It's All in How You Slice

Design your project in working layers to avoid half-baked incremental releases.

BY JEFF PATTON

SUPPOSE FOR A MOMENT YOU'RE IN CHARGE OF development at BigNewIdea Software. You've adopted an Agile development approach that suggests releasing your software in smaller incremental releases is a good idea. BigNewIdea's marketing folks and business managers exclaim excitedly, "If we build the highest-value features first, the users will love our first release!" You develop for a few months, always shifting the highest-value features to the top of the stack. You eventually release a well-built, relatively bug-free release. First responses from people using the software seem positive. "This looks really good!" they say. "It's got all these features we've wanted for a long time!" But strangely, sales seem to be slower than expected. Salespeople find that many customers who have purchased the software aren't actually using it. "What's wrong?" marketing asks through questionnaires and expensive focus groups. The reply: "What you have is fine, but there's not enough here

TOM SCHIERLITZ/GETTY IMAGES

It

for me to get my job done. I still have to do much of my work manually. I have to transfer information from my paperwork to the software and back again. Sometimes I can't even figure out a paper workaround. I'll wait for the next release to see if you get farther along."

[When releasing] software incrementally, how do you choose a first bundle of features that is both high value *and* immediately useful?

You expected that all those high-value features would make a great product, but it turned out you needed some of those low-value features to hold everything together—to make the product useful to those trying to accomplish work with the software. If you still want to release the software incrementally, how do you choose a first bundle of features that is both high value *and* immediately useful?

Key Ingredients

2–3	large sheets poster paper
3–4	packs markers
1	roll tape
1–2	packs 3×5 index cards

Plus, the fundamental element of all good meetings: food.

In this article, we'll walk through a simple, collaborative, card-based planning model that does just that.

PREP WORK

Before you begin, you need to choose a good cross section of folks to participate in creating the model. While the model could be prepared by one person, I wouldn't recommend it. Having a mix of people will help increase understanding of the software throughout the team. Choose people familiar with the users and functionality of your software, such as domain experts, testers, and user interface designers. Choose people who have some ideas on how the software will earn your company money—your stakeholders. Choose people who know something about how long this work will take to build—a developer or two. Four to eight people total is a good number. Use the creation of the model as an opportunity to elicit discussion about the features being built.

STEP 1: COLLECT FEATURES

With the prep work done, it's time to start assembling the model. The first step is to answer the question "What does our software do?" You should start with a user-centric list of features. Depending on your situation, this might be trickier than it sounds.

My definition of a good feature is one that is expressed from a user's perspective. For example, if I were building new software for a retail store, a feature might be "sell items at point of sale" as opposed to "the system supports EAN-13 barcodes." There's a difference there that I hope is not so subtle. The first feature describes an activity done by a person; the second describes an attribute of an object. Look for features that start with or include some action verb; that's a good sign. When describing your software, it helps to indicate how it will be used rather than how it might look or the details of its implementation. Keeping your focus on the usefulness of the software at this stage helps to ensure that the bits of software released incrementally will be useful.

If you're not already describing features for your software in a user-centric way, you may need to spend a little time reframing your features.

Write the features on 3×5 cards or on something else that you can easily move around in your model. I've found it's easy to merge features originating in a spreadsheet with a word processor document that will print them on precut 3×5 cards or business cards. This way, the cards are easy to read and work well within a card-modeling exercise. Leave room under the feature statement for the details we'll add in Steps 2 and 3.

Suppose I'm building some software for small retailers. I know that their business processes go a bit like this (notice that each one starts with an action verb):

- Create purchase order for vendor
- Receive shipment from vendor
- Create tags for received items
- Sell items
- Return and refund items
- Analyze sales

STEP 2: ADD DETAILS

To help you model these features, let's note three important details on the cards: who uses the feature, how often the feature is used, and how valuable the feature is.

First, for each feature, detail the *kind* of user who uses it. When describing this feature, you likely envisioned someone using it—who was he? You can identify him with a job title, a role name, a persona, or in any other way most appropriate for your system. (See this issue's StickyNotes for more on roles.)

