# Bringing User-Centered Design Practices into Agile Development Projects

Jeff Patton
**Thought**Works
jpatton@thoughtworks.com
jpatton@acm.org

# Today Day at a Glance

## Part 1: The Agile Project Context

In part one I'll lay a foundation for understand Agile Software Development and its place in software development approaches generally. You'll learn that although Agile Development doesn't refer to a single specific methodology that most Agile approaches follow a fairly consistent process flow.

You'll begin to learn a bit about the communication and collaborative work styles typical of Agile Development by modeling business goals – a concern near and dear to the heart of most Agile approaches and practitioners.

## Part 2: Project Inception & Planning

In part two I'll describe Garrett's simple model of User Centered Design that's helpful in explaining the work of designers to Agile practitioners. Using that model and correlating it with our model of Agile Development we'll get some suggestions about how best to handle project inception and planning.

We'll build a simple user model to understand our users. We'll build a simple task model to help us visualize our users' workflow then use that model to help us plan multiple incremental releases of our software.

## Part 3: Building & Validation

In part three we'll dive headfirst into a simulated Agile Development cycle. You'll need to understand just a couple quick concepts before you can plan your product releases. Then, given that plan you'll use paper prototyping to build the first release of the software you've planned.

## Part 4: Adapting & Thriving

Because no one ever gets things right the first time, and Agile development places big emphasis on adaptation, we'll make adjustments to the software we built in our first release. We'll also add additional features to create a better second release.

We'll close the day by talking about a few organizations that have successfully adopted Agile Development and strong User Centered Design practices. We'll discuss common attitudes and approaches they share that have allowed them to be successful.

# Handouts Table of 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

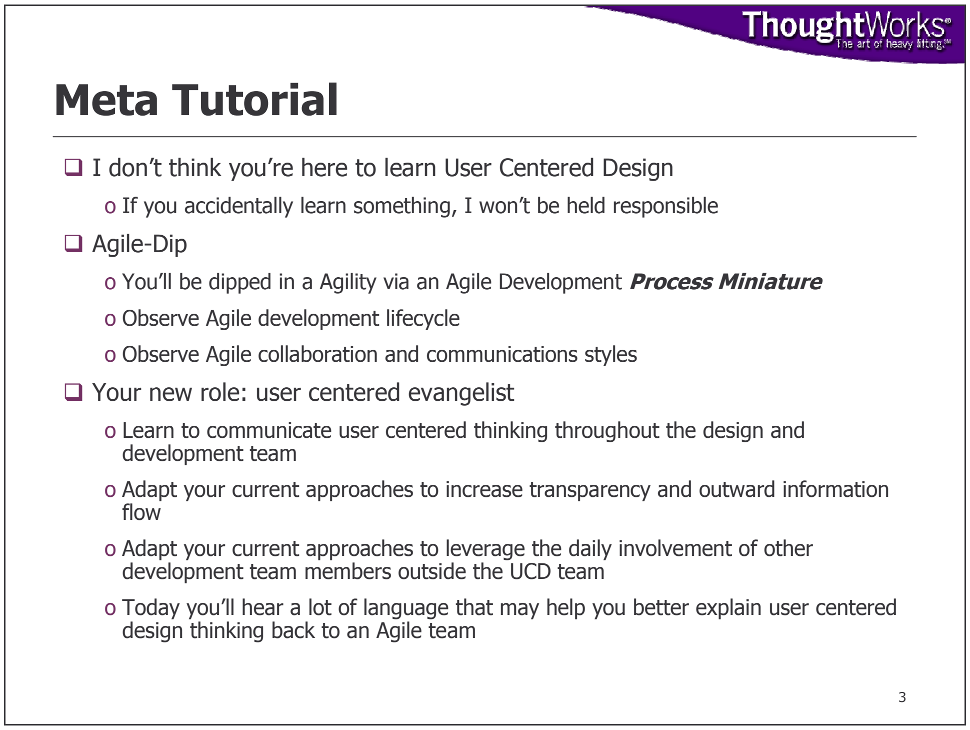
**Discussion group:** tech.groups.yahoo.com/group/agile-usability

## Part 1: The Agile Project Context

In part one I'll lay a foundation for understand Agile Software Development and its place in software development approaches generally. You'll learn that although Agile Development doesn't refer to a single specific methodology that most Agile approaches follow a fairly consistent process flow.

You'll begin to learn a bit about the communication and collaborative work styles typical of Agile Development by modeling business goals – a concern near and dear to the heart of most Agile approaches and practitioners.

Notes



ThoughtWorks® The art of heavy lifting.℠

### Meta Tutorial

- I don't think you're here to learn User Centered Design
  - If you accidentally learn something, I won't be held responsible
- Agile-Dip
  - You'll be dipped in a Agility via an Agile Development ***Process Miniature***
  - Observe Agile development lifecycle
  - Observe Agile collaboration and communications styles
- Your new role: user centered evangelist
  - Learn to communicate user centered thinking throughout the design and development team
  - Adapt your current approaches to increase transparency and outward information flow
  - Adapt your current approaches to leverage the daily involvement of other development team members outside the UCD team
  - Today you'll hear a lot of language that may help you better explain user centered design thinking back to an Agile team

3

## You're engaged in a *Process Miniature*

You'll learn about this approach by participating in a "**process miniature**." You'll rapidly discover requirements for, plan the incremental release of, design, and build software in an Agile project. Although user centered design approaches are similar conceptually, many of the specific UCD techniques we'll be using are adapted from Constantine & Lockwood's Usage-Centered Design.

*"New processes are unfamiliar and perplexing. The longer the process, the longer before new team members understand how the various parts of the process fit with each other. You can speed this understanding by shrinking the time taken by the process. This is the Process Miniature."*

*"Run the entire process in a very short time period (a few minutes to a few days)."*

*-- Alistair Cockburn*

You'll learn collaborative user centered design and project planning by doing it – at least a little bit of it. While participating, think about how you might attempt some of these approaches in your organization.

- What sorts of challenges would you encounter?
- What sorts of problems could it solve?
- Also take note of the card sorting and modeling activities.
- Are there other tasks where card sorting and modeling might be useful?

## A note about these notes:

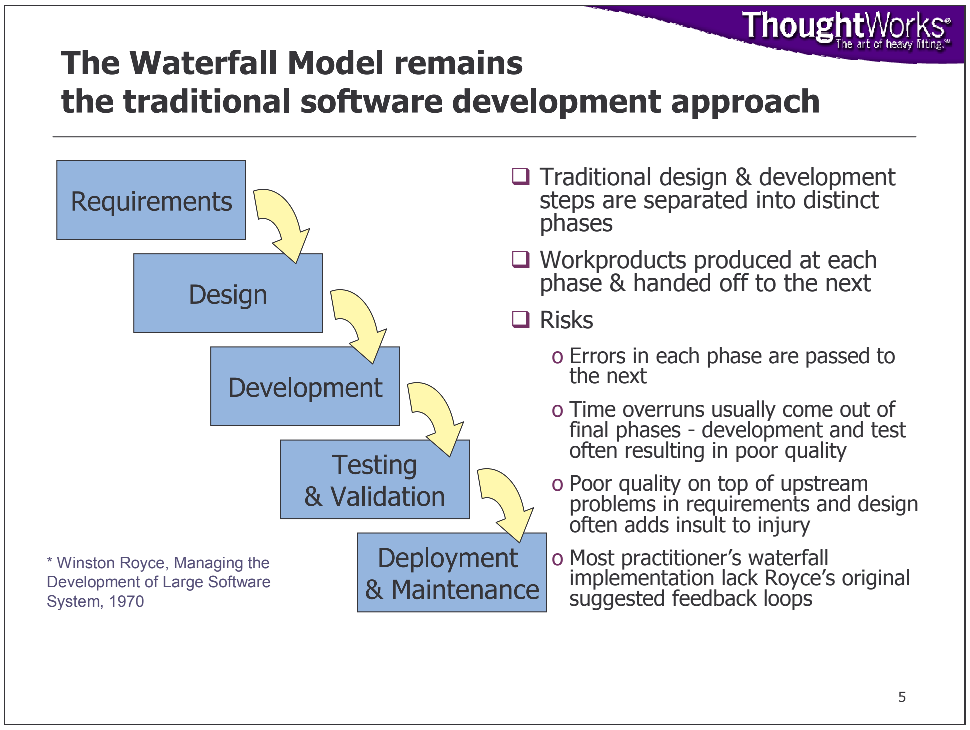
While the notes do follow along with the slides and the subject matter we'll be discussing in this tutorial, they also include lengthy excerpts from articles and other work not yet published. Please don't stop to read them during the tutorial. Do use these notes as reference later on should use choose to use some of these concepts or techniques in your day to day work.

## Additional Reading

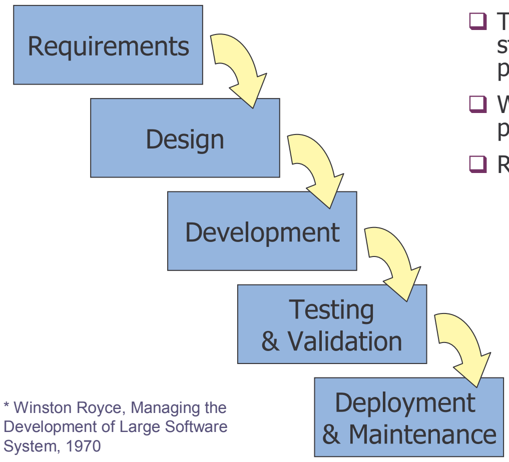
Constantine & Lockwood, **Software for Use**, 1999, Addison-Wesley

Cockburn, **Process Miniature**, http://c2.com/cgi/wiki?ProcessMiniature

Notes



## The Waterfall Model remains the traditional software development approach



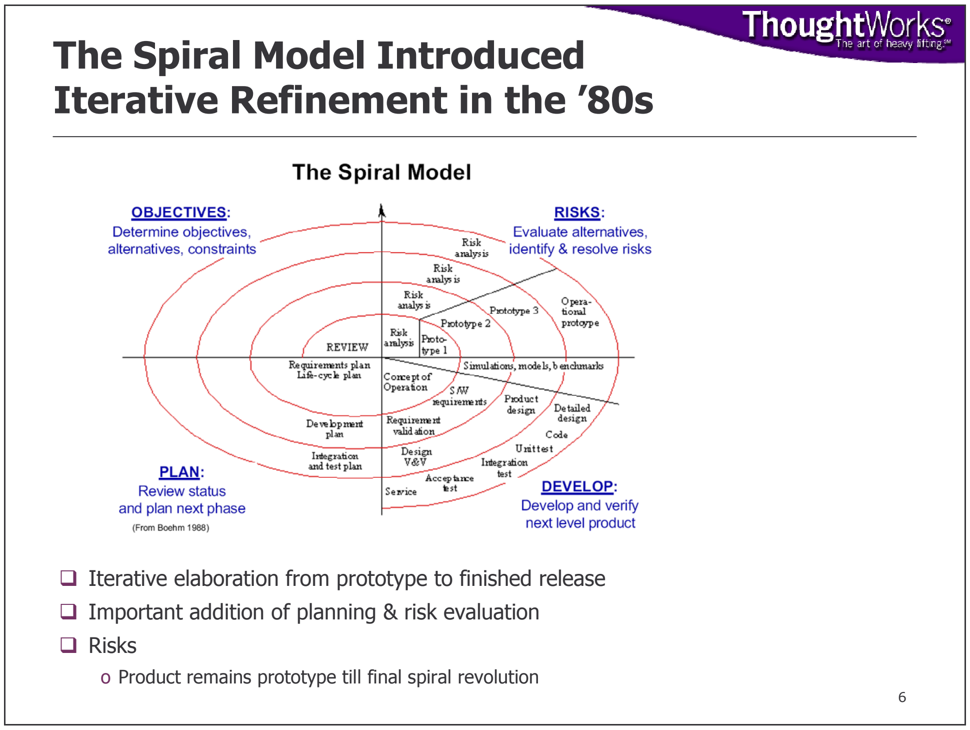
* Winston Royce, Managing the Development of Large Software System, 1970

- ❑ Traditional design & development steps are separated into distinct phases
- ❑ Workproducts produced at each phase & handed off to the next
- ❑ Risks
  - o Errors in each phase are passed to the next
  - o Time overruns usually come out of final phases - development and test often resulting in poor quality
  - o Poor quality on top of upstream problems in requirements and design often adds insult to injury
  - o Most practitioner's waterfall implementation lack Royce's original suggested feedback loops

5

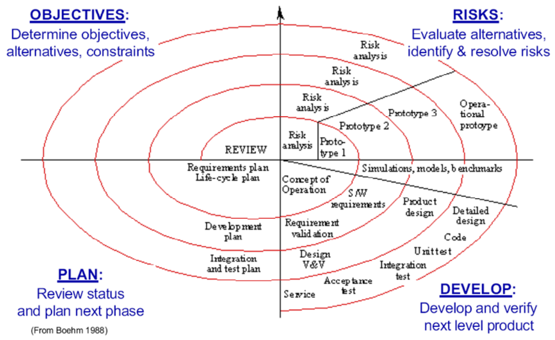
**Additional Reading:**

- Winston Royce, **Managing the Development of Large Software Systems** http://www.cs.umd.edu/class/spring2003/cmsc838p/Process/waterfall.pdf
- WikiPedia, **Waterfall Model**, http://en.wikipedia.org/wiki/Waterfall_model



## The Spiral Model Introduced Iterative Refinement in the '80s

The Spiral Model



(From Boehm 1988)

- ❑ Iterative elaboration from prototype to finished release
- ❑ Important addition of planning & risk evaluation
- ❑ Risks
  - o Product remains prototype till final spiral revolution

6

**Additional Reading:**

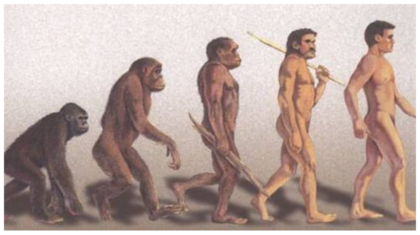
- Barry Boehm, **A Sprial Model of Software Development and Enhancement,** http://www.sce.carleton.ca/faculty/ajila/4106-5006/Spiral%20Model%20Boehm.pdf
- WikiPedia, **Spiral Model**, http://en.wikipedia.org/wiki/Spiral_model

Notes

ThoughtWorks® The art of heavy lifting.℠

## The Origins of Agile Development Spring From Early Discussions on Adaptive Incremental Development

- ❑ The Psychology of Computer Programming – Gerald Weinberg, 1971
- ❑ The Mythical Man Month, Fred Brooks, 1986
- ❑ Scrum, Ken Schwaber, Mike Beedle, 1986
- ❑ PeopleWare, DeMarco & Lister, 1987
- ❑ Borland's Software Craftsmanship, 1994
- ❑ Dynamic Systems Development Methodology, 1994
- ❑ Crystal Methodologies, Alistair Cockburn, 1997
- ❑ Feature Driven Development, Jeff DeLuca, 1998
- ❑ Adaptive Software Development, Jim Highsmith, 2000
- ❑ Extreme Programming, Kent Beck, 2000 (origins in 1996)



7

ThoughtWorks® The art of heavy lifting.℠

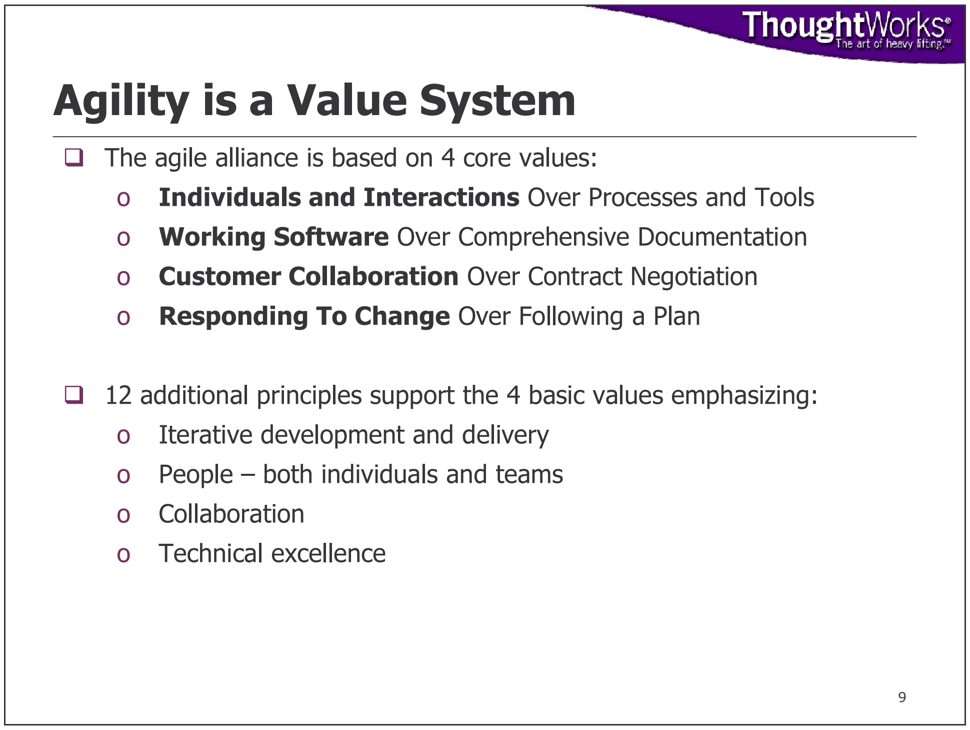
## Coining The Agile Software Development Label

- ❑ XP's success acts as a catalyst
- ❑ Meeting of 17 at Snowbird, Utah, 2001
- ❑ All participants disagree on specifics
- ❑ All agree they have something in common
- ❑ 4 principles of the Agile Manifesto

❖ www.agilealliance.org

Agile Alliance

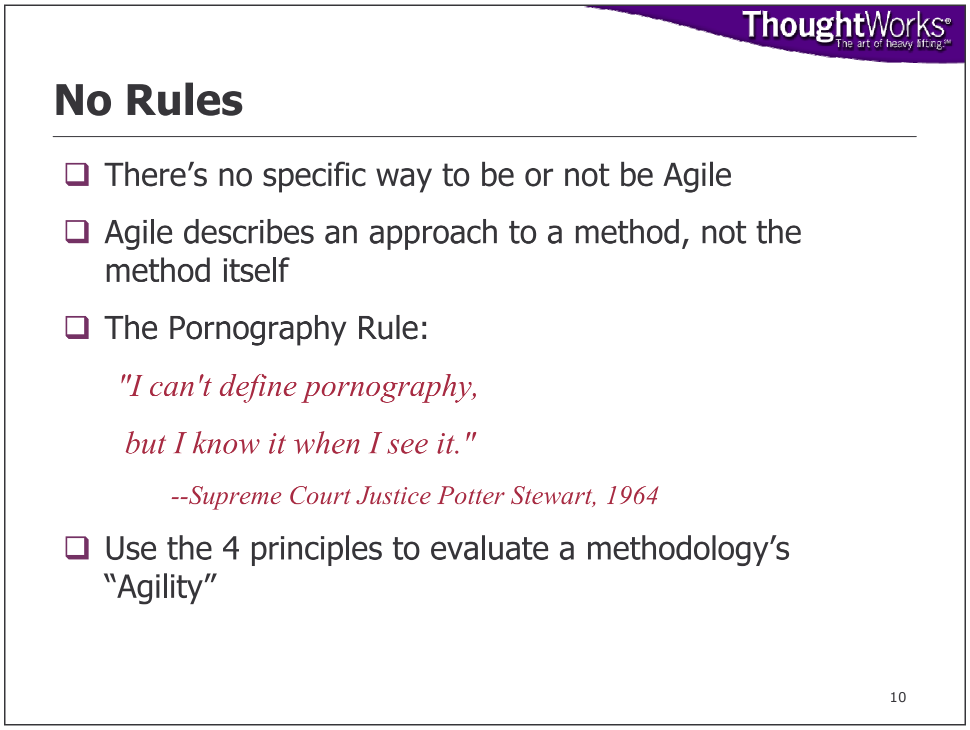
8



## Agility is a Value System

- The agile alliance is based on 4 core values:
  - **Individuals and Interactions** Over Processes and Tools
  - **Working Software** Over Comprehensive Documentation
  - **Customer Collaboration** Over Contract Negotiation
  - **Responding To Change** Over Following a Plan
- 12 additional principles support the 4 basic values emphasizing:
  - Iterative development and delivery
  - People – both individuals and teams
  - Collaboration
  - Technical excellence

9



## No Rules

- There's no specific way to be or not be Agile
- Agile describes an approach to a method, not the method itself
- The Pornography Rule:

  *"I can't define pornography,*

  *but I know it when I see it."*

  *--Supreme Court Justice Potter Stewart, 1964*
- Use the 4 principles to evaluate a methodology's "Agility"

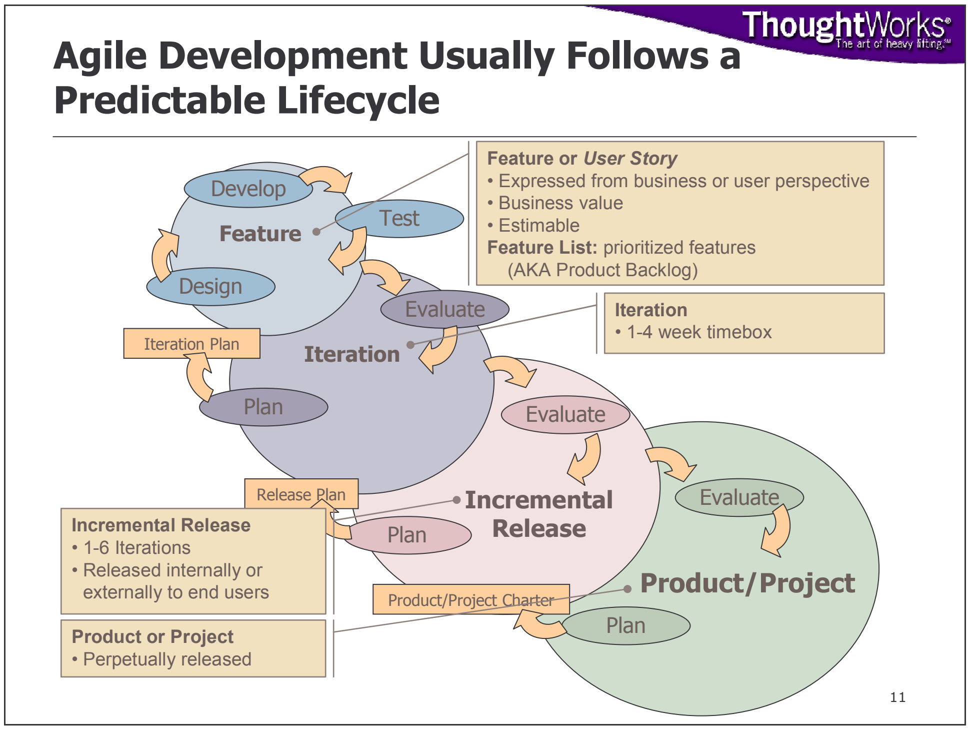
10

Notes

**Additional Reading:**

- Martin Fowler, **The New Methodology**, http://www.martinfowler.com/articles/newMethodology.html

Notes



## Agile Development Usually Follows a Predictable Lifecycle

**Feature or *User Story***
- Expressed from business or user perspective
- Business value
- Estimable

**Feature List:** prioritized features (AKA Product Backlog)

**Iteration**
- 1-4 week timebox

**Incremental Release**
- 1-6 Iterations
- Released internally or externally to end users

**Product or Project**
- Perpetually released

## 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.

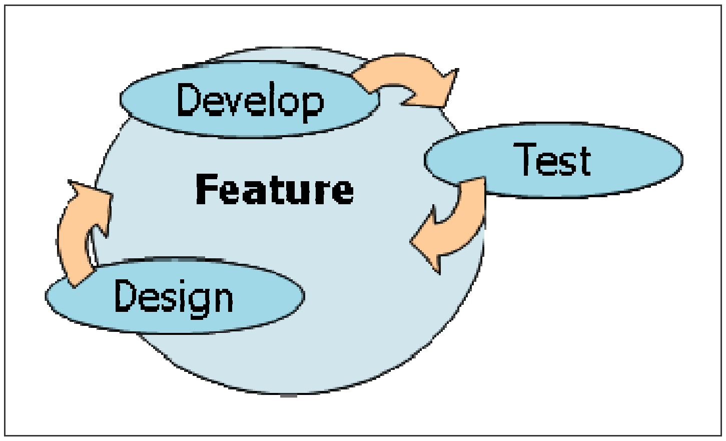


**Figure 1.1: feature development**

## 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.

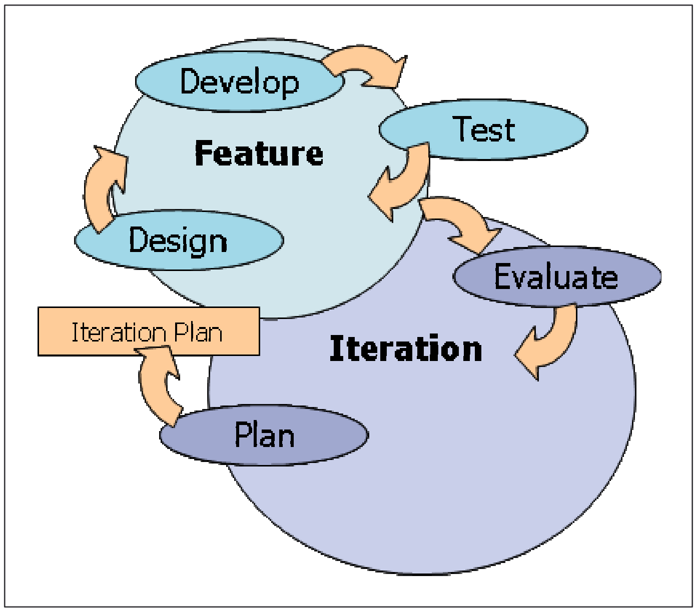


Figure 1.2: Iterative development of features

## 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.

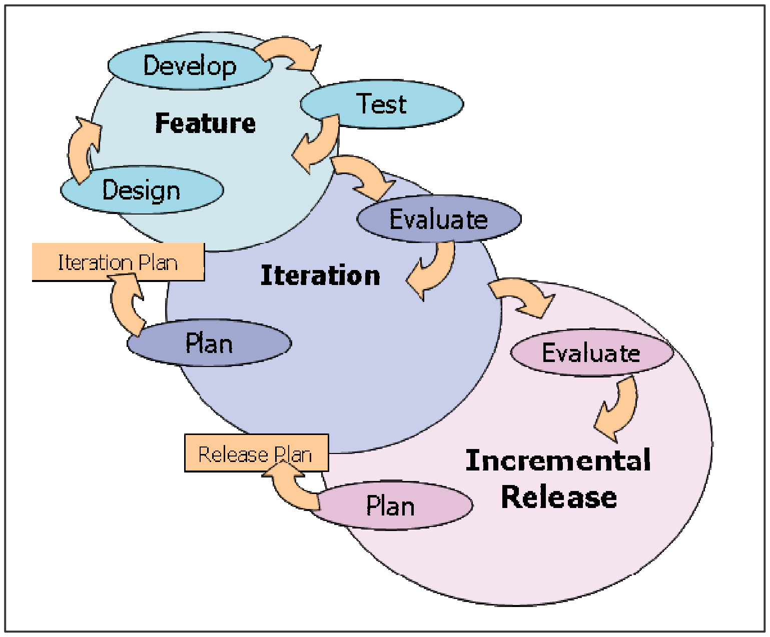


Figure 1.3: Incrementally released, iteratively developed features

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:

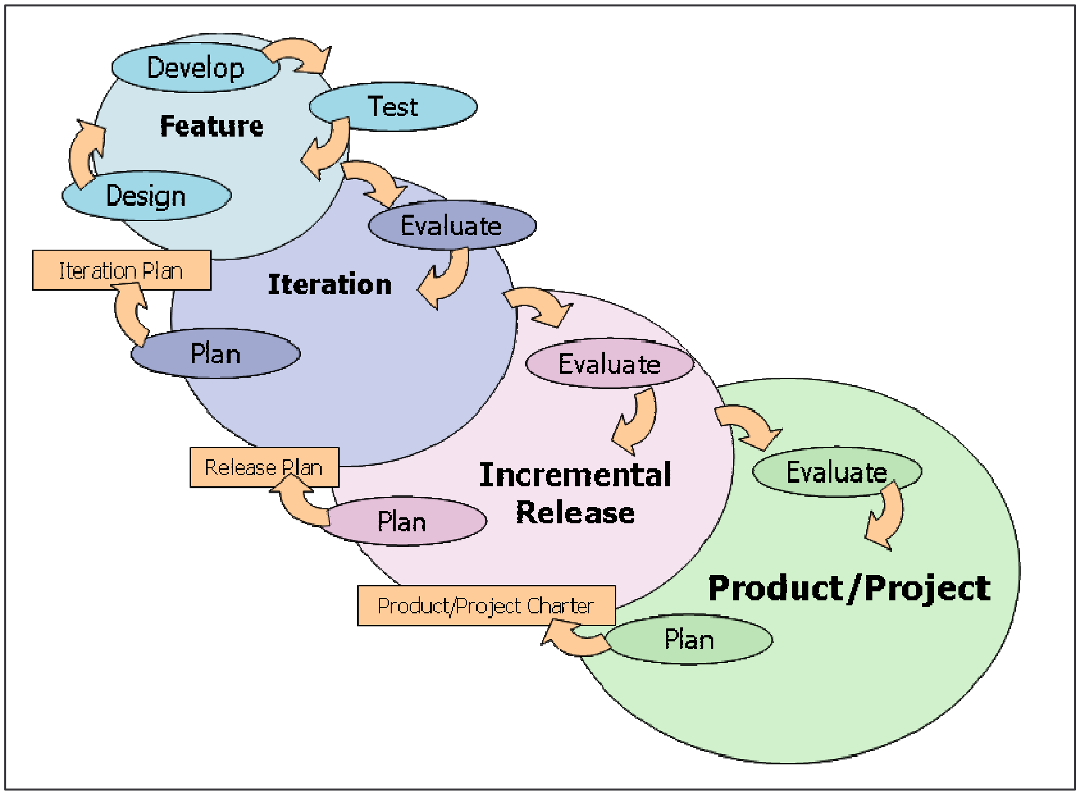


Figure 1.4: Chartering an incrementally released product or project

## 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.

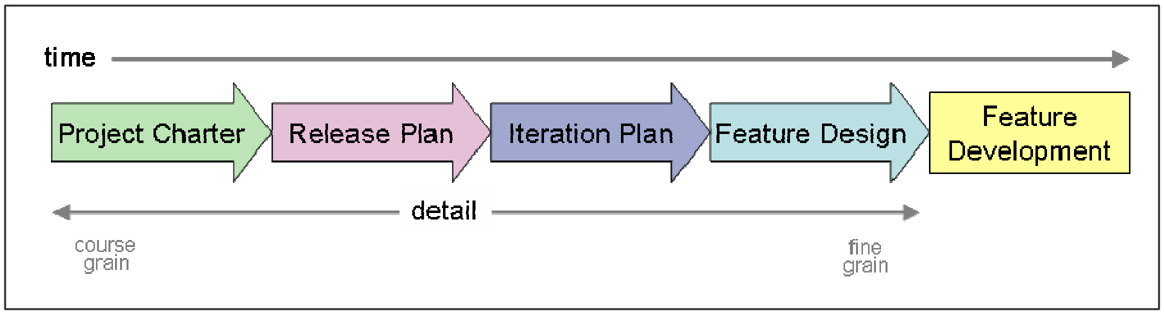


Figure 1.5: Granularity of planning over time.

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.

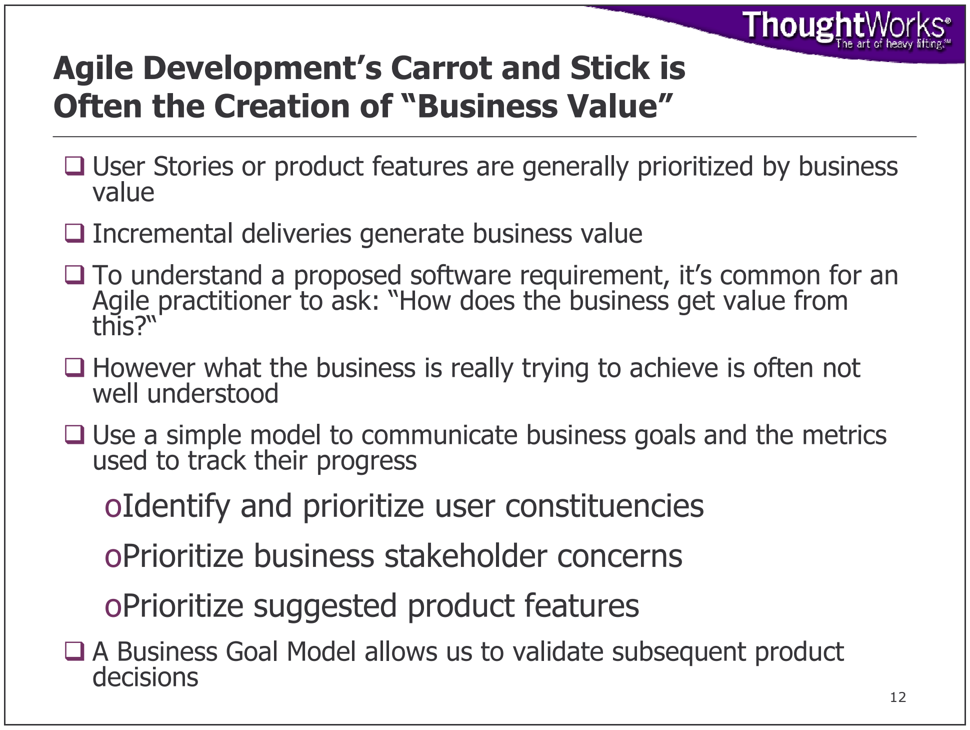
Cycles may be at a very detailed and tangible level – like the design, building, and testing of a single feature. Cycles may be at a higher abstract level like the design and planning of a product release, its construction, and subsequent evaluation of the finished product.

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)

Notes

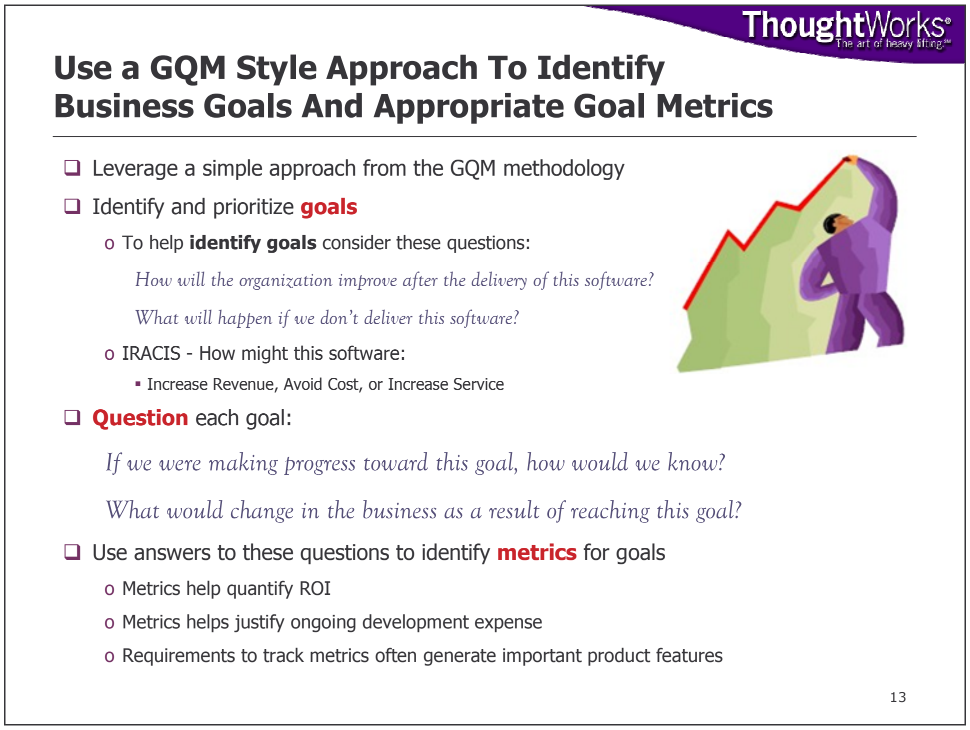


ThoughtWorks® The art of heavy lifting.℠

## Agile Development's Carrot and Stick is Often the Creation of "Business Value"

- ❑ User Stories or product features are generally prioritized by business value
- ❑ Incremental deliveries generate business value
- ❑ To understand a proposed software requirement, it's common for an Agile practitioner to ask: "How does the business get value from this?"
- ❑ However what the business is really trying to achieve is often not well understood
- ❑ Use a simple model to communicate business goals and the metrics used to track their progress
  - o Identify and prioritize user constituencies
  - o Prioritize business stakeholder concerns
  - o Prioritize suggested product features
- ❑ A Business Goal Model allows us to validate subsequent product decisions

12



ThoughtWorks® The art of heavy lifting.℠

## Use a GQM Style Approach To Identify Business Goals And Appropriate Goal Metrics

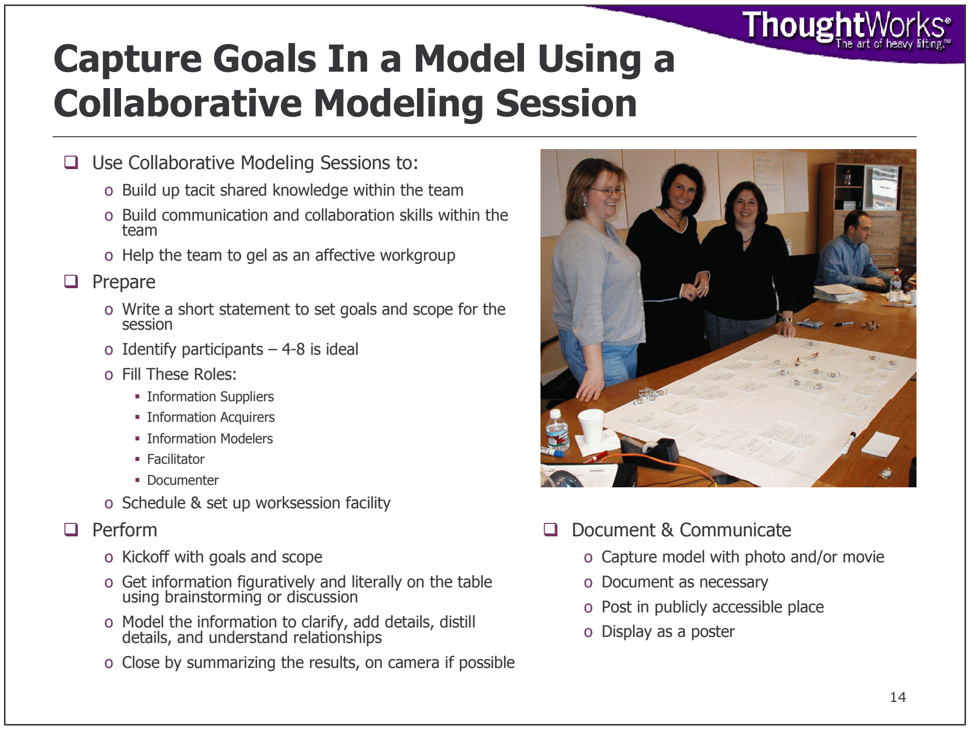
- ❑ Leverage a simple approach from the GQM methodology
- ❑ Identify and prioritize **goals**
  - o To help **identify goals** consider these questions:
    - *How will the organization improve after the delivery of this software?*
    - *What will happen if we don't deliver this software?*
  - o IRACIS - How might this software:
    - ▪ Increase Revenue, Avoid Cost, or Increase Service
- ❑ **Question** each goal:
  - *If we were making progress toward this goal, how would we know?*
  - *What would change in the business as a result of reaching this goal?*
- ❑ Use answers to these questions to identify **metrics** for goals
  - o Metrics help quantify ROI
  - o Metrics helps justify ongoing development expense
  - o Requirements to track metrics often generate important product features

13

**Additional Reading:**

- Van Solingen & Berghout, **Goal/Question/Metric Method**, 1999, McGraw-Hill Education
- Forrester Research, **Goal-Question-Metric Method Is Still The Most Pragmatic Way To Develop Metrics**, http://www.forrester.com/Research/Document/Excerpt/0,7211,37381,00.html
- Goldsmith, **Discovering Real Business Requirements for Software Project Success**, 2004, Artech House Publishers

Notes



## Capture Goals In a Model Using a Collaborative Modeling Session

- Use Collaborative Modeling Sessions to:
  - Build up tacit shared knowledge within the team
  - Build communication and collaboration skills within the team
  - Help the team to gel as an affective workgroup
- Prepare
  - Write a short statement to set goals and scope for the session
  - Identify participants – 4-8 is ideal
  - Fill These Roles:
    - Information Suppliers
    - Information Acquirers
    - Information Modelers
    - Facilitator
    - Documenter
  - Schedule & set up worksession facility
- Perform
  - Kickoff with goals and scope
  - Get information figuratively and literally on the table using brainstorming or discussion
  - Model the information to clarify, add details, distill details, and understand relationships
  - Close by summarizing the results, on camera if possible
- Document & Communicate
  - Capture model with photo and/or movie
  - Document as necessary
  - Post in publicly accessible place
  - Display as a poster

14

## Collaborative Modeling Session

The collaborative modeling session has a goal that blends the ideas of acquiring information, distilling information, and potentially collaboratively designing solutions to problems. It's best used when information exists in the heads of participants. While the information may also exist in documents, or across other models, the goal of a collaborative modeling session is extract information from the participants of the session, distill it, and modeling it in such a way that it's easy to understand and can be communicated to others.

