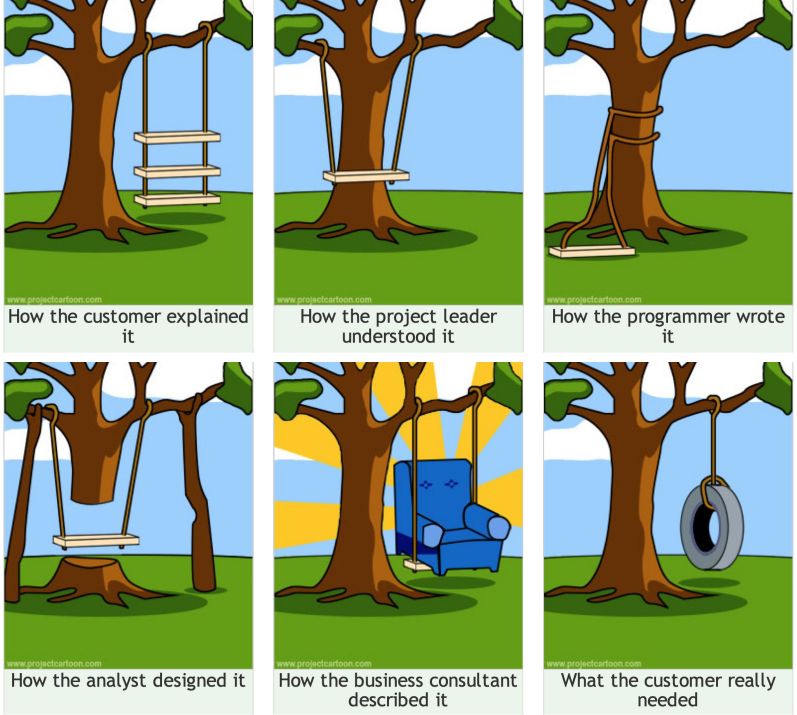
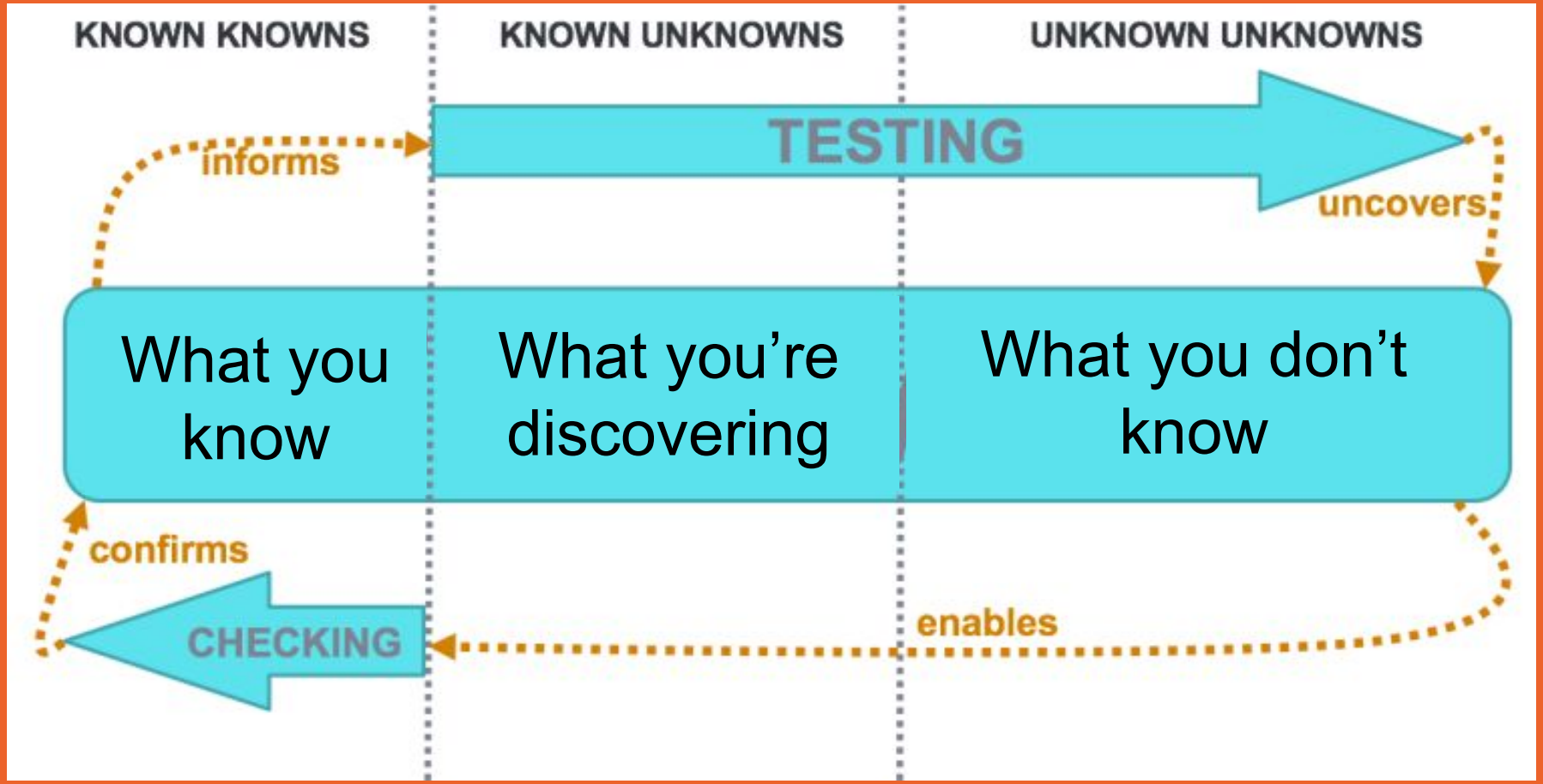
Requirements including User Stories



Elisa Baniassad, Ivan Beschastnikh

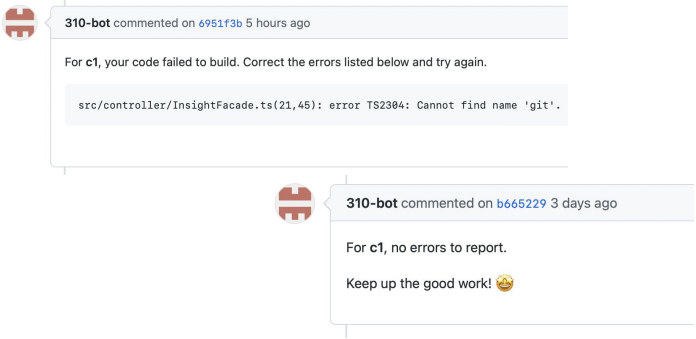
But, first,

- **C1 feedback reminder**
- **Quiz 1 ongoing!**
- **CBTF Quiz review: Mondays**
  - You have to sign up



[https://findingdeefex.com/2016/05/20/the-testing-checking-synergy/]

1. Dev branch feedback



2. Main branch feedback: smoke tests

2. *#check* (on any branch)

We have a new bot command: #check! 🎉

#check will provide feedback on your test suite:

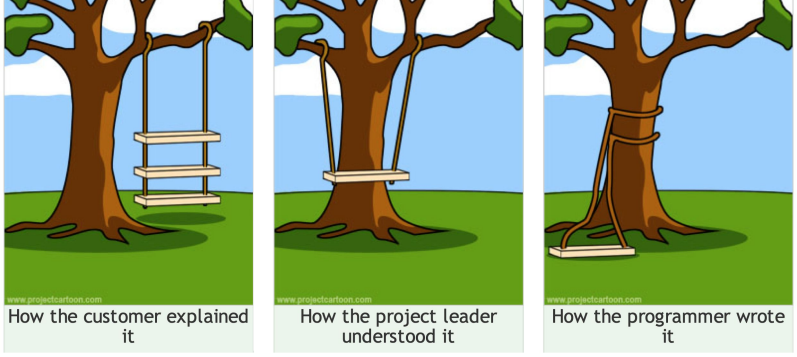
1. **Missing files**. Any missing files required for tests.
2. **Test feedback**. How your tests ran against our implementation.
3. **Performance Hints**. If the test suite took too long, some hints on what can be improved.

