Incremental Releases

Users and Stakeholders Will Love

How to deliver functionally complete, valuable, incremental releases

Handout supplement

Jeff Patton
ThoughtWorks
jpatton@thoughtworks.com

Handout Contents:

Speaker Information

Jeff Patton has designed and developed software for the past 12 years on a wide variety of projects from on-line aircraft parts ordering to electronic medical records. Since first working on an XP team in 2000, Jeff has been heavily involved in Agile methods. In particular Jeff has focused on the application of user centered design techniques to drive design in Agile projects resulting leaner more collaborative forms of traditional UCD practices. Jeff has found that adding UCD thinking to Agile approaches of incremental development and story card writing not only makes those tasks easier but results in much higher quality software.

Some of his recent writing on the subject can be found at www.abstractics.com/papers and in Alistair Cockburn’s Crystal Clear. Jeff’s currently a proud employee of ThoughtWorks, an early adopter and leader in Agile Development approaches. Jeff is founder and list moderator of the agile-usability discussion group on Yahoo Groups.

Website and blog: www.agileproductdesign.com

Discussion group: tech.groups.yahoo.com/group/agile-usability

Agile Software Development from 10,000 Feet

The Agile Methodology Isn't

Before we get too far along, it's important to underscore one particular point: **Agile Software Development isn't a specific methodology.** Rather, Agile Software Development refers to a general class of software development approaches.

In 2001 a group of thought leaders using a variety of light weight, low formality software development approaches met to discuss what, if anything, they had in common. The result of that discussion was a common core set of values and principles described in the Agile Manifesto. A specific approach might be considered Agile if it honors those values and principles.

If you own any books regarding Agile development methods, or you've done a little research, you've probably read the Agile manifesto already, but I'll include it here for you to ponder. The Agile manifesto is expressed as a series of four simple paired value statements where both items in the pair have value, but one item is valued more than the other.

> **Individuals and interactions** over process and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan

When determining if a methodology is Agile, I find that I have to use the pornography rule as quoted by Supreme Court justice Potter in 1964: "I can't define pornography, but I know it when I see it." The same seems to be true of Agile Development methodologies. Understand Agile values and you'll know an Agile methodology when you see one.

For many these value statements read like motherhood statements – simple truths hard to disagree with. Others believe that the "this over that" format incorrectly asserts that one side of the statement is at odds with the other – for example that valuing individuals and their interactions is somehow at odds with valuing a solid process and effective tools. If one of the goals for the Agile Manifest was to cause discussion about these things, it's indeed accomplished that. And oddly what comes out of discussions about the value of people, collaboration, working software, and responding to change are flexible resilient methodologies, effective tools, innovative approaches to documentation, the rethinking of the contract, and new approaches to planning and project management. Go figure.

There are a few named Agile Methodologies very well described in books written by their creators and/or practitioners: Extreme Programming (Beck, 1999), Scrum (Schwaber and Beedle, 2000), Feature Driven Development (Coad, LeFebvre & DeLuca, 1999), and Crystal (Cockburn, 1999-2004). All these books describe methodologies that share the common value system expressed in the Agile manifesto.

I'm confident that there are named Agile approaches that I've missed. I'm equally confident that there are a multitude of variations currently in use within a number of software development organizations. I often meet people and talk with them about their development approach. After just a little conversation with them, it's easy to describe what they're doing as Agile. They've adopted and adapted a variety of processes and techniques to best fit their organization. Along the way, and often without specific intent, they've used what we're calling Agile values to guide their approaches. Agile Development didn't invent those values – just gave us a single phrase to refer to them by.

Those foundational values are the important thing. It's quite possible to adopt a named Agile approach such as XP or Scrum and use all the techniques, but do so in a spirit contrary to Agile values. So while process, tools, documentation, contracts, and plans are valuable and many Agile approaches describe approaches for all of them, it's the soft stuff – people, collaboration, responsiveness - Agile Development emphasizes that makes the difference.

This book describes the innovative techniques and approaches that emerge when the Agile value system is applied to the process of software product design and requirements.

General Specifics:

If you're new to Agile Development approaches, I might have led you to believe that there's little consistency among Agile approaches. That's not exactly true. There are a few durable concepts that can be found in most Agile approaches – sort of an emerging Agile best practices. When we combine these best practices, and step far enough away, most Agile approaches actually begin to look very similar. If we're going to work with requirements in an Agile context we need to understand these common ideas. Let's discuss a model that gives structure to these common ideas.

Feature Development:

[I don't like the term feature – it has baggage in that it indicates a solution to a problem – not an unsolved problem]

Most Agile development approaches break work down into small hopefully independent pieces of work. XP might call this a "user story," while Scrum might call it a "backlog item." I'll use the somewhat neutral term "feature" here. An Agile feature will dependably have a few qualities:

Customer-Centric:

A feature is described from the perspective of the people or group of people requesting it and not the language of the engineering group creating it.

Value:

The feature will have some understandable value to its user or the organization purchasing the software.

Cost:

Enough is understood about the feature so that the cost of developing, testing, and integrating the feature into the software can be *roughly* estimated by a development team.

Verifiable:

Once the feature is added to the system, there will be a way to test that the feature is in place and behaving as expected. This test may take the form of observation or manual use by a human or automated test case executed by a computer system.

Features are developed in repeating cycles of designing, building, and testing. When designing we'll decide a few things about the feature. We'll build those few things. Then we'll test that the feature meets the small amount of design we've done. We'll cycle through these three steps a number of times before we can call the feature completed.

Sometimes a feature might be broken down into smaller development tasks completed by different developers. Each task might spin through the design-develop-test cycle a number of times before they're all integrated together to form a completed feature.

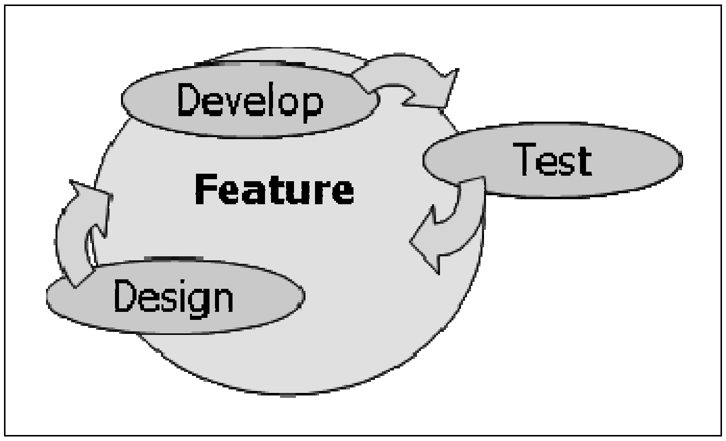


Figure 1.1: feature development

Iterative Development

Agile development uses an iteration to build features. An iteration is a time box with specific start and end dates. Iterations are commonly two to three weeks in length, but might vary shorter or longer in any specific organization – although much longer than a month is uncommon.

An iteration is preceded by a **planning session** to choose the features that will be built within the time box. This activity is normally collaborative and involves developers, testers, and business people requesting the features.

With a plan in place the iteration begins and developers start work on features business people have chosen to build.

As the end of the iteration time box approaches, iteration testing might begin. Though each feature was tested independently, the group of features are generally tested together as a coherent whole.

Test Driven Development

The development technique: Test Driven Development seems to indicate that we're writing the tests prior to writing the code – which seems to break the design-develop-test cycle suggested. But that's not exactly true.

Test driven development has a software developer describing the expected functionality of a piece of code in an executable unit test prior to writing that code. Once the code is written correctly, the executable unit test will pass.

Test driven development isn't actually testing, but designing.

The code I might write in my unit test describes the design I'd like to see reflected in the code I'll write. The running unit test doesn't confirm my code is bug free, just that it meets the design I described in my unit test. Test Driven Development is a thinking technique; a designing technique.

You'll find many Agile approaches recommend writing tests prior to writing functional code – both unit tests for individual bits of code, and acceptance tests for features visible to end-users. The act of writing these tests is an act of describing functionality – an act of design. Depending on how thoroughly the tests were written, executing them after the code was written might be enough to call that code tested. In practice it's common to look back at the running code, and the tests that go with it and ask "what could possibly go wrong with this code?" At that time additional tests might be added and executed to support those conditions – to thoroughly *test* the functionality.

When the time box ends, the iteration ends. Uncompleted features are carried forward for consideration in the next iteration's planning session.

Wrapping feature development with iterative development grows our model to look like figure 1.2.

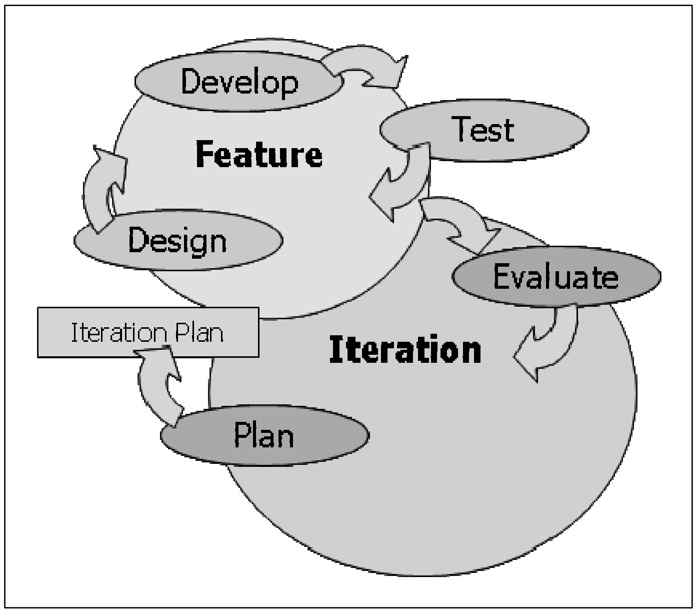


Figure 1.2: Iterative development of features

Increment to Release

Short iterations give more frequent opportunities to gauge development performance, evaluate the current state of the product, and adjust features and priority. It may be difficult in this short amount of time to build a set of features complete enough to be placed into use by actual end-users. In situations such as this, it's helpful to package several small increments into a software release.

A release is preceded by a **release planning session** where one or more software releases are planned. During the planning session the features for each release are determined with an emphasis on choosing a set of features that makes each release useful to the people who'll receive it. During release planning, rough estimates for development time by feature are made. Using these estimates, release dates can be forecast along with the number of iterative development cycles it might take to complete a release.

With a release plan in place the features for a release are fed forward to **iteration planning** where features will be chosen to be built in the first iteration. The rhythm of iterative development kicks in and the features of the release are built an iterative chunk at a time.

As the end of the release date approaches, release testing might begin. Just as features were combined during iterations and looked at as a coherent whole, all the finished features for the release will be looked at and evaluated as a coherent whole with a special emphasis on the usefulness of the group of features in the release. Will the release be useful? Will it be marketable?

Wrapping iterative development with incremental releases grows our model to look like figure 1.3.

