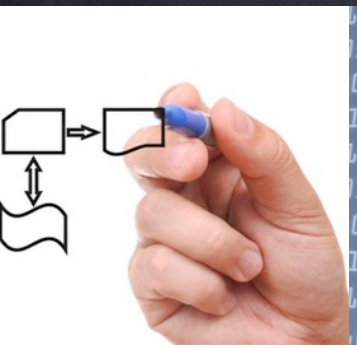
San Jose State University

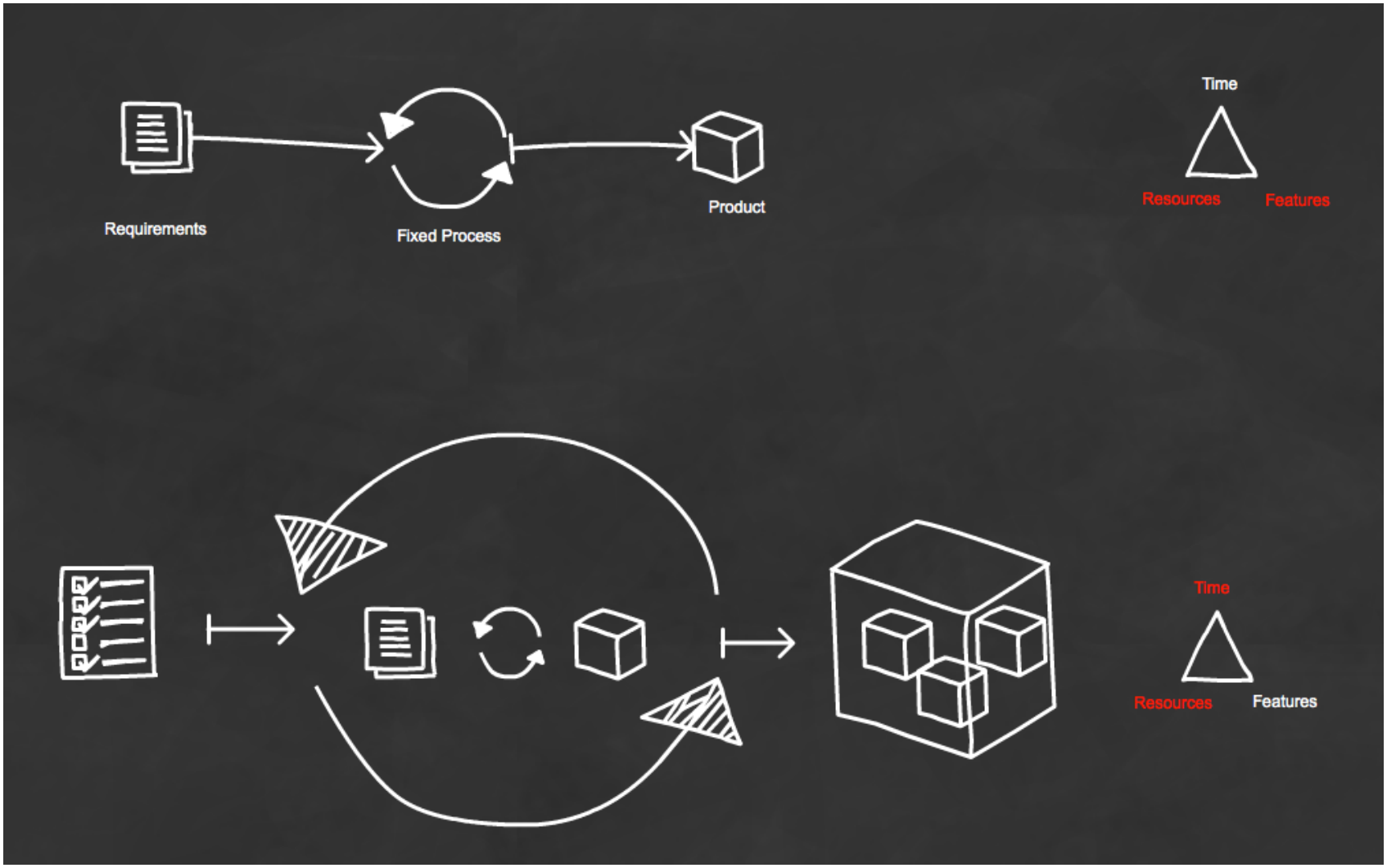
CMPE 202

Software Systems Engineering

Agile Software Engineering

Paul Nguyen

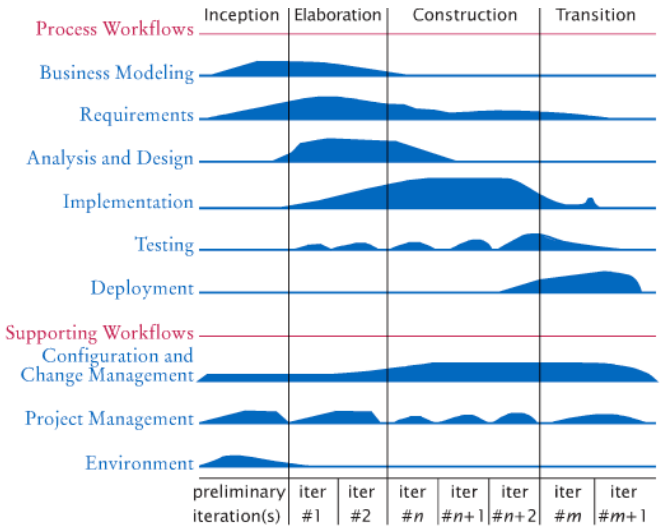


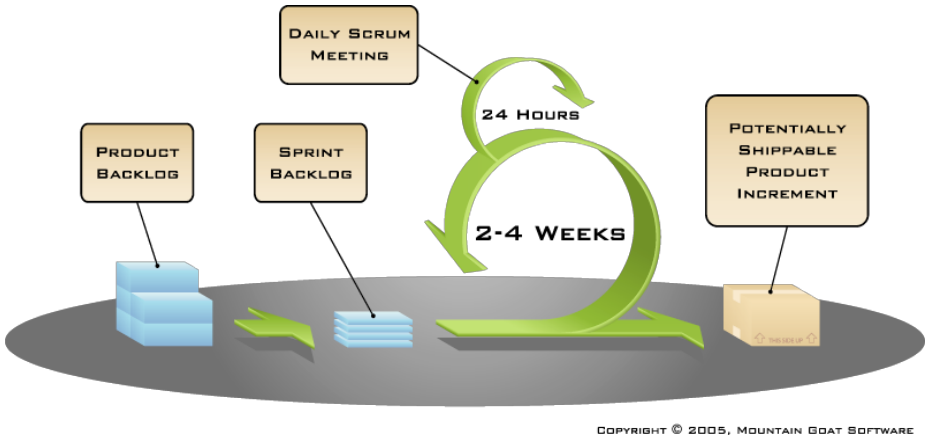



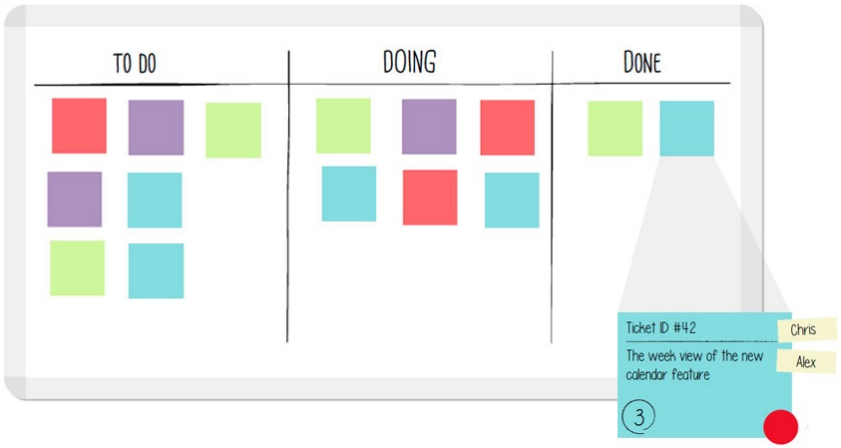
Key Ideas:

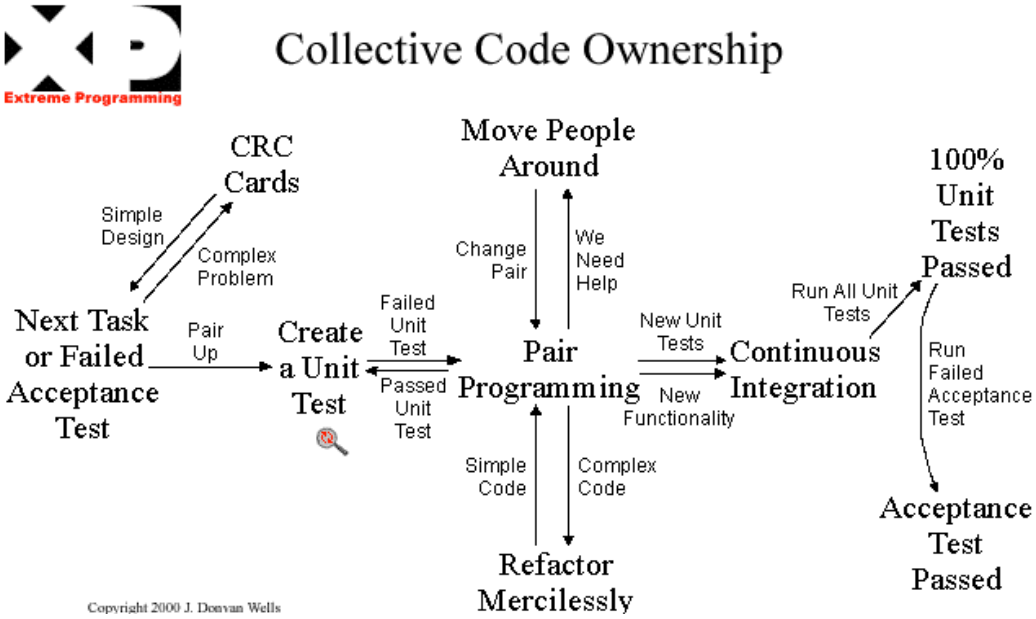
1. **Evolve** Requirements (specifications)
2. **Iterative** Development Cycles (repeat the process)
3. Deliver Software in **Increments** (in functionally independent "chunks")

Quick Sampling of Agile Methodologies

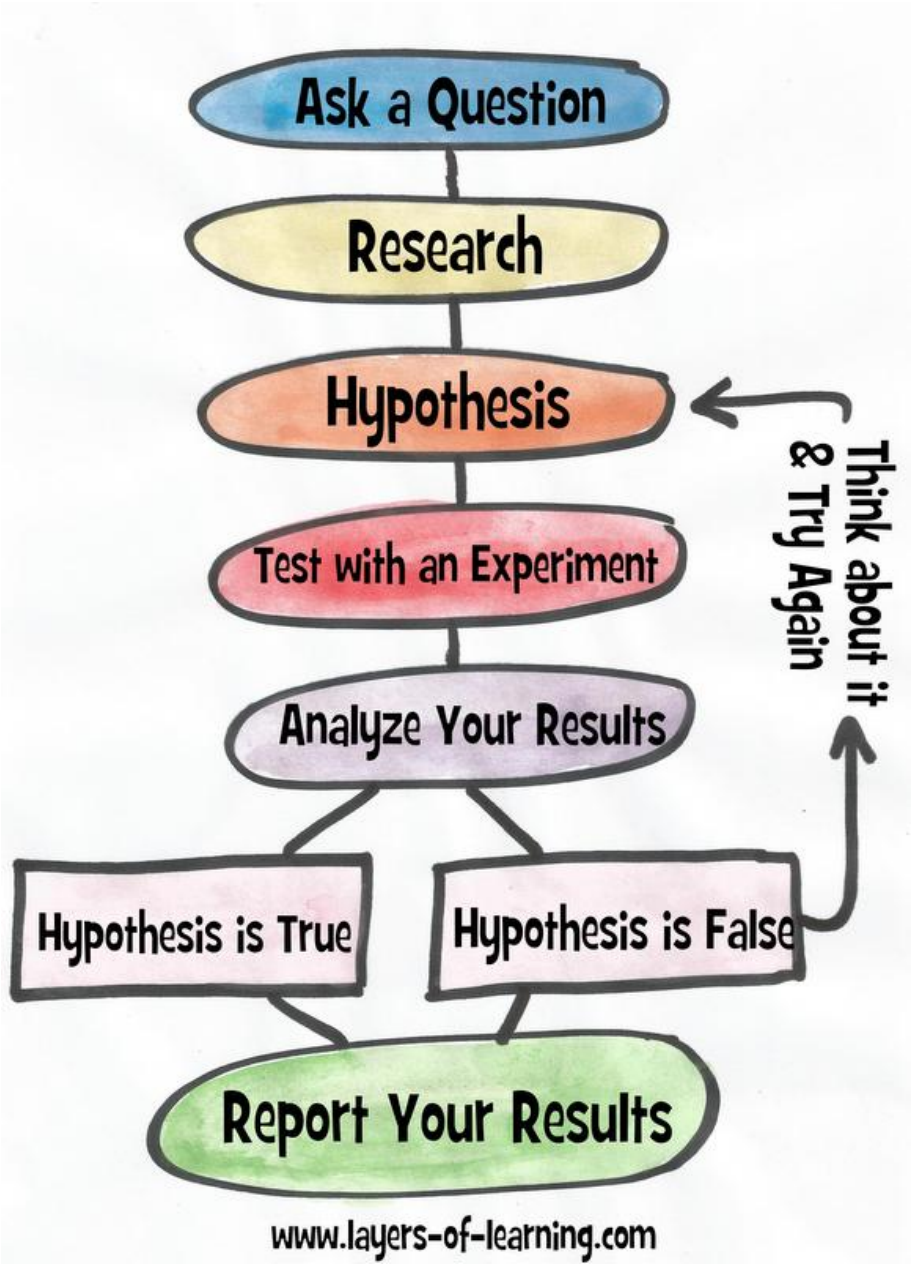








The Essence of Agile Processes

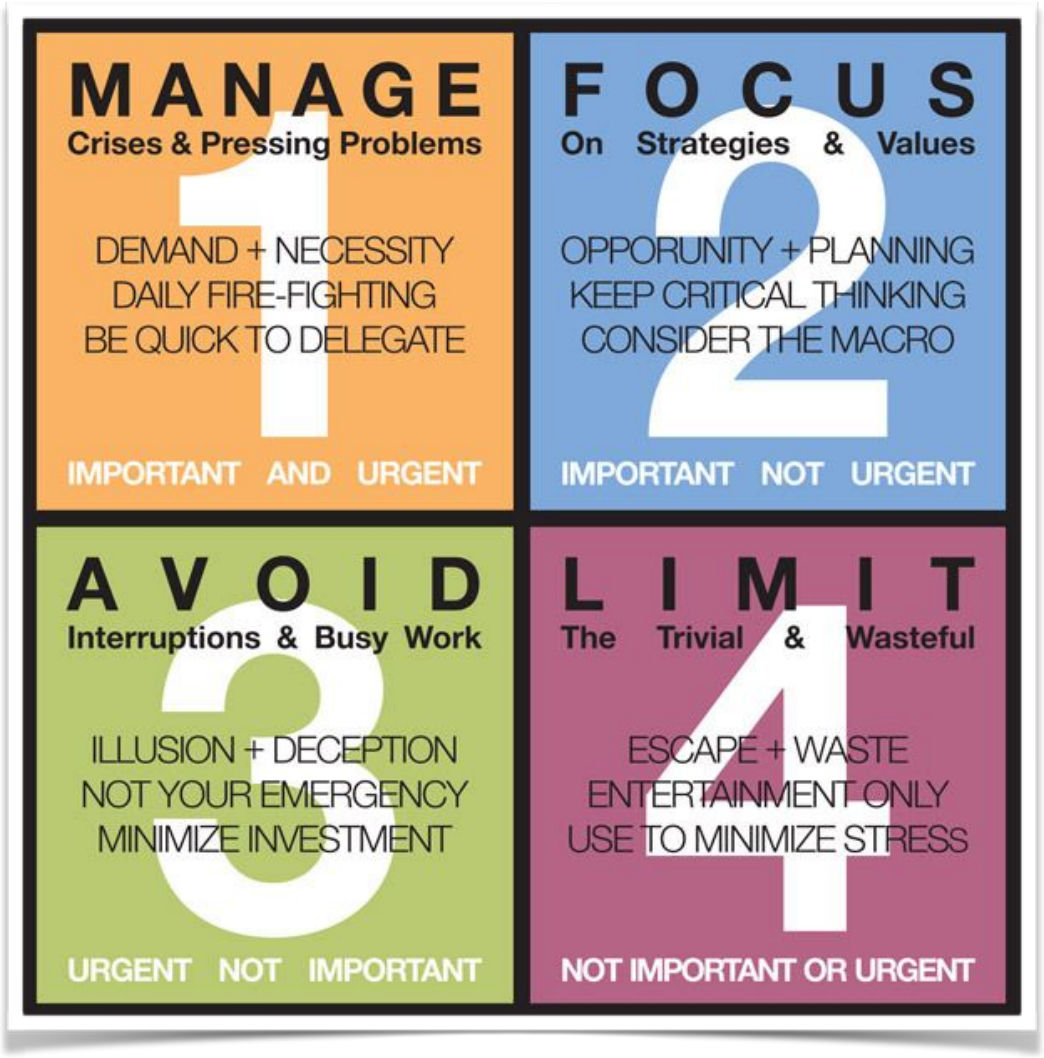


Evolution of Time Management

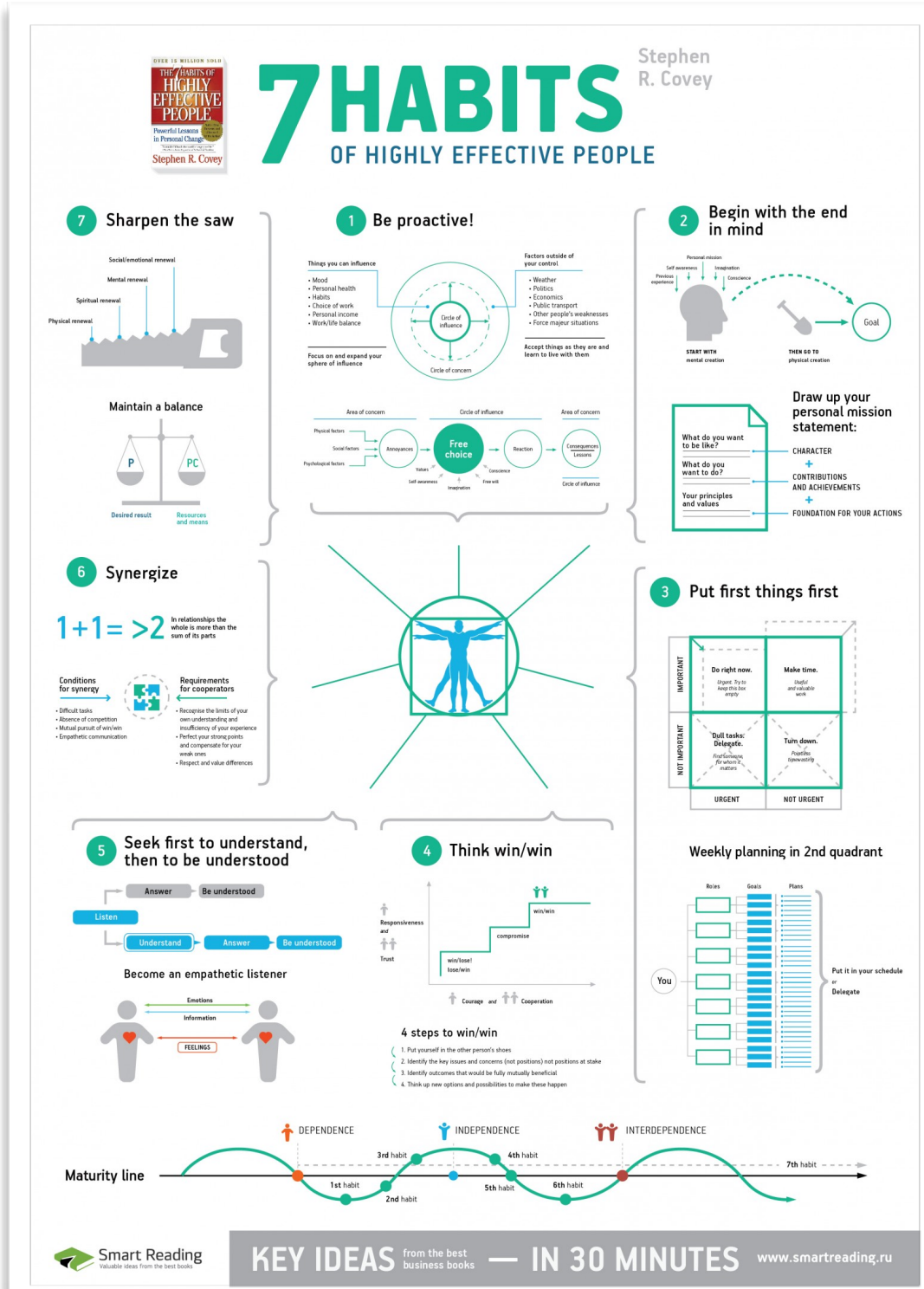
Task Driven

Schedule Driven

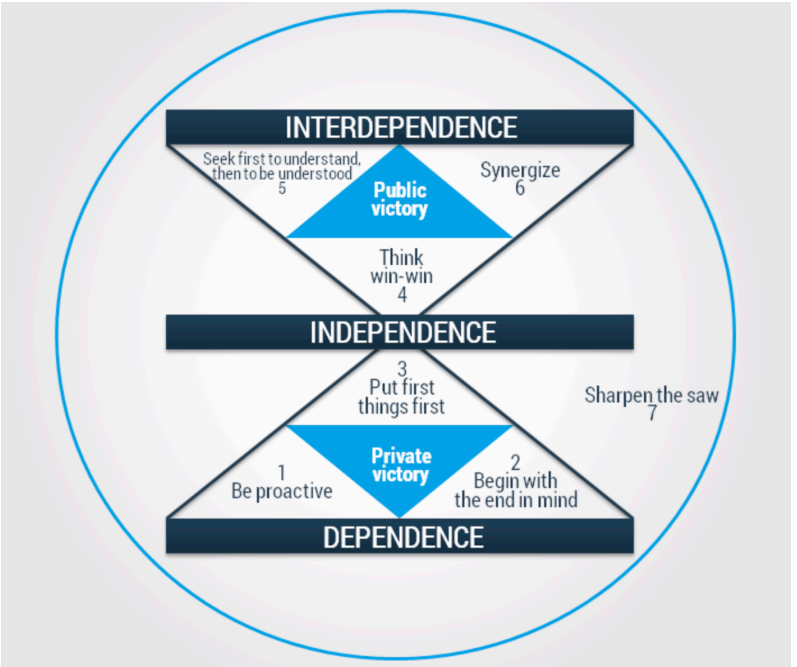
Evolution of Time Management



Value Driven



Steve Covey's 7 Habits



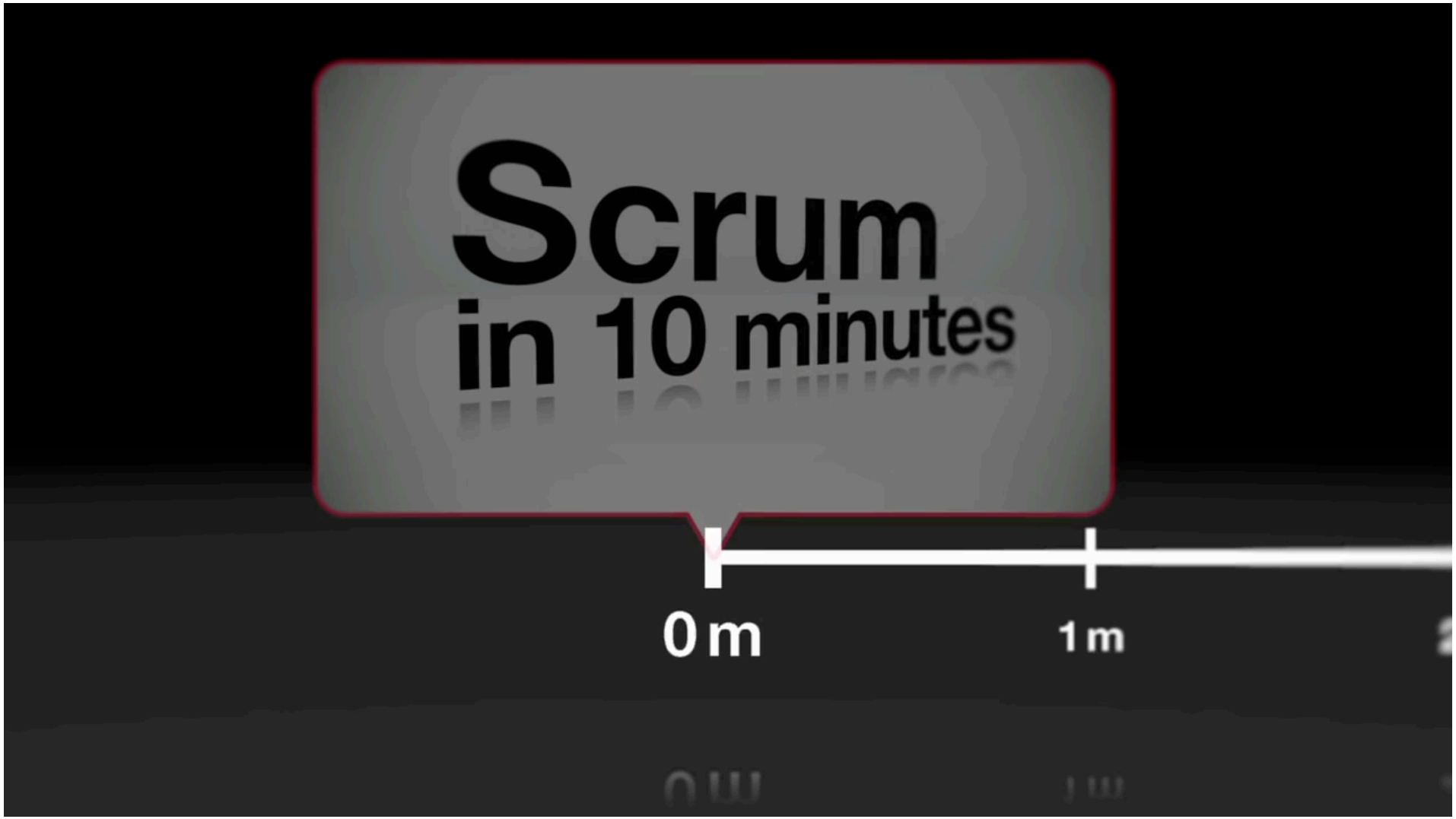
Agility

https://www.youtube.com/watch?v=cJMwBwFj5nQ

Don’t get set into one form, adapt it and build your own, and let it grow, be like water.