Submissions are limited like C1 - #check can be run once per 6 hours per person. For example if Bob and Alice are on a team, Bob can run it at 12pm and then Alice at 1pm. However, Bob must wait until 6pm and Alice until 7pm to run it again.

More detail is given in the new #check portion of the spec.

https://sites.google.com/view/ubc-cpsc310-22w2/project/checkpoint-1?authuser=0#h.wrmsa0sjodzm

Requirements including User Stories



Elisa Baniassad, Ivan Beschastnikh

Waterfall Process

Failure

Shipping

$$$$$$$

Coding

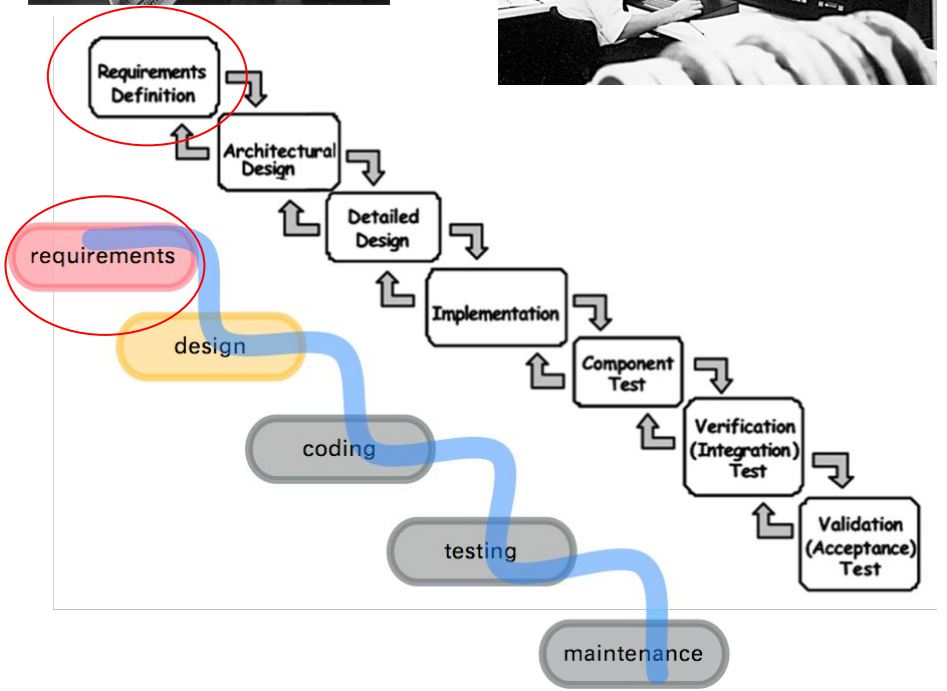
Testing

Compiling

Cost

$

Planning



Pictures are not to scale

Requirements size aligns with technological and economic changes

Large requirements (large requirements specifications or formal specs) -- needed for situations where coding, linking, building, are expensive. (still true for e.g., aircraft, medical devices)

Medium requirements (use cases) -- came in when coding, linking, building and shipping became cheaper, but were NOT free

Small requirements (user stories) -- an option finally when coding, testing, shipping are effectively free (now, humans are the most expensive element)

Requirements: Starting point

When computation was expensive, people took a long time planning their implementation, and that involved writing lengthy requirements.

Another reason was that systems were applied to high-risk problems, so fully specifying the requirements was important.

Hence:

> Requirements used to be very very large documents
>
> And used to be written very formally
>
> Actually they still are, in situations where life is at risk!

https://www.nap.edu/read/2222/chapter/16#160

First came big requirements

NASA

National Aeronautics and Space Administration

NSTS 08271

Lyndon B. Johnson Space Center

Houston, Texas 77058

SPACE SHUTTLE
FLIGHT SOFTWARE
VERIFICATION AND VALIDATION REQUIREMENTS
NOVEMBER 21, 1991

FOREWORD

Efficient management of the Space Shuttle program dictates that effective control of program activities be established. To provide a basis for management of the program requirements, directives, procedures, interface agreements, and information regarding system capabilities are to be documented, baselined, and subsequently controlled by the proper management level.

Program requirements to be controlled by the Director, Space Shuttle (Level I), have been identified and documented in Level I program requirements documentation. Program requirements controlled by the Deputy Director, Space Shuttle Program (Level II), are documented in, attached to, or referenced from Volume I through XVIII of NSTS 07700.

This document, which is to be used by members of the Flight Software community, defines the Space Shuttle Program baseline requirements for the Flight Software Verification and Validation process. All Flight Software Verification and Validation activity should be consistent with this plan and the unique items contained herein. The top level policies and requirements for Flight Software Verification and Validation are contained in NSTS 07700, Volume XVIII, Computer Systems and Software Requirements, Book 3, Software Management and Control.

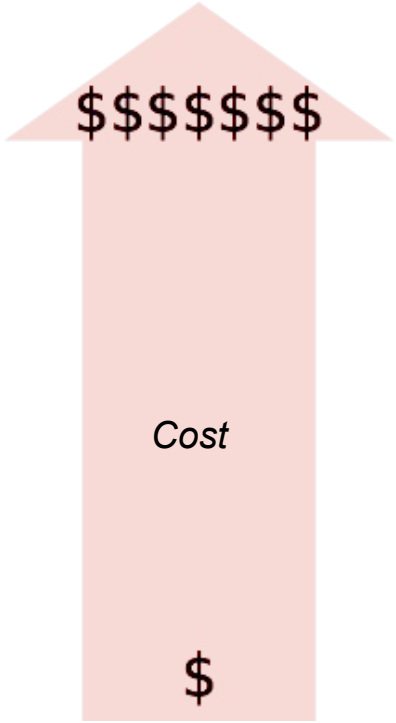
Problems with big requirements

- Very difficult for a client to play out the behaviour based on the description because the description is so in-depth
- These are long, almost legalese documents that take a long time to convert into a specification or detailed design