Looking back at my set of retail store features, I know that the same person usually doesn't do all this stuff. I know that the work is divided between merchandise buyers, stock receivers, customer consultants, and sales analysts. I note each user under the feature state-

create po for vendor
(merchandise buyer)
frequency: weekly
value: medium

Figure 1: A completed feature card.

Tips for Success

When assembling the batch of features for your project, pay attention to the following details to help ensure success.

When using use cases, focus first on those where the actors are a single user and the system. Avoid planning with use cases that describe the internal workings of the system. Avoid use cases that describe business processes at a very high level; make them what author Alistair Cockburn calls "sea level system scope use cases."

If you're using user stories, include a user of the feature in a concise story statement, such as the one author Mike Cohn credits practitioner Rachel Davies with inventing: As a [type of user] I want [some particular feature] so that [some benefit is received]. For example: "As a bank customer I want to view my current account balance so that I know my recent deposit went through."

The completed model shows us features arranged in the order they're needed by people and business processes, but this really only gives us an indication of dependence. As you slice off releases, scan each feature for dependencies that might not be in this release or a prior one. I've found that if I slice releases horizontally starting from the top of the model down, I rarely run into dependencies I haven't already resolved in a previous release.

Sometimes folks suggest features that aren't really about users and the functionality they need. Features like "migrate to an Oracle database" or "change the look and feel to match our new branding." These sorts of features don't work well in this type of model.

When talking about certain features, you might find it's tough to defer some of them completely. When slicing off a set of features to release, discuss how usable that release will be. For each user of the system ask if she will be able to do her work with this subset of features. For each important feature left out, is there a paper process or software workaround that allows her to live without the feature—no matter how annoying that might be? Could the feature be split into a crude minimal version for earlier release and a more elaborate version for later release?

See this issue's StickyNotes for some reading that will help ensure a good outcome.

ment he is involved with.

Next, note how frequently you believe each feature will be used. You can use simple notation like high, medium, or low. I use a little more precise continuum, writing under the user on the feature card either hourly, daily, weekly, monthly, or quarterly.

Finally, for each feature, note its value to the purchasers of this system. If your company has a good understanding of where ROI comes from on this system, this may not be too hard—but for the rest of us, this is usually a subjective judgment. Using high, medium, and low will work fine for our use today. I'll write the value under the frequency on each card.

When all of these details have been added, one of the feature cards might look like the one in Figure 1.

When trying this at home, make adding these details a collaborative activity. Assuming you've got your features written or otherwise printed on cards, spread those cards out on the table. Take turns picking up cards and adding user, frequency, and value. If you've got a good mixed group, you'll notice that some folks have strong opinions about some of these details. Some folks may know a bit about the user and frequency, but nothing about value. You'll find with a good mixed collaborative group, you'll be able to quickly fill in all these details. You'll notice lots of good discussion while doing it.

When writing your features on cards, the same information should appear in the same place all the time. This makes the cards easy to read when placed in the model. They might start to look like playing cards in a game. That's good. Building the model *should* feel a bit like you're playing a game.

STEP 3: PLACE CARDS IN SEQUENTIAL ORDER

To build this model, lay a few sheets of poster paper on a large worktable. This model is generally wide, so arrange sheets and tape them together to form a broad poster.

Draw a horizontal line across the top of the page and label it **usage sequence.**

Draw a line on the left side of the page from top to bottom and label it **criticality.** Label the top endpoint of this line **always used,** the bottom endpoint **seldom used.** The resulting diagram should look like the one in Figure 2.