—Bruce Lee

SCRUM - It Works!



https://www.youtube.com/watch?v=XU0llRltyFM

SCRUM - Why it works!

http://www.youtube.com/watch?v=O7cA1q0XwhE

Kanban, Agile Scaling & DevOps

A REFRESHER ON KANBAN & HOW IT IS USED

KANBAN (LITERALLY SIGNBOARD OR BILLBOARD IN JAPANESE) (看板)

4 PRINCIPLES:
1. START WITH WHAT YOU DO NOW
2. AGREE TO PURSUE INCREMENTAL, EVOLUTIONARY CHANGE
3. RESPECT THE CURRENT PROCESS, ROLES, RESPONSIBILITIES & TITLES
4. ENCOURAGE ACTS OF LEADERSHIP AT ALL LEVELS

6 PRACTICES:
1. VISUALIZE WORK
2. LIMIT WIP
3. MANAGE FLOW
4. MAKE PROCESS EXPLICIT
5. IMPLEMENT FEEDBACK LOOOPS
6. IMPROVE COLLABORATIVELY, EVOLVE EXPERIMENTALLY

LITTLE'S LAW - "THEORY OF WAITING IN LINES"

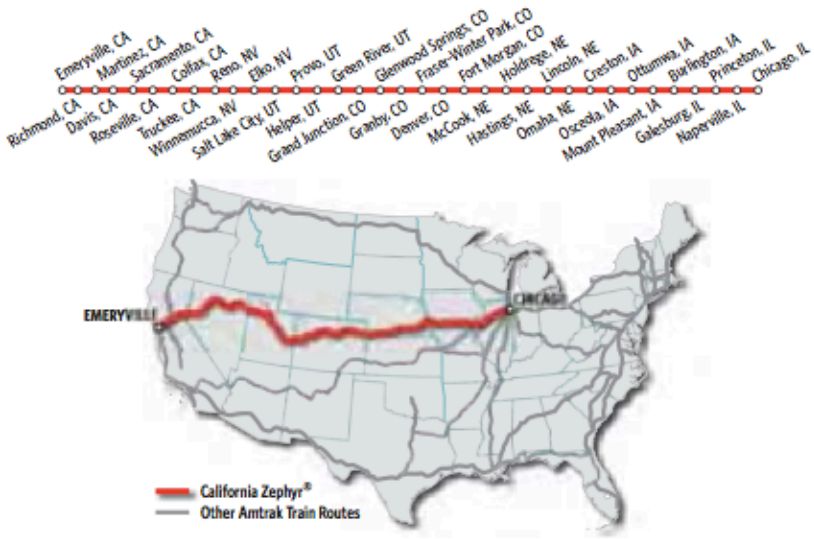
KANBAN ISN'T JUST THE BOARD, OR USING A TOOL!

https://www.youtube.com/watch?v=z7uBxMUNi-o&sns=em

Agile Practices

5	‹ Train Number ›					6
Daily	‹ Normal Days of Operation ›					Daily
R 🛏 ✕ ☕ 🧳	‹ On Board Service ›					R 🛏 ✕ ☕ 🧳
Read Down	Mile			Symbol		Read Up
🧳2 00P	0	Dp	Chicago, IL–Union Station (CT)	●♿QT	Ar	🧳2 50P
R2 34P	28		Naperville, IL (METRA/BN Line)	●♿QT		D1 43P
3 44P	104		Princeton, IL	○		D12 23P
4 38P	162		Galesburg, IL -S. Seminary St. [77]	●♿QT		D11 31A
5 25P	205		Burlington, IA	○♿		10 36A
5 59P	233		Mount Pleasant, IA	●♿		9 54A
6 53P	279		Ottumwa, IA	●♿		9 09A
8 09P	359		Osceola, IA (Des Moines)	○♿QT		7 40A
8 41P	392		Creston, IA	○♿		7 04A
🧳10 55P	500	Ar	Omaha, NE	●♿	Dp	🧳5 14A
🧳11 05P		Dp			Ar	🧳4 59A
🧳12 08A	555	Ar	Lincoln, NE	●♿	Dp	🧳3 26A
🧳12 14A		Dp			Ar	🧳3 20A
🧳1 47A	652		Hastings, NE (Grand Island)	●♿		🧳1 42A
2 34A	706		Holdrege, NE	○♿		12 54A
3 43A	783		McCook, NE (CT)	○♿		11 49P
5 05A	960		Fort Morgan, CO (Sterling) (MT)	○♿		8 25P
🧳7 15A	1038	Ar	Denver, CO	●♿QT	Dp	🧳7 10P
🧳8 05A		Dp	🚌 Colorado Springs, Pueblo, Vail, Glenwood Springs —see back		Ar	🧳6 38P
10 07A	1100		Fraser-Winter Park, CO	○♿		3 50P
10 37A	1113		Granby, CO (Rocky Mt. Nat'l Park)	○♿		3 12P
🧳1 53P	1223		Glenwood Springs, CO (Aspen)	●♿		🧳12 10P
🧳4 10P	1311		Grand Junction, CO	●♿		🧳10 23A
5 58P	1417		Green River, UT	○♿		7 59A
7 20P	1488		Helper, UT (Price)	○♿		6 37A
9 26P	1563		Provo, UT	○♿		4 35A
🧳11 05P	1608	Ar	Salt Lake City, UT (MT)	●♿	Dp	🧳3 30A
🧳11 30P		Dp	🚌 Ogden, Boise, Las Vegas —see back		Ar	🧳3 05A
3 03A	1871		Elko, NV (PT)	○♿		9 31P
5 40A	2013		Winnemucca, NV	○♿		7 08P
🧳8 36A	2202		Reno, NV	●♿QT		🧳4 06P
9 37A	2237		Truckee, CA (Lake Tahoe)	○♿		2 38P
11 48A	2301		Colfax, CA	○♿		12 21P
12 57P	2336		Roseville, CA	○♿QT		11 35A
🧳D2 13P	2353		Sacramento, CA	●♿QT		🧳11 09A
🧳D2 44P	2367		Davis, CA	●♿QT		🧳10 36A
🧳D3 26P	2411		Martinez, CA (San Joaquin Trains)	●♿QT		🧳9 54A
D3 59P	2430		Richmond, CA	○♿QT		9 22A
🧳4 10P	2438	Ar	Emeryville, CA (PT)	●♿QT	Dp	🧳9 10A
			🚌 San Francisco—see back			

CALIFORNIA ZEPHYR ROUTE MAP and SYMBOLS



A	Time Symbol for A.M.	QT	Quik-Trak self-serve ticketing kiosk
P	Time Symbol for P.M.	○	Unstaffed station
D	Stops only to discharge passengers; train may leave before time shown.	●	Staffed ticket office; may or may not be open for all train departures
R	Stops only to receive passengers.	♿	Station wheelchair accessible; no barriers between station and train
CT	Central time	♿	Station wheelchair accessible; not all station facilities accessible
MT	Mountain time		
PT	Pacific time		
🚌	Bus stop		

BOY SCOUTS OF AMERICA®

Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools

Working software over comprehensive documentation

Customer collaboration over contract negotiation

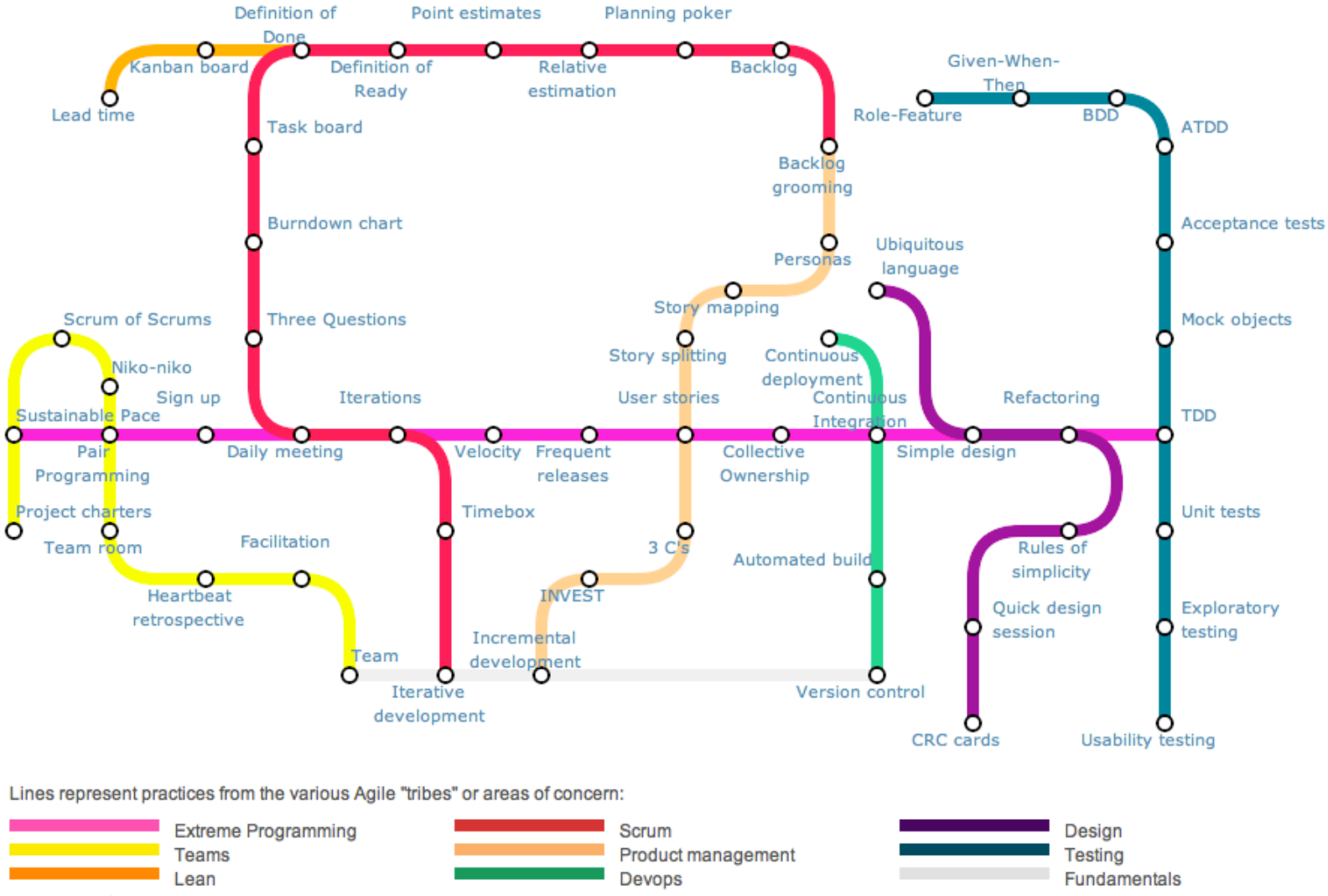
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Agile Principles

1	Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.	7	Working software is the primary measure of progress.
2	Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.	8	Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
3	Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.	9	Continuous attention to technical excellence and good design enhances agility.
4	Business people and developers must work together daily throughout the project.	10	Simplicity--the art of maximizing the amount of work not done--is essential.
5	Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.	11	The best architectures, requirements, and designs emerge from self-organizing teams.
6	The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.	12	At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

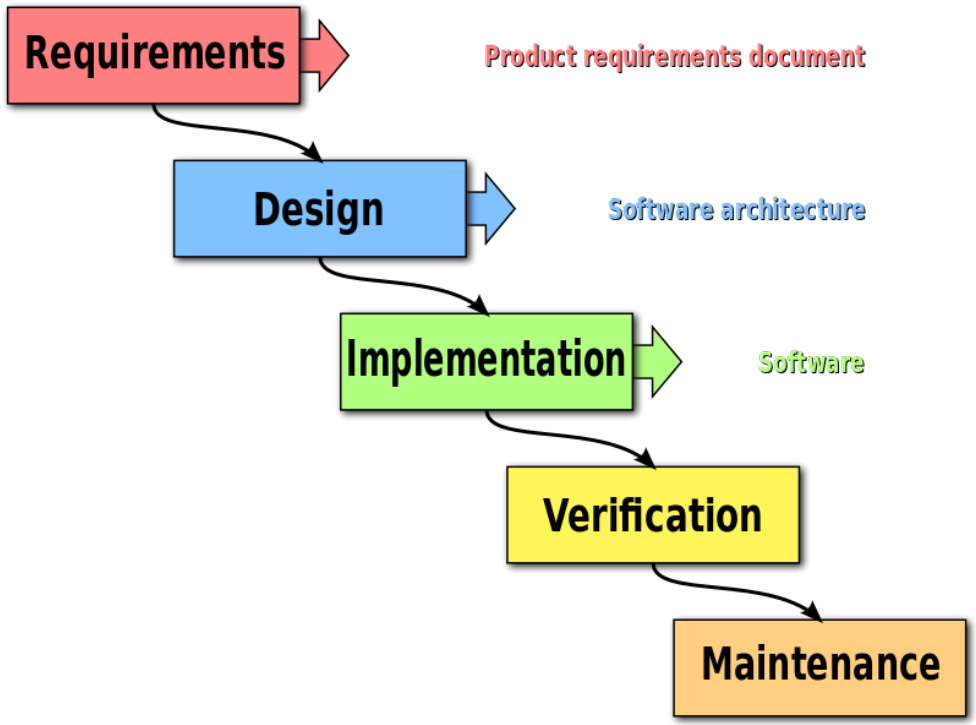
Agile Practices



https://www.agilealliance.org/agile101/subway-map-to-agile-practices/

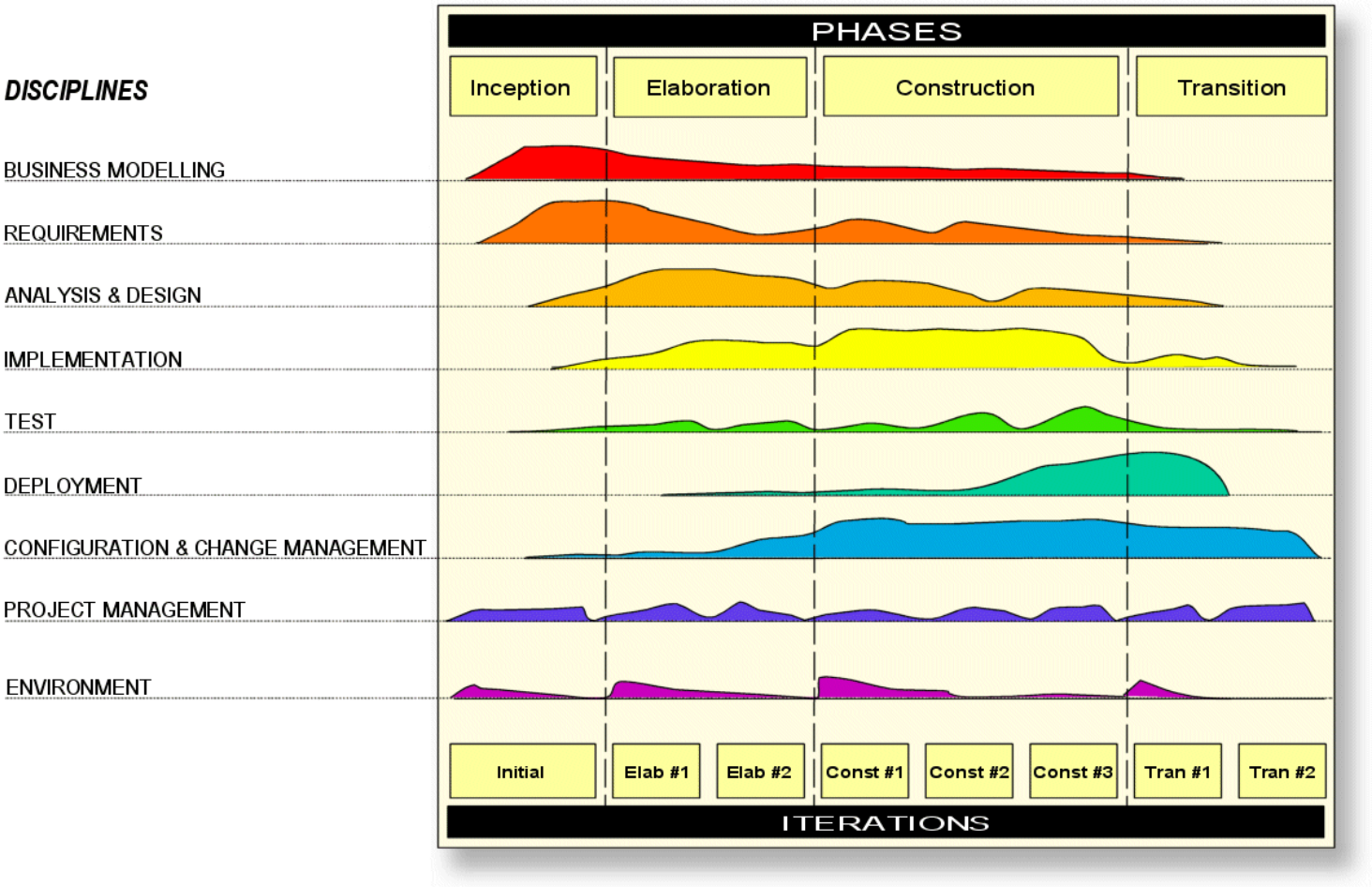
Methodologies

Waterfall



http://en.wikipedia.org/wiki/Waterfall_model

RUP



http://en.wikipedia.org/wiki/Rational_Unified_Process

Rational Unified Process

Although the Rational Unified Process (RUP) is independent of the UML, the two are often talked about together. So I think it's worth saying a few things about it here.

Although RUP is called a process, it actually is a process framework, providing a vocabulary and loose structure to talk about processes. When you use RUP, the first thing you need to do is choose a **development case**: the process you are going to use in the project. Development cases can vary widely, so don't assume that your development case will look that much like any other development case. Choosing a development case needs someone early on who is very familiar with RUP: someone who can tailor RUP for a particular project's needs. Alternatively, there is a growing body of packaged development cases to start from.

Whatever the development case, RUP is essentially an iterative process. A waterfall style isn't compatible with the philosophy of RUP, although sadly it's not uncommon to run into projects that use a waterfall-style process and dress it up in RUP's clothes.

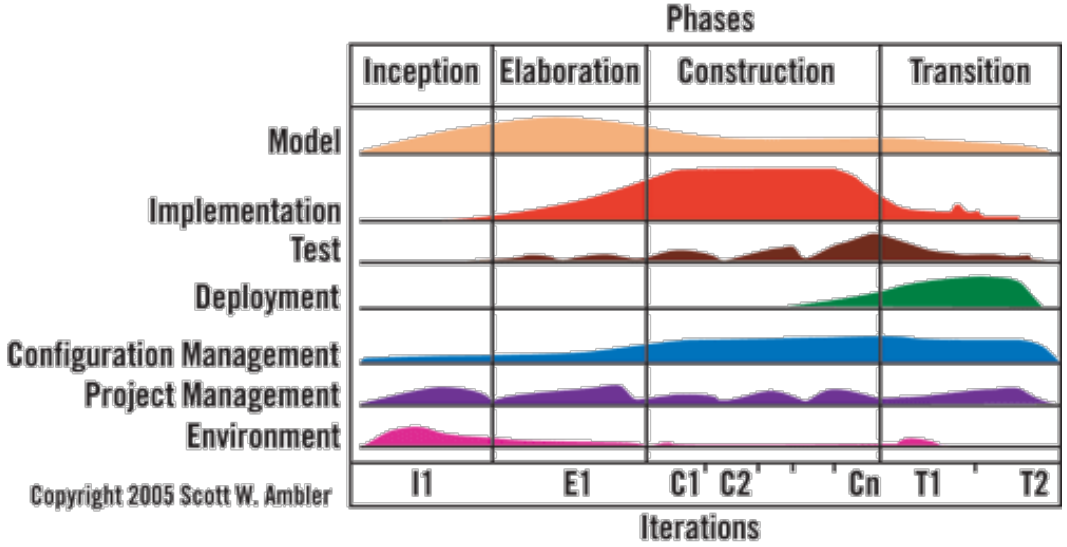
All RUP projects should follow four phases.

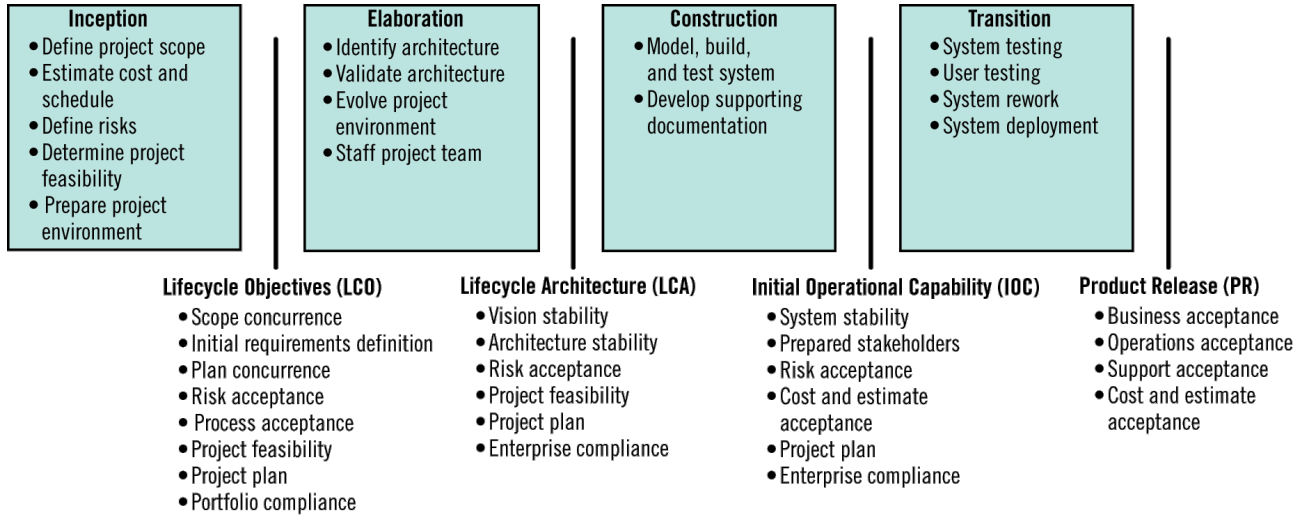
1. **Inception** makes an initial evaluation of a project. Typically in inception, you decide whether to commit enough funds to do an elaboration phase.
2. **Elaboration** identifies the primary use cases of the project and builds software in iterations in order to shake out the architecture of the system. At the end of elaboration, you should have a good sense of the requirements and a skeletal working system that acts as the seed of development. In particular, you should have found and resolved the major risks to the project.
3. **Construction** continues the building process, developing enough functionality to release.
4. **Transition** includes various late-stage activities that you don't do iteratively. These may include deployment into the data center, user training, and the like.

There's a fair amount of fuzziness between the phases, especially between elaboration and construction. For some, the shift to construction is the point at which you can move into a predictive planning mode. For others, it merely indicates the point at which you have a broad vision of requirements and an architecture that you think is going to last the rest of the project.

Sometimes, RUP is referred to as the Unified Process (UP). This is usually done by organizations that wish to use the terminology and overall style of RUP without using the licensed products of Rational Software. You can think of RUP as Rational's product offering based on the UP, or you can think of RUP and UP as the same thing. Either way, you'll find people who agree with you.

Agile UP





http://www.drdobbs.com/the-agile-edge-unified-and-agile/184415460

The Values of Extreme Programming

Extreme Programming (XP) is based on values. The rules we just examined are the natural extension and consequence of maximizing our values. XP isn't really a set of rules but rather a way to work in harmony with your personal and corporate values. Start with XP's values listed here then add your own by reflecting them in the changes you make to the rules.

Simplicity: We will do what is needed and asked for, but no more. This will maximize the value created for the investment made to date. We will take small simple steps to our goal and mitigate failures as they happen. We will create something we are proud of and maintain it long term for reasonable costs.

Communication: Everyone is part of the team and we communicate face to face daily. We will work together on everything from requirements to code. We will create the best solution to our problem that we can together.

Feedback: We will take every iteration commitment seriously by delivering working software. We demonstrate our software early and often then listen carefully and make any changes needed. We will talk about the project and adapt our process to it, not the other way around.

Respect: Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it's simply enthusiasm. Developers respect the expertise of the customers and vice versa. Management respects our right to accept responsibility and receive authority over our own work.

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 they happen.

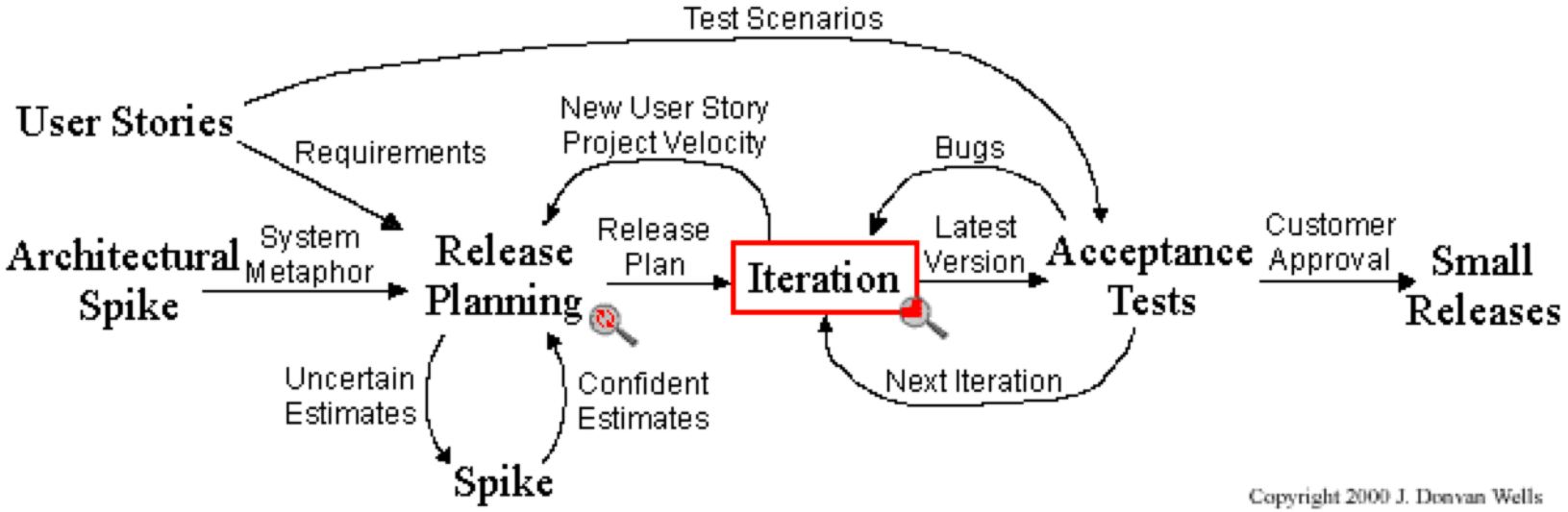
What lessons have we learned about implementing XP so far.

ExtremeProgramming.org home | XP Rules | XP Map | Lessons Learned | About the Author

http://www.extremeprogramming.org/rules.html

http://www.extremeprogramming.org/map/project.html

Extreme Programming Project



Designing

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

Testing

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

Let's review the values of Extreme Programming (XP) next.

ExtremeProgramming.org home | XP Map | XP Values | Test framework | About the Author

XP

When XP teams use **test-first programming**, they build unit tests first that express the behavior of the code that's about to be written, and then write code to make those tests pass.

Teams create a **10-minute build**, or an automated build that runs in under 10 minutes.

