User Stories

As mentioned in the reading “Intro to Software Development Processes” requirements are intended to form a communication bridge, establishing both what the customers want a product to do and what the developers must do in order to implement those needs.

From the beginning of Software Development, this has been a problem. Customers often don’t know exactly what they want, and developers, analysts, and managers have struggled to translate customer needs into a cohesive product.

In plan driven approaches to requirements gathering, analysts try to make every decision about what the software is going to do up front, before development ever begins. This fails to take into account the changes that the customer’s go through, as well as all of the inaccuracy that tends to come with software prediction in general.

Instead of making all decisions up front, decisions should be made incrementally throughout the project process, incorporating new information as it is received. When using incremental development, users will see early versions of the product and change their minds about what they wanted as well as come up with new ideas.

In agile software development User Stories are used to bridge the communication gap between customers and developers by clearly stating the customer requirements in a language that can be understood by both groups.

What are User Stories?

User Stories describe something the user or purchaser is able to do that provides them some value.

User Stories are composed of three aspects:

- A written description of the story used for planning and as a reminder (Card)
- Conversations about the story that serve to flesh out the details of the story (Conversation)
- Tests that convey and document details that can be used to determine when a story is complete. (Confirmation)

People often mistake the Card as being the most important aspect of a User Story, because it is the most visible thing you write down.

However, the Card is only supposed to serve as a reminder for a conversation that to be had with the Product

Owner. While the card has the initial written record, details about the functionality are established through the Conversation and recorded as Acceptance Tests.

If this seems confusing, we will go into much more detail on the other parts of the User Story in a future lecture. For now, we are going to be concentrating on the statements on the Card.

The typical template for writing a User Story Card is:

```
As a < type of user >, I want < some goal > so that < some reason >.
```

It may be that the reason is obvious and can be left out, but in some cases the reason (or value provided to the customer) is important to include.

Here are some example Cards involving a made up Job Search Website from *User Stories Applied* by Mike Cohn:

```
A user can post her resume to the website.
A user can search for jobs.
A company can post new job openings.
A user can limit who can see her resume.
```

Each of these are simple sentences, framed around what the user is doing through using the system. These are appropriate for some high-level User Stories. These stories will probably need to broken down into smaller pieces before development begins, but they are still useful at this point for planning the project.

Some anti-examples, things that look like User Stories but are not, from *User Stories Applied*:

```
The software will be written in C++.
The program will connect to the database through a connection pool.
```

These requirements do not involve the user, and while they may be necessary requirements for the system, they are not User Stories. Instead they would need to be repackaged into other stories that do focus on the user getting system value.

How Do I Write Good User Stories?

According to Cohn, A good story is:

- Independent
- Negotiable
- Valuable to users or customers

- Estimable
- Small
- Testable.

Independent

Of course, stories are going to relate to each other as a sequence of events, this is not what is meant by User Stories being dependent on each other. Imagine the following User Stories:

```
A user can search for an item.
A user can place an item in their cart.
A user can pay for their cart.
```

Searching for an item and putting an item in a cart are actually independent. As long as the user can get to a page showing an item, the search functionality does not need to be built in order for someone to implement and test adding to the cart. However, you could consider checking out a cart to be dependent on putting an item in a cart.

That does not mean that the stories have to be dependent on each other though. Instead we can rewrite the third story as

```
A user can pay for items.
```

While eventually the cart functionality might need to be integrated, this separates the value the user is trying to get out of the system, being able to buy goods, from the cart functionality. This User Story can be partially accomplished much earlier by adding a pay button on each item page.

The resulting User Stories would be:

```
A user can search for an item.
A user can place an item in their cart.
A user can pay for items.
```

While these activities are all part of a sequence, they are still what Mike Cohn is calling “independent”.

Negotiable

A User Story is not meant to be full of details about how the task is to be accomplished. It is instead a reminder to have a discussion about how this feature will be accomplished.

The following User Story is acceptable:

```
The User is able to provide their birthday in order to receive free birthday coupons.
```

Whereas the following is not:

```
The User inputs their birthday on the User settings screen using the google date pick
er
in the upper left hand corner, this causes an email to be sent out 26 days before tha
t
date informing them that they will receive one free medium sized pizza with a choice
of
the following toppings...
```

Remember, customers are always going to be changing their minds. Which of the previous two User Stories is most likely to still be correct months after that was written?