In addition to quickly distilling and representing large amounts of information, the collaborative nature of these sorts of sessions have the affects of:

- Building up tacit shared knowledge within the team
- Building communication and collaboration skills within the team
- Helping the team to gel as an affective workgroup

As with interviews and collaborative design, the collaborative modeling session follows a similar structure of prepare, perform, evaluate.

## Preparation

Preparation takes a bit of time and engagement from one or more team members, ideally including the person who will later serve as a facilitator during the session. Preparation itself takes a bit of thought and creativity. Consider performing this activity with two team members pairing at a whiteboard or at a tabletop using card modeling. Capture information for later documentation.

## Set goals and/or scope

For a collaborative modeling session you'll have some information you wish to gather or consolidate from the participants of that session. In addition you may have a goal to leverage collaboratively interpret that information and subsequently leverage it to make design decisions. To keep the session focused, decide on the goals and scope for the session. You'll have a hard time ending it if you don't.

Try completing the sentence "this session would be successful if…" to help identify the goals.

Try completing the sentence "The information we'll need to consider is…" to help identify the scope of concern for this session.

Summarize your ideas for goals and scope in a concise statement.

## Identify participants

Choosing the best possible participants of a collaborative modeling session is one of the most important elements of success. Given the goals of collecting and distilling information and spreading that information throughout the team, who you choose will affect the information you gather, the quality of that distillation, and the ultimate destination for that information – the heads of those who participated.

Leveraging the goals and scope for the session will help you choose appropriate participants. Look for participants that can fill the following roles:

### *Information suppliers*

Information suppliers contain the critical information in their heads that needs to be modeled or distilled. They might "just know" the information because they're subject matter experts of some type. They may have studied documents that contain the information. They may have performed interviews with others that have the information in their heads. However it came to get there, information suppliers have information valuable to everyone in their heads.

Information suppliers may come to a collaborative worksession with documents or other sources of information ready to reference. They're the experts on these documents and can quickly navigate them to identify relevant details to the group.

### *Information modelers*

The ideal information modeler has used collaborative modeling techniques before to represent, organize, and distill information. They're comfortable working with index cards, sticky notes, or whiteboards. While it's true that most everyone in the session will engage in modeling, an expert modeler or two helps things move more smoothly.

### *Information acquirers*

Information acquirers need to learn and leverage the information modeled in the session. They may be a business person who needs to leverage the information to make better decisions, a designer that will need to help make specific scope decisions, a developer who'll need to make specific technical design suggestions, or a tester that needs to understand how best to test the resulting software. A good collaborative modeling sessions contains people that can acquire and leverage the information to help the design and development process.

In addition seek to fill these following roles to help the session run smoothly and preserve the outcome of the session.

### *Facilitator*

The facilitator understands the collaborative modeling approach and can help guide all the other participants through it to help meet the goals of the session. The facilitator will keep an eye on the scope of the session use a **parking lot** to park out of scope ideas that come up during the session. Sometimes multiple team members can perform the facilitator role by alternating the responsibility at different times during the worksession. But, only one person should be in the facilitator role at any given time.

### *Documenter*

At the end of the session someone will need to take responsibility for gathering up the model or models that result and documenting them in such a way that they're easily leveraged by the session participants, others in the design and development team, and possibly other external stakeholders. The documenter should be prepared to shoot a **model photo**, prepare a **model poster**, and/or **presentable distillation**.

Remember that these are roles that can be filled by anyone in the team. One team member may fill multiple roles simultaneously. For instance information suppliers will likely move to the role of modelers, and assuming they acquire information they didn't have coming into the session, they'll also be information acquirers.

However, it is difficult for the role of facilitator and documenter to be filled by someone also in the role of information inquirer. The demands of those roles make it difficult to also be sharing information effectively.

A good collaborative workgroup might contain 5-8 participants. Smaller groups might work OK, but lack the high "bandwidth" of information that can be supplied and acquired with more participants. Larger groups may also work, but take clever facilitation to make the best use of participants' time and produce useful modeled output.

Question the participation of people that don't easily fit into any role. For instance someone who doesn't have much information, won't be participating in the project longer term, and hasn't done any of this sort of collaborative modeling won't be very valuable.

## Prepare collaboration supplies

Gather **collaboration supplies** that will help the team effectively model. It's difficult to get the right people into the room. Don't let the effort stall for lack of some sticky notes, index cards, or tape.

## Schedule modeling session

A good collaborative modeling session might run 90 minutes. If the session needs to run longer consider scheduling a 15 to 30 minute break to split the session in to two. Schedule the session in a room with a large work table to model on and/or wall space to hold models and a **parking lot**.

## Performing

### Kickoff

The facilitator should kickoff the session by reminding everyone why they're here.  Review the goals and scope with the collaborative team.  Adjust the scope and goals if necessary.

Make sure everyone on the collaborative team knows each other.  Consider allowing time for introductions if people haven't worked together before.

Review the roles with participants: suppliers, modeler, acquirers, facilitator, and documenter.  Let people know which role you think they fill.

Discuss the process you'll be following during the session.

### Model

During modeling you'll perform a variety of modeling techniques in a sequence that best suits your goals.  It's common to engage in brainstorming or **card storming** to get a lot of information on the table.  It's common to build **affinity diagrams** to find distill and find relationships in the information.  It's common to build other ad hoc models to show the relationships of information in the model.  It's common to use **prioritization** mechanisms to identify the most important or focal areas of the model.

While modeling you'll find that important ideas emerge that might fall out of scope for this session.  Make sure you have **parking lots** and **feed forward bins** set up to receive that information.

### Summarize and reflect

A few minutes prior to end the session take a moment to gather up all the elements of the model.  Allow one or more team members to verbally summarize what the model represents.  Allow for time to make minor corrections.

If more time for modeling is needed, discuss a subsequent session.

The documenter should discuss what they will do to document and communicate the information to others.

Before parting, take a moment to gather **parting takeaways** from the team members.

## Document, & Communicate

The documenter will have the responsibility of collecting the model or models built and communicating them back to others.  A good documenter will find help or delegate as much as possible.  Before leaving the documenter should snap some **model photos** to help recall the exact positions of elements in the model and to help communicate the results to others.

The results of the session need to be communicated to others.  This allows them to reflect on what happened after the session and add or change details if necessary.  Choose a communication strategy that is most visible for the team working on the design and development of the software.  A **model poster** prominently displayed in a public area allows members of the team to gather around it for discussions and to easily write ad hoc changes on to the model.  A **presentable electronic distillation** allows others to use it in meeting presentations to communicate outward to others, or for the information to be posted for sharing with others not collocated with the design and development team.

## CardStorming

You've likely engaged in brainstorming activities before.

Often times brainstorming occurs by those in the room announcing their ideas, and a facilitator writing them down on a white board or flip chart paper. We're then left with the challenge of refining the list – removing duplicates and obviously poor candidate ideas. Then given a smaller subset of good ideas, we may then need to find common ideas or themes, or simply prioritize the list. All this is challenging when starting from a list on paper or whiteboard. Basically the brainstorming part was easy, making sense of the results becomes the difficult bit.

But we're card modelers now, right? We know that using card sorting and modeling techniques we can make sense of large amounts of information.

CardStorming is a brainstorming technique described by Larry Constantine & Lucy Lockwood in Software For Use. It's a very effective way to brainstorm then move quickly and directly to using card sorting and modeling techniques to do the tough work with the information we've acquired.

### To run a CardStorming session:

- Arrange a group of participants facing each other around a worktable
- Place on the table a couple big stacks of index cards and fat felt tipped markers
- Designate one or two participants as **recorders**. One recorder is sufficient in a group of 5 or less, two is better in a group of 6 or more. It'll be the recorders' responsibility to record ideas directly onto index cards as participants shout them out.
- Determine how you'll end your session: either time-box the session using a kitchen timer, or agree the session ends when there's a long uncomfortable silence between ideas.
- Start the brainstorming by inviting participants to shout out ideas while recorders write them on 3x5 cards and toss them into the middle of the worktable where participants can see them.
- When the time is up, or an uncomfortable silence is reached, it's time to stop.

Remember the most important rule of brainstorming: **do not discuss or criticize ideas during the session**. It's okay to ask a clarifying question but nothing more.

Brainstorming is supposed to be fast paced and fun. You can kick creativity up by shouting out silly or obviously inappropriate ideas.

If the recorder gets behind, it's ok for him to shout "whoa, whoa – give me a minute here." It's also okay for another person to help out and begin recording. But it is important the participants see and hear the ideas being tossed out. If everyone talks at once and/or writes at once, they're not hearing, thinking about, and leveraging each other's ideas to come up with more ideas.

When brainstorming is complete there are a few immediate actions you can take:

## Filtering and Prioritization

### Filtering

Filter and clean the list by sorting the list into three piles: obviously useful ides, ideas to discuss, and ideas to discard. You'll find silly things and duplicates obviously arrive in the pile to discard. To filter:

- Label three areas on your table: keep, discuss, and discard.
- Split the deck of ideas into two or three piles
- Hand a pile to a workshop participant
- Participants start placing the pile into one of the three areas. Place the cards such that everyone else can see them. If someone disagrees with the placement, they're free to move the card to another pile.
- When all cards are placed, bundle up the "discard" pile and toss it in the trash.

### Simple prioritization

With a list of ideas written on cards, it's easy to begin prioritization.

- Select an area on a worktable to place the cards while you're establishing their priority.
- Label one area "high priority" – usually far left of top
- Label another are "low priority" – usually far right or bottom
- Split the deck of ideas into 2 or 3 piles
- Hand a pile to a workshop participant
- Participants start placing the ideas on the table where they believe the priority should be – toward the high end if it's high, toward the low end if it's low.

This can work well because it's often easier to prioritize an idea against one or two ideas that it's similar to. Laying the ideas out spatially allows us to easily find and place an idea near ideas of relative importance.

From here it's easy to begin detailed discussions about ideas and priority. It's easy to change our mind and slide a card to a different place on the table.

### Democratic prioritization:

Oftentimes it's not critical that we understand where all ideas fit in priority but rather understand what the top three or four ideas to focus on are. Sometimes you've already spent time arranging index cards into a model that represents your understanding of relationships between ideas and rearranging the ideas into priority order at that point will break the model showing these relationships.

Use democratic prioritization to allow collaborators to vote for the highest priority ideas.

- Arrange index cards on a table in an arrangement that that makes it easy to find information. You may already have a card arrangement such as a role model or task model that you're working with.
- Give each member a number of tokens to vote with and ask collaborators to place votes on the ideas they think are most important. They may place multiple tokens on an idea. As with all democratic processes participants may try to persuade each

other to vote differently – that's good. The conversation that occurs helps deepen everyone's understanding of the context these ideas are prioritized in.

*For tokens: stickers work okay, or simply making marks on index cards that you vote for. But, I prefer physical objects that are easy to move so that collaborators can change their minds or coerce each other to vote a certain way. The wrapped pieces of candy mentioned as useful in your toolkit work well as voting tokens.*
*The number of votes to give each member is tricky. Each member should have one or two votes less than they wished they had – to really force them to think hard about what they believe is important. The number of votes is a function of the number of participants, and the number of ideas in model. For a model with 50-60 cards built by 3-4 collaborators four votes per collaborator might be in order. Generally 2-5 votes are appropriate for most circumstances.*

- When all votes are placed, total the number of votes on each index card that received votes. You'll find that most arrangements of index cards have 2-3 ideas that are clear winners, and 6 or more that receive votes. Before removing any voting tokens, mark the cards with a symbol such as a star for each vote, four votes get four stars. Use a bright color of pen ink to make it easy to identify the index cards with votes. It's easy in a model marked this way to identify the high ranking ideas, and distinguish the highest ranking ideas from lower ranking ideas and unranked ideas.

## Myths and misconceptions about prioritization

- Prioritization is based on context – which might change over time and prioritization with it
- Fine grain prioritization difference often don't matter – except when they're near the "edge"
- Use "focal" as a label for high priority items

## Additional Reading

- Gottesdiener, **Requirements by Collaboration** (Addison-Wesley, 2002)

## 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:

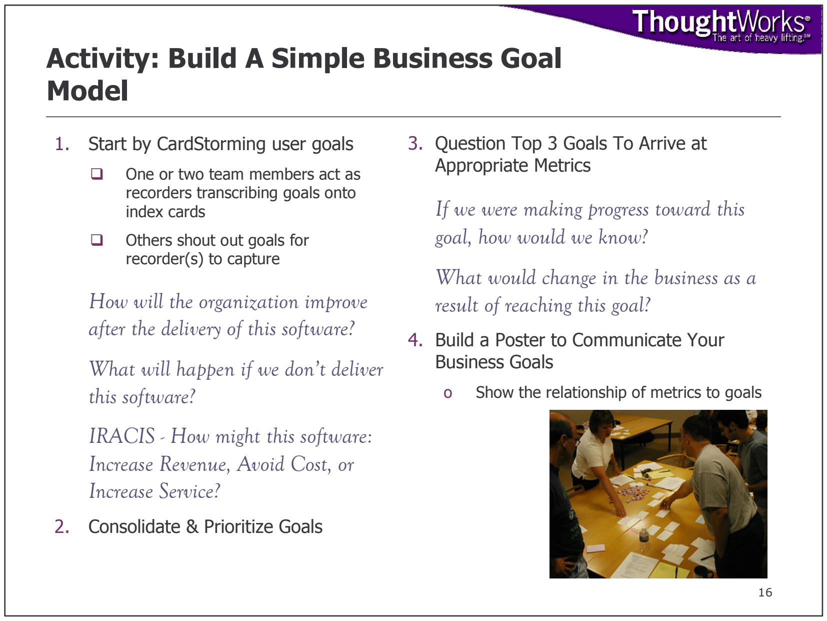
While considering requirement, many designers make the mistake of only considering direct users of the software. Those with concerns about the effectiveness of the software and/or its ROI are often set aside as "stakeholders." Stakeholder concerns are often addressed in the software's detail design by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

## Business Goal and Metric Model

The business desires to build software to reach particular goals. Those goals might be stated a number of different ways in documents that charter or fund the project. Goals may be spoken of in a number of different ways by business stakeholders within the organization. Often those business goals are stated in an ambiguous or inconsistent manner. This makes it difficult for any one person to deliver a consistent answer on what the business goals are. If the business goals are unclear, how then can we determine if software being built helps the business reach those goals?



ThoughtWorks® The art of heavy lifting.℠

### Activity: Build A Simple Business Goal Model

1. Start by CardStorming user goals
   - One or two team members act as recorders transcribing goals onto index cards
   - Others shout out goals for recorder(s) to capture

   *How will the organization improve after the delivery of this software?*

   *What will happen if we don't deliver this software?*

   *IRACIS - How might this software: Increase Revenue, Avoid Cost, or Increase Service?*
2. Consolidate & Prioritize Goals
3. Question Top 3 Goals To Arrive at Appropriate Metrics

   *If we were making progress toward this goal, how would we know?*

   *What would change in the business as a result of reaching this goal?*
4. Build a Poster to Communicate Your Business Goals
   - Show the relationship of metrics to goals

16

Unclear business goals make it difficult to easily determine if software you've chosen to build actually helps the business reach those goals.

A good first step is to pull business stakeholders together and attempt to agree on goals. This is often difficult and results in goals written in such a way that all business stakeholders can agree on them which often means they're imprecise. What's also common are goals stated in such a way that it's difficult or impossible to determine if the goal is being met or if progress is being made towards meeting a goal. What helps a project most are concise easy to leverage business goals coupled with metrics that allow the business and the software's designers to determine if the product is successful. These goals act first as a target for choosing user constituencies to support, and features to build. Without this design target, successful results are unlikely, accidental and often but sadly undetectable.

Building a simple business goal and metric model distills what we understand about business goals and gives its reader an understanding of how progress towards that goal is measured. The business goal model acts as the design target that allows us to choose software users to support and features to build that help reach these goals.

## The Model

The business goal model actually doesn't look the way you'd normally expect a model to. It's not composed of boxes connected by lines, rather it's a simple bulleted list of goals, and metrics associated with each goal. There should be very few goals, 1-5. Each goal should be supported by metrics that help us measure progress towards that goal, 1-3 metrics per goal. I refer to it as a model because it's a distillation of what we believe to be true at the time it was created. We'll base other decisions on this model. But, since it's a model, we know it might change; which might result in changes to decisions based on this model.

A best way to construct a goal model is to use a collaborative modeling session. Follow the general approach to a collaborative modeling session along with the specific advice below.

## Plan a goal modeling session

### 1. Identify scope of concern

Goals can occur at a variety of different levels. Starting at very high levels like: happiness and world peace, to very low or detailed levels like: get to work on time. Set a scope of concern for the goals you're about to model by writing a short statement about that scope. A good statement might be:

> "We'd like to identify goals that would allow us to determine if the next release our software is successful."
>
> or
>
> "We'd like to identify goals for the new product we're considering funding the development of."

To help focus scope consider the product or business area of focus and the timeframe. A single product for next release for example. Other example might be an entire business unit over the next year or a single team developing a single set of features for the next internal release of a software product.

Given the session goal of creating a simple goal model, and the goal scope you'd like to target, you're ready to move forward to identify best participants.

### 2. Identify participants

An effective goal modeling session might include 5-8 participants. Start by identifying the participants of the session.

- One participant serves as **facilitator**. The facilitator may be a member of the implementation team, but it's a bad idea for the role to be filled by a business stakeholder.
- Participation from **key individuals from the development team** responsible for building the software is important. Key individuals might include a leader developer, lead tester, lead business analyst, lead user experience or designer. Include at least two individuals from the development team. They will function as information acquirers and modelers.
- **Business stakeholders** make up the remainder of the participants. These might include a project sponsor, members of constituent user groups, sales and marketing team members, technical support team members, or members of the organizations executive team. They will function primarily as information suppliers.

If you're unable to assemble a team balanced with business stakeholders and design and development team leaders, you may assemble others that can make assumptions about business goals. But consider it a risk to the quality of the results. Try to obtain feedback on the resulting model from the business stakeholders and expect that feedback will likely result in change to the model.

You may elect to conduct business stakeholder interviews and bring the distilled data from those interviews to the modeling session. One or more members of the team should study this data, preferably the person who conducted interviews and distilled the data. They will function as information suppliers.  Plan time in the meeting to review this data prior to building the goal model.

### 3. Prepare collaboration supplies to use for capturing results during the modeling session.

You'll likely need index cards to model, pens, tape, and poster paper.

### 4. Set a time and place for the meeting.

Expect the meeting to be as short as 30 minutes, but don't plan to spend longer than 90 minutes.

## Conduct the modeling session

### 1. Kickoff modeling session and set goal context

Kickoff the modeling session.

Making sure the group knows each other.  Allow them to introduce themselves if they haven't worked together before.

Describe to the group the process you'll be following.

Set up parking lots of feed forward bins as necessary.

Discuss the scope of concern for the goals you'll be modeling.  Adjust the scope of concern as necessary based on the discussion.

### 2. Brainstorm Goals

Start by creating a candidate list of business goals and business problems to solve. Brainstorm the goals and problems on to index cards.  You may wish to use a brainstorming technique such as cardstorming.

#### *Goals or pains*

The evil twin of the goal is the pain point.  At times it may be difficult to understand exactly what the goal of a particular activity is, while it may be easy to name the pain or problem that you're trying to eliminate.

For example say you're working for a hypothetical software company where the current release of the product has generated huge numbers of support calls.  This has resulted in the need to increase support staff at great expense.  And still customer hold times are high, and customer satisfaction is decreasing.  This is clearly a problem that needs to be solved in the next release.  A brainstorming session focused around the next release of the software might yield the pain points: "too many customer support calls," or "poor customer satisfaction."  It's easy to move from these statements of pain to goals that might alleviate the pain such as "Decrease need for customer support."

For the purpose of brainstorming business goals, consider problems or pain points as interchangeable with goals.  We'll have opportunity to consider the language we choose to express them later.  Coach the team brainstorming goals and pains to give both goals and pains.

## 3. Refine Goals

Once brainstorming has slowed or stopped, cluster the goals and pains using an affinity diagram.

For each cluster of goals and pains, as a team write a new card that best summarizes the goals and pains in the cluster. If it's difficult to write a goal statement splitting the cluster may help separate out ideas that are making it difficult to summarize.

## 4. Prioritize goals

Gather the cards that summarize each cluster of goals. These will be the refined list of goals we'll prioritize.

Use the scope of concern to help set foundation for the prioritization. It's good to restate that scope, along with questions that frame the prioritization exercise. Questions like: "which of these goals are most critical to achieve for this product for the scope we're considering?" are helpful.

You could lay the goal card on the table to perform simple prioritization. Or, you may choose to prioritize goals using a voting approach as in democratic prioritization.

Hopefully your session contains a mixed group of business stakeholders responsible for defining goals and setting priority, and development team members responsible for designing and building the product to meet those goals. You may choose to allow only the business stakeholders perform the prioritization while others look on and ask questions.

At this point a good prioritized goals list has 1-3 goals that are easy to agree on as high priority. The goal list could be fairly long, but it's difficult to manage the design and development of a product with too many goals to consider. Choose the highest priority goals to single out and emphasize. One to three are ideal. Try to choose no more than five. These become our **focal business goals**.

## 5. Question goals and capture metrics for focal business goals

To make these goals more tangible and allow us to make better downstream design choices it's important to identify metrics that help us determine if we're making progress towards these goals.

For each goal ask the question:

> *"How would we know if we were making progress towards this goal?"*

Brainstorm and capture the answers to these questions on index cards as candidate metrics for each goal.

For example given a goal like:

> "decrease need for customer support"

possible metrics might be:

- "number of support calls relative to products sold."
- "average duration of support calls"
- "number of email requests for support'

Defining measurable goals like this might compel us to do further analysis on the nature of support calls we're getting today. Areas of the product that have the highest frequency and duration of support goals may be best areas to focus design and development efforts on.

Allow subjective metrics along with a more objective way of gathering an measuring change in them. For example let's assume we're upgrading software used in our internal call center. Today our agents claim that they really hate the software. It's often mentioned in exist interviews as a factor in quitting.

Given a goal like:

> "increase user satisfaction"

Possible metrics might be:

> "satisfaction rating 1 to 10, 10 being very satisfied"

Take a measurement today by polling the current agents of the software. The same question might be asked of agents testing interim released of the software, and of agents after the software is released and installed. Changes in this measurement will give some indication of increases or decreases in user satisfaction.

After writing out possible metrics for a goal, eliminate obviously bad ideas. As a group discuss the metrics. Do they really help us determine if we're making progress on this goal? Are there other factors that influence the metric such that if may give us inaccurate measurements of our goal? Multiple metrics per goal better give a picture of progress on the goal. If one metric is influenced by other factors, possibly the other metrics will help give better indications.

### *Rewrite goals*

Often the discussion of metrics results in the realization that a goal is difficult or impossible to measure. It's at this time we need to ask if this is a reasonable goal. If our organization can't detect if we're making progress against it, then it'll be very difficult to let this goal guide us in making detailed design and scope decisions.

> *Rewrite or eliminate goals that can't easily be measured.*

For each goal identify one to three metrics you might use to measure progress on the goal.

### *Feed forward product feature ideas*

Often when defining metrics for goals you may detect that the software may need functionality that captures and reports on these metrics so that can be observed. You might realize that someone in management would be very interested in seeing these metrics on a regular basis. This information is useful to place in a feed forward bin labeled "product feature ideas."

As ideas come up for possible software features, park them in a feed forward bin for later consideration. Then continue on with your discussion.

## 6. Summarize and reflect

Close the modeling session by reviewing the prioritized focal business goals. Ask a business sponsor or other business stakeholder to do this review. There should be one to three goals, ideally no more than five. For each goal review the metrics that measure progress towards that goal. If possible use a camera to record a video of the review.

If the model wasn't completed before the end of the timebox, discuss times to continue the session.

Identify the team member who will be responsible for documenting and posting the goals where they can be seen and used.

Close by circling around the group for parting takeaways.

## Distill, document, and communicate goals

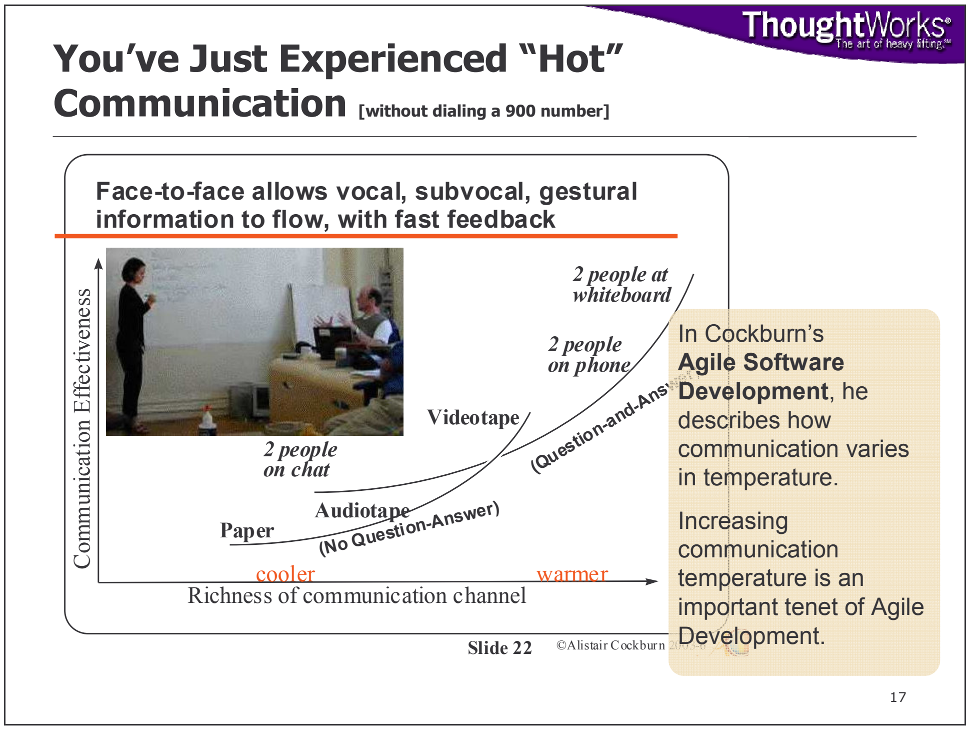
For these goals to be relevant and leveraged, they must be accurately captured and displayed in a prominent location.

Record the names of the modeling session participants.

Take a model photo to help participants recall what happened during the session.

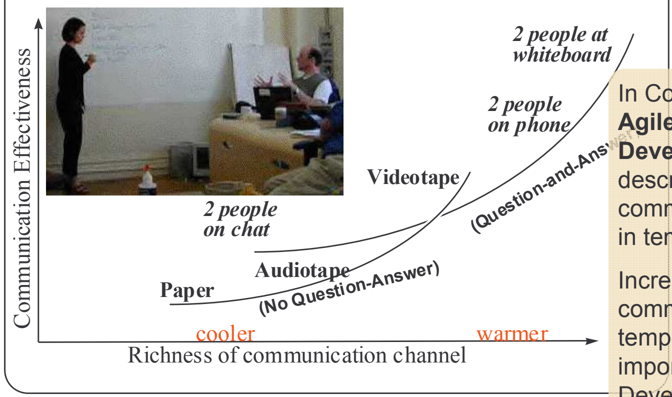
Document the model as a model poster, powerpoint distillation, or in some other document that can be easily used by the project team and external business stakeholders.

Along with the focal goals and metrics, the documented business goals should contain the names of the group responsible for arriving at them and model photo to help others relate to the process used to arrive at these goals.



## You've Just Experienced "Hot" Communication [without dialing a 900 number]

Face-to-face allows vocal, subvocal, gestural information to flow, with fast feedback

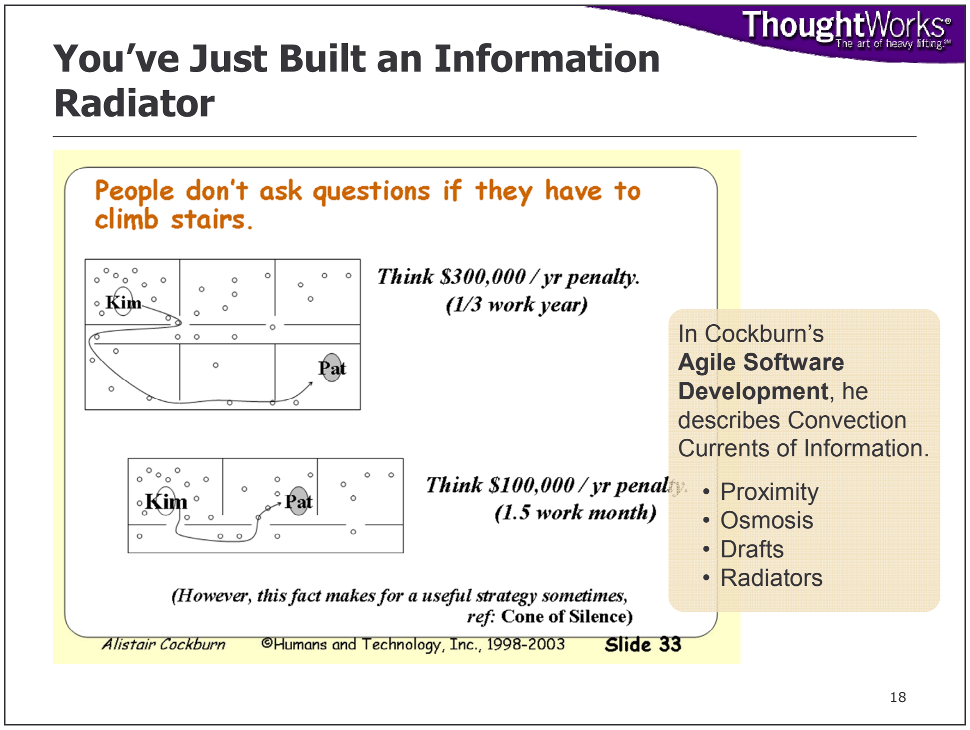


Slide 22 ©Alistair Cockburn

In Cockburn's **Agile Software Development**, he describes how communication varies in temperature.

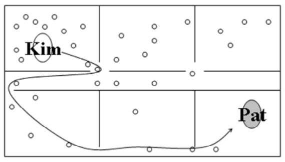
Increasing communication temperature is an important tenet of Agile Development.

17

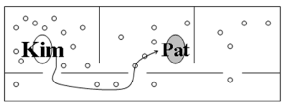


## You've Just Built an Information Radiator

People don't ask questions if they have to climb stairs.



*Think $300,000 / yr penalty.*
*(1/3 work year)*



*Think $100,000 / yr penalty.*
*(1.5 work month)*

*(However, this fact makes for a useful strategy sometimes, ref:* Cone of Silence)

*Alistair Cockburn* ©Humans and Technology, Inc., 1998-2003 **Slide 33**

In Cockburn's **Agile Software Development**, he describes Convection Currents of Information.

- Proximity
- Osmosis
- Drafts
- Radiators

18

Notes

**Additional Reading:**

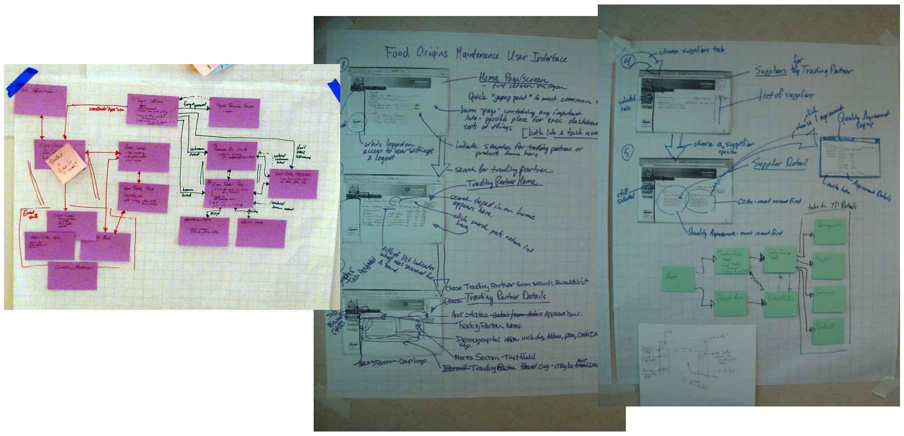
- Cockburn, **Agile Software Development 2nd Edition** (2006, Addison-Wesley)

## Agile Environments Leverage Information Radiators to Socialize Information

A task model shows workflow, supports release planning and incremental development

19

## Agile Environments Leverage Information Radiators to Socialize Information



Navigation Maps and Storyboards describe user interactions

20

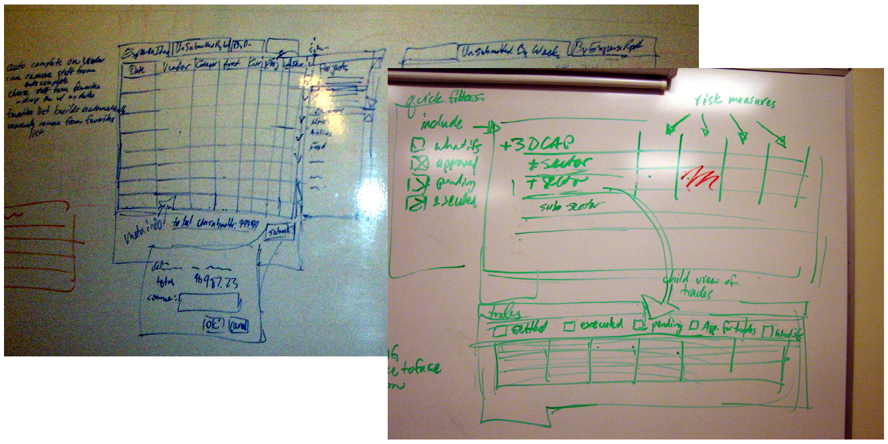
Notes

**Additional Reading:**

- Patton**, Finding the Forest In the Trees** (OOPSLA practitioner report, 2005, http://www.abstractics.com/papers/FindingTheForest.pdf)

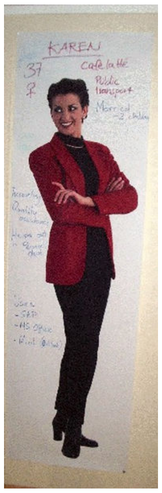
Notes

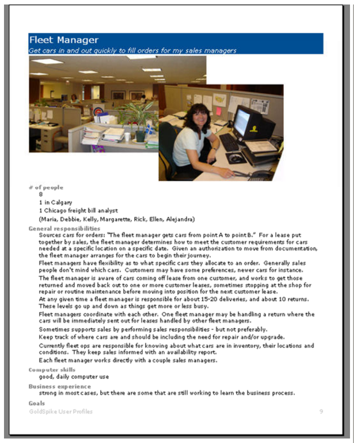
## Agile Environments Leverage Information Radiators to Socialize Information



Development often proceeds leveraging whiteboard wireframe prototypes

## Agile Environments Leverage Information Radiators to Socialize Information





Fleet Manager

*Get cars in and out quickly to fill orders for my sales managers*

**# of people**
8
1 in Calgary
1 Chicago freight bill analyst
(Maria, Debbie, Kelly, Margarette, Rick, Ellen, Alejandra)

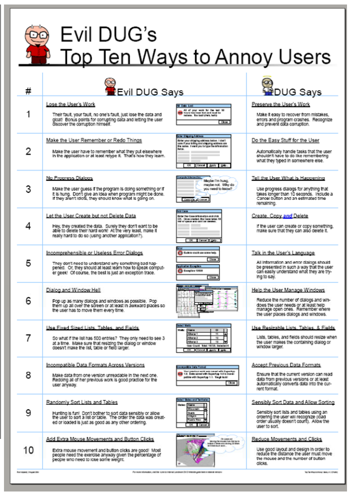
**General responsibilities**
Sources cars for orders: "The fleet manager gets cars from point A to point B." For a lease put together by sales, the fleet manager determines how to meet the customer requirements for cars needed at a specific location on a specific date. Given an authorization to move from documentation, the fleet manager arranges for the cars to begin their journey.
Fleet managers have flexibility as to what specific cars they allocate to an order. Generally sales people don't mind which cars. Customers may have some preferences, newer cars for instance.
The fleet manager is aware of cars coming off lease from one customer, and works to get those returned and moved back out to one or more customer leases, sometimes stopping at the shop for repair or routine maintenance before moving into position for the next customer lease.
At any given time a fleet manager is responsible for about 15-20 deliveries, and about 10 returns. These levels go up and down as things get more or less busy.
Fleet managers coordinate with each other. One fleet manager may be handling a return where the cars will be immediately sent out for leases handled by other fleet managers.
Sometimes supports sales by performing sales responsibilities - but not preferably.
Keep track of where cars are and should be including the need for repair and/or upgrade.
Currently fleet ops are responsible for knowing about what cars are in inventory, their locations and conditions. They keep sales informed with an availability report.
Each fleet manager works directly with a couple sales managers.

**Computer skills**
good, daily computer use

**Business experience**
strong in most cases, but there are some that are still working to learn the business process.

**Goals**

GoldSpike User Profiles 9



Evil DUG's
Top Ten Ways to Annoy Users

| # | Evil DUG Says | DUG Says |
|---|---|---|
| 1 | Lose the User's Work — Their fault, your fault, no one's fault, just lose the data and gloat! Bonus points for corrupting data and letting the user discover the corruption himself. | Preserve the User's Work — Make it easy to recover from mistakes, errors and program crashes. Recognize and prevent data corruption. |
| 2 | Make the User Remember or Redo Things — Make the user have to remember what they put elsewhere in the application or at least retype it. That's how they learn. | Do the Easy Stuff for the User — Automatically handle tasks that the user shouldn't have to do like remembering what they typed in somewhere else. |
| 3 | No Progress Dialogs — Make the user guess if the program is doing something or if it is hung. Don't give an idea when program might be done. If they aren't idiots, they should know what is going on. | Tell the User What is Happening — Use progress dialogs for anything that takes longer than 10 seconds. Include a Cancel button and an estimated time remaining. |
| 4 | Let the User Create but not Delete Data — Hey, they created the data. Surely they don't want to be able to delete their hard work! At the very least, make it really hard to do so (using another application?). | Create, Copy *and* Delete — If the user can create or copy something, make sure that they can also delete it. |
| 5 | Incomprehensible or Useless Error Dialogs — They don't need to understand why something bad happened. Or, they should at least learn how to speak computer geek! Of course, the best is just an exception trace. | Talk in the User's Language — All information and error dialogs should be presented in such a way that the user can easily understand what they are trying to say. |
| 6 | Dialog and Window Hell — Pop up as many dialogs and windows as possible. Pop them up all over the screen or at least in awkward places so the user has to move them every time. | Help the User Manage Windows — Reduce the number of dialogs and windows the user needs or at least help manage open ones. Remember where the user places dialogs and windows. |
| 7 | Use Fixed Sized Lists, Tables, and Fields — So what if the list has 500 entries? They only need to see 3 at a time. Make sure that resizing the dialog or window doesn't make the list, table or field larger. | Use Resizable Lists, Tables, & Fields — Lists, tables, and fields should resize when the user makes the containing dialog or window larger. |
| 8 | Incompatible Data Formats Across Versions — Make data from one version unreadable in the next one. Redoing all of her previous work is good practice for the user anyway. | Accept Previous Data Formats — Ensure that the current version can read data from previous versions or at least automatically converts data into the current format. |
| 9 | Randomly Sort Lists and Tables — Hunting is fun! Don't bother to sort data sensibly or allow the user to sort a list or table. The order the data was created or loaded is just as good as any other ordering. | Sensibly Sort Data and Allow Sorting — Sensibly sort lists and tables using an ordering the user will recognize (load order usually doesn't count). Allow the user to sort. |
| 10 | Add Extra Mouse Movements and Button Clicks — Extra mouse movement and button clicks are good! Most people need the exercise anyway given the percentage of people who need to lose some weight. | Reduce Movements and Clicks — Use good layout and design in order to reduce the distance the user must move the mouse and the number of button clicks. |

User models and UI guidelines communicated in posters

**Additional Reading:**

- Vutpakdi , **Top Ten Ways Poster**, http://tech.groups.yahoo.com/group/agile-usability/files/Vutpakdi%20Examples%26Presentation/

Notes

## Large Displayed Models Serve as a Backdrop for Ad Hoc Collaboration

Brian, Frank, and Justin discuss their work with Mark against the backdrop of a workflow model

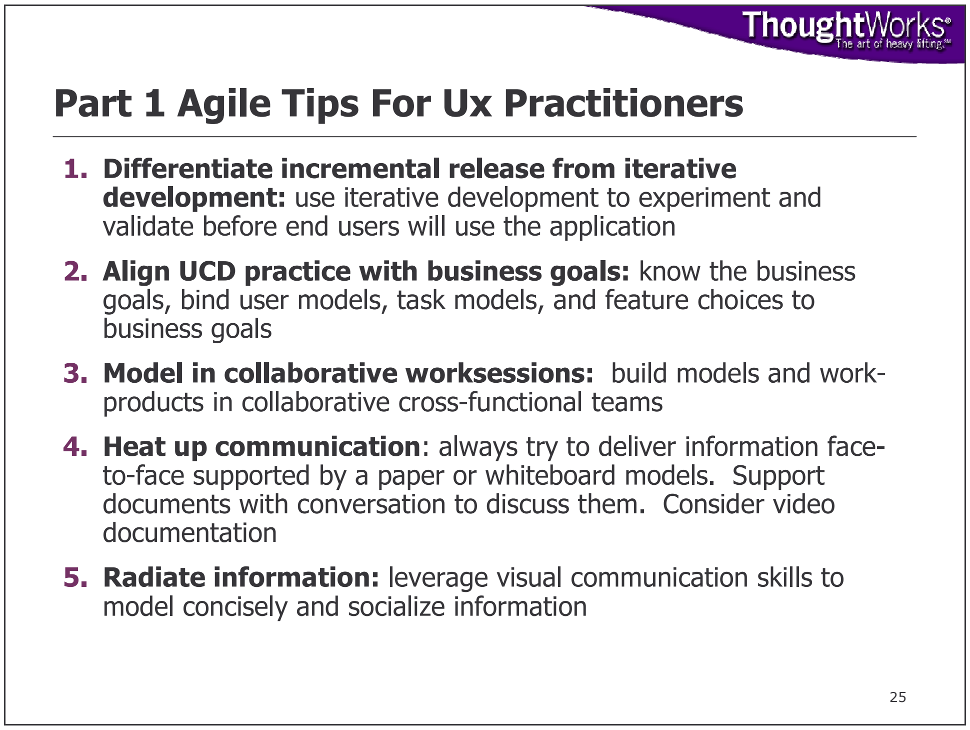
23

## Recorded Discussions While Building a Model Serve as Documentation

Zack explains the lifecycle of a railroad car lease to me using the domain objects in the system

24

Notes



## Part 1 Agile Tips For Ux Practitioners

1. **Differentiate incremental release from iterative development:** use iterative development to experiment and validate before end users will use the application
2. **Align UCD practice with business goals:** know the business goals, bind user models, task models, and feature choices to business goals
3. **Model in collaborative worksessions:** build models and work-products in collaborative cross-functional teams
4. **Heat up communication**: always try to deliver information face-to-face supported by a paper or whiteboard models. Support documents with conversation to discuss them. Consider video documentation
5. **Radiate information:** leverage visual communication skills to model concisely and socialize information

25

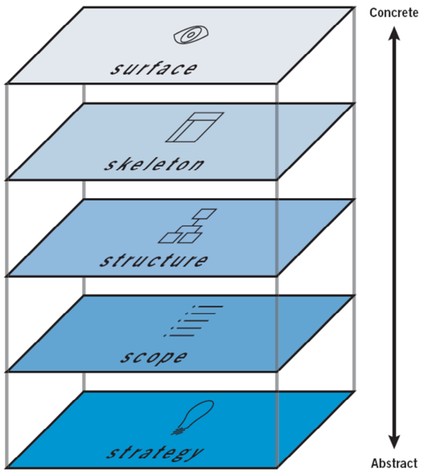
# Part 2: Project Inception & Planning

In part two I'll describe Garrett's simple model of User Centered Design that's helpful in explaining the work of designers to Agile practitioners.  Using that model and correlating it with our model of Agile Development we'll get some suggestions about how best to handle project inception and planning.