Dawn of Extreme Programming

COMMUNICATION
SIMPLICITY
FEEDBACK
COURAGE

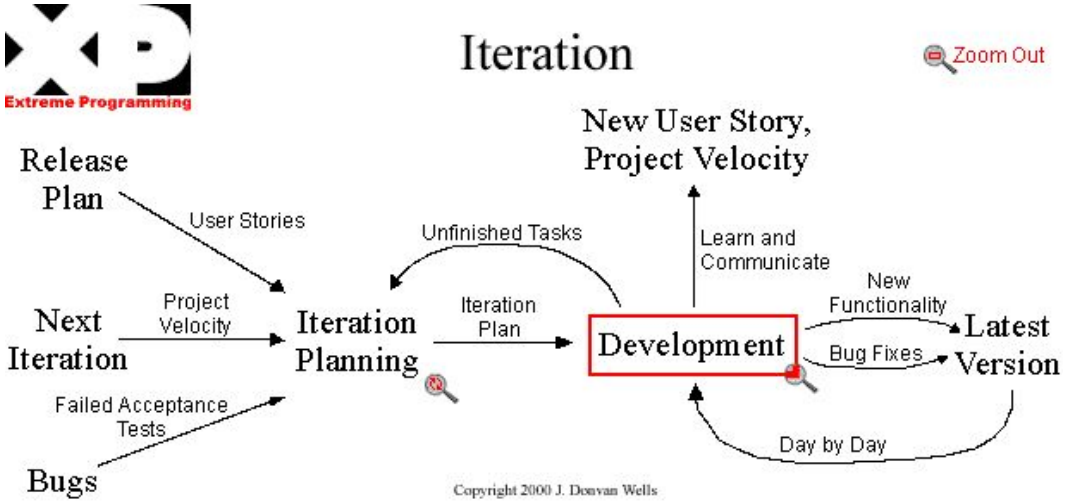
Courage: *We will tell the truth about progress and estimates. We don't document excuses for failure because we plan to succeed. We don't fear anything because no one ever works alone. We will adapt to changes when ever [sic] they happen.*



Failure
(used in less critical applications)
Shipping

Coding Testing Compiling

Planning



Pictures are not to scale

Then came medium requirements: USE CASES

Use Case 1: Buy something

Context of use: Requestor buys something through the system, gets it.

Scope: Corporate - The overall purchasing mechanism, electronic and non-electronic, as seen by the people in the company.

Level: Summary

Preconditions: none

Success End Condition: Requestor has goods, correct budget ready to be debited.

Failed End Protection: Either order not sent or goods not being billed for.

Primary Actor: Requestor

Trigger: Requestor decides to buy something.

Main Success Scenario

1. **Requestor**: initiate a request
2. **Approver**: check money in the budget, check price of goods, *complete request for submission*
3. **Buyer**: check contents of storage, find best vendor for goods
4. **Authorizer**: *validate approver's signature*
5. **Buyer**: *complete request for ordering, initiate PO with Vendor*

And on...

6. Vendor: deliver goods to Receiving, get receipt for delivery (out of scope of system under design)

7. Receiver: *register delivery*, send goods to Requestor

8. Requestor: *mark request delivered*.

Extensions

1a. Requestor does not know vendor or price: leave those parts blank and continue.

1b. At any time prior to receiving goods, Requestor can change or cancel the request.
Canceling it removes it from any active processing. (delete from system?)
Reducing price leaves it intact in process.
Raising price sends it back to Approver.

2a. Approver does not know vendor or price: leave blank and let Buyer fill in or call back.

2b. Approver is not Requestor's manager: still ok, as long as approver signs

2c. Approver declines: send back to Requestor for change or deletion

3a. Buyer finds goods in storage: send those up, reduce request by that amount and carry on.

3b. Buyer fills in Vendor and price, which were missing: gets resent to Approver.

4a. Authorizer declines Approver: send back to Requestor and remove from active processing.

5a. Request involves multiple Vendors: Buyer generates multiple POs.

5b. Buyer merges multiple requests: same process, but mark PO with the requests being merged.

6a. Vendor does not deliver on time: System does *alert of non-delivery*

7a. Partial delivery: Receiver marks partial delivery on PO and continues

7b. Partial delivery of multiple-request PO: Receiver assigns quantities to requests and continues.

8a. Goods are incorrect or improper quality: Requestor does *refuse delivered goods*. (what does this mean?)

8b. Requestor has quit the company: Buyer checks with Requestor's manager, either *reassign Requestor*, or return goods and *cancel request*.

And on!

Deferred Variations
none

Project Information

Priority	Release Due	Response time	Freq of use	
Various	Several		Various	3/day

Calling Use Case: none
Subordinate Use Cases: see text
Channel to primary actor: Internet browser, mail system, or equivalent
Secondary Actors: Vendor
Channels to Secondary Actors: fax, phone, car
Open issues

When is a canceled request deleted from the system?
What authorization is needed to cancel a request?
Who can alter a request's contents?
What change history must be maintained on requests?
What happens when Requestor refuses delivered goods?

Problems with medium requirements

- Still difficult for a client to play out the behaviour based on the description because the description is so in-depth
  - Fairly formal descriptions (algorithmic)
- Still interconnected -- they would refer to one another! “Buy Something” might refer over to “Procure goods”
- Would weave together multiple roles: requestor/buyer/vendor/…

Then came smaller requirements: SMALLER use cases

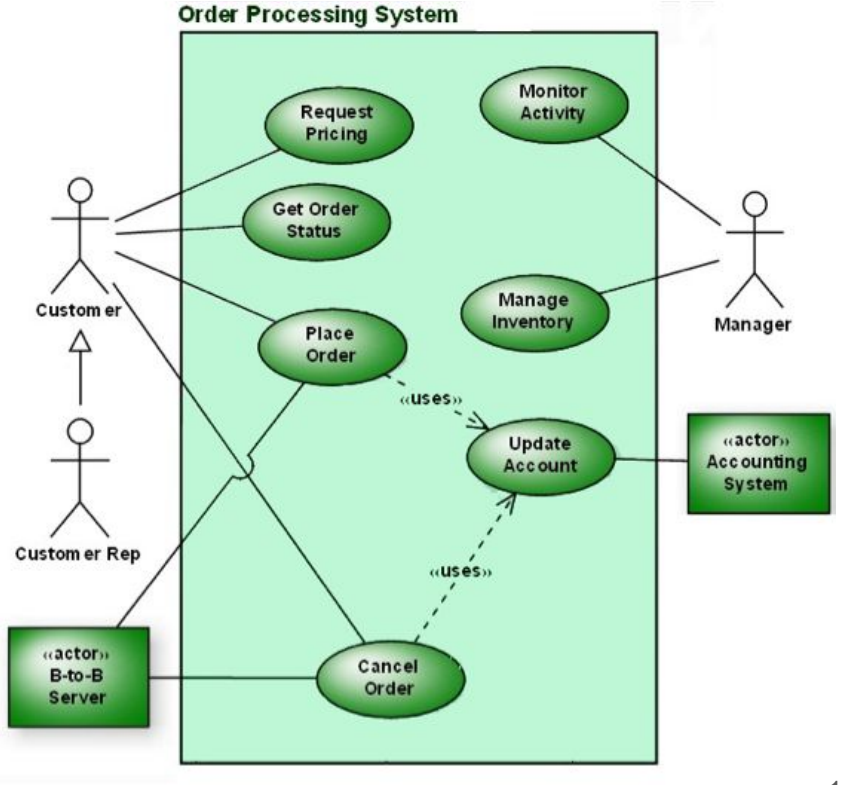
Use Case 1: Buy something

The Requestor initiates a request and sends it to her or his Approver. The Approver checks that there is money in the budget, check the price of the goods, completes the request for submission, and sends it to the Buyer. The Buyer checks the contents of storage, finding best vendor for goods. Authorizer: validate approver's signature . Buyer: complete request for ordering, initiate PO with Vendor. Vendor: deliver goods to Receiving, get receipt for delivery (out of scope of system under design). Receiver: register delivery, send goods to Requestor. Requestor: mark request delivered..