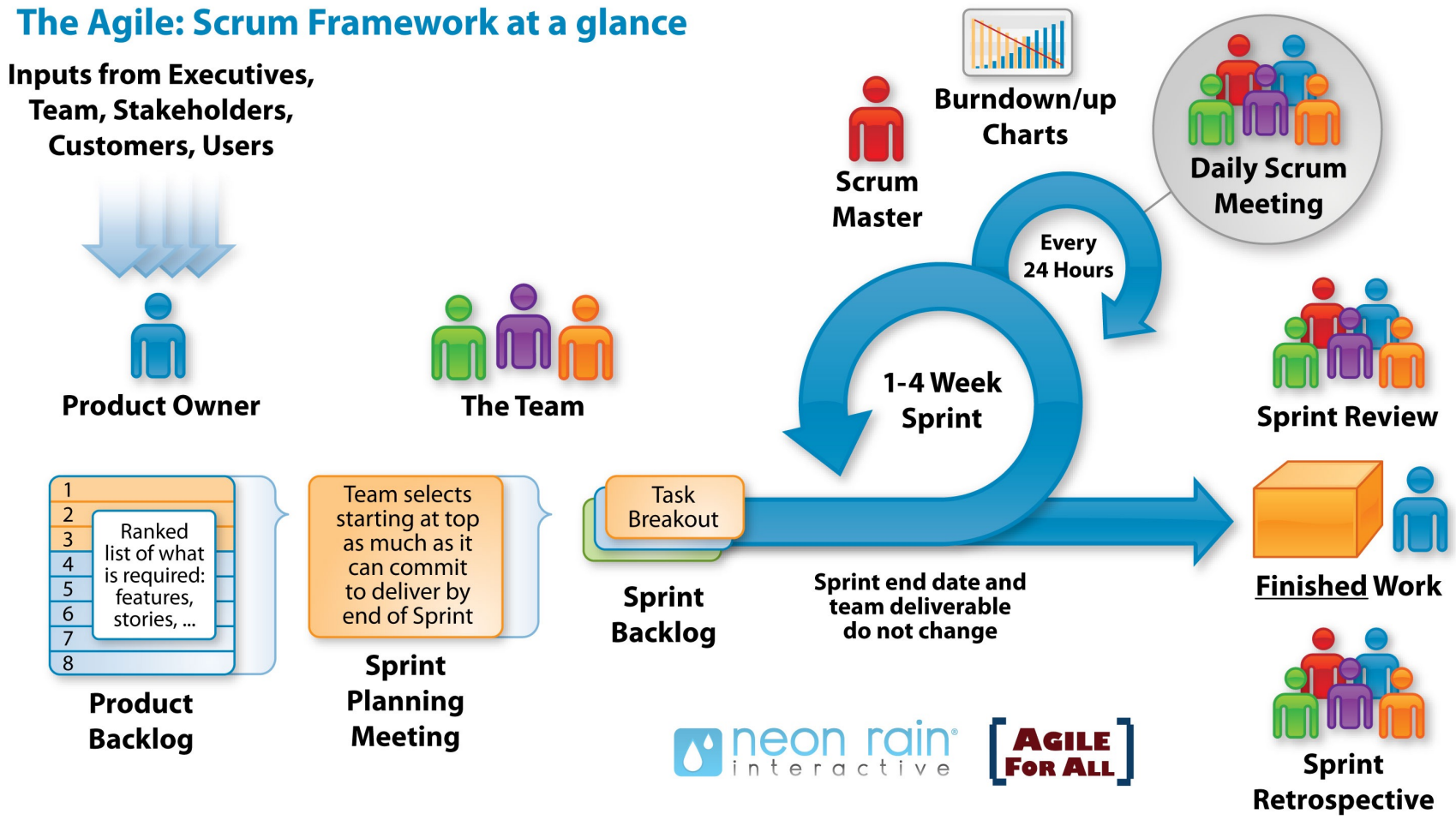
Individual developers use **continuous integration** to constantly integrate changes from their teammates so everyone's sandbox is up to date.

XP teams use **iteration** with their weekly cycle and quarterly cycle, and use stories in the same way that Scrum teams do.

When XP teams **sit together**, they periodically absorb project information through osmotic communication.

XP teams work in an **informative workspace** that using wall charts to automatically communicate information to people.

Scrum



https://www.scrum.org/

SCRUM

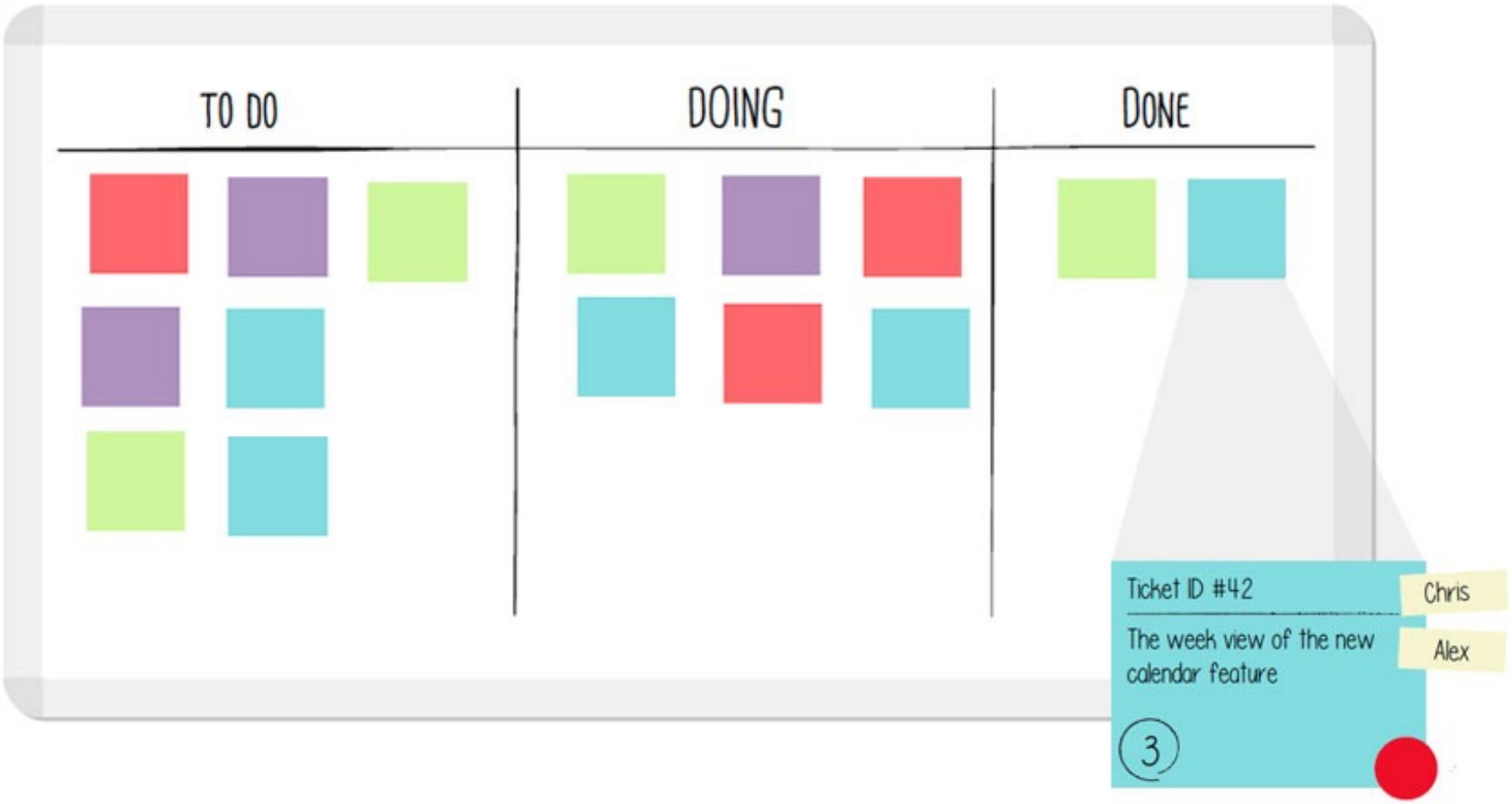
A **user story** helps the team understand their users by clearly explaining a specific need, and how the software will meet that need.

Each user story has **conditions of satisfaction** to help the team know when the story is done.

Teams use **story points and velocity** to estimate how many stories they can include in each sprint.

Posting a **burndown chart** clearly where everyone can see it can help everyone on the team understand what's completed, what's left to build, and whether or not they're on track.

Kanban



http://leankit.com/kanban/what-is-kanban/

LEAN

Lean is a mindset, not a methodology, which is why it **doesn't have practices** that help you run projects.

There are values shared between lean thinking and the larger world of agile: **decide as late as possible** (or making decisions at the last responsible moment) and **amplify learning** (using feedback and iterations).

The Lean value empower the team is very similar to the Scrum value of focus, and the XP value of **energized work**.

A **value stream** map is a visual tool to help Lean teams see the reality of how their project works by showing the entire lifespan of an Minimal Marketable Feature (**MMF)**, including the working and waiting time at every stage.

KANBAN

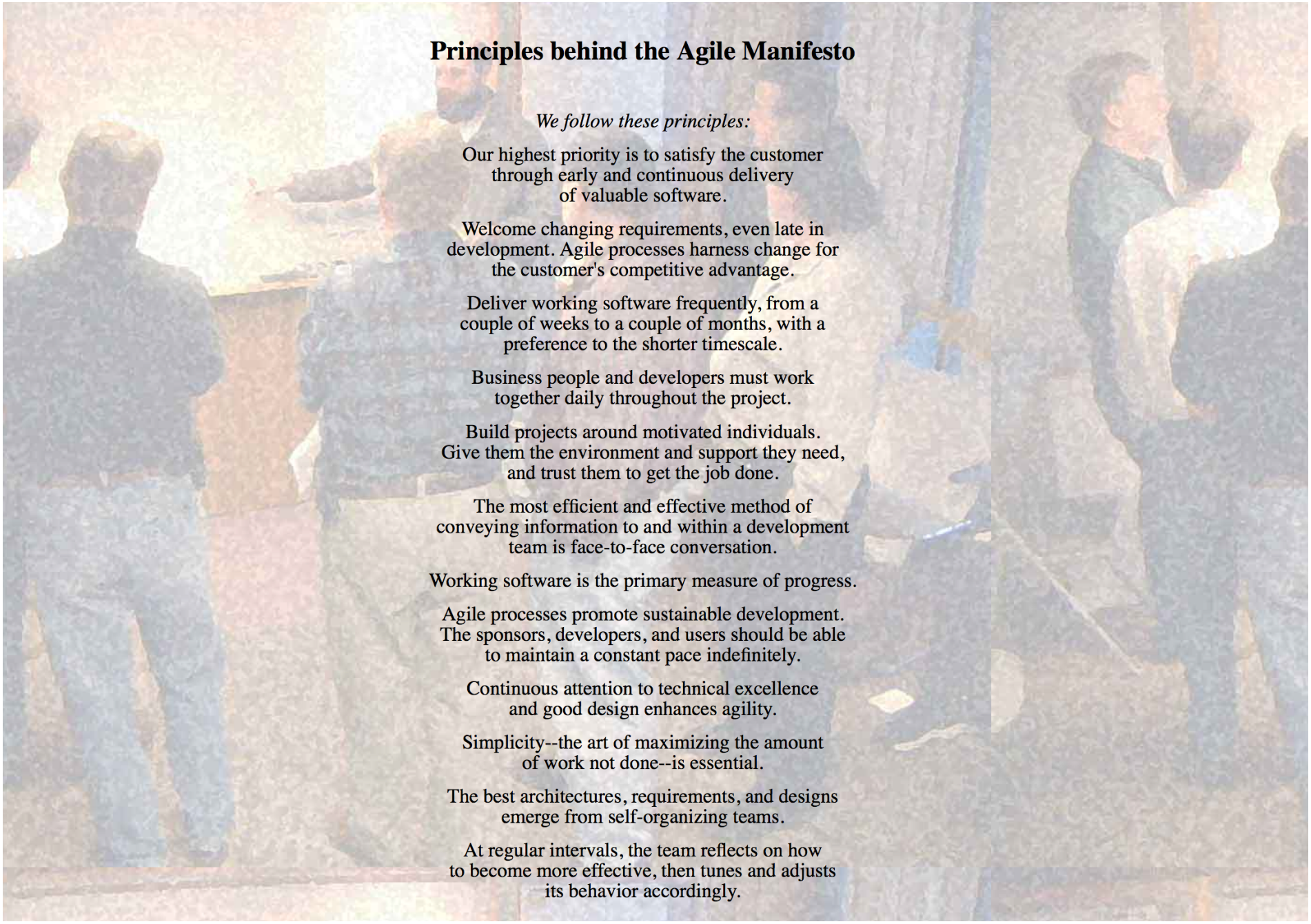
Kanban is a method for **process improvement**, or a way to help teams improve the way they build software and work together as a team, that's based on the **Lean mindset**.

Kanban teams **start with what you do now** by seeing the whole as it exists today, and **pursue incremental, evolutionary change** to improve the system gradually over time.

The Kanban practice improve collaboratively, evolve experimentally means **taking measurements**, making **gradual improvements**, and then confirming that they worked using those measurements.

Kanban teams gradually improve the system by adding **WIP limits**, **managing flow**, and making **process policies explicit**, improved behavior often emerges in the rest of the company.

http://www.agilemanifesto.org/



Principles behind the Agile Manifesto

We follow these principles:

Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

Business people and developers must work together daily throughout the project.

Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Working software is the primary measure of progress.

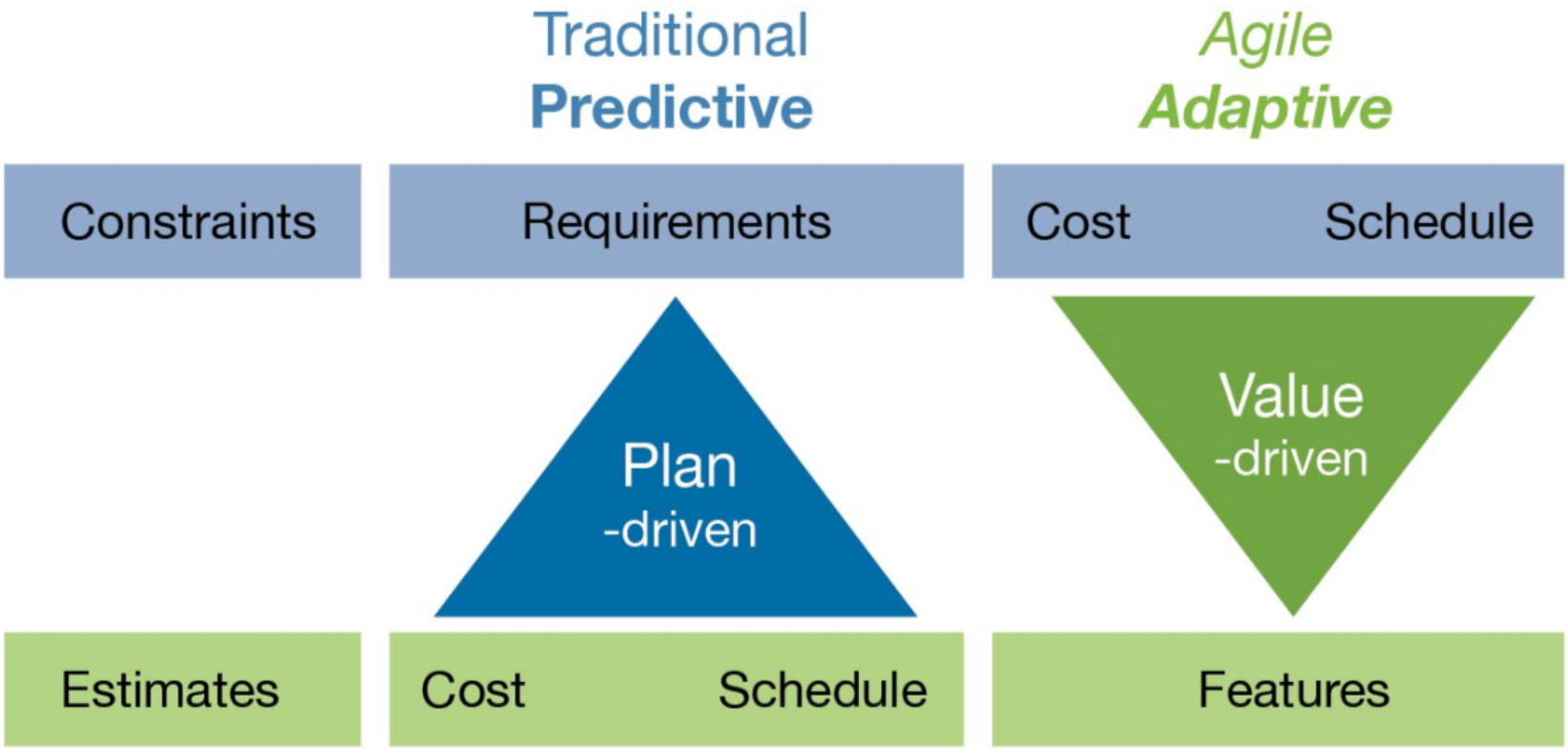
Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

Continuous attention to technical excellence and good design enhances agility.

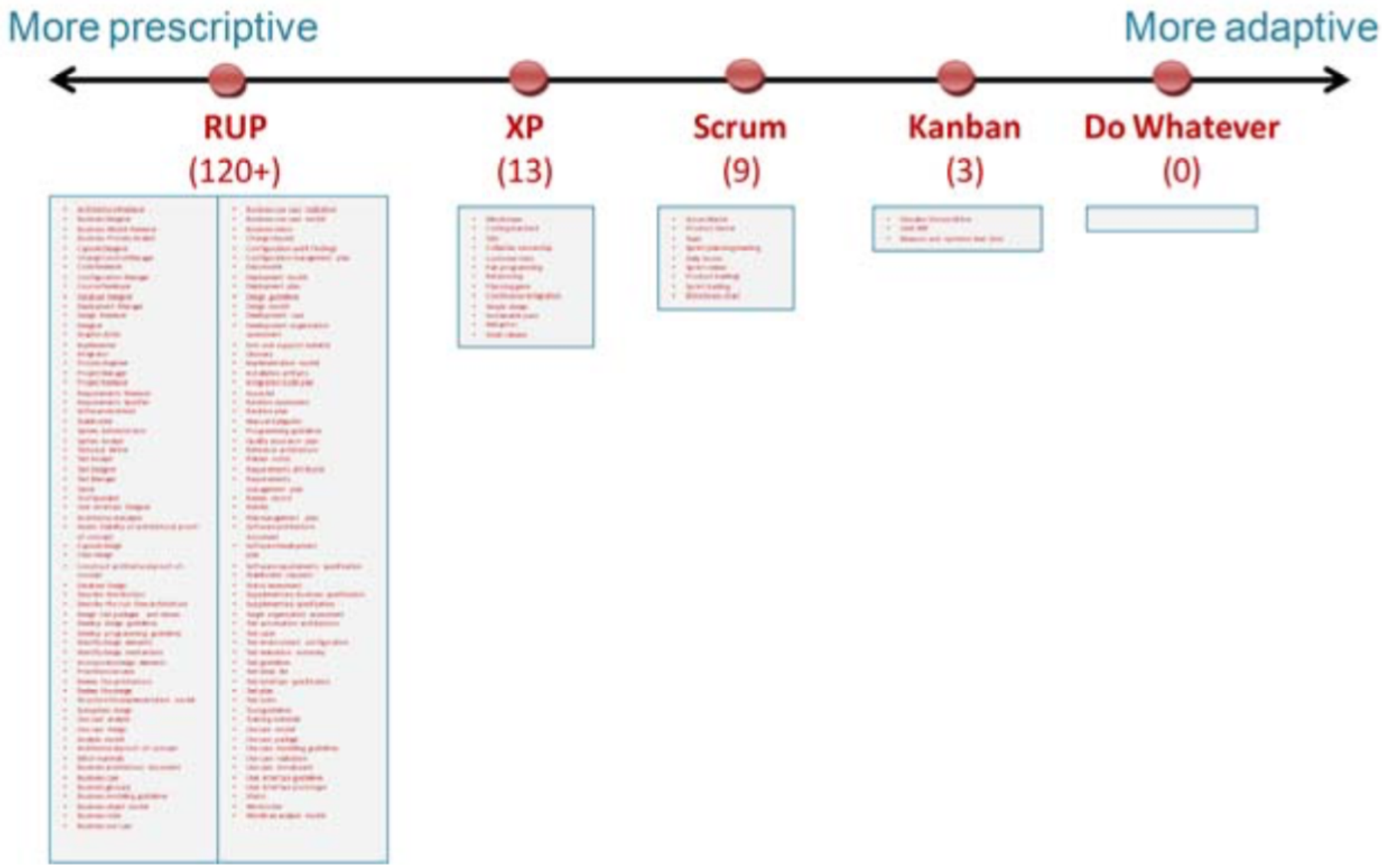
Simplicity--the art of maximizing the amount of work not done--is essential.

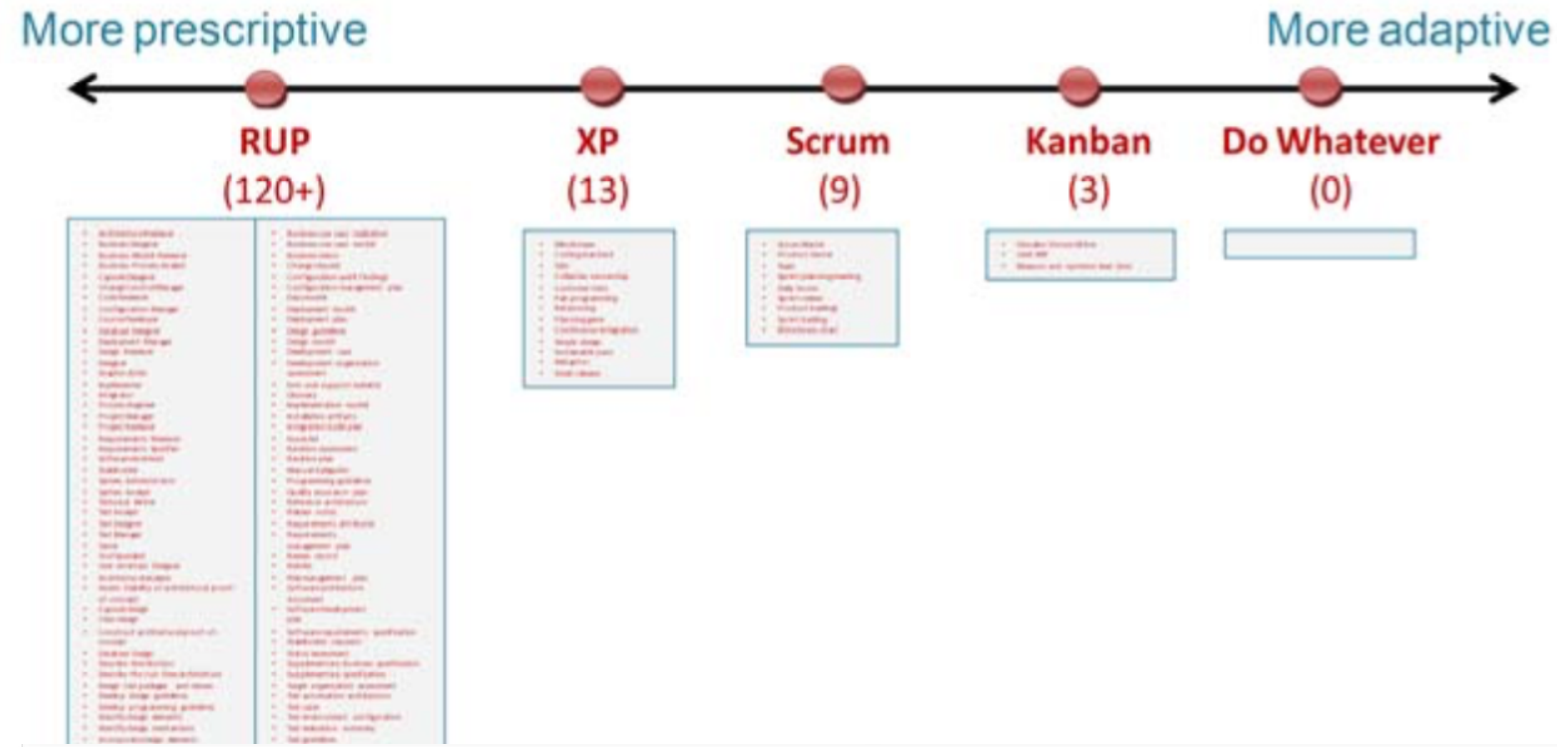
The best architectures, requirements, and designs emerge from self-organizing teams.

At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

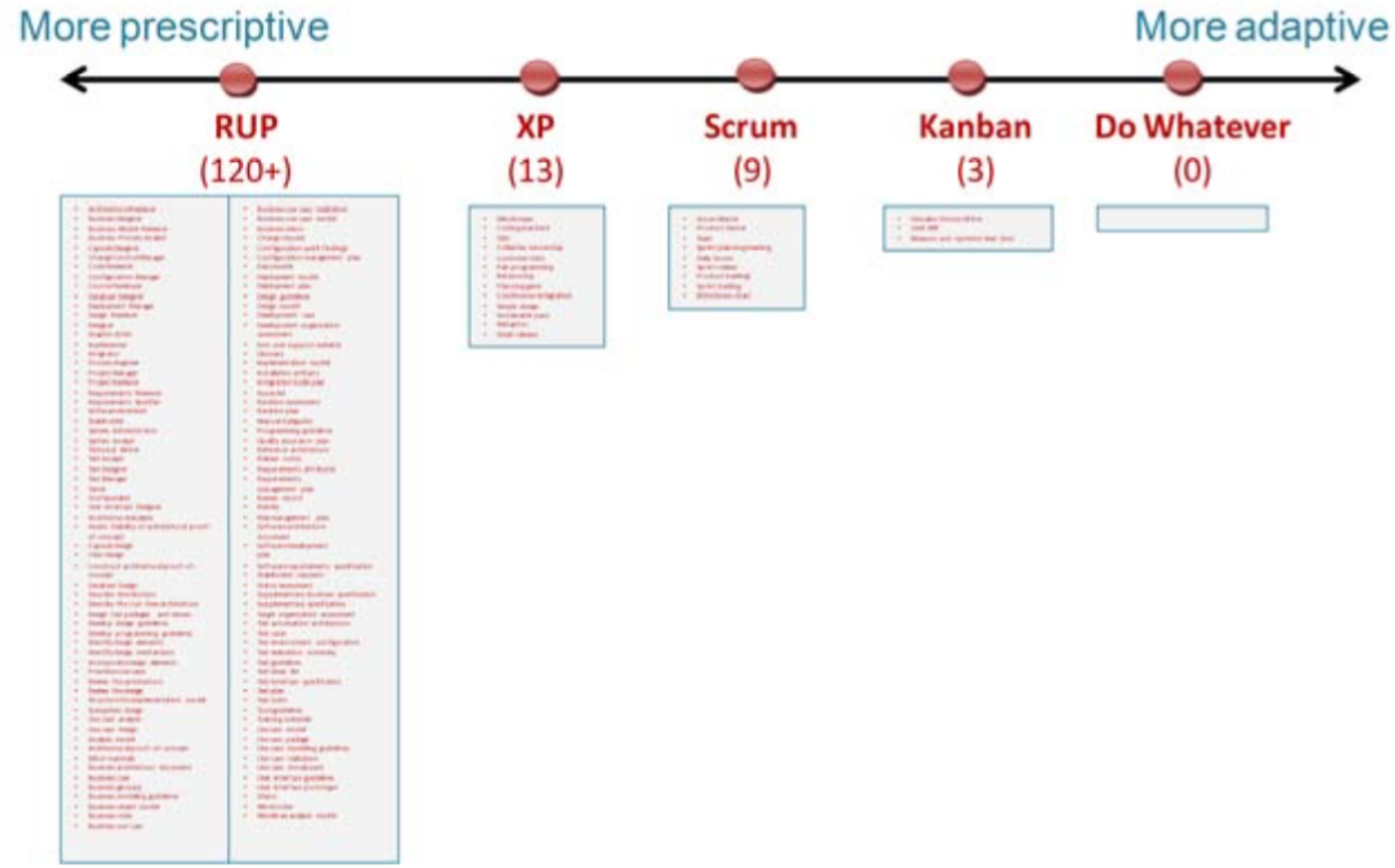


In the traditional predictive model, the project manager attempts to constrain scope in order to provide reliable estimates for time and cost. In practice we experience a "triple constraint" or "iron triangle" situation and quality may become an undesired variable. In Agile we accept time and cost as real constraints. The product manager then works with the delivery team to maximize value by delivering iteratively and incrementally. The plan is regularly adapted to match reality. (How could we ever believe the other way around could work?)