We'll build a simple user model to understand our users.  We'll build a simple task model to help us visualize our users' workflow then use that model to help us plan multiple incremental releases of our software.

ThoughtWorks® The art of heavy lifting.℠

## Garrett's Elements Model Explains Clearly How User Experience is Built From Dependent Layers

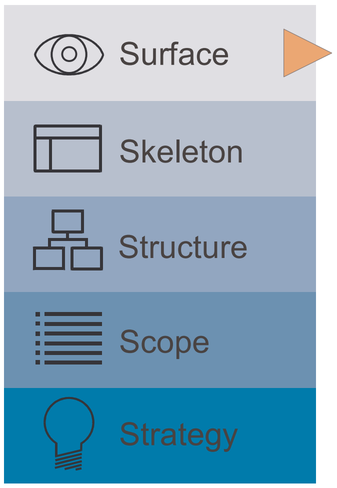


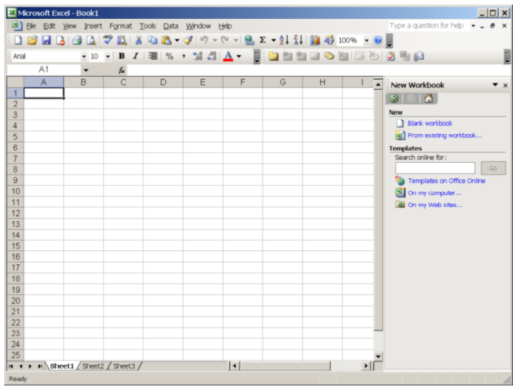
Jesse James Garrett's **Elements of User Experience**

28

ThoughtWorks® The art of heavy lifting.℠

## The Surface Layer Describes Finished Visual Design Aspects





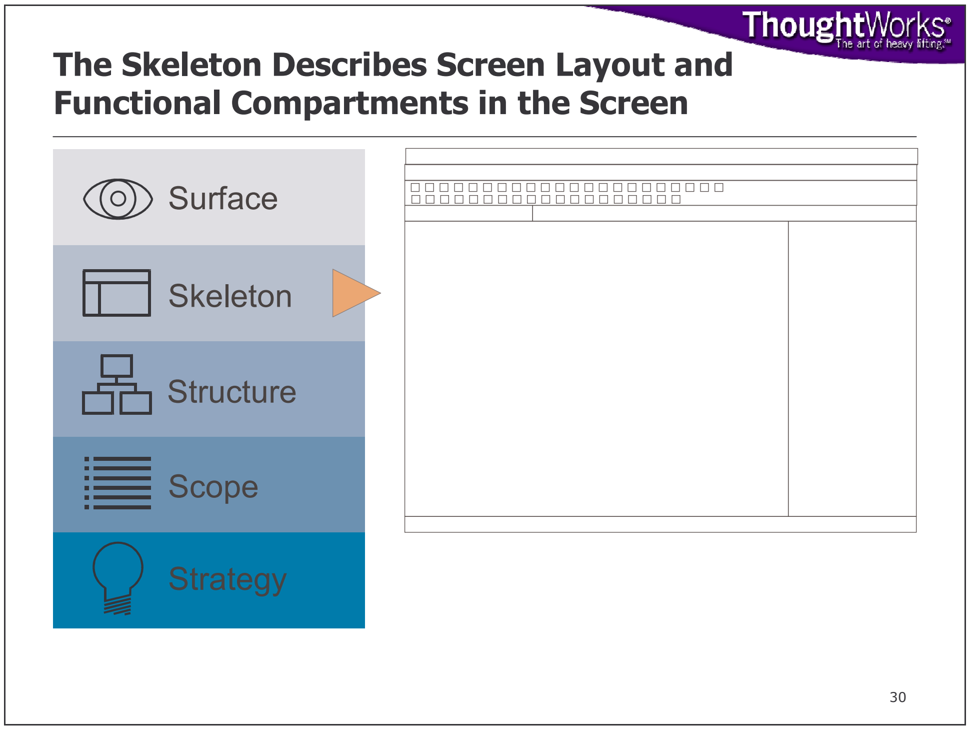
29

## Notes

### Additional Reading:

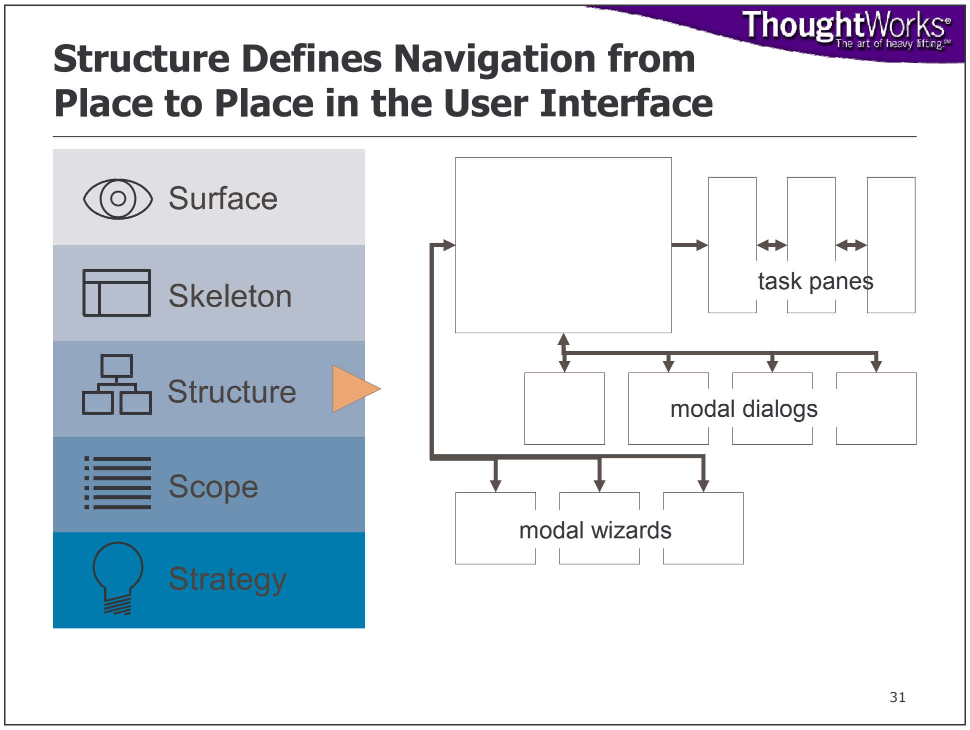
- Garrett, **Elements of User Experience** (http://jjg.net/elements/)

Notes



## The Skeleton Describes Screen Layout and Functional Compartments in the Screen

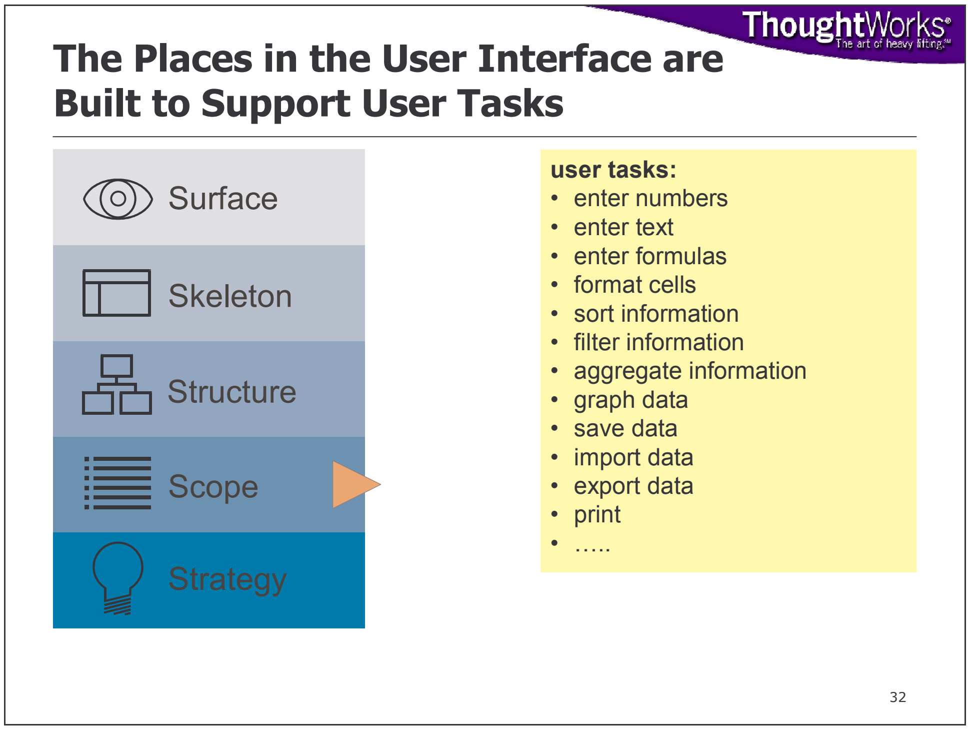
30



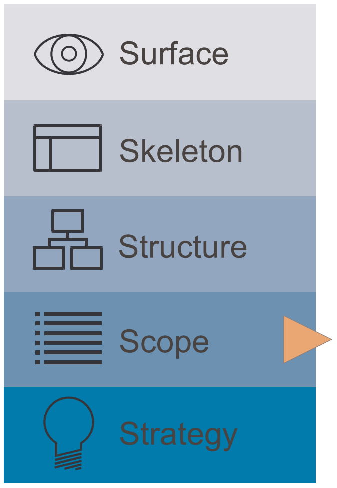
## Structure Defines Navigation from Place to Place in the User Interface

31

Notes



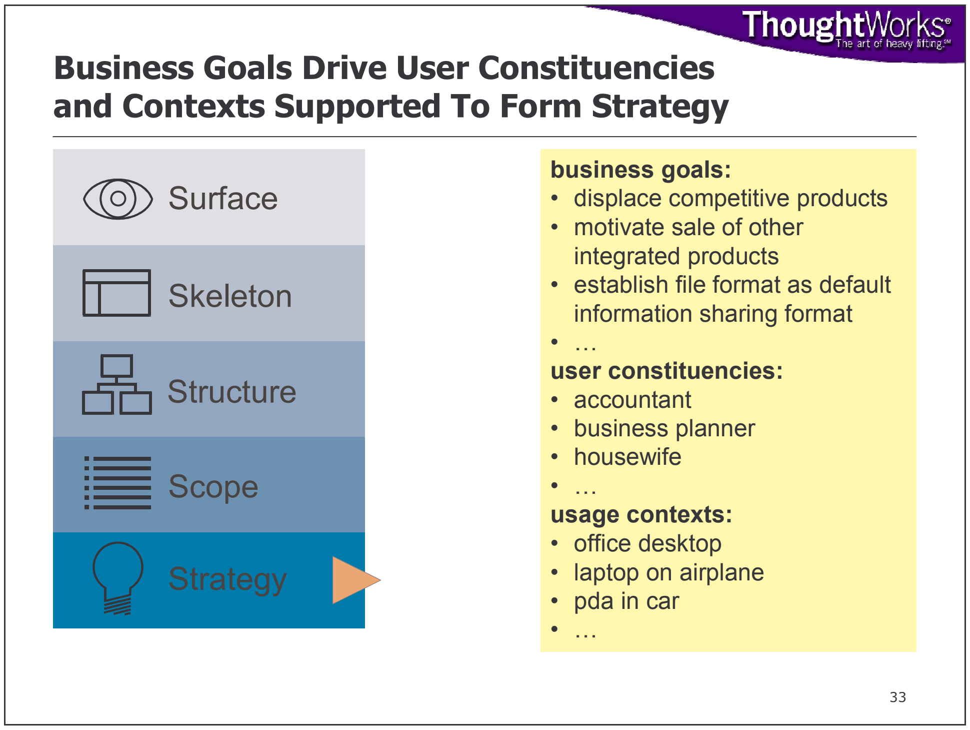
## The Places in the User Interface are Built to Support User Tasks



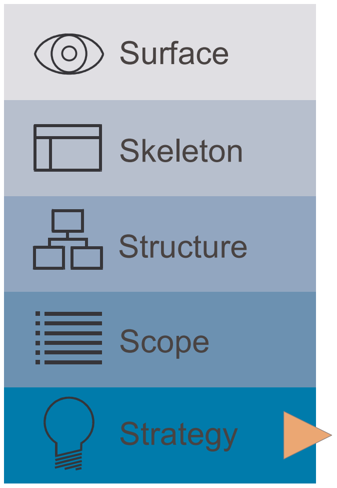
**user tasks:**

- enter numbers
- enter text
- enter formulas
- format cells
- sort information
- filter information
- aggregate information
- graph data
- save data
- import data
- export data
- print
- …..

32



## Business Goals Drive User Constituencies and Contexts Supported To Form Strategy



**business goals:**

- displace competitive products
- motivate sale of other integrated products
- establish file format as default information sharing format
- …

**user constituencies:**

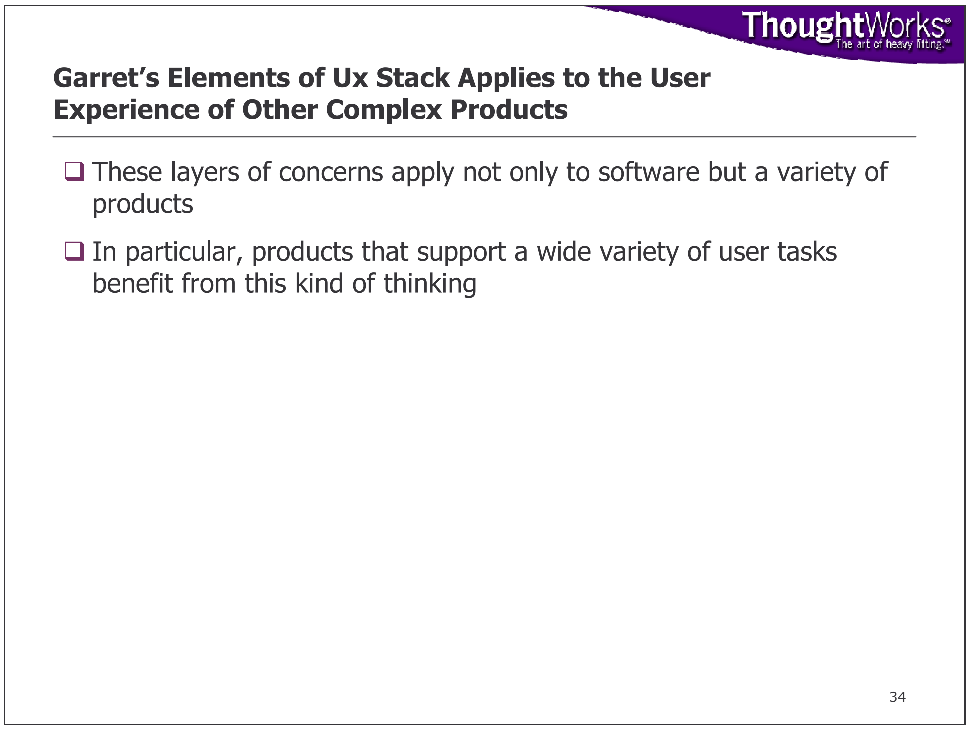
- accountant
- business planner
- housewife
- …

**usage contexts:**

- office desktop
- laptop on airplane
- pda in car
- …

33

Notes

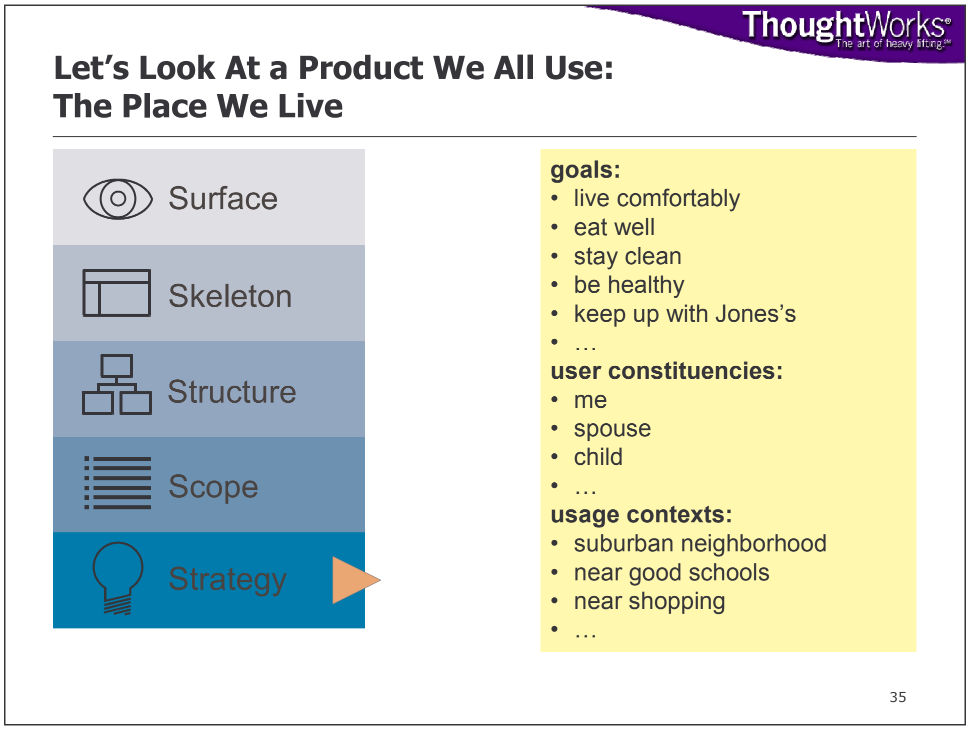


ThoughtWorks® The art of heavy lifting.℠

## Garret's Elements of Ux Stack Applies to the User Experience of Other Complex Products

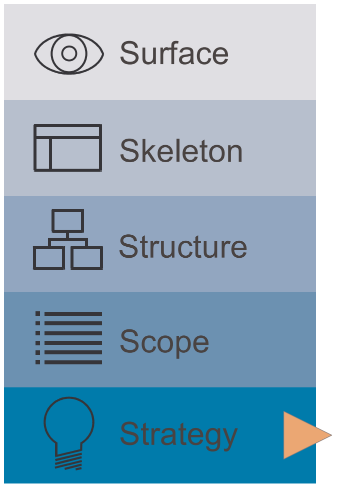
- ❑ These layers of concerns apply not only to software but a variety of products
- ❑ In particular, products that support a wide variety of user tasks benefit from this kind of thinking

34



ThoughtWorks® The art of heavy lifting.℠

## Let's Look At a Product We All Use: The Place We Live



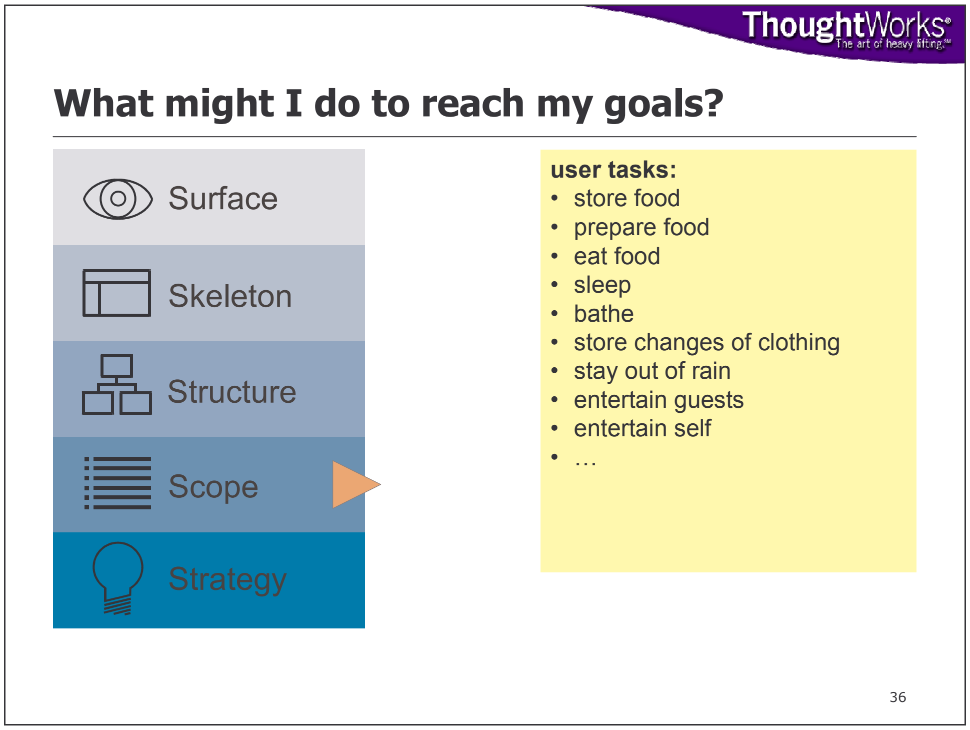
**goals:**
- live comfortably
- eat well
- stay clean
- be healthy
- keep up with Jones's
- …

**user constituencies:**
- me
- spouse
- child
- …

**usage contexts:**
- suburban neighborhood
- near good schools
- near shopping
- …

35

Notes

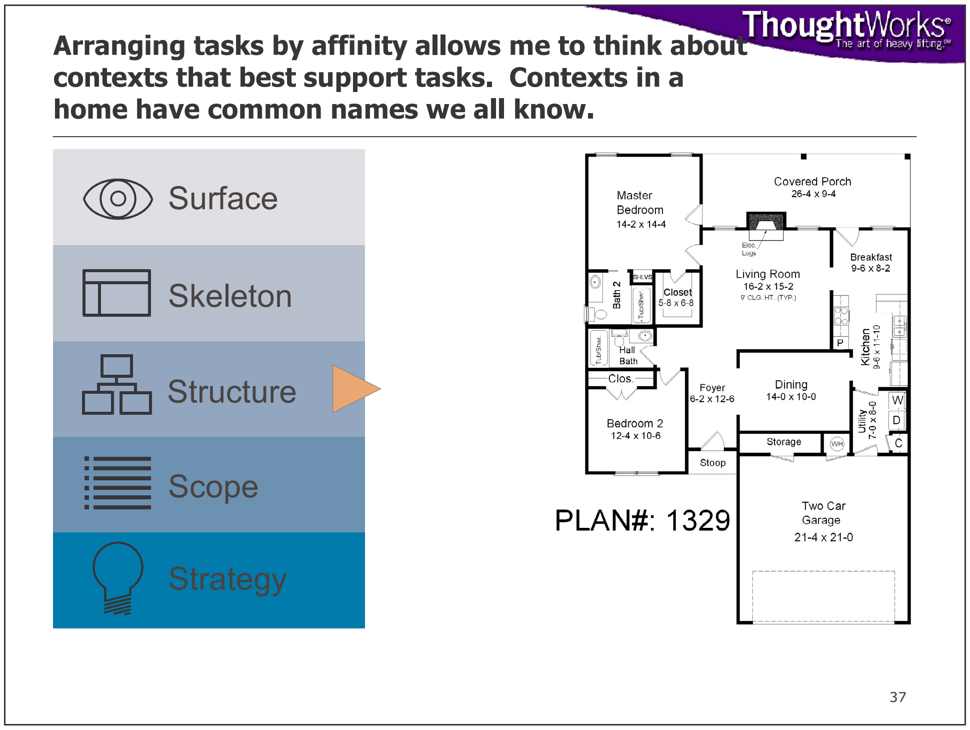


## What might I do to reach my goals?

- Surface
- Skeleton
- Structure
- Scope
- Strategy

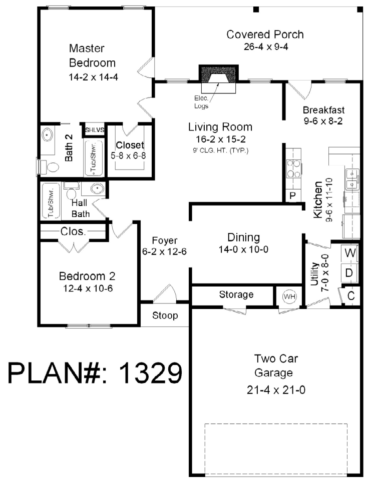
**user tasks:**
- store food
- prepare food
- eat food
- sleep
- bathe
- store changes of clothing
- stay out of rain
- entertain guests
- entertain self
- …

36



## Arranging tasks by affinity allows me to think about contexts that best support tasks. Contexts in a home have common names we all know.

- Surface
- Skeleton
- Structure
- Scope
- Strategy



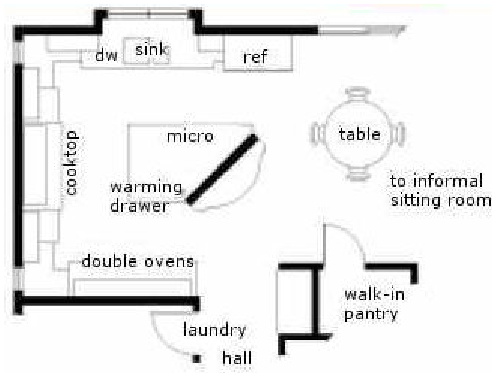
37

Notes

ThoughtWorks® The art of heavy lifting.℠

## When designing a particular interaction context – a kitchen for instance – I optimize layout and tool choices to support tasks I'll do there.

- Surface
- Skeleton ▶
- Structure
- Scope
- Strategy



38

ThoughtWorks® The art of heavy lifting.℠

## I'm going to spend a lot of time here, I want my experience to be as pleasant as possible…

- Surface ▶
- Skeleton
- Structure
- Scope
- Strategy

39

## Product Requirements and Design at 10,000 Feet

### Requirements Are Designed

To quote Alistair Cockburn from an aside conversation: ”A requirement is a relationship to a decision: If you get to make or change the decision, it’s design to you; if you don’t get to make or change that decision, it’s a requirement to you.”

As we move from very high level business goals for a product or project through to the details of what that product will specifically look like and do, we have thousands of individual decisions to make. All of those decisions are tested or evaluated against what we know about the problem we’re solving. It could be a high level problem like “what sort of features would appeal to stock portfolio managers in our new portfolio management software?” or a low level problem like “should the edit feature be accessed with a right click from the mouse, a button, or a menu choice?” All of these decisions are made based on what we know at the time. All of these decisions are made by a variety of different people at a variety of different times.

Once written down and agreed upon, all of these decisions can be thought of as requirements. The process we go through to determine what facts are important to consider, and the subsequent process of making that decision is what I’ll refer to here as design. It’s a difficult and creative process that demands a variety of skills and techniques to do well.

### Requirements are Captured

Once design decisions are made, they can indeed be written down, or captured. But, I’ll assert here that the hardest part of this process isn’t the capturing, but the deciding, the designing. The general approach we’ll leverage here is for design is User Centered Design (UCD).

An important factor of User Centered Design is the capture and documentation of not only the decisions made, but the factors used in making those decisions. For example a UCD practitioner might capture what is known about a type of user being served in a user profile. While that user profile doesn’t necessarily describe any feature of the product, it does describe the important factors about users used to make decisions about what features are important in the product. (In chapter 3 we’ll discuss a variety of models to capture what we know about users including user profiles.)

<illustration: simple user profile>

The captured part of requirements are the decisions made and the important factors considered to make those decisions.

### Traditional Requirements Processes Only Deal with Half the Story

In his 1987 essay “No Silver Bullet” Fred Brooks said:

> “The hardest single part of building a software system is deciding precisely what to build.”

Traditional requirements engineering, if there is such a thing, focuses on the activities of elicitation, analysis, specification, and validation, then on managing the resulting requirements as they change throughout the product’s development. Terms like *elicitation* imply that people already know the requirements; we need only draw them out. Terms like *analysis* seem to imply a methodical process of the breaking down of large things into their simpler parts. Neither term adequately describes the process of creatively inventing a number of possible solutions then deciding precisely what to build from among those solutions. Traditional requirements approaches, while useful at giving us notations for expressing requirements and mechanisms for classifying requirements, are light on approaches to help us invent, decide, and design.

The iterative and incremental nature of Agile development approaches distributes the event of deciding throughout the development lifecycle and out to more team members than other more traditional approaches. This makes understanding how we decide and design requirements even more critical. For this reason User Centered Design is used as foundation for a number of the design techniques discussed here.

## The User Centered Design Process Isn't

Just as it's important to understand that Agile Software Development describes a class of methodologies and not a specific one, User Centered Design describes a class of design approaches – not a specific design approach.

The term User Centered System Design was first used in Norman & Draper's 1986 book of the same name. The book brought together a collection of papers on human computer interaction design and emphasized the common theme of leveraging users to create the most effective software design. Dropping the "System" from the title left us with User Centered Design and that name stuck.

## The Missing Manifesto

The term "Agile Software Development" was invented fairly recently (2000) and is backed by the brief but useful manifesto discussed earlier that gives everyone describing and practicing Agile development approaches a common understanding to leverage. User Centered Design doesn't benefit from the same sort of manifesto. There seems to be a common abstract understanding of what UCD means, but individual practitioners might resolve the specifics differently.

In the absence of a specific manifesto, UCD approaches can be characterized as:

> Design approaches that drive decisions about products and product features from research and understanding about the actual or prospective users of the product, their goals, and an understanding of how they'll use the product to meet their goals.

Various specific UCD approaches have arisen over the years; Carrol & Rosson's Scenario-Driven Design, Holzblatt & Beyer's Contextual Design, Constantine & Lockwood's Usage-Centered Design, Cooper's Goal Directed Design, and many others. With an emphasis placed on specific design approaches described by each of these authors, it's made it hard to arrive at a clear understanding of what the common ground is for User Centered Design approaches.

## Jesse James' Generalization

In The Elements of User Experience (2004), Jesse James Garret deviates from other author's tacks of giving a specific approach to User Centered Design and describes a useful model for looking at the general activities done in UCD. He describes a model divided into five planes where each plane represents a scope of design concern. For each plane general types of activities are suggested along with some suggestions on when activities in one plane might occur relative to activities in another plane.

Let’s take a closer look at the planes of this model, shown in figure 2.1.

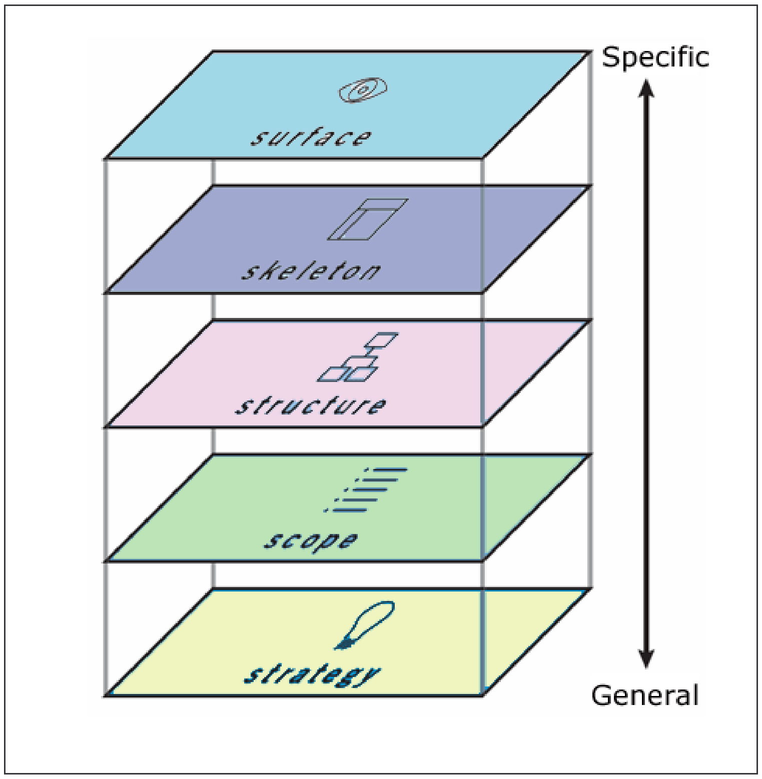


Figure 2.1 Jesse James Garrett’s Elements of User Experience

## The Surface Plane

As its name suggests the surface plane describes the surface of the software, its specific user interface.  This means everything we see and click on in the user interface, its color, exact placement, the fonts and size of text used, etc.  We’d use visual design techniques to describe the surface of our software.  For example a well designed ecommerce website page would be pleasant to look at and make it easy to see the items I’m interested in and the information about them that helps me make a buying decision.

## The Skeleton Plane

The skeleton lies just below the surface.  The skeleton describes roughly the layout of the software – the placement of buttons, menu bars, form elements, lists of information etc..  A skeleton might be visually represented as a wire-frame user interface drawing.  An interaction designer might build this wire-frame by arranging the elements in the user interface to optimally support the tasks that will be executed by the applications users.  A page for eCommerce website would likely place the item or items I’m looking for at the center of the page, and provide on each page a way to view elements in a shopping cart, or search for other items.  The skeleton describes the consistent useful placement of all these page elements.

## The Structure Plane

The skeletal user interface exists in some overall structure.  While the skeleton would describe placement of elements in the screen, the structure would describe navigation from one screen to another.  An interaction designer or information architect might create structure to arrange functions and information in the software to best support the activities the users will be engaged in.  For example moving smoothly from shopping to checkout in our ecommerce site is the responsibility of good structural design.

## The Scope Plane

While the structure might describe how we navigate from feature to feature, on the scope plane we describe what those features might be and how these features fit together.  In our ecommerce website we know that we need to show items and allow their purchase.  We might also decide it's important to save previous customer addresses we've shipped orders to.  These are scope decisions.

## The Strategy Plane

In the strategy plane we'll look at the goals for the organization paying for the software to be built.  We'll then look at the prospective users of the software and their goals when using the software.  Having a good understanding of all these goals allows us to determine what features are in scope.  For our ecommerce site we might come to understand that we're selling an item that is a one-time purchase.  We expect users will spend time making a buying decision then, upon deciding, quickly purchase the item.  Since we don't expect them to come back for another, saving a previous customer addresses may in fact not be a feature we'll need.

## Move from General to Specific

The model divides concerns into five planes from general to specific.  Under that is the implication that we'll resolve general stuff before we resolve specific stuff because the specific stuff is based on the general stuff.  Strategic choices inform scope.  Scope choices inform software structure.  Structure choices inform each screen or page's skeleton.  The skeleton provides the foundation for the visual design.  At each plane, specific User Centered Design approaches might offer specific thoughts and techniques for dealing with that plane's concerns.

After introducing the model Garret appropriately advises that the activities and decision made for each plane not be resolved before the next plane's activities begin.   In practice work will be going on in several different planes simultaneously.  Garret does advise not finishing work on one plane before work done in its preceding plane's work is complete.  For example fixing scope on the software before strategic goals for the software were understood would be a bad idea.  Or fixing the navigation structure without knowledge of the features the software will have might yield bad results.

## General to Specific Planes and Snow to Eskimos

Moving from general to specific is a common approach used in traditional requirements approaches.  The three layers of requirements commonly referred to by requirements engineers are:

- Business Requirements,
- User Requirements, and
- System Requirements

We can see that Business Requirements maps pretty cleanly to Garrett's Strategy plane.  The Scope plane maps to what we might call User Requirements.  But when we reach System Requirements, Garrett uses three Planes; Structure, Skeleton, and Surface to divide up design decisions around the physical appearance and behavior of software.

There’s a linguistic myth that Eskimos have hundreds of words for snow [ref Pullmans book “The Great Eskimo Vocabulary Hoax”] because they are around it enough and concerned with it enough to have different concepts and names for different varieties of snow. Although the old myth may or may not be true, the concept here is what’s important. If your primary focus is on a particular area, you’ll richen up the vocabulary you use to discuss that area.

Garrett’s model is on user experience, so it’s understandable there are more planes in his model in the area of describing the system than in a traditional requirements model. In fact your organization may use some sort of progression for moving through its requirements. Take note of how many layers it has and in what particular areas. You may find the areas where there are more layers point to the areas of biggest concern.

The important concept here is that that when deciding what specifically to build, there are dependent layers of decisions that move from more general to more specific.

## UCD Specialists

I emphasize User Centered Design techniques as a foundation for designing requirements. This doesn’t imply that you need to delegate this work to a UCD specialist, rather that it is important to understand the nature of these decisions and that everyone internalize user centered design thinking into the way they think about requirements.

That said, bringing in trained UCD specialists can be incredibly valuable. It’s important to understand that like doctors, UCD people do tend to specialize. I’ll generalize UCD people into three common categories, Interaction Designers, Visual Designers, and Usability Engineers. While all of these specialist practitioners are concerned with the entire process of design, they each have specific emphasis and expertise.

> **Interaction Designers** tend to focus on the “what to build” issues. You’ll find them with more to offer around the Strategy, Scope, and Structure layers of the design. Don’t expect them to be the best at building detailed user interface prototypes or leading extensive usability lab tests. Do expect them to conduct research on users and their needs and make strong substantiated recommendations on what software might best address those needs.
>
> **Visual Designers** tend to focus on the “how it looks and feels” issues. You’ll find them with more to offer on the Structure, Skeleton, and Surface layers of the design. Don’t expect them to lead user research efforts or to conduct rigorous usability tests. Do expect them to translate functional requirement decisions into detailed and effective user interface design suggestions.
>
> **Usability Engineers** tend to focus on the “is the software working effectively for users” issues. You’ll find them most effective at validation of design at various stages of completeness. Don’t expect them to lead user research or develop beautiful user interface prototypes. Do expect them to lead usability tests to validate that prototyped or built software works effectively for the target user constituencies. Do expect them to make detailed recommendations for improvement to the software based on these tests.

It's neither accurate nor fair to generalize all UCD people into these categories, but that hasn't seemed to stop me.  When encountering a UCD specialist you might find that they contain one or all of these three emphases.  However, I tend to find that one are of concern is dominant in their expertise and approach.

## Plan is a Synonym for Design

At this point you might be starting to see some similarities between Garrett's model and the Agile Development Model.  When looking at the Agile development lifecycle we see lots of plans made at varying degrees of detail.  The closer we get to actually building features, the more specific the plans get.  Similarly, in Garrett's model, the closer we get to the surface plane the more specific the information gets.  The various levels of Agile planning start to mirror the various levels of design concern in Garrett's model.  If we understand plan to be a synonym for design, then there might be something going on here.