At any time prior to receiving goods, Requestor can change or cancel the request. Canceling it removes it from any active processing. (delete from system?) Reducing the price leaves it intact in process. Raising the price sends it back to Approver.

Then PICTORIAL requirements:

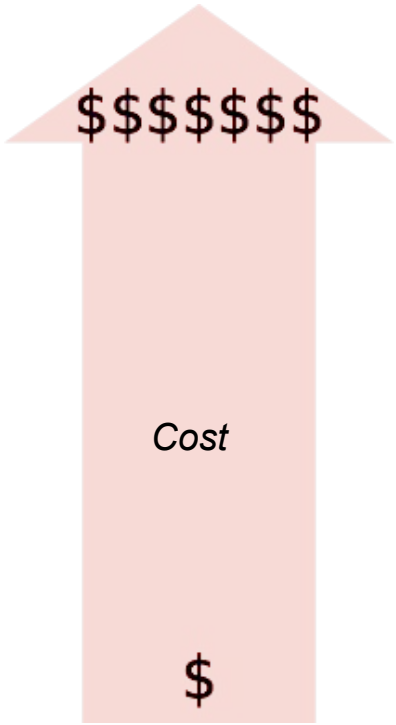
- **Use case diagrams** show packaging and decomposition of use cases not their content
- **Each ellipse is a use case**
  - Only top-level services should be shown
  - Not their internal behaviour
- **Actors can be other systems**
- The system (black outline) can be an actor in other use case diagrams
- Are not enough by themselves
  - Must individually document use cases



Problems with smaller use case requirements

- STILL difficult for a client to play out the behaviour based on the description because the description is so in-depth, and now leaving room for ambiguity … what do each of the behaviours really look like in the end?
- This still contributes to a mismatch between client expectations and what the developer does
- Does not link problem domain to solution domain explicitly
    - Problem domain: The needs of the client
    - Solution domain: The implementation (how client needs will be satisfied)

Agile Development



Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Failure (in non-critical systems)

Planning
Coding Testing
Compiling
Shipping

User Stories
(lightweight specs)

Test Driven Dev

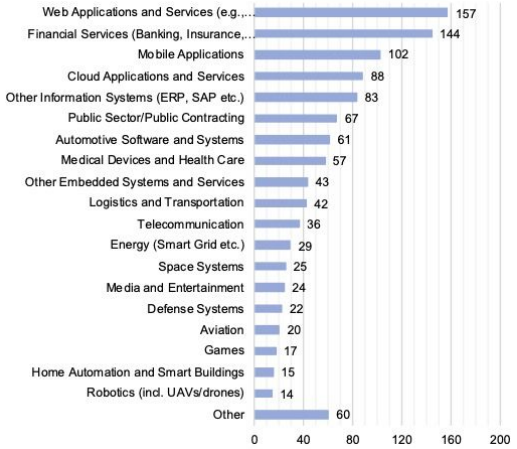
Pictures are not to scale

Agile adoption: it's complicated

What Makes Agile Software Development Agile?

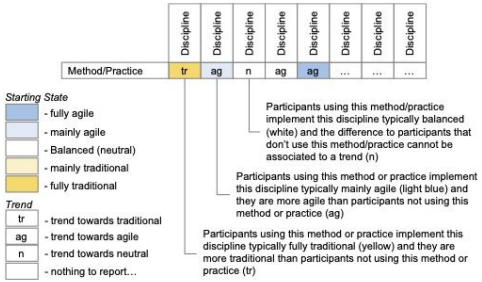
Marco Kuhrmann, Paolo Tell, Regina Hebig, Jil Klünder, Jürgen Münch, Oliver Linssen, Dietmar Pfahl, Michael Felderer, Christian R. Prause, Stephen G. MacDonell, Joyce Nakatumba-Nabende, David Raffo, Sarah Beecham, Eray Tüzün, Gustavo López, Nicolas Paez, Diego Fontdevila, Sherlock A. Licorish, Steffen Küpper, Günther Ruhe, Eric Knauss, Özden Özcan-Top, Paul Clarke, Fergal McCaffery, Marcela Genero, Aurora Vizcaino, Mario Piattini, Marcos Kalinowski, Tayana Conte, Rafael Prikladnicki, Stephan Krusche, Ahmet Coşkunçay, Ezequiel Scott, Fabio Calefato, Svetlana Pimonova, Rolf-Helge Pfeiffer, Ulrik Pagh Schultz, Rogardt Heldal, Masud Fazal-Baqaie, Craig Anslow, Maleknaz Nayebi, Kurt Schneider, Stefan Sauer, Dietmar Winkler, Stefan Biffl, Maria Cecilia Bastarrica, and Ita Richardson

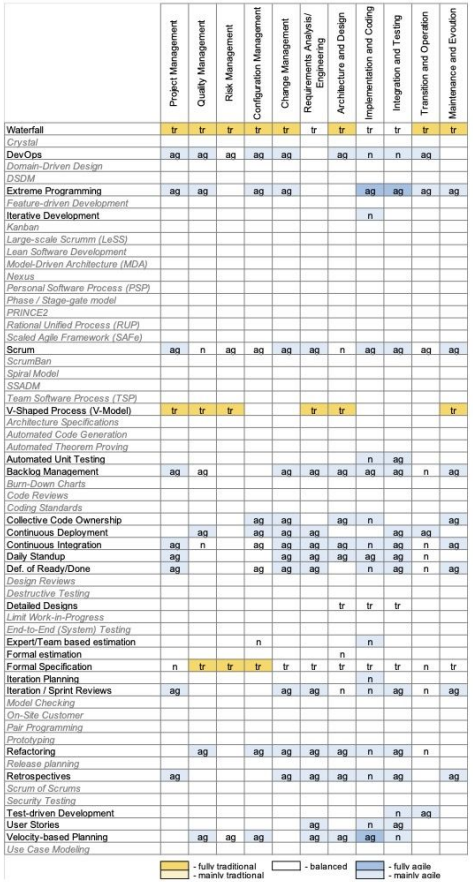
Abstract—Together with many success stories, promises such as the increase in production speed and the improvement in stakeholders' collaboration have contributed to making agile a transformation in the software industry in which many companies want to take part. However, driven either by a natural and expected evolution or by contextual factors that challenge the adoption of agile methods as prescribed by their creator(s), software processes in practice mutate into hybrids over time. Are these still agile? In this article, we investigate the question: what makes a software development method agile? We present an empirical study grounded in a large-scale international survey that aims to identify software development methods and practices that improve or tame agility. Based on 556 data points, we analyze the perceived degree of agility in the implementation of standard project disciplines and its relation to used development methods and practices. Our findings suggest that only a small number of participants operate their projects in a purely traditional or agile manner (under 15%). That said, most project disciplines and most practices show a clear trend towards increasing degrees of agility. Compared to the methods used to develop software, the selection of practices has a stronger effect on the degree of agility of a given discipline. Finally, there are no methods or practices that explicitly guarantee or prevent agility. We conclude that agility cannot be defined solely at the process level. Additional factors need to be taken into account when trying to implement or improve agility in a software company. Finally, we discuss the field of software process-related research in the light of our findings and present a roadmap for future research.