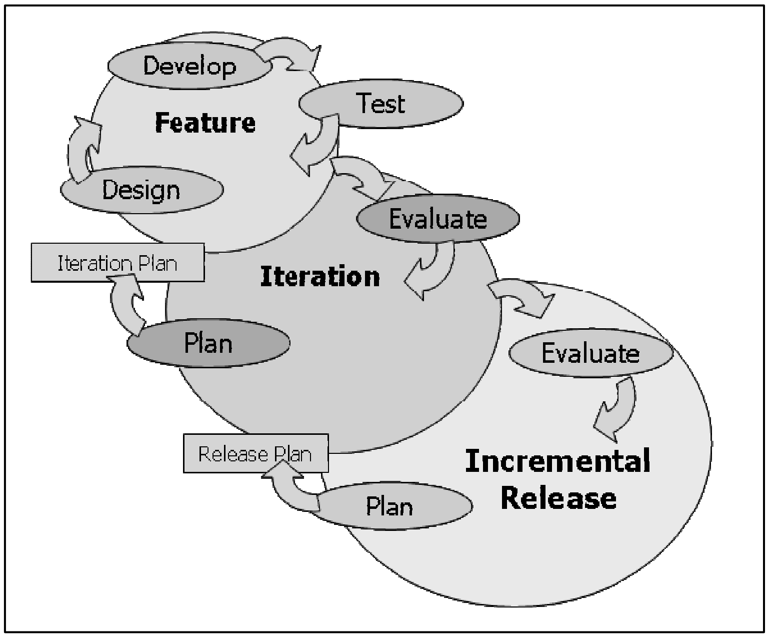


Figure 1.3: Incrementally released, iteratively developed features

Products and Projects

Every release contributes to further the goals and vision of a product or project. If we're doing this work in support of a product company, we're keen to make sure that each release helps us gain or at least hold onto market share. It's important that each release furthers the goals of the product. If we're releasing against an internal IT project it's important that each release support the goals of the project which are usually to increase the productivity of the organization paying for the development.

A product or project is usually driven by a set of high level goals or a **project charter**. This charter states the goals for the software. These high level goals might be expressed as financial objectives for product sales or increased efficiency along with a plan for how those objectives might be realized.

A product's life might span years or decades. Hopefully projects won't live that long. But as releases of the product are built and delivered the product/project is evaluated against the goals established for it in the charter.

Wrapping incremental release with product and project chartering might change our model to look like figure 1.4:

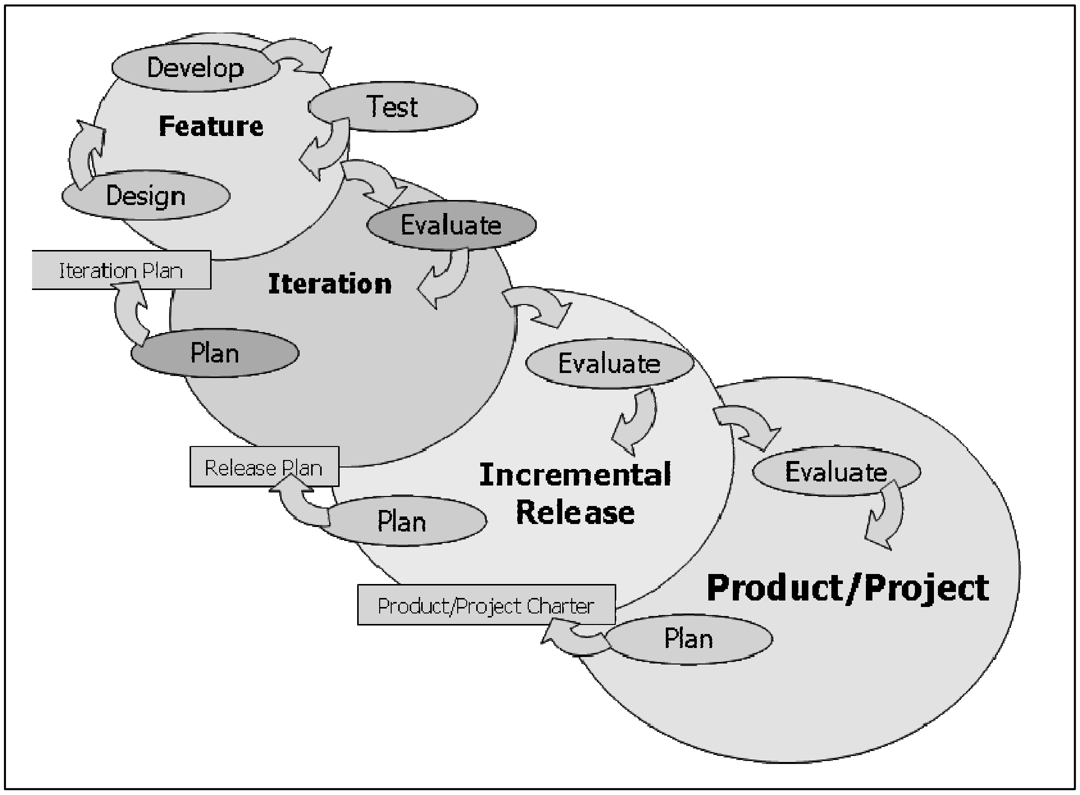


Figure 1.4: Chartering an incrementally released product or project

The Agile Waltz

As I'm writing this someplace in the back of my mind I'm recalling my feeble approaches at ballroom dancing. I'm hearing a voice in my head chanting "One-two-three, one-two-three..." as my feet try to remember which way to step when dancing to a waltz. As I think about Agile development I'm seeing that one-two-three waltz tempo repeated in the cycles I've described above "plan-build-test, "plan-build-test…" Agile development, when done well, is as rhythmic as a good waltz.

Think about dancing to a nice waltz for a minute. (If you're not a good dancer, go ahead and imagine that you are.) When you're done, let's look back at the model in figure 1.4.

The Recursive Build

This is where our dance metaphor breaks.

You might notice that the feature is a three step process, but iterations, releases, and projects might look like they only have two steps. The feature has a step labeled "develop" where the feature we've conceived of is coded in some programming language. This is the "building" part of the features. You'll notice that the feature "bubble" on our model is glued into the iteration bubble. Iterations are built out of features – or design, developing, and testing features is the "building" part of our iteration. Continuing out, releases are built from iterations, and products and projects are built from releases. The building part of each cycle is accomplished by the smaller cycles it encloses.

At the building bit of each cycle we start by planning again, then building, then evaluating. At the feature level, the bubble marked "develop" over-simplifies what's really going on there. In reality an Agile developer will cycle through the plan-build-test cycle many times in the process of development. A developer might do a tiny bit of planning or design by writing some unit test code. That would be followed by writing some actual software code. That would be followed by running the unit test code to test that the design worked as hoped. One-two-three, One-two-three….

The Big Plan Up Front

You might be starting to notice how much planning is done in Agile development. Every cycle begins with a plan. In a new product or project all the planning bits of a cycle feed into each other. Plan after plan after plan after plan. In our model of swirling bubbles, it might be a little hard to see what's happening over time. In figure 1.5, we'll squash it flat and take a look.

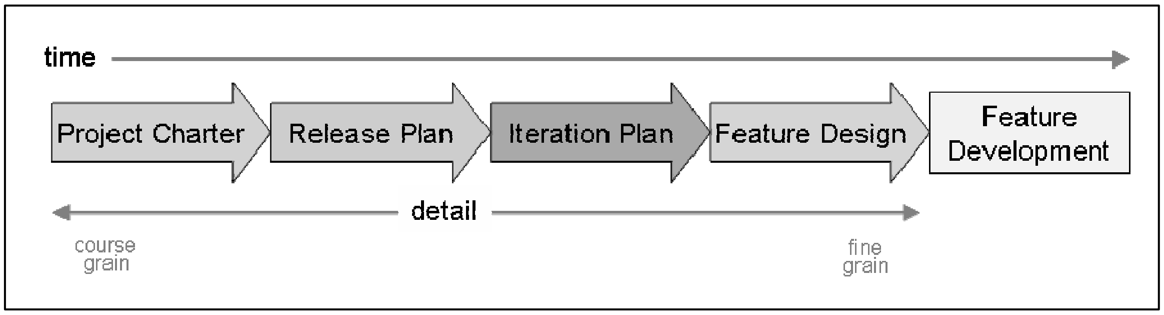


Figure 1.5: Granularity of planning over time.

In figure 1.5 you're seeing how each cycle's planning precedes and feeds into the next. What's important to note is that the granularity of planning changes for each cycle with the planning step for each cycle growing more detailed or fine grain as the cycle time decreases in length.

Design is a Synonym for Plan

You might notice that I've been a little loose with my use of the terms "plan" and "design." In my head they used to be very different things. Planning was an approach I used to schedule activities and resolve their dependencies in order to reach a goal. Design was a creative process where I'd choose specific qualities of a thing it might have in order for it to satisfy a goal. The difference might have been the accepted use of creativity in design vs. a firmer analytical approach that I might adopt when planning. Over the years the boundaries for me have begun to blur.

As I've learned more about user centered design approaches, software design has adopted a flavor of being more analytic – analytic without losing its dependence on creativity. Using the analysis models and techniques supplied in user centered design approaches has allowed me to be more successfully creative.

As I've run projects over the last decade, I've seen that the job of keeping a project on course and dependably delivering requires much more creativity and intuition than I'd originally suspected.

Merriam Webster clearly calls out design as a synonym for plan – which, given my previous views confused me. Today I'm seeing software development as a continuous process of design and planning, followed by doing and evaluating. One-two-three, one-two-three…

Given that we'll be doing so much continuous design, how does a user-centered design approach lend itself to our need to segment design into packages that vary in granularity from course to fine grain? Jesse James Garrett's User Experience model discussed in the next chapter will help to visualize that.

Feedback, Feedback, Feedback

You might also notice that each cycle's testing step feeds the next higher cycles'. Individual features are tested against the original goals described for the feature. All iteration features are tested together with respect to each other and the goals of the iteration. The results of all iterations are tested together with respect to the goals of the product.

As with planning and design, testing is done continuously and at different levels of granularity from the smallest individual feature to the product as a coherent whole.

Test is a Synonym for Evaluate

OK, it's not really a synonym from the dictionary's perspective, but I'll use it that way here. It's important to understand that all this testing isn't about the features merely being implemented and working as expected, but about the features accomplishing the goals set out for the product, release, and iteration. It's quite possible to have a feature that works as designed and required, but simply doesn't have the qualities predicted for the product and release. It's quite possible for all features designed and required to be implemented for a release, but for that release not to fulfill the goals of end users or business people who paid for the development. At the end of each cycle of feature development, iterative development, or release development, it's critical to evaluate the value of the feature relative to the goals of the product, release, and iteration.

The results of all this testing and evaluation feed back into the iteration plan, release plan, or project charter.

By feeding back, I mean that the tangible results from feature development, iteration development, and product release might indeed change features, iteration plans, release plans, and project charters. Yes, I mean they might change requirements. Evaluation cycles give us a chance to learn from what we've done thus far and adapt early to information we didn't know when initially planning and designing.

These feedback cycles that seem to invite change to requirements are an example of a process adaptation to the Agile value of responding to change over following a plan. Effectively gathering and responding to feedback is one of Agile development's defining characteristics, and most troublesome problems to solve.

The Other Important Thing to Evaluate

A common technique used in Agile approaches is a Reflection or Retrospective session. Normally this takes place at the end of an iteration or release. One of the primary goals of the reflection session is to look back at the specific process followed over the last cycles and determine if the development approach is meeting its intended goals. Just as the evaluation of features may result in changes to the project plans or requirements, evaluating process may result in changes to process or tools.

[references on reflection, forward reference for reflection technique discussion]

But Wait, There's More…

The Agile Development model discussed here stops at the product and project. But, we all know the world is a bit bigger than that. Products and projects usually exist in a portfolio of possible products and projects. The decisions we make about specific products are usually made in the context of other products and the time resources available to them. The portfolio level decisions are made against the backdrop of the company who's paying for the products and their general goals, financial or otherwise. If that company is owned by a larger parent company, that company's general goals and direction is help set by the parent company. The concentric cycles continue to radiate up and out. In all cycles there's an element of decision making and planning, an element of execution and building, and finally an element of evaluation, reflection, and re-planning.

Agile Development Distilled

Agile development may be characterized by short cycles of planning and design, building, and testing and evaluation.

Cycles may be at a very detailed and tangible level – like the design, building, and testing of a single feature. Cycles may be at a higher abstract level like the design and planning of a product release, its construction, and subsequent evaluation of the finished product.

