your final release to the world. A much more aggressive goal would be to have one release in the middle of the class and one at the end.

An equally important second question is, what should be in the release?

At this point you have a large list of user stories, many of which have been estimated. However, after doing some of the estimation, you may have realized by now that it is unlikely that every one of these user stories is going to be able to be accomplished by the end of this class. You also know that some of these stories are absolutely critical to the app, while others are just things that would be nice to have.

Given that there isn't enough time to do everything, you need to figure out what to focus on to make the final

release as good as possible.

Cohn suggests using a technique called MoSCoW rules. MoSCoW is an acronym for:

- Must have.
- Should have.
- Could have.
- Won't have this time.

The must haves are those stories that are absolutely critical to your app, without them there isn't a reason for it to exist. Should haves are features that are important, but that there are ways for the app to work without having. Could haves are features that can be left out of the release, but if you have extra time, they can be put in. Won't have this time are those that you would like to include but your group agrees they would need to come in a later release.

In *User Story Mapping*, Jeff Patton writes, "Many times I, and the teams I've worked with, have placed all our ideas about the perforce process into a map and have been overwhelmed by the amount of work we'd have if we created it all. It all seems important. But then we take a step back and think about the specific people who will use our product, and what they'll need to accomplish to be successful. We distill that into a sentence or two. Then we carve away everything we don't need, and we are shocked at how small our viable solution really is. It's magic."

Your one release should at least be the **Minimum Viable Product**, which Patton defines as "the smallest product release that successfully achieves its desired outcomes". You need to make sure you at least produce that, and then you are free to add stuff that makes it even better.

Once you have grouped your user stories into these categories, you need to figure out which ones need to be implemented first. The stories should be prioritized along the MoSCoW lines, meaning that a *must have* should always have a priority over a *should have*.

How to prioritize the stories within these groups?

A method is needed for determining what stories need to happen first. Some of the criteria by which to do this that Cohn uses are:

- The risk that the story can not be completed as desired.
- The impact the story will have on others if deferred.
- The desirability of the story to a broad base of users or customers.
- The desirability of a story to a small number of important users or customers.
- The relation of a story to other stories (zoom in, zoom out).

At this point, any stories that are going to be prioritized need to have been estimated.

The group will take into account both the estimation and the priority of the story to place all the stories in a sorted order that they feel maximizes the value that the product delivers. Keep the stories separated by MoSCoW categories.

At this point, the group is not considering whether all of these things can fit into a release, just using the estimate as part of the calculation of the value of the story.

If the group is having a lot of trouble finding the place for a story in the list, the story may need to be split up into smaller stories.

There has been a long-standing debate in Agile development whether the parts of the application that are the riskiest (might not work/you have a lot to learn to be able to implement) or the parts of the application that have the most value should be considered higher priority. As with many debates, the truth is probably in a mixture of the two. Make sure to account for risk in determining the value of each story.

You will also want to account for stories that will need to change the infrastructure of your project. An application that will need databases or networking should try to incorporate stories using those sooner rather than later.

Now that all the cards in the product backlog have been estimated and sorted, they are likely in a state where useful estimates about the project can be made.

At this point, sum up all of the story points in the backlog. You might want to keep the sums separate according to the MoSCoW rankings.

You now have a rough estimate for the number of days of work it will take to complete this project. This semester, there are 3 sprints each lasting 3 weeks (not counting Sprint 0), so you now have an estimate for the number of story points you would need to be able to finish per sprint in order to complete the backlog on time. You also can adjust this number by only including your must haves, your should haves, etc.

At this point, your team will only have a guess at what number of story points they will be able to complete during each sprint. This estimate of velocity will improve as you complete sprints. For now, you will have to come up with a number per sprint that your group feels tenuously comfortable to committing to. You might base this number on the sum of the must haves plus some additional amount of the should haves. If you’re feeling aggressive, add in some of the could haves.

For full time developers an initial velocity (number of story points per sprint) guess is typically around one third the actual number of days in the sprint. As students, your estimated velocity should be lower than that.

At this point, you have a velocity, for example 20 per sprint. Using your product backlog and team discussion,

take 20 (or whatever your estimated velocity is) points of the highest priority items and put them into a column for the first planned sprint. Then take another 20 points worth of cards and place them in a column for the next planned sprint, and so on until all four sprints are filled.

You now have a very rough idea of which application features will be delivered when.

Do not put too much faith in this plan, use it to set initial expectations but constantly reset them as new information about your project comes in.

Summary

In this reading we have covered:

- Breaking down your user stories and adding additional details as needed.
- Estimating the time that developing a user story will take.
- Using the User Stories and their estimates to prioritize our product backlog.
- Planning our software release using the prioritized product backlog.

In the next reading, we will cover how to use the user stories for properly planning your sprints as well as the purpose of an initial "Sprint 0”.