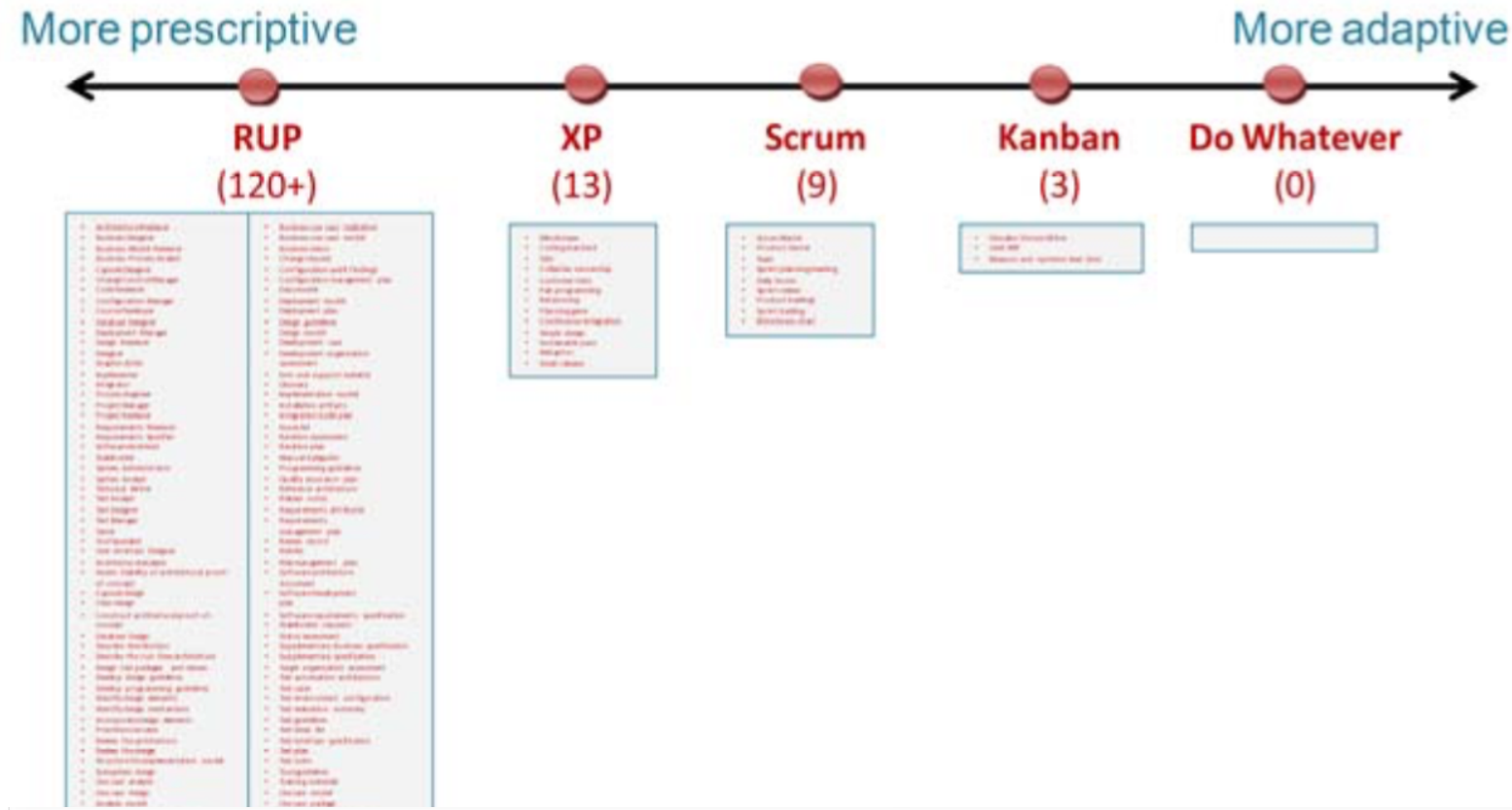

More prescriptive
More adaptive
RUP
(120+)
XP
(13)
Scrum
(9)
Kanban
(3)
Do Whatever
(0)



RUP is pretty prescriptive – it has over 30 roles, over 20 activities, and over 70 artifacts; an overwhelming amount of stuff to learn. You aren't really supposed to use all of that though; you are supposed to select a suitable subset for your project. Unfortunately this seems to be hard in practice. "Hmmmm... will we need *Configuration audit findings* artifacts? Will we need a *Change control manager* role? Not sure, so we better keep them just in case." This may be one of the reasons why RUP implementations often end up quite heavy-weight compared to Agile methods such as Scrum and XP.

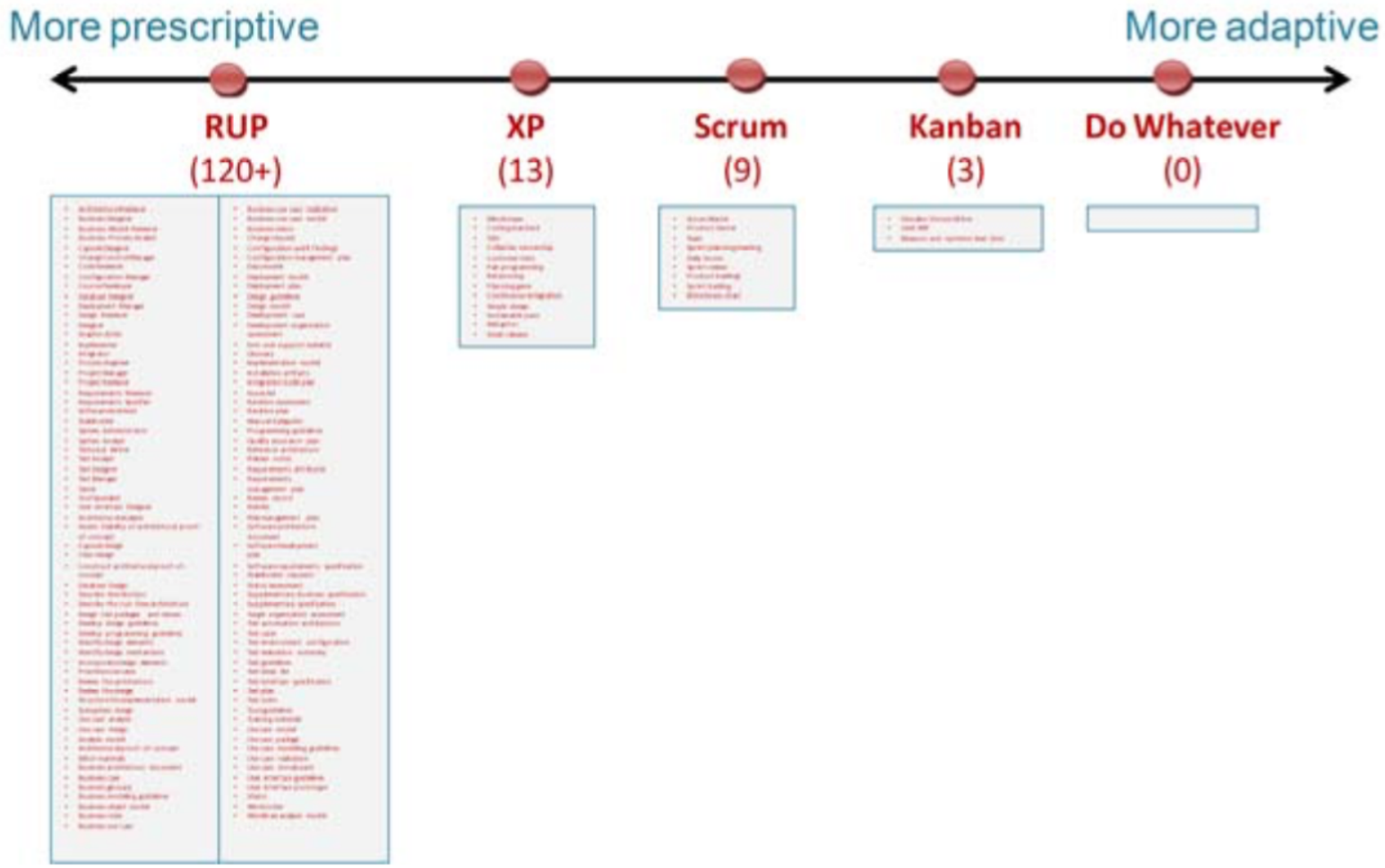


XP (eXtreme Programming) is pretty prescriptive compared to Scrum. It includes most of Scrum + a bunch of fairly specific engineering practices such as test-driven development and pair programming.



Scrum is less prescriptive than XP, since it doesn't prescribe any specific engineering practices. Scrum is more prescriptive than Kanban though, since it prescribes things such as iterations and cross-functional teams.

One of the main differences between Scrum and RUP is that in RUP you get too much, and you are supposed to remove the stuff you don't need. In Scrum you get too little, and you are supposed to add the stuff that is missing.



Kanban leaves almost everything open. The only constraints are Visualize Your Workflow and Limit Your WIP. Just inches from Do Whatever, but still surprisingly powerful.

Don’t limit yourself to one tool!

Mix and match the tools as you need! I can hardly imagine a successful Scrum team that doesn’t include most elements of XP for example. Many Kanban teams use daily standup meetings (a Scrum practice). Some Scrum teams write some of their backlog items as Use Cases (a RUP practice) or limit their queue sizes (a Kanban practice). Whatever works for you.

Miyamoto Musashi a 17th century Samurai who was famous for his twin-sword fighting technique, said it nicely:

> Do not develop an attachment to any one weapon or any one school of fighting.
>
> - Miyamoto Musashi

http://www.ambysoft.com/essays/agileManifesto.html

1. **Individuals and interactions over processes and tools**. Teams of people build software systems, and to do that they need to work together effectively – including but not limited to programmers, testers, project managers, modelers, and your customers. Who do you think would develop a better system: five software developers and with their own tools working together in a single room or five low-skilled "hamburger flippers" with a well-defined process, the most sophisticated tools available, and the best offices money could buy? If the project was reasonably complex my money would be on the software developers, wouldn't yours? The point is that the most important factors that you need to consider are the people and how they work together because if you don't get that right the best tools and processes won't be of any use. Tools and processes are important, don't get me wrong, it's just that they're not as important as working together effectively. Remember the old adage, a fool with a tool is still a fool. As Fred Brooks points out in The Mythical Man Month, this can be difficult for management to accept because they often want to believe that people and time, or men and months, are interchangeable.

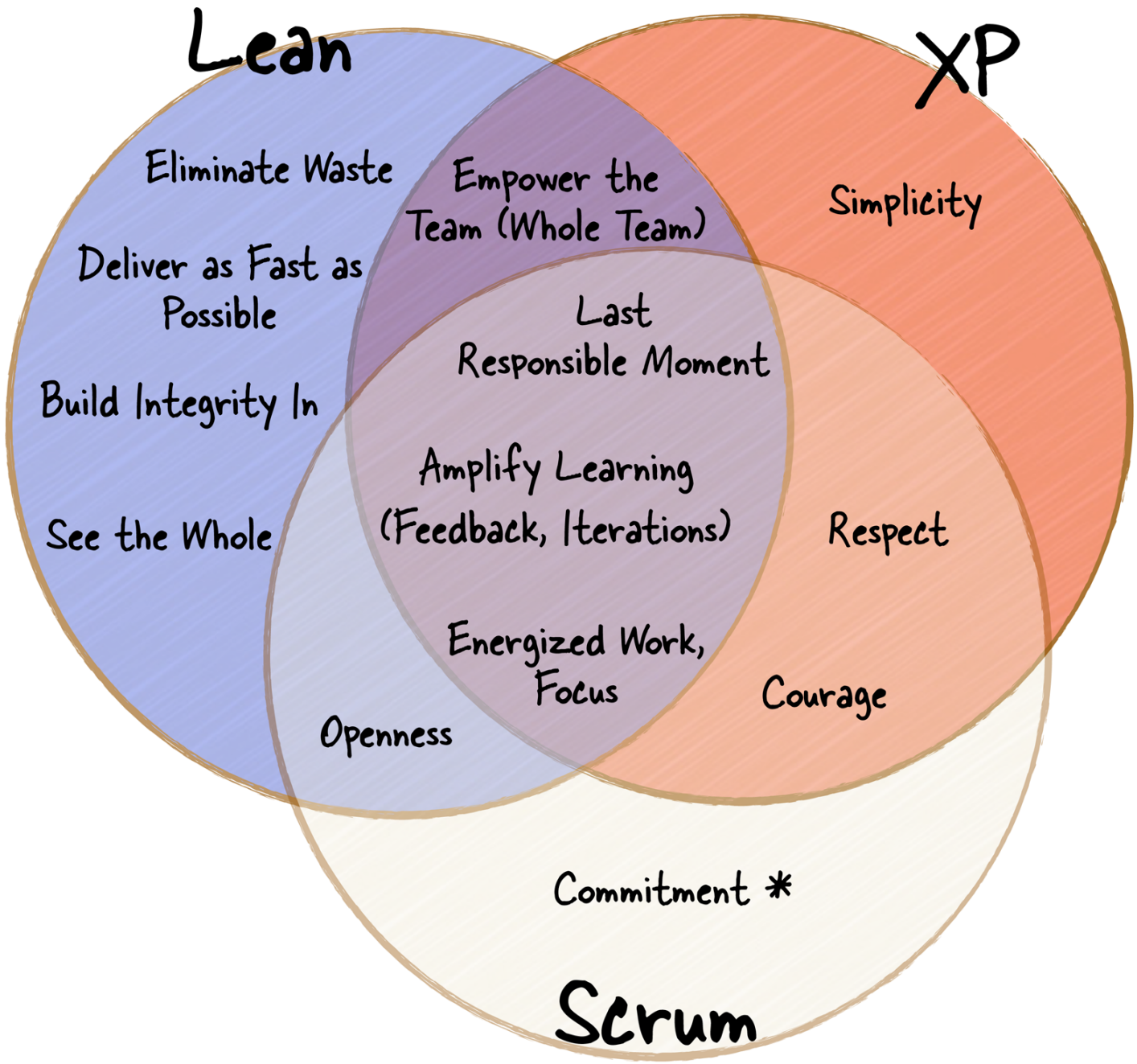
2. **Working software over comprehensive documentation**. When you ask a user whether they would want a fifty page document describing what you intend to build or the actual software itself, what do you think they'll pick? My guess is that 99 times out of 100 they'll choose working software. If that is the case, doesn't it make more sense to work in such a manner that you produce software quickly and often, giving your users what they prefer? Furthermore, I suspect that users will have a significantly easier time understanding any software that you produce than complex technical diagrams describing its internal workings or describing an abstraction of its usage, don't you? Documentation has its place, written properly it is a valuable guide for people's understanding of how and why a system is built and how to work with the system. However, never forget that the primary goal of software development is to create software, not documents – otherwise it would be called documentation development wouldn't it?

3. **Customer collaboration over contract negotiation**. Only your customer can tell you what they want. Yes, they likely do not have the skills to exactly specify the system. Yes, they likely won't get it right the first. Yes, they'll likely change their minds. Working together with your customers is hard, but that's the reality of the job. Having a contract with your customers is important, having an understanding of everyone's rights and responsibilities may form the foundation of that contract, but a contract isn't a substitute for communication. Successful developers work closely with their customers, they invest the effort to discover what their customers need, and they educate their customers along the way.

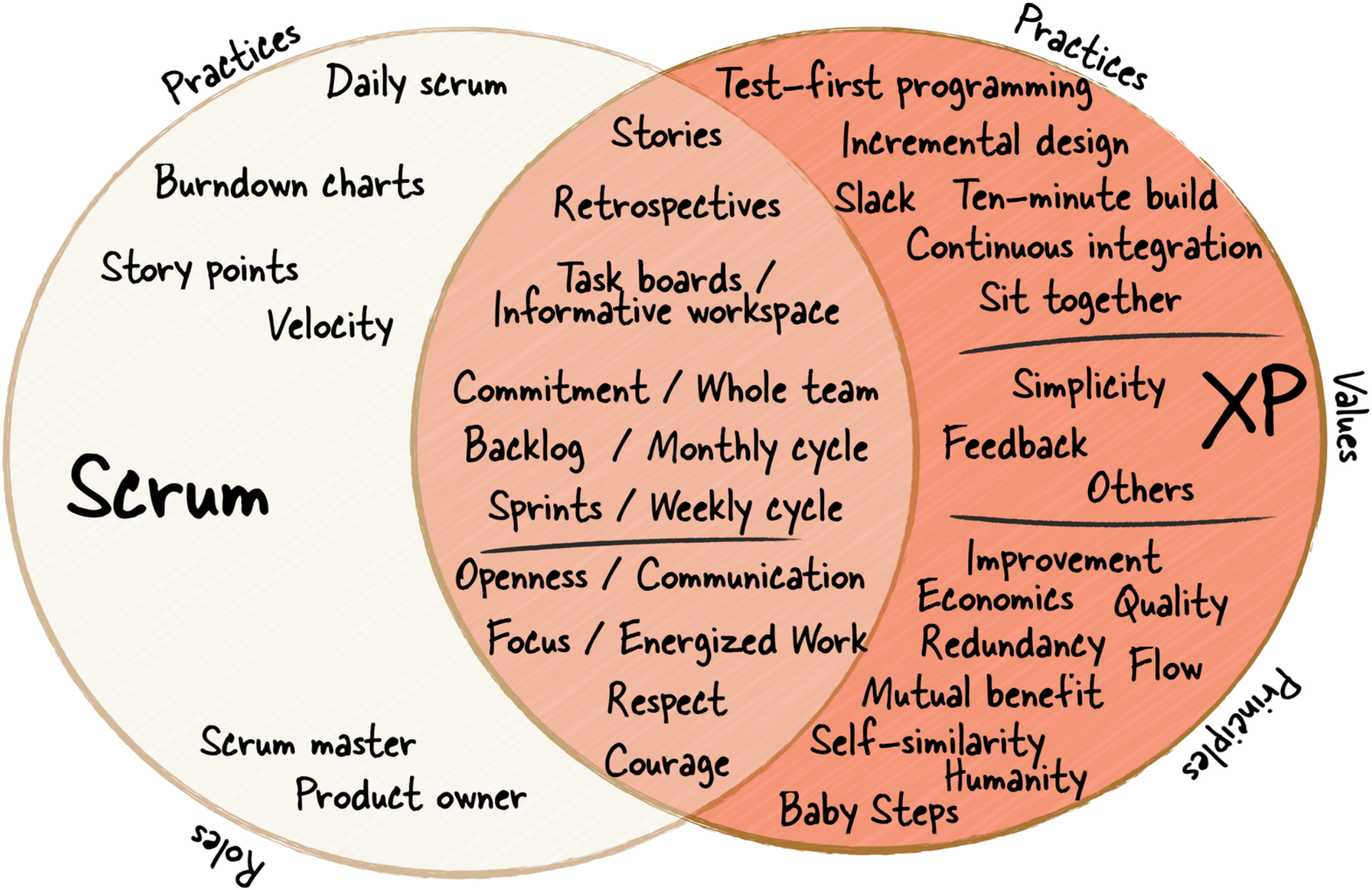
4. **Responding to change over following a plan**. People change their priorities for a variety of reasons. As work progresses on your system your project stakeholder's understanding of the problem domain and of what you are building changes. The business environment changes. Technology changes over time, although not always for the better. Change is a reality of software development, a reality that your software process must reflect. There is nothing wrong with having a project plan, in fact I would be worried about any project that didn't have one. However, a project plan must be malleable, there must be room to change it as your situation changes otherwise your plan quickly becomes irrelevant.

Comparison

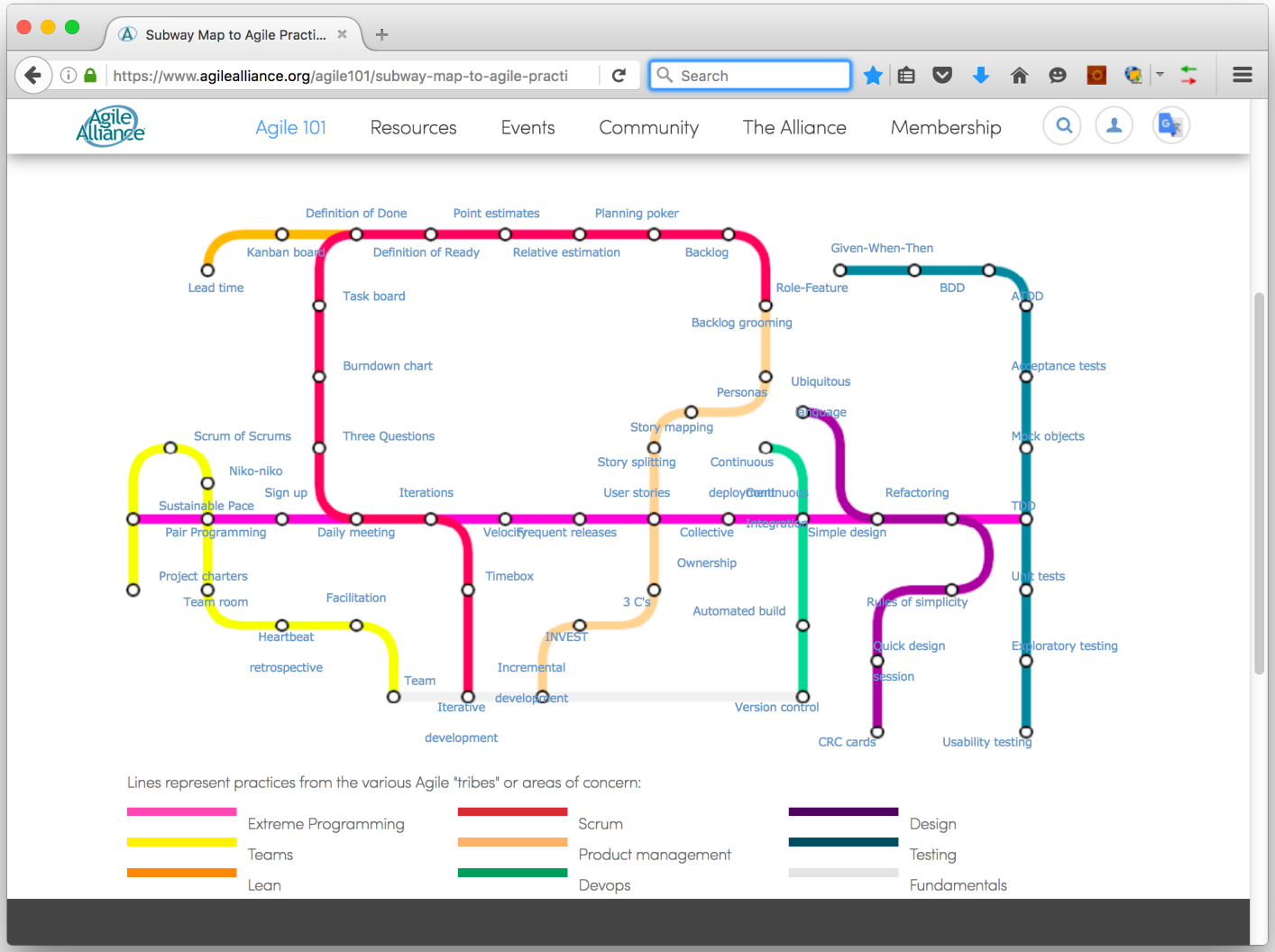
	Pre-Req	Strategy	Goal	Plan	Track	Key Metrics
Waterfall	Predictability Low Risk	Fixed Scope Fixed Time Variable Cost	Try to Stay on Budget	Planned Hours Milestones Schedule	Actual Hours Blockers/Risks	% Complete Gantt Charts
Scrum	Dedicated Team Independent Stories	Fixed Time Fixed Cost Variable Scope	Try to Predict Delivery Date & Team Capacity	Sprint Scope Define “Done”	Remaining Hours	Team Velocity Burn Down Chart
Kanban	N/A	Limit Work in Progress	Find Process Bottleneck	N/A	Status Queue	Avg. Cycle Time Cumulative Flow



* Commitment is an important part of XP and Lean, but Scrum explicitly calls it out as a value.