Furthermore, having that level of precision makes the reader assume that the information is more accurate than it really is. Whereas the first User Story encourages the developer to have conversations about where the user should enter their birthday, and what kind of rewards should be received, the second User Story encourages the developer to sit down and do the coding without talking to anyone. This makes it far more likely that incorrect, out of date details make their way into the product.

Details that have been confirmed via discussion become acceptance tests, not stored on the forefront of the story. The story itself should remain lightweight and able to be grasped quickly for planning purposes

Valuable to users or customers

It is not true to say every story should be valuable to the direct users of the application. A User Story might have value to someone that is purchasing the system but is not the direct user.

For example a child video streaming app might have parental controls accessible not to the users of the app, the children, but rather the parents who purchased that app.

Similar situations arise in business applications when there are features that provide value to the IT department that configures the software but that are never seen by the software's actual users.

What you want to avoid however are User Stories that are really geared towards either the user interface designers or the programmers:

```
The user can select which movie they are wanting to see showtimes for by using the dr
op down menu.

The item purchase requests must be routed through the checkout lambda function.
```

It is not a good idea to tie requirements to the user interface. This is due to the fact that it will be extremely common during the early phases of app development to update and improve the user interface which would require updates to all User Stories. The interface types the user is using, what page they are doing it on, or anything to do with the look of the app, should not be included. These details should emerge from discussion, which possibly can result in new wireframes or mockups (discussed later).

Implementation details should also be avoided. They are likely to change over the course of the project and have the unfortunate additional effect of excluding from discussions anyone unfamiliar with the inner workings of the code.

User Stories should make clear how value is added to the user. Take for example:

```
A user can enter their name.
```

While this may be something the user can do in your app it is not clear how this adds value to the user.

This story would be better rewritten as:

```
A user can enter their name in order to receive personalized messages.
```

Estimable

Cohn says that there are three common reasons why a story may not be estimable:

1. Developers lack domain knowledge
2. Developers lack technical knowledge
3. The story is too big.

Domain knowledge

All the members of a team should understand the gist of what a story means. If it involves domain elements that they are unfamiliar with, the product owner, or someone knowledgeable in that domain should explain.

Technical knowledge

This a common problem that comes up a lot in this class. The app requires a database but no one has ever made a database before. How then can you estimate how long the database related tasks will take?

When this happens for any technological knowledge gap, one or more team members should be assigned to learn just enough about that technology in order to provide a useful estimate about how long it will take to implement those tasks. The tasks involving those technologies should be split off as much as possible to provide estimates for the parts not involving the technology.

The Story Is Too Big

For a shopping app it would be ridiculous to ask to estimate how long the following story will take:

```
The user can shop for items.
```

This is obviously the craziest of too big examples, but there are many other tasks that you might not be able to think of an estimate on because it is too large in scope. The solution is typically to bring this up. It is possible you have some misconceptions as to the story's meaning, or it might be that the story needs to be broken up.

Small

Stories should be small enough to be easily understood and to be useful in planning and estimation. If you can't easily figure out how to approach a task and how long it will take, it probably needs to be broken up.

However, a story can be too small. Consider the following:

```
A customer can pay for orders with MasterCard

A customer can pay for orders with Visa
```

If the underlying logic for these two cases are the same, there is no need to have them be separate stories.

Testable

When writing a story, make sure there is a way to test whether it has been completed successfully. Consider the following story:

```
A user finds the software easy to use.
```

What does this mean? How does a developer run a test and know that this is done?

The following story is testable:

```
A new user can work their way through the shopping process without need of a tutorial.
```

Sit down with people who have never seen the app and have them try it.

User Roles

User Roles are great for figuring out new stories, as well as prioritizing existing stories.

Cohn defines a user role as "a collection of defining attributes that characterize a population of users and their intended interactions with the system".

Basically, this means defining groups of people that use your app and what they do while they are using it.

A lot of people in past semesters have tried to do this incorrectly. For example in every game app students would use exactly 1 user role named “Player” and in every other app they would use “User” or maybe “Admin”. In each of these types there are actually several more.

Let’s take the newest Pokémon game for example:

To use this tool incorrectly, there are only “Players”, they do everything the game can do.