Much of the challenge in Agile development is found in the area of planning, design, test, and evaluation. How can we plan at a high level and defer the design of small feature details for later? How can we frequently evaluate what we've built against our high level plans and adjust those plans accordingly?

Additional Reading

- Beck, **Extreme Programming Explained** (Addison-Wesley, 1999)
- Beck, **Extreme Programming Explained 2nd Edition (**Addison-Wesley, 2004)
- Cockburn, **Crystal Clear: A Human-Powered Methodology for Small Teams (**Addison-Wesley, 2004)
- Cockburn, **Agile Software Development 2nd Edition** (Addison-Wesley, 2006)
- DSDM Consortium, **DSDM: Business Focused Development (**Pearson Education, 2003)
- Highsmith, **Adaptive Software Development: A Collaborative Approach to Managing Complex Systems** (Dorset House, 1999)
- Larman, **Agile and Iterative Development: A Manager's Guide** (Addison-Wesley, 2003)
- Palmer & Felsing, **A Practical Guide to Feature-Driven Development** (Prentice Hall, 2002)
- Poppendiek & Poppendiek, **Lean Software Development** (Addison Wesley, 2003)
- Schwaber & Beedle, **Agile Software Development with SCRUM** (Prentice Hall, 2001)

Our Design Problem: Barney's Information Kiosk Project

You're on the in-house software development team for Barney's Media. You and your team have been called in by the operations management team to discuss a new piece of software that Barney's needs written. Before you can ask too many questions, the operations management people start telling you what they want.

They remind you that: Barney's is a new but growing national retail chain. Their stores contain, on average, 4000 square feet of floor space housing over 20,000 unique titles of both new and used CDs, DVDs, and video games. While the store is conducive to browsing, it's tough at times to find a particular item quickly. Customers who know what they want have a hard time finding it in the store. Customers have a choice between new and used items and often an item that isn't in stock as used may be available new, or vice versa. Often the title they're looking for isn't in this store at all, but may be in another Barney's store or could easily be special ordered. In those cases Barney's would like to special order the item for them.

Today the only way to get help locating or special ordering an item is to wait in line for a cashier, or trap a sales associate in the aisles. Currently, sales associates hate to leave the safety of the cashier desk. A walk from the cashier desk to the back office can often take 10 minutes to a half hour as a result of all the folks stopping them to ask for help finding an item. The folks at the information desk stay pretty busy all day fielding questions as well.

While discussing the domain consider:

- Who are the people who will be using this system?
- Why would they use it?
- What goals do they have?
- What kinds of activities might they do to meet their goals?
- What happens if they don't meet their goals? – Who loses?
- What happens when they do meet their goals? – Who wins?
- Are there users who monitor and protect the interests of other users?

The management of Barney's believes they can enhance the customer's experience at the store and ultimately sell more product by creating self-service touch-screen information kiosks within the store. At these kiosks, customers could get answers to their questions about the items they're looking for, their availability, and their location in the store. Optionally if the item isn't in stock, they could arrange for it to be special ordered or sent to them from another Barney's location or set aside at that location for pickup.

The types of customers coming into the store vary immensely. Some may be very comfortable with information kiosks while others may have never used one before. Some may be using the kiosk to quickly find a CD, DVD or game; others may be using it as an alternative way to browse the available titles in the store.

Executives at the Barney's corporate office believe they can enhance store sales by "suggesting" like-titles to customers looking for a specific title. They believe they can enhance store sales by encouraging customers to special order titles they don't currently have in stock. They believe it would be valuable to know how often customers look up or ask for titles not currently in stock. They also believe they can reduce labor costs at the store a bit by allowing customers to help themselves. So these executives can feel comfortable they've spent their money wisely, they'll expect statistics on how many people, by location, are using the kiosks. It would be valuable to know how many times customers looked closer at suggestions made by the kiosk. It would be valuable to know how many special orders were placed through the kiosk.

Your design and development team has been given the task to design and build this new information kiosk. Barney's already has massive databases of the items they carry, inventory systems that tell them which and how many items are in stock at each location, and order entry systems to place special orders. Your team will need to integrate this information in a piece of kiosk software.

The operations management team doesn't have specific functional requirements past those discussed here. They're looking for an estimate from your design and development team that suggests the functionality they should build and the timeframe it will take to build it. They'd like to see a functional kiosk in stores as soon as possible. In fact, if you can't get something functional in stores within a couple months, we'll outsource it to another team that can.

What will you build? And, how soon before we can see something running and put it into pilot stores?

Show me the money:

While considering requirement, many designers make the mistake of only considering direct users of the software. Those with concerns about the effectiveness of the software and/or its ROI are often set aside as "stakeholders." Stakeholder concerns are often addressed in the software's detail design by ensuring the software captures necessary information to meet stakeholder needs.

Consider promoting your stakeholders to users. If the stakeholder were a user, what could the software do to demonstrate to these stakeholder-users how much money it's earning for them? Should stakeholder-users have access to current information on software performance? Should they receive warning when things aren't going well?

For more information see:
http://www.abstractics.com/papers/showMeTheMoney.pdf

User Task

Concept

A physical action taken by a person in pursuit of a goal. Name tasks without implying that the task must be performed a particular way or be performed using a particular tool to allow choices about the tool and its design to be made later.

Represent the Work People Do as Tasks

Understanding a User Task

One early morning I sat with a friend at breakfast. Being a bit of a coffee snob, he didn't trust the hotel restaurant where we were eating to have good coffee, so he'd stopped at a coffee shop on the way in [you know the brand] and picked up a big cup of coffee. Together we sat at breakfast talking; me with my hotel restaurant coffee in a white ceramic cup with a handle, him with his likely better coffee in a tall paper cup with a plastic lid and a brown cardboard sleeve to prevent him from burning his hand.

We both shared the goal to enjoy a hot tasty beverage with our breakfast. In response to that goal he stopped at a coffee shop and bought coffee before meeting me. I chose to order coffee form the waiter at the hotel restaurant. But we both bought coffee. And, during breakfast, we both drank coffee.

There's a point to my story, honestly.

Tasks represent the little bits of work we do to move us toward a goal. Two of the tasks my friend and I shared this morning were: "buy coffee" and "drink coffee." Although we went about at least one of our tasks in very different ways, and with different results, we shared the same tasks.

- **Tasks represent physical actions taken by person in pursuit of a goal.**

Tasks contain other tasks

Tasks are interesting things. If we were to take a task, place it in the floor and hit it with a big sledge hammer it would likely splinter into a number of pieces. If you were to pick up each one of these pieces and look at it, you'd find each piece was also a task. The tasks contained inside of other tasks compose the steps I might take to achieve the larger task that contains them.

For example at breakfast to accomplish my task of "buying coffee" I might first get the attention of the waiter and next ask the waiter for coffee. The waiter might then ask me if I'd like cream and sugar to which I'd respond "yes please." Later on I might check that the coffee was on my breakfast bill and check the price. Then I'd pay the bill by charging it to my hotel room. If I look at the buying coffee task, it might contain a list of tasks like this:

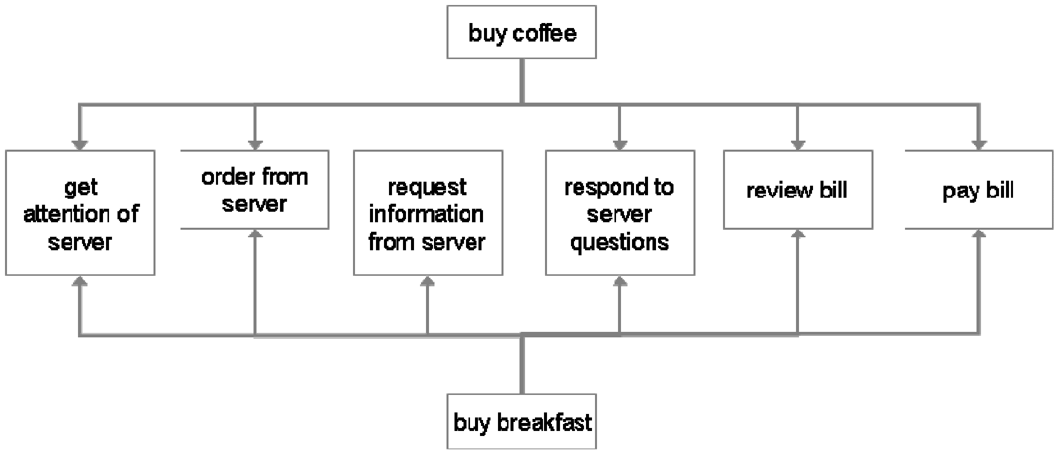
Buying coffee

- Get attention of server
- Order coffee
- Respond to server's inquiries regarding coffee options
- Review coffee bill
- Pay coffee bill

Small tasks recombine to support other tasks

The example above is really a bit of a lie. It's true I ordered coffee, but I actually ordered coffee as part of ordering breakfast. As part of ordering breakfast, I chose a main dish, asked the server questions about it, ordered coffee, and ordered toast. My friend also ordered breakfast at the same time. When it came time to pay the bill, although there was coffee on the bill, it contained breakfast for both my friend and I, and since it was my turn I paid the bill for all of breakfast – not just the coffee.

The tasks to order coffee, review the bill, and pay the bill were essentially the same tasks as if I'd just ordered coffee, but they were recombined in support of a larger task to buy breakfast.



Tasks use short term goals to support the pursuit of intangible personal goals

Tasks support the pursuit of larger, often less tangible goals that are closely associated with the person engaged in the task. However, the tasks themselves generally have a smaller more easily evaluated goals. Think of these smaller goals as success criteria. I'll know when I'm successful buying coffee because I'll see it right there in my hand. If I bought coffee in support of the less tangible goal of having a pleasant morning – my evaluation of whether my morning is pleasant or not might change from moment to moment.

My head constantly evaluates this intangible personal goal "Is my morning pleasant?" If my morning isn't pleasant, or could be more pleasant, what might I do? You might have guessed it already… some kind of task with a smaller more tangible goal. Achieving that smaller goal allows me to reevaluate my standing on the larger goal.

[insert and discuss Don Norman's model from the design of everyday things here.]

Tasks have an abstraction level

We often express tasks abstractly. By that I mean I say generally what I intend to do, not precisely. In my previous examples I've said "buy coffee." I could have said "buy coffee to take out from the counter of a corner coffee franchise store that serves gourmet style coffees." That's more precise – more concrete. It's also a mouthful. I could have also said "buy hot breakfast beverage," or simply "buy beverage." That's pretty imprecise, pretty abstract, so much so that it may not be useful to help me visualize how I might go about it if I were a person trying to complete such a task.

You'll find that if you express tasks in writing, you'll likely do some generalizing about that task. This generalization doesn't omit details, rather it defers specifying details. It leaves the specification for a later activity, an activity that's a design or decision making activity.

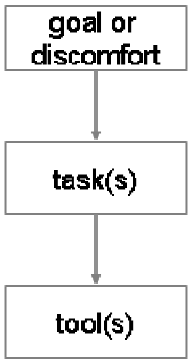
Keeping tasks written abstractly allows us to rewind design decisions. If I make some specific choices about how a person would complete a task, I may at some later time decide those weren't the best choices and wish to consider alternatives. Looking back to and starting design from the more abstract task name allows me to start to think about those other alternatives.

Expressing tasks too generally isn't helpful. When looking at the tasks most commonly done in software, it's easy to abstract things up to tasks like: create information, update information, delete information, find information, review information. While at an abstract level that may be what I'm doing, it's certainly not enough detail to allow me to effectively make more detailed design decisions.

