

# Becoming an Effective Product Owner

Mike Cohn

14 November 2006



1

## Four questions

1

What are the primary responsibilities of the product owner?

2

What are some differences between the role of the product owner and the ScrumMaster?

3

What skills should the ideal product owner possess?

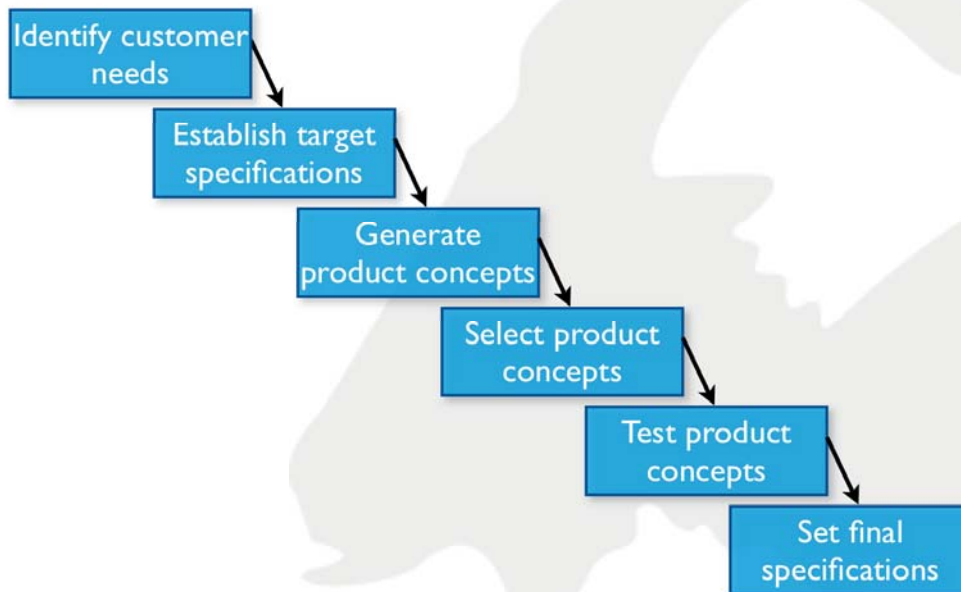
4

What things would you expect to happen on a project without a product owner?



2

## Traditional view of product management



© Mountain Goat Software, LLC

3

## Traditional development

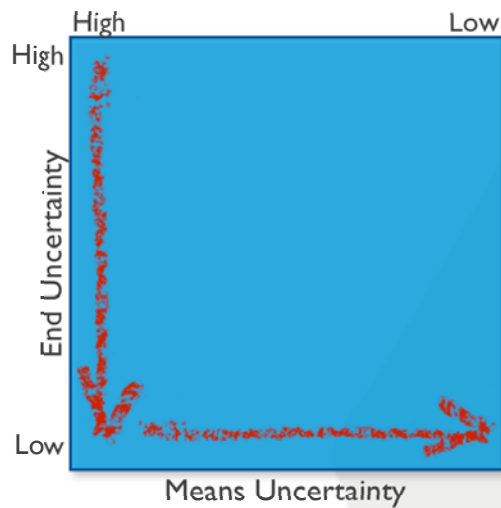
- Prior to Scrum and Agile, it was considered prudent to understand what was wanted and how it was going to be delivered at the very start of the project.
- All future work hung off, depended on this work.
- The theory was that changes at the start of the project cost \$1, but the same change made when the project was 60% complete cost \$100.



© Mountain Goat Software, LLC

4

## Product owner helps reduce uncertainty



Waterfall



Agile

© Mountain Goat Software, LLC

5

## Emergence

- It is impossible to know all requirements in advance
- “Thinking harder” and “thinking longer” can uncover some requirements, but

Every project has some emergent requirements

- Emergent requirements are those our users cannot identify in advance

© Mountain Goat Software, LLC

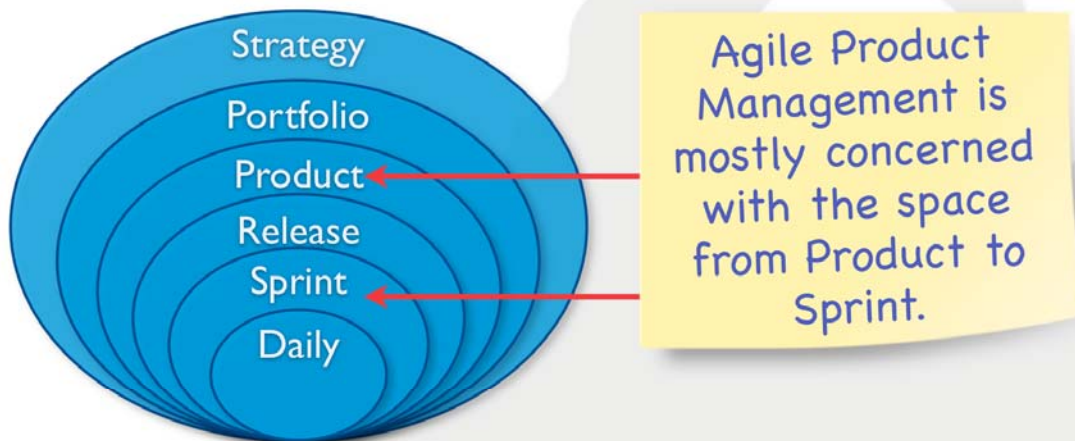
6

# So what do we do?

- We talk more, write less
  - But write some if you need to
- Show software to users
- Acknowledge that requirements emerge
  - And all that this implies
- Progressively refine our understanding of the product
  - And express this progressive refinement in the product backlog



# The planning onion





## Telling Stories with the Product Backlog

© Mountain Goat Software, LLC

9

## Stories make great backlog items

### Card

- Stories are traditionally written on note cards.
- May be annotated with notes, estimates, etc.

### Conversation

- Details behind the story come out during conversations with product owner

### Confirmation

- Acceptance tests confirm the story was coded correctly

Source: XP Magazine 8/30/01, Ron Jeffries.

© Mountain Goat Software, LLC

10

# Samples from a travel website

As a user, I want to reserve a hotel room.

As a user, I want to cancel a reservation.

As a vacation planner, I want to see photos of the hotels.

As a frequent flier, I want to rebook a past trip, so that I save time booking trips I take



## Where are the details?

- As a user, I can cancel a reservation.
  - Does the user get a full or partial refund?
    - Is the refund to her credit card or is it site credit?
  - How far ahead must the reservation be cancelled?
    - Is that the same for all hotels?
    - For all site visitors? Can frequent travelers cancel later?
  - Is a confirmation provided to the user?
    - How?



## Details as conditions of satisfaction

- The product owner's conditions of satisfaction can be added to a story
  - These are essentially tests

As a user, I can cancel a reservation.

- Verify that a premium member can cancel the same day without a fee.
- Verify that a non-premium member is charged 10% for a same-day cancellation.
- Verify that an email confirmation is sent.
- Verify that the hotel is notified of any cancellation.



## Details added in smaller stories

As a user, I can cancel a reservation.

As a premium site member, I can cancel a reservation up to the last minute.

As a non-premium member, I can cancel up to 24 hours in advance.

As a site visitor, I am emailed a confirmation of any cancelled reservation.

