# Comparing User Stories With Other Techniques

CSCI 5828: Foundations of Software Engineering
Lecture 21 — 11/04/2014

## Goals

- Cover material from Fred Brooks's The Design of Design, Chapters 4-5
  - Finish the last two chapters from Part 1 of The Design of Design
- Return to our User Stories Book
  - Chapter 12: What User Stories Are Not
  - Chapter 13: Justification for Using User Stories
  - Chapter 14: A Catalog of Story Smells

## Requirements

- Brooks spends a few pages discussing requirements as input to design
    - He starts with a horror story of reviewing the requirements created for a new military helicopter
        - the requirements had been created by a committee representing multiple groups
            - the new helicopter was intended to replace FOUR existing helicopters
    - The requirements started with a list of features required for “mission support”
        - carry X people, avoid radar, be highly maneuverable
    - and then a last requirement
        - be capable of flying itself across the Atlantic (!)
- The latter requirement made no sense given the other requirements

## “Created by Committee”™

- This horror story is reflective of what happens in practice
    - Too many stakeholders at the table each with a wish list of features that need to go into the product
        - with no concern for conflicts and no discussion of constraints
    - Each person is invested in getting their list adopted and quickly the politics of the situation becomes “I’ll leave your list alone if you leave mine.”
- In this situations, few have the power to stand up for a set of requirements that will allow for a design with conceptual integrity
    - A designer arguing for such a thing will not have enough facts on his/her side to make a convincing argument => the product does not yet exist!
- With OS 360, Brooks was handed a “beast” like this and rejected it

## Requirements Bloat and Creep

- Brooks talks a bit about how to fight requirements creep (adding requirements throughout a design process) and bloat (having too many requirements at the start)
    - He references successful projects from the past that had
        - one or two clear overriding objectives AND schedule urgency
    - The latter was key; by having a pressing deadline people were more attuned to focus on the essence of what needed to be present in order to meet the objectives
        - It thus became clear what was in scope and what was out-of-scope
    - To fight creep, **requirements traceability** was cited as a useful technique (although it goes against Agile); here Brooks is saying that it is useful to be able to trace each feature back to its design and transitively back to its requirements and transitively again back to the original objectives
        - if you can’t do that, you’ve discovered a feature that “creeped in”

## Sin (a misnamed section)

- In a somewhat unfortunately named section, Brooks sets the stage for the role that contracts play in commercial requirements, design, and development
    - He postulates an “ideal” scenario for building a house
        - A client ready to pay his/her architect and builder fair rates for their expertise and labors
        - An architect who never fails at representing his/her client
        - A builder who implements a set of blueprints perfectly producing a high-quality product with the best possible value/cost ratio, within budget, and on schedule
        - All the players are honest and communication is excellent

## Sin (still a misnamed section)

- If those things are true, then Brooks claims that the best approach is that
  - The client pays the architect and builder for their costs plus a small percentage for their profit; this provides the best value per dollar
  - An iterative design-build process is the best way to then transform the design into the final product while meeting the client's needs
    - Brooks recommends Boehm's Spiral Model which is essentially a point in between the waterfall model and agile life cycles
      - Recall, the spiral model is an iterative waterfall driven by risk analysis
- The problem? It is almost impossible to achieve the conditions on the previous slide; communication is hard, players may employ a variety of strategies for how they interact, all sorts of problems can crop up that are external to the process, etc. This is just being realistic not due to "sin"

## Contracts

- Brooks then discusses how the fact that since an ideal “requirements/design/implementation” situation is hard to achieve, we need contracts to make people (and more importantly) organizations behave
- Two interesting items emerge from this short section
    - The first: Brooks thinks that the need for contracts is a driving force for why the waterfall model has dominated the software industry for so long (essentially 45 years)
    - The second: he discusses how large-scale buildings actually get built (page 47) and that this process exhibits an agile like quality:
        - not everything can be specified all at once, so architects will design those systems that construction needs first: foundations, site work, and long lead-time components (steel)
            - the builders can get busy while the architects move on to the next stage of design; agile plays a similar role, I think, in software

## Are There Better Design Process Models?

- In Chapter 5 of The Design of Design, Brooks asks the question
  - Are there models other than the Rational Model or even the Waterfall model for doing software design?
- He only gives a cursory surface-level treatment of this question
  - The best part, however, is his observations related to Eric Raymond's paper The Cathedral and the Bazaar
    - The open source model is powerful since it is inherently iterative
    - The open source model of prestige works because the participants are otherwise fed (i.e. employed)
    - The process is better at producing tools/frameworks than applications
    - There is still typically a designer (Torvalds for Linux) and a spec (Unix)
    - This process works because the builders of the systems are also the users; that is, open source is developers building for developers

