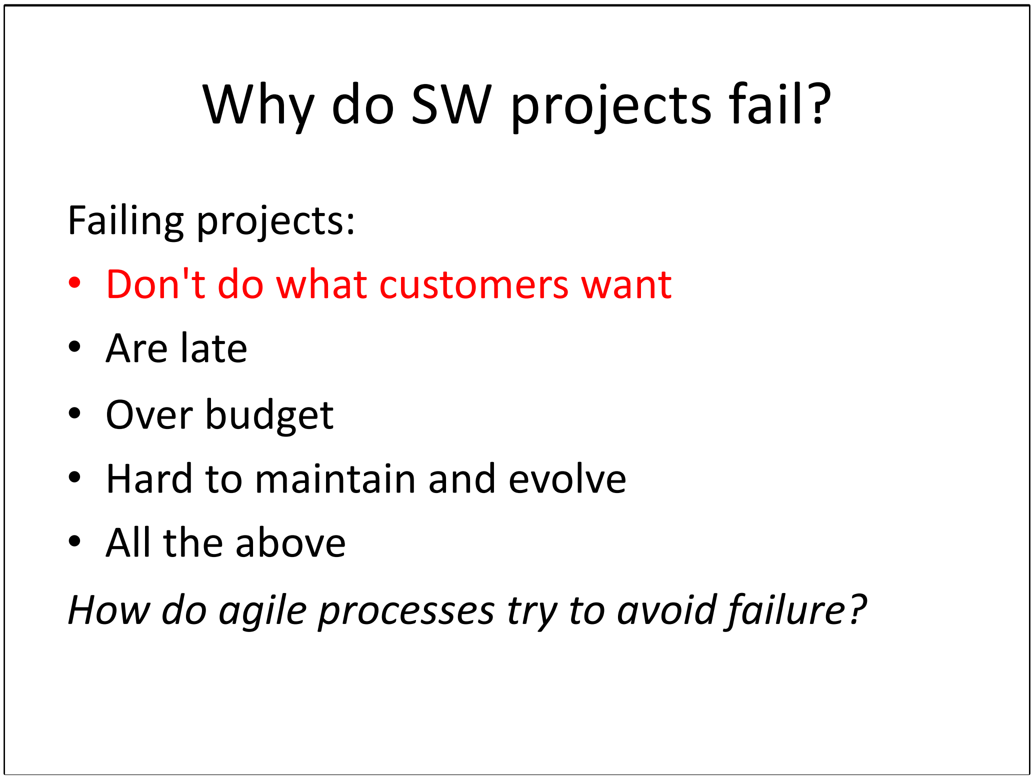


Why do SW projects fail?

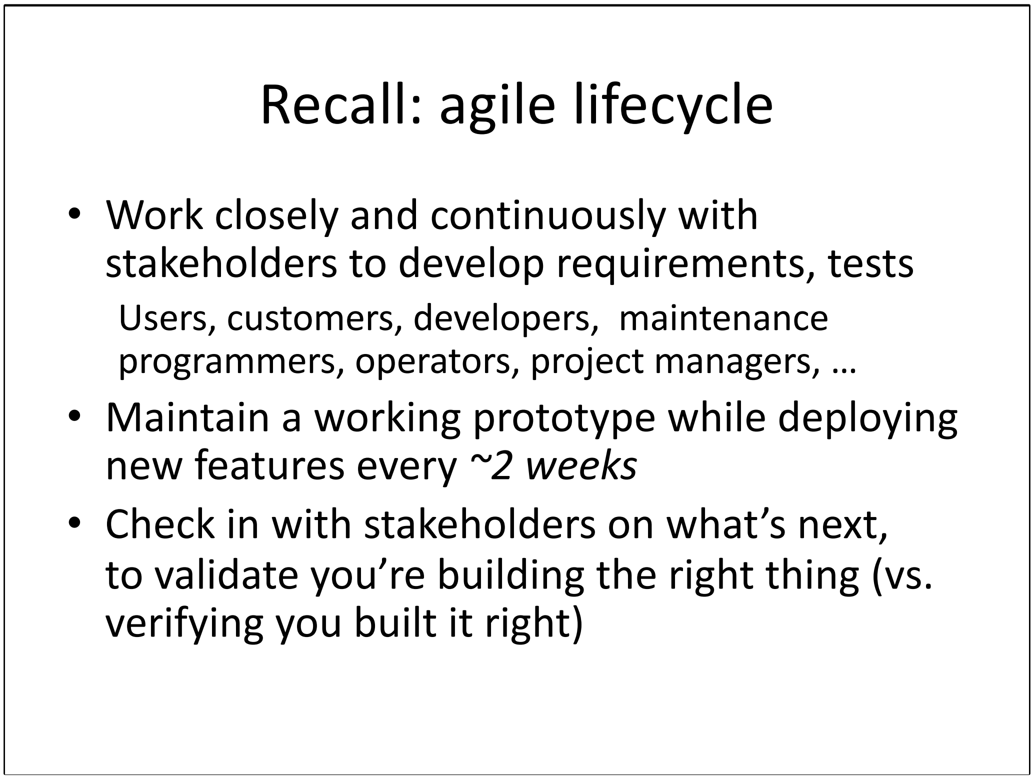
Failing projects:

- Don't do what customers want
- Are late
- Over budget
- Hard to maintain and evolve
- All the above

How do agile processes try to avoid failure?

But particularly don’t do what customers want … (although these are likely related; an application that is hard to maintain or evolve will be expensive and behind schedule in evolving to meet customer needs/desires).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



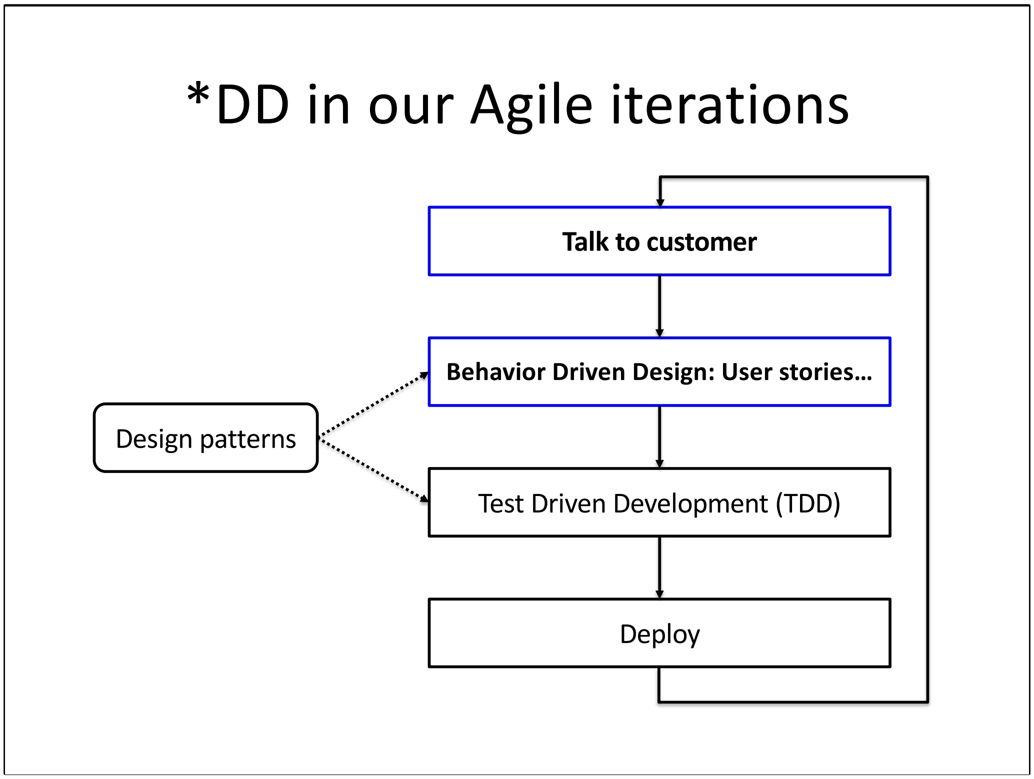
Recall: agile lifecycle

- Work closely and continuously with stakeholders to develop requirements, tests

  Users, customers, developers, maintenance programmers, operators, project managers, …
- Maintain a working prototype while deploying new features every *~2 weeks*
- Check in with stakeholders on what's next, to validate you're building the right thing (vs. verifying you built it right)

Recall …

These short cycles provide frequent opportunities to check in with stakeholders for validation. What is the difference between validation and verification? Validation is "Did we build the right thing?", verification is "Did we build it right?"