Practices
Daily scrum
Burndown charts
Story points
Velocity
Scrum
Scrum master
Product owner
Roles
Stories
Retrospectives
Task boards / Informative workspace
Commitment / Whole team
Backlog / Monthly cycle
Sprints / Weekly cycle
Openness / Communication
Focus / Energized Work
Respect
Courage
Practices
Test-first programming
Incremental design
Slack
Ten-minute build
Continuous integration
Sit together
Simplicity
XP
Feedback
Others
Improvement
Economics
Quality
Redundancy
Flow
Mutual benefit
Self-similarity
Humanity
Baby Steps
Principles
Values

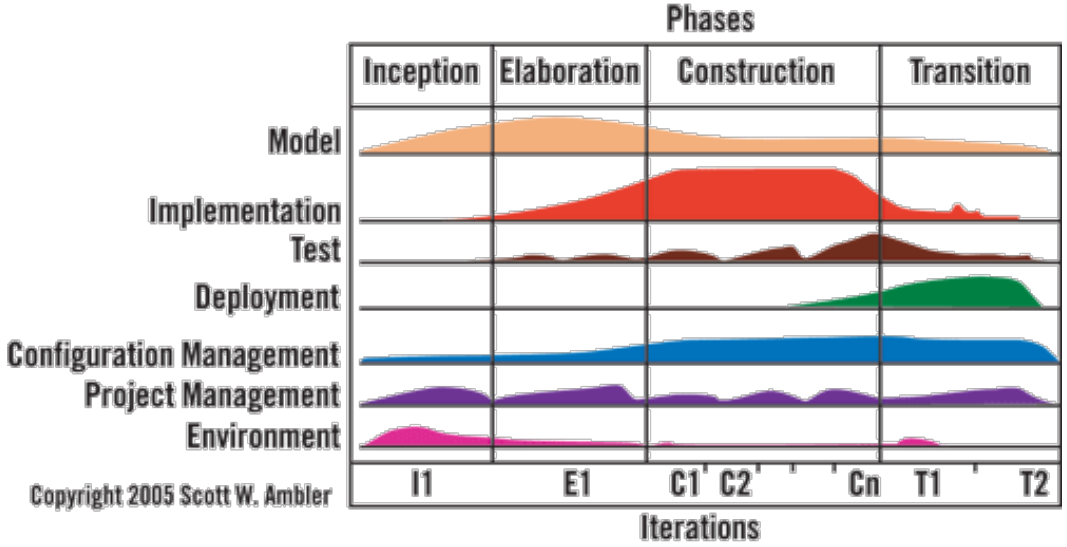
Agile Practices

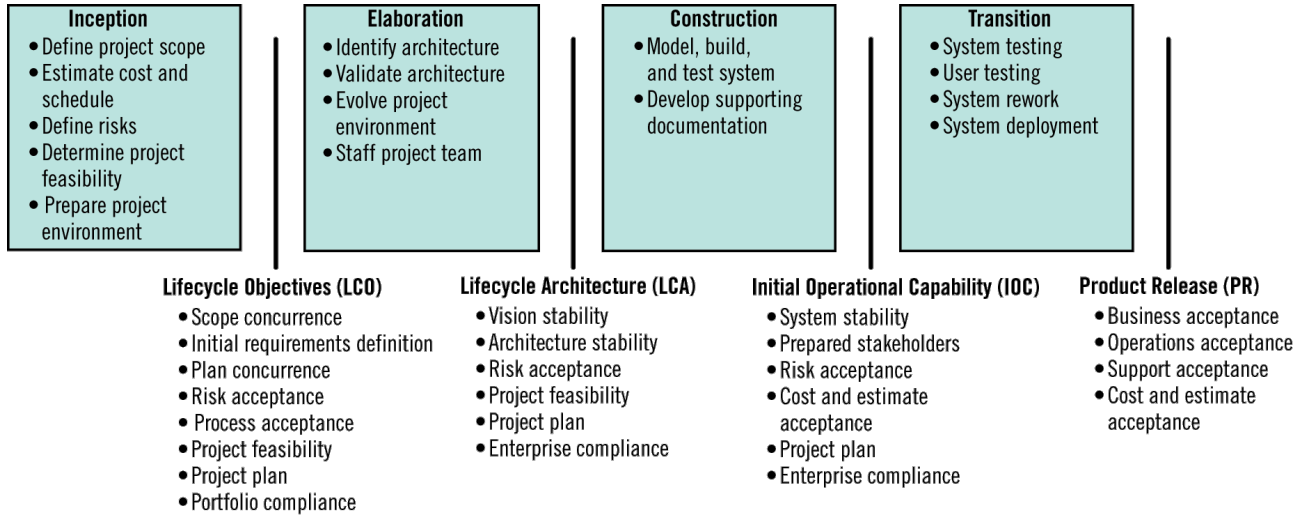


https://www.agilealliance.org/agile101/subway-map-to-agile-practices/

Agile UP

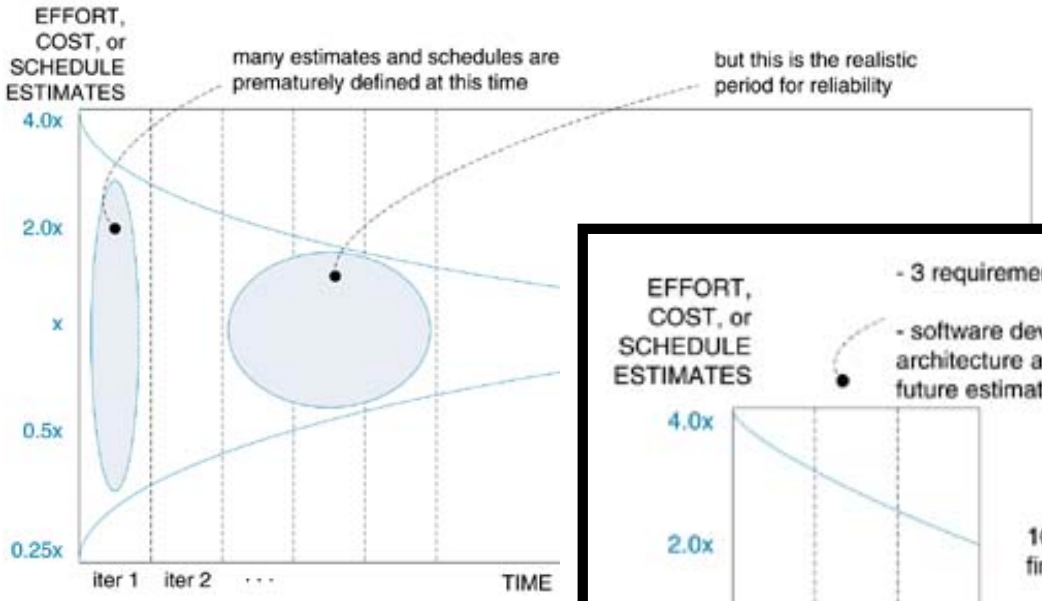
Agile UP

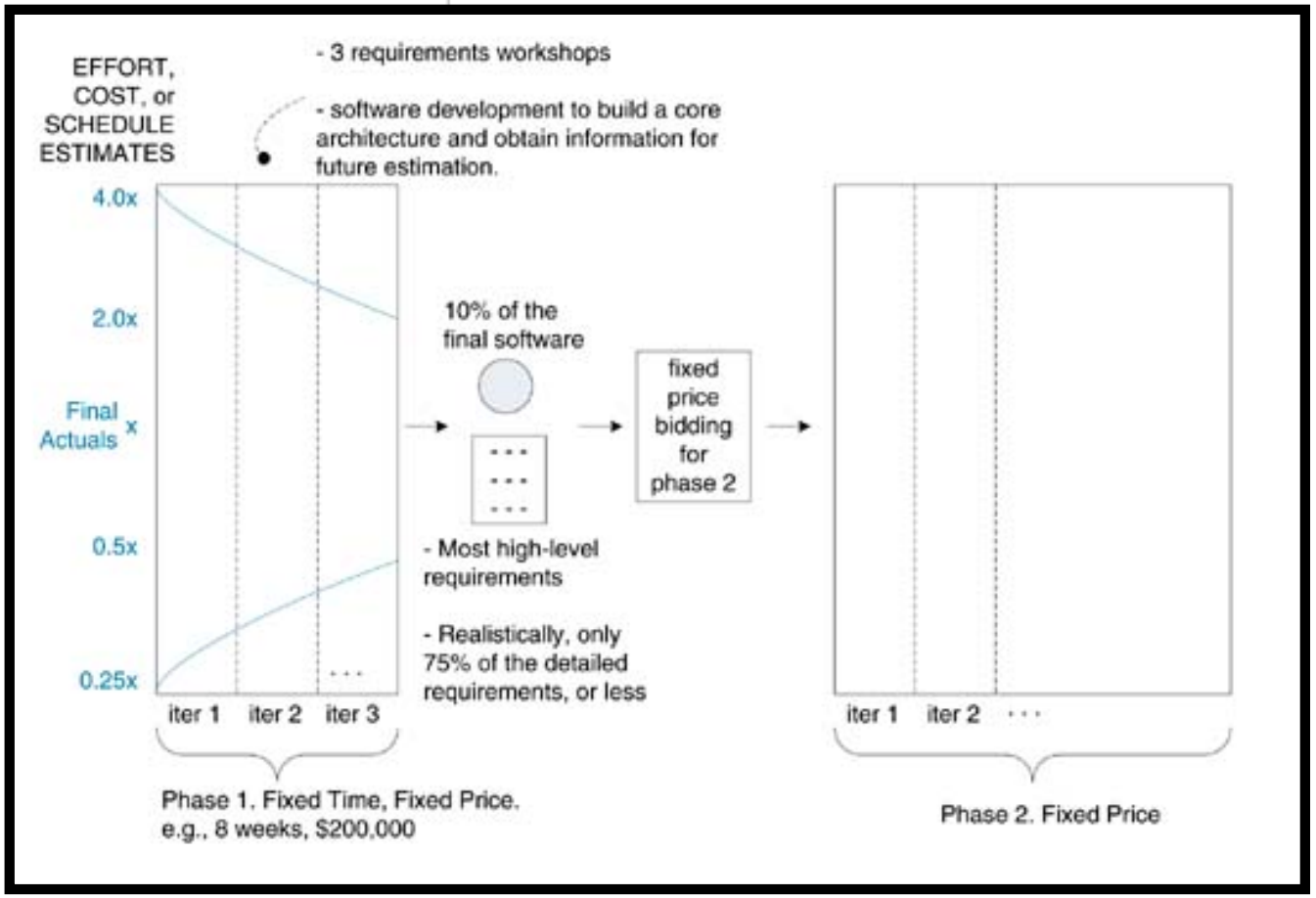




http://www.drdobbs.com/the-agile-edge-unified-and-agile/184415460

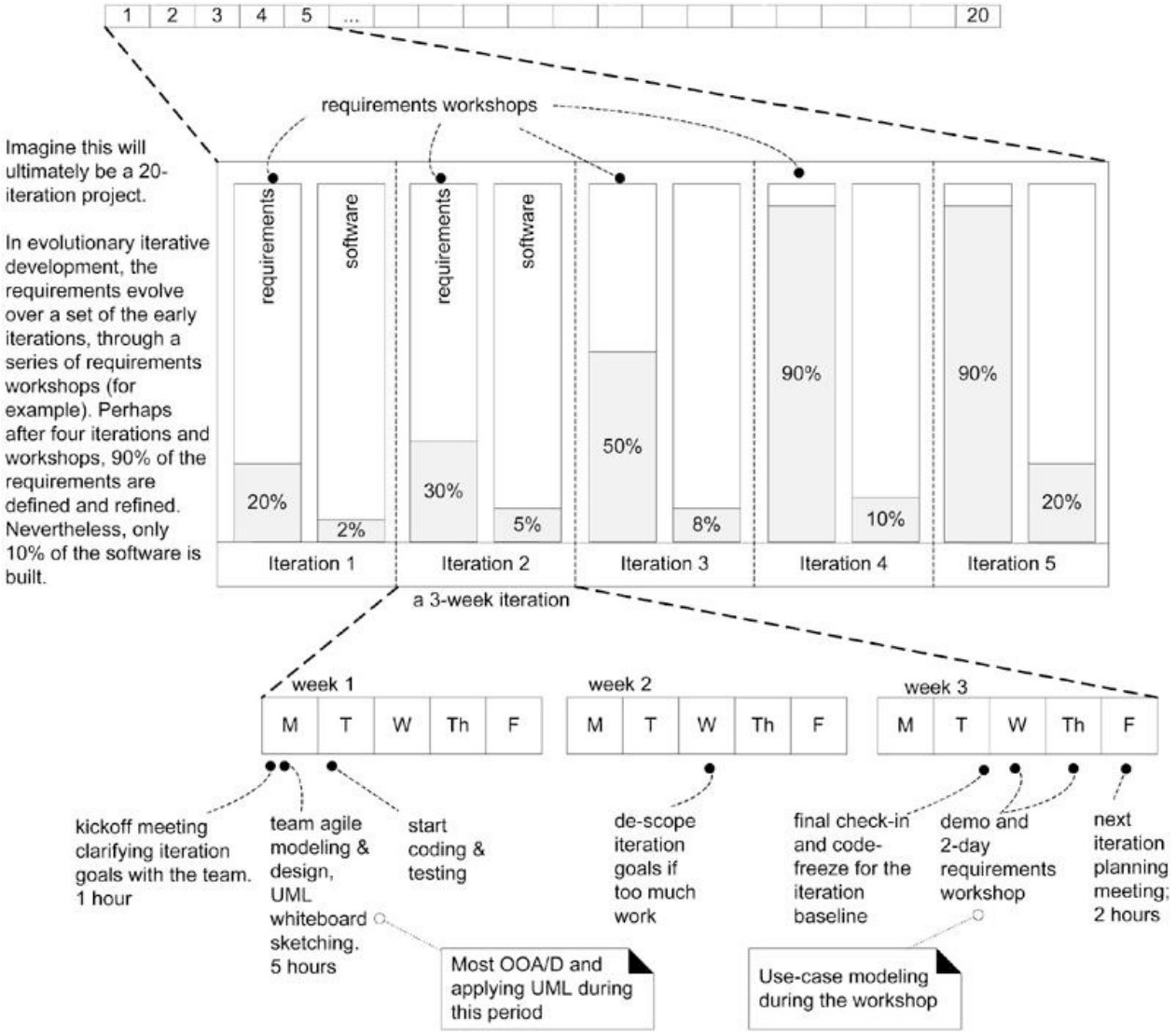
The Cone of Uncertainty -- *Craig Larman*





Evolutionary Analysis & Design

Evolutionary Analysis & Design



FEEDBACK

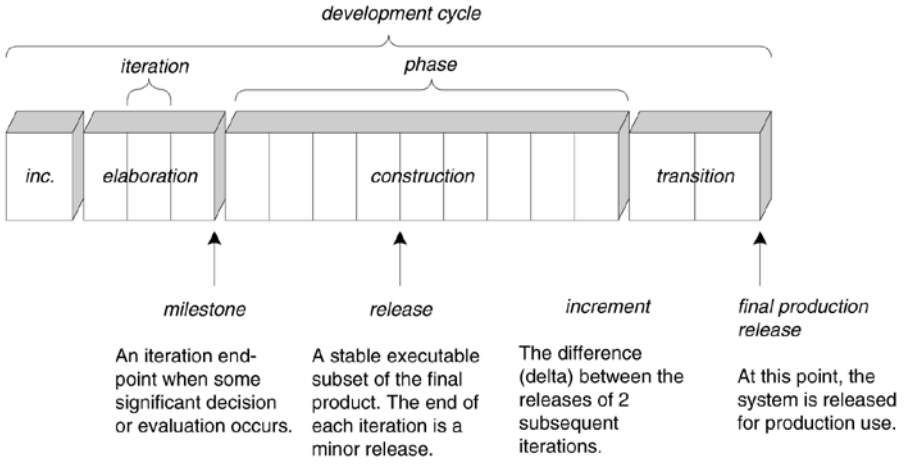
On Early Dev: Demo's to Client, Q&A on Req's Specs ==> Adj. Req's

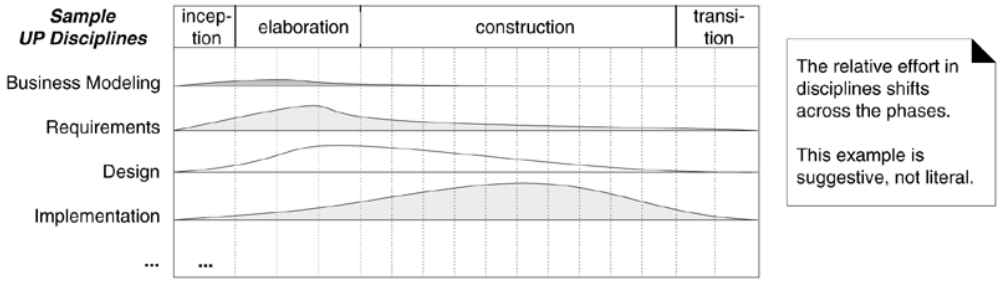
From Testing: Adjust Design? Interfaces Incomplete? ==> Adj. Design.

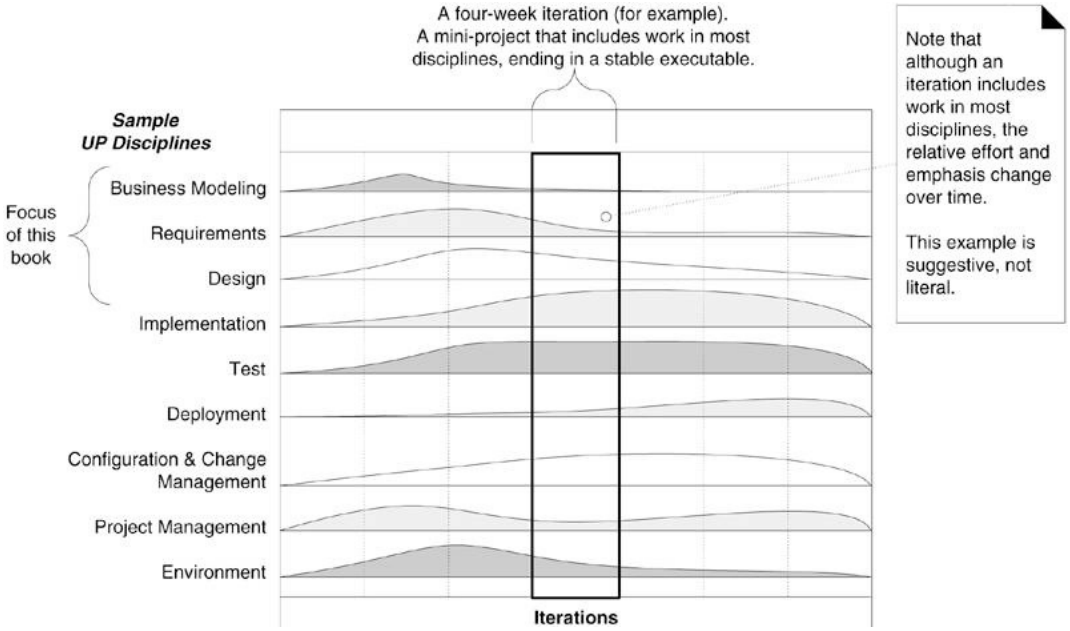
From Progress: Measure Dev Pace (velocity) ==> Adj. Schedule/Scope/ Resources

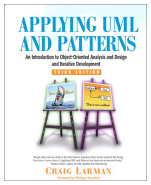
From Market: Features no longer needed ==> Adj. Priority

UP Phases & Disciplines





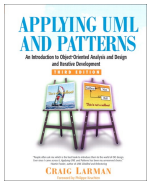




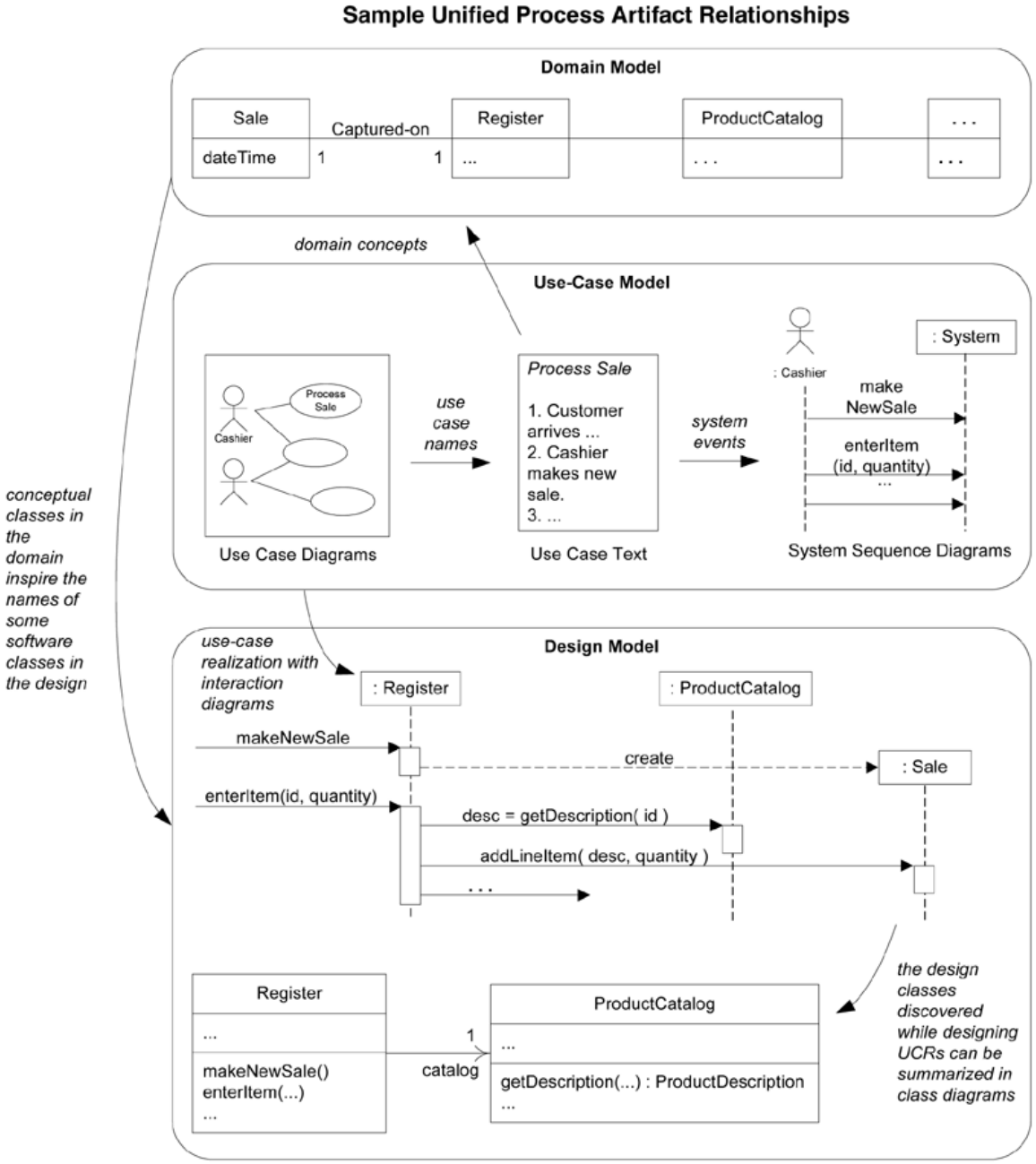
Unified Process

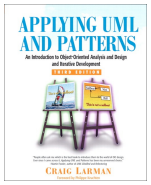
Sample Unified Process Artifacts and Timing (s-start; r-refine)

Discipline	Artifact	Incep.	Elab.	Const.	Trans.
	Iteration→	I1	E1..En	C1..Cn	T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model (code, html, ...)		s	r	r



Unified Process





Unified Process

General Responsibility Assignment Software Patterns or Principles (GRASP)

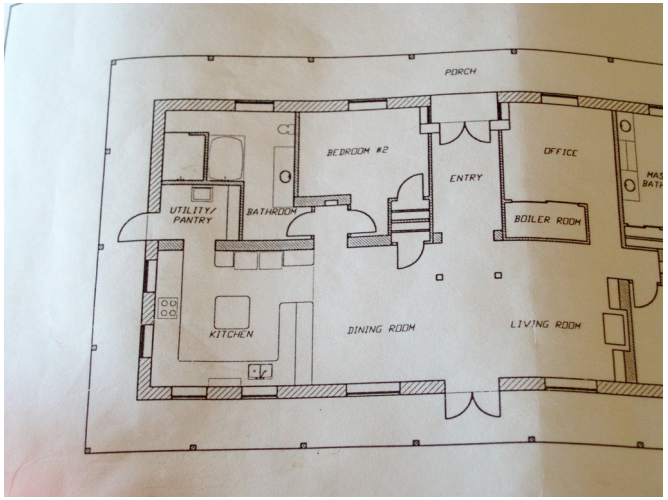
Pattern/Principle	Description
Information Expert	A general principle of object design and responsibility assignment?
	Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true:
	1. B contains A / 4. B records A
	2. B aggregates A / 5. B closely uses A
	3. B has the initializing data for A
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? Assign the responsibility to an object representing one of these choices: **1.** Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). **2.** Represents a use case scenario within which the system operation occurs (a use-case or *session controller*)

Low Coupling (evaluative)	How to reduce the impact of change? Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.
High Cohesion (evaluative)	How to keep objects focused, understandable, and manageable, and as a side-effect, support Low Coupling? Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.
Polymorphism	Who is responsible when behavior varies by type? When related alternatives or behaviors vary by type (class), assign responsibility for the behavior —using polymorphic operations—to the types for which the behavior varies.
Pure Fabrication	Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling? Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent a problem domain concept —something made up, in order to support high cohesion, low coupling, and reuse.
Indirection	How to assign responsibilities to avoid direct coupling? Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.
Protected Variations	How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements? Identify points of predicted variation or instability; assign responsibilities to create a stable "interface" around them.

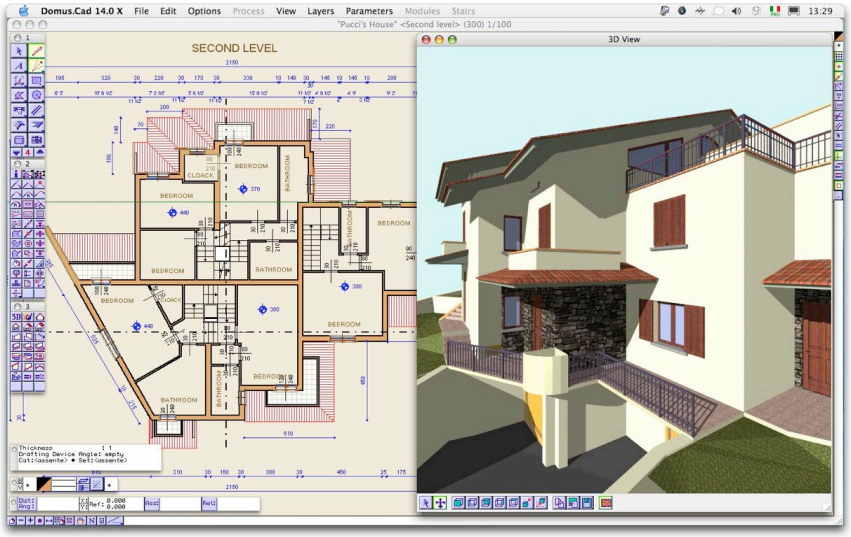
Agile Modeling

Modeling Context

Prototype (Vision Building)



Sketch (Analysis, Design)

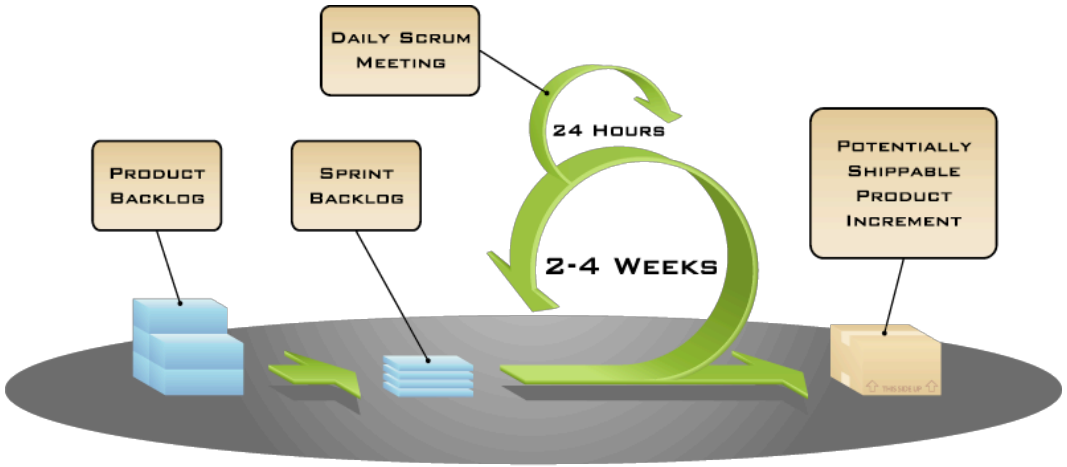


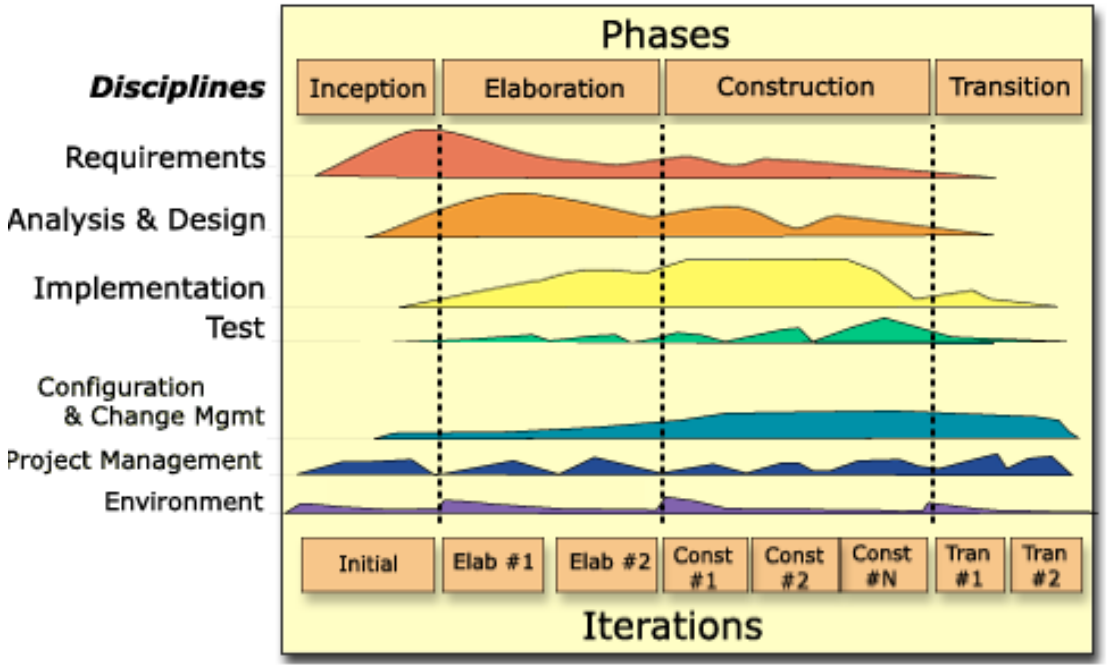
CAD (Correctness Proofs, Estimation)

Build (MDD, Architecture Guidance)

Process + Discipline

The Best of Both!