## Transition to Agile

- That concludes are review of Part 1 of The Design of Design
  - We'll return to it later this semester and sample from additional chapters
- Now, we return to our Agile textbook and look at additional issues related to user stories
  - In particular,
    - what user stories are not
    - why user stories are effective
    - how user stories can be misused

## What User Stories Are Not

- There are other ways to perform requirements specification
    - The book discusses briefly three of them and shows how they are different from the approach taken by user stories
        - IEEE 830 software requirements specifications
        - use cases
        - interaction design scenarios

## IEEE 830 (I)

- The IEEE has previously published a standard on how to write requirements specifications, known as IEEE 830
    - It covers
        - how to organize the requirements specification document
        - the roll of prototypes
        - the characteristics of good requirements
- The defining structure of a good IEEE functional requirement is that it starts with the phrase "The system shall…"
    - 23.1 The system shall allow a company to pay for a job posting with a credit card
        - 23.1.1 The system shall accept Visa and Mastercard but not Discover Card

## IEEE 830 (II)

- The problem?
  - This type of specification is tedious to write, easy to get wrong, very time consuming, and impossible to keep consistent
    - They are also boring to read and can thus fail to serve their purpose
    - (We used to force students to write these documents in SE classes)
- While there may be tremendous appeal to the idea that you can write all requirements up front, the problem is that the needs of the users shift too rapidly and too often for this to be realistic
  - With user stories, you document a few needs and **then go build something**; this act of building clarifies the needs and ensures that the requirements are actually surfaced and understood

## IEEE 830 (III)

- Another problem is that “shall”-style requirements do not clearly convey a user’s goals
  - The author presents an example of IEEE requirements that clearly indicate that the system is about cars
    - but then he shares the user’s goal and it concerns the ability to quickly take care of large front and back yards, i.e. a riding lawn mower
- User stories do not capture user goals directly but they will be written in such a way as to keep the goals visible
  - The Homeowner can easily adjust the height of the lawn mower.
  - The Lawn Service can specify a speed for the lawn mower (via an accelerator)
- User stories incorporate the domain into the statements to help maintain the user perspective in statements of desired functionality

## Use Cases (I)

- We encountered use cases back in lecture six when I used them as an example with respect to writing styles for requirements
    - A use case is a generalized description of a set of interactions between a user and a system
        - The use case specifies an interaction that will allow a user to achieve one of his/her goals
            - The use case typically contains extensions that allow a user to recover from error conditions and still achieve their goal
    - Use cases are designed to slot into an object-oriented analysis and design process; they are ideal for identifying candidate objects that need to be present in a system under design and for then identifying the interactions those objects have with the user and each other

## Use Cases (II)

- Use cases are not the same thing as user stories
  - User stories do not specify work flow or scenarios
    - They capture statements of functionality
      - Such a statement might capture the functionality of a single scenario contained in a use case (such as the success case or one of the extension cases)
  - Thus it is sometimes claimed that “user story + test cases” == “use case”
    - There is some truth to this but the purposes are different
      - use cases are analysis artifacts meant to be kept around a long time
      - user stories are “functionality placeholders” used for planning and scheduling and which serve as reminders to hold a conversation with the user about the associated feature

## Interaction Design Scenarios

- The final requirements technique compared with user stories is interaction design scenarios
    - These are detailed scenarios that describe potentially many users interactions with a system
        - They are meant to “set the stage” for a design process that produces candidate user interfaces that can support the detailed interactions
- Such a scenario will contain a setting, actors (always human users), goals and objectives, and a variety of actions and events
    - While this is useful for UI design, it should be clear that they are different from user stories; user stories target specific functionality and contain details that allow that functionality to be tested
        - One interaction scenario might encompass multiple user stories

## Why User Stories?

- In this chapter, Mike Cohn spends some time identifying the advantages of user stories over alternative approaches for requirements specification
    - Emphasize verbal communication (in line with Agile's values)
    - Comprehensible by a wide range of stakeholders
    - Designed for planning
    - Support iterative development
    - Encourage deferring detail (until we have more information)
    - Support opportunistic design
    - Encourage participatory design
    - Build up tacit knowledge