*DD in our Agile iterations

Talk to customer

Behavior Driven Design: User stories…

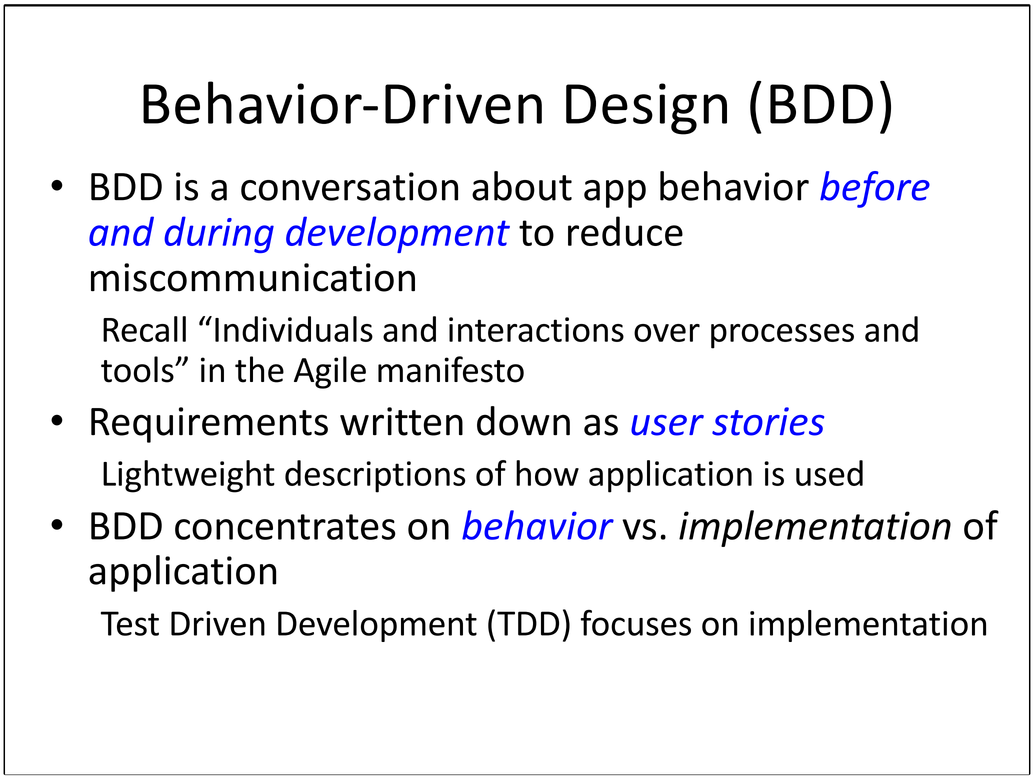
Design patterns

Test Driven Development (TDD)

Deploy

Recall we covered TDD in previous classes. TDD is the process of writing the tests first. It is a technique for verification – “Did we build it right”. Today we will talk about Behavior Driven Design or BDD. This is a process to help us elicit user requirements and validate we are building the right thing.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Behavior-Driven Design (BDD)

- BDD is a conversation about app behavior *before and during development* to reduce miscommunication

  Recall “Individuals and interactions over processes and tools” in the Agile manifesto

- Requirements written down as *user stories*

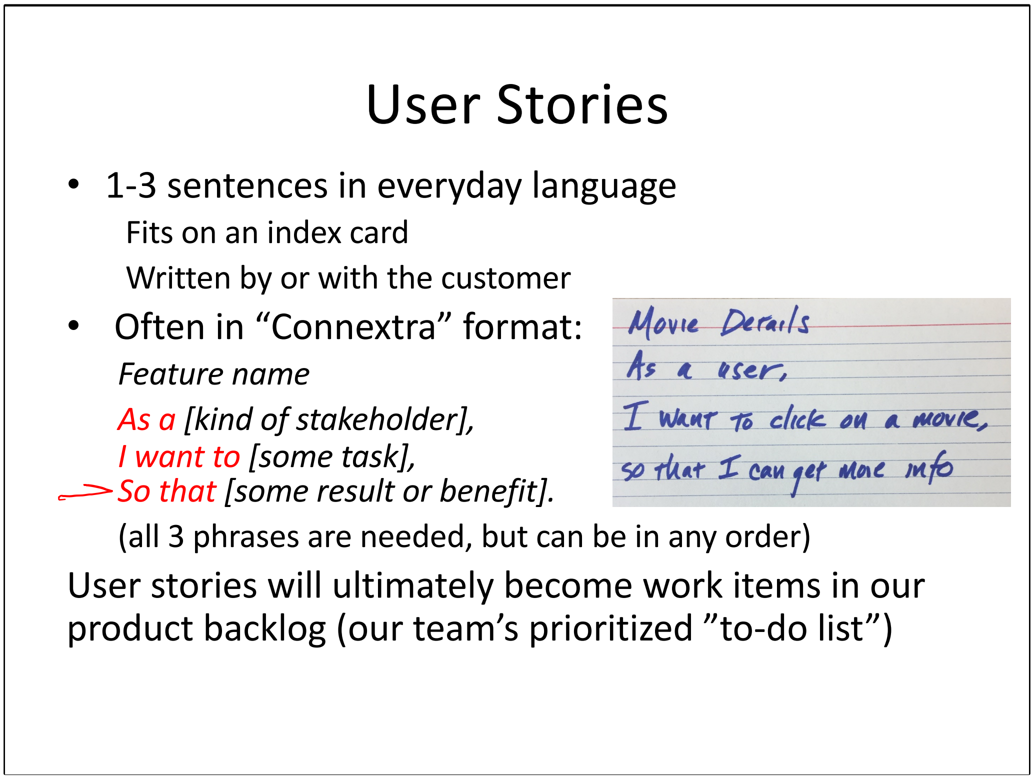
  Lightweight descriptions of how application is used

- BDD concentrates on *behavior* vs. *implementation* of application

  Test Driven Development (TDD) focuses on implementation

As it name suggests, BDD concentrate on what the application does as opposed to how the application does it.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



User Stories

- 1-3 sentences in everyday language

  Fits on an index card

  Written by or with the customer

- Often in “Connextra” format:

  Feature name

  As a [kind of stakeholder],
  I want to [some task],
  So that [some result or benefit].

  (all 3 phrases are needed, but can be in any order)