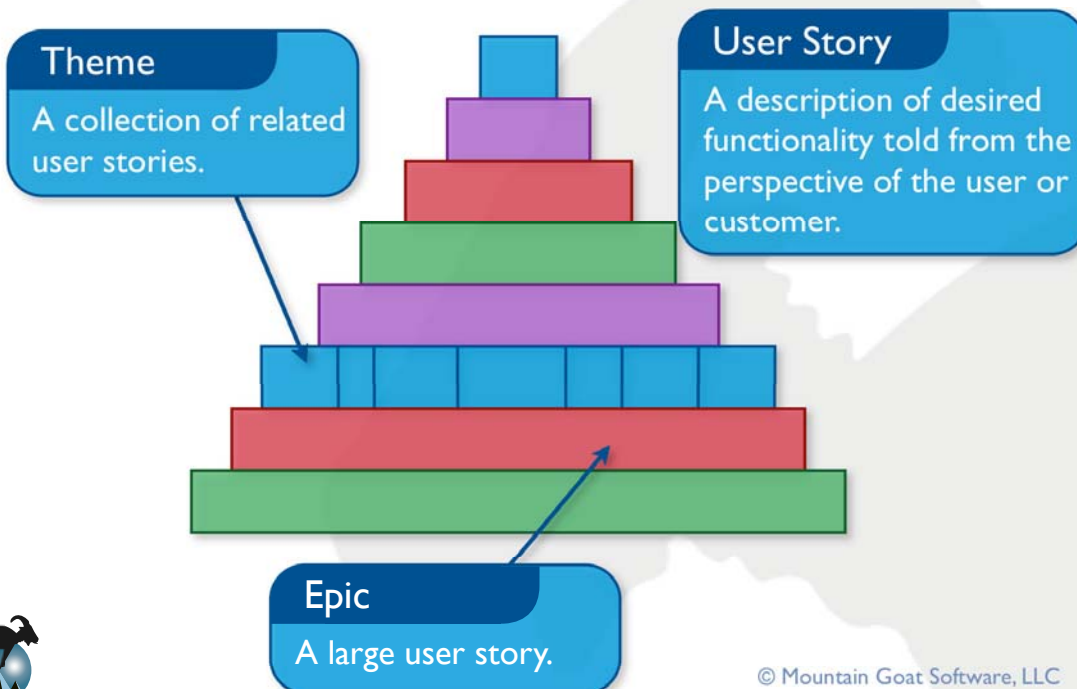


# Stocking the product backlog

- You can start by identifying only a sprint's worth of backlog items
- But, it's often quick and easy to stock the product backlog with most of its items
  - This is helpful for release planning, expectation setting, and can influence design and coding
- The key is to write product backlog items with different levels of detail
  - Fine-grained for stories about to be worked on
  - Coarse-grained for stories further in the future



# The product backlog iceberg





# An example

As a VP Marketing, I want to review the performance of historical promotional campaigns so that I can identify and repeat profitable ones.

An epic;  
weeks to implement

As a VP Marketing, I want to select the timeframe to use when reviewing the performance of past promotional campaigns, so that I can identify and repeat profitable ones.

Implementation-size stories;  
days to implement

As a VP Marketing, I can select which type of campaigns (direct mail, TV, email, radio, etc.) to include when reviewing the performance of historical promotional campaigns.



# An example

As a VP Marketing, I want to see information on direct mailings when reviewing historical campaigns.

As a VP Marketing, I want to see information on television advertising when reviewing historical campaigns.

As a VP Marketing, I want to see information on email advertising when reviewing historical campaigns.

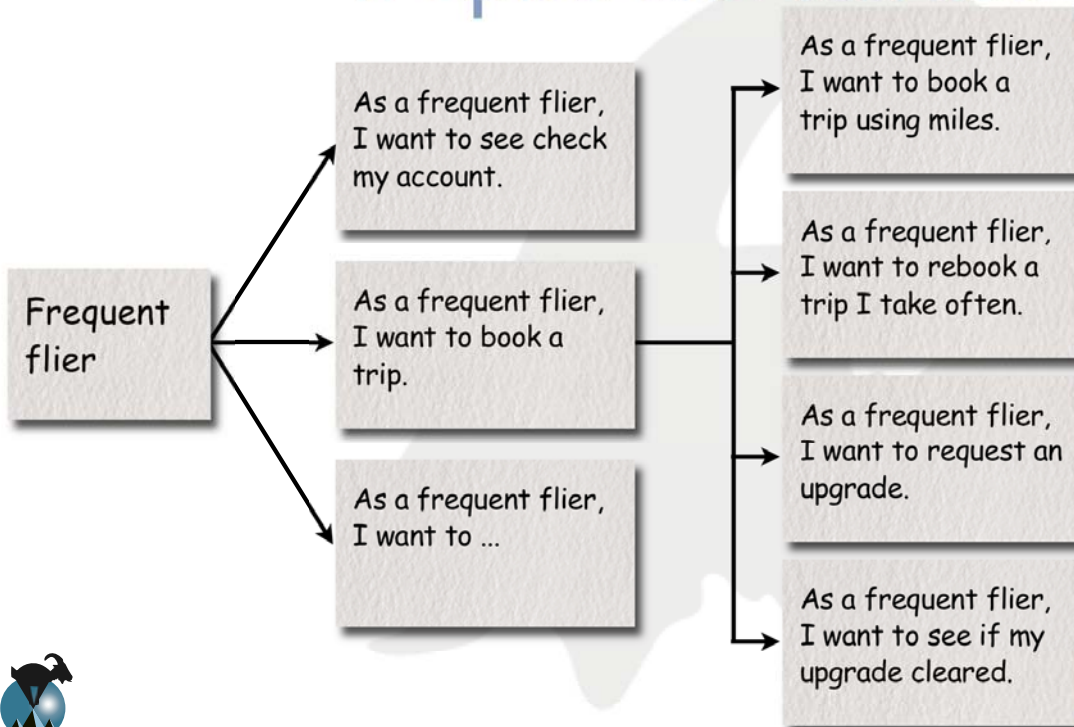


# Story-writing workshops

- Includes developers, users, customer, others
- Brainstorm to generate stories
- Goal is to write as many stories as possible
  - Some will be “implementation ready”
  - Others will be “epics”
- No prioritization at this point



## Start with epics and iterate



# Another approach

- Walk through a low-fidelity (paper) user interface
  - Ask open-ended, context-free questions as you go:
    - What will the users most likely want to do next?
    - What mistakes could the user make here?
    - What could confuse the user at this point?
    - What additional information could the user need?
  - Consider these questions for each user role



## MyCookSpace.com

Your team has been hired and given gobs of VC money to create a social networking site (like MySpace) for cooks. The idea is that cooks will exchange recipes and tips. They'll also buy kitchen-related products from the advertisers on the site. We want some novel way of rewarding cooks who post the best and most popular recipes as noted by other cooks on the site.

Write some user stories for MyCookSpace.com.

Try this template:

“As a <user role>, I want <goal> so that <reason>.”



# How much detail?

Just-in-time  
just-enough

Process

- Fixed

Output

- Fixed

Input

- Must be described in just enough detail to be turned into the output by the process

© Mountain Goat Software, LLC

23

# What makes a good story?

- I Independent
- N Negotiable
- V Valuable
- E Estimatable
- S Sized Appropriately
- T Testable

© Mountain Goat Software, LLC

24

## I Independent

- Dependencies lead to problems estimating and prioritizing
- Can ideally select a story to work on without pulling in 18 other stories

## Negotiable

- Stories are not contracts
- Leave or imply some flexibility

## V Valuable

- To users or customers, not developers
- Rewrite developer stories to reflect value to users or customers



## E Estimatable

- Because plans are based on user stories, we need to be able to estimate them


## S Sized Appropriately

- Small enough to complete in one sprint if you're about to work on it
- Bigger if further off on the horizon

## T Testable

- Testable so that you have a easy, binary way of knowing whether a story is finished
- Done or not done; no “partially finished” or “done except”





## Prioritizing the Product Backlog

© Mountain Goat Software, LLC

27

## Prioritizing the product backlog

### Three steps

1. Organize needs into themes
2. Assess importance of each theme
3. Prioritize themes



© Mountain Goat Software, LLC

28

# Why themes?

- Often individual stories cannot be prioritized against each other
- What's more important in a word processor?
  - The A key or the E key?
  - Tables or undo?
- What's more important on a car?
  - The left front wheel or the right front wheel?
  - Increased leg room or a larger engine?