> *[Alistair's sea level metaphor works well here. Is there another metaphor that would help me identify a task of a suitable abstraction level?]*

Tasks use tools

Humans engage in tasks as part of an activity in support of goals or to relieve some discomfort. Once tasks are undertaken, we'll look for tools in our environment to leverage. Tools may be something concrete like a hammer, paperclip, or computer, or less tangible like business processes, or rules of thumb. If you're helping to build software, you're in the tool business. You help to build tools that people will hopefully use to make their tasks go more smoothly. The design work is in choosing a best tool to accomplish this secondary goal of making the task go more smoothly.



It's important to keep in mind that the goal of completing the task itself is a secondary goal. No one completes a task for the sake of doing so. It's the goal that motivated the task in the first place that's important.

Looking at the coffee example we started with, I and my friend both bought and drank coffee. The tools he used were a corner gourmet coffee shop, and a paper cup with a lid. The tools I used were a hotel restaurant and a ceramic coffee cup – without a lid. Someone put some thought into designing these tools for us. Someone gave a little thought to our goals, and our context of use and made choices to use ceramic vs. paper, corner coffee shop vs. restaurant. The design choices they made were tuned for a particular type of user/consumer in a particular context.

The software we choose to build has the same forces at work: people in a particular context of use trying to reach a particular goal. To help us better recognize when we're designing, it's valuable to separate the tool from the task that motivates its design and construction.

- **Tasks are not tools. Tools support the execution of tasks.**

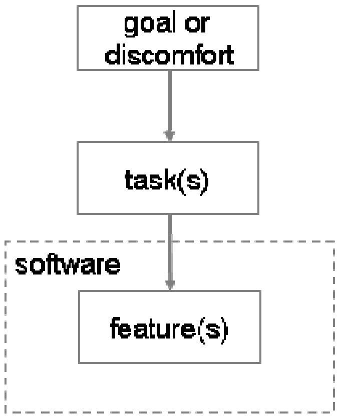
Stepping one level back, it's important to separate the tasks people choose to execute from the goals they actually hold. Specific individuals choose a wide variety of tasks to reach their goals. In an ideal world the task would be as effortless and automatic as possible – it would almost go away.

I used to have a task to mow my lawn. Well I still sort of do. But, with all the traveling I've done lately I've hired a service to mow my lawn. Now the task of mowing my lawn has mutated to paying the monthly lawn care bill. The lawn still gets mowed but my specific task changed. And my goal of feeling good about the curb appeal of my house is still met.

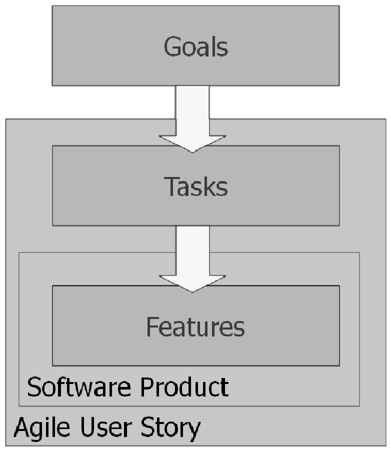
Software is a big multi-function tool

While the information here about tasks may be useful for all sorts of things, from buying coffee to building restaurants or airplanes, the reason you're likely reading it is because you're involved in some way with building software. Software is one of the tools people reach for to perform tasks in pursuit of their goals, but it's a big tool. Picture a garage full of tools. Software is more like a garage outfitted to support woodworking, auto repair, or practicing and recording punk rock. Like software, there are lots of things you can do with a garage. The tools you outfit your garage with – like wood lathes, ratchet wrenches, or Fender guitars - hint at what goals and tasks it supports. It's likely that people go to your software like they go to their garage. They'll spend some time there engaged in a number of tasks in pursuit of a number of goals.

In software we can refer to the stuff we build to support specific tasks as functions or features. In this book we'll use the term **feature**. Throughout this book you'll hear repeated the natural dependency between the goals a person wishes to achieve or discomfort to eliminate, the tasks they choose to pursue to do so, and the software features we choose, design, and develop to help them.



In Agile development approaches it's common to use the term **user story** to refer to bits of software we plan on building. It's a good term because as the name implies, it should be a story told by our users that should contain information about who they are, what the goal is, and what they might do to reach it. As we approach development time, the user story may come to mean the specific feature we intend to build. I'm not going to regularly use the term user story partly because all Agile development methodologies don't use the term, and because the term could be used to indicate a user task, a product feature, or a mixture of both.



Available tools change the way we think about tasks

On a daily basis I brush and floss. Those are tasks I perform. Notice how the name of the task *is* the name of the tool I use? I call the task brushing my teeth because the tool I use is a tooth brush. Same thing goes for flossing my teeth. Really what I'm trying to accomplish with both these tasks in keeping my teeth clean; that's the real task here. If were going to invent something better than a toothbrush or dental floss, I'd need to understand that.

Often the common name for a task refers to the tool that's commonly used to perform it.

But, what if I didn't have floss?

I suspect many years ago, floss didn't exist. I'm not sure about that since the dentist has made me feel guilty for not flossing enough for as long as I can remember. As far as I know floss was on the planet before we were. But – pretend for a moment that there wasn't floss. Flossing likely wouldn't be a task I'd do every day then. But, some dentist, or someone in collusion with a dentist, invented the idea of floss and the daily task to go with it. And, unless it is somehow proven the entire dental profession is for some reason pulling off a huge scam on the flossing public, flossing does help me better meet my goal of "keeping my teeth till I die," so I am mostly happy it was invented.

The tools we invent may motivate the performance of new tasks.

When designing software you and your design team will invent tasks and tools in a similar way. Being aware of users' goals will hopefully motivate you to invent better ways for them to reach their goals. With luck you'll invent tools more enjoyable to use than floss.

I floss because I know I should, and because the tool is available to me. People with goals or discomforts will reach for the tools available in their environment. If I was in my garage and I wanted to pound in a nail, I'd reach for a hammer. If I was camping and I wanted to pound in a tent stake, I might use a handily available rock. But no, I'm a technology professional so I'd go more high-tech and employ my camp shovel. If I had a hammer I could use that as well, but a camp shovel seems to do the trick. The shovel tool wasn't built specifically for hammering in tent stakes, but looking at it and feeling its weight while holding on to its handle, I could easily imagine it could work for a bit of hammering.

People will invent alternative tasks to perform using a tool in their environment that wasn't originally designed for that task.

When we're designing and building software, we may think we know what tasks our users are likely to perform, but it's very common to find out they're using our tools in ways we'd never anticipated. They may have goals we didn't understand, and they certainly devised tasks we hadn't anticipated. The availability of features in our software, tools in their environment, combined with their goals or discomforts motivated this alternative use.

Tasks aren't use cases

A use case is a common approach to documenting a software requirement that originated in the Object Oriented software development communities [Jacobson]. A use case describes the interaction between two or more parties – referred to as actors.

A typical use case might have a simple name that indicates what a person is doing, and a narrative that describes step by step what a user does in a computer system, and what the system does in response. Use cases might also include other humans, or other computer subsystems. For example a use case from a retail sales environment could describe the interaction between a customer, a cashier, a cash register, and the register's on-line credit card approval service – four actors total.

Use cases are an effective mechanism for documenting design decisions but by including tools as actors, they already include some design decisions.

As its name implies, the use case often documents the use of a specific software tool.

Using the retail sales application above, the third and fourth actors mentioned are the cash register and the on-line credit card approval service. These are tools, not tasks. They're good tools that allow the tasks of selling something to a customer go faster and smoother, but they are tools. There are still retail stores that may not use electronic cash registers connected to on-line credit card approval services. For example, I like to buy my wife jewelry made by a distinctive local jewelry maker. I deal with the owner whose name is on the sign. He writes up receipts in his receipt book, and calculates tax with a calculator. If I use a credit card, he might call it in for an authorization using a phone. His tool choices are very different than my grocery store, for instance. But both still have the task of selling me something and collecting my money.

Use a use case to document interactions between a human and some chosen tool – usually a piece of software. But, don't loose sight of the task the user is trying to perform.

The odds are your tool won't completely support the task. I use a software tool to enter expense reports when I travel. It asks that I record expenses for hotel bills and meals separately. I often order room service. So in the context of entering my expenses into the tool I have to take the hotel bill and subtract off the meals I ordered from room service and enter them as separate line items. I use a calculator to do this math. A better designed expense tool would probably support doing the math in the tool. But the real point here is that my task of recording expenses includes steps supported by the computer system, and some that aren't. A use case might document my interaction with the software, but may not document my tasks executed outside the software – tasks using different tools.

Task Goal Level

Concept

Since tasks may contain each other and each task has a short term goal that serves as its success criteria, identifying a task by its goal level allows us to easily identify the size and abstraction level of the task. Identify tasks with goal level.

Tasks have a goal level

While tasks are used to achieve a less tangible personal goal for the human engaged in them, the task itself has a more tangible goal that a human uses to determine if the task was successfully completed. Since bigger tasks are composed of smaller tasks, when looking at a particular task it's nice to know if I'm looking at one of the big ones, a small one, or a really tiny one hardly worth discussing. Identifying the goal level of the task helps me do that.

In Writing Effective Use Cases, Alistair Cockburn introduces the useful concept of goal level. To explain goal level he uses an altitude metaphor with sea level falling in the middle of the model. He refers to sea level as "functional" level. The test for a sea level goal is: would I as someone engaged in this task expect to finish that task in a single sitting, typically without interrupting or setting aside the task to complete later. Sending an email message might be such a task. Buying coffee might be such a task.

Alistair suggests a few tests to determine a function goal level including the coffee break test: "after I get done with this, I can take a coffee break." Ironically, "take a coffee break" may be a task with a sea level goal.

It's important to note that Alistair originally uses goal level as a characteristic of a use case. A use case describes the interaction between two parties – typically a user and a computer. But, when we think about tasks, we don't want to assume that the task takes a computer to perform. We'd like to assume as little as possible. But, even without the second party I'd find in a use case, the concept of goal level still applies handily to user tasks. So, apply it.

Starting from sea level, goal levels moves up and down to neighboring areas of the ocean and atmosphere.

Moving down we'd encounter "fish level goals" or sub-functional goals. These are the smaller tasks that help us complete the functional level task. In our coffee example above we need to get the attention of the server, and order coffee as part of buying coffee. Getting the attention of the server by itself doesn't accomplish much. That's why it's a sub-function.

Moving down even farther we find a very low level goals – clam level as Alistair labels them. To get the attention of the waiter I'll first look around, then throw my arm up and gesture wildly. "Look around" and "gesture wildly" are low clam level. Alistair makes the point, and I'd agree, that thinking about things at this low a level isn't always productive. Since clams are down in the sandy sea bed, thinking at this level is considered down in the sand.

Moving up from sea level we'll encounter the summary level – or kite level. Summary level goals may be achieved over a longer period of time using several function level goals. I met my friend for breakfast to discuss an upcoming client meeting. "Plan an upcoming client meeting" is a task with a summary level goal that may take place over the course of many days and involve many specific tasks including a breakfast discussion.

High summary level is represented with a cloud. This goal level surrounds goals that may take a very long time to achieve. Planning the client meeting was an activity that took place in the context of my job. The cloud level activity here might be to "have a successful career."

Cloud or high summary level: very high level ongoing goals that may never be completely achieved but that I'll use summary level goals to drive towards.

Kite or summary level: long term goals that I'll use various functional level goals to achieve.

Sea or function level: tasks I'd reasonably expect to complete in a single sitting.

Fish or sub-function: smaller tasks that by themselves may not mean much, but stitched together allow me to reach a function level goal.

Clam or low sub-function level: small details that make up a sub function goal.

It's important to note that goal level is a continuum – much like an analog dial for adjusting volume on a stereo. The goal level dial may have five labeled settings but a particular task can easily fall between two of those settings.

Task goal level varies based on its context

It's common for a task in one context to be function level, while in another context that same task is sub-function. For example, buying coffee may be a function level goal if I'm on my way to work and all I'd like is coffee. But, ordering coffee might also be a sub-function inside of ordering breakfast.