Movie Details
As a user,
I want to click on a movie,
so that I can get more info

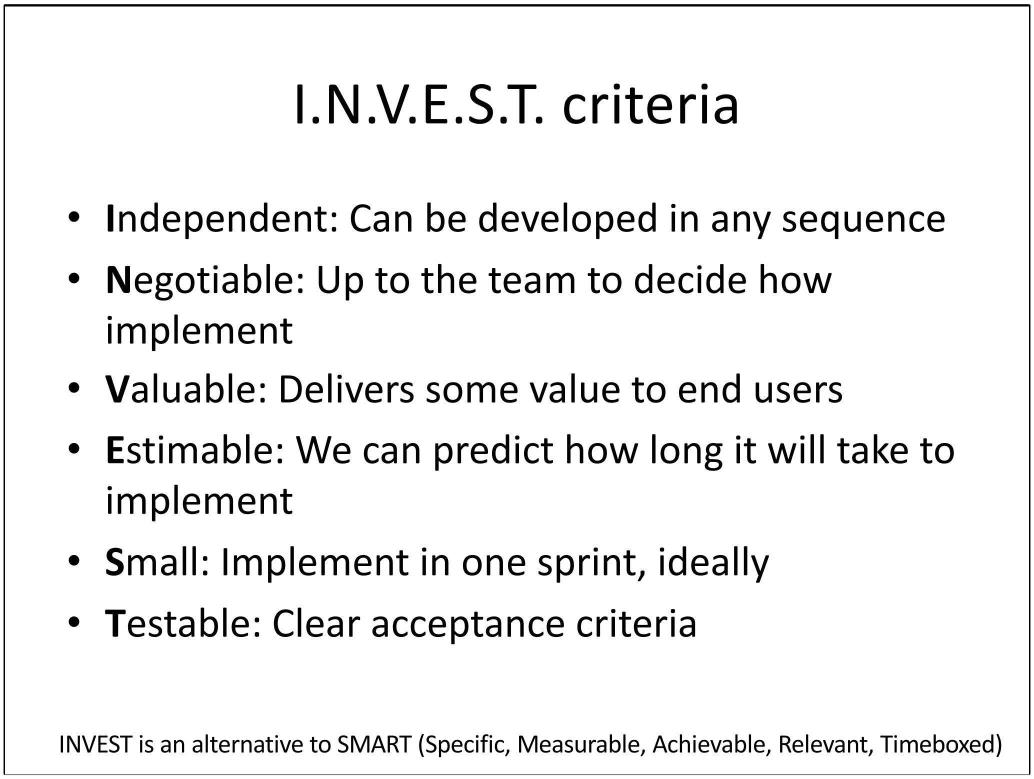
User stories will ultimately become work items in our product backlog (our team’s prioritized ”to-do list”)

Why index cards?

- Nonthreatening: All stakeholders participate in brainstorming
- Easy to re-arrange: All stakeholders participate in prioritization
- Helps keep stories short and low-cost to change during development

<end> Why are all 3 parts needed? And why is the result/benefit part particularly needed? That part captures the value of this feature and will be a key piece of information for prioritizing our work. In the extreme, if we can't articulate why this feature is valuable, then we shouldn't be building it!

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.

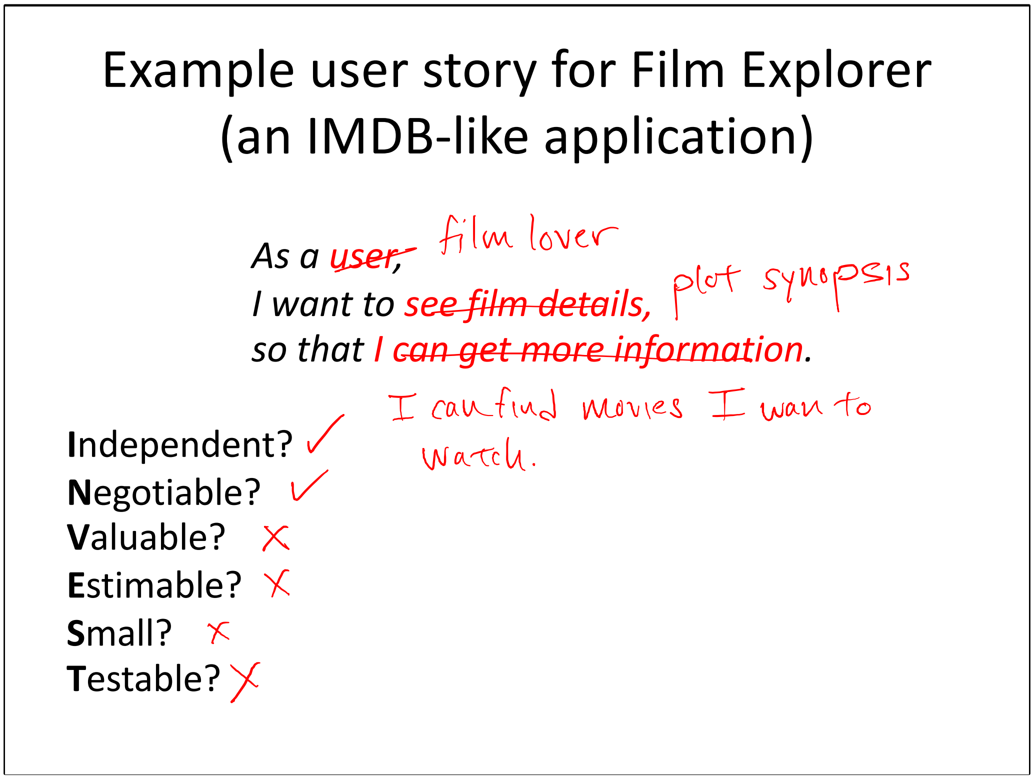


I.N.V.E.S.T. criteria

- **I**ndependent: Can be developed in any sequence
- **N**egotiable: Up to the team to decide how implement
- **V**aluable: Delivers some value to end users
- **E**stimable: We can predict how long it will take to implement
- **S**mall: Implement in one sprint, ideally
- **T**estable: Clear acceptance criteria

INVEST is an alternative to SMART (Specific, Measurable, Achievable, Relevant, Timeboxed)

This process (and any CS312 process) wouldn't be complete without an(other) acronym… A good user story is INVESTable. This is a finer grain but similar criteria to SMART (Specific, Measurable, Achievable, Relevant, Timeboxed), as described in the ESaaS book.



Example user story for Film Explorer (an IMDB-like application)

As a ~~user,~~ film lover
I want to ~~see film details,~~ plot synopsis
so that I ~~can get more information~~. I can find movies I wan to watch.

Independent? ✓
Negotiable? ✓
Valuable? ✗
Estimable? ✗
Small? ✗
Testable? ✗

Consider: As a user, I want to see film details, so that I get more information

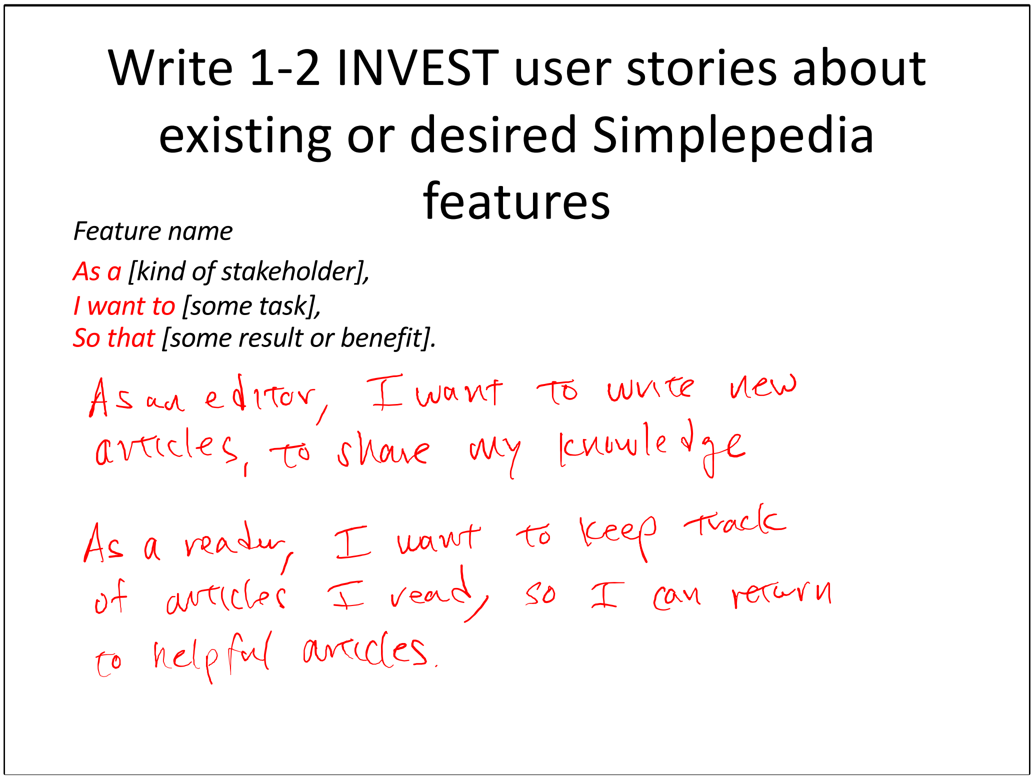
Does our example user story meet our INVEST criteria? Why or why not?