## Verbal Communication (I)

- User stories shift the focus in software development from
  - shared documents
- to
  - shared understanding
- We want to be able to focus on what users want
  - this focus can get lost if we require our developers to write down their interpretation of what users want

## Verbal Communication (II)

- With an emphasis on verbal communication we avoid the problems of imprecise language
    - With your dinner you have a choice of soup or salad and bread
        - (Soup or Salad) and Bread? Soup or (Salad and Bread)
    - Buffalo buffalo buffalo <- legal English sentence (use of obscure verb)
        - Buffalo buffalo Buffalo buffalo <- so is this one
    - The system should prominently display a warning message whenever the user enters invalid data
        - “should enter” — is this required or optional? We don’t know
- Instead, user stories encourage conversation to allow us to resolve ambiguities and document details in tests

## Comprehensible

- Users stories are so simple in their statements of functionality that
    - they are easily understood by a wide range of stakeholders
        - including (most importantly) non developers
        - and in comparison to IEEE 830 and use cases
        - they avoid technical jargon because we write them from the user's perspective
    - after a few examples, users can start generating their own stories

## Planning

- User stories are at the right level of granularity for planning
    - Each one represents a feature that can be implemented on its own
        - Test cases are specified to let us know when we are done
        - Details are added to the code and the test cases rather than to requirements/design documents
- IEEE 830 requirements are too small and interdependent
    - The author reports that customers will typically say that 90% of a IEEE 830 requirements specification are mandatory
        - making it hard for the developers to know where to start
- Use cases are often too big, containing multiple scenarios that encompass many features

## Iterative Development

- User stories are flexible enough for iterative development
    - We can start with just the functionality that user needs first
        - New stories can be added at an “epic” level when the user mentions an area of functionality that is eventually needed
            - and then decomposed in later iteration when the time is right and more information is available
- IEEE 830 documents have a hard time with this due to its facade of being complete (no need to add detail later, we did it all up front!)
- Scenarios also try to capture lots of detail up front and thus are not ideal for starting with what we know and adding detail later
- Use cases CAN support progressive levels of detail but are often used in traditional life cycles that then encourage IEEE 830 style “completness”

## Deferring Detail

- User stories are designed to be placeholders
    - "We know we need something like this…"
- We can generate a bunch of stories up front to provide developers with the sense of a system
    - The "low hanging fruit" have a bunch of details and we can start with them
    - The others can be captured at a high level and addressed later
- In this way, user stories let developers get to design/implementation much more quickly than other techniques

## Opportunistic Development (I)

- Our themes collide: in this section Cohn sites the Parnas and Clements “How and Why to Fake It” paper on rational design
  - Cohn states that it is tempting to think we can build systems in a top-down manner (the rational design process); but Parnas and Clements state
    - Users and customers do not generally know exactly what they want
    - Even if we knew all the requirements, many design details do not become apparent until implementation has begun
    - Even if we knew all the design details, humans encounter difficulties understanding/managing that many things at once
    - Even if we could, products and requirements CHANGE
    - Finally, people make mistakes

## Opportunistic Development (II)

- Instead, developers (and designers) work more opportunistically, shifting from
  - thinking about requirements
  - to thinking about usage scenarios
  - to thinking about useful abstractions
  - to implementing code
  - until a solution emerges
- User stories are designed to support this opportunistic process
  - they embrace change and do not rely on developers/users to know everything up front

## Participatory Design

- Participatory design is studied in human-centered computing courses
    - It attempts to get users deeply involved in producing the software tools they will use in their daily work
        - Came originally from work done in Scandinavia
- It stands in contrast to empirical design in which designers/developers observe users in their work place and make requirements/design decisions based on those observations
- As previously stated, user stories are easy for users to understand and easy for users to create and so they encourage a more participatory style of design in general software development

## Tacit Knowledge

- With a focus on conversation and shared understanding
  - user stories promote the accumulation (and distribution) of knowledge about a project across the entire team
- If you spend all of your time talking about your project
  - and your project has no internal boundaries (i.e. code ownership issues)
- then over time, the entire team builds up deep knowledge of the project in general and the software in particular
- User stories support this by avoiding having most of the knowledge in documents and more in the heads of the developers and the code/tests

## User Story Limitations

- Mike Cohn identifies three potential drawbacks of user stories
    - On a large project with lots of stories, it can be difficult to understand the relationships between stories
        - which stories are related to one another? which ones will impact each other?
            - Cohn states that user roles can help address this to a certain extent
    - It’s hard to do requirements traceability with user stories
        - This makes sense; requirements traceability comes from a desire to establish linkages between (lots of) documents; it is antithetical to agile life cycles
    - User stories (and Agile) may not scale to large teams
        - at a certain point, using conversation to distribute knowlege will not work for large teams and shared documents will be needed