Another example might be looking up an email address on your computer. If a friend calls you and asks for another friend's email address, looking up that address may be considered function level. If it's done in the context of sending an email message, it may be considered sub-function.

User Activity

Concept

Activities collect and organize the tasks a person might perform in support of a high or summary level goal. Use activities to describe at a high level the work people are doing in pursuit of their goals.

Activities are composed of tasks

For my purposes here, I won't draw a distinction between tasks at sea level or below. I'll call them all tasks and you'll know that they have sub-goal levels and they can be combined and recombined in a variety of different ways to meet different goals depending on their context.

However, I will use a different name to refer to things people do that are summary level and above. Here I'll use the term *activity*.

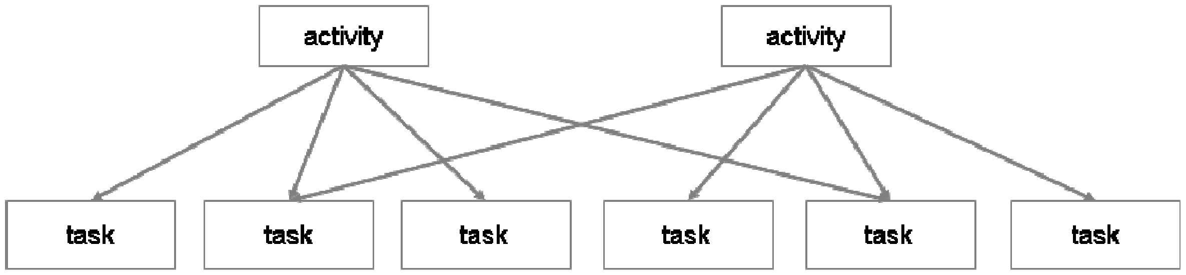
- **An activity refers to a collection of tasks done in support of a summary level goal. An activity may take place once, many times, or may be never ending.**

Examples of activities may be planning meetings, managing email, or handling incoming customer service calls.

It will take a number of tasks collected together to allow me to participate in an activity. In any given activity instance I might use only some of the tasks depending on the specific details of this particular instance of the activity. Each time I engage in this activity it might go a little bit different.

At any given time I may be engaged in many different activities including multiple instances of the same activity. For example I may be in the process of planning several meetings as part of my regular meeting planning activity.

The very same tasks may find themselves supporting multiple activities. For example looking up an email address may support an activity of managing email, and an activity of planning meetings. Both those activities use my contacts and their email addresses.



Plan Incremental Releases as System Spans

When releasing software incremental as in Agile Development approaches it's important that each incremental release be useful to users. These means a useful set of their goals must be met, and a sufficient number of tasks to support those goals must be implemented in the software.

Identifying the tasks to support in an incremental software release can be difficult.

Tasks are combined by users, often in an ad hoc manner, in order to meet their goals. To construct a useful release of software we must support the most critical tasks necessary to meet the user's goals.

Once most critical tasks are supported, my can then seek to support the most frequently used tasks that aren't critical.

The software release must span the full business process supporting not just one user, but all users necessary to consider a full end-to-end business process to be demonstrable.

Business people and project stakeholders need some simple way to understand why some features were included in a release and some weren't. At times some features may seem to have higher value while on closer inspection, less valued featured are actually the features critical to the success of a user in pursuit of a goal.

Use a Span Plan task model to show a full business process and task dependencies in order to construct successful incremental release plans.

A span plan leverages the idea of a system span described in Poppendiek and Poppendiek's Lean Software Development. User tasks arranged in order of dependency across the entire span of the system helps a release planner quickly identify the tasks which are most critical to support in early releases of the software.

The following text is an extract from Jeff Patton's Better Software article "How You Slice it" and gives a detailed description on how a span plan is constructed. Use user tasks as the features called for in step 1.

Step 1: Collect Features

What does your 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 about its implementation. Keeping your focus on the usefulness of the software at this stage helps to ensure that the bits of the software released incrementally will be useful.

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

Suppose I'm building some software for small retailers, I know that their business process goes a bit like this:

- create purchase order for vendor
- receive shipment from vendor
- create tags for received items
- sell items
- return and refund items
- analyze sales

Notice these features start with an action verb. Gosh, they could almost be issued as direct commands to a particular person. To support your model building you'll need the features written on 3x5 cards, or on something 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 pre-cut 3x5 cards or business cards. This way the cards are easy to read and work well with in a card modeling exercise.

Step 2: Annotate Features with A Few Important Details:

To help you model these features, let's note three important details on them.

Who uses this feature?

Note on each feature the *kind* of user that uses it. When describing this feature, you likely envisioned someone using it – who were they? You can identify them with a job title, a role name, a persona, or in any other way most appropriate for your system. Constantine & Lockwood's Software for Use can coach you on roles. Alan Cooper's The Inmates are Running the Asylum gives an overview of personas.

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. With these roles noted, these features might now read:

- create purchase order for vendor: merchandise buyer
- receive shipment from vendor: stock receiver
- create tags for received items: stock receiver
- sell items: customer consultant
- return and refund items: customer consultant
- analyze sales: sales analyst

How often are these features used?

For each feature, note roughly how frequently you believe it will be used. You can use simple notation like High, Medium, or Low. Or, be a little more precise with a continuum like Hourly, Daily, Weekly, Monthly, Quarterly. With frequency noted, the preceding features might read like this:

- create PO for vendor: merchandise buyer, weekly
- receive shipment from vendor: stock receiver, daily
- create tags for received items: stock receiver, daily
- sell items: customer consultant, hourly
- return and refund items: customer consultant, daily
- analyze sales: sales analyst, monthly

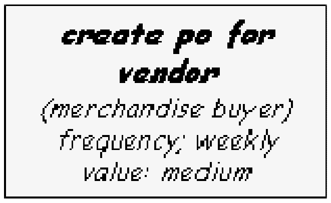
How valuable is this feature?

For each feature, estimate roughly 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. The features may now contain this information:

- create PO for vendor: merchandise buyer, weekly, medium
- receive shipment from vendor: stock receiver, daily, high
- create tags for received items: stock receiver, daily, medium
- sell items: customer consultant, hourly, high
- return and refund items: customer consultant, daily, medium
- analyze sales: sales analyst, monthly, high

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, it's good if the same information appears 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. Because building the model should feel a bit like you're playing a game. A feature card may look something like the example below.


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

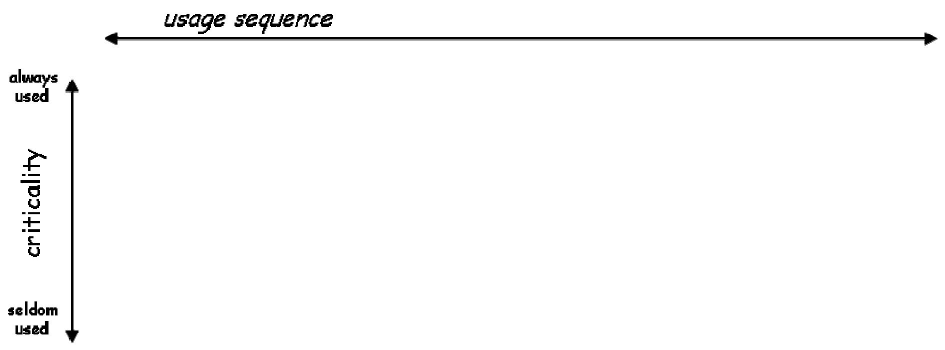
Step 3: Build Your Incremental Release Plan

Set Up Model Axis:

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



Place Features in Your Model

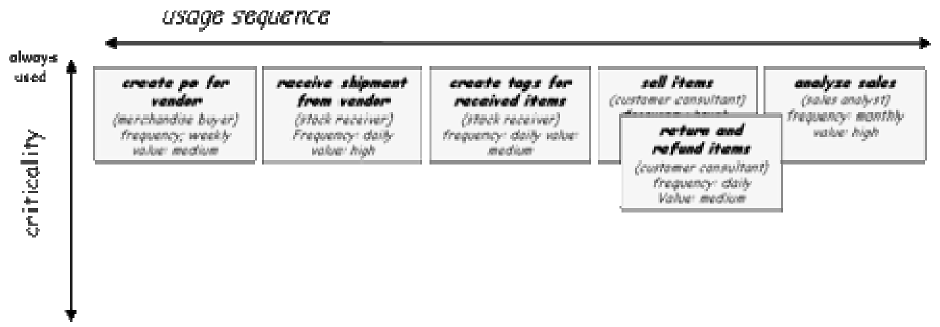
You now need to place features in the model according to usage sequence and criticality. So let's say a little more about these two axis:

On sequence:

Using the features we wrote for out for our retail software above, 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 analyzed after some items are sold. That's what I mean by **usage sequence**.

But, 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. One reason we want this model is to help us tell stories about our software; so arrange them in an order that makes it easy to tell stories.

Distribute the cards among participants. Then, everyone, as orderly as possible, place the cards in your model by usage sequence from left to right, features used early on the left, later on the right. Go ahead and overlap features that might happen at about the same point in time. If you get confused about the position of a feature, try to just 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.

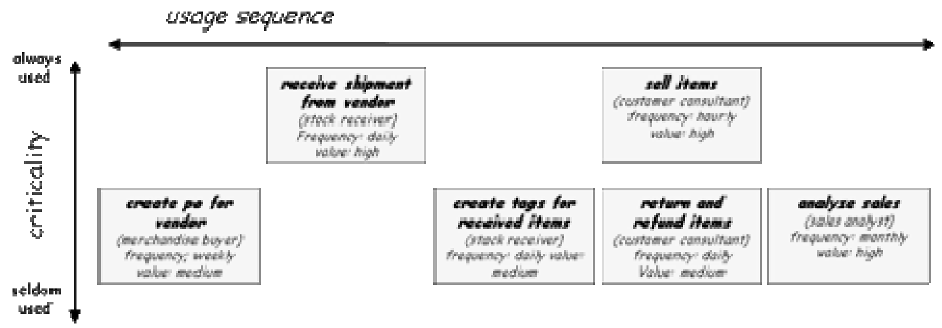

usage sequence
always used
criticality
create po for vendor (merchandise buyer) frequency; weekly value: medium
receive shipment from vendor (stock receiver) Frequency: daily value: high
create tags for received items (stock receiver) frequency: daily value: medium
sell items (customer consultant)
return and refund items (customer consultant) frequency: daily Value: medium
analyse sales (sales analyst) frequency: monthly value: high

If we arrange our features written on cards in sequence, it might look a bit like this.

On criticality:

For each of these features, how critical to our business is it that someone actually uses them? Let’s look at our retail features: When working with the business people who know how their business is run, they inform us that often orders 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*.

All together, 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 done, but not always, 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 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.



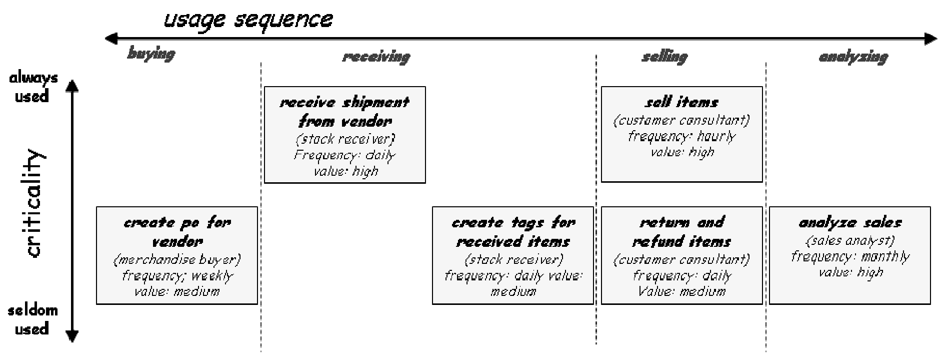
If we adjust our features for criticality, our model might look a bit like this.