Likely independent, and negotiable. But not clearly valuable, and knowing if it is small, estimable and testable would likely require more information, such as what details. How could we rewrite this user story to better indicate the value?

One area of improvement is to be more specific (so the value can be clearer). For example, ”User” is not a very specific stakeholder (and a term we should generally avoid in user stories as it doesn’t tell us much of anything). What details? And how does the stake holder benefit from those details?

As a film lover, I want to read plot synopses, so that I can find movies I might like to watch

Note that we could conceive of multiple user stories that target the same feature from different perspectives. Maybe a casual fan is just interested in posters, or an industry professional is interested in dates or other data.



Write 1-2 INVEST user stories about existing or desired Simplepedia features

Feature name

As a [kind of stakeholder],
I want to [some task],
So that [some result or benefit].

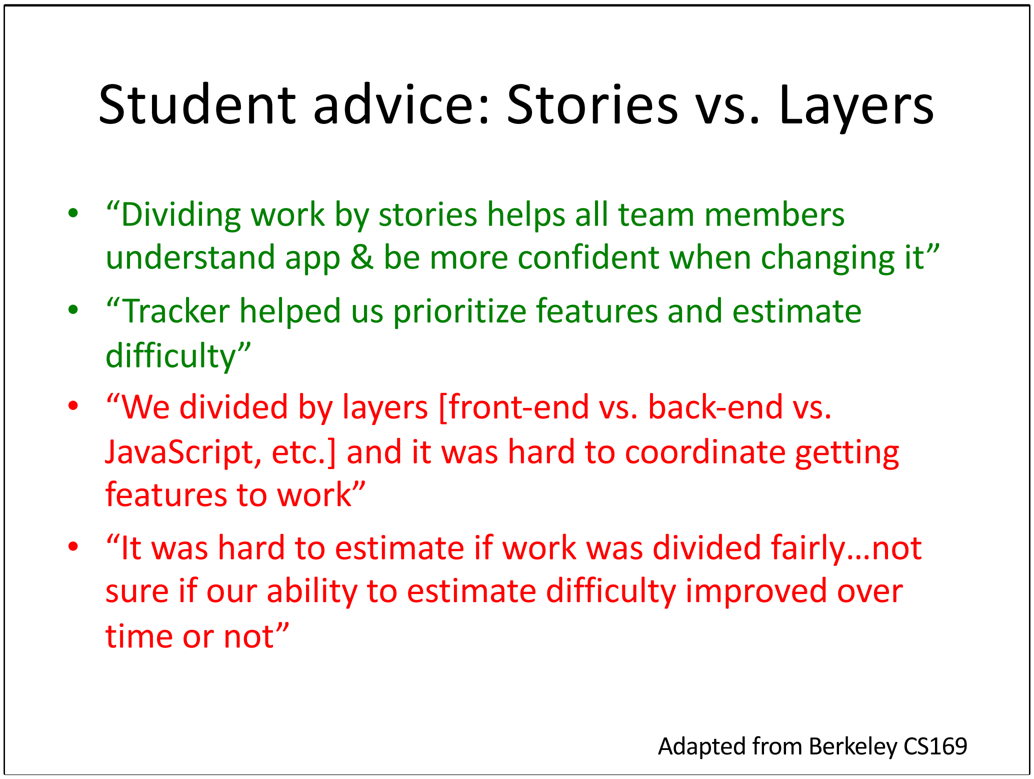
As an editor, I want to write new articles, to share my knowledge

As a reader, I want to keep track of articles I read, so I can return to helpful articles.

1. As a reader, I want to see the edit history of an article, so I am aware what changes have been made and by whom.
2. As a contributor, I want to be able to use rich text formatting, so that I can create more readable and useful articles (with structured formatting, links).
3. As a reader, I want to be able to perform full text search, so I can find relevant articles based on their content.

Why does this matter? Writing a user story is a tool for avoiding building what we as the developer(s) think is cool but is not ultimately what the user/customer wanted or needed. As the ESaaS authors note: “User stories help all stakeholders prioritize development and reduce chances of wasted effort on features that only developers love.” If you can’t articulate the value of the feature, should you be building it? (Probably not!)

By linking eventual development tasks to User valuable features, you ensure that even those “developer facing” tasks, e.g., building some “behind the scenes” infrastructure, are valuable, i.e., serves a feature that is valuable to the user.



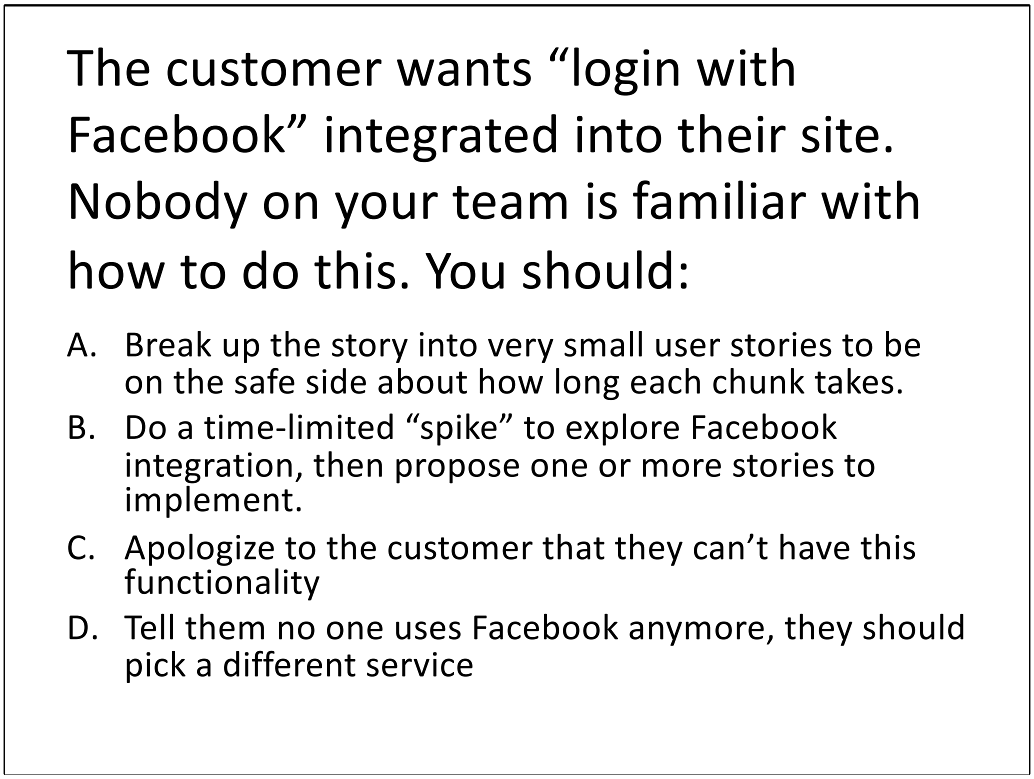
Student advice: Stories vs. Layers

- “Dividing work by stories helps all team members understand app & be more confident when changing it”
- “Tracker helped us prioritize features and estimate difficulty”
- “We divided by layers [front-end vs. back-end vs. JavaScript, etc.] and it was hard to coordinate getting features to work”
- “It was hard to estimate if work was divided fairly…not sure if our ability to estimate difficulty improved over time or not”

Adapted from Berkeley CS169

We noted earlier that the user stories become the items in our prioritized work queue. We will talk more about that process next week, in the meantime I want to emphasize the importance of defining these items to organize and facilitate parallel work streams, i.e., the whole team contributing at once (recall the "I", "Independent" in INVESTable).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



The customer wants “login with Facebook” integrated into their site. Nobody on your team is familiar with how to do this. You should:

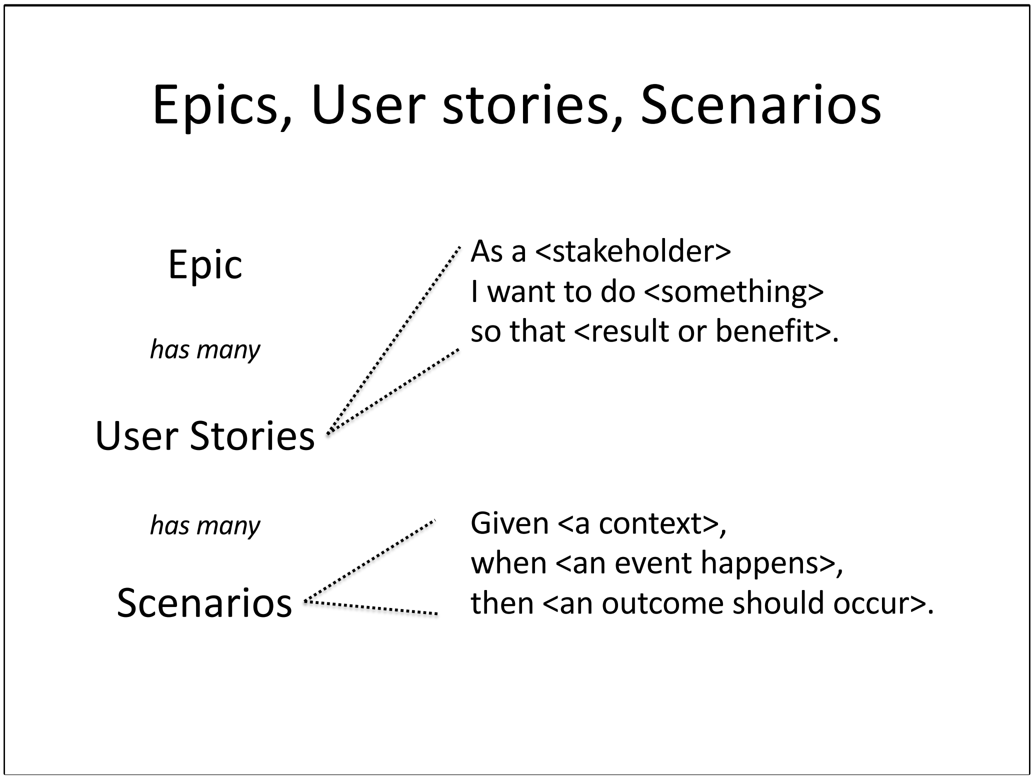
A. Break up the story into very small user stories to be on the safe side about how long each chunk takes.
B. Do a time-limited “spike” to explore Facebook integration, then propose one or more stories to implement.
C. Apologize to the customer that they can’t have this functionality
D. Tell them no one uses Facebook anymore, they should pick a different service

Answer: B

I think B is the best answer, but A could be arguable. C (and D) are not correct in this context, but that doesn't mean you should never say no to the customer.

A **spike** is a product-testing method originating from Extreme Programming that uses the simplest possible program to explore potential solutions[1]. It is used to determine how much work will be required to solve or work around a software issue. I think of it is as what is the minimal effort that would bring something like integrating Facebook login into the realm of the known. In this context it helps ensure the user stories are "estimable" (we know what is involved) and are as "small" as we think (or hope).

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license



Epics, User stories, Scenarios

Epic

has many

User Stories

As a <stakeholder>
I want to do <something>
so that <result or benefit>.

has many

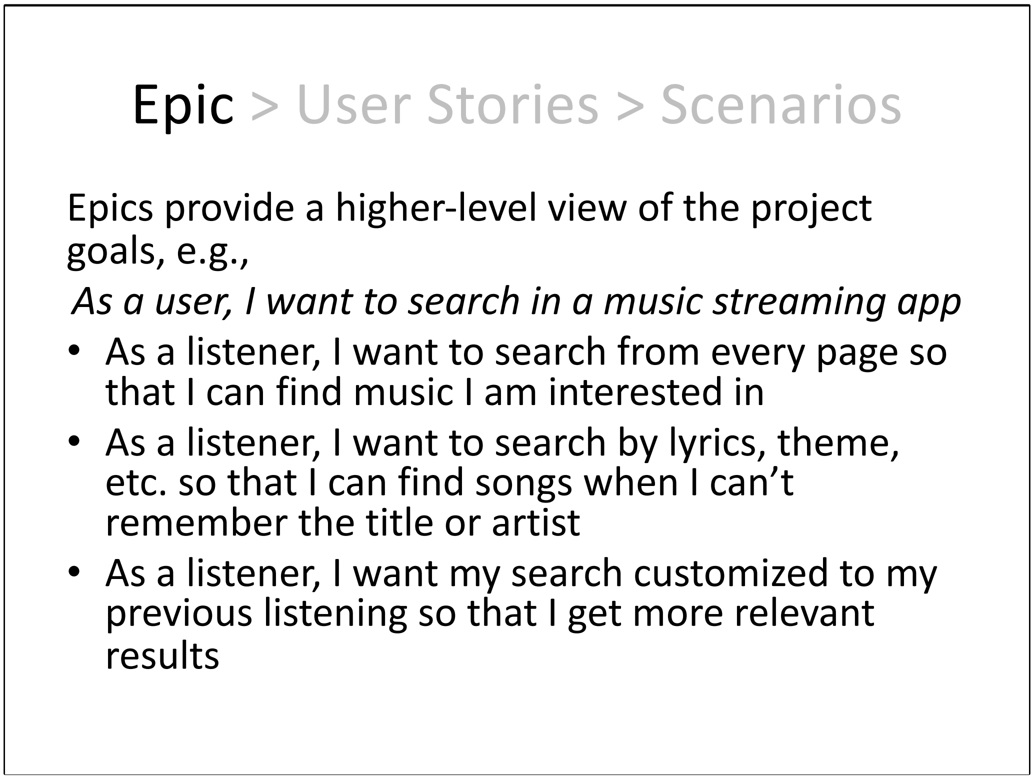
Scenarios

Given <a context>,
when <an event happens>,
then <an outcome should occur>.

We will think about user stories as a part of hierarchy that starts with Epics. An epic provides a high-level description of the application's goals/features. If we can express it in Connextra format, great! An epic can be expressed/refined into multiple user stories, which can each be expressed/refined as multiple (testable) scenarios.

[At end]

Not all work items will be user stories. Some work-items will be bugs, Sometimes a task is necessary but far removed from the user, e.g., read an arbitrary byte range from a local or remote file. Recall moderation in all things. We don't want to force everything we need to into this framework. It is most relevant for user-facing features.



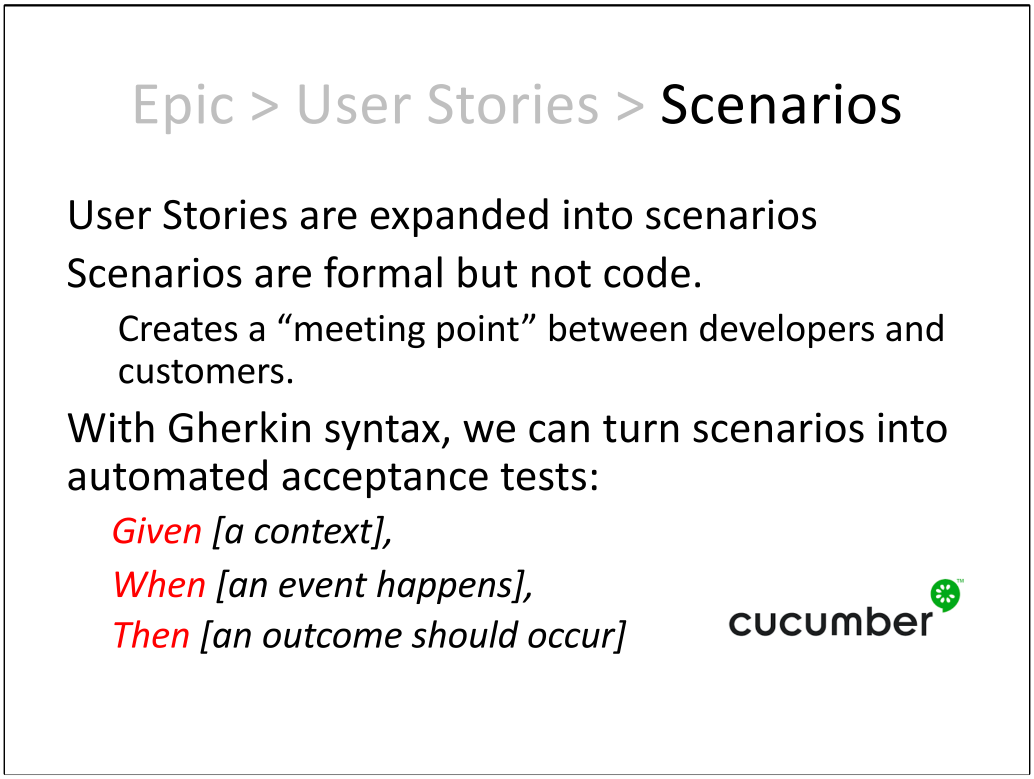
Epic > User Stories > Scenarios

Epics provide a higher-level view of the project goals, e.g.,

As a user, I want to search in a music streaming app

- As a listener, I want to search from every page so that I can find music I am interested in
- As a listener, I want to search by lyrics, theme, etc. so that I can find songs when I can’t remember the title or artist
- As a listener, I want my search customized to my previous listening so that I get more relevant results