Finding 5a: Methods and practices have a stable influence towards either a high or low degree of agility, which does not change with the project discipline.

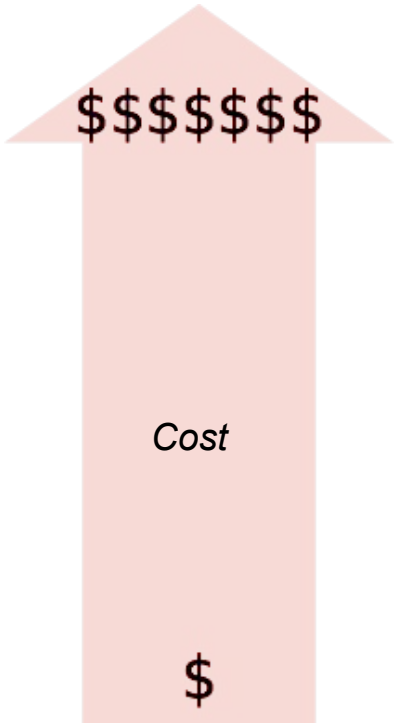
Finding 5b: No method or practice determines whether a project is traditional or agile, i.e., any method or practice can be found in traditional and agile development.





Full paper @ https://homepages.ecs.vuw.ac.nz/~craig/publications/tse2021-kuhrmann.pdf

Agile Development



Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

Responding to change over following a plan

Failure (in non-critical systems)

Planning
Coding Testing
Compiling
Shipping

User Stories
(lightweight specs)

Test Driven Dev

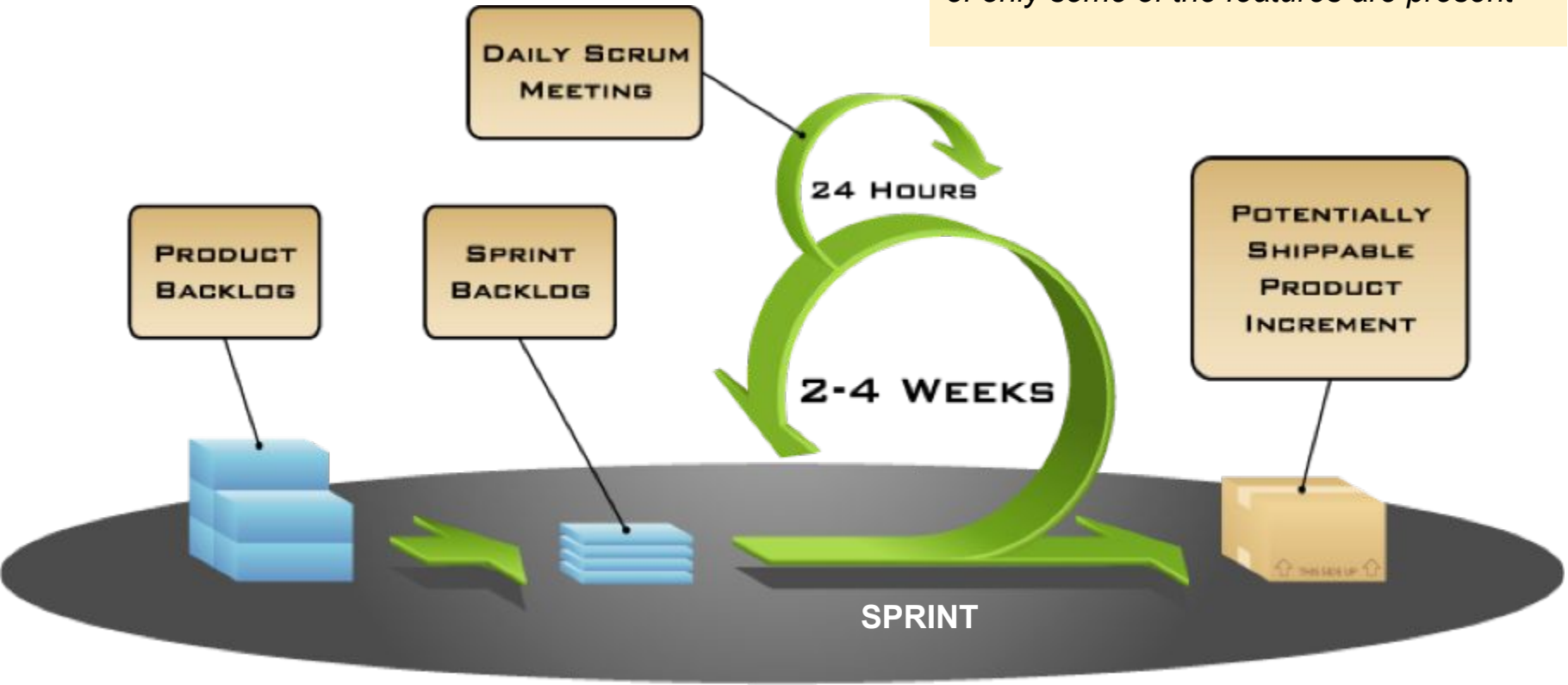
Pictures are not to scale

SMALL REQUIREMENTS:

USER STORIES

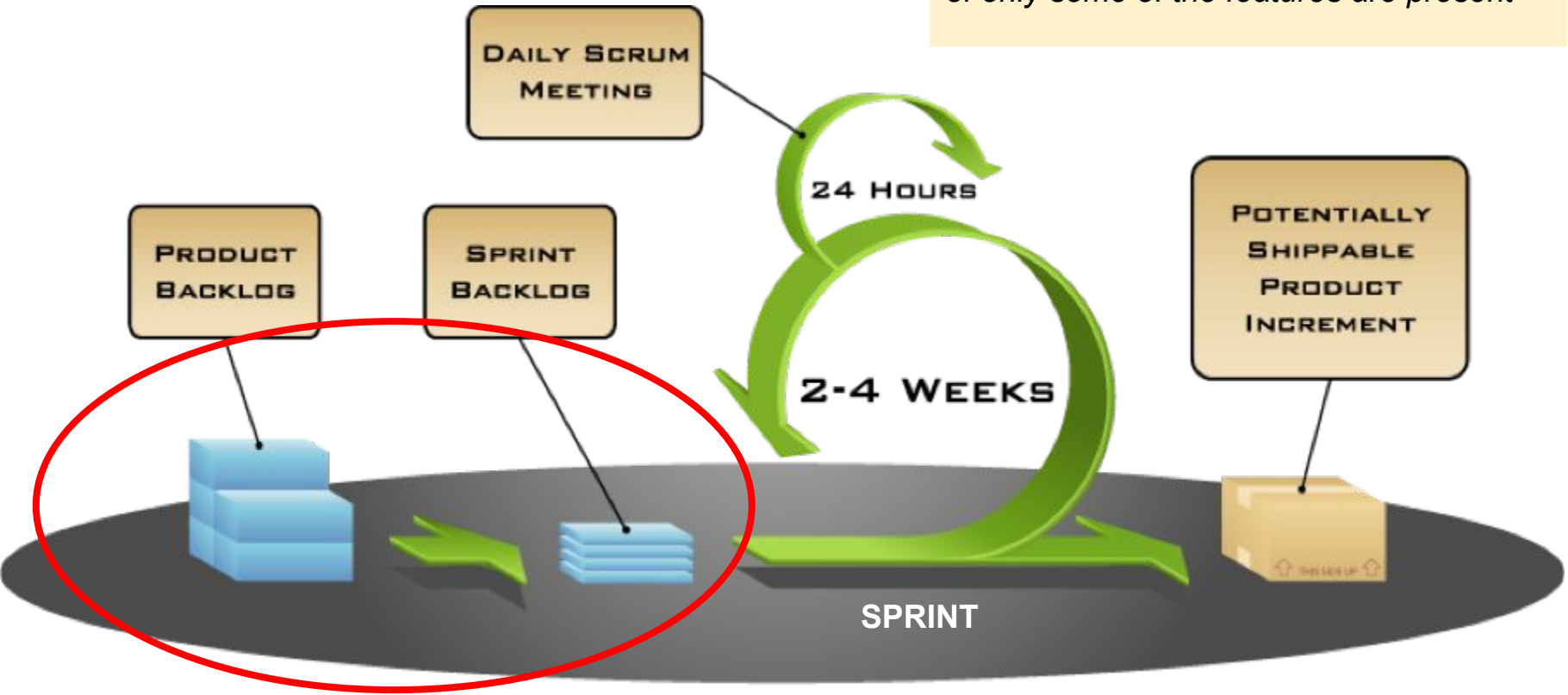
Scrum Timeline

Note that you are always iterating from working product, to working product, even of only some of the features are present



Scrum Timeline

Note that you are always iterating from working product, to working product, even of only some of the features are present



USER STORIES

Role Goal Benefit

Sometimes just called the “User Story”

Definitions of Done

Sometimes called Acceptance Criteria
(Solution to Role-Goal-Benefit)

Engineering Tasks

Contract:

USER STORIES

Role Goal Benefit

Sometimes just called the "User Story"

Definitions of Done

Sometimes called Acceptance Criteria
(Solution to Role-Goal-Benefit)

Engineering Tasks

USER STORIES

Role Goal Benefit

Sometimes just called the “User Story”

Definitions of Done

Sometimes called Acceptance Criteria
(Solution to Role-Goal-Benefit)

Engineering details:

Engineering Tasks

User story examples

RGB: As a shopper, I want to be able to buy something and then see it in my purchased list so that I can spend money on the site.

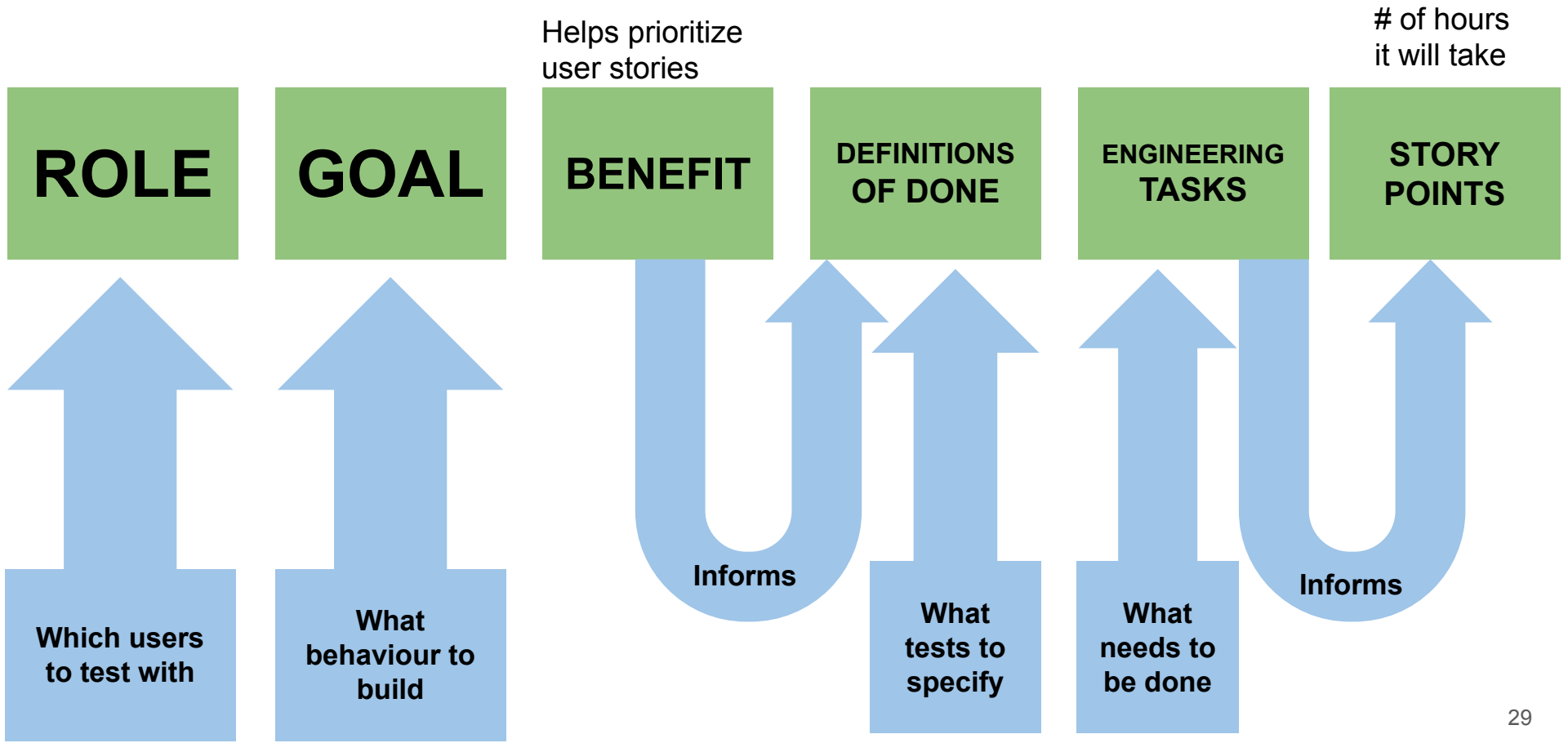
Definition of done: User clicks the button buy, and it appears in their purchased items, and is shipped to their home and the user will see the money deducted from their account

BAD USER STORY:

Role Goal Benefit: **As a buyer,** When I'm told that I'm not approved for purchase by the system, I want to be able to click "request approval" ***(solution domain!)*** and then receive confirmation that the approval request has been sent. *(no benefit! How valuable is this?)*

Definition of done: User is seeing "not approved, and clicks "request approval", and this triggers a react function which makes user's ID appear in the list of approval requests, and an email is sent back to the user that their request is in processing *(DoD is user-level: should not mention code; DoD is client-oriented solution domain)*

Connecting user stories + soft. engineering



Role Goal Benefit Statement

Have a **Role** (the specific type of user expressing the need)

Have a **Goal** (the behaviour desired)

Have a **Benefit** (the outcome of the behaviour)

- As a user, I want to search for contacts so I can message them.
- As a customer, I want to search for product items, so I can buy them.
- As an employer, I want to post a job on the website so people can apply for it.