Add swim lanes:

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 and some features used by this next role.

As a group discuss where you see breaks or pauses in the business process. Then draw vertical *swim lanes* on this model and label them for each process. If you’re finding it hard to draw swim lanes, 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?

After drawing swim lanes in your model, it might look a bit like this:


usage sequence
buying
receiving
selling
analyzing
always used
criticality
seldom used
receive shipment from vendor (stock receiver) Frequency: daily value: high
sell items (customer consultant) frequency: hourly value: high
create po for vendor (merchandise buyer) frequency; weekly value: medium
create tags for received items (stock receiver) frequency: daily value: medium
return and refund items (customer consultant) frequency: daily Value: medium
analyze sales (sales analyst) frequency: monthly value: high

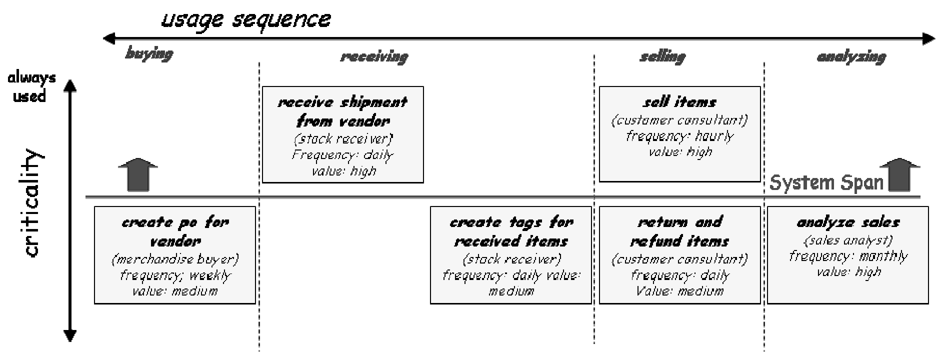
What Is This Model Telling Us?

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

Mark the System Span:

The first system span is the smallest set of features necessary to be minimally useful in a business context. It turns out that the very top row on our model is the first, most minimal system span. This will be true of your model too. For our simple list of features it turns out to be receiving items, and selling items.

Draw a line under the top row of your model to indicate the features that make up this first system span. Your model may look a bit like the example below:


usage sequence
buying
receiving
selling
analyzing
always used
criticality
receive shipment from vendor (stock receiver) Frequency: daily value: high
sell items (customer consultant) frequency: hourly value: high
System Span
create po for vendor (merchandise buyer) frequency; weekly value: medium
create tags for received items (stock receiver) frequency: daily value: medium
return and refund items (customer consultant) frequency: daily Value: medium
analyze sales (sales analyst) frequency: monthly value: high

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, since 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.

The span isn't really enough to release to our customers, why should I worry about building that first?

The 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 use the system. 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 coherently together.

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.

I first encountered the term *span* in Lean Software Development. There the Poppendieks describe building a spanning application as an important first step to building a larger application. If the span can be built quickly, it makes it easier for a larger team or several teams to contribute to the application.

If you're developing commercial software, the span may not be sufficient for a release to the marketplace – unless you don't yet have competitors. If you're writing software for use internally in an organization, the functionality contained within the span may or may not be sufficient for your organization to begin to use the software. The important things to note are that the span should always be your first release, but it need not be the first *public* release of your software.

OK, so building a first span is a good idea, 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. Mike Cohn's User Stories Applied offers lots of 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.

Mark Subsequent Spanning Releases:

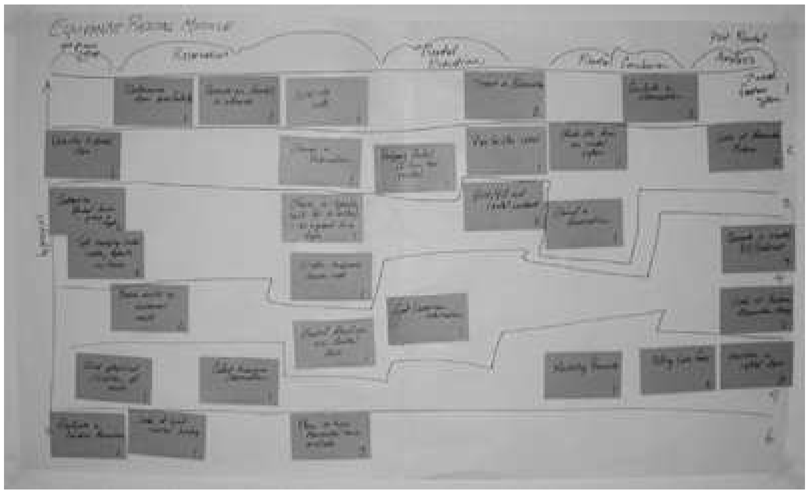
The plan may now be "sliced" horizontally into spanning releases. Well, sort of horizontally. Choose features below the marked first span that group together logically. Choose enough features such that the estimated elapsed development days fit within an appropriate release date. Lines drawn through the plan make it start to look like haphazard layer cake. 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.

At this point in the collaborative activity, the business people responsible for the release should step forward. Let those folks 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 finds its way into an earlier release than another.

Responsible business people continue to slice the 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 swim lanes. 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 at the beginning of this article. Notice in this span plan 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.



What Just Happened Here?

You've just built a span plan. Because you've arranged features in sequential order you understand what features depend on each other. Because you've arranged them by criticality, the important features are emphasized at the top of the plan. Because you've drawn in swim lanes by business process, you know roughly the functionality that supports each major business process in your software. Because you've arranged features this way, you've found the minimal feature span that lets you get your system up and running, end to end as soon as possible. All this information is in one convenient picture. With 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 them.

Additional Reading

- Patton, **How You Slice It** (Better Software, http://www.abstractics.com/papers/HowYouSliceIt.pdf)

Manage Feature Scale

The ideal feature may not be

I hope you're reading this book because you ultimately want to see great software built and in use by your customers. So in spite of all the words in this book dedicated to deciding what to build and evaluating if it was the right thing to build, really the important thing is actually building something.

When we do the work necessary to understand our business goals, our users and their goals, and the work we need to support in the system we're building, we can often clearly see an *ideal* solution.

That ideal solution is compelling. It becomes a bit of a holy grail to quest for. We often set out in out on our quest to build this ideal product by dividing it up into the ideal features that make up this ideal product. We then estimate the cost to design and develop each feature. However, the ideal solution may not be within our budget to construct. It may take more time than we have to meet our release and return on investment goals. It may take more development hours than is economical to spend for a particular feature. Or simply placing so much focus on designing and building one particular ideal feature may distract us from looking at the other features we need. This concentration on a single feature pulls our focus from evaluating the product as a whole and all the features the product will need to be successful. This can be risky.

Designing and building each feature in its ideal form may be unacceptably time consuming, expensive, or risky.

Let's dig a little deeper into this problem.

Of course we want the best product possible. When using incremental development a common approach is to start with a list of features for a product and estimate the time to build each feature based on high level design and assumptions about how that feature might look and behave. Given those development estimates, plus some suitable time for contingency to account for unknown risks, we as a team can then come up with how many of these features we can place in a release of our software. That sounds reasonable, right?

This is where things go bad.

Prioritizing features doesn't always work well

The business paying for the software might like to see the release in a certain amount of time. Let's say 6 months. When we add up the development times plus the contingency for the features it might add up to 12 months. Ouch.

One obvious solution might be to prioritize features and simply not build some of them. Then we'd still get an ideal product with the most important ideal features, right? But, prioritization can be surprisingly difficult. Considering the detail that we have, all the features look necessary. Prioritizing one over the other might be easy in some cases, but hopelessly problematic in others.

Let take an example far away from software. Let's say we're building a car. Now I know cars aren't designed and built like software, but suppose for a minute they were. Suppose in front of you was an empty parking space, and a few months from now you'd like to see car there. Let's start by building a backlog of the features you'd include in your car:

- Engine
- Transmission
- Tires
- Suspension
- Breaks
- Steering wheel
- Drivers seat
- …

I'll stop there, since there are lots of basic features that describe a car, I think you can imagine enough for us to continue. A feature list like this lacks detail. Now if I were to ask you to prioritize a list like this, could you? Is the engine more important than the transmission? Is the steering wheel more important than the breaks? Aren't questions like this just silly?

Often questions about the priority of features in scope seem just as silly, especially for a new product that needs a large set of features constructed to be viable. Yet, it's common to ask business stakeholders to perform this sort of prioritization.

De-scoping features isn't always possible

Assume we were able to prioritize features sufficiently to come up with a release we could live with. The interesting thing about prioritizing features for a product release is that the attention isn't on the top of the list – because these those are the features we know will make it in the release. The emphasis is on the bottom of the list; the cutoff area between features that make it and features that won't. To get an acceptable set of features for a release, lots of heated discussion generally happens around these features. It's common for people making those decisions to push hard on the development estimates for those features.

Assuming software development goes as it often does, toward the end of development time we find that we're a little behind schedule. To meet our schedule we'll need to de-scope a few of these features. Conversations at this time usually don't go well. The features left in scope are all necessary. Decision makers are often left with the problem described when prioritizing car features. They may be asked to de-scope either the steering wheel or the brakes… or more likely if it's near the end of the list, the headlights or the brake lights. These aren't just hard choices to make, those making them know that the resulting product may likely be unacceptable.

This is when business stakeholders usually call for more overtime, offer bonuses, and silently make a vow to consider off-shoring next time – or bringing it back on-shore if this was already the next time.

Features have a hidden dial

Let's go back to our car example. We all know that there's a huge variance in the cost of cars. Yet they usually have the same general features: engines, transmissions, tires, etc. So what makes one car more expensive than another?

To understand that we need to look closely at the specifications of those features – the features of the features. Bigger engines cost more money than smaller engines. Automatic transmissions cost more than standard transmissions. Large, hard, low-profile, rated tires cost more than skinny, soft ones that slide in the rain. So, while a car always needs an engine, transmission, and tires, the choices the manufacturer makes about those things, as well as all the other features of the car, drastically affect the cost of that car.

I'll call that hidden dial in a feature the scale of that feature. For any feature in a car, or in software, we can scale it up, or scale it back. For the features in our car or in our software we can control cost by scaling the feature back.

To control the cost of building all the necessary features in a software release, scale the features back to economical levels.

It's not all about money

While scaling back a feature may allow us to buy it for less, it's not just about the cost of the feature. Let's revisit our story above where we were nearing the end of our release, but unfortunately we're running behind schedule. Let's assume we knew about feature scale, so we scaled the features appropriately before estimated the time and money to build them. We've built most of the features already, except for the last few. Knowing now about the hidden scale dial we now have two choices: de-scoping some of the remaining features, or further reducing the scale of the remaining features.

Since it's the near the end of the release, there's only a few features left. If we reduce the scale of those features to a level that will allow us to deliver on time, we may be back in the same situation we were before – delivering a product that may be unacceptable. Using our car example again, we may be forced to duct tape flashlights to the hood to use as head lights, and cover the windows clear plastic sheeting instead of glass. This looks bad on a car – especially an otherwise high quality car like a Mercedes Benz. Sadly, software often ends up in this position with many well designed and built features and few that look like a clumsy afterthought.

Happily we're designing and building software, not cars. One of the options we have to manage this risk is to scale the features to a very low level early in the development cycle then gradually add scale back to the features. As the design and development of the software release proceeds, the software scale continuously improves. When we near the end of the release development cycle, and development is a little behind we now have most features up to full scale. We need not substantially reduce the scale of some features, but only slightly reduce the scale of a few features. We'd hope that slight scale reduction was not easily perceived by users, or the business paying for the software.