We then break that Epic down into INVESTable user stories, for example… An Epic will translate into multiple (many) user stories. Recall our goal is to get our user stories to be small enough to implement in one sprint (the "S", "Small", in INVESTable).



Epic > User Stories > Scenarios

User Stories are expanded into scenarios

Scenarios are formal but not code.

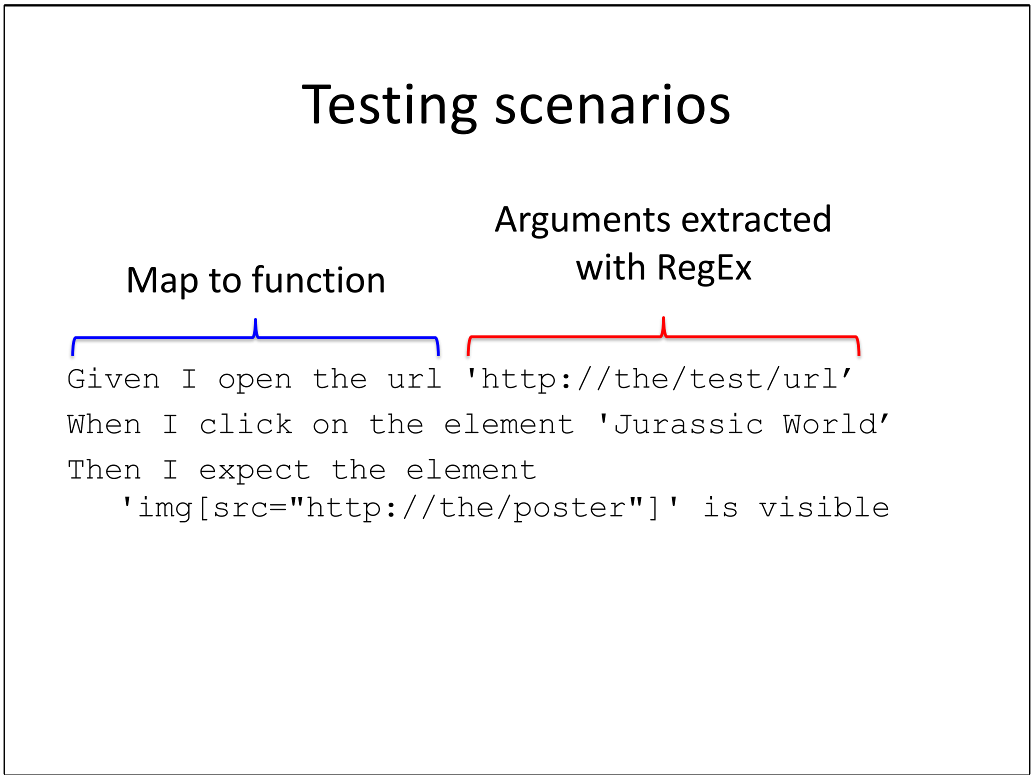
- Creates a “meeting point” between developers and customers.

With Gherkin syntax, we can turn scenarios into automated acceptance tests:

- *Given [a context],*
- *When [an event happens],*
- *Then [an outcome should occur]*

cucumber

1. `Given` steps represent state of world before event, the preconditions
2. `When` steps represent event, e.g., simulate user pushing a button
3. `Then` steps represent expected post-conditions, the test expectations
4. `And` and `But` extend any previous step.



Testing scenarios

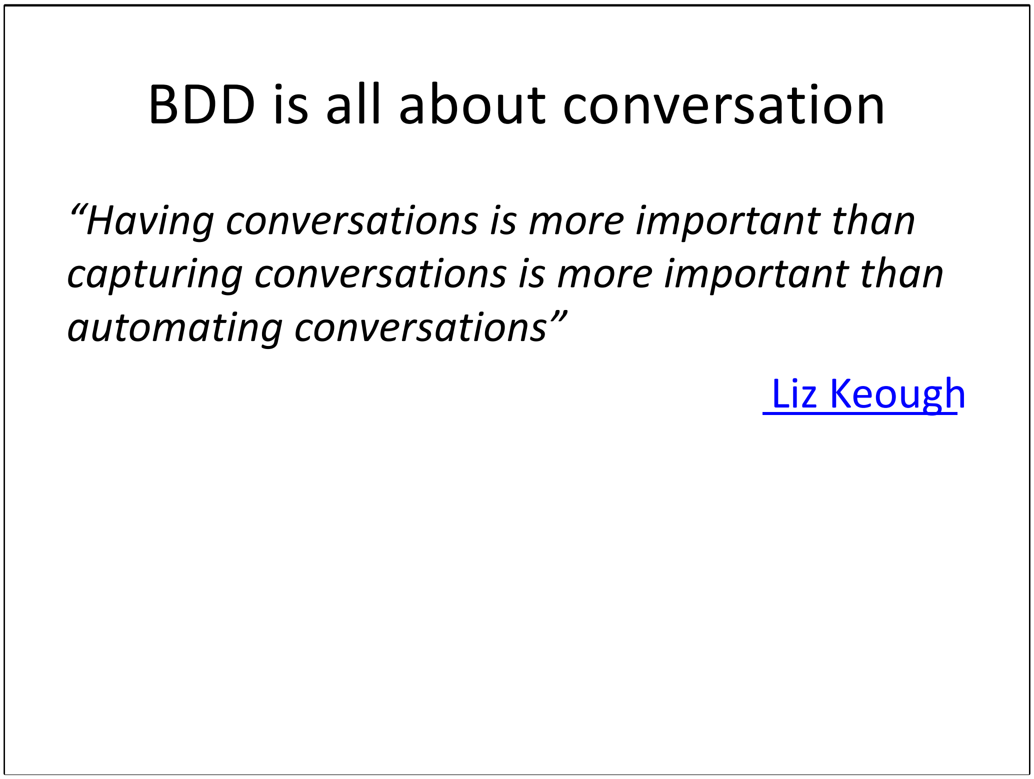
Map to function

Arguments extracted with RegEx

```
Given I open the url 'http://the/test/url’
When I click on the element 'Jurassic World’
Then I expect the element
   'img[src="http://the/poster"]' is visible
```

There are tools, like Cucumber, that can directly execute these tests, i.e., the first part corresponds to a function call, and the second is its argument. That said, you don't have to use Cucumber or its equivalents to implement and get values from "Given-When-Then"-style tests. For simplicity, we will use code instead of trying to incorporate Cucumber. Why? Cucumber brings non-trivial overhead. I think the real value is expressing scenarios in a way that can be readily translated into tests. Hopefully, you can already imagine how this might translate to tests using React Testing Library, e.g.,

1. Render the React component associated with that URL (or use a mock router to render that page)
2. Find the element with the text
3. ”Fire” click even on found element
4. Query for image tag and assert is is displayed