# Steps for organizing into themes

1. Write each story on its own note card or post-it
2. Eliminate redundant stories
3. Group similar stories
4. Label each group with a theme name
5. If you have a lot of themes, consider making them into larger themes
6. Review the results

If you started with a story-writing workshop you may already have themes.



# Affinity grouping

- Distribute cards equally to all participants
  - No particular pattern to how you do this
- Someone reads a card and places it on wall / table
  - Others look for similar cards and add them to it
- Next person reads a card, places it, and others place similar cards with it
- Continue repeating until out of cards



# An example

As a frequent flyer, I want to book a flight.

..., I want to book a flight using miles.

..., I want to book a flight and pay for it.

..., I want to re-book a flight I take often.

..., I want to request an upgrade to first class.

..., I want to see if my upgrade cleared.

Be sure you are organizing stories around user needs;  
not around how you'll build the software or  
how the company is organized.





# Typical results



## Step 2

# Assess importance of each theme

- Two general approaches
  1. Team opinion
  2. Survey users
- Some specific approaches
  - Theme screening
  - Theme scoring
  - Relative weighting
  - Kano analysis
  - Financial analysis
  - Analytic Hierarchy Process

We'll look at the first four today.



# Choosing your approach

	Expert Opinion	User Interview
Theme screening	✓	
Theme scoring	✓	
Relative weighting	✓	
Kano analysis	✓	✓
Financial analysis	✓	
Analytic Hierarchy Process	✓	✓



## Theme screening

- Identify 5-9 (approximately) selection criteria for what is important in the next release
- Select a baseline theme
  - Likely to be included in the next release
  - Understood by most team members
- Assess each candidate theme relative to the baseline theme



# Theme screening: an example

Selection Criteria	Themes						
	Theme A	Theme B	Theme C	Baseline Theme	Theme E	Theme F	Theme G
Importance to existing customers	+	+	-	0	-	+	0
Competitiveness with ABC Corp.	+	-	0	0	0	0	0
Starts us integrating product lines	+	0	0	0	+	+	+
Generates revenue in Q2	0	0	0	0	+	0	+
Net score	3	0	-1	0	1	0	2
Rank	1	4	5	4	3	4	2
Continue?	Y	N	N	Y	Y	N	Y

+ = better than  
 0 = same as  
 - = worse than

© Mountain Goat Software, LLC

37

# Theme scoring

- Like theme screening but selection criteria are weighted
- Need to select a baseline theme for each criteria
- Avoids compression of a category
- Each theme is assessed against the baseline for each selection criteria

Much worse than reference	1
Worse than reference	2
Same as reference	3
Better than reference	4
Much better than reference	5

© Mountain Goat Software, LLC

38

# Theme scoring: An example

Selection Criteria	Weight	Theme A		Theme B		Theme C	
		Rating	Weighted Score	Rating	Weighted Score	Rating	Weighted Score
Importance to existing cust.	25	3	0.75	1	0.25	4	1.00
Competitive. with ABC	10	2	0.20	3	0.30	3	0.30
Starts us integrating...	15	3	0.45	4	0.60	4	0.60
Generates Q2 revenue	50	5	2.50	2	1.00	3	1.50
Net score			3.90		2.15		3.40
Rank			1		3		2
Continue?			Yes		No		Yes



## Prioritizing MyCookSpace.com

- Assume we have a minimally functional site up with 4,000 registered cooks
  - We want 400,000 cooks
- As groups, decide whether you want to use theme screening or theme scoring.
- Identify 4-5 themes
- Identify some selection criteria
  - What's important to the company in making this decision?
- Complete a theme screening/scoring worksheet



# Theme Screening Worksheet



		Themes						
Selection Criteria								
	Net score							
	Rank							
	Continue?							

+ = Better than

0 = Same as

- = Worse than



# Relative weighting

- Assess the impact of having a story/theme from 1-9
- Assess impact of NOT having it from 1-9
- Calculate the value of each story or theme relative to the entire product backlog
  - This gives you the relative value of that story or theme
- Developers estimate the cost of each story theme
- Calculate the cost of each story or theme relative to the entire product backlog
  - This gives the relative cost of that story or theme
- Priority is given by (Relative Value ÷ Relative Cost)



## Relative weighting: an example

		Relative Benefit	Relative Penalty	Total Value	Value Percent	Estimate	Cost Percent	Priority
Themes	More investment choices	8	6	14	40	64	44	91
	Portfolio rebalancing	9	2	11	31	40	27	115
	Comply with new law	1	9	10	29	42	29	100
Total				35	100	146	100	

$$\text{Total Value} = \text{Relative Benefit} + \text{Relative Penalty}$$

$$\text{Value Percent} = \text{Total Value} / \sum (\text{Total Value})$$

$$\text{Cost Percent} = \text{Estimate} / \sum (\text{Estimate})$$



# An example with weights

		Weight →						
		2	1	Relative Benefit	Relative Penalty	Total Value	Value Percent	Estimate
Themes	More investment choices	8	6	22	41	64	44	93
	Portfolio rebalancing	9	2	20	38	40	27	141
	Comply with new law	1	9	11	21	42	29	72
Total				53	100	14	100	



© Mountain Goat Software, LLC

43

# Priority poker

- An iterative approach to estimating:
  - Stakeholders with a say in prioritizing are invited
  - Each is given a deck of cards with the values A-9
  - A moderator reads a theme and it's discussed briefly
  - Each estimator selects a card that is his or her estimate of the relative benefit of the theme
  - Cards are turned over so all can see them
  - Discuss differences (especially outliers)
  - Re-estimate until estimates converge
  - Repeat for relative penalty

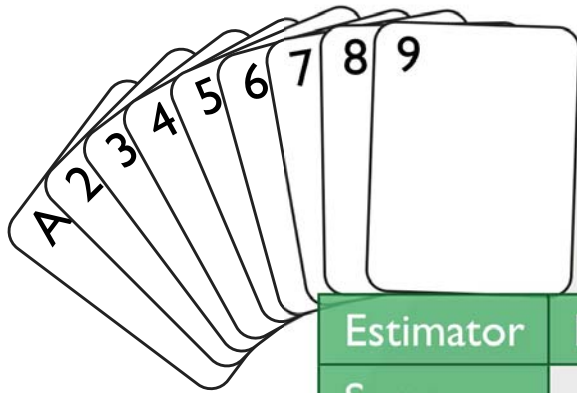


© Mountain Goat Software, LLC

44



# Priority poker - an example



Estimator	Round 1	Round 2
Susan	3	5
Vadim	8	5
Ann	2	5
Chris	5	7



## Relative weighting MyCookSpace.com

- Using priority poker and the relative weighting worksheet provided, prioritize the themes you've previously identified for MyCookSpace.com