RGB statement:

As a customer,

I want to be able to buy something

and then get it

<u>Not</u> Role Goal Benefit statements:

- implement contact list view ContactListView.java
- define the product table database schema
- automate the job posting algorithm
- Refactor the code to make it more readable

These are **engineering tasks**

Good definitions of done

These are your *contracts* with your clients

This is how you know you have completed a user story, and can mark it resolved

It's like a sequence diagram that explains how the features plays out

> This is how you know whether you can test your user story (if you can't, you need a different user story!!)

Definition of done: User clicks the button buy, and the item appears in their purchased items, and is shipped to their home.

https://www.mountaingoatsoftware.com/blog/the-difference-between-a-story-and-a-task

Engineering Tasks

User stories are then broken down by developers into engineering tasks.

These are NOT from a user perspective -- they are just things that need to be done to get the work completed (finish the parser; investigate the JSON library; set up the database; setup mocks for testing; etc)

Based on those tasks, the developers estimate how much time the story will take.

Estimating Story Points

burn down chart

A story point basically corresponds to an hour of developer work

Estimation traditionally was made by a developer, guessing (based on experience) how long it would take them to do a story.

Estimations are trending to now be based on classifications:

	Single location	Multiple locations
Simple change	1	2
Complex change	3	5

Where each of these ratings would have some standard number of hours associated

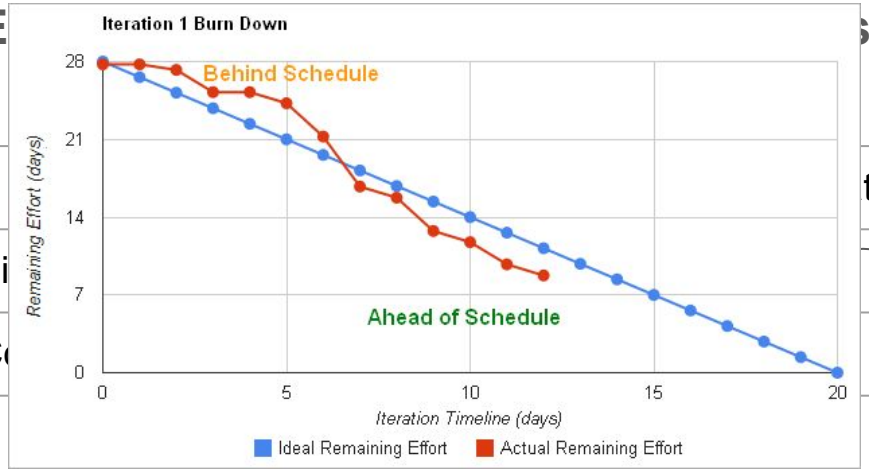
Estimating Story Points

burn down chart

A story point basically corresponds to an hour of developer work

Estimation traditionally was made by a developer, guessing (based on experience) how long it would take them to do a story.

E **sed on classifications:**



tions

Si

C

Where each of these ratings would have some standard number of hours associated

Case of a mechanism for estimation at a local company:

1 - Trivial cosmetic change in very few places
2 - Many trivial changes across a project (following existing patterns, adding properties to existing objects etc.)
3 - Changes that require a new design or concept and are straight forward to implement in the existing design
5 - Changes that require a new design or concept and require some rework of the existing design OR are very complicated.
8 - Changes that require a new design or concept and require some rework of the existing design AND are very complicated.

Why Fibonacci?

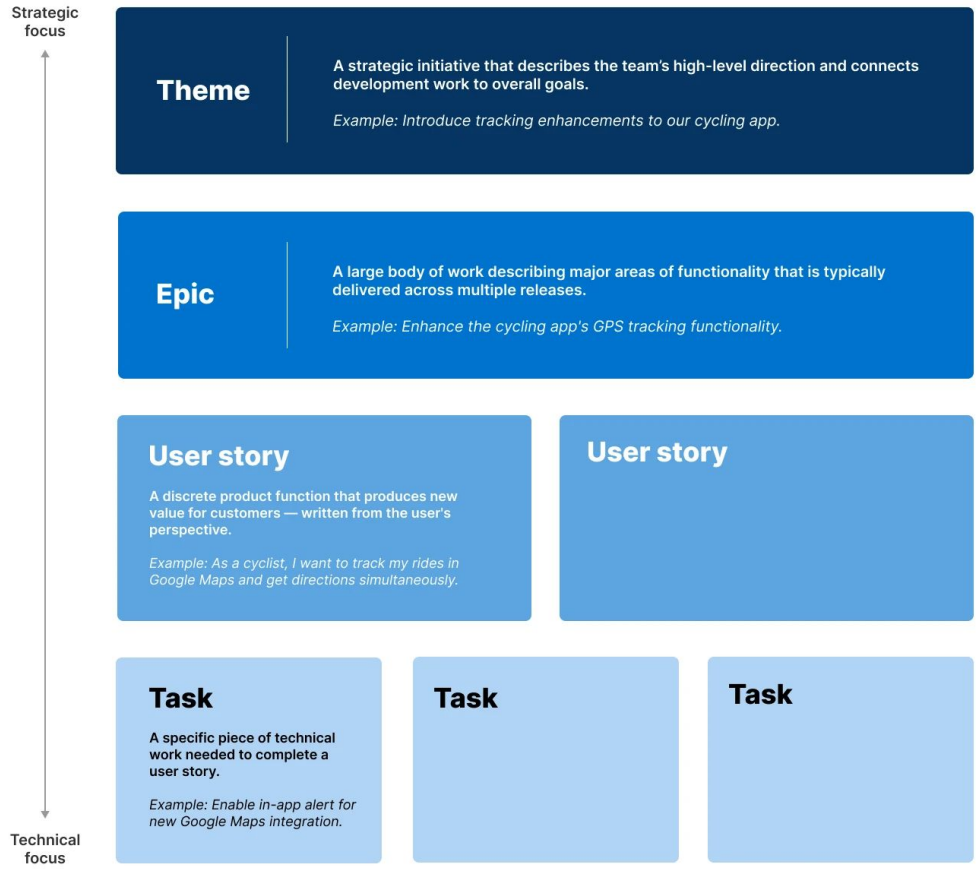
> *I think if it was linear, it wouldn't represent how complex some things are, that being said I think it's more important for the team to have a unified definition of what each number represents. Generally we never pull in [to the sprint] things that are 8. We would instead pull in **research tickets** [AKA: a **research spike**] or break down the work. An 8 in our world is an indicator that too much is unknown*

Good things about small requirements

- Beautiful linkage between problem domain and solution domain because definitions of done describe the solution succinctly
- These are still legal documents!
- They do not have hierarchy the way prior requirements did (though clusters of them might make sense)
- Great way to distribute work between team members!