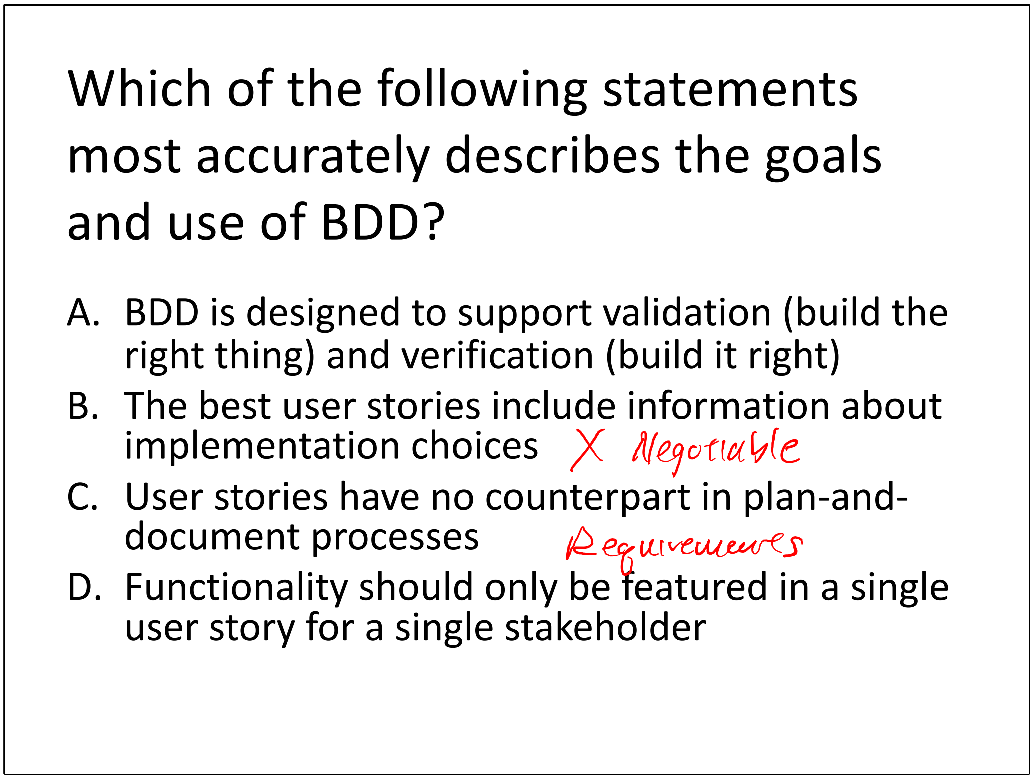


BDD is all about conversation

“Having conversations is more important than capturing conversations is more important than automating conversations”

Liz Keough

That is the overall process we are implementing is more important than the specific tools we use to capture or apply those conversations. The goal for that process is to encourage conversation, not necessarily to produce specific artifacts (i.e., don’t miss the forest for the trees…)



Which of the following statements most accurately describes the goals and use of BDD?

A. BDD is designed to support validation (build the right thing) and verification (build it right)

B. The best user stories include information about implementation choices X Negotiable

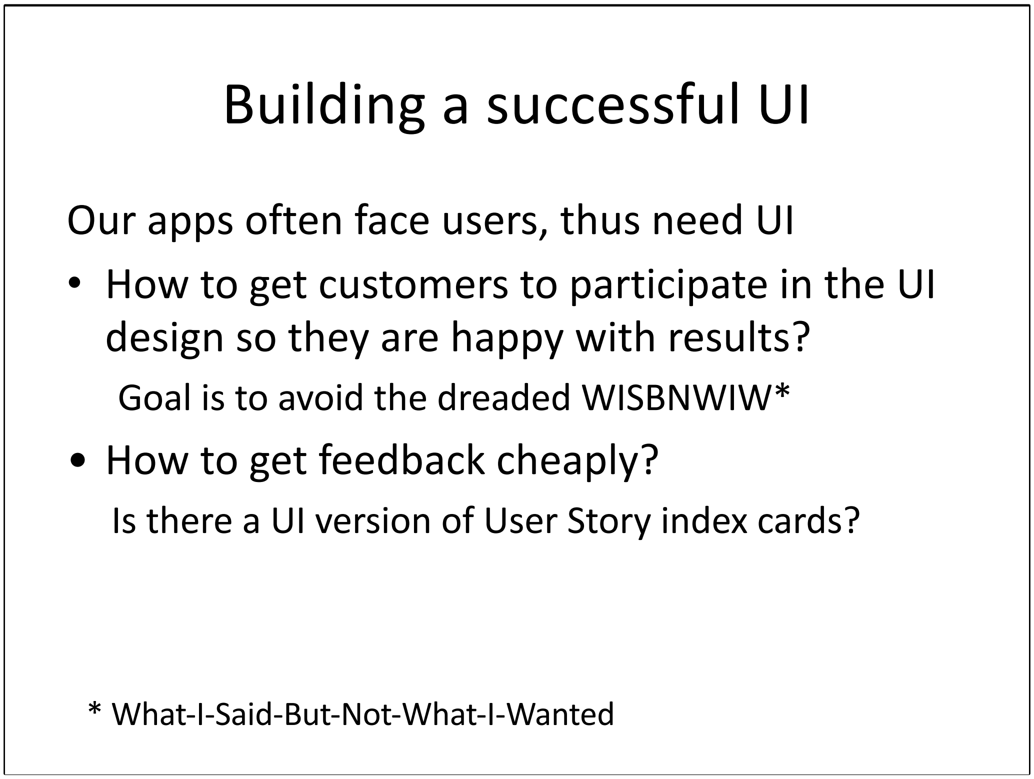
C. User stories have no counterpart in plan-and-document processes Requirements

D. Functionality should only be featured in a single user story for a single stakeholder

Answer: A

User stories are about behavior not implementation (recall the "N", "Negotiable" in INVEST) and are similar to requirements in P&D. Multiple user stories may describe the same functionality, but from different stakeholders' perspectives. Imagine a movie ticketing system that integrates with a social network. From the user's perspective "so that I can see movies with my friends", from a theater owner's perspective "so that I can sell more tickets". Knowing these perspectives can help us during the design and implementation of the feature.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license



Building a successful UI

Our apps often face users, thus need UI

- How to get customers to participate in the UI design so they are happy with results?

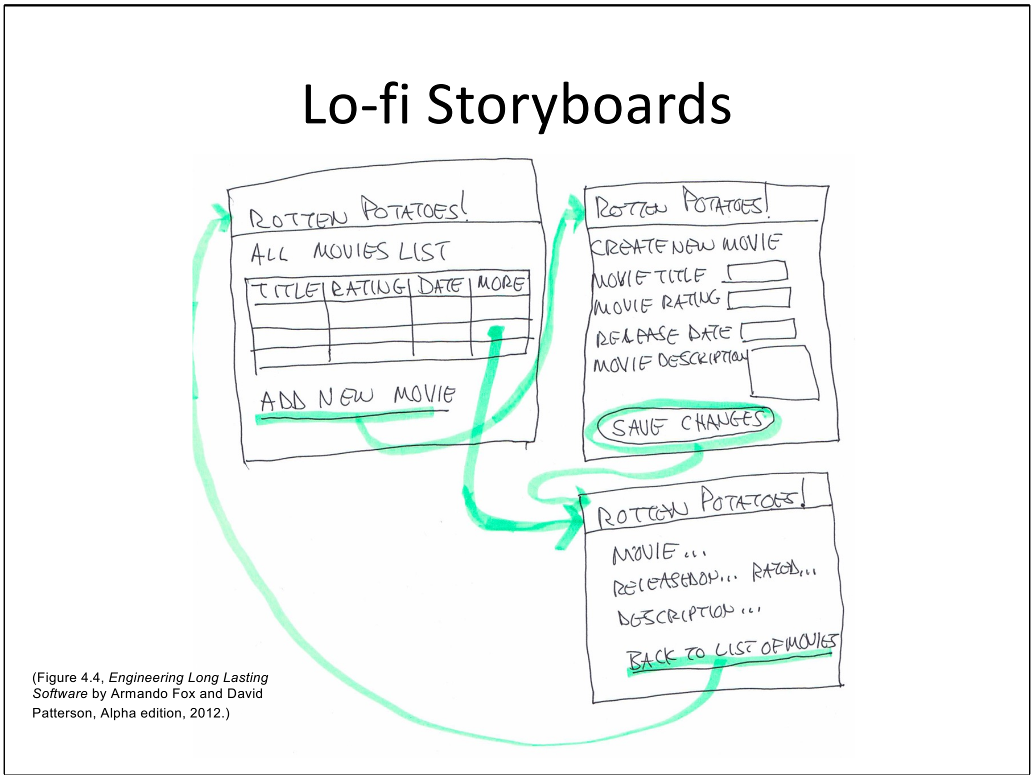
  Goal is to avoid the dreaded WISBNWIW*

- How to get feedback cheaply?