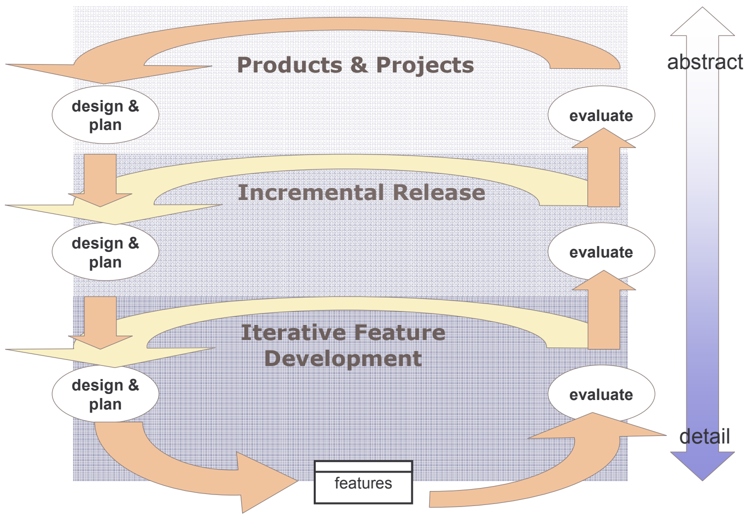
Notes

## The Agile Concept of “Test First” Isn’t About Testing, It’s About Designing

- Test First Development refers to the practice developers engage in where they automate unit tests before writing code that allows those tests to pass
  - Writing these tests first forces developers to think about how the code will be used and what it must do prior to writing it
  - Agile developers often say: “How do I know I’ve written the correct code if I don’t have a running test to prove it?”
- Models built by user centered design practitioners perform the same role as developer tests
  - Business goals help validate our choices of user constituencies to support
  - User models help validate the work practice we choose to support
  - Work practice models help validate the features we choose to design and implement
  - How could we know we’ve chosen the correct features without business goals, user models, and work practice models?

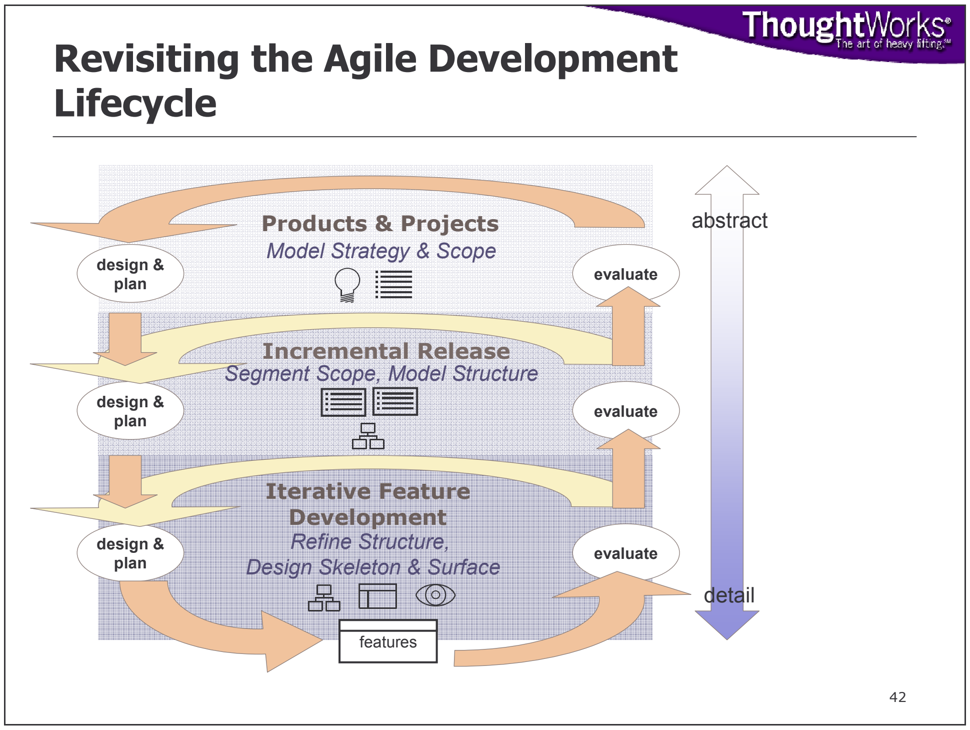
40

## Merging Ux Design Dependencies With an Agile Development Lifecycle



41

Notes



## Revisiting the Agile Development Lifecycle

## Common Concepts In Agile Development and User Centered Design

Looking at both these models, a couple common concepts emerge.

### Moving from Abstract to Detail

Both the Agile and Elements models move from a high level of abstraction to a lower or more detailed level of abstraction.

Using the Agile Development model I'm ultimately concerned with developing features, but I get there by planning iterations, and I get there by planning releases, and I get there by chartering a project.

Using Garrett's User Experience model, I'm ultimately concerned with describing an effective user interface, but I get there by understanding page structure, and I get there by understanding navigation structure, and I get there by understanding the scope of tasks and features supported, and I ultimately get there by understanding the goals of the organization building the product and the people who will use it.

Both models give useful names to these layers of abstraction: feature, iteration, release, project; and surface, skeleton, structure, scope, strategy. The names and the concerns they represent help us constrain our thinking to what matters at the particular level of detail we're concerned with at the moment.

### Decision Dependencies

It may be immediately obvious that there are dependencies between these named levels of abstraction. The decisions made at a high level of abstraction directly affect the options I have and the decisions I make at the next level of abstraction.

In the Agile Development model, the features I choose for a product are a function of the goals for the product. The features in a particular iteration are chosen from the features chosen for a release.

In the User Experience Model, the navigation of the user interface of the software is a function of the features chosen for the software. The general layout, or skeleton, of a particular page of the software is a function of the features that page must support and the navigation structure of the software. The specific visual design of the page is a function of the skeleton of that page.

### Evaluation & Decision Dependency

When working with software, there's a tendency to progress directly to the lowest, most concrete level of abstraction – the specific user interface for the specific feature we'd like to start building now. That's not by itself a bad thing. But, it's critical that when we make a decision on a more detailed plane, we must validate that decision with those made on the higher level of abstraction planes. That process might go a bit like this:

- Here's the way I think this screen should look for the software.
- What features or activities does this user interface support?
- Does this user interface support this feature or user activity well?
- Is this feature or activity important to our target users of the application?
- Are both these target users and these features or activities the ones we want to support to earn the best return on our software development investment?

It’s easy to see that if we don’t have information about our financial goals, the users we intend to support, and their activities that even a simple level of validation is difficult. Without that information, it’s impossible to say if a particular design decision is indeed good or bad. Yet, on most software projects individuals are often expected to make such evaluations on the fly without a good understanding of the decisions made at higher levels of abstraction. How do they manage that? They guess. OK, I guess that they guess. I know I do.

Guessing isn’t entirely bad. In fact I’m a pretty good guesser. The difficult part comes when I forget later why I made a particular guess. Or when someone else working alongside me makes different guesses. This is often how design arguments occur. The next time you observe a couple team members arguing a design decision, instead of discussing the merits of one decision over another, try moving up an abstraction level and discussing their reasons for making that decision. You’ll observe very quickly the dependency between these abstract levels of understanding and decisions we make at each.

## Differences

In spite of their similarities, these two models emphasize different things.

The Agile Development model emphasizes the cyclic nature of development and the dependencies higher level cycles have on the lower level cycles they’re composed of.

The User Experience Model emphasizes the dependency of information acquired and decisions made at higher levels of abstraction on the information acquired and decisions made at lower levels of abstraction.

The Agile model is about process.

The User Experience model is about design decisions.

Since we can’t engage in a process of building software without making some decisions about specifically what to build, we’ll need to contend with both models.

## Waterfall Head

There’s a misconception that we might use UCD techniques or other requirements elicitation techniques on Agile projects by first doing requirements stuff, then engaging the Agile software development to do what it does. That is; we’ll use the user experience model first to create our design, then slice it into features to build using an Agile incremental development process.

There’s a misconception that we that we might decide initially what features to include in software, build the software roughly, then have a user interface designer “make it pretty” as the product nears completion. That is; do the scope setting part of the user experience design work first, slice it into features to build using Agile incremental development processes, then have developers guess at best structure, skeleton, and surface design with the intent of correcting those decisions later.

Both of these strategies, while they may work in many contexts, are symptoms of “waterfall” thinking. By waterfall I’m referring to the general approach referred to as waterfall development where all work of a particular type is done in its own phase: requirements, then design, then development, then test. Royce [ref], the person first describing the waterfall approach, never intended for the phases to be isolated – for the decisions made in a design phase to not be informed by information learned in a development phase for instance.

In an Agile context where we have the explicit goal to defer design decisions and then learn by evaluating our work in progress, both strategies are especially risky.

Fixing the scope early in a project strips out much of the purpose and power of evaluation at the end of feature development, iterative development, and incremental release.

Deferring user interface decisions till late strips away the ability for actual users to evaluate the suitability and the quality of the software for their day-to-day use. Not being able to place software in front of actually users for their evaluation strips away an important layer of feedback, and an important opportunity for change and adaptation.

To really leverage these two models and their ideas well, lopping them into pieces and sequencing them waterfall style isn't going to work. We'll need to weave them together into something better. Something that allows us to constantly make effective design and requirements decisions based on the continuous evaluation found in the Agile lifecycle.

## Weaving the Requirements and Design Process into Agile Software Development

If we lay these two models side by side, we can start to get a little guidance on how we might begin to weave these two lines of thought together.

Recalling the cyclic model for Agile Software Development, planning and design happens inside of every cycle. Design decision making happens continuously, not all at once at the beginning. It's threaded throughout the lifecycle. Removing the design and plan thread from the Agile lifecycle would unravel it. The challenge now becomes what sorts of design decisions to make, and when. If we lay the Agile Development model side by side with Garrett's Elements Model, we'll begin to get some direction on specifically what and when.

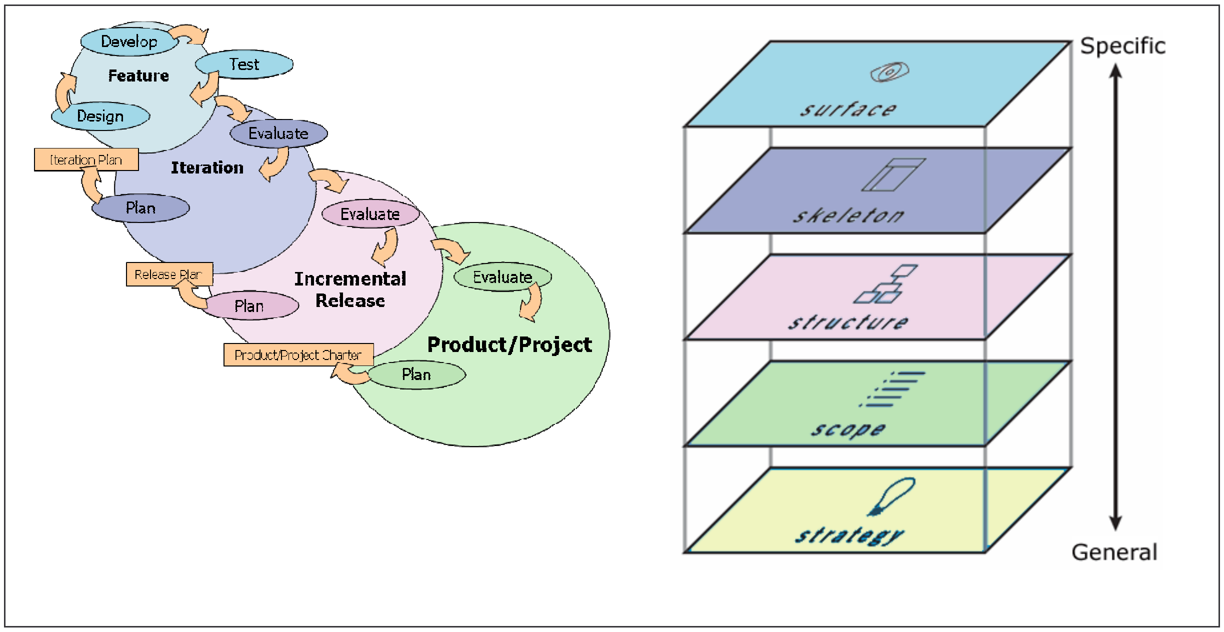


Figure 3.1 – Side by side Agile and UCD Models

## What to Design and Plan, When

### For the Product and Project

We'll need some sort of project charter to start.  The Element's models Strategy and Scope planes help inform us a bit here.  Understanding strategy will help us work through how the software can earn some return for its builders, what user constituencies it should support to do so, and what sorts of features it should have to appeal to and be useful for those user constituencies.

Project chartering is about Strategy and Scope.

### For the Incremental Release

We'll need some sort of release plan that lets us understand what software we'll be building over time and some rough idea of the effort it will take to build it.  A rough idea of scope is important here, especially the most important scope to meet the strategy goals of the project.  To give rough estimates on development time, it may be important to understand the shape of the software a bit – how many screens might it have, roughly what sorts of activities will each screen support.  Understand that Structure may help us guess at complexity and estimate effort to build.

Incremental release planning is about Scope and Structure.

### For Iterative Feature Development

To plan iterations and deliver more precise estimates, well need a better understanding of the specific user interface of the software, more specifically how it looks and behaves in use. We'll need details about the Structure, Skeleton, and Surface of our software.

Iterative feature development is about Structure, Skeleton, and Surface.

### Evaluating Up

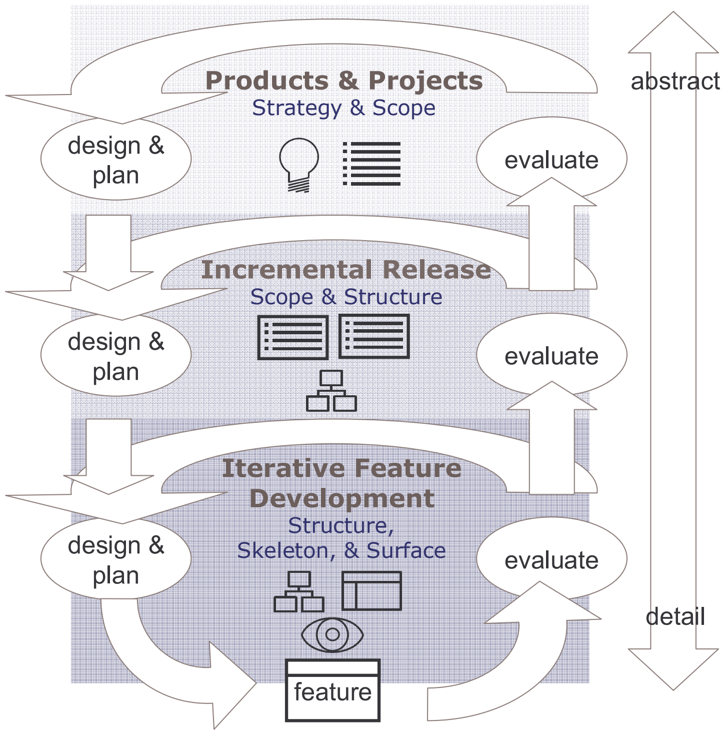
As we evaluate finished features we need to be concerned that the finished software behaves as we'd predicted it would when we initially designed it.  We'll look closely at the Structure, Skeleton, and Surface to make changes to those based on our evaluation of these features.

As we evaluate finished iterations of features, we'll begin to see a better representation of how the features work together.  We'll look more closely at the Structure and Features and make changes to those based on the evaluation of the group of features.

As we evaluate finished releases, we'll better see how large groups of features coordinate to support specific users.  We can determine if we're supporting users well enough to earn the return we expected on our software.  We'll look closely at the Scope and Strategy and make changes to those based on the evaluation of the release.

### Merging the Models into an Agile Requirements Model

If we flip and merge these two models we might end up with something a bit like this.



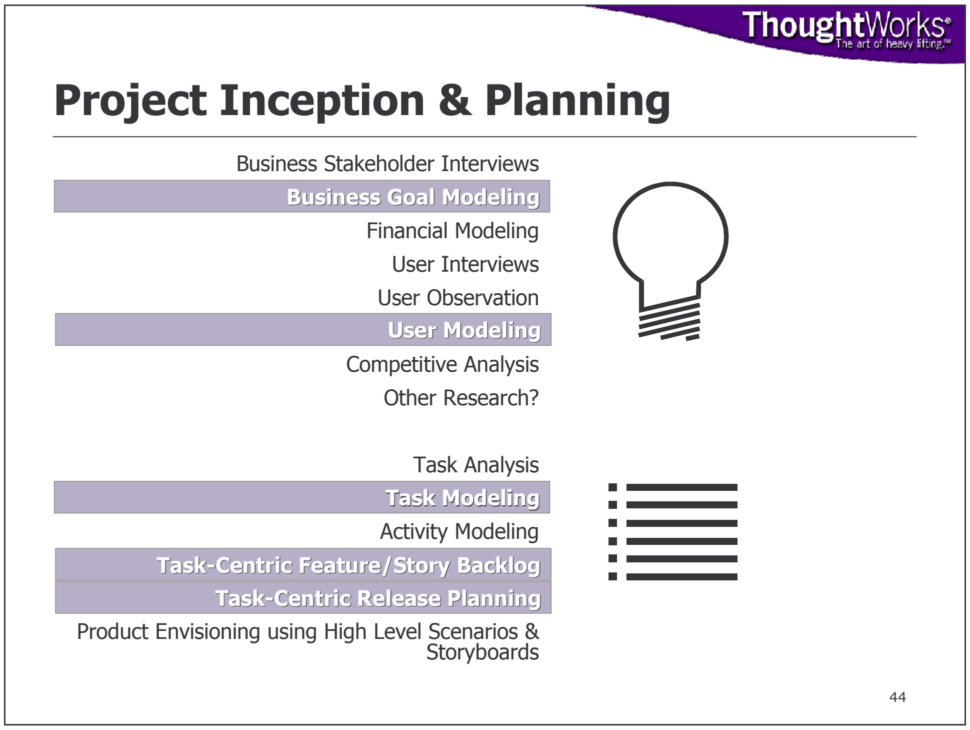
## The Planes

Planes in this Agile Requirements Model correspond with both an Agile Development Lifecycle, and an element of design and requirements concern. The levels are arranged from abstract to detailed.

## Arrows Point the Way:

Think of this chart as a simple board game. Start your token in the upper left circle to design and plan for a project. Notice that you can move down the planning and design stack until you reach the feature level, then you can start to move up the evaluation stack. From any circle in the evaluation stack, you can circle back to design and plan in that same plane. The circles and arrows give us an idea of the dependencies involved when designing and documenting requirements and planning our Agile project. While there are always exceptions, focusing on the design, planning, and evaluation concerns at the right abstraction level works pretty well.

Notes



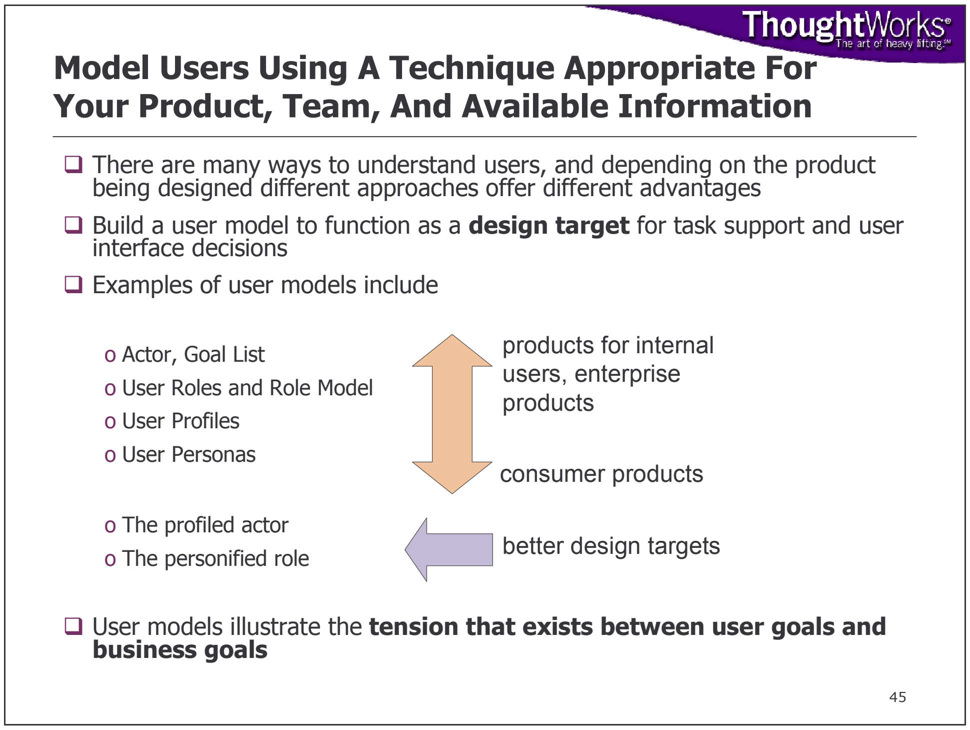
ThoughtWorks® The art of heavy lifting.℠

## Project Inception & Planning

Business Stakeholder Interviews
**Business Goal Modeling**
Financial Modeling
User Interviews
User Observation
**User Modeling**
Competitive Analysis
Other Research?

Task Analysis
**Task Modeling**
Activity Modeling
**Task-Centric Feature/Story Backlog**
**Task-Centric Release Planning**
Product Envisioning using High Level Scenarios & Storyboards

44



ThoughtWorks® The art of heavy lifting.℠

## Model Users Using A Technique Appropriate For Your Product, Team, And Available Information

- There are many ways to understand users, and depending on the product being designed different approaches offer different advantages
- Build a user model to function as a **design target** for task support and user interface decisions
- Examples of user models include
  - Actor, Goal List
  - User Roles and Role Model
  - User Profiles
  - User Personas

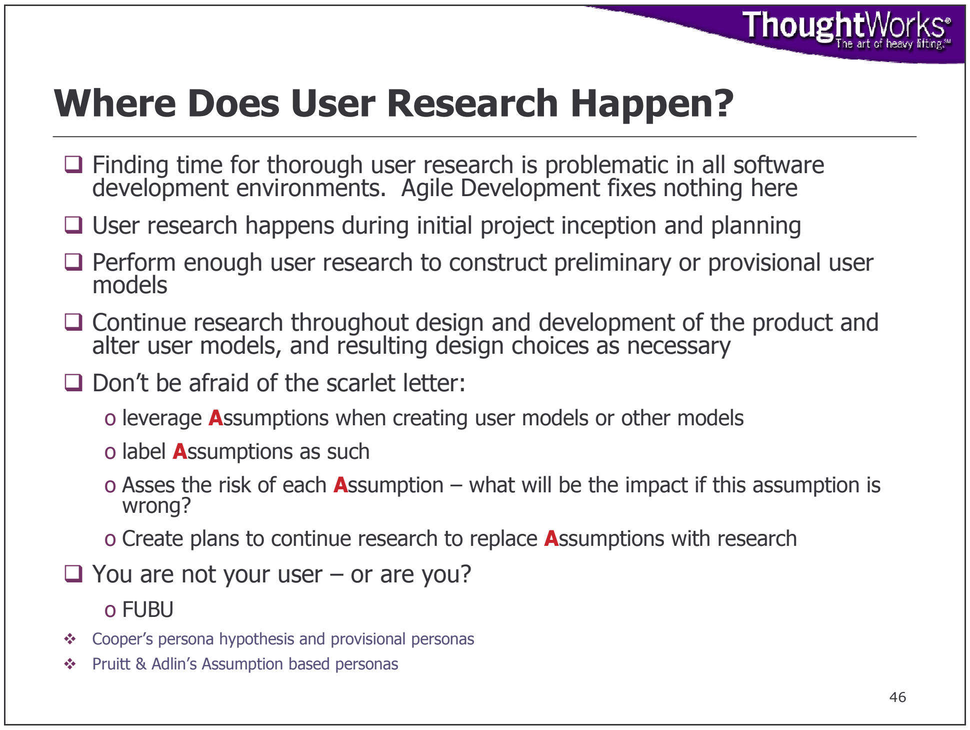
  products for internal users, enterprise products

  consumer products

  - The profiled actor
  - The personified role

  better design targets
- User models illustrate the **tension that exists between user goals and business goals**

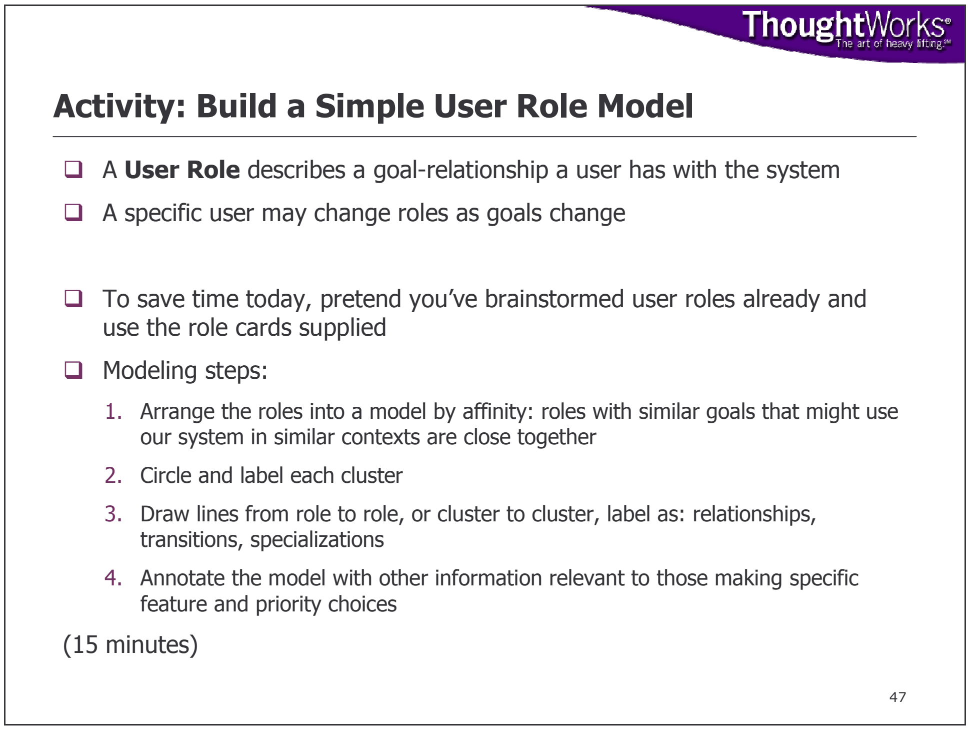
45



## Where Does User Research Happen?

- Finding time for thorough user research is problematic in all software development environments. Agile Development fixes nothing here
- User research happens during initial project inception and planning
- Perform enough user research to construct preliminary or provisional user models
- Continue research throughout design and development of the product and alter user models, and resulting design choices as necessary
- Don't be afraid of the scarlet letter:
  - leverage **A**ssumptions when creating user models or other models
  - label **A**ssumptions as such
  - Asses the risk of each **A**ssumption – what will be the impact if this assumption is wrong?
  - Create plans to continue research to replace **A**ssumptions with research
- You are not your user – or are you?
  - FUBU
- Cooper's persona hypothesis and provisional personas
- Pruitt & Adlin's Assumption based personas

46



## Activity: Build a Simple User Role Model

- A **User Role** describes a goal-relationship a user has with the system
- A specific user may change roles as goals change
- To save time today, pretend you've brainstormed user roles already and use the role cards supplied
- Modeling steps:
  1. Arrange the roles into a model by affinity: roles with similar goals that might use our system in similar contexts are close together
  2. Circle and label each cluster
  3. Draw lines from role to role, or cluster to cluster, label as: relationships, transitions, specializations
  4. Annotate the model with other information relevant to those making specific feature and priority choices

(15 minutes)

47

Notes

**Additional Reading:**

- Norman, **Why doing user observation first is wrong** (http://www.jnd.org/dn.mss/why_doing_user_obser.html)

## A User Role is a Type of User in Pursuit of a Goal

The software we're building will have lots of features that support lots of user activities.

To choose the most appropriate activities we must know who will be using our software and what their goals are.

Through research or anecdotal discussion we may have some ideas about the demographics of our users, their ages, computer skills, likes, and dislikes. These details are important.

We may also have a strong sense of what people want to do with our software. Marketing may furnish ideas for features. Customer support may furnish lists of common problems. Observation of users might give us ideas about why they might use our software. Interviews and collaboration with users will result in lists of goals and concerns.

Users goals frequently change while they're using our software. For example someone looking at an eCommerce website might start with a goal of getting product information then change their goal to purchasing a product at the best price.

The demographics that matter most are the ones that would affect the design of our software. The user goals that matter most are those user goals that our users might use our software to meet.

Having a concise set of users and goals will help us identify the most appropriate product features to help our users meet their goals.

Use a user role to describe a particular type of user in pursuit of a goal. Build a concise list of user roles to describe the type of users and goals the software you're building will address.

When choosing user role names, I prefer the form "thing-doer." For example a person looking at an information kiosk with the goal of finding a specific used CD might be a "Used CD Finder."

To add color and context, "decorate" user roles with adjectives. For instance some shoppers might be in a hurry. Their goal is to find the item they're looking for very quickly then exit the store. This kind of person might be a "Hurried CD Finder", or a "Harried Hurried CD Finder. (I guess it's best not to go too crazy.)

The names you give user roles are very important. The more concise and meaningful the user role names, the less supporting documentation you'll need about them.

Choose names that describe the users goal with respect to the software you're describing. For example, in our kiosk system it might be inappropriate to describe a user as a CD-buyer. While it's true their ultimate goal may be to buy a CD, they're using our system to find it in the store. They'll then physically locate it and carry the CD to a cashier to buy. Referring to them as a CD-Finder more accurately describes their goal for using the kiosk.

Think of roles as hats an individual might change. A user might start in one role, change to another as their goals or context changes. This distinguishes a user role from a user profile or persona that more elaborately describes a user and their goals. A software system that can be described with 3-5 personas might likely require 10-20 or more user roles.

When creating user roles, do add additional detail to the role such as demographic information that has bearing on the design, usage context information, frequency that a user in this role might use our software, and roles they might likely be changing from or changing to. A pretty well documented role should fit on a 4" x 6" card, or a half sheet of paper.

Larry Constantine & Lucy Lockwood describe user roles in Software for Use.

### Additional Reading

- Constantine & Lockwood, **Software for Use** (Addison-Wesley, 1999)

## Building a User Role Model

If your system is reasonably complex you'll have a number of good user role candidates, enough that it may be hard to see common themes or interdependencies amount them.

It's important to understand how similar user roles relate to each other and how user roles may be dependent on each other.

Discussion about user roles is always valuable, but discussion alone may leave some participants with a different understanding than others. A common visible artifact helps everyone see what they understand.

It's common to have several roles that share very similar goals but vary in terms of their skills or usage context. It's important we understand those shared goals.

It's common for a user in one role to transition to another role. It's also common for a user in one role to do work that another role depends on. It's important to understand these role transitions and dependencies.

When you have a large landscape of interdependent user roles, downstream prioritization of features for these roles can become difficult. Calling out most critical or focal roles to support with the software is a valuable aid to priority setting.

A user role model arranges role cards spatially to cluster roles with common goals, uses cluster arrangement and markup to show role transition and dependence, and clearly calls out the most critical roles as focal roles.

Building a user role model using flip chart paper and role names written on 3x5 cards is a fast, simple, collaborative way to make sense of a large number of user roles. Start this process with 3x5 cards, flip chart paper, markers, tape, post-it notes, and individually wrapped pieces of candy.

The following text extracted from the Better Software article "An Elephant in the Room" describes a collaborative role modeling process. The picks up after a user role cardstorming.

Now that you've got the properly named roles, discuss each with the intent of agreeing on the goals for each role. The name of the role may get you most of the way there. But, make sure the role specifically states what this user's goals are.

Be careful, the goals may not be what you expect. While management may believe that a call center employee may have the goals of quickly and effectively answering as many calls as possible, the person doing that job may more accurately have the goals to get through the workday without incident, to look good to their boss, and to not have the software they use make them feel stupid.

Write the goal or goals for each role in short statements on the 3x5 cards.

**What do these people have to do with each other?** Now we'll start building a model that helps show how user roles relate to each other. Grab a sheet of poster paper and lay it on a large flat surface. Take up the role cards and start placing them on the poster paper in a card arrangement that "feels" right. Don't think to hard about it. Do try to cluster the cards a bit. Do this by placing role cards similar to each other closer together or overlapping. Place role cards dissimilar to each other far apart. The group should discuss the arrangement till they decide it feels correct.

Using clear tape, fix the cards to the sheet of poster paper. This is the start of a good role model.

**Where did we put those pieces of candy?** Each participant should grab a couple pieces. Look at the arrangement of roles, and the goals of each. Of these roles, who is it most important to satisfy for this software to be successful? Alternatively ask, which of these roles will we be in big trouble with if we don't satisfy them? Everyone choose a couple roles, and vote for them by placing a piece of candy on that important role. You'll find, as you place your candy votes, that not everyone agrees. That's always good information to understand. Remove each piece of candy, and write a letter "F" prominently on the bottom left corner of the card that the candy came from. Some cards my have 2, 3 or more "F"s. These are our *focal* roles – the ones we need to focus on. The ones with more "F"s are higher priority. Now eat the candy.

**It's time for some creative annotation.** Look more closely at the clusters. Why did they cluster? What makes them similar? Use a pen to draw a circle around the cluster and label it with what makes these roles similar. While doing this we often end up with labels like "back office accounting," "customers," or "process oversight people." Is there a common goal that all these roles share? Label them with whatever makes sense to the team.

Now look at all the clusters. Do they relate to each other? Wherever possible draw a line from one cluster to another and label that line. We often end up with line labels like "sends work orders to," "supervises," or "tries to hide from."

Look at the X and Y axis your role cards are placed on. There's likely an unsaid reason that some cards are higher than other cards and some cards lower, some cards to the left and some to the right. I find it's common to place roles who have participation early in a process on the top left, those with participation late in the process on the bottom right. More senior people often end up at the top of the model. In that example the X axis might approximate time, the Y axis seniority. Look for evidence that your roles are arranged on axis lines, decide what they are, draw some lines on the model and label them.

Now it's time to add whatever other markings or notations to the model you want. Are there other bits of information about these users that are important? Think of their work environments. Are they hectic or peaceful? Think of their skill levels; are they highly skilled, or just trying to get by? Think of their computer experience. Are they real geeks, or do mice frighten them? Make notes on the model about these things and any other interesting details you can think of.

Lastly, give this model a title that indicates the software the role model pertains to. Something like "Widget Performance Analysis Role Model."

**This should now look like a kindergarten art project.** Congratulate yourselves. Optionally grab a blanket and a graham cracker and take a short nap. I'm only partly joking. When everyone is fully engaged in the process, a huge amount of information is being exchanged over the creation of this model. It can be pretty exhausting.

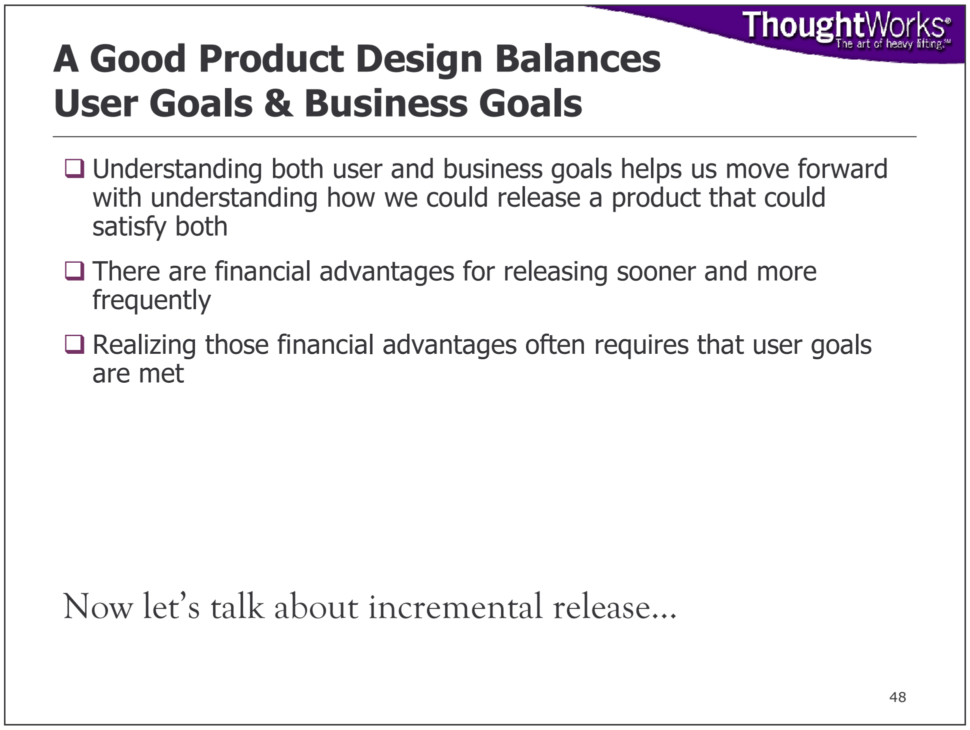
What you've built is a wildly creative variation of a user role model from Constantine & Lockwood's Usage-Centered Design. This form of role model has a few strong advantages, and couple disadvantages.

You'll find that when you look at the role model it will remind you of the conversations you had while creating it. It will be easier to recall the specific users you talked about and the reasons behind the goals you noted down. The notations you invented will continue to remind you of the relationships the roles have with each other. As time progresses knowing who focal roles are will help you make critical decisions about the software, and how to design and test it.

While there's a lot of information packed in this role model, it's very specific to those present during its creation. You'll find that when you want to explain information about the people using your software, you can't simply direct them to the role model. You'll have to stop and explain your kindergarten artwork to them. But, after some initial shock, they'll catch on.