From *user stories* to *epics* to *themes*

- User stories are independent, but often related
- Epics group together multiple related user stories
  - Usually delivered over multiple sprints
  - Grouping of stories that share an overall goal
- Themes group epics, and describe even higher-level objectives



https://www.aha.io/roadmapping/guide/agile/themes-vs-epics-vs-stories-vs-tasks

Functional vs. non-functional requirements

- Functional requirements: what we have see so far!
  - Specifies what the system should do: inputs/outputs/behaviors
- Non-functional requirements: *properties* that the product must have
  - Usually described using adjectives
  - Capture the experience a user might have

Non-functional requirement examples:

- **Security**: confidentiality, integrity, availability
- **Reliability**: uptime, fault tolerance
- **Privacy**: anonymity, tracking
- **Performance**: scalability, response time, capacity
- **Legal** or regulatory: GDPR
- **Usability**: effort to learn, use, interpret

Functional vs. non-functional requirements

- Functional requirements: what we have see so far!
  - Specifies what the system should do: inputs/outputs/behaviors
- Non-functional requirements: *properties* that the product must have
  - Usually described using adjectives
  - Capture the experience a user might have

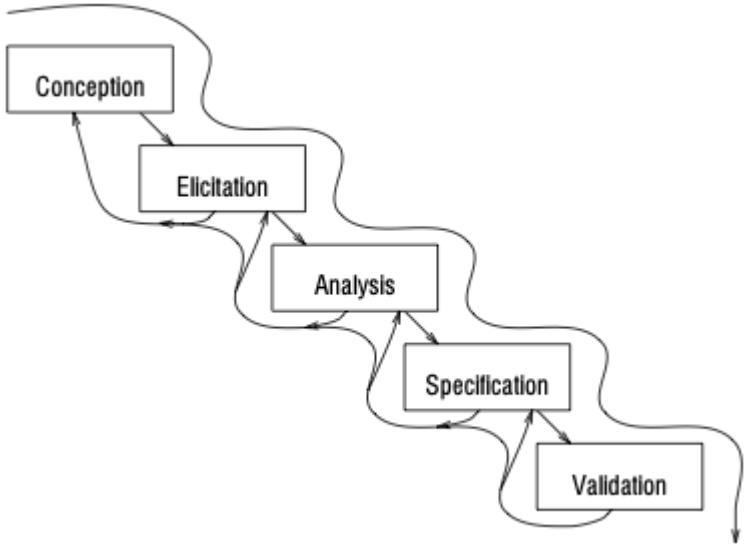
Non-functional requirement examples:

- **Security**: confidentiality, integrity, availability
- **Reliability**: uptime, fault tolerance
- **Privacy**: anonymity, tracking
- **Performance**: scalability, response time, capacity
- **Legal** or regulatory: GDPR
- **Usability**: effort to learn, use, interpret

The key to high quality *non-functional* requirements are **measurable** objectives.

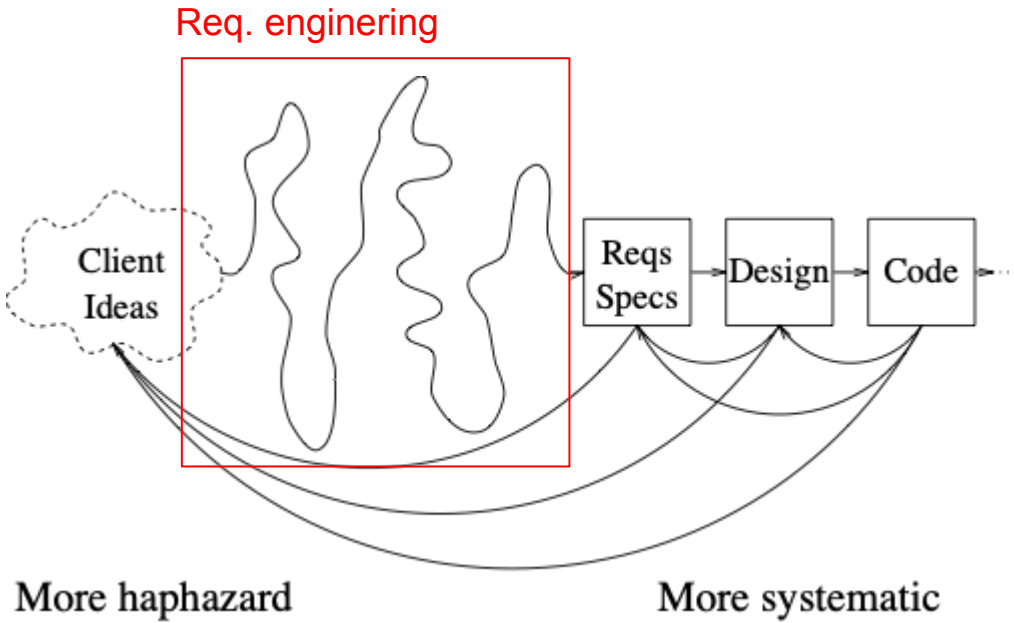
Requirement enginering lifecycle

- Elicitation:
  - The process by which requirements are gathered
  - Using whatever sources of information are available: client, users, observation, videos, documents, interviews, etc.
- Validation
  - Have we elicited and documented the right reqs?

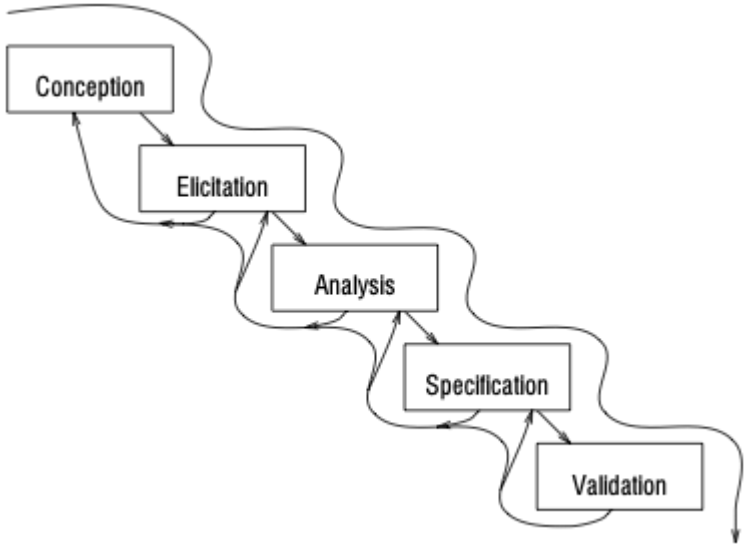


https://cs.uwaterloo.ca/~dberry/ATRE/Slides/IcebergSlides.pdf

Requirement enginering lifecycle



Reality



Idealistic RE process

https://cs.uwaterloo.ca/~dberry/ATRE/Slides/IcebergSlides.pdf