But do all players actually do everything the game can do?

No! There are several types of players:

- The newbie/ child: These players have never played this type of game before, they might need more assistance getting through tough battles or puzzles, and might not even finish the core game. We would want to emphasize positive early interactions and helpful hints.
- The casual player: These players are probably not going to use much more than their starter Pokémon, play through the main story, and then move on to something else. We want to emphasize the core gameplay and main story.
- The completionist: These players are going to try and get everything this game has to offer, they will catch every Pokémon, explore every path, and talk to every person. We want to concentrate on having lots of little extra things to do.
- The competitor: This person is motivated by beating other real people. We want to provide them with lots of strategic options and the ability to challenge other people.
- And so on...

While these people are all players of the same game, they are all interested in very different features.

With such a huge audience, Nintendo has tried to take into account all these groups and strike a delicate balance of cross group appeal.

In this class, it can be enough to focus on only a few roles, identify which features are most appealing to them, and focus on those. This will go a long way in helping you make a product that is appealing to real users, rather than a hypothetical “Super User” that wants to do everything you want your app to do.

Role Modeling Steps

Cohn recommends using the following steps to come up with user roles:

- Brainstorm an initial set of roles
- Organize the initial set
- Consolidate the roles
- Refine the roles

Brainstorming an initial set

Have each person grab a stack of sticky notes or index cards.

Everyone should then start by writing one role name on each card, placing it on the table and announcing each name as it is placed on the table.

At this time, there is no discussion or evaluation of the roles being written down, just have each person write as many roles as they can think of. There are no turns or questions being asked, each person just needs to write a card whenever a new role is thought of and announce the name when placing it on the table. This should continue until progress stalls and your group is no longer coming up with new roles.

Organize the initial set

At this point, organize all of the roles produced by putting them on a surface, and indicating how much each role overlaps by having them physically overlap with each other.

If two roles are very different, they should be very far apart, whereas if they are exactly the same, they should be on top of each other. If they overlap a little bit, have them only overlap a little bit.

Consolidate the roles

Next, try to condense the roles. Start with the cards that are entirely overlapping. Have the authors of overlapping cards describe what they meant by the role names. If the roles are the same, they can be

condensed into a single new card, or one of the roles can be thrown away. The roles do not have to be exactly the same to be condensed, just similar enough to not be worth distinguishing between.

In addition to consolidating overlapping roles, also get rid of any roles that are unimportant for the success of the system.

After the roles have been reduced, rearrange them to show any relationships between the roles. Roles that are a part of a group should be placed together.

Refining the Roles

It is now time to define some attributes of each role. Any information that differentiates one role from another makes for a good role attribute:

Here are some examples from User Stories Applied:

- The frequency with which the user will use the software.
- The user's level of expertise with the domain.
- The user's general level of proficiency with computers and software.
- The user's level of proficiency with the software being developed.
- The user's general goal for using the software. Some users are after convenience, some favor a rich experience, and so on.

Try these along with any other attributes you find useful for distinguishing the users. As you come up with them, write notes on the card or an attached card.

Story Writing Workshops

During a Story Writing Workshop the participants write as many stories as possible. At this point, these stories are not estimated or given a priority. An effective workshop typically combines brainstorming with some rough planning for the flow of the application.

First, decide which of of the system's user roles you would like to start with. The order doesn't matter as you will be doing this with every role. Next, being a user of that role type and ask the other participants what the user would like to do. The participants should start throwing out ideas for what actions the role might take. For each action, draw a box with a line connecting it to the previous box.

Cohn recommends using a depth first rather than a breadth first approach. Meaning, once a user does an action, keep going with what happens after that action until that line of actions is exhausted or you return to a previous place.

As you walk through the sequence of actions, ask yourself questions that will help you identify missing stories like:

- What will the user most likely want to do next?
- What mistakes could the user make?
- What would confuse the user at this point?
- What additional information could the user need?

The focus during the workshop should be on quantity rather than quality, to help uncover any missing ideas. Do not engage in long discussions on each story. This is not the time to solve problems or design screens. If a story is found to be redundant or is replaced later, it can be thrown away. Low quality stories can be assigned a low priority during release planning. Even if you are eventually going to be using an electronic tool for the stories, during the workshop use physical sticky notes or index cards.