Let's look back at our car and pretend its metal was as malleable as software. If I knew at the end of the release I needed a pretty good car, and I took on this strategy, I might choose to scale the features down and get the car built as quickly as possible. My first car might have a tiny engine, usable brakes, and a steering wheel, but the driver seat was a lawn chair bolted to the floor. I could drive it down the street without a roof or doors, so left those off. I was able to build this sort of car with all the basic features I needed rather quickly, and now I have a lot of time left.

One at a time I begin pulling and replacing some features, and adding in some I could initially do without. The engine gets bigger, the car gets doors and a roof, the lawn-chair initially gets replaced with a simple fabric seat, then towards the end of the release replaced again with a leather seat. At release time I have a pretty good car. There's still more I wished I could have added, but all the features I needed are present, and I haven't really skimped on any of them. This strategy of scaling features up over time seemed to work.

Reduce scale of features aggressively early in a software release development cycle, then build up scale as the release design and development continues.

Given that there's some good reasons to reduce the scale of a feature both when planning a software release, and while designing and constructing the release, let's look closer at how we might thin the scale a of a particular software feature. **Feature Thinning Guidelines** describe four basic areas to look at when making choices to thin the scale of a feature.

Feature Thinning Guidelines

Feature thinning guidelines help make decisions for thinning product features to make them lighter, simpler, and faster to develop. Use feature thinning guidelines to thin proposed features during release planning, and choosing and designing specific work to iteratively develop during release development.

These feature thinning guidelines are based on Gerard Meszaros Storyotype approach for splitting bloated XP stories.

Whether you're using eXtreme Programming and user stories or not, the four categories identified by Gerard offer a simple way to thin features or stories. Looking at a proposed product feature, these guidelines offer a simple way to divide up characteristics of the feature. These characteristics identify perforations in the feature that make it easy to divide up. I've taken liberty with the names of the characteristics so that I could more easily remember and apply them. Hopefully you can to.

Let's continue using our car metaphor. Imagine you're making choices about scale on each particular feature of a car. The characteristics of the feature we'll look at are:

- **Necessity:** what minimal characteristics are necessary for this feature? For our car a minimal engine and transmission are necessary – along with a number of other features.
- **Flexibility:** what would make this feature more useful in more situations? For our car, optional all wheel drive would make it more useful for me to take on camping trips. A hatchback might make it easier for me to load bigger stuff into the back.
- **Safety:** what would make this feature safer for me to use? For our car making the brakes anti-locking would make the car safer.
- **Comfort, Luxury, and Performance:** what would make this feature more desirable to use? I'd really like automatic climate control, the seats to be leather, and a bigger V6 engine.

If we were buying and using a car, we all need the necessities. However, we all may have different opinions on what's more important between flexibility, safety, and comfort, luxury, or performance. We want varying degrees of all those things up to what we can afford. We expect each feature of the car to have at least the necessities and some of all the other categories.

Some characteristics don't easily fit into one category. Is all-wheel drive a safety characteristic or a flexibility characteristic? I guess it depends if it's raining hard outside, or if I choose to drive the car off road to go camping.

Now let's look at a high level software feature and apply some thinning guidelines. Let's choose a feature in software we likely all use. We'll express the need for this feature as a user task – since the feature we build needs to support that user task.

Send an email message

We'll look at how these four guidelines might apply to the feature or features that support this task, and how these guidelines might apply while planning and estimating a release, and while iteratively designing and building the software.

Necessity

To identify what's necessary look at the user of the software and the simplest possible use. Start by looking at the task the user intends to perform. Write a simple **user scenario** or **task case** to understand the steps of usage. Identify only the absolutely necessary steps that allow the task to be considered successful. A feature that supports necessity supports only those necessary steps.

Using our "send an email message" task, success for me is to get a message sent.

The steps I as a user might likely follow would be to:

1. Indicate I'd like to send a message
2. Indicate who I'd like to send the message to
3. Write the message
4. Indicate I'd like the message sent now.

I've really cut back here. Notice I didn't enter a subject line, send it to multiple people, carbon copy anyone, or add any attachments. It's likely I wouldn't choose to buy a product that did only those things. But, if the product couldn't do those basic things, well, that would just be silly.

If we described a "send an email message" feature as having just those things, it would have only the necessities.

For any given software feature, we can thin it to include only its necessities.

When planning and estimating a release: make sure each feature has support for at least the necessities.

When designing and incrementally building your software: initially introduce a new feature into the system using a necessity-only feature or user story.

Flexibility

To identify specific options that make the feature more flexible look back at the user and possible use of the feature. For the task the user is performing, write a simple **user scenario** or **task case**. For each step in that task, ask what your user might alternatively do, or do in addition to the actions performed in that step. In use case writing these might be considered our alternative steps. A feature that supports some degree of flexibility supports some or all of these alternative uses.

Using our "send an email message" task, I might optionally type a subject line – actually I usually do that, but I guess if you press me, it's not really a necessity. I might optionally send the message to many people. I might optionally carbon copy or blind copy others. I might send an attachment, save it to my sent folder, or forward someone else's message.

Adding support for some or all these "might dos" adds flexibility to the feature.

When planning and estimating a release: consider how much flexibility the feature might likely need and estimate development time to include that assumption about flexibility.

When designing and incrementally building: split away flexibility from features to add later as individual flexibility features or user stories.

Safety

To identify specific options that make the feature more safe look again to the user and use. For the task the user is performing write a simple **user scenario** or **task case**. For each step consider things that could go wrong. How might our user enter data incorrectly and cause trouble downstream? What does the business paying for the software want to make sure the user doesn't do? Characteristics that help the user by validating or correcting input, or by applying business rules that restrict or block some actions are considered safety features. A feature that contains this sort of input validation, correction, or business rule applications has some amount of safety built in.

Using our "send an email message" task, if I typed in an improperly formatted email address, sending the message would fail. I might appreciate it if the email address were validated. In some corporations large attachments are considered unacceptable because they bloat email storage, and consume lots of bandwidth. If the system were to stop me from attaching a large file, the business paying for this software might appreciate that. Adding these sorts of characteristics to the "send an email message" feature adds safety. Features in our software often have some number of safety characteristics.

When planning and estimating a release: consider how much safety the feature might likely need and estimate development time to include that assumption about safety.

When designing and incrementally building: split away safety from features to add later as individual safety features or user stories.

Comfort, Performance, and Luxury

To identify specific options that make the feature more pleasant to use look again to the user and use. For the task the user is performing write a simple **user scenario** or **task case**. For each step consider things that make the step easier to accomplish, faster to accomplish, or more fun to accomplish. You may have to base your decisions on what's easier, faster, or more fun based on what the feature already looks like, or what's considered the bare necessity feature. It helps to look at similar or competitive products to identify these areas for improvement.

Using our "send an email message" task, I really hate typing the same email addresses over and over. It would be nice if the system auto-completed addresses I'd used before. I'm a rotten speller, and my grammar aint so good. Could the system check my spelling and grammar in the email message and subject line? Can I insert amusing smiley icons into the message? Adding these sorts of characteristics to the "send an email message" feature adds comfort, performance, and luxury. I'll like this product much better if it has some of these things.

When planning and estimating a release: consider how much comfort, performance, and luxury a feature might need. Estimate development time to include that assumption about comfort, performance, and luxury.

When designing and incrementally building: split away comfort, performance, and luxury from features to add later as individual features or user stories.

Considering Scale When Planning and Estimating a Release

If we follow a progression of understanding our business goals, then our users and their goals, then the work they do to meet their goals, we'll begin to see the "product shaped hole" that allows us to design a tool that helps users meet their goals. The design of that tool starts with understanding usage and the necessary tasks our users need to perform with our prospective tool.

When planning a software release, start with tasks that users will perform.

For each of these tasks we must consider necessity, flexibility, safety, and comfort, performance or luxury. When building a release plan, leverage a business goal model, a user model, and a task model. Look to those business goals, and the focal or most important users and tasks. All tasks will be supported by features that have some measure of each of these four characteristics. It's easy to assume that focal tasks performed by focal users in direct support of a business goal require not only necessities be met, but a high degree of flexibility, safety, and comfort, performance, or luxury as well.

At early planning stages it's not necessary to discuss specific feature decisions or resolve specific feature design, rather use these guidelines to scale estimates accordingly.

Look at the four guidelines to both determine if each task should be supported in the release or not, and to estimate the amount of time each feature might require to design and develop.

Think about necessity

If our software doesn't support the necessary tasks, it simply can't be used. However, it's likely your users have been meeting their goals for a while using inadequate software tools, paper and a pen, or some other approach.

While planning a software release, features to support some tasks may not be necessary if the user can easily use a tool they already have or some other manual process to work around the absence of the feature in your software.

When planning a release, think about which features really are necessary, and what could be worked around.

Think about flexibility

Flexibility, or the usage alternatives we provide in the software, can be tricky. Some variations may be frequently used by some types of users, infrequently by others. You may develop suspicions about the amount of flexibility needed in your features by the sophistication level of your user constituencies: sophisticated users generally take advantage of more options when performing tasks.

Complex business processes generally allow for more variation. Building features to support those sorts of business processes will likely require more flexibility. However, if the business paying for the development of the software is also the business using the software, they'd do well to question complex business processes. Could those processes be made simpler?

To estimate the level of flexibility needed, look to the sophistication of the users using the software and to the complexity of the work being performed. Expert users appreciate more flexibility. Complex business processes require more flexibility.

When planning a release on a feature by feature basis, consider how much flexibility might be necessary based on the sophistication level of the users and the complexity of the work.

Think about safety

When considering safety think first about the user constituencies using the software. Novice users may likely follow the rules about inputting or manipulating information, but may have trouble learning them or be more error prone. They'll need a fair bit of friendly validation. Users working very fast may also make mistakes – but they'll need input validation that doesn't stop them from moving fast.

Consider the rules of the business paying for the software. What do they want to ensure the user does or doesn't do? What would happen if they enforced less?

To estimate the level of safety needed consider the expertise of the users of the system and the number of rules the business would like to see enforced. Novice users may need more safety features. Complex business processes may require more safety rules.

When planning a release, on a feature by feature basis, consider how much safety might be necessary. Look at the sophistication level of your use audience. Look at the complexity of business rules the business would like to be enforced. Ask "what would happen if we didn't validate user input here?" and "What would happen if we didn't enforce these business rules?" You may need less safety than you think.

Think about comfort, performance, and luxury

Start by thinking about the users of the system. What other tool choices do they have? If this is a commercial application, what comfort, performance, and luxury features to competitive applications have that they consider valuable? What is the sophistication level of the user? Advanced users can better utilize feature characteristics that allow for faster performance. Frequent users appreciate feature characteristics that make frequent use more pleasant. Novice users appreciate feature characteristics that make the software easier to learn.

Next consider the business paying for the software. If this is a commercial product, it may be necessary to have features that rival competitors for the software to be viable in the market place – whether your users really need them or not. If the organization earns more money if users are more effective at their work, consider performance and luxury as a way to increase return on investment as a result of increased efficiency.

For comfort, performance, and luxury features, always consider the person buying the software separately from the person using the software. What features or feature characteristics attract the attention and sway the opinion of a buyer? For large enterprise class software, the buyer is often not the user. It may pay big dividends to your company to enhance features attractive to buyers, although they may never touch the keyboard.

To estimate the amount of comfort, performance, and luxury necessary consider the affects of these features on the sales, adoption, and use of the software. Look more closely at the financial drivers when estimating. Opportunities for increasing return on investment drive additions to comfort, performance, and luxury.

When planning a release, tasks by task, consider where to best place comfort, performance, and luxury to make the largest impact on the software's return on investment.

Estimates at the release level become budgets

The estimates given against prospective features to support tasks will serve as development time budgets during iterative design and development. For now, use these estimates to size and plan releases that will be successful in the eyes of their target users and the business paying for the development. As you begin to make specific design decisions for the features to support users' tasks, let the budget set limits for the amount of flexibility, safety, and comfort, performance, or luxury you add.