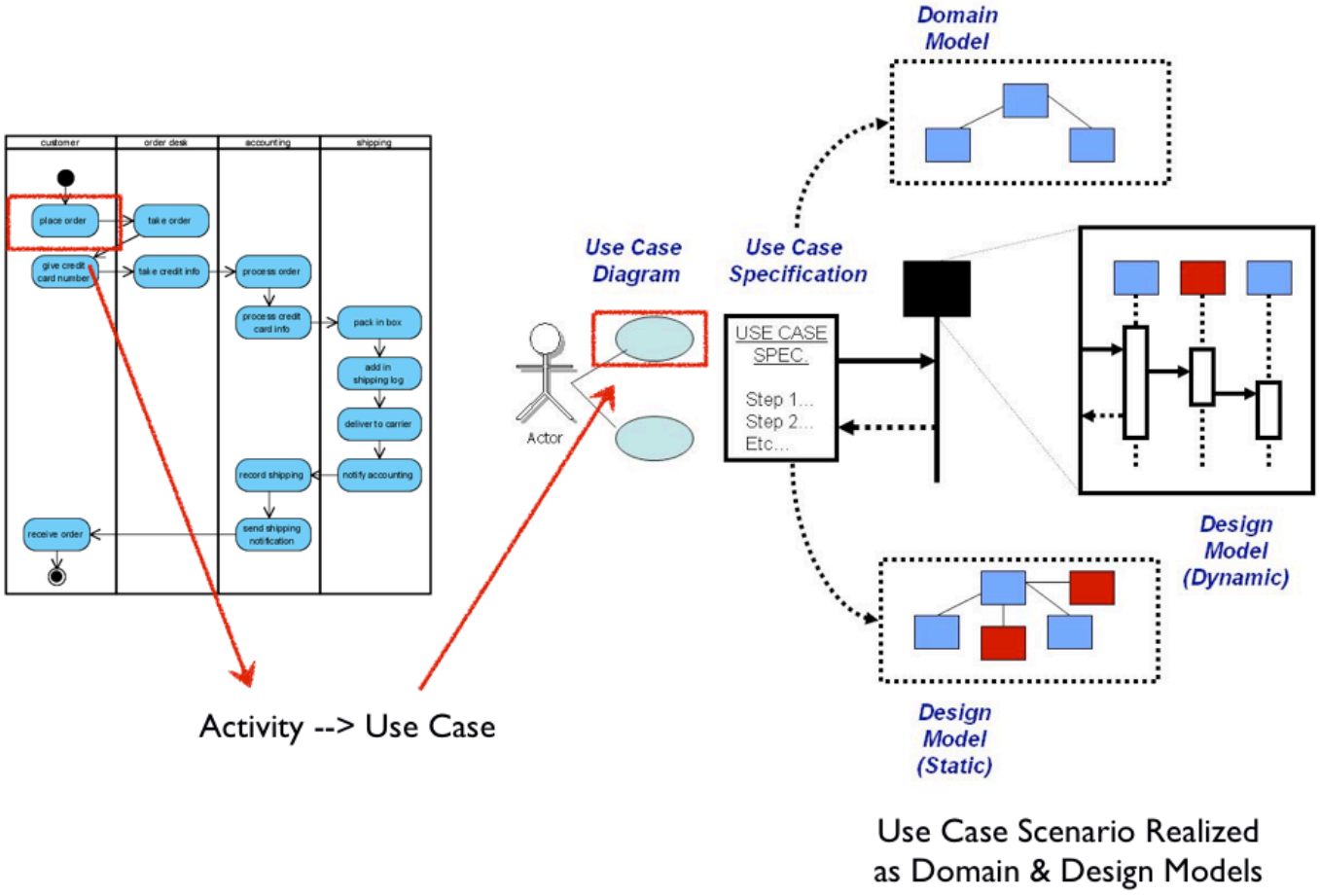




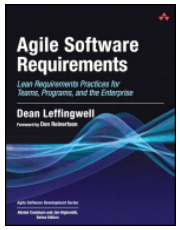
Agile Requirements

UML

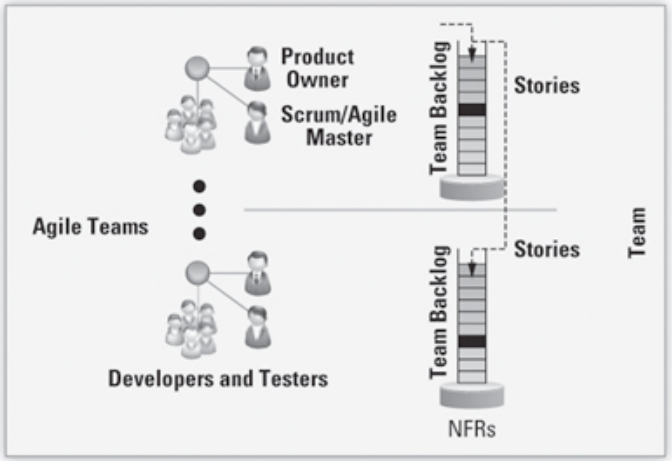
OO Analysis & Design - Core Diagrams



Use Case Scenario Realized as Domain & Design Models



User Stories



User Story Basics

User stories are the agile replacement for most of what has been traditionally expressed as software requirements statements (or use cases in RUP and UML), and they are the workhorses of agile development. Developed initially within the constructs of XP, they are now endemic to agile development in general and are taught in most Scrum classes as well. We'll define a user story as follows:

> A user story is a brief statement of intent that describes something the system needs to do for the user.

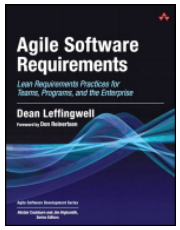
As commonly taught, the user story takes a standard (user voice) form:

> As a <role>, I can <activity> so that <business value>.

In this form, user stories can be seen to incorporate elements of the problem space (the business value delivered), the user's role (or persona), and the solution space (the activity that the user does with the system). Here's an example:

> "As a Salesperson (<role>), I want to paginate my leads when I send mass e-mails (<what I do with the system>) so that I can quickly select a large number of leads (<business value I receive>)."

User stories are so important that **Chapter 6** is devoted entirely to this seminal agile construct.



User Story Form

As a <role>,
I can <activity>
So that <business value>

Details in discussion between PO and team

- A list of what will make the story acceptable to the product owner

User Story Voice

In the last few years, a newer, fairly standardized form has been applied that strengthens the user story construct significantly. The form is as follows:

> As a <role>, I can <activity> so that <business value>.

where:

- <role> represents who is performing the action or perhaps one who is receiving the value from the activity. It may even be another system, if that is what is initiating the activity.
- <activity> represents the action to be performed by the system.
- <business value> represents the value achieved by the activity.

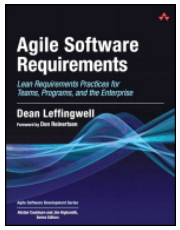
We call this the *user voice* form of user story expression and find it an exceedingly useful construct[5] because it spans the problem space (<business value> delivered) and the solution space (<activity> the user performs with the system). It also provides a user-first (<role>) perspective to the team, which keeps them focused on business value and solving real problems for real people.

[5] While looking for the origin of this form, I received the following note from Mike Cohn: "It started with a team at Connextra in London and was mentioned at XP2003. I started using it then and wrote about it in my 2004 book, *User Stories Applied*."

This user story form greatly enhances the "why" and "how" understanding that developers need to implement a system that truly meets the needs of the users.

For example, a user of a home energy-management system might want to do the following:

> As a Consumer (<role>), I want to be able to see my daily energy usage (<what I do with the system>) so that I can lower my energy costs and usage (<business value I receive>)."



User Story Detail

User Story Detail

The details for user stories are conveyed primarily through conversations between the product owner and the team, keeping the team involved from the outset. However, if more details are needed about the story, they can be provided in the form of an attachment (mock-up, spreadsheet, algorithm, or whatever), which is attached to the user story. In that case, the user story serves as the "token" that also carries the more specific behavior to the team. The additional user story detail should be collected over time (just-in-time) through discussions and collaboration with the team and other stakeholders before and during development.

User Story Acceptance Criteria

In addition to the statement of the user story, additional notes, assumptions, and acceptance criteria can be kept with a user story. *Many* discussions about a story between the team and customers will likely take place while the story is being coded. The alternate flows in the activity, acceptance boundaries, and other clarifications should be captured along with the story. Many of these can be turned into acceptance test cases, or other functional test cases, for the story.

Here's an example:

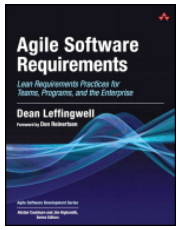
As a consumer, I want to be able to see my daily energy usage so that I can lower my energy costs and usage.

Acceptance Criteria:

- Read DecaWatt meter data every 10 seconds and display on portal in 15-minute increments and display on in-home display every read.
- Read KiloWatt meters for new data as available and display on the portal every hour and on the in-home display after every read.
- No multiday trending for now (another story).

Etc . . . .

Acceptance criteria are not functional or unit tests; rather, they are the conditions of satisfaction being placed on the system. Functional and unit tests go much deeper in testing all functional flows, exception flows, boundary conditions, and related functionality associated with the story.



User Stories to Tasks

Tasks

To assure that the teams really understand the work to be done and to assure that they can meet their commitments, many agile teams take a very detailed approach to estimating and coordinating the individual work activities necessary to complete a story. They do this via the *task*, which we'll represent as an additional model element, as is illustrated in **Figure 3-9**.

Figure 3-9. Stories are implemented by tasks.

Stories are implemented by tasks. Tasks are the lowest-granularity thing in the model and represent activities that must be performed by specific team members to accomplish the story. In our context:

> A task is a small unit of work that is necessary for the completion of a story.

Tasks have an owner (the person who has taken responsibility for the task) and are estimated in hours (typically four to eight). The burndown (completion) of task hours represents one form of iteration status. As implied by the one-to-many relationship expressed in the model, there is often more than one task necessary to deliver even a small story, and it's common to see a mini life cycle coded into the tasks of a story. Here's an example:

Story 51: Select photo for upload

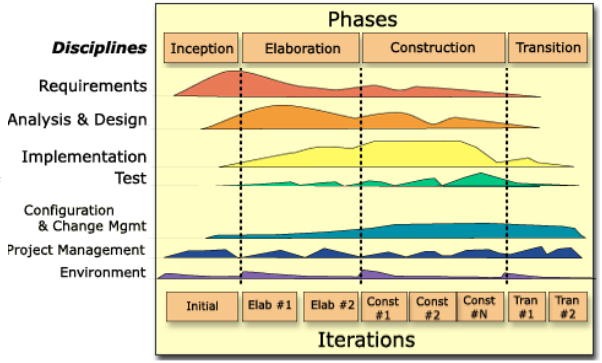
Task 51.1: Define acceptance test—Juha, Don, Bill

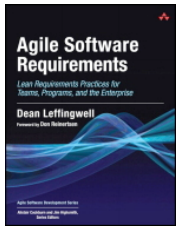
Task 51.2: Code story—Juha

Task 51.3: Code acceptance test—Bill

Task 51.4: Get it to pass—Juha and Bill

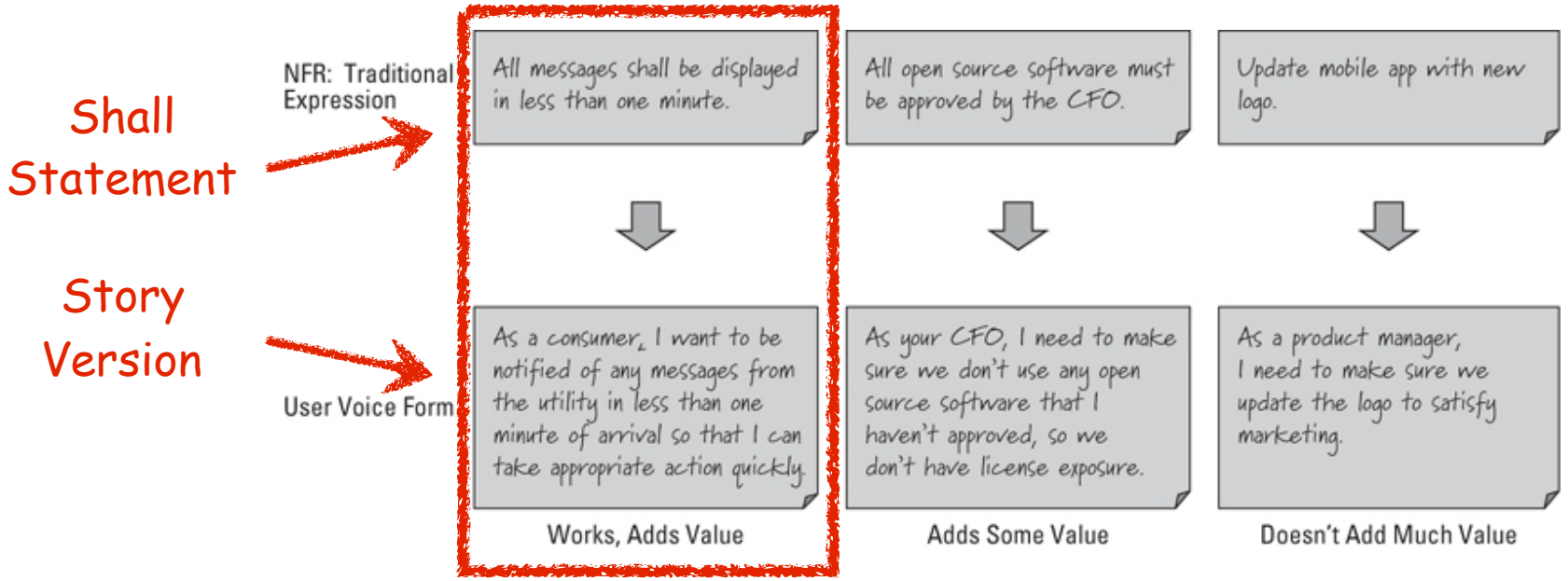
Task 51.5: Document in user help—Cindy





NFR (Non-Functional)

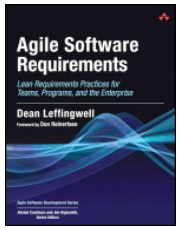
Figure 17-4. Expressing nonfunctional requirements in user voice form



Traditionally, one way to think about *all* the types of requirements that affect overall fitness for use has been the acronym FURPS, which stands for Functionality, Usability, Reliability, Performance, and Supportability [**Grady 1992**, **Leffingwell and Widrig 2003**].

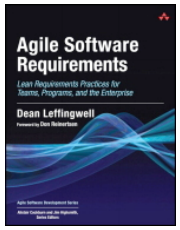
The FURPS acronym reminds us that we must build and manage the behavior of the system from a number of different perspectives:

- *Functionality:* What the system does for the user
- *Usability:* How easy it is for a user to get the system to do it
- *Reliability:* How reliably the system does it
- *Performance:* How often, or how many of it, it can do
- *Supportability:* How easy it is for us to maintain and extend the system that does it



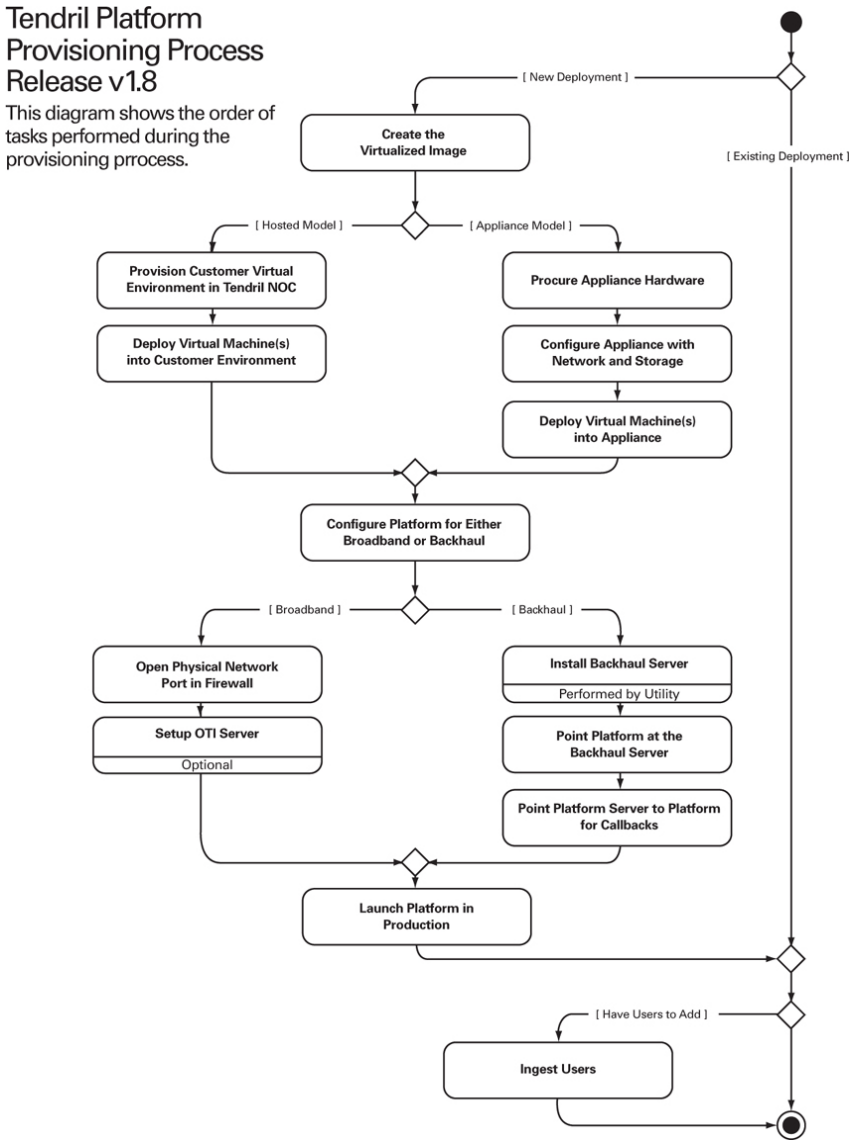
Agile Requirements Toolkit

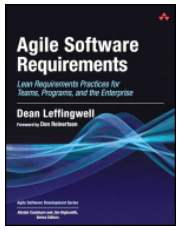
1. Activity diagrams (flowcharts)
2. Sample reports & Wireframes
3. Pseudocode
3. Decision tables and decision trees
4. Finite state machines
5. Message sequence diagrams
6. Entity-relationship diagrams
7. Use cases



Requirements Toolkit

Activity Diagram



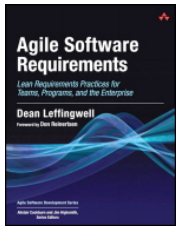


Requirements Toolkit

Sample Report

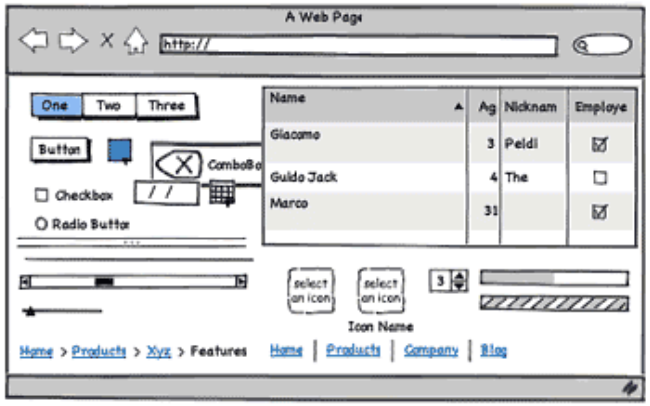
EVENTS

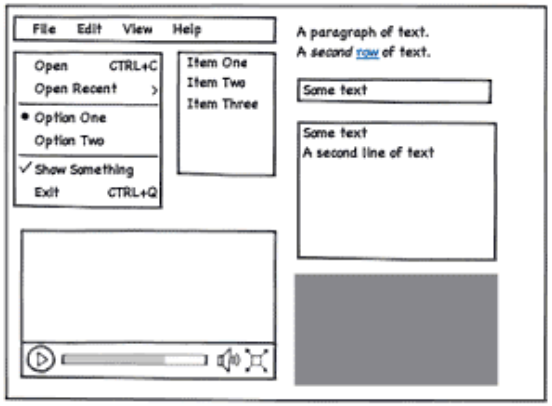
Status	Event ID	Type	Start	End	Duration	Customer Response
▼ PENDING						
Opt-Out	78	Load Control	4:00PM Jul 14	10:00PM Jul 14	6:00	Opt In
Opt-Out	79	Load Control	4:00PM Jul 15	10:00PM Jul 15	6:00	Opt In
Opt-Out	80	Load Control	4:00PM Jul 16	10:00PM Jul 16	6:00	Opt In
▼ COMPLETE						
	77	Load Control	4:00PM Jul 12	10:00PM Jul 12	6:00	Partial Participation
	76	Load Control	4:00PM Jul 11	10:00PM Jul 11	6:00	Opt In
	75	Load Control	4:00PM Jul 10	10:00PM Jul 10	6:00	Opt In
	74	Load Control	4:00PM Jul 8	10:00PM Jul 8	6:00	Opt In
	73	Load Control	4:00PM Jul 7	10:00PM Jul 7	6:00	Opt In
	72	Load Control	4:00PM Jul 6	10:00PM Jul 6	6:00	Opt In
	71	Load Control	4:00PM Jul 4	10:00PM Jul 4	6:00	Opt In
	70	Load Control	4:00PM Jul 3	10:00PM Jul 3	6:00	Opt In
	69	Load Control	4:00PM Jul 2	10:00PM Jul 2	6:00	Opt In
	68	Load Control	10:00AM Jun 25	11:00AM Jun 25	1:00	Opt In

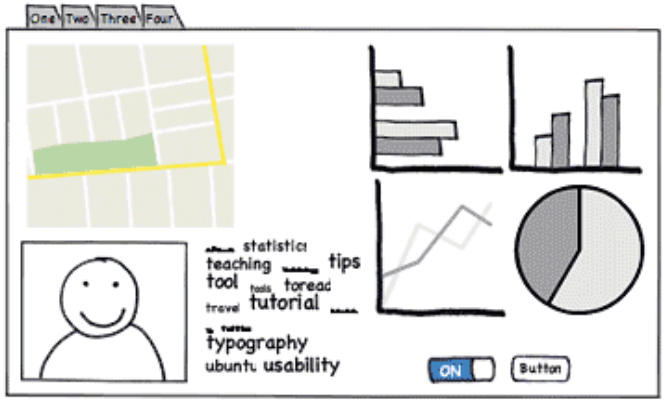


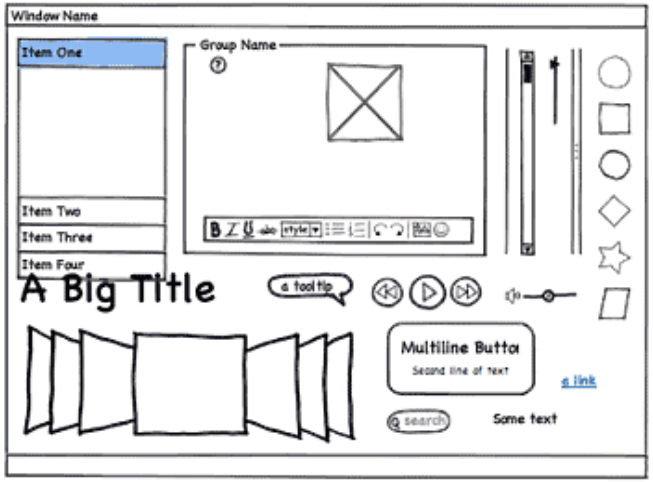
Requirements Toolkit

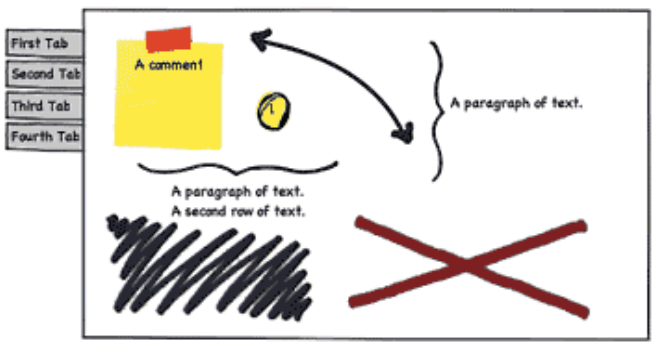
Wireframes

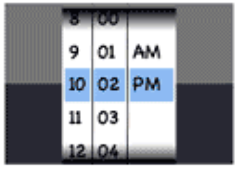


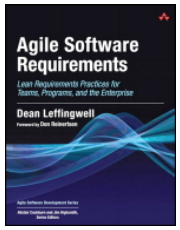








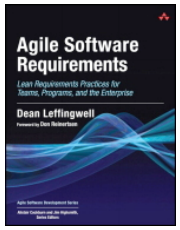




Requirements Toolkit

Pseudocode

```
Set SUM(x)=0
FOR each customer X
        IF customer purchased paid support
            AND((Current month)>=(2 months after ship date))
            AND((Current month)<=(14 months after ship date))
        THEN Sum(X)=Sum(X)+(amount customer paid)/12
END
```