# Relative Weighting Worksheet

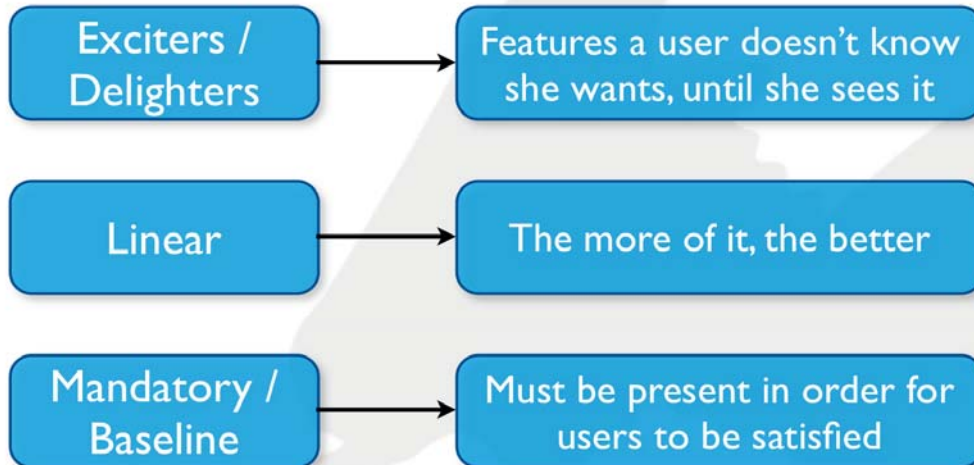


		Weight:						
Themes		Relative Benefit	Relative Penalty	Total Value	Value Percent	Estimate	Cost Percent	Priority
		Total:				100		100

Total Value = Relative Benefit + Relative Penalty (× weights if used)  
 Value Percent = Total Value ÷ ∑(Total Value)  
 Cost Percent = Estimate ÷ ∑Estimate  
 Priority = Value Percent / Cost Percent (higher = higher priority)

# Kano analysis

## Three types of features

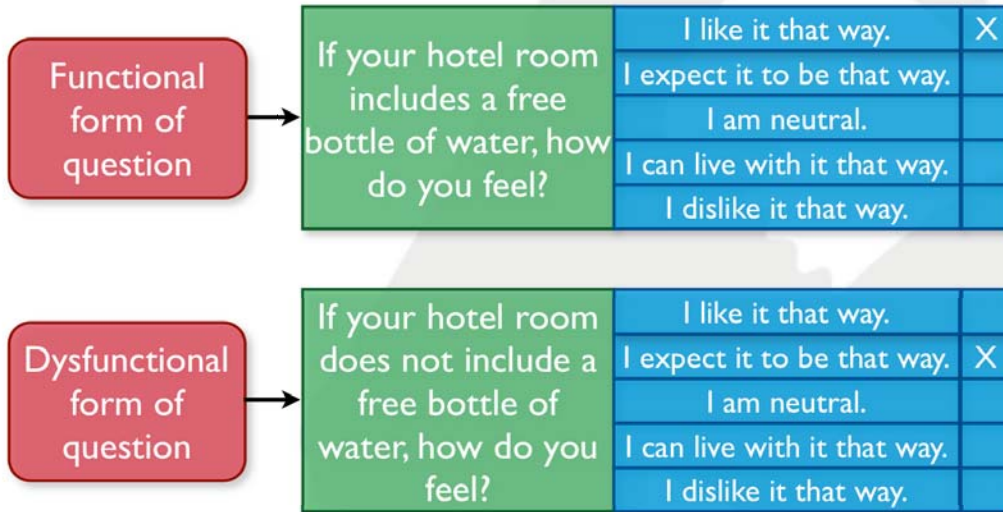


# Surveying users

- To assess whether a feature is baseline, linear, or exciting we can:
  - Sometimes guess
  - Or survey a small set of users (20-30)
- We ask two questions
  - A functional question
    - How do you feel if a feature is present?
  - And a dysfunctional question
    - How do you feel if that feature is absent?



# Functional and dysfunctional forms



# Categorizing an answer pair

		Dysfunctional Question				
		Like	Expect	Neutral	Live with	Dislike
Functional Question	Like	Q	E	E	E	L
	Expect	R	I	I	I	M
	Neutral	R	I	I	I	M
	Live with	R	I	I	I	M
	Dislike	R	R	R	R	Q

- M Mandatory
- L Linear
- E Exciter
- Q Questionable
- R Reverse
- I Indifferent



# Aggregating results

Theme	Exciter	Linear	Mandatory	Indifferent	Reverse	Questionable
Apply formatting themes	3	11	31	1	3	2
Automate report execution	4	22	20	4	1	0
Export reports to PowerPoint	21	9	14	5	1	1



# What to include

- All of the baseline features
  - By definition, these must be present
- Some amount of linear features
- But leaving room for at least a few excitors



# Upcoming public classes

Date	What	Where
January 16-18	Certified ScrumMaster (with Ken Schwaber) Agile Estimating & Planning	Orlando, FL
February 14-15 and 19-20, 2007	Certified ScrumMaster	London
February 27- March 1, 2007	Certified ScrumMaster Agile Estimating & Planning	Denver, CO
April 10-12, 2007	Certified ScrumMaster Agile Estimating & Planning	Santa Clara, CA

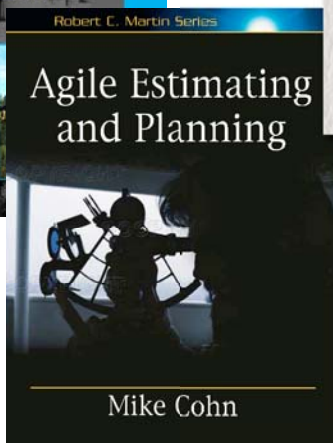
Register at  
[www.mountangoatsoftware.com](http://www.mountangoatsoftware.com)



© Mountain Goat Software, LLC

53

# Mike Cohn contact info



[mike@mountangoatsoftware.com](mailto:mike@mountangoatsoftware.com)

[www.mountangoatsoftware.com](http://www.mountangoatsoftware.com)

(720) 890-6110 (office)

(303) 810-2190 (mobile)



© Mountain Goat Software, LLC



54

## Chapter 2

---

---

# Writing Stories

In this chapter we turn our attention to writing the stories. To create good stories we focus on six attributes. A good story is:

- Independent
- Negotiable
- Valuable to users or customers
- Estimatable
- Small
- Testable

Bill Wake, author of *Extreme Programming Explored* and *Refactoring Workbook*, has suggested the acronym INVEST for these six attributes (Wake 2003a).

---

### Independent

As much as possible, care should be taken to avoid introducing dependencies between stories. Dependencies between stories lead to prioritization and planning problems. For example, suppose the customer has selected as high priority a story that is dependent on a story that is low priority. Dependencies between stories can also make estimation much harder than it needs to be. For example, suppose we are working on the BigMoneyJobs website and need to write stories for how companies can pay for the job openings they post to our site. We could write these stories:

1. A company can pay for a job posting with a Visa card.
2. A company can pay for a job posting with a MasterCard.

3. A company can pay for a job posting with an American Express card.

Suppose the developers estimate that it will take three days to support the first credit card type (regardless of which it is) and then one day each for the second and third. With highly dependent stories such as these you don't know what estimate to give each story—which story should be given the three day estimate?

When presented with this type of dependency, there are two ways around it:

- Combine the dependent stories into one larger but independent story
- Find a different way of splitting the stories

Combining the stories about the different credit card types into a single large story (“A company can pay for a job posting with a credit card”) works well in this case because the combined story is only five days long. If the combined story is much longer than that, a better approach is usually to find a different dimension along which to split the stories. If the estimates for these stories had been longer, then an alternative split would be:

1. A customer can pay with one type of credit card.
2. A customer can pay with two additional types of credit cards.

If you don't want to combine the stories and can't find a good way to split them, you can always take the simple approach of putting two estimates on the card: one estimate if the story is done before the other story, a lower estimate if it is done after.