You now need to place features in the model according to usage sequence and criticality. By using the features we wrote out for our retail software earlier, we've already listed them in the order the features will be used. PO creation happens before shipments are received from the vendor. Tags are created before the items are put on the shelf and sold. Sales are

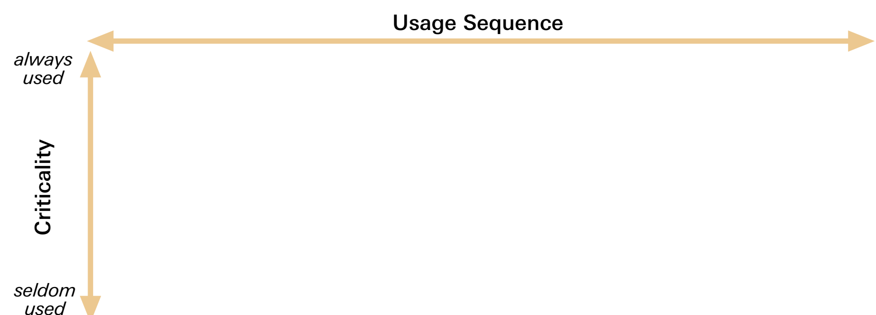

Figure 2: The model starts with an x and a y axis.

analyzed after some items are sold. That's what I mean by **usage sequence.**

In reality, it may not seem so cut and dried. If we really look at a retail store, we might find buyers on the phone placing orders at the same time receiving clerks are in the back room receiving and tagging. If the store's open, we hope customers will be on the retail floor happily buying our products and customer consultants will be ringing them up. It looks like all these features are being used simultaneously and indeed they are. So when sequencing them in your model, arrange them in the order that seems logical when explaining to others the business process. If you explain the business process starting with the selling part, that's OK, put that feature first. We want this model to help us tell stories about our software, so arrange them in an order that makes it easy to tell stories.

In reality, it may not seem so cut and dried. . . . It looks like all these features **are being used simultaneously** and indeed they are.

Distribute the cards among participants. Then have everyone, in as orderly a manner as possible, place the cards in your model by usage sequence, with features used early on the left and later on the right. Have them overlap features that might happen at about the same point in time. If someone gets confused about the position of a feature, suggest that he look at the feature and its immediate neighbors. It's sometimes easier to answer the question "does this happen before that" than to try to take everything into account at once.

If we arranged our feature cards in sequence, it might look a bit like the diagram in Figure 3.

STEP 4: GROUP BY FREQUENCY

For each of these features, how critical to our business is it that someone actually uses it? Let's look at our retail features: When working with the business people who know how their business is run, they inform us that orders often are placed with vendors informally over the phone without a purchase order being created in the system. So in those cases, we'll receive the items into inventory without a PO. This is generally the exception, but it happens and should be supported. So our feature "create PO for vendor" is important to our system and is used frequently, but not *always*.

As a group, adjust vertical positioning of your cards based on how critical they are to the business process. If the feature is always done, place it on the top. If the feature is often —but not always—done, place it a bit below the top line. If it's seldom done, place it toward the bottom. If you've got enough people working on the model simultaneously, this may start to look like a game of Twister. You'll observe people moving cards down only to see them adjusted back up by someone else. Use these conflicting card movements to elicit discussions on why someone might believe a particular feature is more critical than another feature.

If we adjust our features for criticality, our model might look a bit like the one in Figure 4.

STEP 5: NOTE LOGICAL BREAKS IN WORKFLOW

If your system is anything like those I've worked on, you'll have knitted together a few distinct processes done by different people at different times. When you look across your model from left to right, you might start to see logical breaks in the workflow. Remember how for each feature you noted a type of user or role that primarily used the feature? You'll find that these breaks often occur when there's a role change. Reading left to right you'll see some features are used by one role, then you'll see a change to another role with some features used by this next role.

As a group, discuss where you see breaks or pauses in the business process. Then, at each break draw a vertical line, dividing the model into something that looks a bit like swim lanes. Label the resulting columns for each process as shown in Figure 5. If you're finding it hard to divide the model in this way, discuss why. Is there really only one type of user doing one process? Or, do we have different user's features mixed up in the same timeline?