Requirements Toolkit

Decision Tree/Table

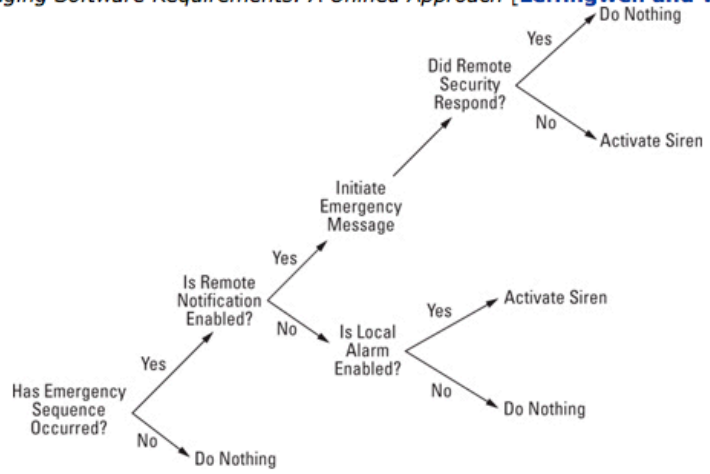
Figure 18-4. Decision table for debugging a printer failure

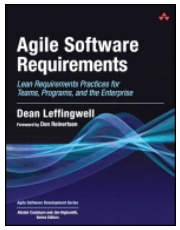
		Rules							
	Printer does not print.	Y	Y	Y	Y	N	N	N	N
Conditions	A red light is flashing.	Y	Y	N	N	Y	Y	N	N
	Printer is unrecognized.	Y	N	Y	N	Y	N	Y	N
	Check the power cable.			X					
	Check the printer-computer cable.	X		X					
Actions	Ensure printer software is installed.	X		X		X		X	
	Check/replace ink.	X	X			X	X		
	Check for paper jam.		X		X				

Source: Wikipedia *(en.wikipedia.org/wiki/Decision_table)*

Figure 18-5. Example of a graphical decision tree

Source: *Managing Software Requirements: A Unified Approach* [**Leffingwell and Widrig 2000**]





Requirements Toolkit

Finite State Machines

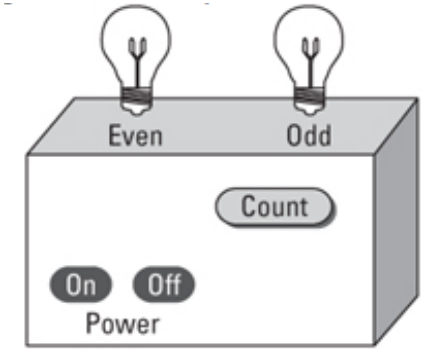


Figure 18-7. Example of a state transition diagram

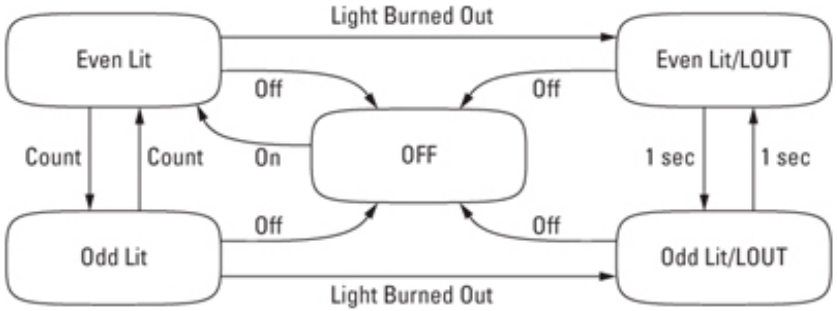
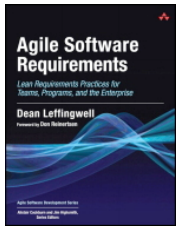


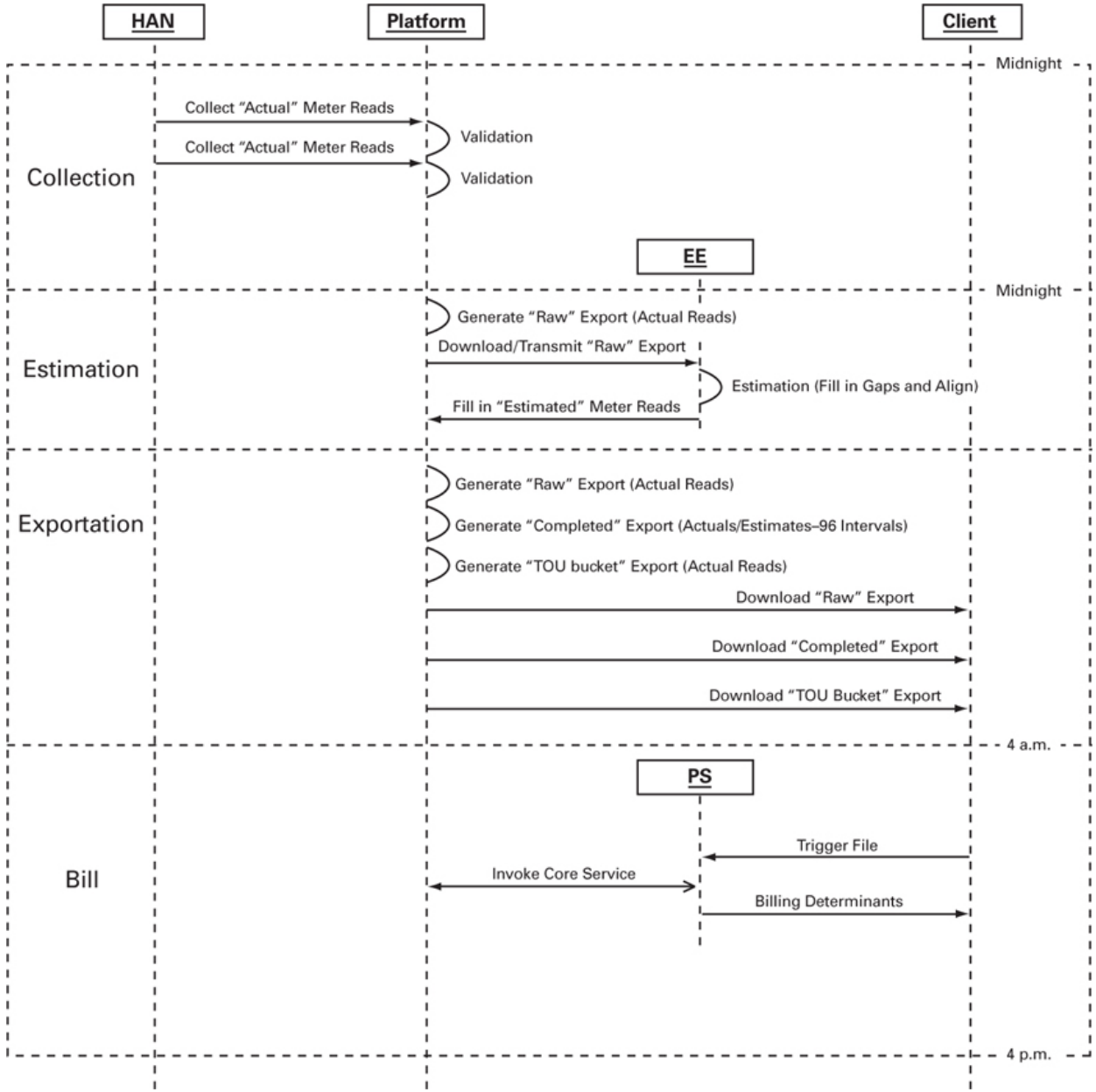
Figure 18-8. Example of a state transition matrix

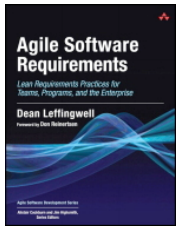
	Event					
State	**On Press**	**Off Press**	**Count Press**	**Bulb Burns Out**	**Every Second**	**Output**
Off	Even Lit	—	—	—	—	Both Off
Even Lit	—	Off	Odd Lit	LO/Even Lit	—	Even Lit
Odd Lit	—	Off	Even Lit	LO/Odd Lit	—	Odd Lit
Light Out/Even Lit	—	Off	—	Off	LO/Odd Lit	Even Lit
Light Out/Odd Lit	—	Off	—	Off	LO/Even Lit	Odd Lit



Requirements Toolkit

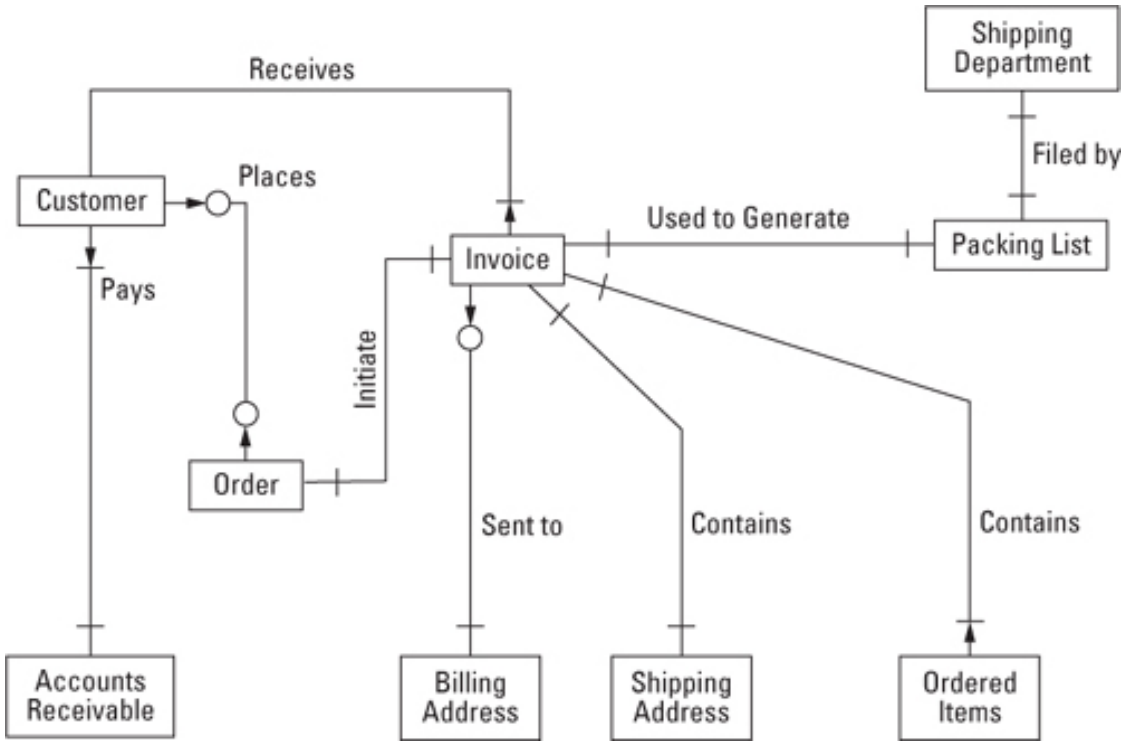
Message Sequence Diagrams



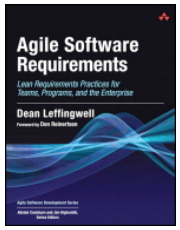


Requirements Toolkit

Entity-Relationship Diagrams



Note: ERD kept at high-level (i.e. conceptual) focusing only on entities and relationships



Requirements Toolkit

Use Cases

Chapter 19. Use Cases

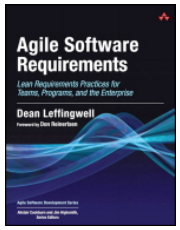
A user story is to a use case as a gazelle is to a gazebo.

—Alistair Cockburn

In **Chapter 1**, we provided a brief history of requirements methods and briefly mentioned the role of use cases as a form of requirements capture and expression. Popularized originally within the context of the Rational Unified Process (RUP), which was *use case–driven* and architecture-centric, for many, use cases have been the requirements analysis and communication expression of choice. Even outside RUP, they appeared in most contemporary works on software requirements and systems analysis. Use cases were also the container for functional requirements capture, analysis, and behavioral specification within the context of the Unified Modeling Language (UML). Those who used the UML most likely used use cases. If you have been doing iterative development, you are probably using use cases too.

In agile development, however, the picture changed as the user story (or even more simply, the backlog item in Scrum) became the predominant form. As we have described throughout this book, there can be no doubt of their value in lightening requirements expression, driving more and more incremental thinking, and generally increasing the agility of the teams that use them. User stories are good requirements tools. Use cases were largely banned from the agile tribes.[1]

[1] One Certified Scrum Product Owner course stated, "Don't use use cases. They are too hard to write, and users don't understand them."



Requirements Toolkit

The Problems with User Stories and Backlog Items

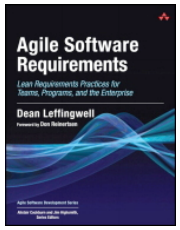
Alistair Cockburn is one agile thought leader with his foot in both of these camps. As both an authority on use cases [**Cockburn 2001**] and a respected agilist and signer of the Agile Manifesto, he bemoans the apparent loss of use cases from agile development. He notes that there are many problems with the user story and backlog forms of requirements expression:[2]

[2] See ***http://alistair.cockburn.us/A+user+story+is+to+a+use+case+as+a+gazelle+is+to+a+gazebo***. Portions reproduced here with permission. Minor edits by the author.

User stories and backlog items don't give the designers a context to work from—when is the user doing this, what is the context of their operation, and what is their larger goal at this moment?

User stories and backlog items don't give the project team any sense of scope or potential "completeness"—a development team estimates a project at (e.g.) 270 story points, and then as soon as they start working, that number keeps increasing, seemingly without bound. The developers and sponsors are equally depressed. How big is this project, really?

User stories and backlog items don't provide a mechanism for looking ahead at upcoming work. Seeing a set of extension conditions (alternate flows) in a use case lets the analysts understand which ones will be easy and which will be difficult so they can stage the work accordingly. With user stories, the extension conditions are usually detected mid-sprint, when it is too late.



Requirements Toolkit

Five Good Reasons to Still Use Use Cases

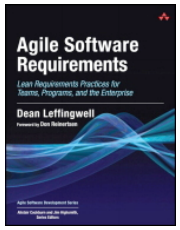
The list of goal names provides executives with a short summary

The main success scenario of the use case provides everyone with an agreement as to what the system will...and will not do.

The extension conditions of the use case provide a framework for investigating all the little, niggling things that somehow take up 80% of the development time and budget.

The use case extension scenarios provide answers to the many detailed, tricky business questions programmers ask: "What are we supposed to do in this case?"

The full use case set (use case model) shows that the developers/analysts have thought through every user's needs, every goal they have with respect to the system, and every business variant involved.



Use Cases

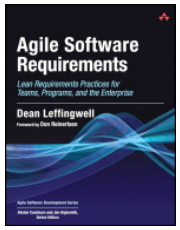
Figure 19-1. Use case template

Use Case Name
Description
Actor(s)
Flow of Events
Basic Flow
Event 1
Event 2
.....
Alternate Flow
Preconditions
Exit Conditions
Success Guarantee
Minimum Guarantee

A use case has four mandatory elements.

- *Name:* The name describes the goal, that is, what is achieved by the interaction with the actor. The name should be a few words in length, and it must be unique. Names such as pop up an emergency warning message and shed refrigerator load are good examples. They are short, are descriptive, and define the goal.
- *Brief description:* The purpose of the use case should be described in one or two sentences. Here's an example: this use case describes what happens when the Tendril system receives an event notification from the utility.
- *Actor(s):* A use case has no meaning outside the context of its use by an actor, so each actor who participates in the use case is listed with the use case.
- *Flow of events:* The main body of the use case is the event flow, usually a textual description of the interactions between the actor and the system. The flow of events can consist of both the *basic flow*, which is the main path through the use case, and *alternate flows*, which are executed only under certain circumstances.

  It is the *alternate flows* (or *alternate scenarios*) that provide much of value to the agile system builder. It is here where they are forced to think through all the "what ifs" that might affect our user story. Here's an example: "what happens if the device does not respond to the message?" or "if I'm programming the system, and an event happens—what happens then?" Understanding all the alternate flows of a use case defines the various (usually less likely but equally important) scenarios that the system must handle with grace in order to assure reliability and quality.



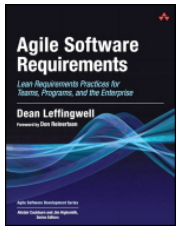
Use Cases

Figure 19-1. Use case template

Use Case Name
Description
Actor(s)
Flow of Events
Basic Flow
Event 1
Event 2
.....
Alternate Flow
Preconditions
Exit Conditions
Success Guarantee
Minimum Guarantee

In addition to the mandatory elements, a use case may have optional elements.

- *Preconditions:* Preconditions are those conditions that must be present in order for a use case to start. They usually represent some system state that must be present before the use case can be used. For example, a precondition of the set thermostat to shed load use case is that the system must have opt-in for load shedding enabled.
- *Exit conditions:* Exit conditions describe the state of the system after a use case has run its course. These can include both a *success guarantee*, which describes the state of the system after a successful execution, and a *minimum guarantee*, which describes the state of the system if the execution of the use case fails for some reason. For example, a *success guarantee* of the set thermostat to shed load use case is that the system remains in the load shedding state. A *minimum guarantee* might be that load shedding is not initiated but an error message is displayed.
- *System or subsystem:* In a system of subsystems, it may be necessary to identify whether a use case is a system-level use case (one that causes multiple subsystems to interact) or a subsystem use case. In either case, the team needs to identify what system or subsystem a use case is identified with.
- *Other stakeholders:* It may also be useful to identify other key stakeholders who may be affected by the use case. For example, a manager may use a report from the system, and yet the manager may not personally interact with the system and would therefore not appear as an actor.
- *Special requirements:* The use case may also have special requirements that apply to that specific use case. Often these are nonfunctional requirements (support 10,000 homes without performance degradation) that apply to the specific use case.



Use Case Example

Use Case Name: Issue brownout notification
Brief Description: Upon determining that a brownout condition is pending, the utility sends a message to all registered devices in the utility's Tendril domain. This will include notification to all the utility's registered on-premise displays, mobile devices, and portals.
Actors:
• Utility Network Operating Center operators (UNOC) (primary actor)
• Tendril customers
Basic Flow of Events:
1. A UNOC operator determines that a brownout event is pending.
2. The operator composes a broadcast message in TUP.
3. TUP sends the message to all affected customers, either over the utility backhaul or through an Internet (IP) connection, to all registered Tendril devices in the consumer's home, and logs that the message has been sent.
4. Each Tendril device in the home returns to TUP a confirmation of successful message receipt and presents the upcoming brownout message to the homeowner in its device-specific format.
5. TUP adds each confirmation to its record of confirmations received.
Alternate Flows:
• 4a. Home Tendril device fails to confirm (initially): At established intervals, TUP resends the brownout event notice to all Tendril devices that did not reply.
• 4b. Home Tendril device fails to confirm after the specified number of retries: TUP notifies the UNOC operator of the situation of the nonacknowledging Tendril home device.

Preconditions:
• Receipt of overload conditions pending on the utility grid.
• Determination of areas to be affected by brownout and matching areas to preset Tendril notification zones.
Success Guarantee:
• The brownout notice has been sent.
• Every home device has been accounted for, through either confirmation or total failure to confirm.
• The UNOC operator as been notified of all home devices that failed to confirm.
Minimal Guarantee:
• In the worst case, the TUP log has captured the state of all notification-confirmation pairs for every home device on the subscription list.

Tips for Applying Use Cases in Agile

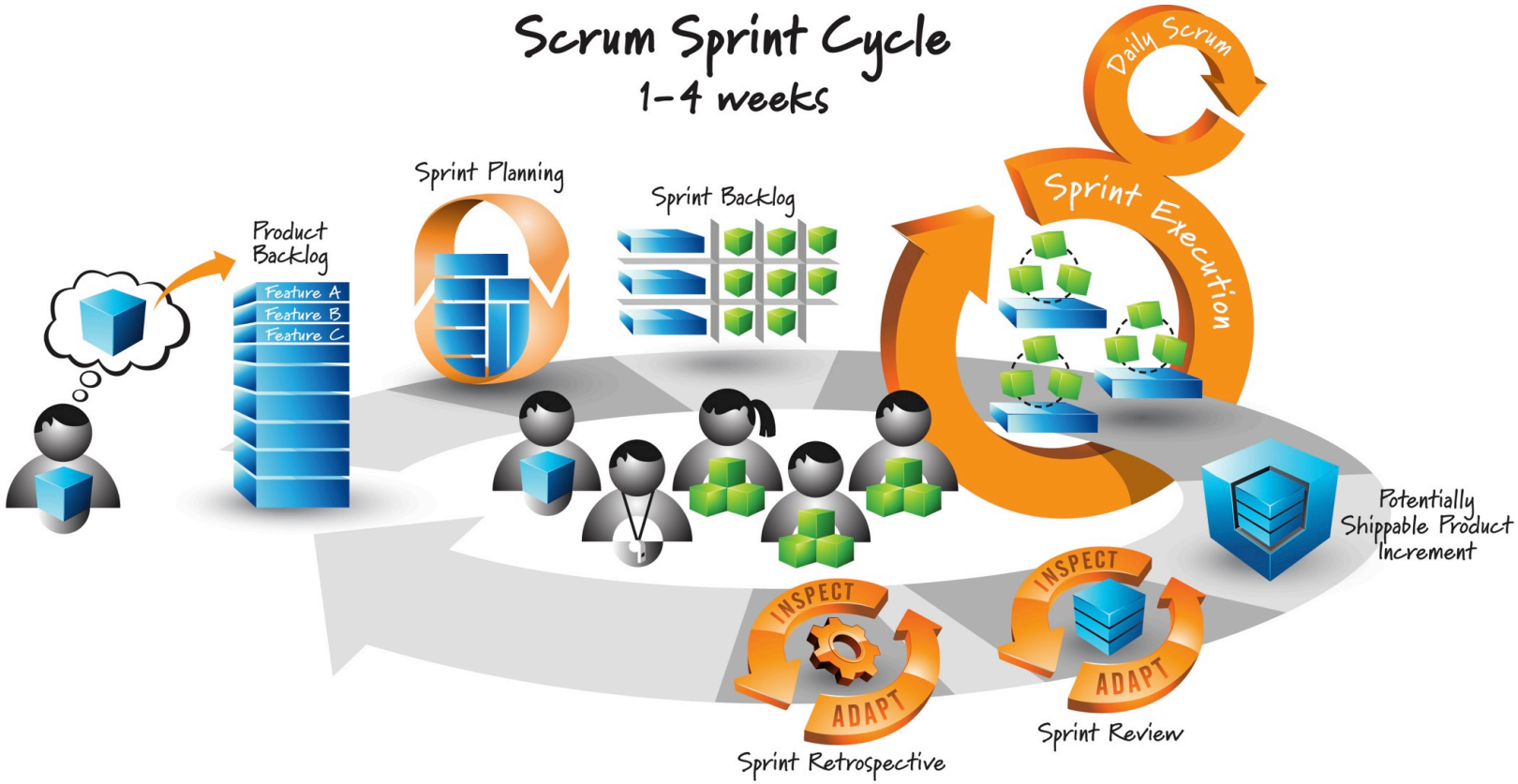
Finally, here are a few tips to keep in mind when you apply use cases in agile development.

- Keep them lightweight—no design details, GUI specs, and so on.
- Don't treat them like fixed requirements. Like user stories, they are merely statements of intended system behavior.
- Don't worry about maintaining them; they are primarily thinking tools.
- Model them informally—use whiteboards, lightweight tools, and so on.

Remember, you don't have to use use cases, but no one can tell you not to use them either. And if your system is complex, you'll likely be quite happy if you do.

Scrum

Scrum



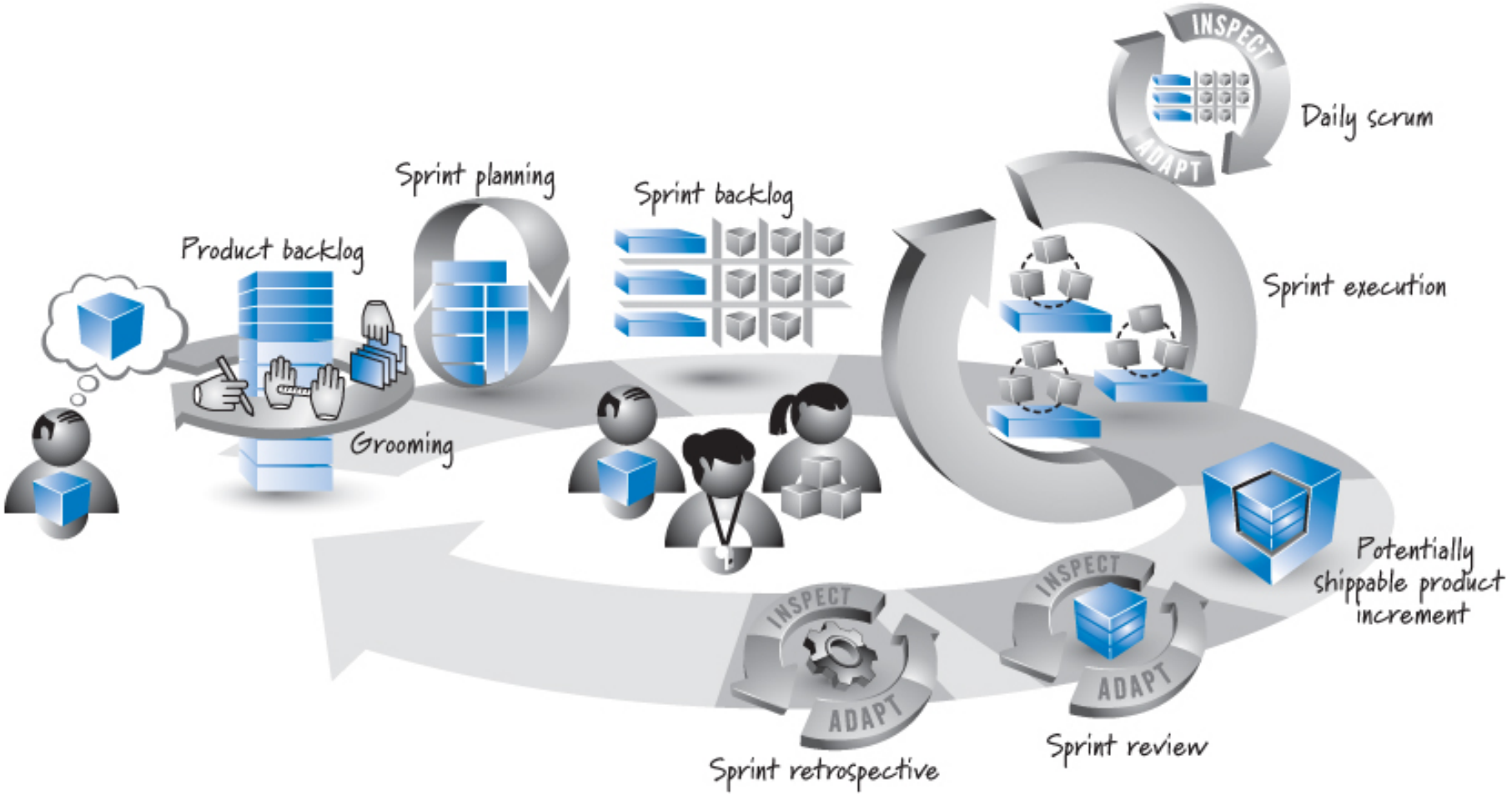
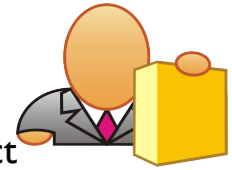


Figure 2.3. Scrum framework

Product owner



- Define the features of the product
- Decide on release date and content
- Be responsible for the profitability of the product (ROI)
- Prioritize features according to market value
- Adjust features and priority every iteration, as needed
- Accept or reject work results