---

## Negotiable

Stories are negotiable. They are not written contracts or requirements that the software must implement. Story cards are short descriptions of functionality, the details of which are to be negotiated in a conversation between the customer and the development team. Because story cards are reminders to have a conversation rather than fully detailed requirements themselves, they do not need to include all relevant details. However, if at the time the story is written some important details are known, they should be included as annotations to the story card, as shown in Story Card 2.1. The challenge comes in learning to include just enough detail.

Story Card 2.1 works well because it provides the right amount of information to the developer and customer who will talk about the story. When a devel-



*A company can pay for a job posting with a credit card.*

*Note: Accept Visa, MasterCard, and American Express. Consider Discover.*

- Story Card 2.1 A story card with notes providing additional detail.

oper starts to code this story, she will be reminded that a decision has already been made to accept the three main cards and she can ask the customer if a decision has been made about accepting Discover cards. The notes on the card help a developer and the customer to resume a conversation where it left off previously. Ideally, the conversation can be resumed this easily regardless of whether it is the same developer and customer who resume the conversation. Use this as a guideline when adding detail to stories.

On the other hand, consider a story that is annotated with too many notes, as shown in Story Card 2.2. This story has too much detail (“Collect the expiration month and date of the card”) and also combines what should probably be a separate story (“The system can store a card number for future use”).

*A company can pay for a job posting with a credit card.*

*Note: Accept Visa, MasterCard, and American Express. Consider Discover. On purchases over \$100, ask for card ID number from back of card. The system can tell what type of card it is from the first two digits of the card number. The system can store a card number for future use. Collect the expiration month and date of the card.*

- Story Card 2.2 A story card with too much detail.

Working with stories like Story Card 2.2 is very difficult. Most readers of this type of story will mistakenly associate the extra detail with extra precision. However, in many cases specifying details too soon just creates more work. For example, if two developers discuss and estimate a story that says simply “a company can pay for a job posting with a credit card” they will not forget that their discussion is somewhat abstract. There are too many missing details for

them to mistakenly view their discussion as definitive or their estimate as accurate. However, when as much detail is added as in Story Card 2.2, discussions about the story are much more likely to feel concrete and real. This can lead to the mistaken belief that the story cards reflect all the details and that there's no further need to discuss the story with the customer.

If we think about the story card as a reminder for the developer and customer to have a conversation, then it is useful to think of the story card as containing:

- a phrase or two that act as reminders to hold the conversation
- notes about issues to be resolved during the conversation

Details that have already been determined through conversations become tests. Tests can be noted on the back of the story card if using note cards or in whatever electronic system is being used. Story Card 2.3 and Story Card 2.4 show how the excess detail of Story Card 2.2 can be turned into tests, leaving just notes for the conversation as part of the front of the story card. In this way, the front of a story card contains the story and notes about open questions while the back of the card contains details about the story in the form of tests that will prove whether or not it works as expected.

*A company can pay for a job posting with a credit card.*

*Note: Will we accept Discover cards?*

*Note for UI: Don't have a field for card type (it can be derived from first two digits on the card).*

- Story Card 2.3 The revised front of a story card with only the story and questions to be discussed.

---

## Valuable to Purchasers or Users

It is tempting to say something along the lines of "Each story must be valued by the users." But that would be wrong. Many projects include stories that are not valued by users. Keeping in mind the distinction between *user* (someone who uses the software) and *purchaser* (someone who purchases the software), suppose a development team is building software that will be deployed across a

*Test with Visa, MasterCard and American Express (pass).*

*Test with Diner's Club (fail).*

*Test with good, bad and missing card ID numbers.*

*Test with expired cards.*

*Test with over \$100 and under \$100.*

- Story Card 2.4 Details that imply test cases are separated from the story itself. Here they are shown on the back of the story card.

large user base, perhaps 5,000 computers in a single company. The purchaser of a product like that may be very concerned that each of the 5,000 computers is using the same configuration for the software. This may lead to a story like “All configuration information is read from a central location.” Users don't care where configuration information is stored but purchasers might.

Similarly, stories like the following might be valued by purchasers contemplating buying the product but would not be valued by actual users:

- Throughout the development process, the development team will produce documentation suitable for an ISO 9001 audit.
- The development team will produce the software in accordance with CMM Level 3.

What you want to avoid are stories that are only valued by developers. For example, avoid stories like these:

- All connections to the database are through a connection pool.
- All error handling and logging is done through a set of common classes.

As written, these stories are focused on the technology and the advantages to the programmers. It is very possible that the ideas behind these stories are good ones but they should instead be written so that the benefits to the customers or the user are apparent. This will allow the customer to intelligently prioritize these stories into the development schedule. Better variations of these stories could be the following:

- Up to fifty users should be able to use the application with a five-user database license.
- All errors are presented to the user and logged in a consistent manner.

In exactly the same way it is worth attempting to keep user interface assumptions out of stories, it is also worth keeping technology assumptions out of stories. For example, the revised stories above have removed the implicit use of a connection pool and a set of error handling classes.

The best way to ensure that each story is valuable to the customer or users is to have the customer write the stories. Customers are often uncomfortable with this initially—probably because developers have trained them to think of everything they write as something that can be held against them later. (“Well, the requirements document didn’t say that...”) Most customers begin writing stories themselves once they become comfortable with the concept that story cards are reminders to talk later rather than formal commitments or descriptions of specific functionality.

---

## Estimatable

It is important for developers to be able to estimate (or at least take a guess at) the size of a story or the amount of time it will take to turn a story into working code. There are three common reasons why a story may not be estimatable:

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

First, the developers may lack domain knowledge. If the developers do not understand a story as it is written, they should discuss it with the customer who wrote the story. Again, it’s not necessary to understand all the details about a story, but the developers need to have a general understanding of the story.

Second, a story may not be estimatable because the developers do not understand the technology involved. For example, on one Java project we were asked to provide a CORBA interface into the system. No one on the team had done that so there was no way to estimate the task. The solution in this case is to send one or more developers on what Extreme Programming calls a *spike*, which is a brief experiment to learn about an area of the application. During the spike the developers learn just enough that they can estimate the task. The spike itself is always given a defined maximum amount of time (called a *time-box*), which allows us to estimate the spike. In this way an unestimatable story turns into two stories: a quick spike to gather information and then a story to do the real work.

Finally, the developers may not be able to estimate a story if it is too big. For example, for the BigMoneyJobs website, the story “A Job Seeker can find a job” is too large. In order to estimate it the developers will need to disaggregate it into smaller, constituent stories.

---

### **A Lack of Domain Knowledge**

As an example of needing more domain knowledge, we were building a website for long-term medical care of chronic conditions. The customer (a highly qualified nurse) wrote a story saying “New users are given a diabetic screening.” The developers weren’t sure what that meant and it could have run the gamut from a simple web questionnaire to actually sending something to new users for an at-home physical screening, as was done for the company’s product for asthma patients. The developers got together with the customer and found out that she was thinking of a simple web form with a handful of questions.

---

Even though they are too big to estimate reliably, it is sometimes useful to write epics such as “A Job Seeker can find a job” because they serve as placeholders or reminders about big parts of a system that need to be discussed. If you are making a conscious decision to temporarily gloss over large parts of a system, then consider writing an epic or two that cover those parts. The epic can be assigned a large, pulled-from-thin-air estimate.