## Story Smells (I)

- Chapter 14 of our Agile text book presents a set of “smells” that identify “bad stories” or a misunderstanding of how to use stories
    - The use of the term “smells” for this originated in a book on Refactoring by Martin Fowler et al. and is attributed to Kent Beck
        - The idea is that you encounter code that is so bad it makes you say “ew” as if you had just encountered a bad smell
    - In that book, a “code smell” (such as duplication of code) would trigger a refactoring that would leave the code in a better state while maintaining its functionality
- By analogy, our textbook tries to identify problems that can crop up with the use of user stories and presents techniques that can be used to address them

## Story Smells (II)

- **Stories are too small**
    - Lots of small stories with estimates that turn out to be wrong leading to a need to revise estimates frequently
        - Often two small stories are similar; once the first is complete, the second is trivial
    - Combine stories seeking a new statement of the functionality that allows a good chunk of work to be allocated to it
- **Interdependent Stories**
    - You can't add one story to an iteration because it depends on another story and that story depends on another story (etc.)
    - Typically related to small stories; need to combine the stories so that interdependencies are tackled as a group

## Story Smells (III)

- **Goldplating**
  - Developers are adding features that were not planned for the iteration or are interpreting stories too broadly and going beyond what was needed to implement the required functionality
    - Also known as “speculative complexity”
  - This is an agile training issue; focus on simplicity; puts you in a better position to embrace change WHEN IT HAPPENS
    - One technique to combat it, is to raise the level of visibility associated with tasks; if a developer continually does too much (in the sense of implementing things that are not needed) it will get noticed and self correct

## Story Smells (IV)

- **Too Many Details**
  - Too much time is being spent on details well in advance of a story being implemented or more time is spent writing stories than discussing them
  - Mandate the use of physical cards for a while
    - this will constrain the amount of space you have to specify details until people get used to how much information should go into them
    - Still have the problem? Use a smaller card!
- **Including User Interface Detail Too Soon**
  - Stories written early in the project include detail about user interfaces
  - This practice unnecessarily constrains the design; let these details emerge via the prototyping process; keep the required functionality independent of the UI used to access it

## Story Smells (V)

- **Thinking Too Far Ahead**
  - Symptoms include
    - not having enough space on the card for story details
      - and a related request to move to an electronic system for tracking stories
    - proposing story templates to capture all the details of a story
    - or a request to make estimates in finer precision
  - This is common on teams that are used to “up front requirements” and a rational design process
    - More agile training is needed; perhaps switch to a shorter iteration length for two iterations to show the benefits of conversation + prototypes + customer feedback

## Story Smells (VI)

- **Splitting Too Many Stories**
  - Too many epic stories were created and need to be split in order to get them into an iteration
  - Comes from a lack of training and experience; need to learn how to have discussions that add clarity to epics and allow them to be decomposed
    - Sometimes you have to act opportunistically
      - you get a detail from the customer on a story that is not currently active; take a bit of time away from the current iteration to update the epic story to reflect knowledge of the new information
        - this involves decomposing the story ahead of time
        - and will reduce the need for splitting stories only during iteration planning

## Code Smells (VII)

- **Customer Has Trouble Prioritizing**
  - Likely indicates that stories are
    - too big (“Can you do a little of this? And a little of that?”)
    - or do not contain business value
      - (“I’m not sure why I need a thread pool?”)
- **Customer Won’t Write and Prioritize the Stories**
  - Some corporate cultures encourage people to avoid taking responsibility for things (they don’t want to be blamed if it fails)
  - In these situations, you have to find middle ground that allows information to be gathered while addressing their fears
    - Cohn indicated that he would tell them that he was “gathering information” and he would take responsibility if something goes wrong

## Summary

- In this lecture, we've made progress on our agile and design themes
- From Brooks
  - The role of requirements in design and the problems that can occur
  - Are there alternatives to the rational model?
    - which provided useful observations on the open source model
- From Cohn
  - How do user stories stack up against alternative techniques?
  - What are some of the problems that can arise when using user stories and how can you address them?

## Coming Up Next

- Lecture 22: User Stories, Chapters 15-16
- Lecture 23: The Design of Design, Chapter 6-16