## Additional Reading

- Constantine & Lockwood, **Software for Use** (Addison-Wesley, 1999)
- Patton, **An Elephant in the Room** (Better Software, http://www.abstractics.com/papers/ElephantInTheRoom.pdf)

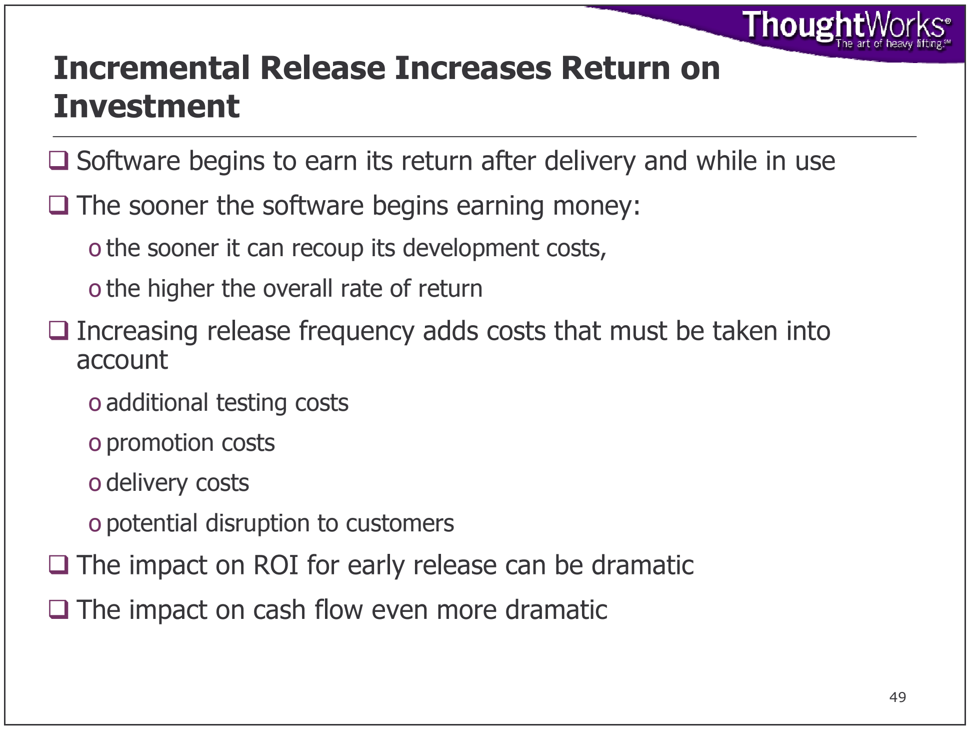


## A Good Product Design Balances User Goals & Business Goals

- Understanding both user and business goals helps us move forward with understanding how we could release a product that could satisfy both
- There are financial advantages for releasing sooner and more frequently
- Realizing those financial advantages often requires that user goals are met

Now let's talk about incremental release…

48



## Incremental Release Increases Return on Investment

- Software begins to earn its return after delivery and while in use
- The sooner the software begins earning money:
  - the sooner it can recoup its development costs,
  - the higher the overall rate of return
- Increasing release frequency adds costs that must be taken into account
  - additional testing costs
  - promotion costs
  - delivery costs
  - potential disruption to customers
- The impact on ROI for early release can be dramatic
- The impact on cash flow even more dramatic

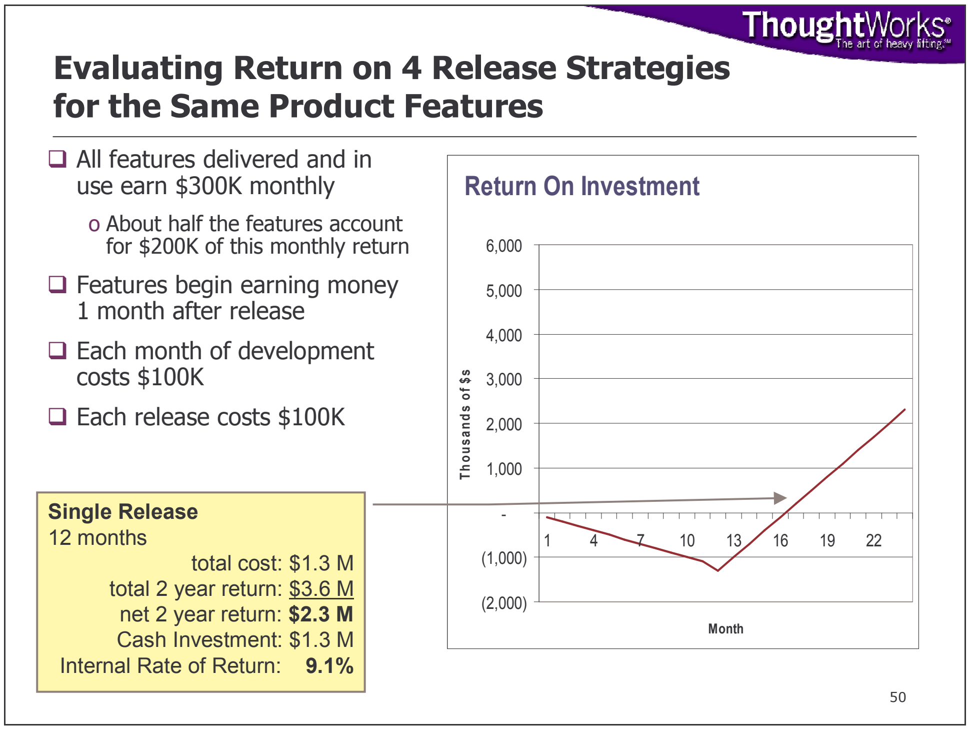
49

Notes

**Additional Reading:**

- Szuc & Gaffney, The constant design balance (http://www.apogeehk.com/articles/constant_design_balance.html

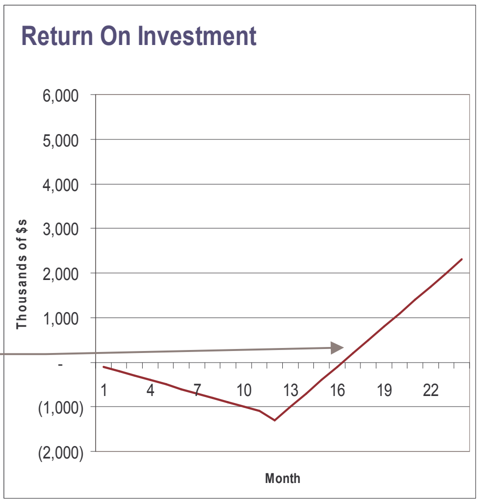
Notes



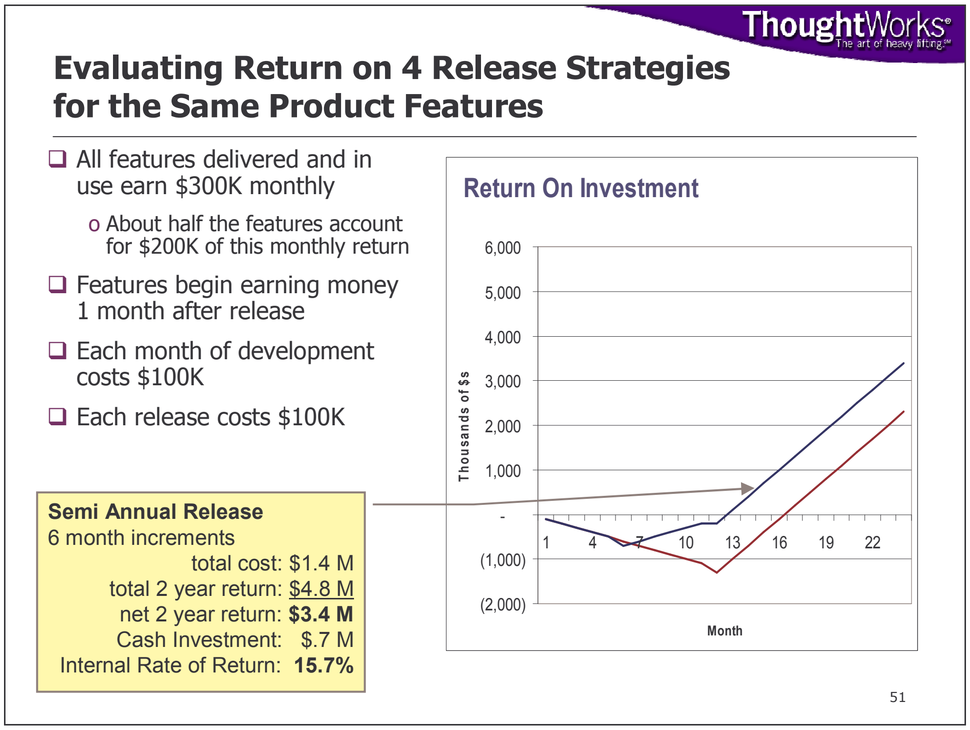
## Evaluating Return on 4 Release Strategies for the Same Product Features

- All features delivered and in use earn $300K monthly
  - About half the features account for $200K of this monthly return
- Features begin earning money 1 month after release
- Each month of development costs $100K
- Each release costs $100K

**Single Release**
12 months
total cost: $1.3 M
total 2 year return: $3.6 M
net 2 year return: **$2.3 M**
Cash Investment: $1.3 M
Internal Rate of Return: **9.1%**



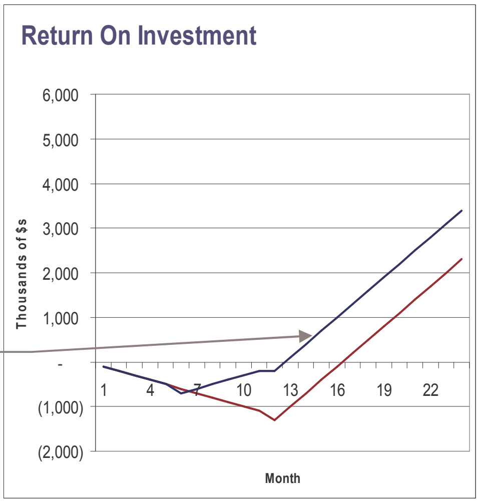
50



## Evaluating Return on 4 Release Strategies for the Same Product Features

- All features delivered and in use earn $300K monthly
  - About half the features account for $200K of this monthly return
- Features begin earning money 1 month after release
- Each month of development costs $100K
- Each release costs $100K

**Semi Annual Release**
6 month increments
total cost: $1.4 M
total 2 year return: $4.8 M
net 2 year return: **$3.4 M**
Cash Investment: $.7 M
Internal Rate of Return: **15.7%**



51

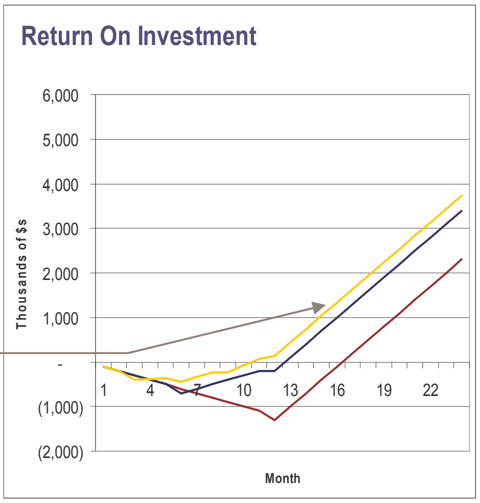
Notes

ThoughtWorks® The art of heavy lifting.℠

## Evaluating Return on 4 Release Strategies for the Same Product Features

- ❑ All features delivered and in use earn $300K monthly
  - o About half the features account for $200K of this monthly return
- ❑ Features begin earning money 1 month after release
- ❑ Each month of development costs $100K
- ❑ Each release costs $100K

**Quarterly Release**
3 month increments
total cost: $1.6 M
total 2 year return: $5.3 M
net 2 year return: **$3.7 M**
Cash Investment: $.44 M
Internal Rate of Return: **19.1%**



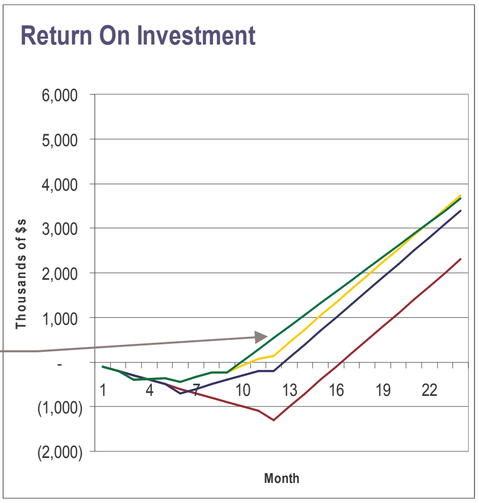
52

ThoughtWorks® The art of heavy lifting.℠

## Evaluating Return on 4 Release Strategies for the Same Product Features

- ❑ All features delivered and in use earn $300K monthly
  - o About half the features account for $200K of this monthly return
- ❑ Features begin earning money 1 month after release
- ❑ Each month of development costs $100K
- ❑ Each release costs $100K

**Quarterly Release – drop the last release**
3 month increments
total cost: $1.2 M
total 2 year return: $4.9 M
net 2 year return: **$3.7 M**
Cash Investment: $.44 M
Internal Rate of Return: **20.4%**



53

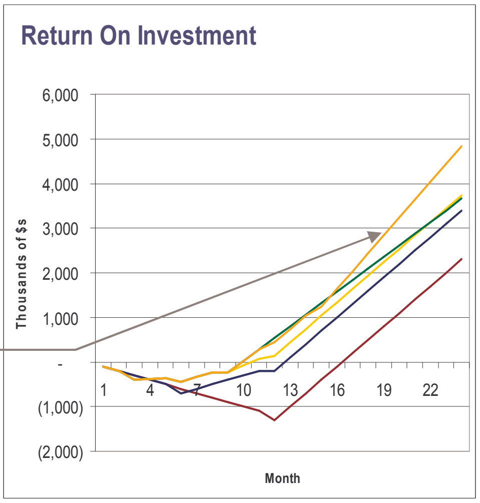
Notes

## Continuing To Add Features May Not Pay The Same Level Of Return

- ❑ Continue development for one additional quarter
- ❑ Additional high value features add $100K monthly increase to return

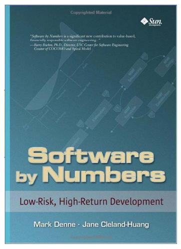
**Quarterly Release – continue with 5th release**
3 month increments

| | |
|---|---|
| total cost: | $2 M |
| total 2 year return: | $6.2 M |
| net 2 year return: | **$4.24 M** |
| Cash Investment: | $.44 M |
| Internal Rate of Return: | **19.0%** |



54

## Software By Numbers & Project Portfolios


Software by Numbers
Low-Risk, High-Return Development
Mark Denne · Jane Cleland-Huang

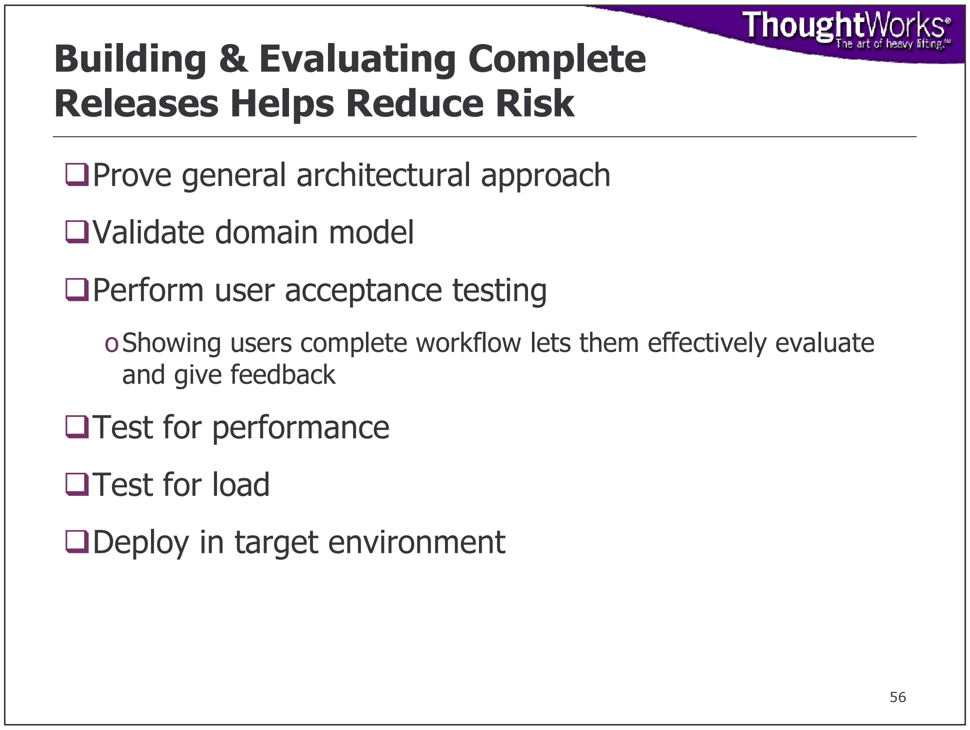
- ❑ Software by Numbers [Denne & Cleland-Huang] describes Incremental Funding Methodology [IFM]
  - o Goal to reduce necessary cash outlay
  - o Make projects self-funding
  - o Increase return on investment
- ❑ SBN Tools:
  - o http://dactyl.cti.depaul.edu/ifm/default.htm
- ❑ SBN introduces the concept of Minimal Marketable Feature – MMF - the smallest sized feature that would have marketable value
- ❑ SBN simple financial models provide guidance on evaluating multiple projects in a portfolio

55

**Additional Reading:**

- **Software by Numbers website:** http://dactyl.cti.depaul.edu/ifm/default.htm
- Cohn, **Project Economics presentation**: http://www.mountaingoatsoftware.com/pres/SDBP05_ProjectEconomics.pdf
- Tockey, **Return on Software: Maximizing the Return on Your Software Investment** (Addison-Wesley, 2005)

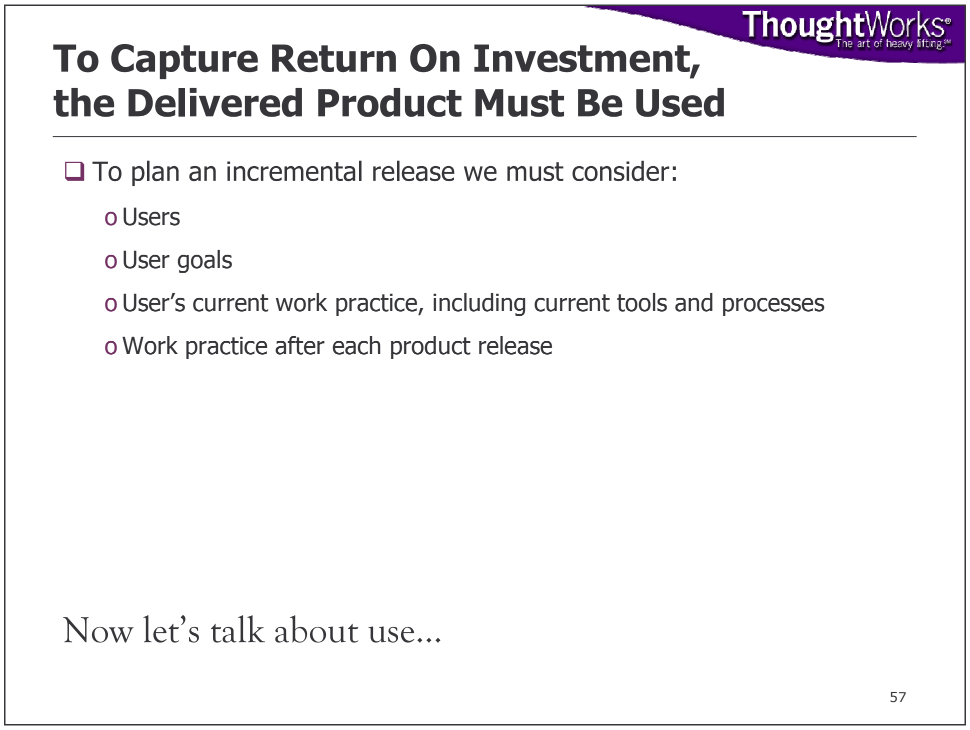
Notes



## Building & Evaluating Complete Releases Helps Reduce Risk

- ❑ Prove general architectural approach
- ❑ Validate domain model
- ❑ Perform user acceptance testing
  - o Showing users complete workflow lets them effectively evaluate and give feedback
- ❑ Test for performance
- ❑ Test for load
- ❑ Deploy in target environment

56



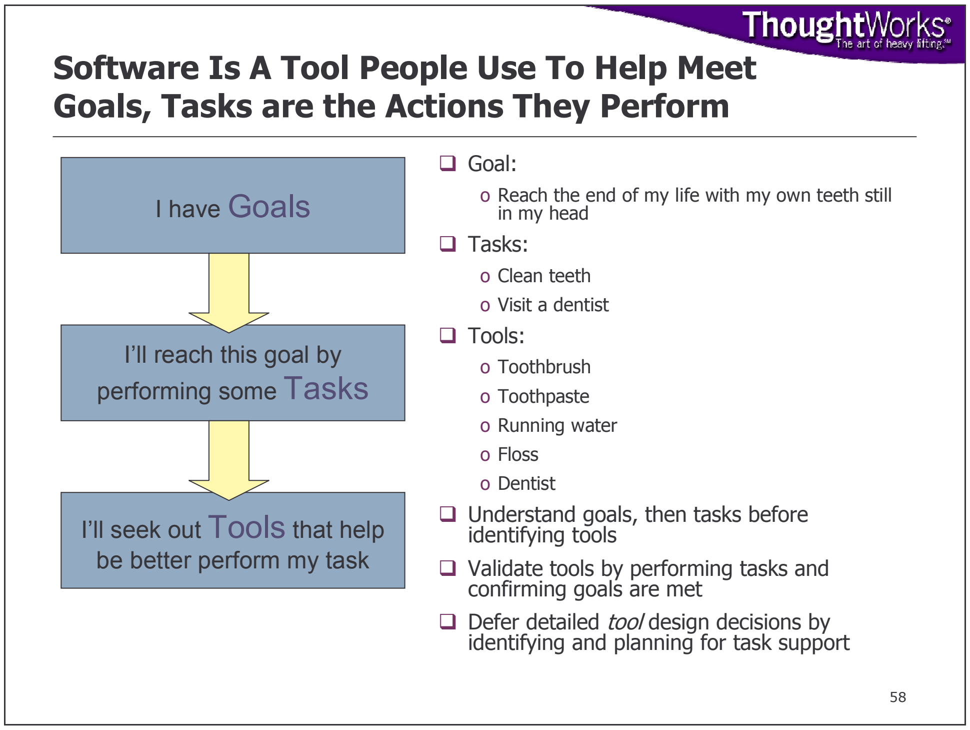
## To Capture Return On Investment, the Delivered Product Must Be Used

- ❑ To plan an incremental release we must consider:
  - o Users
  - o User goals
  - o User’s current work practice, including current tools and processes
  - o Work practice after each product release

Now let’s talk about use…

57

Notes



ThoughtWorks® The art of heavy lifting.℠

## Software Is A Tool People Use To Help Meet Goals, Tasks are the Actions They Perform

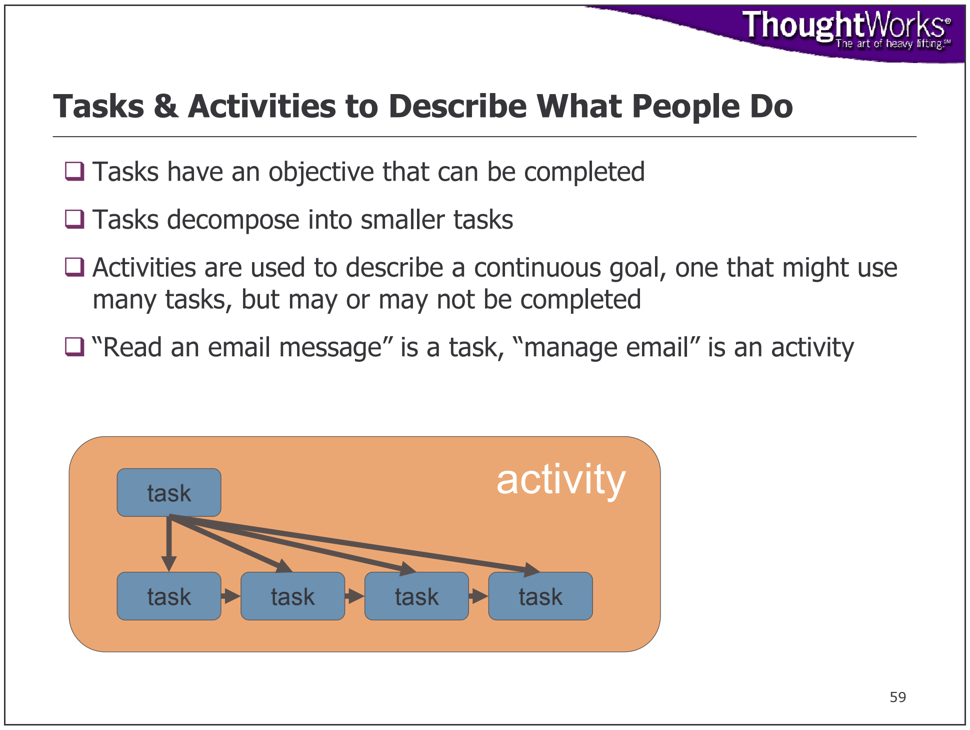
I have Goals

I'll reach this goal by performing some Tasks

I'll seek out Tools that help be better perform my task

- ❑ Goal:
  - o Reach the end of my life with my own teeth still in my head
- ❑ Tasks:
  - o Clean teeth
  - o Visit a dentist
- ❑ Tools:
  - o Toothbrush
  - o Toothpaste
  - o Running water
  - o Floss
  - o Dentist
- ❑ Understand goals, then tasks before identifying tools
- ❑ Validate tools by performing tasks and confirming goals are met
- ❑ Defer detailed *tool* design decisions by identifying and planning for task support

58



ThoughtWorks® The art of heavy lifting.℠

## Tasks & Activities to Describe What People Do

- ❑ Tasks have an objective that can be completed
- ❑ Tasks decompose into smaller tasks
- ❑ Activities are used to describe a continuous goal, one that might use many tasks, but may or may not be completed
- ❑ "Read an email message" is a task, "manage email" is an activity

activity

task

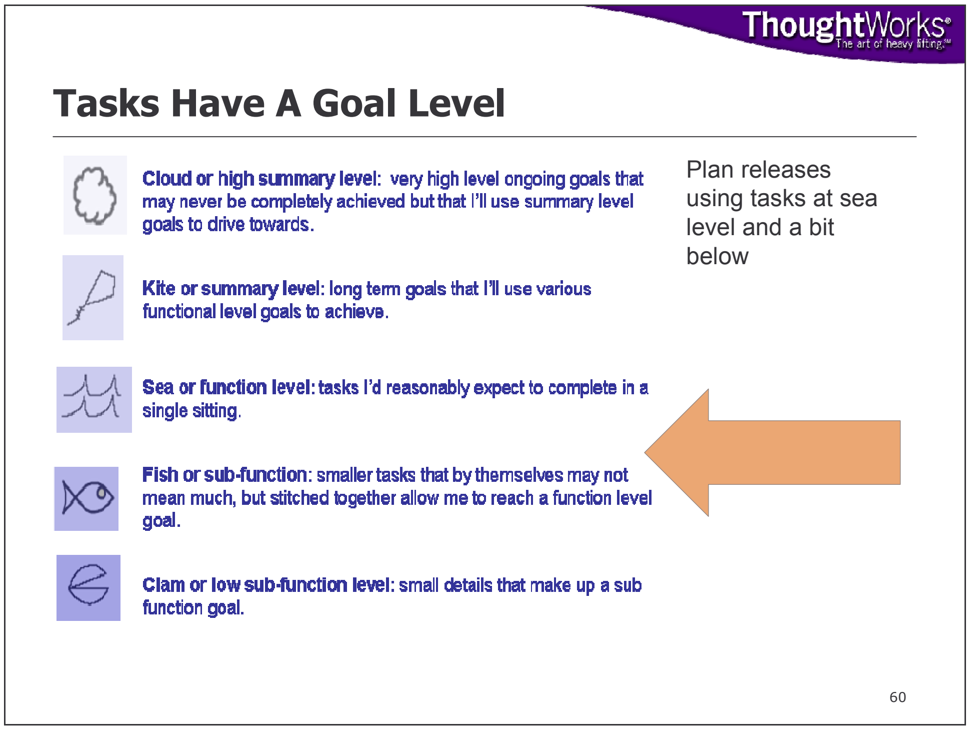
task → task → task → task

59

**Additional Reading:**

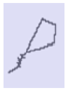
- Norman, **Human-Centered Design Considered Harmful** (August 2005, http://www.jnd.org/dn.mss/human-centered.html)
- Norman, **Logic Versus Usage: The Case for Activity-Centered Design** (2006, http://www.jnd.org/dn.mss/logic_versus_usage_t.html)

Notes



## Tasks Have A Goal Level

**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.

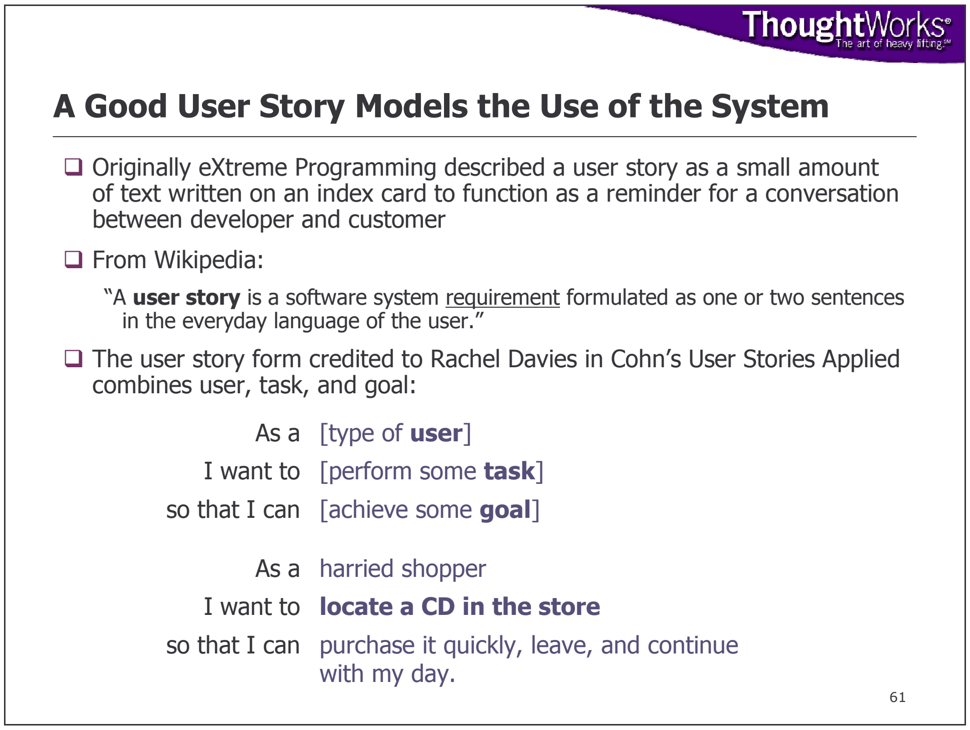
**Sea or function level:** tasks I'd reasonably expect to complete in a single sitting.

**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.

Plan releases using tasks at sea level and a bit below

60



## A Good User Story Models the Use of the System

- ❑ Originally eXtreme Programming described a user story as a small amount of text written on an index card to function as a reminder for a conversation between developer and customer
- ❑ From Wikipedia:

  "A **user story** is a software system requirement formulated as one or two sentences in the everyday language of the user."
- ❑ The user story form credited to Rachel Davies in Cohn's User Stories Applied combines user, task, and goal:

As a [type of **user**]
I want to [perform some **task**]
so that I can [achieve some **goal**]

As a harried shopper
I want to **locate a CD in the store**
so that I can purchase it quickly, leave, and continue with my day.

61

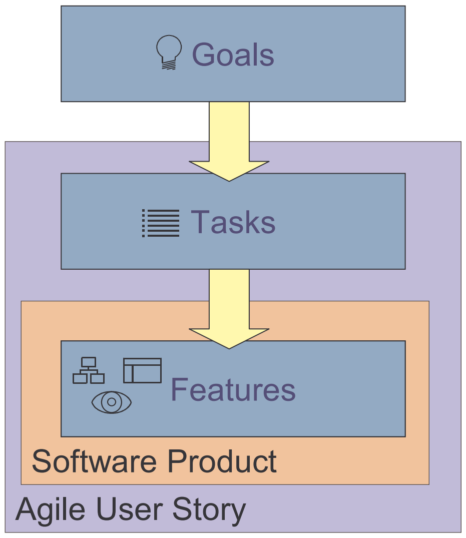
### Additional Reading:

- Cohn, **User Stories Applied** (Addison-Wesley, 2004)

Notes

ThoughtWorks® The art of heavy lifting.℠

## Identify And Plan Using User Tasks Now, Defer Specific Feature Choices Till Later



- ❑ Understand Business & User Goals
- ❑ Understand user's tasks, and/or the business process that supports goals
- ❑ Select tasks to support with software
- ❑ Defer decisions for and designs of specific features till later
  - o The often used phrase "latest responsible moment" comes from Lean Software Development:

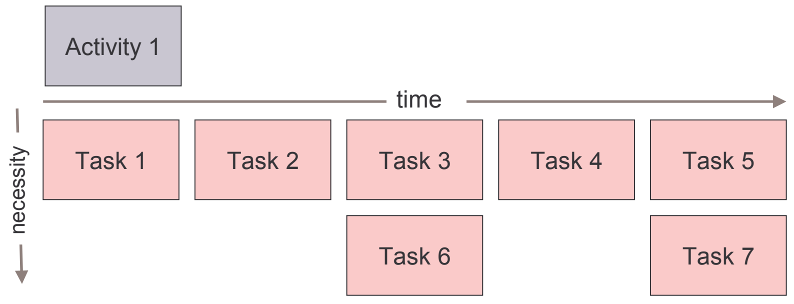
  "Put off decisions as long as you can: to the latest responsible moment. But it's the latest *responsible* moment, not the "*last possible*" moment. That wouldn't be responsible."
- ❑ An Agile-style user story could refer to a feature, or user, task, and goal. Favor the latter.

62

ThoughtWorks® The art of heavy lifting.℠

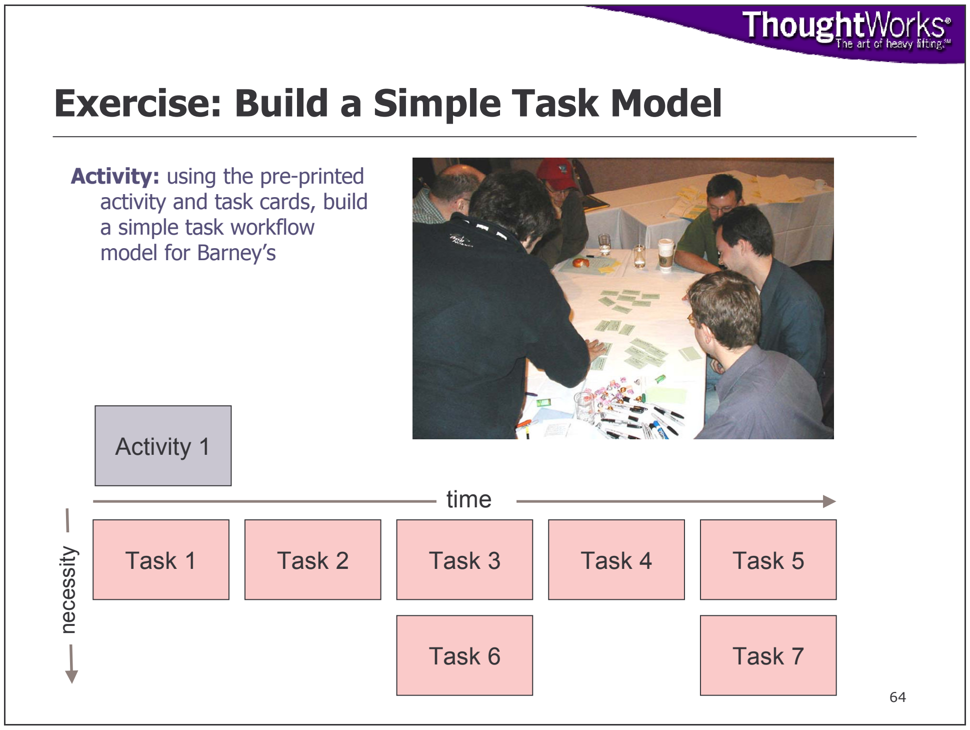
## A Task Workflow Model Organizes Tasks to Represent Workflow

- ❑ To build a simple task workflow model:
  - o Draw a left to right axis representing time, a top to bottom axis labeled necessity
  - o Identify high level activities performed by users of the system and place them above the time axis in the order that seems reasonable
  - o Within each activity, organize tasks in the order they're most likely completed
  - o Move tasks up and down depending on how likely they are to be performed in a typical instance of use



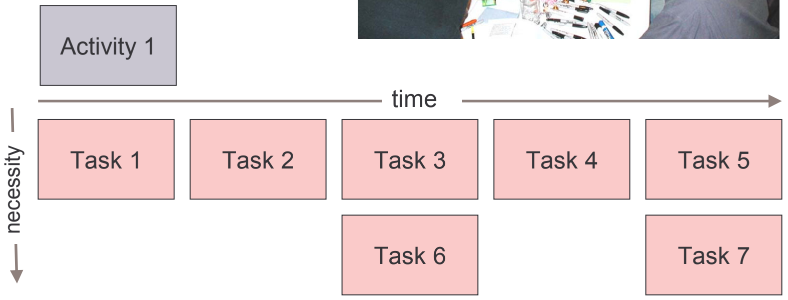
63

Notes



## Exercise: Build a Simple Task Model

**Activity:** using the pre-printed activity and task cards, build a simple task workflow model for Barney's



64

## 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

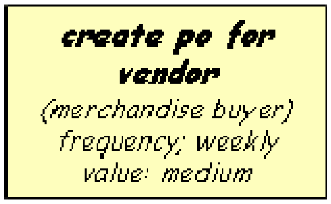
### How valuable is this feature?

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

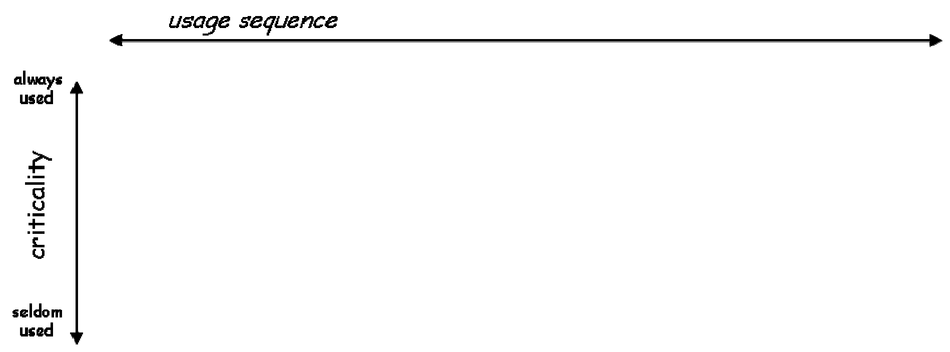
## Step 3: Build Your Incremental Release Plan

### 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

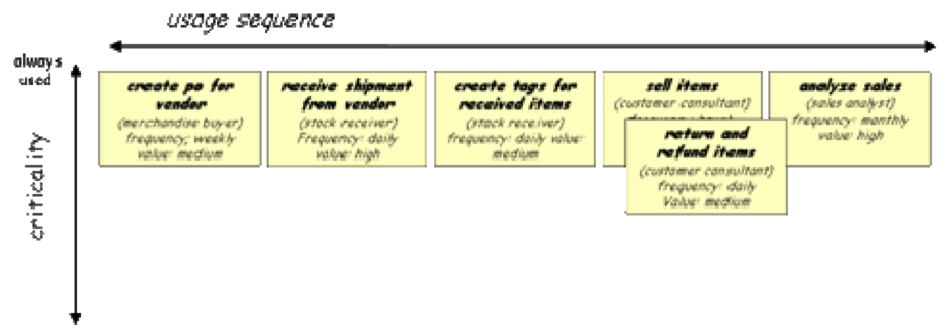
You now need to place features in the model according to usage sequence and criticality. So let's say a little more about these two axis:

### *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.

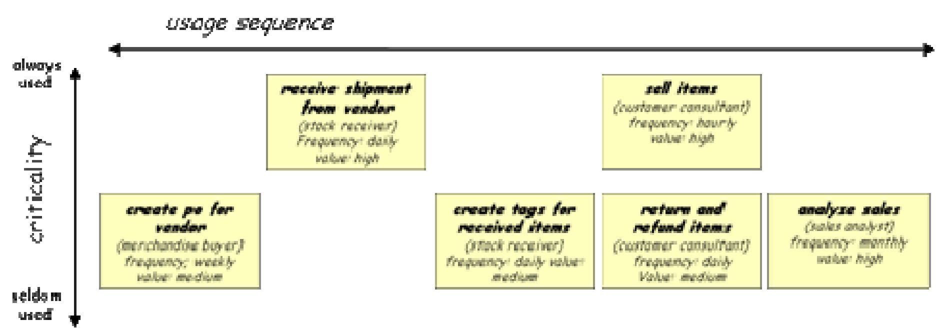

usage sequence
always used
criticality
create po for vendor (merchandise buyer) frequency: weekly value: medium
receive shipment from vendor (stock receiver) Frequency: daily value: high
create tags for received items (stock receiver) frequency: daily value: medium
sell items (customer consultant)
return 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.


usage sequence
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

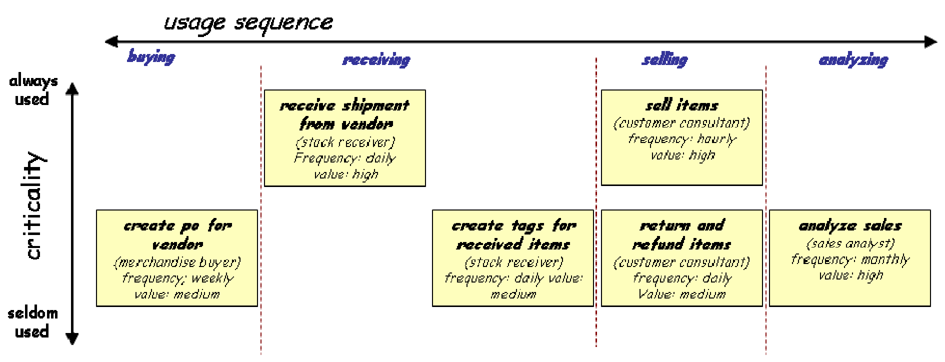
If we adjust our features for criticality, our model might look a bit like this.