---

## **Small**

Like Goldilocks in search of a comfortable bed, some stories can be too big, some can be too small, and some can be just right. Story size does matter because if stories are too large or too small you cannot use them in planning. Epics are difficult to work with because they frequently contain multiple stories. For example, in a travel reservation system, “A user can plan a vacation” is an epic. Planning a vacation is important functionality for a travel reservation system but there are many tasks involved in doing so. The epic should be split into smaller stories. The ultimate determination of whether a story is appropriately sized is based on the team, its capabilities, and the technologies in use.

## Splitting Stories

Epics typically fall into one of two categories:

- The compound story
- The complex story

A compound story is an epic that comprises multiple shorter stories. For example, the BigMoneyJobs system may include the story “A user can post her resume.” During the initial planning of the system this story may be appropriate. But when the developers talk to the customer, they find out that “post her resume” actually means:

- that a resume can include education, prior jobs, salary history, publications, presentations, community service, and an objective
- that users can mark resumes as inactive
- that users can have multiple resumes
- that users can edit resumes
- that users can delete resumes

Depending on how long these will take to develop, each could become its own unique story. However, that may just take an epic and go too far in the opposite direction, turning it into a series of stories that are too small. For example, depending on the technologies in use and the size and skill of the team, stories like these will generally be too small:

- A Job Seeker can enter a date for each community service entry on a resume.
- A Job Seeker can edit the date for each community service entry on a resume.
- A Job Seeker can enter a date range for each prior job on a resume.
- A Job Seeker can edit the date range for each prior job on a resume.

Generally, a better solution is to group the smaller stories as follows:

- A user can create resumes, which include education, prior jobs, salary history, publications, presentations, community service, and an objective.
- A user can edit a resume.
- A user can delete a resume.

- A user can have multiple resumes.
- A user can activate and inactivate resumes.

There are normally many ways to disaggregate a compound story. The preceding disaggregation is along the lines of create, edit, and delete, which is commonly used. This works well if the create story is small enough that it can be left as one story. An alternative is to disaggregate along the boundaries of the data. To do this, think of each component of a resume as being added and edited individually. This leads to a completely different disaggregation:

- A user can add and edit education information.
- A user can add and edit job history information.
- A user can add and edit salary history information.
- A user can add and edit publications.
- A user can add and edit presentations.
- A user can add and edit community service.
- A user can add and edit an objective.

And so on.

Unlike the compound story, the complex story is a user story that is inherently large and cannot easily be disaggregated into a set of constituent stories. If a story is complex because of uncertainty associated with it, you may want to split the story into two stories: one investigative and one developing the new feature. For example, suppose the developers are given the story “A company can pay for a job posting with a credit card” but none of the developers has ever done credit card processing before. They may choose to split the stories like this:

- Investigate credit card processing over the web.
- A user can pay with a credit card.

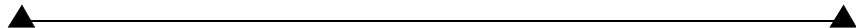
In this case the first story will send one or more developers on a spike. When complex stories are split in this way, always define a timebox around the investigative story, or spike. Even if the story cannot be estimated with any reasonable accuracy, it is still possible to define the maximum amount of time that will be spent learning.

Complex stories are also common when developing new or extending known algorithms. One team in a biotech company had a story to add novel extensions

to a standard statistical approach called expectation maximization. The complex story was rewritten as two stories: the first to research and determine the feasibility of extending expectation maximization; the second to add that functionality to the product. In situations like this one it is difficult to estimate how long the research story will take.

  
**Consider Putting the Spike in a Different Iteration**

When possible, it works well to put the investigative story in one iteration and the other stories in one or more subsequent iterations. Normally, only the investigative story can be estimated. Including the other, non-estimatable stories in the same iteration with the investigative story means there will be a higher than normal level of uncertainty about how much can be accomplished in that iteration.

The key benefit of breaking out a story that cannot be estimated is that it allows the customer to prioritize the research separately from the new functionality. If the customer has only the complex story to prioritize (“Add novel extensions to standard expectation maximization”) and an estimate for the story, she may prioritize the story based on the mistaken assumption that the new functionality will be delivered in approximately that timeframe. If instead, the customer has an investigative, spike story (“research and determine the feasibility of extending expectation maximization”) and a functional story (“extend expectation maximization”), she must choose between adding the investigative story that adds no new functionality this iteration and perhaps some other story that does.

## Combining Stories

Sometimes stories are too small. A story that is too small is typically one that the developer says she doesn’t want to write down or estimate because doing that may take longer than making the change. Bug reports and user interface changes are common examples of stories that are often too small. A good approach for tiny stories, common among Extreme Programming teams, is to combine them into larger stories that represent from about a half-day to several days of work. The combined story is given a name and is then scheduled and worked on just like any other story.

For example, suppose a project has five bugs and a request to change some colors on the search screen. The developers estimate the total work involved



and the entire collection is treated as a single story. If you've chosen to use paper note cards, you can do this by stapling them together with a cover card.

---

## Testable

Stories must be written so as to be testable. Successfully passing its tests proves that a story has been successfully developed. If the story cannot be tested, how can the developers know when they have finished coding?

Untestable stories commonly show up for nonfunctional requirements, which are requirements about the software but not directly about its functionality. For example, consider these nonfunctional stories:

- A user must find the software easy to use.
- A user must never have to wait long for any screen to appear.

As written, these stories are not testable. Whenever possible, tests should be automated. This means strive for 99% automation, not 10%. You can almost always automate more than you think you can. When a product is developed incrementally, things can change very quickly and code that worked yesterday can stop working today. You want automated tests that will find this as soon as possible.

There is a very small subset of tests that cannot realistically be automated. For example, a user story that says “A novice user is able to complete common workflows without training” can be tested but cannot be automated. Testing this story will likely involve having a human factors expert design a test that involves observation of a random sample of representative novice users. That type of test can be both time-consuming and expensive, but the story is testable and may be appropriate for some products.

The story “a user never has to wait long for any screen to appear” is not testable because it says “never” and because it does not define what “wait long” means. Demonstrating that something never happens is impossible. A far easier, and more reasonable target, is to demonstrate that something rarely happens. This story could have instead been written as “New screens appear within two seconds in 95% of all cases.” And—even better—an automated test can be written to verify this.

## Summary

- Ideally, stories are independent from one another. This isn't always possible but to the extent it is, stories should be written so that they can be developed in any order.
  - The details of a story are negotiated between the user and the developers.
  - Stories should be written so that their value to users or the customer is clear. The best way to achieve this is to have the customer write the stories.
  - Stories may be annotated with details, but too much detail obscures the meaning of the story and can give the impression that no conversation is necessary between the developers and the customer.
  - One of the best ways to annotate a story is to write test cases for the story.
  - If they are too big, compound and complex stories may be split into multiple smaller stories.
  - If they are too small, multiple tiny stories may be combined into one bigger story.
  - Stories need to be testable.
- 

## Developer Responsibilities

- You are responsible for helping the customer write stories that are promises to converse rather than detailed specifications, have value to users or the customer, are independent, are testable, and are appropriately sized.
  - If tempted to ask for a story about the use of a technology or a piece of infrastructure, you are responsible for instead describing the need in terms of its value to users or the customer.
- 

## Customer Responsibilities

- You are responsible for writing stories that are promises to converse rather than detailed specifications, have value to users or to yourself, are independent, are testable, and are appropriately sized.



*I Didn't Know I  
Needed That!*

*Finding Features  
to Satisfy Your  
Customers*

BY MIKE COHN



It is essential that the products we develop satisfy our customers. A product that fails in this regard is unlikely to be around for very long. Unfortunately, many companies and development teams do not consciously plan for how a product will satisfy their customers. The problem with this approach is that if you don't have a plan for customer satisfaction, you probably won't achieve it.