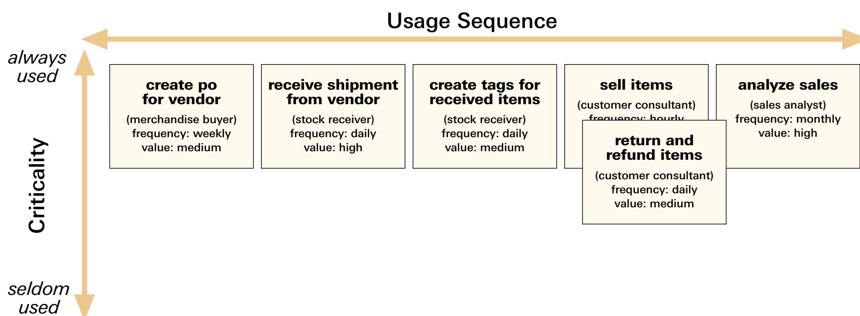

Figure 3: The features cards are first arranged by sequence.

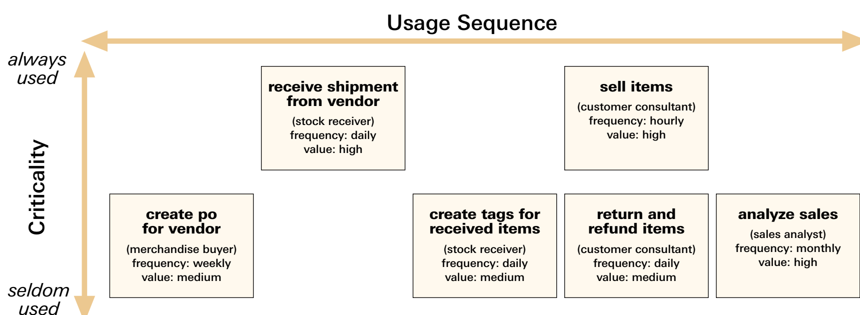

Figure 4: The sequential feature cards are then staggered according to criticality.

page 21
full-page ad
SQE

This is all quite interesting so far, but what are you learning about how to build releases for your software? Let's take a closer look.

STEP 6: MARK THE FIRST SYSTEM SPAN

Now we're going to divide the model into what I call system spans. (The Poppendiecks introduced me to the term "span" in their book *Lean Software Development*.) A system span is a set of features that group together logically and that cut through the business processes horizontally from start to finish.

The first span should be the *smallest* set of features necessary to be minimally useful in a business context. In our model, it turns out that the very top row (receiving and selling items) is the first, most minimal system span. This will be true of your model, too. Draw a line under the top row of your model to indicate the features that make up this first system span, as shown in Figure 6.

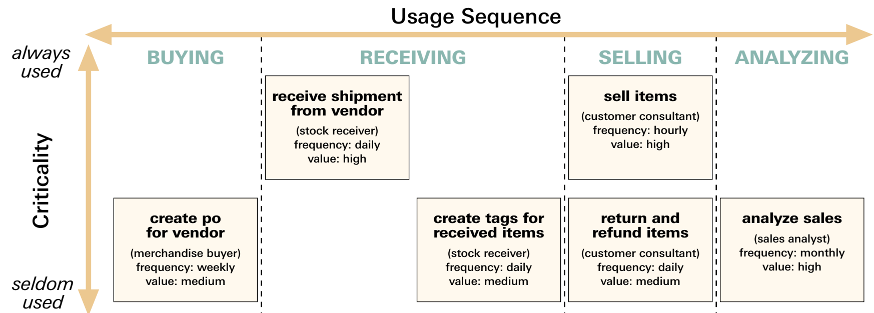

Figure 5: The model is vertically divided into business processes.

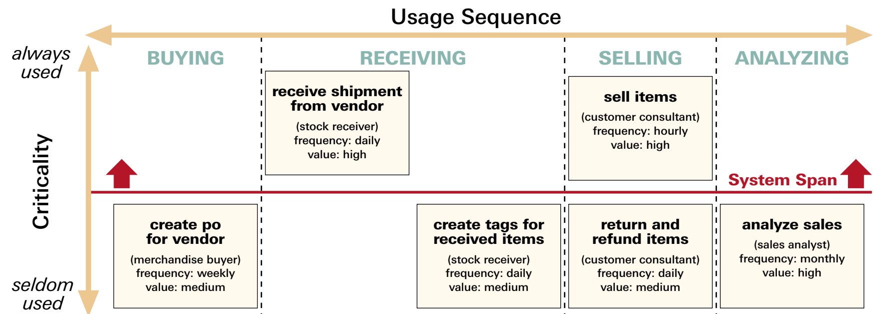

Figure 6: The first system span represents the smallest set of features necessary to be minimally useful in a business context.