## 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

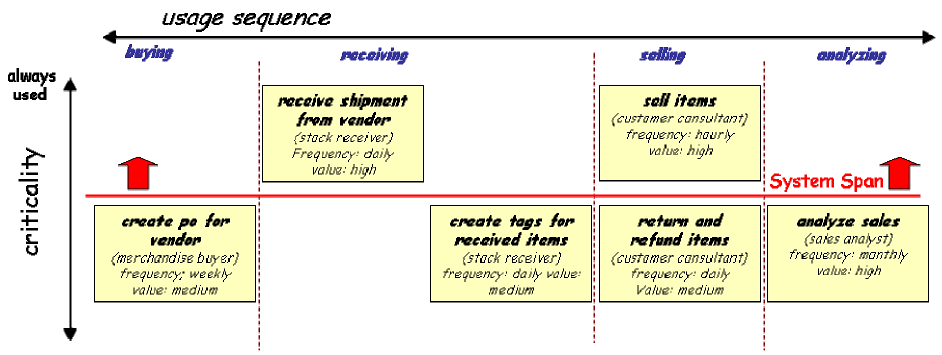
## *What Is This Model Telling Us?*

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:

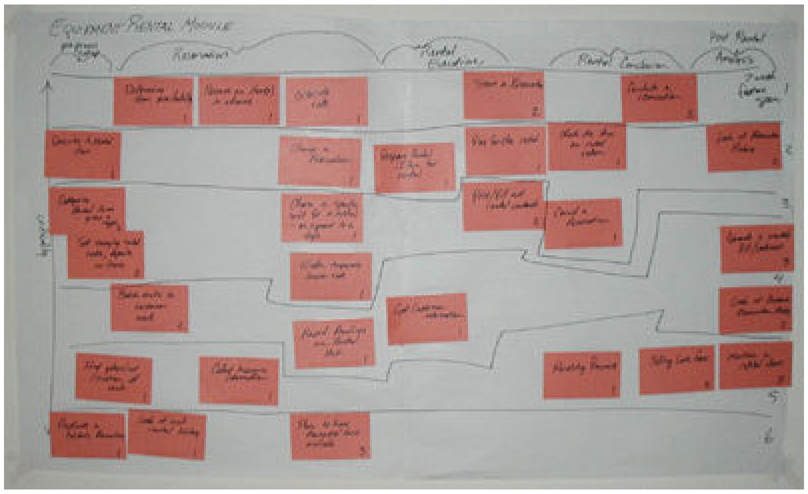
The plan may now be "sliced" horizontally into spanning releases. Well, sort of horizontally. Choose features below the marked first span that group together logically. Choose enough features such that the estimated elapsed development days fit within an appropriate release date. Lines drawn through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



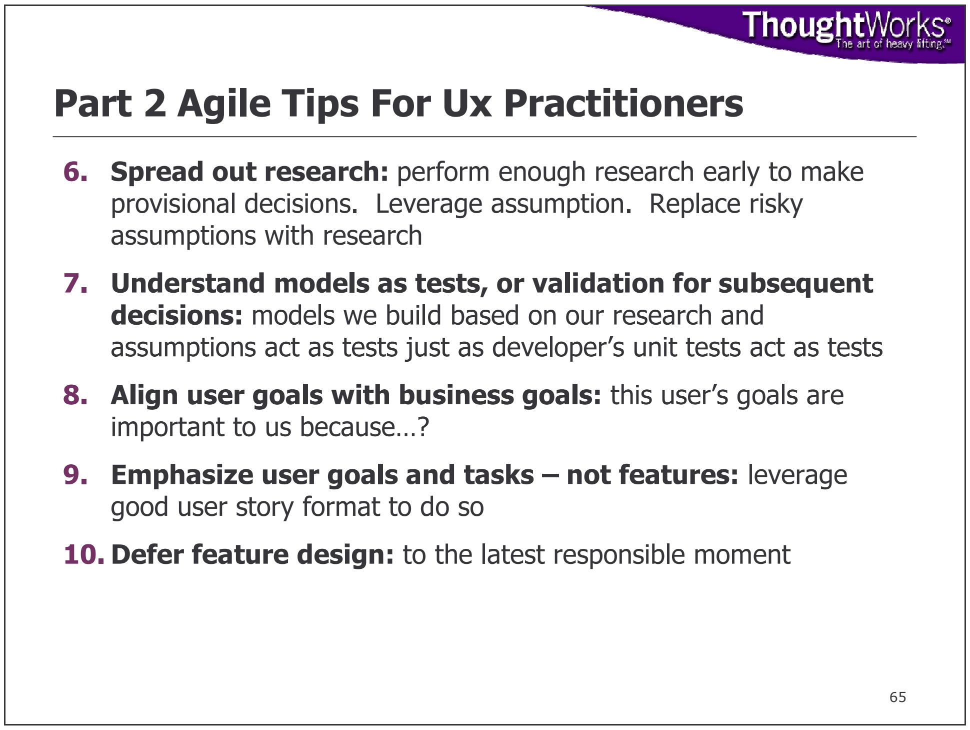
## What Just Happened Here?

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)

Notes



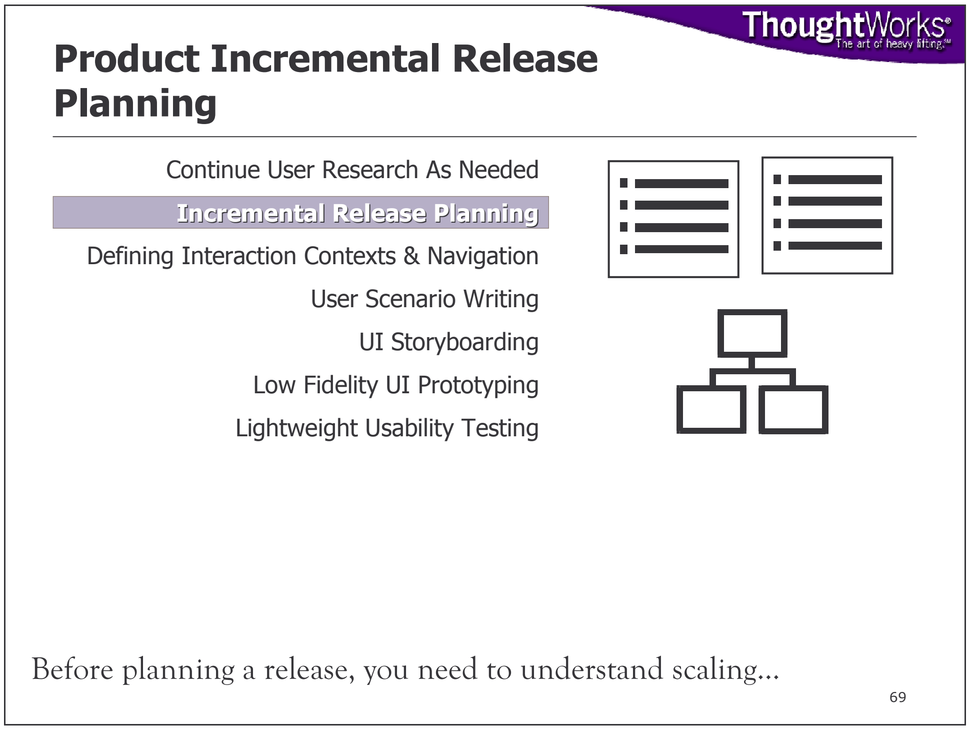
## Part 2 Agile Tips For Ux Practitioners

6. **Spread out research:** perform enough research early to make provisional decisions. Leverage assumption. Replace risky assumptions with research
7. **Understand models as tests, or validation for subsequent decisions:** models we build based on our research and assumptions act as tests just as developer's unit tests act as tests
8. **Align user goals with business goals:** this user's goals are important to us because…?
9. **Emphasize user goals and tasks – not features:** leverage good user story format to do so
10. **Defer feature design:** to the latest responsible moment

65

## Part 3: Building & Validation

In part three we'll dive headfirst into a simulated Agile Development cycle. You'll need to understand just a couple quick concepts before you can plan your product releases. Then, given that plan you'll use paper prototyping to build the first release of the software you've planned.



## Product Incremental Release Planning

Continue User Research As Needed

**Incremental Release Planning**

Defining Interaction Contexts & Navigation

User Scenario Writing

UI Storyboarding

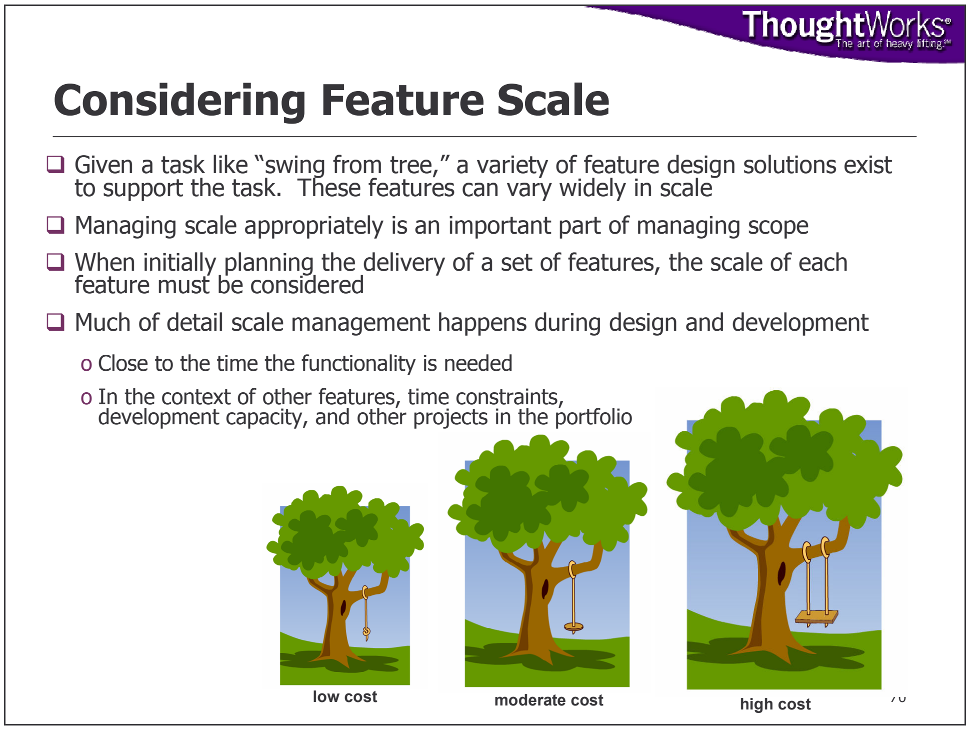
Low Fidelity UI Prototyping

Lightweight Usability Testing

Before planning a release, you need to understand scaling…

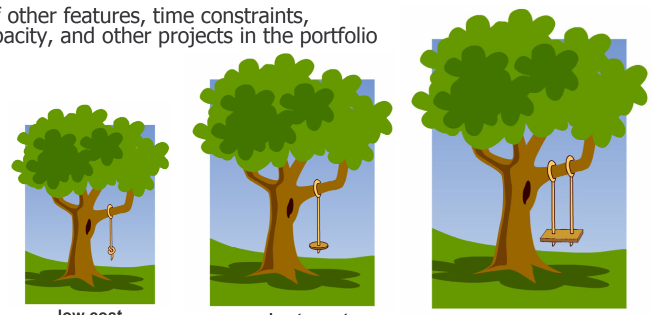
69

Notes



## Considering Feature Scale

- ❑ Given a task like "swing from tree," a variety of feature design solutions exist to support the task. These features can vary widely in scale
- ❑ Managing scale appropriately is an important part of managing scope
- ❑ When initially planning the delivery of a set of features, the scale of each feature must be considered
- ❑ Much of detail scale management happens during design and development
  - o Close to the time the functionality is needed
  - o In the context of other features, time constraints, development capacity, and other projects in the portfolio



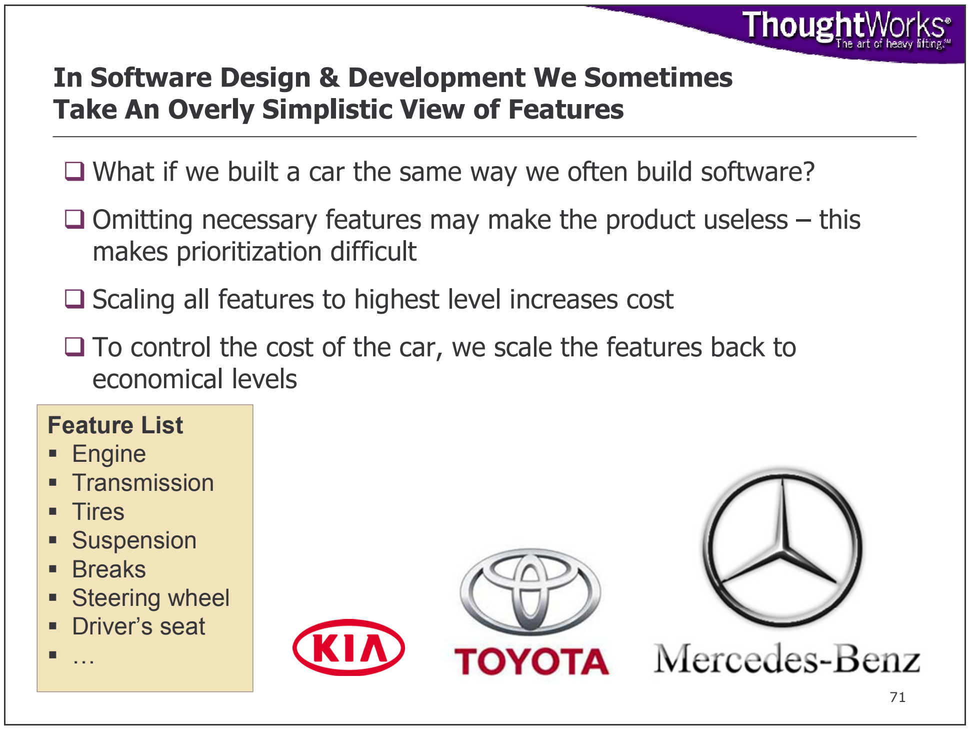
low cost moderate cost high cost

70

**Additional Reading:**

- Patton, **Finish On Time By Managing Scale** (StickyMinds column, www.abstractics.com /papers)

Notes



ThoughtWorks® The art of heavy lifting.℠

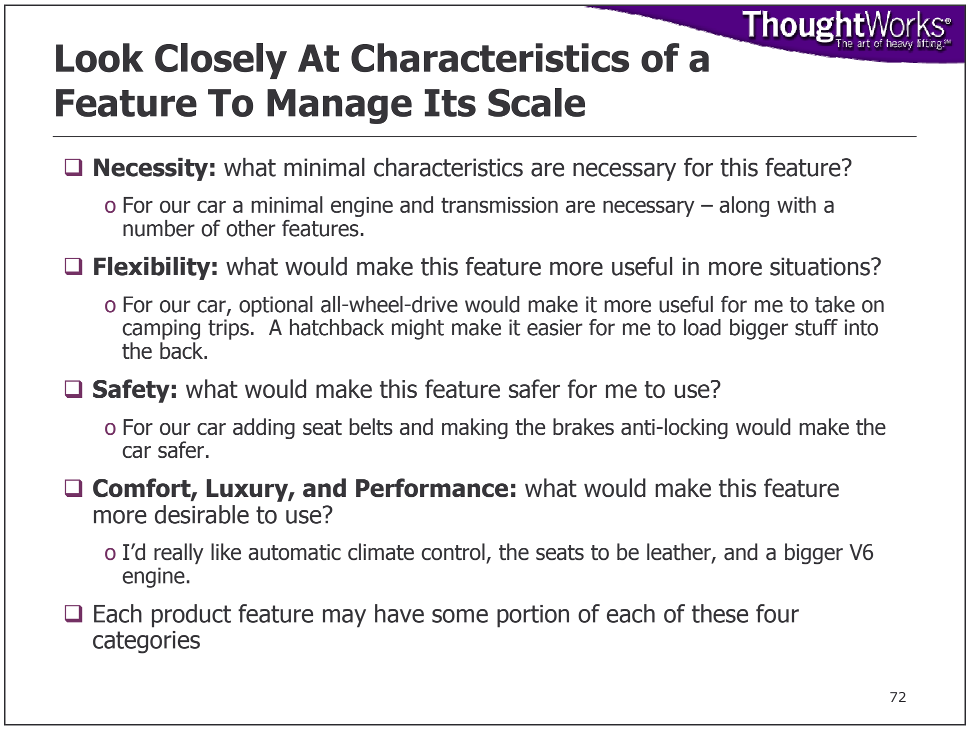
## In Software Design & Development We Sometimes Take An Overly Simplistic View of Features

- ❑ What if we built a car the same way we often build software?
- ❑ Omitting necessary features may make the product useless – this makes prioritization difficult
- ❑ Scaling all features to highest level increases cost
- ❑ To control the cost of the car, we scale the features back to economical levels

**Feature List**
- Engine
- Transmission
- Tires
- Suspension
- Breaks
- Steering wheel
- Driver's seat
- …

KIA TOYOTA Mercedes-Benz

71



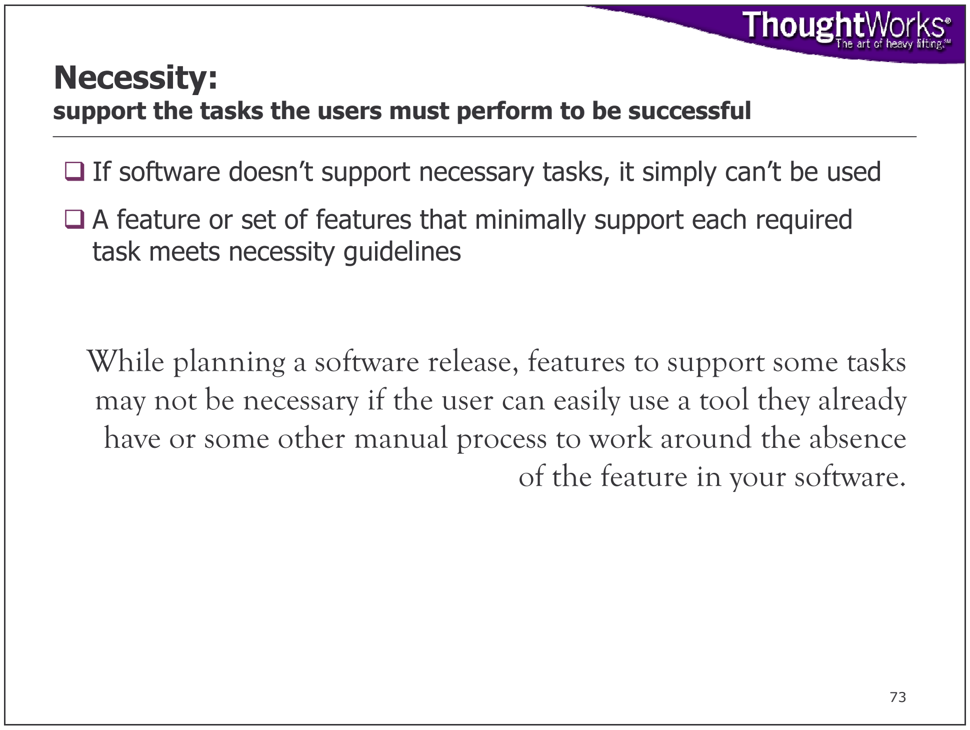
ThoughtWorks® The art of heavy lifting.℠

## Look Closely At Characteristics of a Feature To Manage Its Scale

- ❑ **Necessity:** what minimal characteristics are necessary for this feature?
  - o 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?
  - o 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?
  - o For our car adding seat belts and making the brakes anti-locking would make the car safer.
- ❑ **Comfort, Luxury, and Performance:** what would make this feature more desirable to use?
  - o I'd really like automatic climate control, the seats to be leather, and a bigger V6 engine.
- ❑ Each product feature may have some portion of each of these four categories

72

Notes

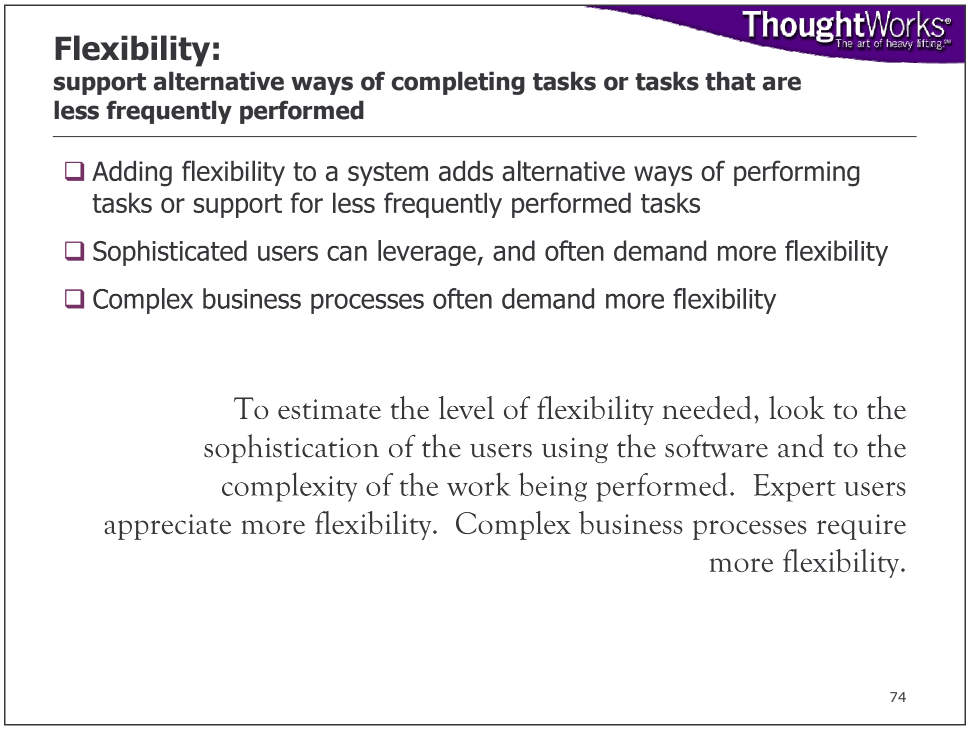


## Necessity:
**support the tasks the users must perform to be successful**

- ❑ If software doesn't support necessary tasks, it simply can't be used
- ❑ A feature or set of features that minimally support each required task meets necessity guidelines

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.

73



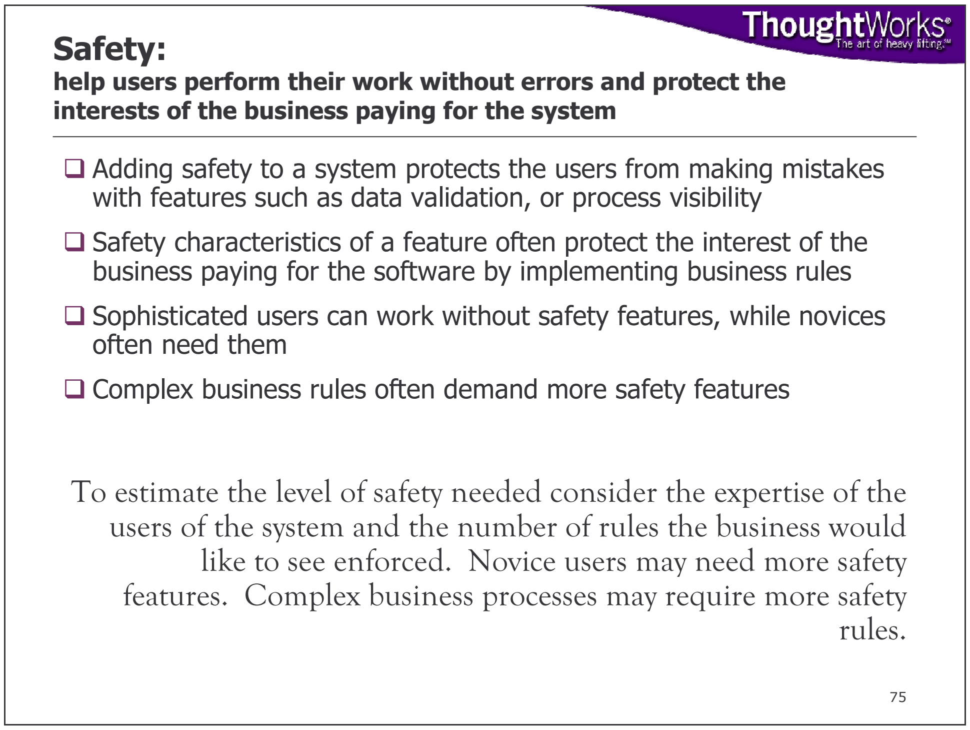
## Flexibility:
**support alternative ways of completing tasks or tasks that are less frequently performed**

- ❑ Adding flexibility to a system adds alternative ways of performing tasks or support for less frequently performed tasks
- ❑ Sophisticated users can leverage, and often demand more flexibility
- ❑ Complex business processes often demand more flexibility

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.

74

Notes



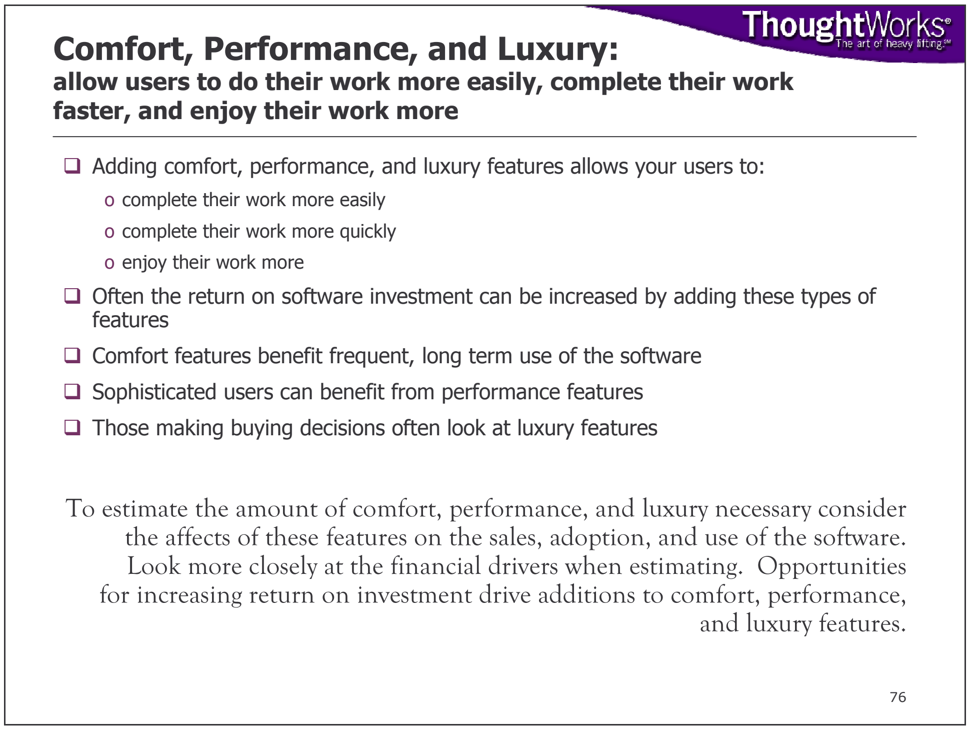
## Safety:

**help users perform their work without errors and protect the interests of the business paying for the system**

- ❑ Adding safety to a system protects the users from making mistakes with features such as data validation, or process visibility
- ❑ Safety characteristics of a feature often protect the interest of the business paying for the software by implementing business rules
- ❑ Sophisticated users can work without safety features, while novices often need them
- ❑ Complex business rules often demand more safety features

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.

75



## Comfort, Performance, and Luxury:

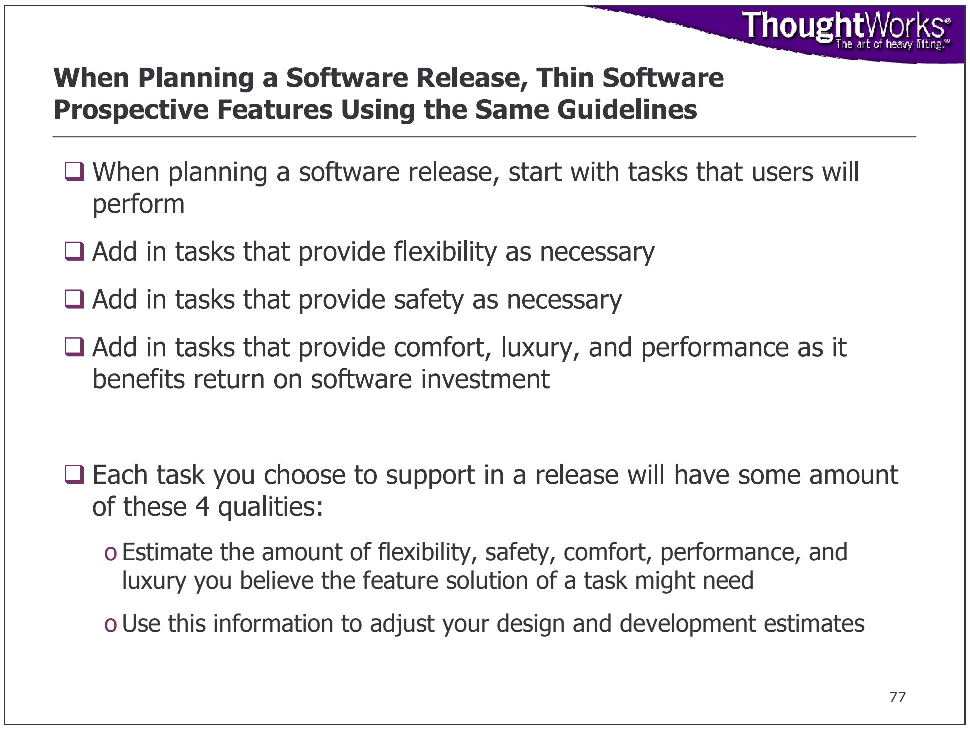
**allow users to do their work more easily, complete their work faster, and enjoy their work more**

- ❑ Adding comfort, performance, and luxury features allows your users to:
  - o complete their work more easily
  - o complete their work more quickly
  - o enjoy their work more
- ❑ Often the return on software investment can be increased by adding these types of features
- ❑ Comfort features benefit frequent, long term use of the software
- ❑ Sophisticated users can benefit from performance features
- ❑ Those making buying decisions often look at luxury features

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 features.

76

Notes

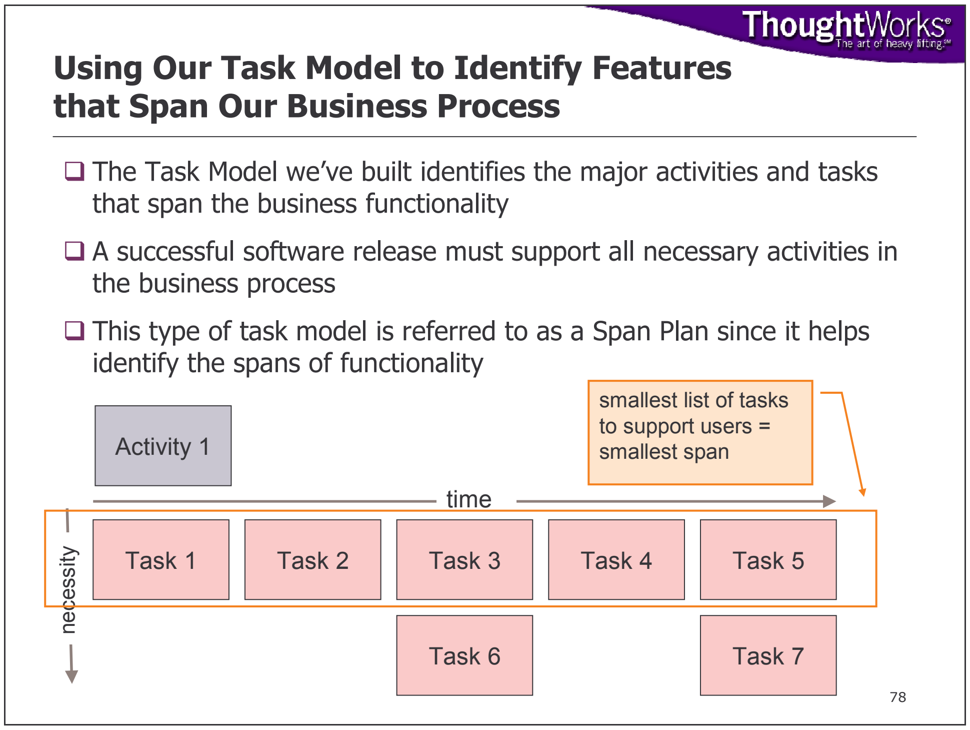


## When Planning a Software Release, Thin Software Prospective Features Using the Same Guidelines

- When planning a software release, start with tasks that users will perform
- Add in tasks that provide flexibility as necessary
- Add in tasks that provide safety as necessary
- Add in tasks that provide comfort, luxury, and performance as it benefits return on software investment

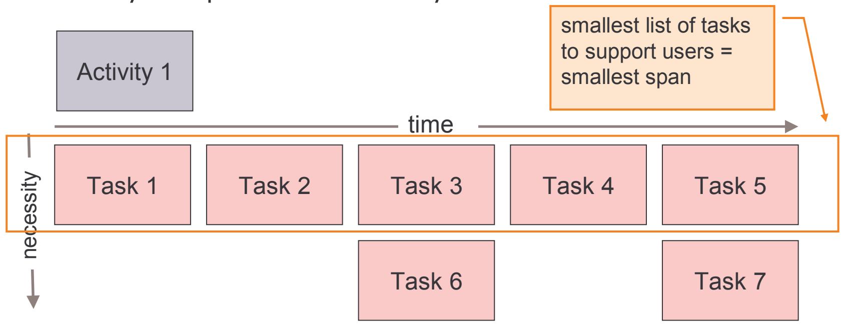
- Each task you choose to support in a release will have some amount of these 4 qualities:
  - Estimate the amount of flexibility, safety, comfort, performance, and luxury you believe the feature solution of a task might need
  - Use this information to adjust your design and development estimates

77



## Using Our Task Model to Identify Features that Span Our Business Process

- The Task Model we've built identifies the major activities and tasks that span the business functionality
- A successful software release must support all necessary activities in the business process
- This type of task model is referred to as a Span Plan since it helps identify the spans of functionality

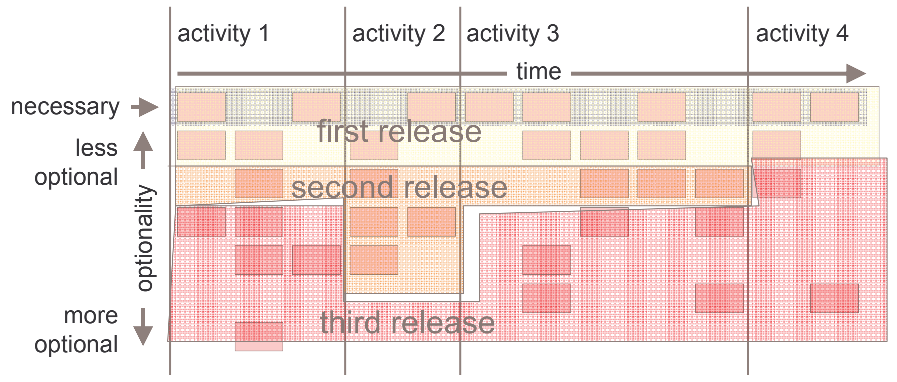


78

Notes

ThoughtWorks® The art of heavy lifting.℠

## Identify Releases In a Span Plan By Slicing Horizontally

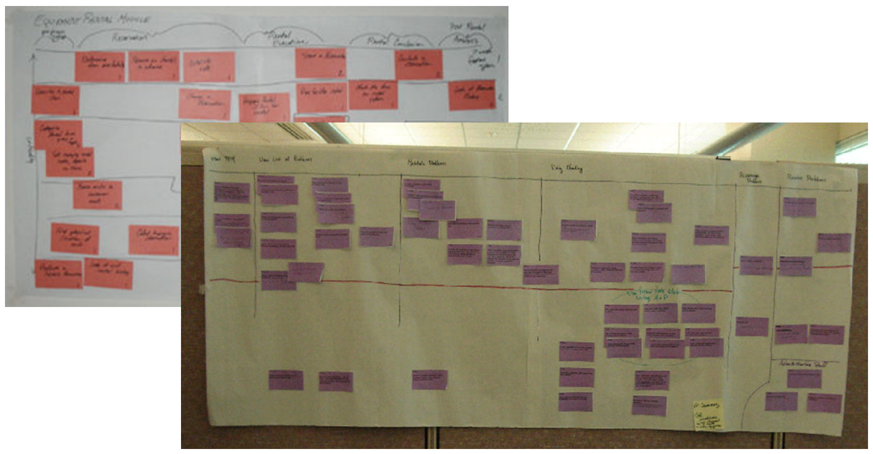


- ❑ Choose coherent groups of features that consider the span of business functionality and user activities.
- ❑ Support all necessary activities with the first release
- ❑ Improve activity support with subsequent releases

79

ThoughtWorks® The art of heavy lifting.℠

## Sliced Span Plans



- ❑ Slices often take irregular shapes to ensure coherent groups of product features

80

Notes

ThoughtWorks® The art of heavy lifting.℠

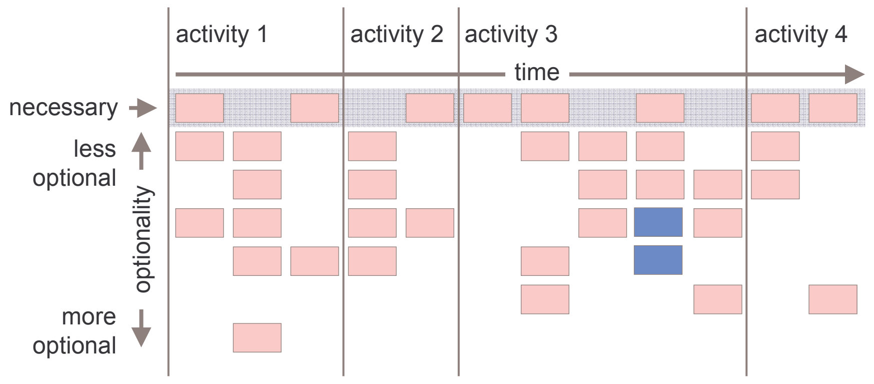
## Use Feature Thinning Guidelines to Reduce the Size of a Release

- The topmost row of the span could be the first, smallest release
- By minimizing a release we can realize financial and risk reduction benefits earlier
- The top span represents the minimal tasks users need to accomplish to reach their goals. How can we split these "high level stories" into smallest parts?
  - Can the feature(s) to support a task have reduced safety?
  - Can the feature(s) to reduce a task have less comfort, performance, and luxury?
  - Are there optional tasks that can be supported in a subsequent release?
  - For necessary tasks, look at the steps – or subtasks that make up the task. Can any of those steps be made optional?
  - Move cards around the model, or split cards into multiple cards to defer task support, or specific feature characteristics till later releases

81

ThoughtWorks® The art of heavy lifting.℠

## Splitting Span Plan Tasks



- Consider tasks more optional
- Split tasks into optional parts

82

## 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.

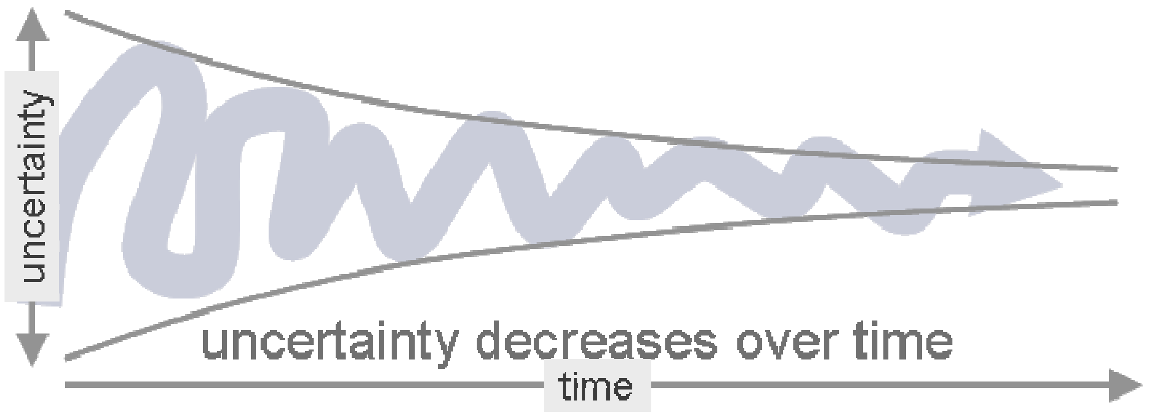
Let's go back to our budding artist.  As I continued to draw, and got a little help, I learned that artists often sketched out the basic shape of the thing they were drawing.  They resolved the positioning and proportions of elements on the page first.  Painters often created an under-painting or a preliminary 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

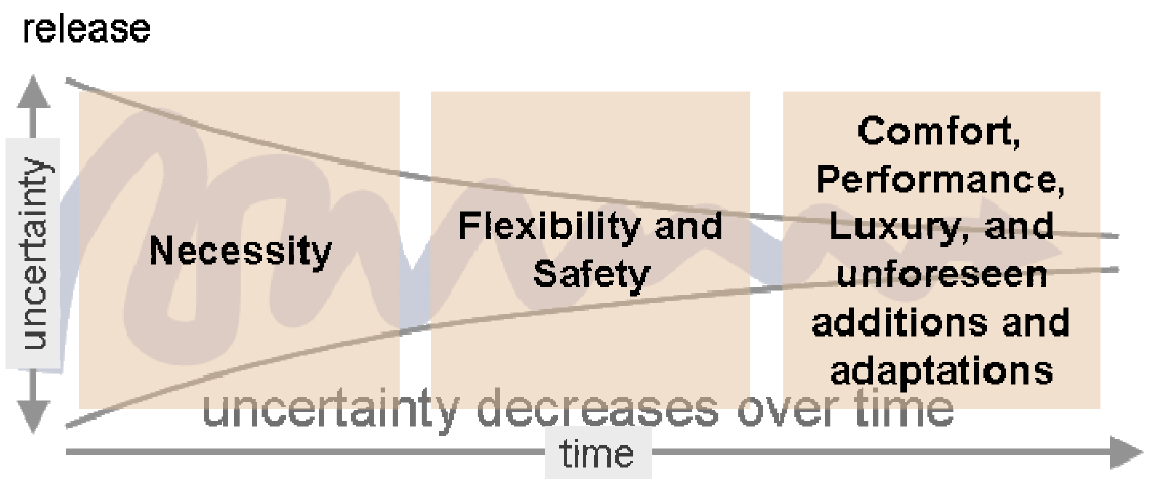
There's a fairly obvious law that we often forget to accommodate. It's difficult to predict outcomes and events in the distant future and easier to predict them in the near future.

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.