When planning to develop a desirable new product, it is useful to group features based on their potential for impacting customer satisfaction. One technique for doing so is known as Kano

I don't want a horrible microphone on my mobile phone, but improving the quality of the microphone can only increase my satisfaction with the phone up to a certain point.

Similarly, I consider a spell checker to be a mandatory feature in a word processor. However, no matter how wonderful a particular spell checker is or how many words are in its dictionary, my opinion of the word processor can only rise so far based on the spell checker. This is why the arrow in Figure 1 flattens rapidly. A mandatory feature is required for a user to consider using a product,

phone itself; other times you'll pay more by purchasing a second battery.

An intriguing and nonintuitive aspect of customer satisfaction is that sometimes the feature that provides the most satisfaction is one that customers didn't know they wanted until they saw it. For example, before a camera practically became a standard feature on cell phones, did you even know you wanted a camera on your cell phone? These types of features are known as excitors and are shown by the upward-pointing arrow that begins at the upper left of Figure 1. This arrow indicates that even a small amount of an excitor can dramatically influence customer satisfaction with the product. Mobile phone manufacturers, for example, started with low resolution, limited-capability cameras. That was enough at the time for the camera to be an exciting addition to a mobile phone. It is often worthwhile to include an excitor in a new product because customers will pay a premium for the exciting feature.

Let's look at how to determine whether a feature is mandatory, linear, or an excitor and consider advice on how to combine feature types into an optimal product.

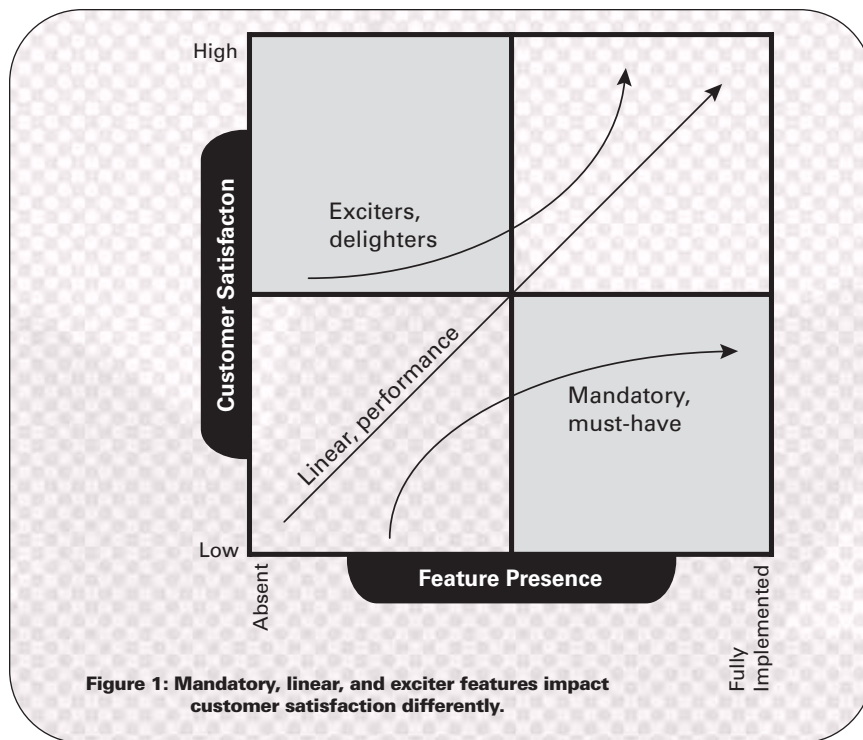


Figure 1: Mandatory, linear, and excitor features impact customer satisfaction differently.

analysis, after its originator Noriaki Kano. In this approach, features are segregated into three categories—mandatory, linear, and excitor. The differences between these types of features are shown in Figure 1.

Mandatory features are those that are required for a product to be competitive in its space. For a mobile phone, mandatory features are a speaker, a microphone, a numeric keypad, and so on. A manufacturer should not plan on building a phone that is missing any of these features. The line through the bottom right of Figure 1 shows that no matter how much of a mandatory feature is added, customer satisfaction never rises above the midpoint.

but more of the mandatory feature or a better implementation of it can only take customer satisfaction so far.

Linear features (shown by the diagonal line through the middle of Figure 1) are those for which customer satisfaction is directly related to the amount of the feature present. Your satisfaction with your cell phone increases linearly with the battery life of the phone. Similarly, the more minutes included in your monthly plan, the more satisfied you become. Linear features have a direct bearing on the price of a product. The longer a cell phone can run without having its battery recharged, the more you are willing to pay for the product. In some cases you may pay extra for the

### Categorizing Features

We can think about a feature and make a reasonable assessment as to whether it is mandatory, linear, or an excitor. If we rely solely on our own judgment, however, we will certainly make some mistakes. What is exciting to us may not be exciting at all to the prospective users of our product. This is especially true if our backgrounds and goals are not closely correlated to those of our product's users. We've seen too many products that developers deemed exciting but ended up being of no interest to users.

For these reasons, it is generally best to survey a set of prospective users about the features under consideration. I see a lot of surveys of this nature, and most of them make the same mistake. They pose the question to the user from a single perspective, usually asking something like "How important is this feature?" Users then are allowed to respond with a range of answers from "unimportant" to "very important."

Asking about feature importance in

this way provides limited information. For example, suppose a few years ago you had asked prospective mobile phone buyers “How important is having a camera in your mobile phone?” Most likely, they would have told you that it’s not very important. However, they might have finished the survey thinking, “I hope they include the camera; that would be great. What a wonderful new idea.” The feature is an exciter, even though a one-dimensional survey doesn’t reveal it.

pad is highly desirable for night use. From the responses to a single question, you would have no way of discerning that the backlit number pad is mandatory and that the camera is an exciter.

The best way to discern how a user feels about a feature is to ask what she thinks of the product if the feature is present and what she thinks of the product if the feature is not present. The first of these questions is known as the functional form, because it refers to the case when a

3. I am neutral.
4. I can live with it that way.
5. I dislike it that way.

Suppose we are building a reporting tool for users within our company and are contemplating four new features:

- The ability to export a graph or chart directly into a PowerPoint presentation
- The ability to schedule reports to run at certain times of the day
- The ability to save a report
- The ability to apply a style sheet to a report

To determine whether each of these is a mandatory feature, a linear feature, or an exciter, we ask prospective users:

- If you can export a graph or chart directly into PowerPoint, how do you feel?
- If you cannot export a graph or chart directly into PowerPoint, how do you feel?
- If you can schedule reports to run at certain times of the day, how do you feel?
- If you cannot schedule reports to run at certain times of the day, how do you feel?
- If you can save a report, how do you feel?
- If you cannot save a report, how do you feel?
- If you can apply a style sheet to a report, how do you feel?
- If you cannot apply a style sheet to a report, how do you feel?

The first pair of these questions and hypothetical answers from one user are shown in Figure 2.