  Is there a UI version of User Story index cards?

* What-I-Said-But-Not-What-I-Wanted

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Lo-fi Storyboards

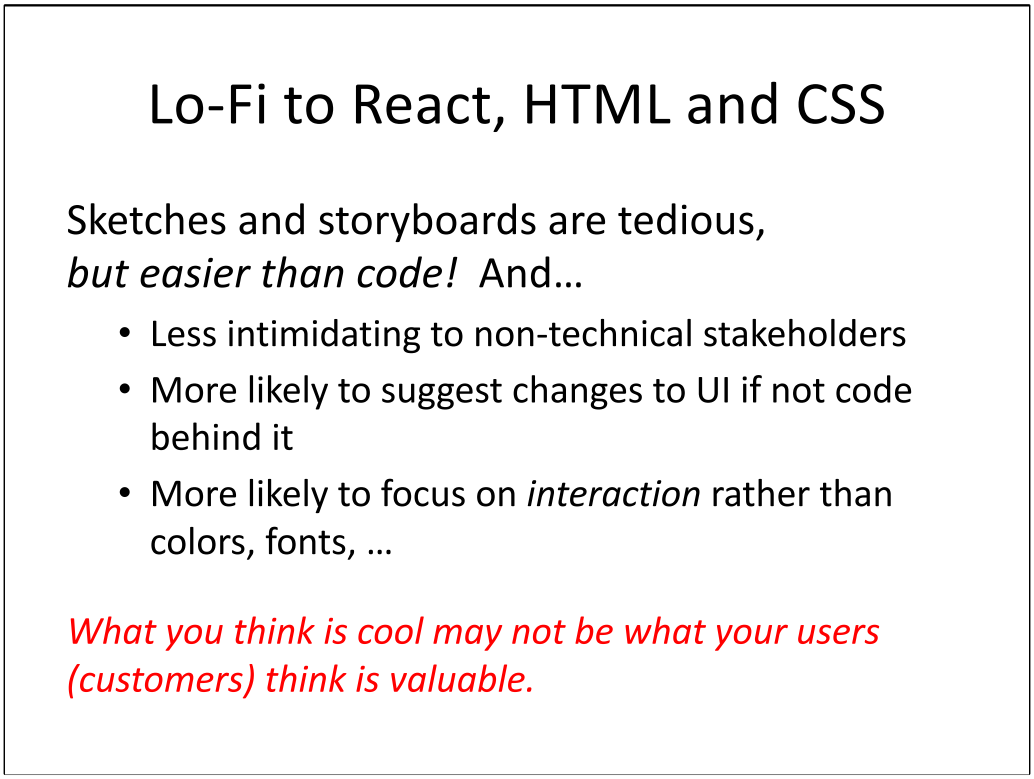
(Figure 4.4, *Engineering Long Lasting Software* by Armando Fox and David Patterson, Alpha edition, 2012.)

I want to emphasize this pitfall from the ESaaS text:

“Sketches are static; interactions with a SaaS app occur as a sequence of actions over time. You and the customer must agree not only on the general content of the Lo-Fi UI sketches, but on what happens when they interact with the page. “Animating” the Lo-Fi sketches—“OK, you clicked on that button, here’s what you see; is that what you expected?”—goes a long way towards ironing out misunderstandings before the stories are turned into tests and code.”

That is Lo-fi storyboards should capture the “What happens when the user does…”, not just what the UI looks at any moment.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Lo-Fi to React, HTML and CSS

Sketches and storyboards are tedious,
but easier than code! And…

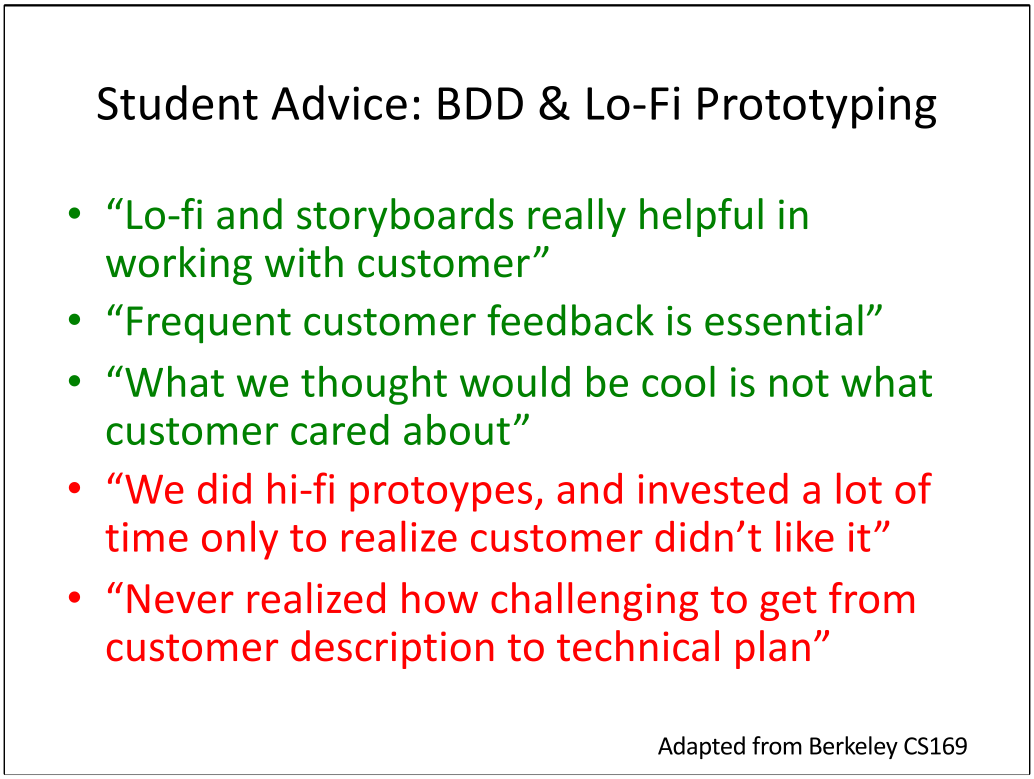
- Less intimidating to non-technical stakeholders
- More likely to suggest changes to UI if not code behind it
- More likely to focus on *interaction* rather than colors, fonts, …

What you think is cool may not be what your users (customers) think is valuable.

lo-fi prototypes are hugely important tools. It takes time to code up a nice-looking application. It could all be wasted time if it isn't what we should be building. I always encourage you to do some early exploration during the design phase when it is cheap to make changes. Always draw at least two designs and try to make them as different as possible. It is very easy to fall into creating "safe" designs that no one wants to use

[click] The key point to remember, is that what you think is cool, may not be what your customer/user may think is valuable. Keep in mind that unless you are making an application for other software developers, you are likely not a very representative user. We need to be soliciting feedback from (all) representative users and lo-fi storyboards are an efficient way to do so.

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.



Student Advice: BDD & Lo-Fi Prototyping

- “Lo-fi and storyboards really helpful in working with customer”
- “Frequent customer feedback is essential”
- “What we thought would be cool is not what customer cared about”
- “We did hi-fi protoypes, and invested a lot of time only to realize customer didn’t like it”
- “Never realized how challenging to get from customer description to technical plan”

Adapted from Berkeley CS169

Adapted from Armando Fox and David Patterson (Berkeley cs169) under CC-BY-SA-NC license.