Notes

## Before You Create A Release Plan, You Need To Know A Bit About Your Development Approach

- ❑ You'll be *developing* your product today using componentized paper prototypes
- ❑ Your first released prototype will be built in 20 minutes of *development*
- ❑ A successful release will span the entire business process supporting all tasks you feel are necessities, then filling in with features to support tasks that are optional
- ❑ Estimate your release based on how much you believe you can complete in 20 minutes of prototyping

83

## Use Span Planning & Feature Thinning Guidelines to Plan Small Coherent Releases

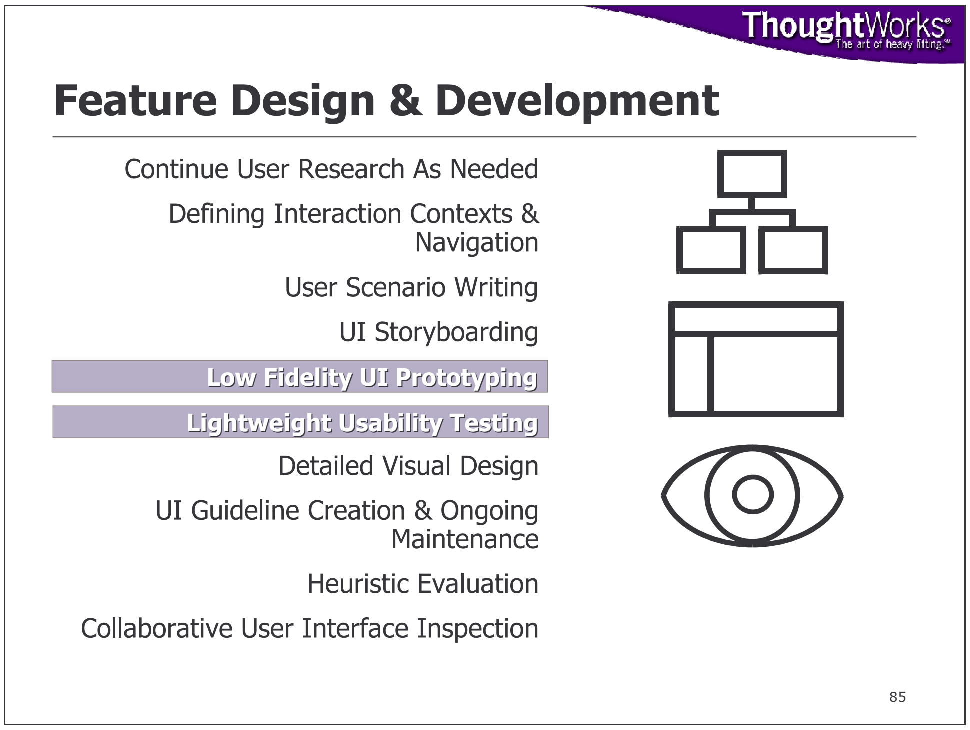
Activity:

- ❑ Identify 2 candidate releases for Barney's
- ❑ Thin your span plan using feature thinning guidelines
- ❑ As a group discuss what sorts of features might support each task, and if and how they could be thinned
- ❑ You have 10 minutes

- ❑ Thin support for tasks using the following guidelines:
  - o Necessity: is supporting this task necessary in this release?
  - o Flexibility: does supporting this task add flexible alternative ways of doing things?
  - o Safety: does supporting this feature add safety for the user or business paying for the software?
  - o Comfort, Performance, and Luxury: does supporting these tasks make the software easier to use, faster to use, more enjoyable to use?

84

Notes



ThoughtWorks® The art of heavy lifting.℠

## Feature Design & Development

Continue User Research As Needed

Defining Interaction Contexts & Navigation

User Scenario Writing

UI Storyboarding

**Low Fidelity UI Prototyping**

**Lightweight Usability Testing**

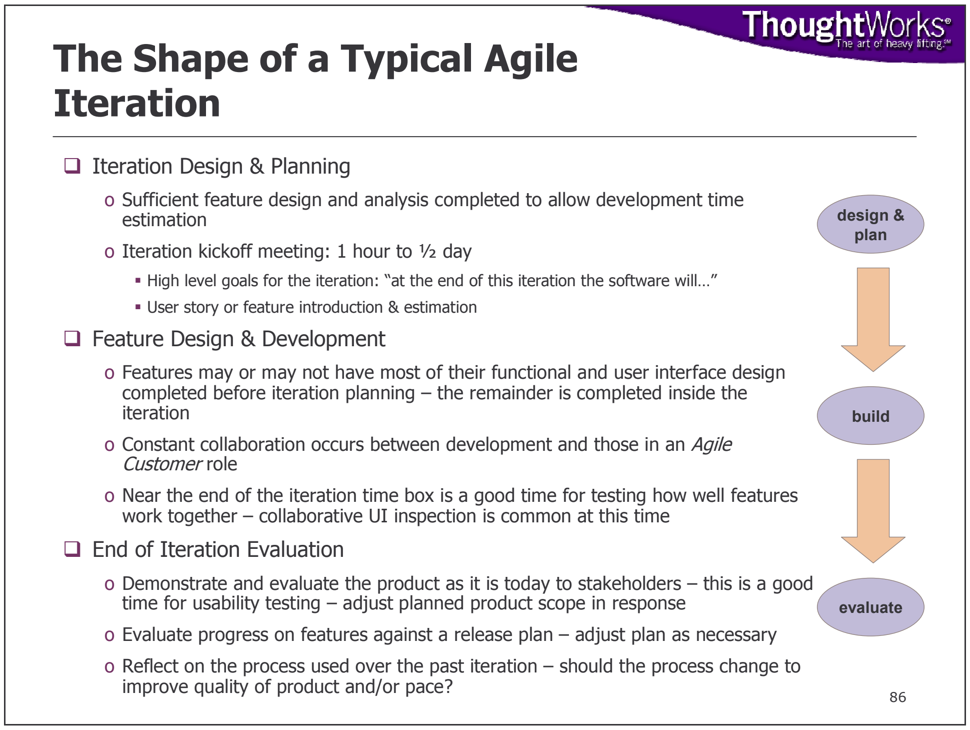
Detailed Visual Design

UI Guideline Creation & Ongoing Maintenance

Heuristic Evaluation

Collaborative User Interface Inspection

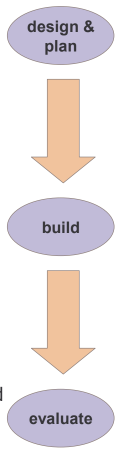
85



ThoughtWorks® The art of heavy lifting.℠

## The Shape of a Typical Agile Iteration

- ❑ Iteration Design & Planning
  - o Sufficient feature design and analysis completed to allow development time estimation
  - o Iteration kickoff meeting: 1 hour to ½ day
    - ▪ High level goals for the iteration: "at the end of this iteration the software will…"
    - ▪ User story or feature introduction & estimation
- ❑ Feature Design & Development
  - o Features may or may not have most of their functional and user interface design completed before iteration planning – the remainder is completed inside the iteration
  - o Constant collaboration occurs between development and those in an *Agile Customer* role
  - o Near the end of the iteration time box is a good time for testing how well features work together – collaborative UI inspection is common at this time
- ❑ End of Iteration Evaluation
  - o Demonstrate and evaluate the product as it is today to stakeholders – this is a good time for usability testing – adjust planned product scope in response
  - o Evaluate progress on features against a release plan – adjust plan as necessary
  - o Reflect on the process used over the past iteration – should the process change to improve quality of product and/or pace?

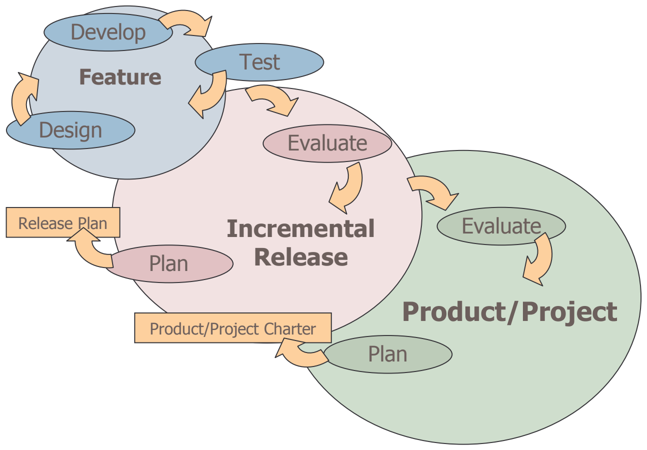


86

Notes

## In Our Process Miniature, We'll Combine Releases With Iterations

Please Don't Try This At Home



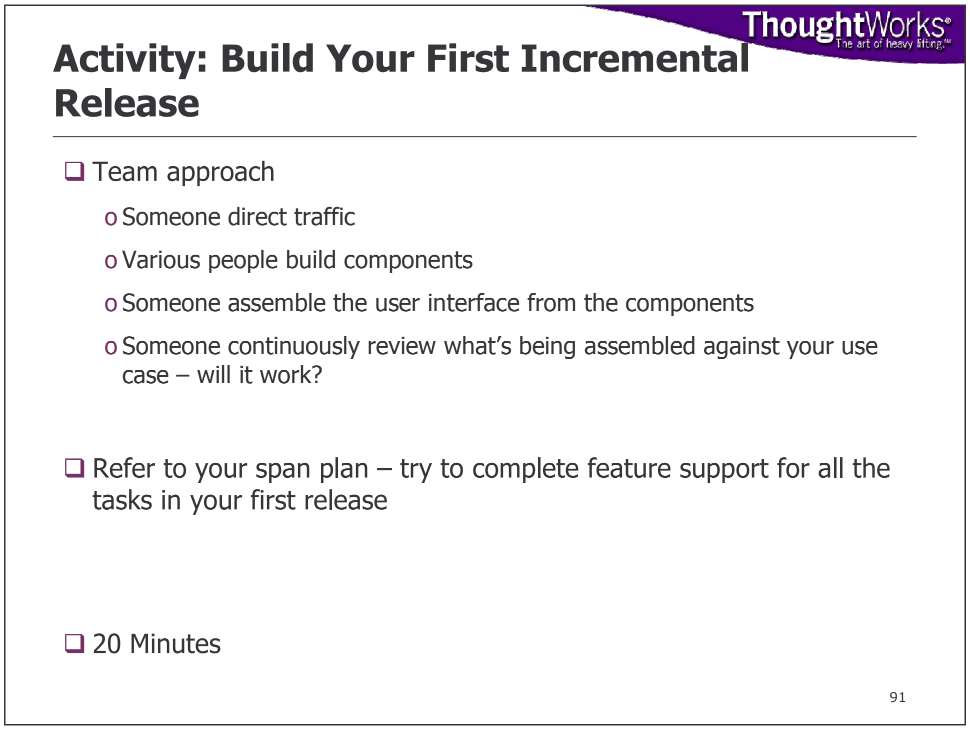
88

## Paper Prototyping Basics

- Tools
  - Card Stock (use for screen backgrounds and cut up for components)
  - Index Cards (lined cards make great lists)
  - Scissors or Xacto knife
  - Cello tape
  - Repositionable tape
  - Pencils
  - Sharp felt tip pens
  - Transparency film (great to write on)
- Team approach
  - Someone direct traffic
  - Various people build components
  - Someone assemble the user interface from the components
  - Someone continuously review what's being assembled against your use case – will it work?

89

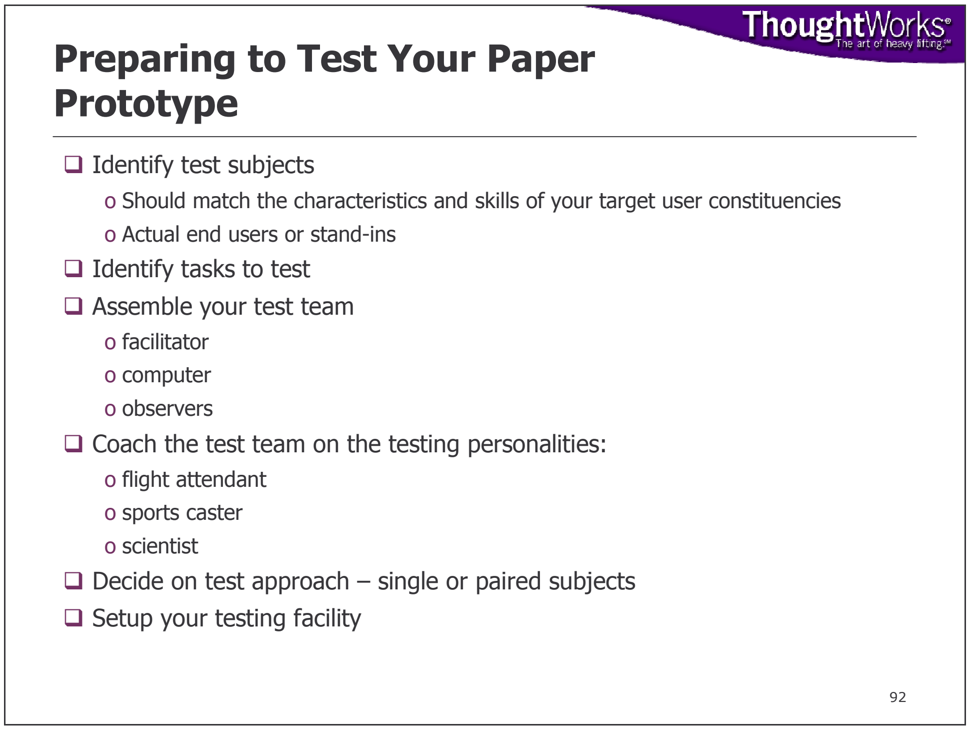
Notes



## Activity: Build Your First Incremental Release

- Team approach
  - Someone direct traffic
  - Various people build components
  - Someone assemble the user interface from the components
  - Someone continuously review what's being assembled against your use case – will it work?
- Refer to your span plan – try to complete feature support for all the tasks in your first release
- 20 Minutes

91



## Preparing to Test Your Paper Prototype

- Identify test subjects
  - Should match the characteristics and skills of your target user constituencies
  - Actual end users or stand-ins
- Identify tasks to test
- Assemble your test team
  - facilitator
  - computer
  - observers
- Coach the test team on the testing personalities:
  - flight attendant
  - sports caster
  - scientist
- Decide on test approach – single or paired subjects
- Setup your testing facility

92

Notes

## Run Your Usability Test

- Facilitator introduces the team.
- Facilitator introduces tasks to perform and goals, then invites test participants to "think out loud" and begin.
- Facilitator plays sports-caster; keeps subject talking, narrating when necessary.
- Observers record data – use post-it notes to make downstream analysis move faster.
- When the test is complete observers may ask test participants questions.
- Thank test participants.
- Consolidate data.
  - How many issues did you detect? Consider issues as items you'd change.

93

## Testing In Action

94

**Additional Reading:**

- Patton, **Test Software Before You Code** (StickyMinds column, www.abstractics.com /papers)

## User Interface Paper Prototyping and Usability Testing

The following for short articles give simple step by step instructions for writing out the steps of a simple user task, moving from that task to a simple user interface prototype, and then testing that prototype. The final article discusses considerations when using user story driven approaches to drive user interface design.

## Task to Abstract Components, Step by Step

## 1. Start with a task or collection of tasks

In user interface design a proposed user of a product will attempt to meet his or her goals by executing "tasks" using the product. For our purposes here, we'll define a task as:

- ***a series of actions taken by a user of a product in pursuit of a goal***

The name of a task should not necessarily imply a particular way of accomplishing that task. For example if I have a goal to happily listen to the most recent Mike Doughty CD in my car, I might walk into a music store to buy it. "Buy a CD" is a task. It's easy to assume, given the bit of context I just gave you, that it will be in a store with shelves of CDs and that I might pay for the CD at a cash register operated by sales person at the music store. But, given a different context, such as at home, the store could be an on-line store where the mechanisms for finding and paying for a CD might be quite different.

### Tasks contain other tasks

For example my "Buy a CD" task might contain the tasks: locate the CD I want in the store, check the price of the CD, purchase the CD. And it doesn't stop there. Each one of those tasks may further break down into smaller tasks. The lower you go, the more you decompose tasks into smaller tasks, the more you need to decide or assume about where and exactly how the task will be accomplished.

### Tasks have goal level

In Writing Effective Use Cases, Alistair Cockburn introduces a useful concept called goal level. To explain goal level he uses an altitude metaphor with sea level falling in the middle of the model. He also refers to sea level as "function" level. The test for a sea level goal is: would I as someone engaged in this task expect to finish it 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 a CD might be such a task.

**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.

- ***For designing user interface start with tasks that are functional or sea level***

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.

## 2. Write the sub-tasks, or steps that allow our user to reach her goal

Select a candidate user, context of use, and functional task. The user selection is important. Me stopping to buy a CD in the context of running errands may have different steps than you choosing to buy a CD while surfing the web during a lunch break at work.

One important part of the context for us to proceed is the assumption that the context includes some product that we'll be designing. For our use here we'll assume the store I stopped in contains a handy kiosk that allows me to find the physical location of CDs in the store. The task we'll be writing sub-tasks or steps for will be "Buy a CD."

*Buy a CD*

**User:** Impatient Buyer

I know what I want and I'm in a hurry. I'll use the Kiosk to help me determine if the item is in stock, how much it costs, and locate it in the store. I'm comfortable with the web and use software frequently.

**Goal:** find and buy a CD quickly

**Context:** busy retail floor with lots CDs, movies, and video games. The store has kiosk software, MediaFinder, that allows me to locate the CD in the store.

| **User Intention** |
|---|
| After locating kiosk indicate the CD title I'd like to find |

| |
|---|
| Determine if the CD is in stock as new, used, or both |
| Determine a price for the CD |
| Determine location of CD in the store and capture it in a way I can that makes it easy for me to find |
| Find the CD, buy it, and leave |

We've just written part of a use case - the part specific to the user of the use case. A use case normally describes the interaction between two or more parties – "actors" in use-case terminology. At each stop we've assumed the kiosk did something sensible to allow us to get to the next step, but we've focused on the user first.

## 3. Identify the product's responsibilities relative to the task steps the user will perform.

Notice the heading "user intention" in our task instructions above. At this point we'd like to capture what the hero, "Impatient Buyer", intends to do. Not necessarily specifically what he does do. Know our user's intention allows us to begin to determine what the kiosk has in way of responsibilities to best help the user.

For each step in the task, list the product's responsibility to help the user move from step to step.

In our example above, the product will be the MediaFinder kiosk.

*Buy a CD*

**User:** Impatient Buyer

I know what I want and I'm in a hurry. I'll use the Kiosk to help me determine if the item is in stock, how much it costs, and locate it in the store. I'm comfortable with the web and use software frequently.

**Goal:** find and buy a CD quickly

**Context:** busy retail floor with lots CDs, movies, and video games. The store has kiosk software, MediaFinder, that allows me to locate the CD in the store.

| User Intention | System Responsibility |
|---|---|
| | Present an easy to find place to enter a CD title |
| After locating kiosk indicate the CD title I'd like to find | |
| | Present a list of CDs that match the title, or if not found, some that closely match the title the user entered<br><br>If nothing similar was found, let the user know |
| Determine if the CD is in stock as new, used, or both | |

| | |
|---|---|
| | For each title found, show how many are in stock, and if they're new or used |
| Determine a price for the CD | |
| | For each title found, show the price of the item<br>If the prices vary for new and used, show that |
| Determine location of CD in the store and capture it in a way I can that makes it easy for me to find | |
| | For each title in stock, indicate the location in the store where it can be found<br>Offer to print a map for one or more of the titles |
| Find the CD, buy it, and leave | |

In the second column under "system responsibility" we've noted the things the system would do in response to what our user did in the "user intention" column. Notice how our system behaves a bit like a person doing something sensible at each process step.

The use case format used here is an Essential Use Case or Task Case - a format described by Constantine & Lockwood in Software for Use. It's a variation on a multi-column format first introduced by Rebecca Wirfs-Brock. The multi column format allows us to easily see what our user does, and what the system does in response. Separating each statement into another row allows us to see the chronological back and forth that's occurring in our user's "conversation" with the system. You could easily merge these into a single column if you wish. Just prefix each statement with "the user will" or "the system will."

## 4. Identify abstract components that help the product meet its responsibilities to the user.

Think of an abstract component as the *idea* of a real user interface component that describes the component's general responsibility; for instance in a typical graphic user interface the user is often presented with the need to make a choice from a series of possible choices. Specific ways to make those choices might include radio buttons, check boxes, or a drop down selection list. All these components afford the selection of a particular choice. If we didn't want to decide just yet what the specific UI component might be, a "choice selector" might be a simple way to abstractly refer to this component.

➢ ***An abstract component refers to a general type of component with a certain responsibility.***

In Constantine, Windl, Noble, & Lockwood's paper *From Abstraction to Realization* they describe the idea of canonical abstract components. They divide components into two general sets of responsibilities: those that contain and present information and those that perform actions. The following symbols are used for each respectively:

□ Container: contains and presents information.

↖ Action: allows execution of an action

Commonly in graphic user interfaces components present information and allow its manipulation. The choice selector mentioned above is just such a component. The two symbols above are easily combined into an actionable container:

☐↖ Actionable Container: contains and presents information and allows the information to be acted on through selection or manipulation.

The paper authors go on to describe a number of canonical abstract components that can be used to construct abstract user interfaces. The canonical abstract components suggested are useful, but most useful is the idea of thinking of UI components in the abstract based on their responsibility/intended usage.

- ***For the Task Case you've built so far, for each system responsibility, decide on an abstract component that would help the system meet its responsibility.***

Do this by noting the name of a canonical component on a post-it note and sticking it directly on your task case. Use a name for the component that makes clear what its responsibility is – like "Title search acceptor." Draw a symbol in the corner of each component if it helps you to remember the nature of its responsibility: container, action, or both. You might find here that having the task case written on a whiteboard, poster paper, or printed large on sheets of paper will help.

| User Intention | System Responsibility | Abstract Component |
|---|---|---|
| | Present an easy-to-find place to enter a CD title | ☐↖ Quick Search Input<br><br>↖ Find titles button |
| After locating kiosk indicate the CD title I'd like to find | | |
| | Present a list of CDs that match the title, or if not found, some that closely match the title the user entered<br><br>If nothing similar was found, let the user know | ☐ List title indicated what was searched for<br><br>☐ List of titles, include artist<br><br>☐ "I couldn't find anything similar to what you were looking for" message |
| Determine if the CD is in stock as new, used, or both | | |
| | For each title found, show how many are in stock, and if they're new or used | ☐ New or Used indicator by title<br><br>☐ In or out of stock indicator by title |
| Determine a price for the CD | | |
| | For each title find, show the price of the item<br><br>If the prices vary for new and used, show that. | ☐ New price by title<br><br>☐ Used price by title |
| Determine location of CD in the store and capture it in a way that makes it easy for me to find | | |

| | | |
|---|---|---|
| | For each title in stock, indicate the location in the store where it can be found<br><br>Offer to print a map for one or more of the titles | ☐ store location by title<br><br>↖ "print me a map to this location" button |
| Find the CD, buy it, and leave | | |

In this example consider each note in the third column a post-it note you might have jotted down while reviewing the task case.

### A helpful product might take on more responsibilities

As you're going through the product's responsibilities you might find more information or actions the system could present or support that you think might help our user succeed.

For example, since the search for the titles could take a couple seconds, a container with a progress bar might be nice so the user knows the request to search was heard and is in progress. The location name within the store might not be enough, showing a store map close by with the locations clearly labeled might be nice.

Think of the product as a very helpful collaborator with our user, then think of any other components to add that might help our collaborative product better succeed in its goal to help our user.

The process of looking at user tasks and considering how the system might best support those activities is a common idea among user interface practitioners. Cooper & Reimann's About Face 2.0 gives a good description of a similar process.

## Additional Reading

- Cockburn, **Writing Effective Use Cases** (Addison-Wesley, 2000)
- Constantine & Lockwood, **Software For Use** (Addison-Wesley, 1999)
- Constantine, Windl, Noble, & Lockwood, **From Abstraction to Realization** (ForUse website: http://www.foruse.com/articles/canonical.pdf)
- Cooper & Reimann, **About Face 2.0** (Wiley, 2003)

## Component to Paper Prototype, Step by Step

Armed with a user tasks and an inventory of candidate user interface components you can begin to arrange the components in a candidate user interface.

## 1. Create candidate interaction contexts

An interaction context is a useful idea from Constantine & Lockwood's Software for Use.

- ***An interaction context is an abstract container for UI components. It has a name and a higher level goal or purpose.***

In computer software an interaction context may be a particular screen or dialog box. However, just like tasks, contexts may contain smaller sub-contexts. For example if you were to look for a product on a typical ecommerce site, you'll find the searched for items appear in a context – let's call it the "found items." But likely next to that search return are contexts that support navigation to other areas of the site, and contexts that show you items you might be interested in. They all occupy the same screen, but contexts are likely clearly separated from each other on that screen and have clearly separate responsibilities.

From the user interface an interaction context switch often happens when the user's goal has changed somewhat substantially. For instance when I first enter an ecommerce website, my first goal might be to find something and the first context I see likely supports that goal. Once I've looked for some things, I may need to look more closely at those things I've found. You'll generally find screens in an ecommerce website that support scrutinizing individual items in more detail. In that context you'll find it better supports those goals by having components and a layout to do so. Contexts may appear adjacent to each other in the user interface or the user may navigate from one context to the next.

Using the task case we've written so far, look for goal changes that might indicate a context change. In this task case starting to search might be one goal, then evaluating what was found might be another big goal. Let's start with two contexts that support those goals. Give those contexts a name, and note their name and goals on a post-it. Stick the post-it on a sheet of paper.

**Starting Point:** give the user a clear starting point for starting a search for titles in the store.

**Search Return Evaluation:** help the user decide if the searched for items were the items she was looking for or an easy way to reinitiate the search if not. Also aid in the quick decision to buy any successfully found item.

## 2. Transfer abstract components to candidate interaction contexts

For each abstract component, transfer it to the interaction context that best matches the goal that the component is helping the user reach.

You might find that some components belong in multiple contexts. If so, write up an additional post-it and place it there.

You'll now have sheets of paper representing your named interaction contexts, each of those with an inventory of post-it note components.

## 3. Arrange components in interaction contexts according to use

At this stage we'll begin to think a bit more concretely about the user interface.

If you're reading this, and at this point I strongly suspect you are, you're likely reading from top to bottom and from left to right. Typically, software user interfaces also read from top to bottom and left to right.

Look back at the task case we've written. Arrange the component post-its in a logical spatial arrangement that allows the user in the task case to encounter each component in a logical order – from top to bottom and from left to right. You should be able to start imagining a user interface screen in your head.

## 4. Validate the abstract contexts and components against the user task(s)

To make sure these components and their arrangement are making sense, let's check what we have so far against our task case.

Place yourself in the role of the user. Think about the user's likely skills, goal and context of use as you do this. Looking at the first interaction context the user would encounter then go through each step in the task case. At each step imagine using the abstract component to accomplish your intention. Assume each component does its job well.

As the task case's user, make sure you have every component you need as you step through the task case. Make sure the component appears where you expect it to in the user interface.

- If you encounter missing components, add them.
- If some components don't get used, consider removing them.
- If the intention of some components could be better stated, rename them.

## 5. Convert each abstract component to a user interface component

At this stage it's time to think much *more* concretely about user interface.

For each abstract component make a preliminary choice about what that component could be. Draw the component in pencil on heavy paper – an index card or card stock works well here. Draw the component the approximate size that you believe it should be in the interaction context. You may need to cut the interaction context down to size if you believe it will fall inside some other interaction context. Place the component on the interaction context in the place where you believe it should be.

Once all components are arranged in an interaction context inspect the layout. You may find you need to recreate or resize some of the components. You may find that drawing boxes or lines directly on the paper for the interaction context helps your layout a bit.

When you feel you've got all the components represented well for now, stick them down with repositionable double-sided tape.

You've now built a componentized testable paper prototype.

You'll find lots of great ideas on materials and approaches for building and testing paper prototypes in Snyder's Paper Prototyping.

## 6. Revalidate the paper contexts and components against the user task(s)

As you did when interaction contexts were sheets of paper and components were post-it notes, assume the role of the user and step through the task case.  As when components were post-its, add, remove, or change components so that they support the task case effectively.

### Additional Reading

- Constantine & Lockwood, **Software For Use** (Addison-Wesley, 1999)
- Snyder, **Paper Prototyping**, (Morgan-Kaufmann, 2003)

## Usability Testing a Paper Prototype, Step by Step

On a typical software project, finished software is tested by testers to make sure it doesn't have functional errors. However, for most software to be considered to be a success, the application must not only be free of bugs, but must be easily used by its primary user constituencies. Testing for this quality is done through usability testing. And, happily, basic usability testing can be done ahead of building the actual software.

If you've built a paper user interface prototype, you've considered the users and their probable usage, and selected screens and components that best support that usage. However, real users are a bit unpredictable. Validating the design on paper with real candidate users will help give you more confidence that your candidate user interface really is meeting its goals, and that your software really will be usable.

### 1. Select test participants

Identify people you could use for your test. Ideal candidates will match the characteristics of the intended audience for the software. The more critical the success of the finished product, the more appropriate it is to locate users that best represent the audience of your product. Snyder's Paper Prototyping gives great guidance on finding and identifying candidate users for usability tests.

However, if you're at a preliminary stage with you user interface design, it's very valuable to quickly find someone that, while not ideal, can help to refine the design you might place in front of a more ideal candidate. For this purpose select individuals from within your office, friends, or family. People close by that you can coach a little on how to best help your user interface design effort. We'll call these sorts of test subjects user stand-ins.

Identify some user stand-ins. For preliminary testing of a user interface design, 2-4 is sufficient.

Coach user stand-ins by explaining to them:

- the purpose of your product
- who the target users are, and what sorts of characteristics they're likely to have
- where the product will likely be encountered and characteristics of that environment

### 2. Identify tasks to test

Identify the functional level tasks you'd like to validate for your user interface prototype. A typical user interface is built to support a number of functional level tasks. Identify the tasks you'd like validated with your candidate user interface.

For the example we've been building "Find a CD" was the primary task we'd started with. We may want to consider related tasks that our user interface might also accommodate:

- Locate a title from an artist I know
- Browse new arrivals
- Browse titles similar to those I already own and like
- Browse items on sale

For each task consider alternative conditions, exceptions, or errors you might also want to test. For example:

- Title doesn't exist
- Artist doesn't exist
- Title was found, but not in stock

## 3. Identify tester roles

For an effective usability test, in addition to test subjects you'll need to fill three primary roles in a usability testing team:

The **facilitator** will be responsible for interacting with the test subject and directing the usability test. The facilitator will set up tasks being tested by describing some starting context then naming the task and the goal. The facilitator should not provide any other instruction on performing the task.

The **computer** will control the paper prototype acting as the computer responding to gestures and verbal commands issued by the test subjects. The computer must not speak. Even if directly spoken to, smile and respond by reminding the test subject that "computers can't talk."

The **observers** will be responsible for quietly and unobtrusively recording results for the test.

Use only one person on the facilitator role. One or two people working together may play the computer role. One to four may fill the observer role.

Combining roles in one person is difficult. But if people are unavailable to fill the roles, combining the facilitator and computer role can work. Combining the facilitator and observer role can also work but is less effective. Combining all three roles in one person is a bad idea.

### Keep three personalities in mind when performing a usability test:

- The flight attendant

  A flight attendant's job is to both provide service to his or her passengers and to keep them safe. The facilitator generally assumes a flight attendant role to make sure the user has proper instruction and feels safe. If the user makes missteps while testing the user interface, make sure they understand that it's not them being tested, but the user interface. Make sure they don't feel foolish.

- The sportscaster

  The sportscaster's job is to make sure that everyone watching or listening to the action knows what's going on. The facilitator balances this personality with the flight attendant personality. While observers are quietly taking notes they may not be able to see specifically what the user is doing or what's happened in the user interface. The facilitator in the sportscaster personality maintains a dialog with the test subjects with the intent of making sure everyone in the room understands the play-by-play of what's going on in the usability test.

- The scientist

  Everyone assumes the scientist personality. Observers, computer, and facilitator all strive to ensure they're getting accurate information about how effectively users are able to reach their goals in the candidate user interface. While eliminating all bias is impossible, the facilitator works to eliminate bias by not leading or suggesting to the users they follow any particular path through the user interface. The computer tries not to hint to the users where to click next. Even reaching for a guess at what the next component may be might inadvertently hint the user on what's next. Observers must sit and record quietly. Non-verbal gestures, sighs, gasps, or groans all contribute to influencing the user to behave or not behave in a particular way. Avoid these behaviors.

Choose the team that will run the usability test. Together decide on who will perform in what role. Review and make sure everyone understands the roles and the three important personalities they all need to be aware of.

## 4. Set up test facility

Chose a test location where all participants including observers, computer, facilitator, and test subjects can sit around a table or work area.

Position test subjects on one side or end of the table.

Position the computer directly across from the test subjects. Tape the background of the user interface down to the table so it doesn't move during the test. Set up the screen to its starting point. Place all the other components you'll need close by. Consider placing them in a folder so they're not visible to the test subjects.

The facilitator should sit to one side of the test subjects.

The observers should sit at available places around the table with notepads or post-it notes ready to take notes.

## 5. Perform tests

Invite users in to the test.

You may perform the test with one test user or two users working together as a pair.

Generally explain the product being tested to the users. Explain to them the test goals of determining if the proposed user interface is effective at helping users meet their goals. Inform them they'll be asked to accomplish a few tasks with the proposed user interface.

Explain to the test subjects how to use the paper prototype and your very low-tech computer. They'll use their finger as a mouse and point directly to objects on the screen and indicate that they wish to click, right-click, or double click on them. If they need to enter data, ask them to write directly on the prototype using a marker. [The computer should cover input fields with removable tape or transparency film to allow users to directly write on the user interface.]

Ask the users to "think out loud." When they first look at the screen ask them to comment on what they see. As they move to perform an action in the user interface, ask them to comment on what they're about to do, why, and what they expect to happen. Pairing two users allows this to happen naturally. Two paired users will generally discuss with each other what they're seeing and what to do next.

Introduce the first task and goal to the test subjects and ask them to begin.

As users step through the user interface the facilitator should offer guidance to the test subjects without suggesting how to use the user interface. This is the flight attendant. The facilitator also keeps conversation active suggesting users comment on what they're doing or what they see or asking them directly. This is the sports caster.

As users step through the prototype observers should write down any observations they have about the user interface. What seems to be working, along with errors, missteps, or confusion the user has with the UI.

When the users successfully achieve the goal that completes the task, the facilitator may stop the test. Alternatively if the users just can't complete the task they can stop the test explaining that, "It looks like we've got quite a bit or work do to on this user interface for it to work effectively. Let's stop the test for now. We've gathered lots of helpful information to help us improve the design." This is the flight attendant speaking.

The facilitator then asks the observers if they have any questions for the test subjects.

The observers, silent until now, ask question being careful not to bias the users or imply that they should have done things differently. This is the scientist and flight attendant.

The facilitator then introduces the next task and goal while the computer quickly resets the user interface if needed.

Continue with the testing of each task until all tasks have been tested or time runs out. One, to one-and-a-half hours is a good time limit for a usability test. This can be tiring work for everyone involved.

After the test is complete, thank the subjects and let them leave. If you're using test subjects from outside your company, it's customary to pay the test subjects or give them a gift of some type to compensate them for their time.

The test team then needs to discuss what they've observed and either respond to feedback immediately before the next test subjects show up or elect to consolidate results later and respond later that day or on a subsequent day.

## 6. Respond to feedback immediately

One advantage of a paper prototype is the ease in which it can be altered. If there are obvious issues with the prototype, there's no need to wait for a next test subject to predictably encounter them. As a team discuss what the big problems were with the test they just observed. Agree on changes to make and alter the paper prototype before the next test subjects arrive.

## 7. Consolidate results and respond

After a few usability tests the observers will have accumulated a number of notes regarding the issues they observed users encounter. We now need to consolidate those notes from multiple interviews to make decisions about where to change the user interface prototype.

Transfer observer notes to post-its – one observation per post-it. Each observer combines their post-its with the others on a wall or table top. Notes that describe similar problems should be placed near each other. Dissimilar notes farther apart. At the end of this consolidation exercise you'll find you have many clusters of notes. Some of the clusters will likely be bigger than others. These are likely the areas with the most severe problems. This type of model is referred to as an "affinity diagram" since the information in it is clustered by affinity.

For each cluster write another post-it note summarizing the contents of the cluster. Use a different colored post-it than the others.

For each summarized cluster, starting from the largest clusters first, decide as a team how best to change the user interface to rectify the problem.

Perform adjustments to the paper prototype and prepare for your next usability test.

## Additional Reading

- Snyder, **Paper Prototyping**, Morgan-Kaufmann, 2003

## User Story Driven Development Processes and User Interface Design

From the time when they were originally described in Extreme Programming Explained, User Stories have become popular as an approach to representing, estimating, and planning requirements in Agile Software Development Processes. The precise definition of a user story varies on the source. Beck's original Extreme Programming Explained describes a user story as a sentence or two written on an index card as a reminder for a future conversation between developer and customer. In User Stories Applied, Mike Cohn puts quite a bit more detail around how you might create and use User Stories throughout an Agile development process. Mike also credits Rachel Davies with a particularly useful story writing form that can work well to combine elements useful for user interface design.

Rachel suggests we write stories like this:

As a *[type of user]*

I want to *[perform some task]*

so that I can *[achieve some goal.]*

Using this story writing form we identify types of users, tasks they wish to perform, and the goal they wish to achieve. In our CD buying example a story for the software we might write could read like this:

As a **harried shopper**

I want to **locate a CD in the store**

so that I can **purchase it quickly, leave, and continue with my day**.

If we were to write the name of such a story as concisely as possible to place in a user story list, or backlog, we might use the simple name: "**locate a CD**" which also happens to be a good task name.

## User stories often describe the tool, not the user and task

In this example we've been describing the software we'd like to build from the perspective of the user who needs to perform particular tasks. However, sometimes we get tempted to describe the way the finished software might look or behave. For example we could just as easily have written the story this way:

As a **harried shopper**

I want to **enter the CD name in the search box**

so that I can **view the CD location in the store from the search return**.

We might then be tempted to concisely describe the story as "**CD search screen**." While more precise, this story has already made the decision that there is a search box where I can type in the CD name and that the CD location will appear as part of the search return. Much of the user interface design has already been pre-supposed in this story. Rather than describing the task abstractly the story describes the user using the software, or the "tool," in a specific way.

While with this particular user interface the use of a typical search box, search button, and search return list may seem like an obvious and adequate solution, with some other design problems the solution will be far less obvious. And, capturing suppositions about the design solution inside your user story may make it likely you and your team may not consider alternative viable design solutions later during a UI prototyping and testing effort.

## Estimable user stories are often low level

A user story as a form of software requirement for Agile Development often carries the constraint that developers be able to complete the construction of story functionality in a single development cycle, or iteration – generally one to three weeks. As Agile Development matures the idea of smaller user stories increases in popularity. Currently it's common for a user story to be no larger than a single developer can complete in less than three days.

Smaller user stories are easier to estimate, quicker to develop, and more accurate as a tool to measure project progress. However, the smaller and more estimable user stories get, the less likely they are to be task-centric, and the more likely they are to describe specific elements of the finished software. If they are task-centric they're more likely to be written at a user goal level too low (below functional level) for easily designing and validating user interface. Generally speaking, smaller, more granular user stories often describe proposed details of the user interface or small user sub-tasks such that it becomes difficult to see how and why a user might use the finished software. This makes the user interface difficult to design.

If your project is moving to smaller user stories, you may find you'll have to maintain task models other than user stories alone to understand your users and their intended usage of your software. You may also find you'll have to engage in user interface design, prototyping, and testing earlier as part of initial story writing and release planning. These smaller more detailed stories represent a more detailed understanding of your software. You'll have to do more work early to gain this level of detailed understanding instead of letting that understanding emerge or be discovered over time.

- ***When possible, scope your project using functional level, task-centric user stories. Defer user interface design, prototyping, and testing activities till the latest responsible moment.***

Poppendiek and Poppendiek use the term "latest responsible moment" in their book Lean Software Development to describe a strategy of deferring decisions until we know as much as possible about the problem we're solving. Jim Shore puts it well it his blog: "…it's the latest responsible moment, not the *last possible* moment. That wouldn't be responsible."

## Prototyping and testing user stories may result in more user stories

It's often easy to use pencil, paper, and tape to prototype and test a user interface that works very well. But, then when it comes time to estimate the cost of building such a user interface, the cost is prohibitively expensive – or at least more than we want to stomach right now.

I've found it makes good sense to have developers involved in the prototyping and testing effort both to gain a better understanding of what works well with end users, and to help understand the development cost of what we're prototyping and testing. If the cost is high, it's worth prototyping and testing variations of the user interface that, while they may be a little less ideal from a user's perspective, could be significantly less expensive to develop. It's possible to write a first user story that requests the building of the lower cost user interface, and a subsequent story to be scheduled for a later release that upgrades the UI to the better performing version.

Alternatively we might decide an expensive user interface is economically feasible for our product, but we just don't believe it's a good idea to release it into development as one large user story. In these situations the user interface may need to be broken into smaller stories that describe portions of the functionality such that as each portion is implemented it can be seen, demonstrated, and tested in the finished software.

In both of these examples a task-centric user story may need to be split into subsequent user stories that may represent different versions of the same user completing the same task, or versions that allow that same user to only complete a fraction of their intended task. In both of these cases the splitting comes as a consequence of better understanding the user interface after prototyping and testing.