Thinning and Building Up Features During Iterative Design and Development

When we set out to build a software release, we make decisions to try and come up with a set of features which will be both feasible to build in the time available, and make the product as valuable as possible both economically to the business paying for the software and functionally to the user constituencies who will use the software.

When we choose those features and make our plans, we do so without detailed knowledge of exactly how each feature may ultimately look and behave in the software and exactly how much time each feature may take to develop.

As we design and build more features we may discover better ways to build features, opportunities for other high value features, or omissions of features we subsequently understand may be critical.

While we're busy designing and building a software release, the world around us isn't standing still. Competitors may release new software or features we must react to. We may come to understand more about our users or our business that causes us to question our feature choices. Other demands on our available design and development time may reduce the time budget we have to complete a release.

Uncertainty about specific feature design and development time along with high likelihood of change requires active management and adaptation in the decisions we make about specific feature characteristics.

If we try to battle uncertainty by resolving more details about feature design before planning the release, we risk delaying the beginning of that release. While we may be more confident in our development estimates, we're certainly not ensured of their accuracy. In fact they're still likely wrong. We may be tempted to reduce risk by padding development times to a more comfortable level, which increases the time to delivery, the cost of the product, and may substantially reduce the return on investment for the product.

If we proceed with feature design and development naively, we're likely to find ourselves with a subset of our features built as we'd anticipated, or some other fraction of features as we're faced to make the choice to de-scope or scale down. The result is often an incoherent product where necessary features are either missing or designed and implemented poorly relative to other features in the product.

An effective strategy for designing and developing during release is implement all intended features scaled down to bare necessities, then gradually adding flexibility, safety, comfort, performance, and luxury.

This design and development strategy has the effect of gradually bringing the software into focus.

Software design and development is an art

When I was very younger I expected I'd be a graphic artist. I spent a lot of time drawing. One of the mistakes I often made was to picture in my head what I wanted to draw and then begin to draw it. I'd render my subject in fairly precise detail. If I were drawing a dinosaur – which I often did when I was a kid, I might start with the head. A T-Rex head is big and mouth full of sharp teeth which I'd take quite a bit of time drawing. As I moved on, I'd eventually get to the neck, body, arms, legs, and tail. This is when I'd figured out I'd got the proportions all wrong. The head was generally way too big for the body. The shape of the whole dinosaur looked as though I was looking at it through a funhouse mirror. Then my mom would call me to dinner and I didn't have time to fix or finish it. So in addition to the odd shape, there were parts, like the head, in extreme detail, and other parts un-drawn or with nearly no detail. This wasn't the scary dinosaur I was looking for.

When developing software iteratively, this is often one of the outcomes. Slightly misshaped software with high quality of detail in some places, and lots of rough spots elsewhere.

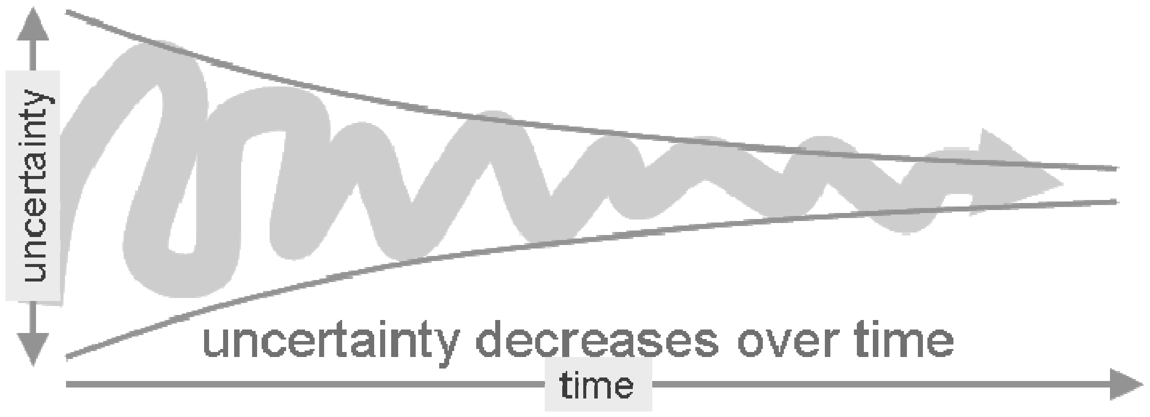
Let's go back to our budding artist. As I continued to draw, and got a little help, I learned that artists often sketched out the basic shape of the thing they were drawing. They resolved the positioning and proportions of elements on the page first. Painters often created an under-painting or a preliminary painting that let them see the basic form, colors, and contrast in the painting. Then the artist might proceed to work on different parts of the painting, moving from one part to another, spending time where it seemed sensible, but gradually working on the whole drawing as a single unit. A graphic artist with a deadline might look to the important, or focal, parts of the work and put more time there. She knows this was where the viewer's eyes would spend more time. She could pace herself to create the best work possible given the allotted time.

Software design and development works well when it follows the same strategy.

Mapping uncertainty to the software release

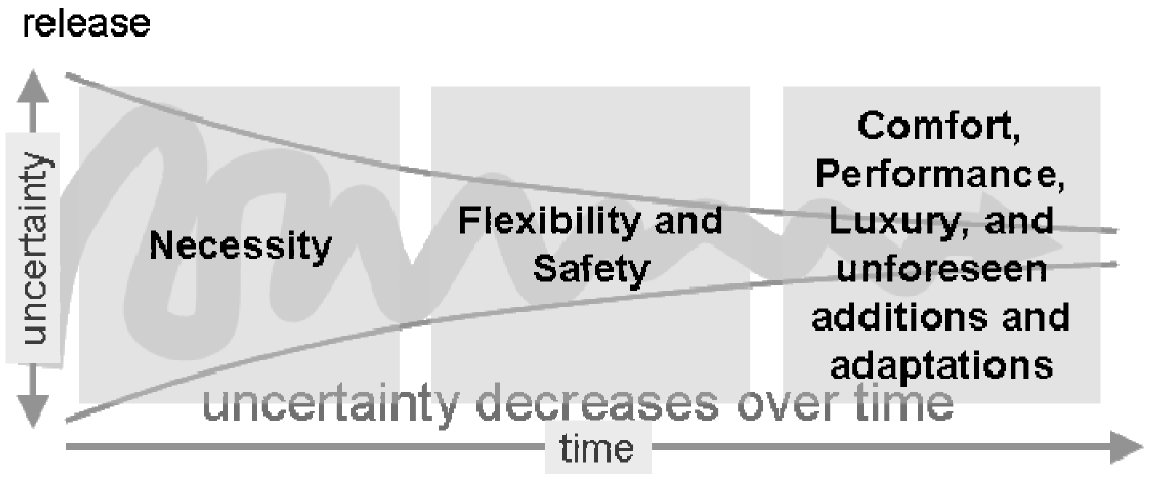
There's a fairly obvious law that we often forget to accommodate. It's difficult to predict outcomes and events in the distant future and easier to predict them in the near future.

Barry Boehm first described the Cone of Uncertainty to illustrate how uncertainty about development estimates decreases over time. Steve McConnell later elaborated on this to say that the same applies for requirements.



The iterative development style of Agile development adapts to this uncertainty by not focusing early on accurate estimates, since they likely won't be, and not focusing too hard on accurate requirements, since they likely won't be.

If we acknowledge and accommodate what looks like an obvious truth and map our feature thinning guidelines to this we're able to come up with a useful strategy for thinning and building back-up features during the time span of a single incremental release.


release
uncertainty
Necessity
Flexibility and Safety
Comfort, Performance, Luxury, and unforeseen additions and adaptations
uncertainty decreases over time
time

The shape of iterative release design and development

To effectively manage the design and development of features throughout a typical release cycle we'll need to aggressively manage the scale of each feature. In the Agile development approach of eXtreme Programming, we might consider the feature or features that support each user task a user story. Built in an ideal form, or in a scaled down form we'd intended for our release, they might be very big stories. We'll use our thinning guidelines to split these big features or big stories into lots of smaller stories. These are the stories we'll focus on during the each iteration of development.

This ideal release design and development strategy breaks the release period in thirds – a three trimester gestation period. Please excuse the bad metaphor.

Necessities: Starting the release development

Focus first on carving away all the necessities in each feature of the software. Use the first part of the release cycle to complete the design and development of all these necessities. If this is a new product and you've used a **span plan** to help plan this release, you'll have implemented the first system span. If this is a subsequent release, you'll have implemented some range of features that span the entire release's functionality.

Alistair Cockburn describes this as a *walking skeleton*; although this term is often used to indicate the design and implementation of basic architectural necessities. This walking skeleton could be considered a functional walking skeleton in addition to an architectural one.

There's no need to split the features further at this point. When managing a feature backlog, I may place a necessity feature into the first iteration, and then leave one remaining feature for "all the rest" of the intended feature characteristics. You'll split these later. There's no need to predict how they need to split now.

Flexibility & Safety: Filling in

By a third of the way in to your release cycle, the end of the first trimester, you now have the necessities completed for your release. You can see the general shape of your software. You've implemented most of your basic domain objects. You've implemented most of the screens of your software and can now see the navigation structure clearly. With luck the basic architecture of your software is complete. This is a good time to begin validating basic interaction design using usability testing, and system performance and scalability using functional testing and load testing. You've mitigated a lot of risk by doing this. Pat yourself on the back.

Now go back through your features and carve out flexibility and safety additions to the features you've implemented. Keep the pieces you carve out reasonably small. Don't do your carving all at once. You might carve an iteration or two ahead, but don't plan the entire rest of your release.

The end game

By two-thirds of the way through, the second trimester, you're seeing a pretty solid product. It may not be quite as sexy as you'd like, but it works well. You've been able to validate the scalability and performance, and adapted to some unforeseen problems there. You've been performing simple end-to-end usability testing and stumbled across some features you'd overlooked. You also identified some changes to navigation that will make things easier to use. Along the way you've identified areas in the software where you can significantly improve the users' experience. If the software really had to release now, it could, but it wouldn't be your best work.

If you have new features to implement, immediately carve off, design, and build necessities. Then move through to flexibility and safety additions for those features.

Go back through all other features and carve off additions that will add comfort, performance, and luxury. Now more than ever your business goals and user models will give you guidance on where best to spend the remainder of your time. Look for additions that benefit focal users performing focal tasks that directly support business goals.

As the release date nears, shift designers and developers into validation roles testing and retesting the software for functional and usability errors.

Always releasable?

There's a goal in eXtreme Programming that the software we're designing and building be always releasable. This goal is adopted by many projects working in an Agile manner. You'll notice that the "trimester" strategy may not leave the software always releasable. During the release's first and second trimester the software wouldn't normally be considered viable. Unlike childbirth, these three cycles don't need to take 9 months. I've typically practiced this pattern using a 45 to 90 day cycle. You can scale your cycle depending on the complexity of your product. Let your release planning help guide the decisions about what a viable product might be.

Thinning is a risk management strategy

No matter how you look at it, it often takes longer to split up a feature into little parts and develop it a piece at a time. It often takes major rearranging or rewriting of existing features to accommodate new flexibility and safety characteristics. Sometimes a comfort, luxury, or performance feature may result in complete redesign of both user interface and underlying code.

Thinning and building up allows deferring of design decisions when uncertainty is at its lowest till later in the release cycle. It affords the earliest possible validation of the functional scope of the release. It affords the earliest possible validation of underlying architecture and domain objects. It affords early testing of usability, performance, and scalability. It preserves time near the end of the release cycle to react and adapt to what you learned from building the software, and what might have changed in the world around you while you were building.

To better manage risk, use a thinning and building up strategy to coordinate the ongoing design and development of features during a release development cycle.