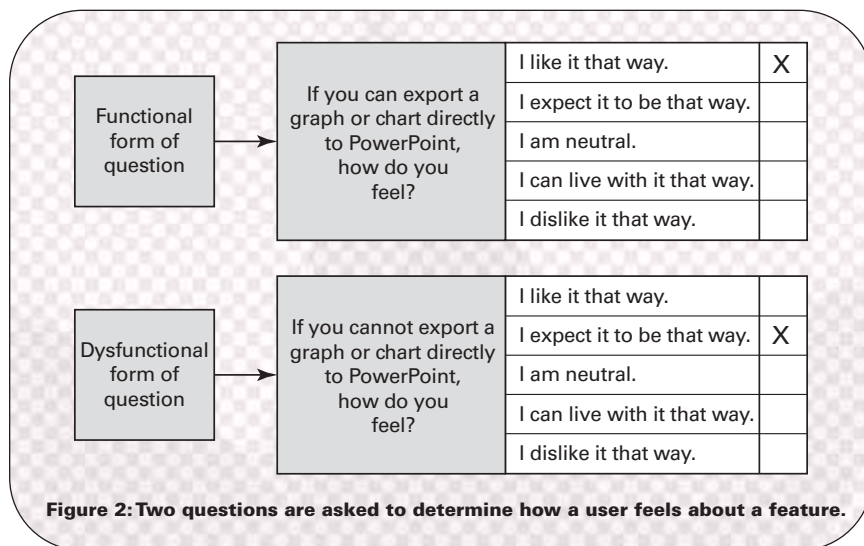


Figure 2: Two questions are asked to determine how a user feels about a feature.

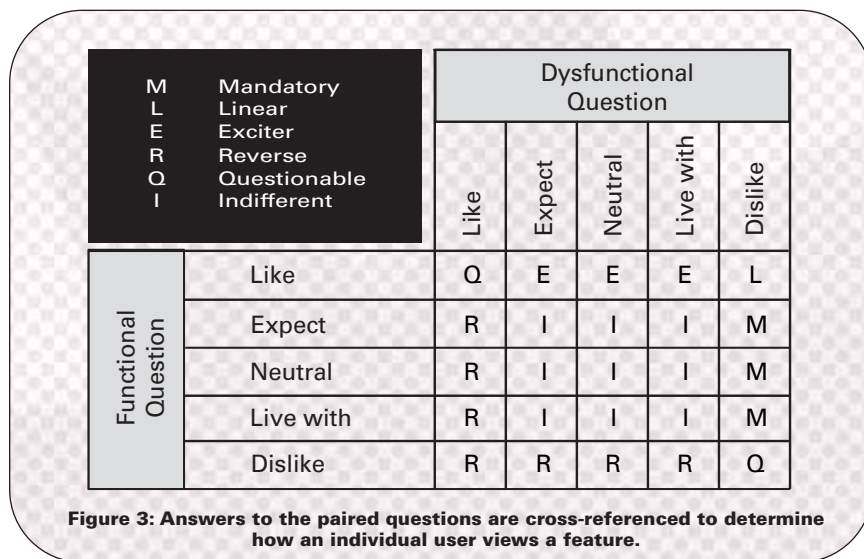


Figure 3: Answers to the paired questions are cross-referenced to determine how an individual user views a feature.

Suppose instead of asking how important a feature is, you ask how desirable it is. You would get different results and might find out that having a camera in a mobile phone is highly desirable. However, users would also respond that having a backlit number

function is present. The second question is known as the dysfunctional form, because it refers to the case when the function is not present. Each question is answered on the same five-point scale:

1. I like it that way.
2. I expect it to be that way.

### Categorizing Responses

Think about the pair of responses in Figure 2. The one user who answered these two questions has said that she’d like to be able to export directly to PowerPoint (the first question), but also that she does not expect the feature to be present. A feature that a user would like but does not expect to get is an exciter, an opportunity to add unexpected value.

Figure 3 provides a way of cross-referencing the functional and dysfunctional forms of a question pair, so that a

prospective user's responses can be reduced to a single meaning. We can cross-reference the answers shown in Figure 2 with Figure 3 and discern that the feature is an exciter. Similarly, if a user says that she expects to be able to save a report and dislikes it if she cannot, we can cross-reference those answers and see that saving a report is a mandatory feature.

In addition to the three feature types discussed thus far, Figure 3 introduces three other results—reverse, questionable, and indifferent. A reverse attribute indicates that the user would like the opposite of the feature being proposed. For example, suppose you think your e-commerce Web site should automatically log out a user after five minutes of inactivity. To determine how users feel about this question, you would ask these two questions:

- If you are automatically logged out after five minutes of inactivity, how do you feel?
- If you are not automatically logged out after five minutes of inactivity, how do you feel?

An indifferent feature is one the user doesn't care about. A lot of software programs (such as Word, which I used when writing this article) include the ability to split a window into multiple panes. I never use that feature, and if surveyed about it, I would respond that I am neutral regarding both its presence and its absence. Including an indifferent feature will not improve customer satisfaction.

### Aggregating Results

The individual results obtained by cross-referencing answer pairs in Figure 3 are aggregated to arrive at an overall category for each feature. After all, we're not as concerned with what individual users think as we are with what users think overall. Useful results often can be obtained by surveying as few as twenty or thirty likely users of your product. After results are tabulated, they are expressed in percentage terms and can be presented as shown in Table 1.

Table 1 shows the percentage of respondents who consider each feature an exciter, linear, mandatory, indifferent, reverse, or questionable. The final

her workgroup or department. Scheduling is critical for this type of user, and she considers the feature mandatory. The other type of user plans to run her own reports and usually no more than one or two a day. Because of her lighter use, this user would like the feature but does not consider it mandatory.

### Which Features to Include?

This article began with the claim that it is important to have a plan for achieving desired levels of customer satisfaction. Having come this far—surveying a set of prospective users, cross-referencing all answer pairs, and aggregating all results—how can we use this information to make sure we are planning a product that will satisfy our intended users?

First, all mandatory features must be included in the plan. Next, the product plan should include as many robustly supported linear features as possible. However, a little room should be left in the plan for at least a few excitors—they go a long way toward boosting customer satisfaction and often enable a product to be sold at a premium. **{end}**

FEATURE	EXCITER	LINEAR	MANDATORY	INDIFFERENT	REVERSE	QUESTIONABLE	CATEGORY
Export to PowerPoint	45%	10%	12%	30%	0%	3%	Exciter
Schedule reports	11%	41%	44%	2%	1%	1%	Linear/Mandatory
Save a report	0%	3%	95%	0%	0%	2%	Mandatory
Apply a style sheet	18%	15%	22%	43%	1%	1%	Indifferent

Table 1: Individual results are aggregated and an overall category selected.

Suppose a user answers that she is neutral about being automatically logged out, but she likes it if she is not. This user is telling you that you are contemplating a feature that will have a negative impact on her satisfaction with your product.

Questionable features result when it is unclear how the user feels about a feature. If you look at Figure 3, you'll notice that questionable results are obtained when the user likes it when a feature is both present and not present, or when she dislikes both its presence and its absence. Response pairs like these happen when someone gets confused while answering a survey.

column indicates the overall classification of the feature based on an interpretation of the results. Exporting a graph or chart to PowerPoint is an exciter based on a preponderance of user opinions.

Most of the responses indicate that the scheduling a report feature is mandatory. However, almost as many people considered it linear. Rather than simply calling the feature mandatory, a feature with two high responses often warrants a bit more thought. Results like this usually mean that two distinct audiences have been surveyed about the product. In this case, let's suppose one type of user will be running reports for

*Mike Cohn is the founder of Mountain Goat Software, a process and project management consultancy that specializes in helping companies adopt and improve their use of agile processes and techniques. He is the author of Agile Estimating and Planning and User Stories Applied: For Agile Software Development. Mike is a founding member of the Agile Alliance and serves on its board of directors. He is a technical editor for Better Software magazine, a regular columnist for the magazine and StickyMinds.com, and a frequent presenter at STAR and Better Software conferences. Mike can be reached at mike@mountaingoatsoftware.com.*