As you prototype and test your user interface be prepared for surprises that may cause the addition of or splitting of user stories in your Agile project.

Combining a solid user interface design approach with a strong story-driven Agile development approach is not without challenges. There aren't hard and fast solutions to many of the complications that might arise. Stay alert and be prepared to improvise and compromise.

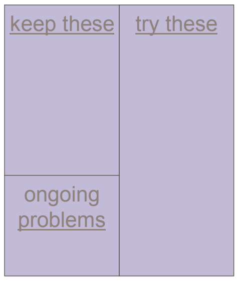
## Additional Reading

- Beck, **Extreme Programming Explained**, Addison-Wesley, 1999
- Beck, **Extreme Programming Explained 2nd Edition**, Addison-Wesley, 2004
- Cohn, **User Stories Applied**, Addison-Wesley, 2004
- Poppendiek & Poppendiek, **Lean Software Development**, Addison Wesley, 2003
- Shore, **Beyond Story Cards**, http://www.jamesshore.com/Multimedia/Beyond-Story-Cards.html, 2005

Notes

## Use Frequent Reflection Sessions To Adaptively Adjust Your Process

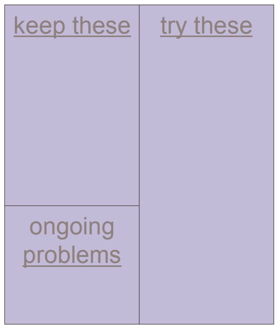
- In Agile Development approaches reflection sessions, or retrospectives, are commonly held at the end of development cycles – iteration or release
- Before engaging in a reflection session:
  - evaluate the **product** you've built thus far against its goals
  - evaluate your team's **progress** building the product relative to plans made
- The goal of the reflection session is to identify opportunities to alter the **process** the team follows to ultimately improve the quality of the product and/or the pace of progress the team is making
- A common reflection session *smell* is to identify what worked, and what didn't work
  - Reflection sessions may turn into complaint sessions where the only goal may be to assign blame
- In a simple reflection session identify methodology characteristics:
  - to keep doing
  - to try doing
  - that are ongoing problems



96

## Exercise: Perform a Reflection Session On Your First Release

- Start by quickly measuring your progress: how many tasks did you end up supporting in your product? Was that more or less than you expected to support?
- Discuss your product: how well did you fare on usability testing?
- Draw the simple 3 sectioned chart pictured
- Populate each section with characteristics of your approach that:
  - You liked doing and want to keep doing next release
  - Things to try doing differently next time
  - Ongoing problems that you don't see clear solutions to



10 minutes

97

## Reflective Process Improvement

### Reflect and Generate Ideas for Process Change

A necessary part of improving a process is looking back reflectively at how that process has worked historically.

Sometimes the process of looking back results in complaints on what didn't work without any clear path to improvement.

Reflection sessions can often turn into gripe sessions or never-ending informal discussion. As with most activities it's good to engage in a reflection session with a goal in mind for what you expect to get out of it. Some of the most valuable things to come from a reflection session are the suggestions for process improvement. Starting with this goal in mind helps focus a reflection discussion.

It's easy to focus too much on negative aspects and improvement. At times we forget to focus on what's working well. If we fail to call out and discuss what's working well, we run the risk of forgetting it and inadvertently dropping the good parts of our process.

We can't always solve all problems, but that doesn't mean we want to avoid them. Often it's important to keep track of problems we've discussed, but that have no easy solution. We need to remember these and keep thinking about them in order to generate future solutions.

Use a reflection session that captures three categories of details: what's working well, suggestions for improvement, and issues to table for future discussion.

To run a reflection session start by choosing a neutral moderator. Borrow a member from another group, a business associate or friend – one with a good sense of humor and good penmanship.

Hang sheets of flip chart paper on the wall. Label one sheet "works well," one sheet "things to try," and another sheet "issues."

The moderator should start the reflection session by asking what worked well, and writing down the ideas that come from the group. When ideas slow down, the moderator should poll the people who didn't contribute ideas. This helps everyone feel involved.

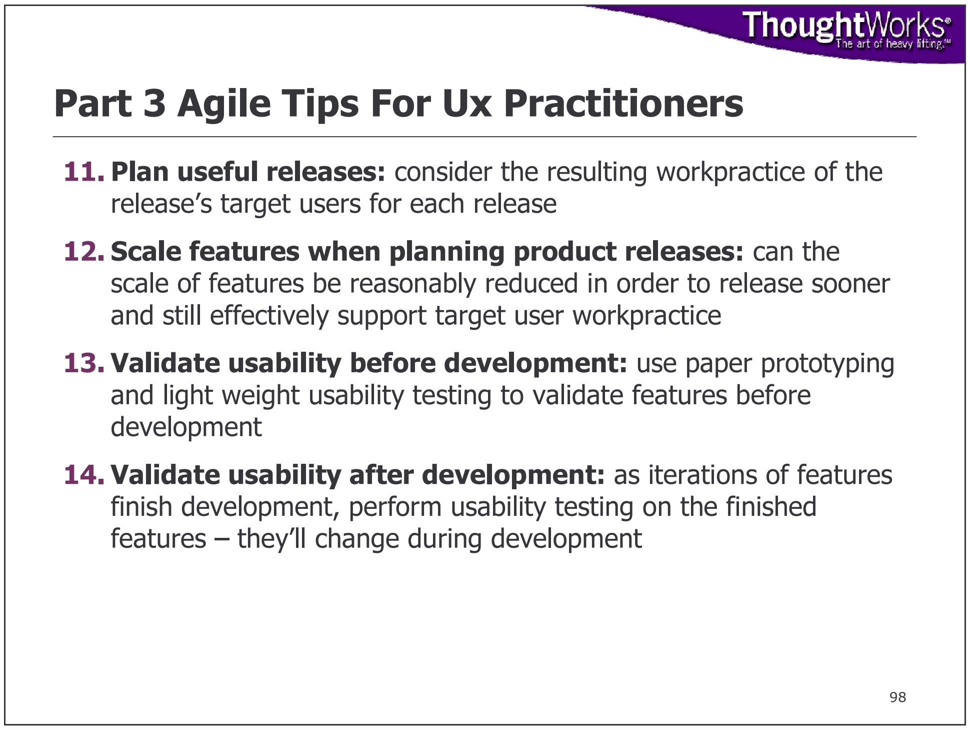
Then the moderator should ask for suggestions about changes in the process to try and write these down as participants say them. The moderator should again poll people who don't volunteer information.

As the discussion proceeds, participants will naturally discuss problems. Give those discussions space to happen. When the discussion starts to slow, it's a good time to ask "can anyone think of something to try to solve this problem? Or, should we note it as an issue to take up later?"

Use the three flip chart categories to capture these most important deliverables of your reflective process improvement session.

## Additional Reading

- Cockburn, **Crystal Clear : A Human-Powered Methodology for Small Teams** (Addison-Wesley, 2004)
- Derby & Larsen, **Agile Retrospectives: Making Good Teams Great** (Pragmatic Bookshelf, 2006)
- Kerth, **Retrospectives: A Handbook for Team Reviews** (Dorset House, 2001)



Notes

## Part 3 Agile Tips For Ux Practitioners

11. **Plan useful releases:** consider the resulting workpractice of the release's target users for each release
12. **Scale features when planning product releases:** can the scale of features be reasonably reduced in order to release sooner and still effectively support target user workpractice
13. **Validate usability before development:** use paper prototyping and light weight usability testing to validate features before development
14. **Validate usability after development:** as iterations of features finish development, perform usability testing on the finished features – they'll change during development

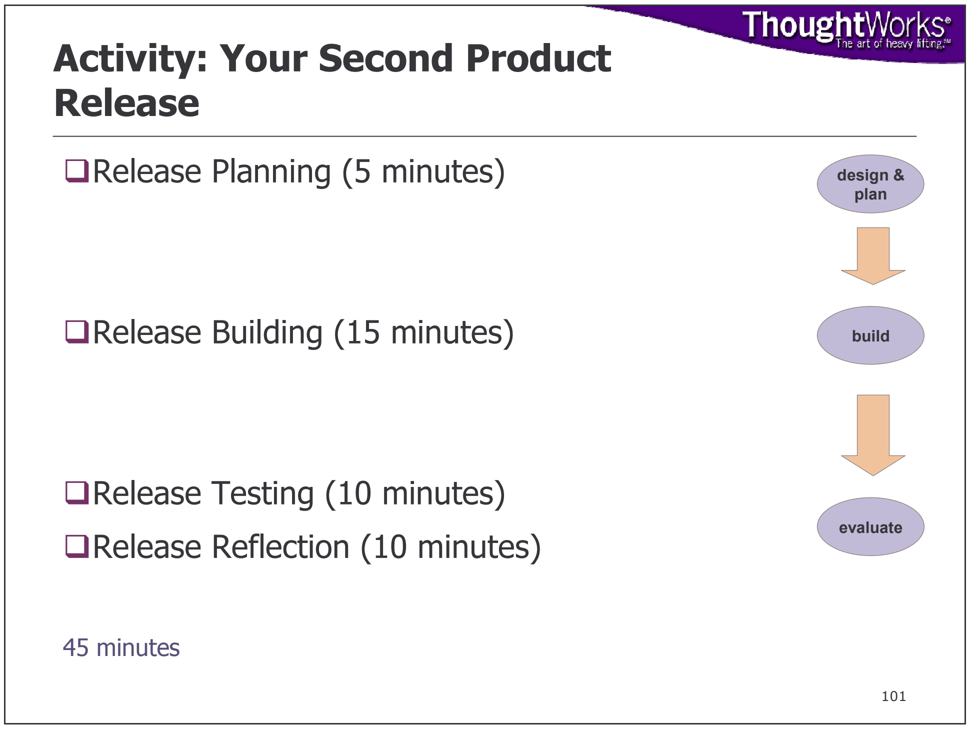
98

## Part 4: Adapting & Thriving

Because no one ever gets things right the first time, and Agile development places big emphasis on adaptation, we'll make adjustments to the software we built in our first release. We'll also add additional features to create a better second release.

We'll close the day by talking about a few organizations that have successfully adopted Agile Development and strong User Centered Design practices. We'll discuss common attitudes and approaches they share that have allowed them to be successful.

Notes



ThoughtWorks®
The art of heavy lifting.℠

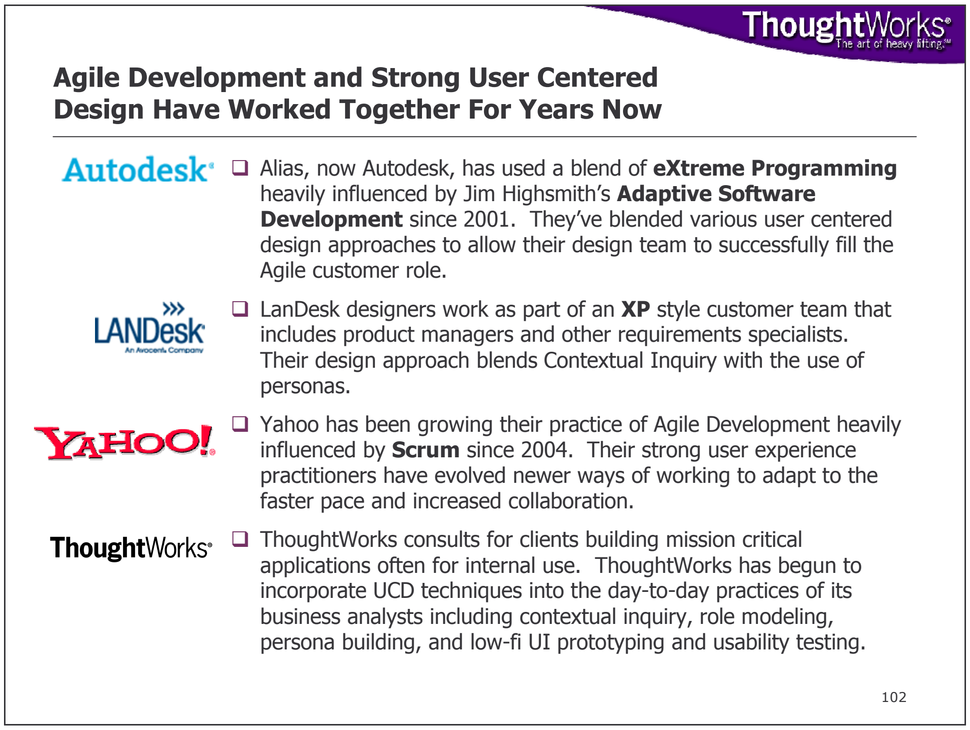
### Activity: Your Second Product Release

- ❑ Release Planning (5 minutes)
- ❑ Release Building (15 minutes)
- ❑ Release Testing (10 minutes)
- ❑ Release Reflection (10 minutes)

design & plan
build
evaluate

45 minutes

101



ThoughtWorks® The art of heavy lifting.℠

## Agile Development and Strong User Centered Design Have Worked Together For Years Now

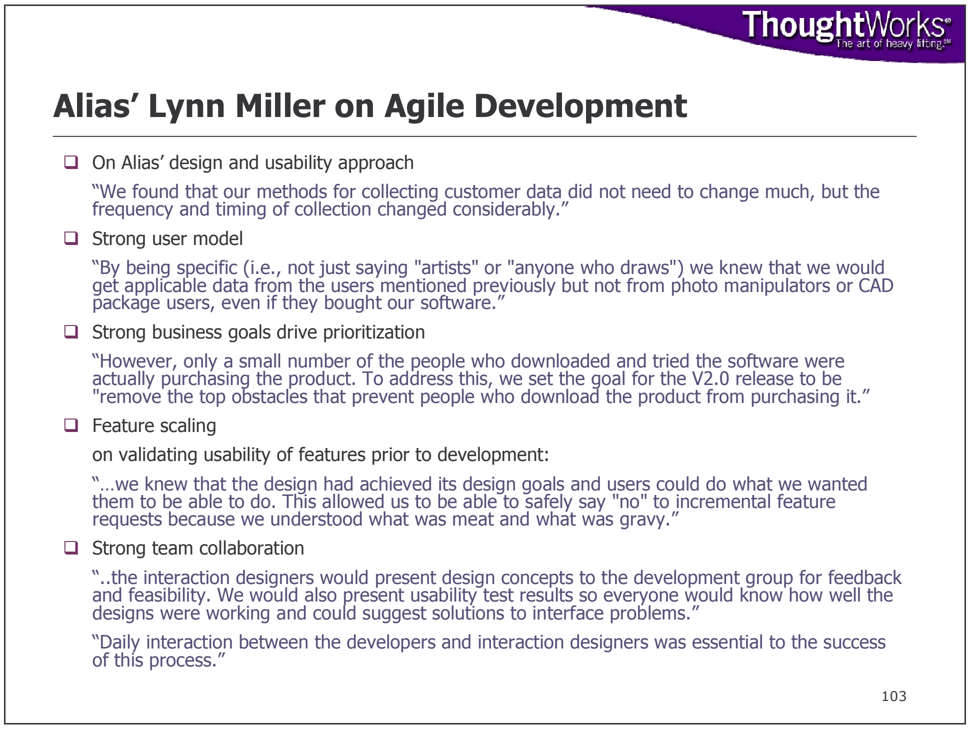
Autodesk®
- Alias, now Autodesk, has used a blend of **eXtreme Programming** heavily influenced by Jim Highsmith's **Adaptive Software Development** since 2001. They've blended various user centered design approaches to allow their design team to successfully fill the Agile customer role.

LANDesk® An Avocent Company
- LanDesk designers work as part of an **XP** style customer team that includes product managers and other requirements specialists. Their design approach blends Contextual Inquiry with the use of personas.

YAHOO!®
- Yahoo has been growing their practice of Agile Development heavily influenced by **Scrum** since 2004. Their strong user experience practitioners have evolved newer ways of working to adapt to the faster pace and increased collaboration.

ThoughtWorks®
- ThoughtWorks consults for clients building mission critical applications often for internal use. ThoughtWorks has begun to incorporate UCD techniques into the day-to-day practices of its business analysts including contextual inquiry, role modeling, persona building, and low-fi UI prototyping and usability testing.

102



ThoughtWorks® The art of heavy lifting.℠

## Alias' Lynn Miller on Agile Development

- On Alias' design and usability approach

  "We found that our methods for collecting customer data did not need to change much, but the frequency and timing of collection changed considerably."
- Strong user model

  "By being specific (i.e., not just saying "artists" or "anyone who draws") we knew that we would get applicable data from the users mentioned previously but not from photo manipulators or CAD package users, even if they bought our software."
- Strong business goals drive prioritization

  "However, only a small number of the people who downloaded and tried the software were actually purchasing the product. To address this, we set the goal for the V2.0 release to be "remove the top obstacles that prevent people who download the product from purchasing it."
- Feature scaling

  on validating usability of features prior to development:

  "…we knew that the design had achieved its design goals and users could do what we wanted them to be able to do. This allowed us to be able to safely say "no" to incremental feature requests because we understood what was meat and what was gravy."
- Strong team collaboration

  "..the interaction designers would present design concepts to the development group for feedback and feasibility. We would also present usability test results so everyone would know how well the designs were working and could suggest solutions to interface problems."

  "Daily interaction between the developers and interaction designers was essential to the success of this process."

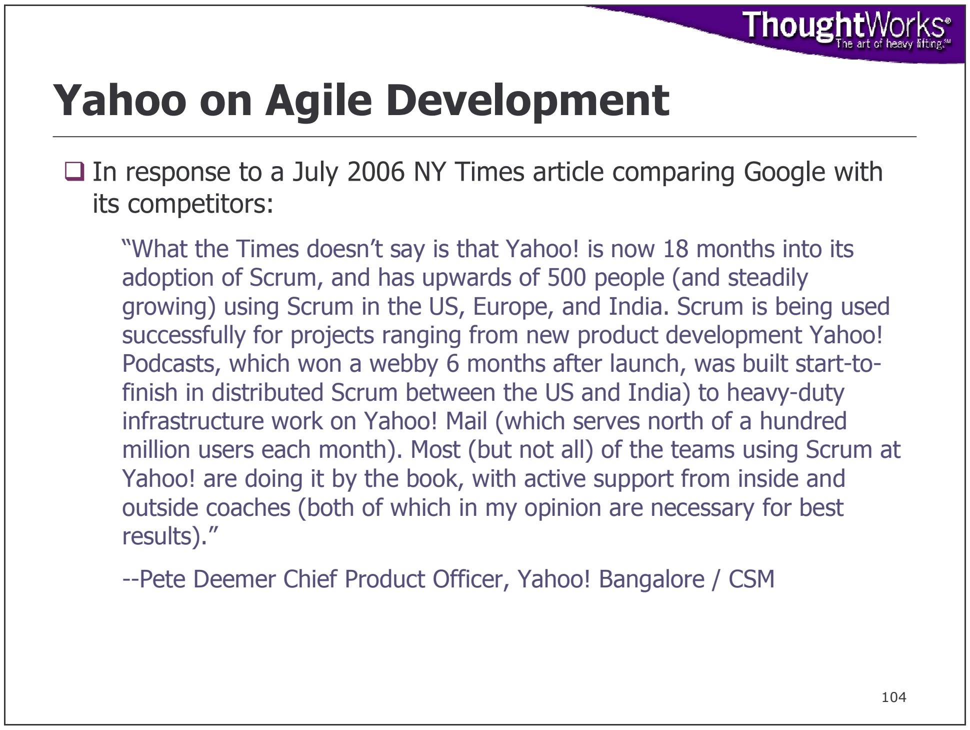
103

## Notes

### Additional Reading:

- Miller, **Case Study of Customer Input for a Successful Product** (Agile 2005 practitioner report, http://www.agile2005.org/XR19.pdf)
- Baker, Beyer, & Holtzblatt, **An Agile Customer Centered Method: Rapid Contextual Design** (XP/Agile Universe, 2004, http://www.incontextdesign.com/resource/pdf/XPUniverse2004.pdf)
- Hansell, **In the Race With Google, It's Consistency vs. the 'Wow' Factor** (NY Times, July 2006)
- Abilla, **Scrum at Yahoo** (http://www.shmula.com/159/scrum-at-yahoo)

Notes



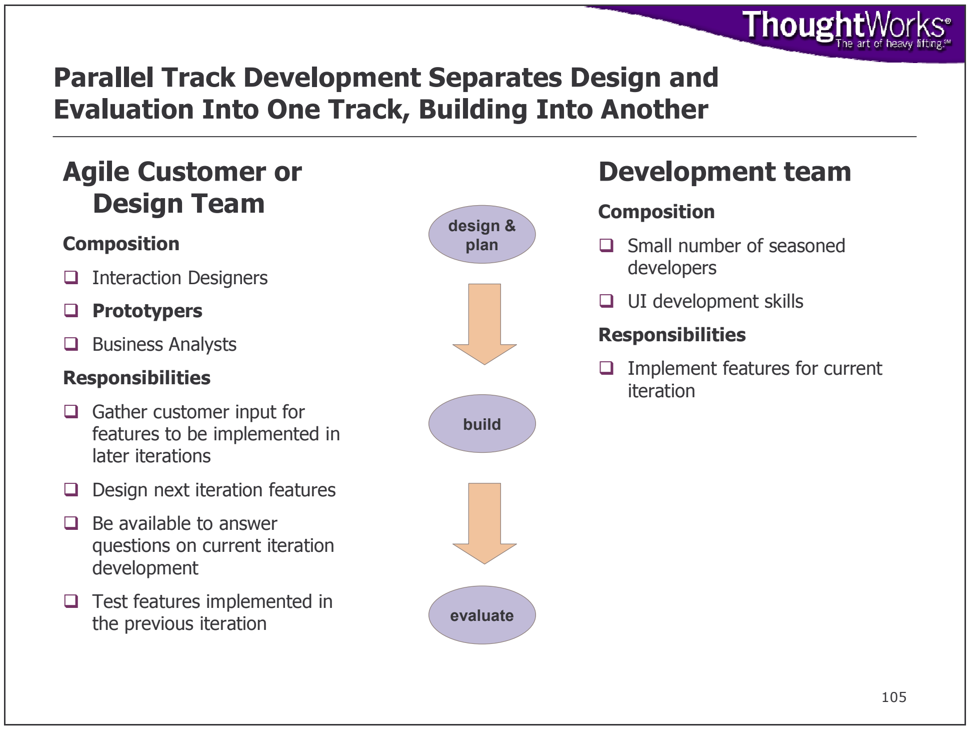
## Yahoo on Agile Development

- In response to a July 2006 NY Times article comparing Google with its competitors:

  "What the Times doesn't say is that Yahoo! is now 18 months into its adoption of Scrum, and has upwards of 500 people (and steadily growing) using Scrum in the US, Europe, and India. Scrum is being used successfully for projects ranging from new product development Yahoo! Podcasts, which won a webby 6 months after launch, was built start-to-finish in distributed Scrum between the US and India) to heavy-duty infrastructure work on Yahoo! Mail (which serves north of a hundred million users each month). Most (but not all) of the teams using Scrum at Yahoo! are doing it by the book, with active support from inside and outside coaches (both of which in my opinion are necessary for best results)."

  --Pete Deemer Chief Product Officer, Yahoo! Bangalore / CSM

104



## Parallel Track Development Separates Design and Evaluation Into One Track, Building Into Another

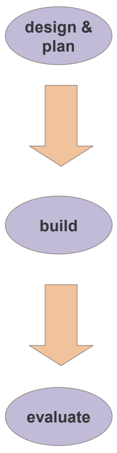
### Agile Customer or Design Team

**Composition**

- Interaction Designers
- **Prototypers**
- Business Analysts

**Responsibilities**

- Gather customer input for features to be implemented in later iterations
- Design next iteration features
- Be available to answer questions on current iteration development
- Test features implemented in the previous iteration



### Development team

**Composition**

- Small number of seasoned developers
- UI development skills

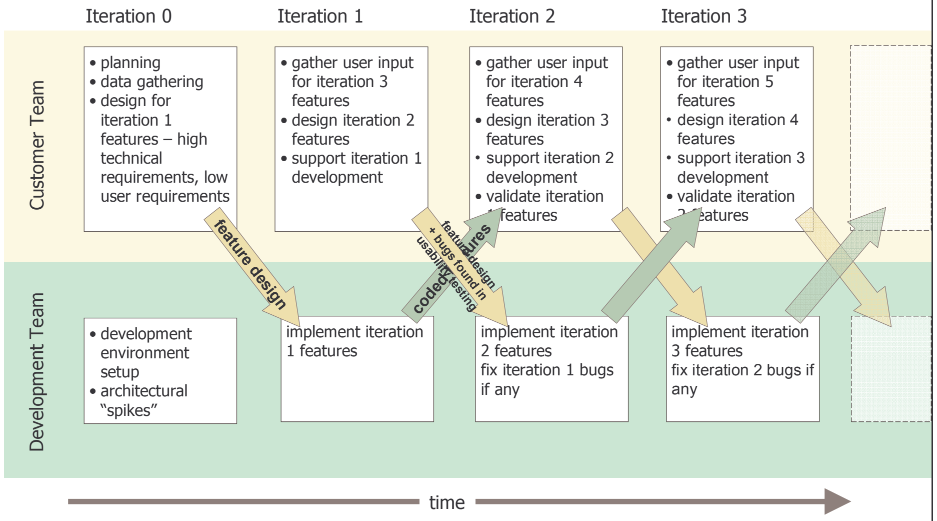
**Responsibilities**

- Implement features for current iteration

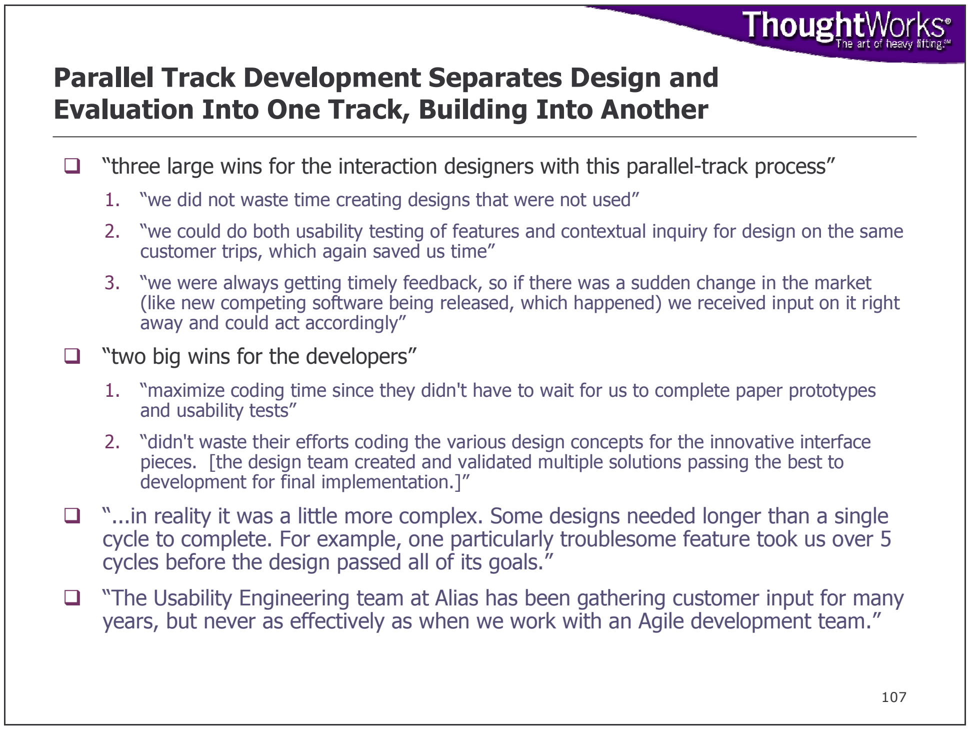
105

Notes

## Design and Coded Features Pass Back and Forth Between Tracks


Iteration 0
Iteration 1
Iteration 2
Iteration 3
Customer Team
Development Team
• planning
• data gathering
• design for iteration 1 features – high technical requirements, low user requirements
• gather user input for iteration 3 features
• design iteration 2 features
• support iteration 1 development
• gather user input for iteration 4 features
• design iteration 3 features
• support iteration 2 development
• validate iteration 1 features
• gather user input for iteration 5 features
• design iteration 4 features
• support iteration 3 development
• validate iteration 2 features
feature design
coded features
feature design + bugs found in usability testing
• development environment setup
• architectural "spikes"
implement iteration 1 features
implement iteration 2 features fix iteration 1 bugs if any
implement iteration 3 features fix iteration 2 bugs if any
time

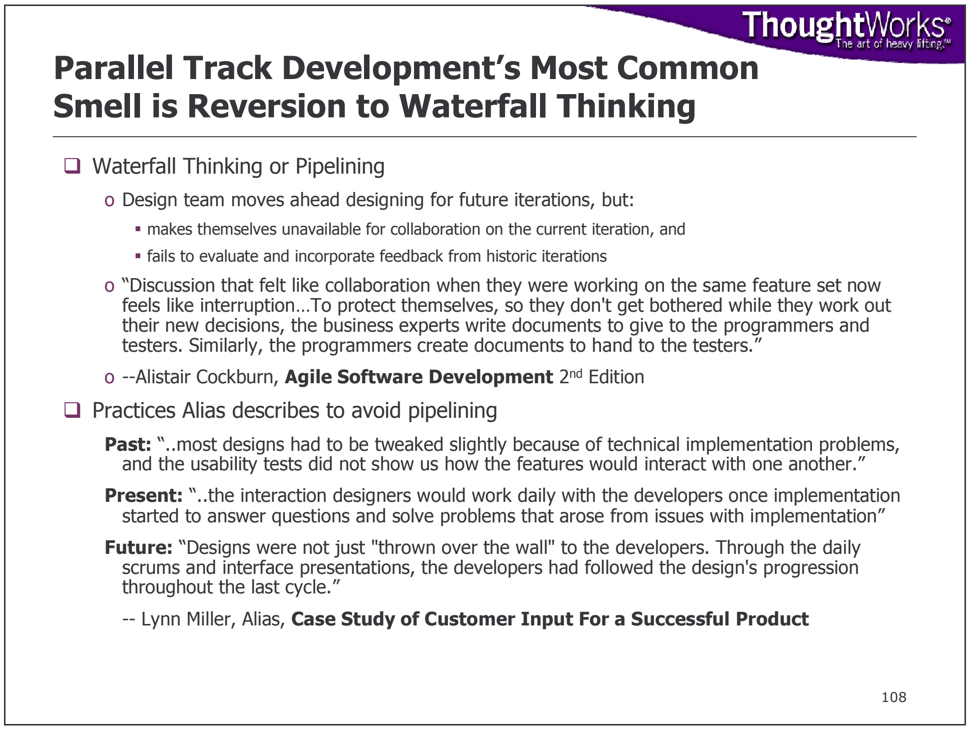
106



## Parallel Track Development Separates Design and Evaluation Into One Track, Building Into Another

- "three large wins for the interaction designers with this parallel-track process"
  1. "we did not waste time creating designs that were not used"
  2. "we could do both usability testing of features and contextual inquiry for design on the same customer trips, which again saved us time"
  3. "we were always getting timely feedback, so if there was a sudden change in the market (like new competing software being released, which happened) we received input on it right away and could act accordingly"
- "two big wins for the developers"
  1. "maximize coding time since they didn't have to wait for us to complete paper prototypes and usability tests"
  2. "didn't waste their efforts coding the various design concepts for the innovative interface pieces. [the design team created and validated multiple solutions passing the best to development for final implementation.]"
- "...in reality it was a little more complex. Some designs needed longer than a single cycle to complete. For example, one particularly troublesome feature took us over 5 cycles before the design passed all of its goals."
- "The Usability Engineering team at Alias has been gathering customer input for many years, but never as effectively as when we work with an Agile development team."

107



## Parallel Track Development's Most Common Smell is Reversion to Waterfall Thinking

- Waterfall Thinking or Pipelining
  - Design team moves ahead designing for future iterations, but:
    - makes themselves unavailable for collaboration on the current iteration, and
    - fails to evaluate and incorporate feedback from historic iterations
  - "Discussion that felt like collaboration when they were working on the same feature set now feels like interruption…To protect themselves, so they don't get bothered while they work out their new decisions, the business experts write documents to give to the programmers and testers. Similarly, the programmers create documents to hand to the testers."
  - --Alistair Cockburn, **Agile Software Development** 2nd Edition
- Practices Alias describes to avoid pipelining

  **Past:** "..most designs had to be tweaked slightly because of technical implementation problems, and the usability tests did not show us how the features would interact with one another."

  **Present:** "..the interaction designers would work daily with the developers once implementation started to answer questions and solve problems that arose from issues with implementation"

  **Future:** "Designs were not just "thrown over the wall" to the developers. Through the daily scrums and interface presentations, the developers had followed the design's progression throughout the last cycle."

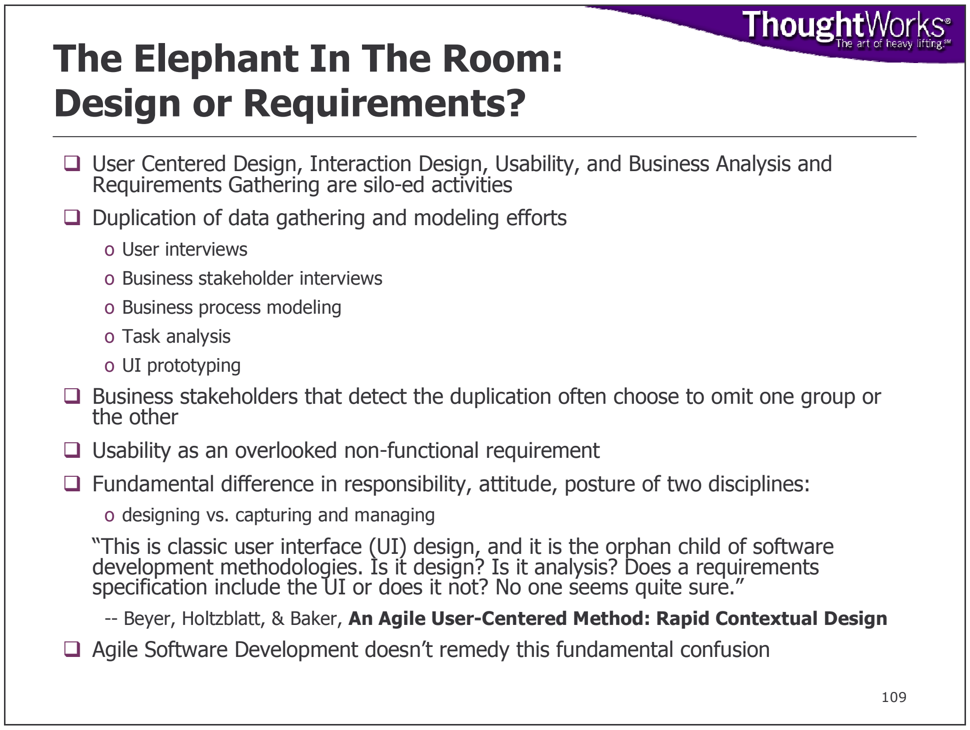
  -- Lynn Miller, Alias, **Case Study of Customer Input For a Successful Product**

108

Notes

**Additional Reading:**

- Cockburn, **Are iterations hazardous to your project?** (http://alistair.cockburn.us/index.php/Are_iterations_hazardous_to_your_project%3F)



## The Elephant In The Room: Design or Requirements?

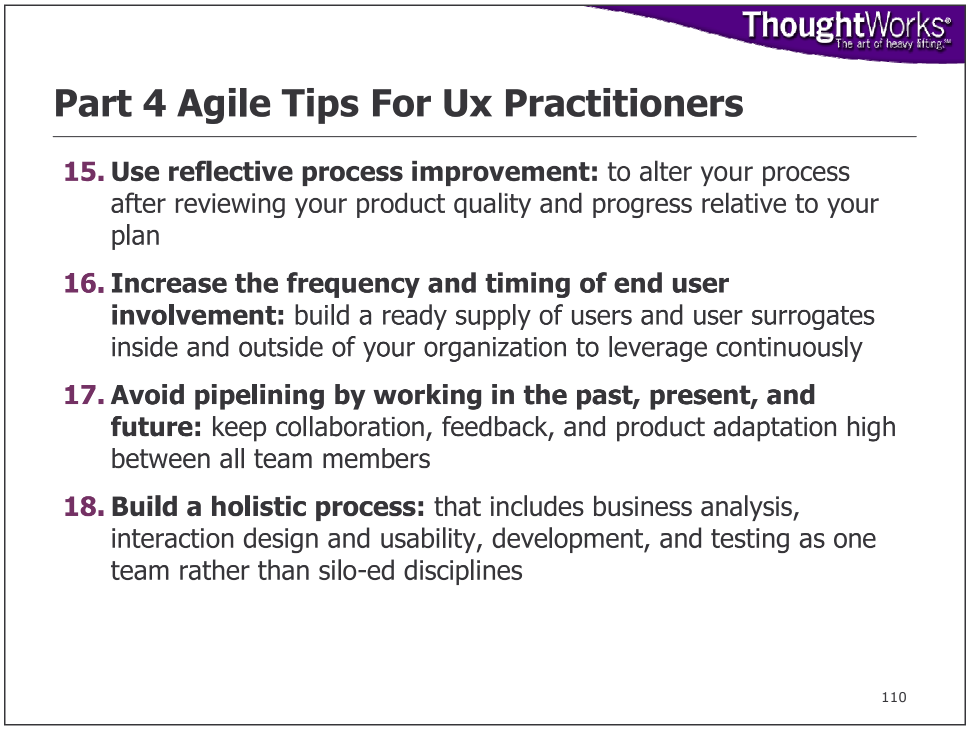
- User Centered Design, Interaction Design, Usability, and Business Analysis and Requirements Gathering are silo-ed activities
- Duplication of data gathering and modeling efforts
  - User interviews
  - Business stakeholder interviews
  - Business process modeling
  - Task analysis
  - UI prototyping
- Business stakeholders that detect the duplication often choose to omit one group or the other
- Usability as an overlooked non-functional requirement
- Fundamental difference in responsibility, attitude, posture of two disciplines:
  - designing vs. capturing and managing

  "This is classic user interface (UI) design, and it is the orphan child of software development methodologies. Is it design? Is it analysis? Does a requirements specification include the UI or does it not? No one seems quite sure."

  -- Beyer, Holtzblatt, & Baker, **An Agile User-Centered Method: Rapid Contextual Design**
- Agile Software Development doesn't remedy this fundamental confusion

109

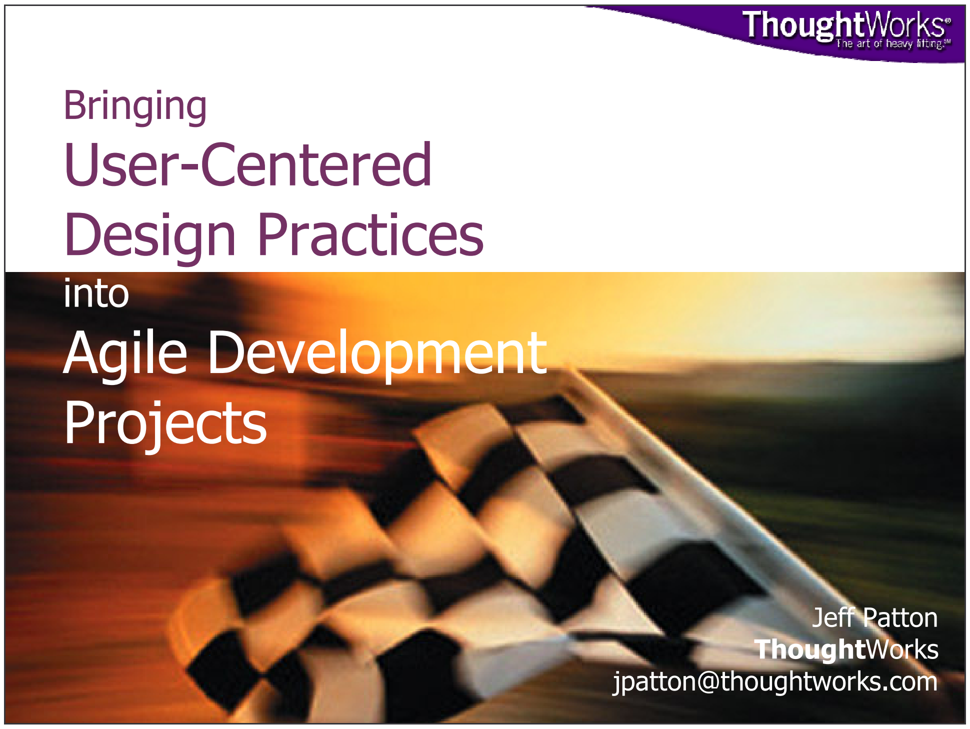
Notes



## Part 4 Agile Tips For Ux Practitioners

15. **Use reflective process improvement:** to alter your process after reviewing your product quality and progress relative to your plan
16. **Increase the frequency and timing of end user involvement:** build a ready supply of users and user surrogates inside and outside of your organization to leverage continuously
17. **Avoid pipelining by working in the past, present, and future:** keep collaboration, feedback, and product adaptation high between all team members
18. **Build a holistic process:** that includes business analysis, interaction design and usability, development, and testing as one team rather than silo-ed disciplines

110



Bringing

User-Centered

Design Practices

into

Agile Development

Projects

Jeff Patton
ThoughtWorks
jpatton@thoughtworks.com