The first span represents the most concise set of features that tunnel through the system's functionality from end to end—the bare bones minimum anyone could legitimately do and still use the system. This small span should always be your first release, but it need not be the first *public* release of your software. Getting this part completed and released, even if only to internal test environments, forces resolution of both the functional and technical framework of your application. Your testers will be able to see if the application hangs together coherently. Your architects will be able to validate the tech-stack functions as expected and may begin working on load tests to validate scalability. The team can begin to relax knowing that from here on in they're adding more features to a system that can be released and likely used.

Notice how in the example we've not built any functionality to support the merchandise buyer or the sales analyst. Ultimately, we know that supporting those folks with some functionality is important. But, because the work they're doing doesn't always happen, we can defer it at least for a little while.

After drawing the line in your model are there roles and business processes that are omitted? Talk about them as a group.

STEP 7: FILL IN BUILD ESTIMATES

OK, so building a first span is a good idea, but how long will it take? If you've got developers participating in this exercise, and you should, this is a good time for them to start giving development estimates for each feature. Write the time estimates in days or weeks directly on the cards. Very rough estimates will do fine. Developers may find that seeing the "big picture" helps them estimate a little better. (See this issue's StickyNotes for more guidance on quick estimation techniques.)

Once you have rough estimates, add up the estimates for the features above the line marking the first span. This is how long it should take to build.

> A system span is a set of features **that group together logically** and that cut through the business processes horizontally from start to finish.

STEP 8: SLICE AND SERVE

The plan may now be "sliced" horizontally into subsequent system spans, or spanning releases. Well, sort of horizontally. Choose features below the marked first span that group together logically. Choose enough features such that the estimated elapsed development days fit within an appropriate release date. This may cause you to draw lines that wander up and down to catch and miss features while traversing the model from left to right. Lines drawn through the plan make it start to look like a haphazard layer cake.

At this point in the collaborative activity, the business people responsible for the release should step forward. Let them use their best judgment to decide what features best make up a release. If you're an observer, ask questions so you understand why one feature rather than

(Continued on page 40)

page 23
full-page ad
SPI Dynamics

(Continued from page 22)
another finds its way into an earlier release.

Responsible business people continue to slice your "cake" into appropriate releases. When choosing features to fill a release, you may want to consider the features with the highest value first. You may also want to consider building up support for a particular type of user or business process. In a release, you might try completing all the valuable features in one of your business process columns. This will result in some funny-shaped lines stretching from left to right.

After slicing the model into releases, you should be able to see how many releases it will take to build this software and what might be contained in each release.

Now, let's get real. Most software worth writing has more than six features. Depending on the granularity of your features, you'll likely have dozens. With a reasonable number of features your plan will likely look like the photo shown in Figure 7. Notice in this model that software spans several business processes. Notice how the releases cut from left to right in some funny jagged lines that catch the features the planner intended for each release.

Figure 7: A real-life model is much more complex and spans several business processes.

THE RESULTS

Because you've arranged features in sequential order, you now understand what features depend on one another. Because you've arranged them by criticality, the important features are now emphasized at the top of the plan. Because you've divided the features into business processes, you have a better idea of the functionality that supports each major business process in your software. You have determined the minimal feature span that lets you get your system up and running, end to end, as soon as possible. All this information is provided in one convenient picture. By employing a little common sense, we should be able to carve off the smallest possible releases that will still be useful to the people who ultimately receive those releases.

Incremental release may be one of the more valuable aspects of the various Agile development methodologies. An early release can help capture market share, generate early return on investment, and reduce money risked on software development. A strong early release can increase those benefits immensely. The model we've built can give you a better picture of your software's features and help your organization construct the most useful and coherent early release possible. **{end}**

Jeff Patton leads teams of Agile developers to build the best software possible. He proudly works at ThoughtWorks.

1/3
square
left

Rally