The ScrumMaster

- Represents management to the project
- Responsible for enacting Scrum values and practices
- Removes impediments
- Ensure that the team is fully functional and productive
- Enable close cooperation across all roles and functions
- Shield the team from external interferences

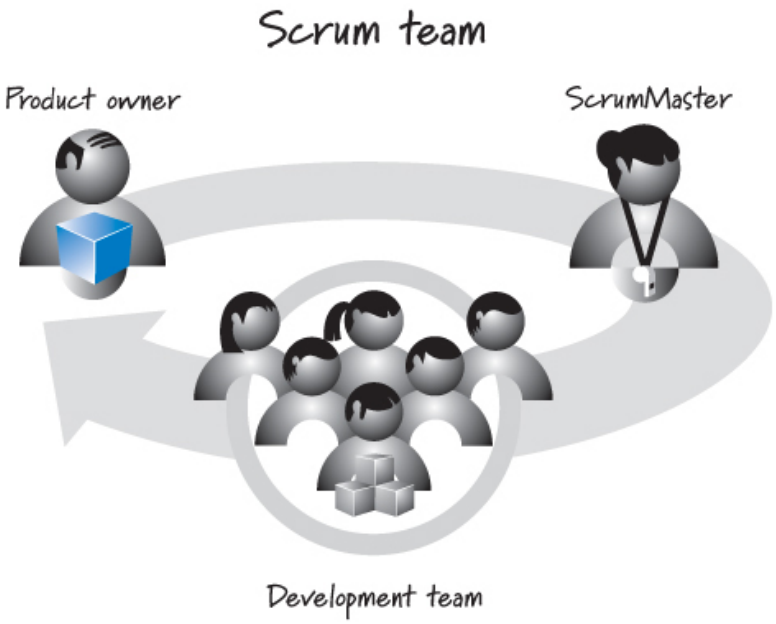
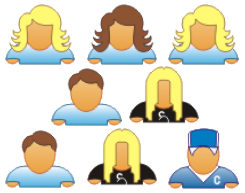


Figure 2.2. Scrum roles

The team



- Typically 5-9 people
- Cross-functional:
  - Programmers, testers, user experience designers, etc.
- Members should be full-time
  - May be exceptions (e.g., database administrator)
- Teams are self-organizing
  - Ideally, no titles but rarely a possibility
- Membership should change only between sprints

The Product Backlog:

A list of all desired work on the project

Ideally expressed such that each item has value to the users or customers of the product

Prioritized by the product owner

Reprioritized at the start of each sprint

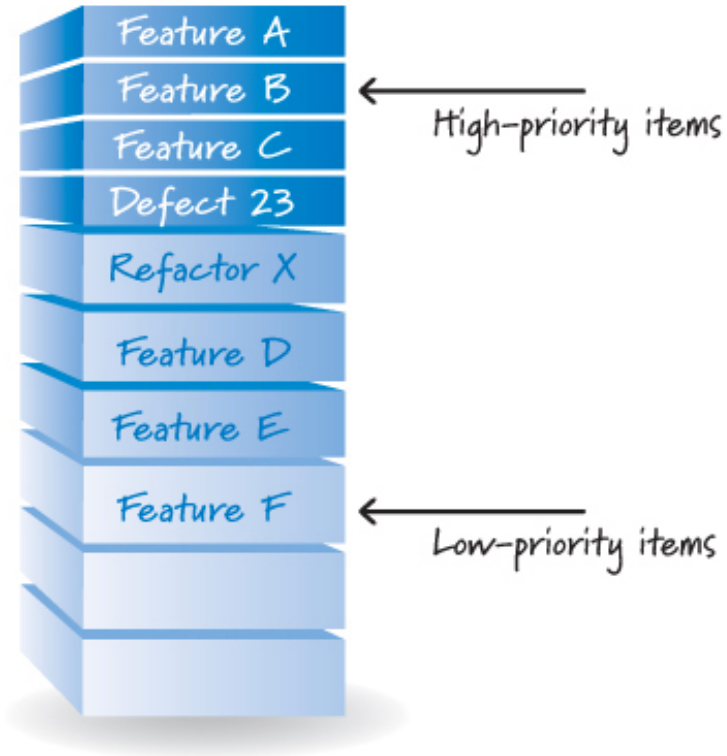


Figure 2.4. Product backlog

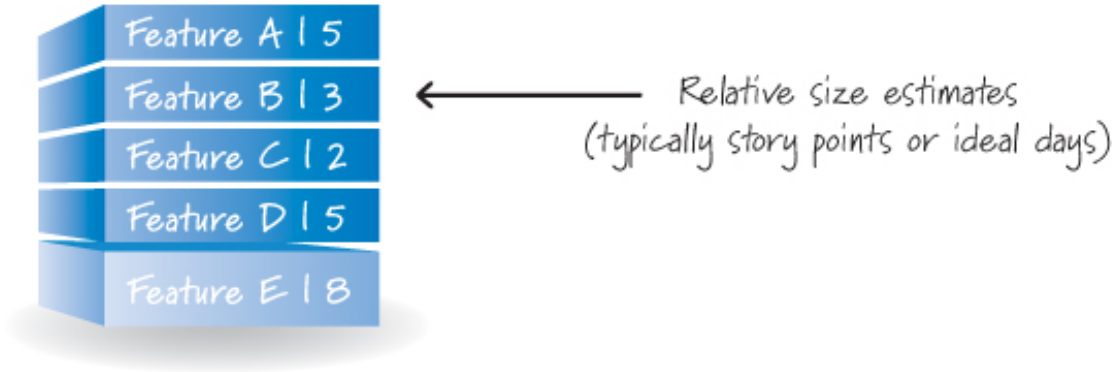


Figure 2.6. Product backlog item sizes

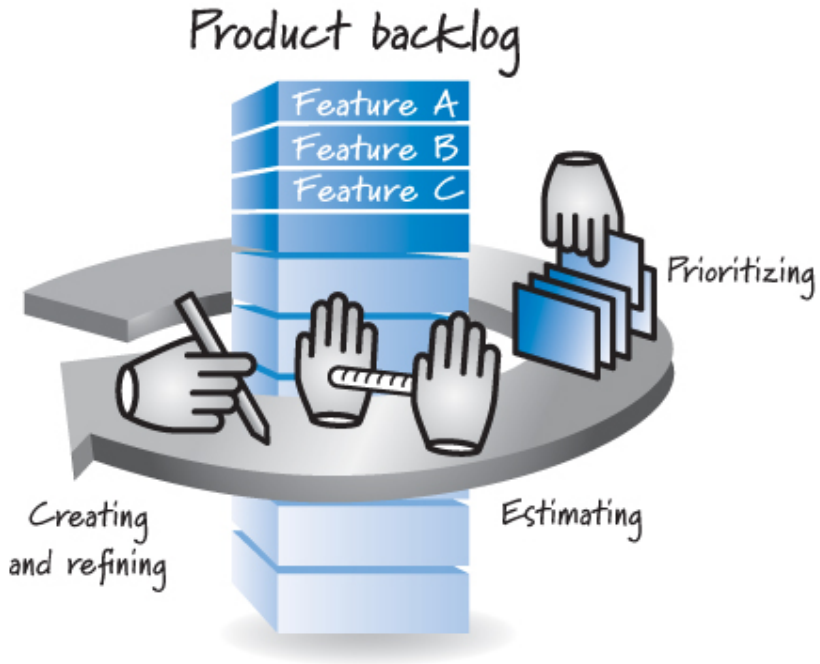
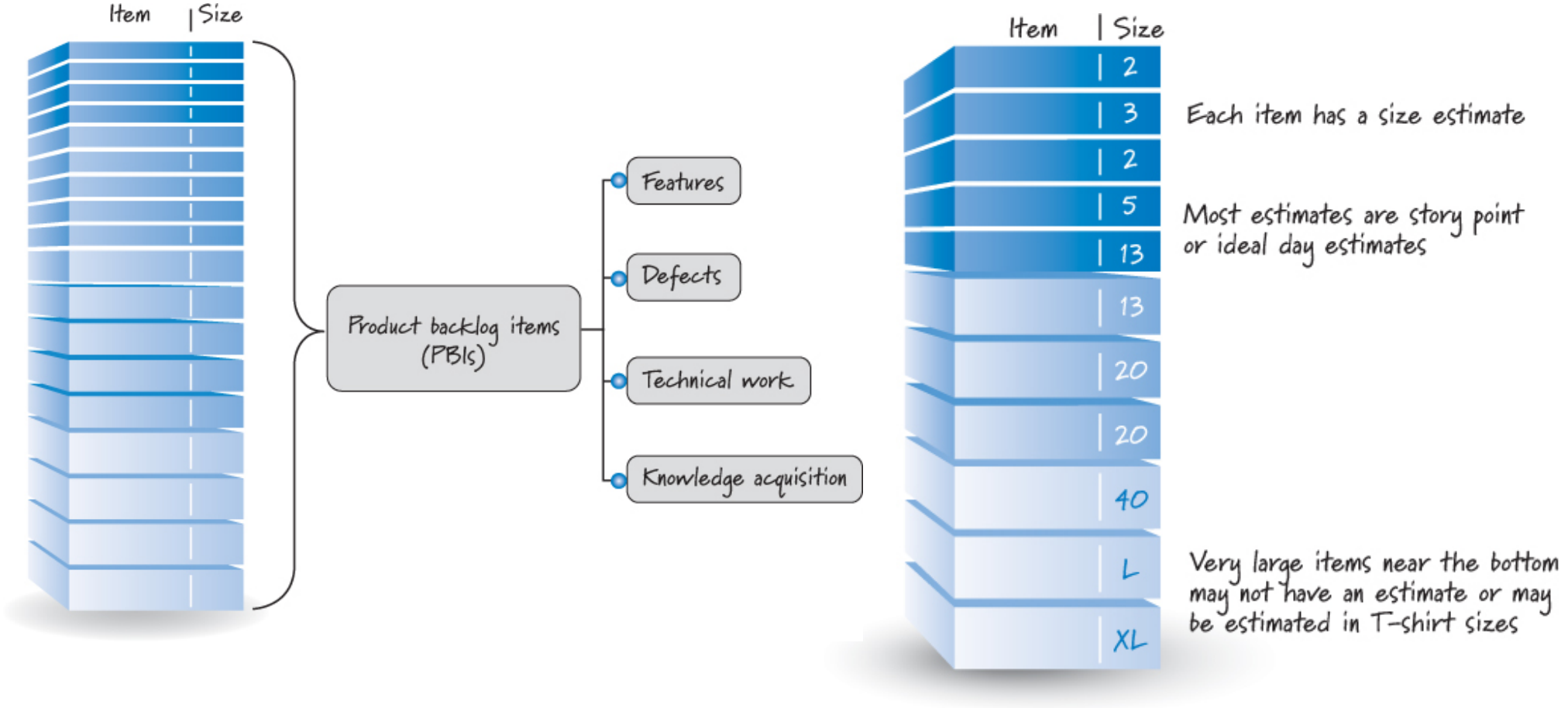
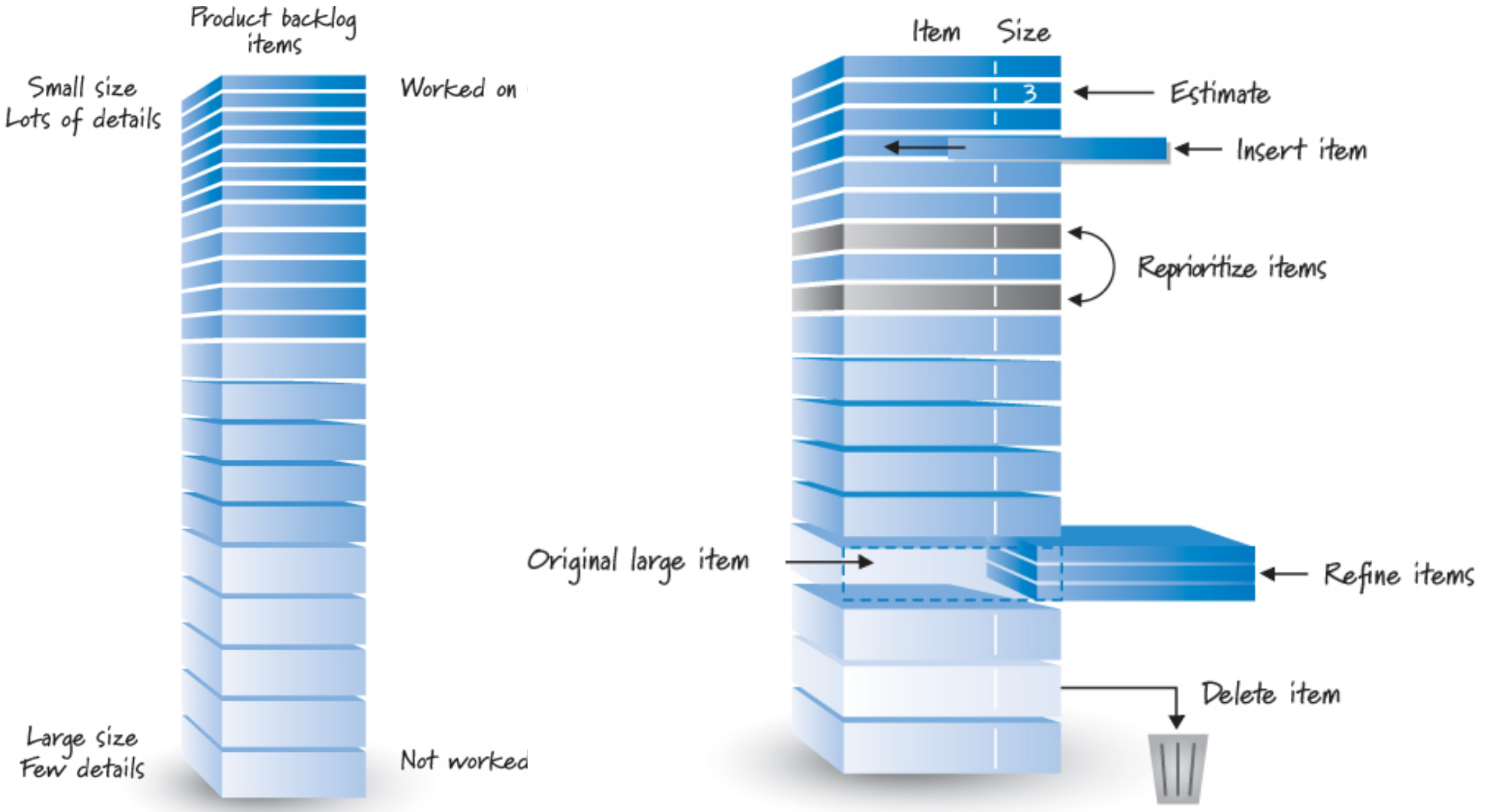


Figure 2.5. Product backlog grooming

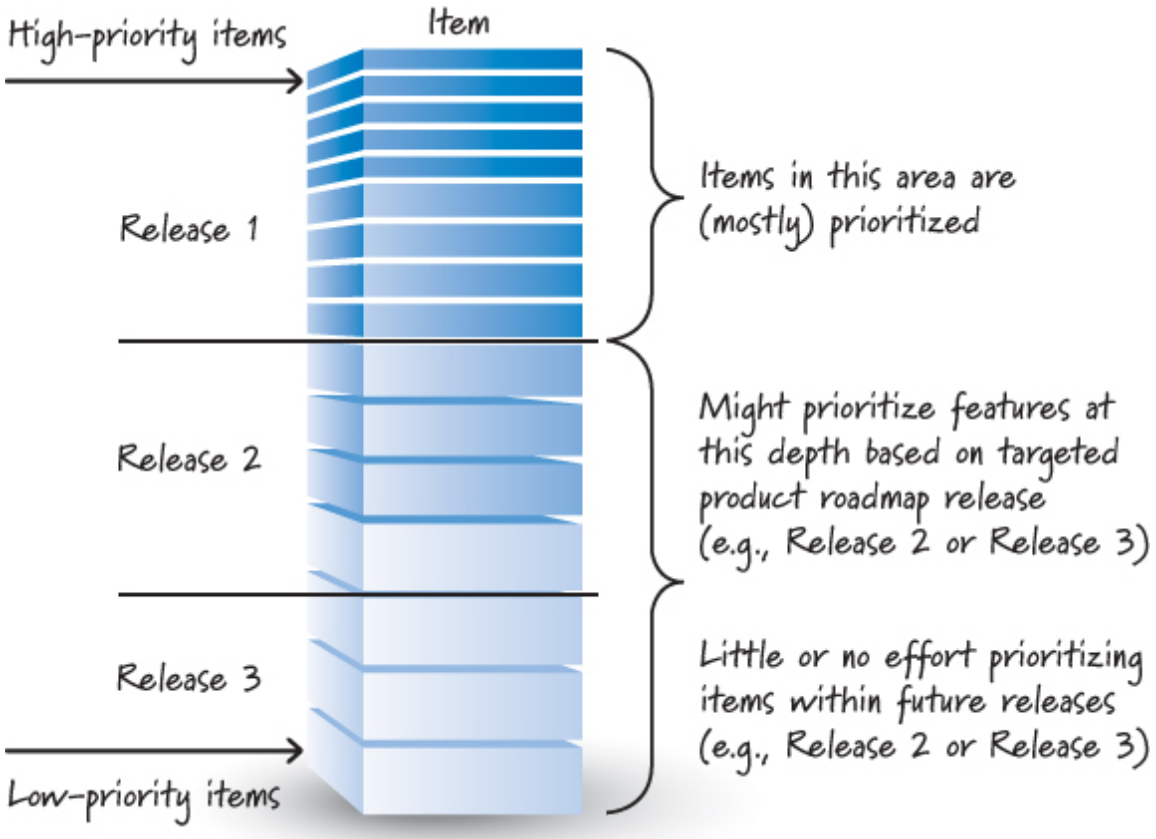
Product Backlog Sizing

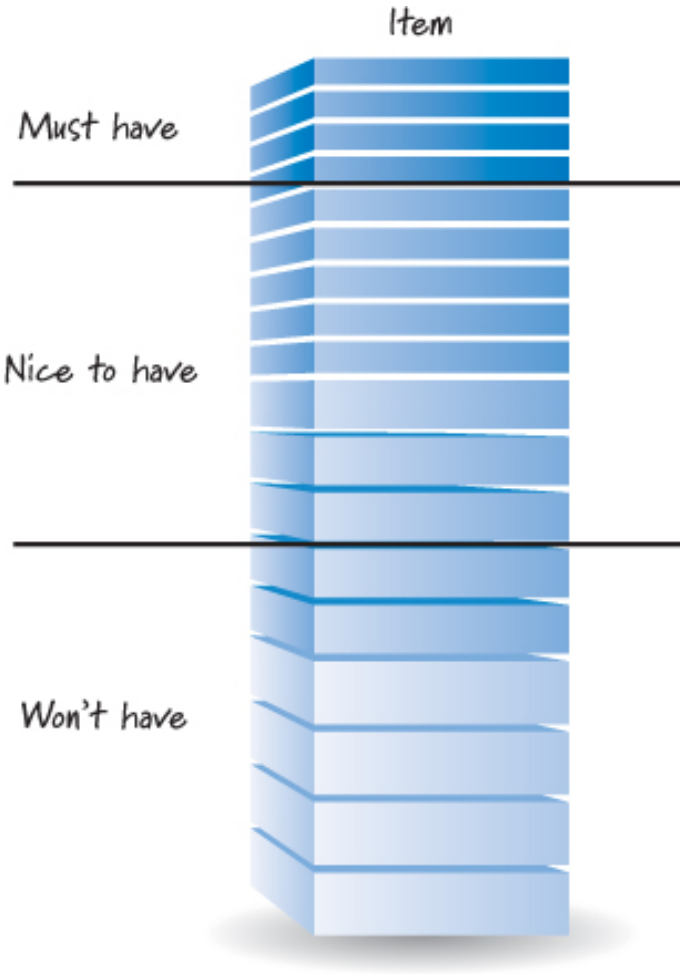


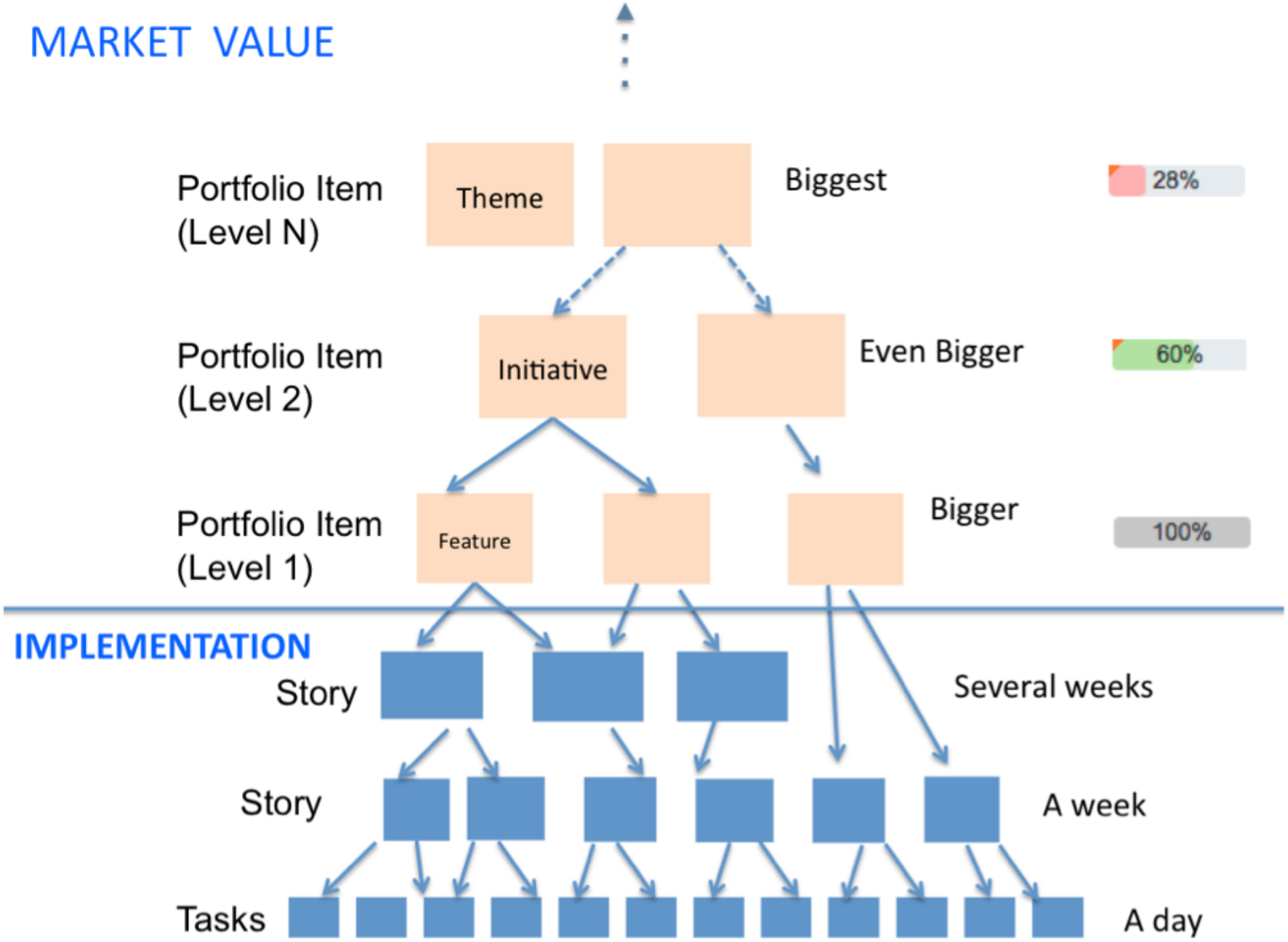
Product Backlog Pruning



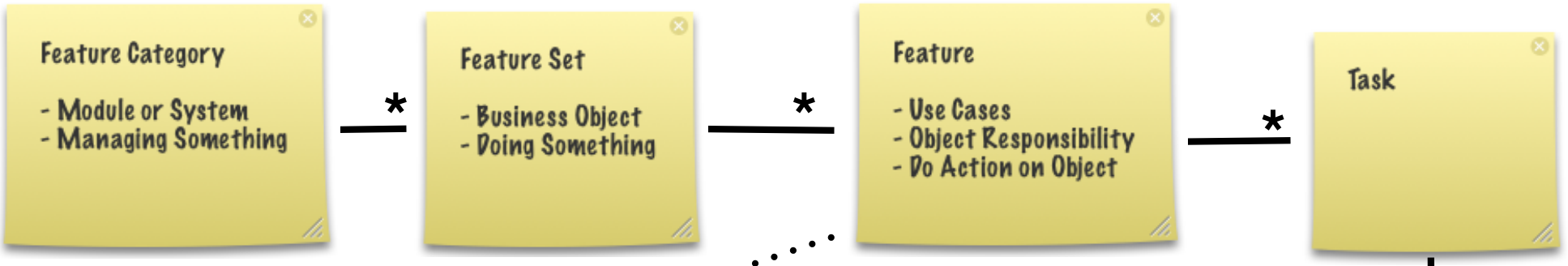
Product Backlog Release Scheduling






MARKET VALUE
Portfolio Item
(Level N)
Theme
Biggest
28%
Portfolio Item
(Level 2)
Initiative
Even Bigger
60%
Portfolio Item
(Level 1)
Feature
Bigger
100%
IMPLEMENTATION
Story
Several weeks
Story
A week
Tasks
A day

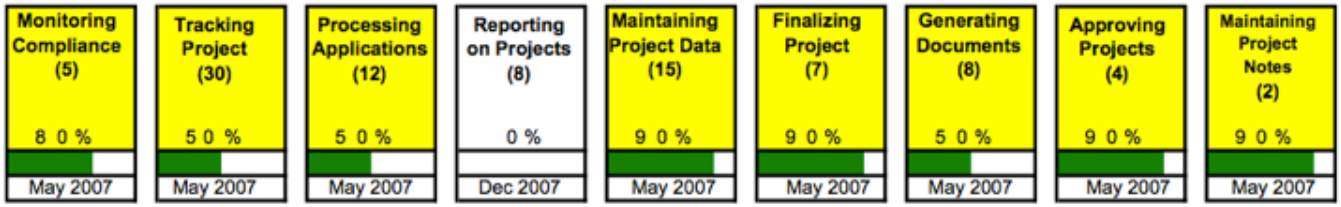
Feature Models (as backlog items)



Backlog Item	Task	Task Owner	Initial Estimate	Hours of Work Remaining on Each Day of the Sprint									
				Day 1	Day 2	Day 3	Day 4	Day 5	Day 6	Day 7	Day 8	Day 9	Day 10
Enable all users to place book in shopping cart	Design business logic	Sanjay	4	4	3	3	1	0					
	Design user interface	Jing	2	2	1	1	1	1					
	Implement back-end code	Philip	6	6	2	5	2	0					
	Implement front-end code	Tracy	4	4	3	2	2	2					
	Complete unit testing	Sarah	4	4	3	3	3	3					
	Complete regression testing	Sarah	2	2	3	3	3	3					
	Write documentation	Sam	3	3	4	2	0	0					
Upgrade transaction processing module (must be able to support 500 transactions /sec)	Merge DCP code and complete layer-level tests	Jing	5	5	2	2	1	0					
	Complete machine order for pRank	Jing	4	4	2	0	0	0					
	Change DCP and reader to use pRank http API	Tracy	3	3	3	2	2	2					
	Total		**50**	**50**	**48**	**44**	**43**	**34**					

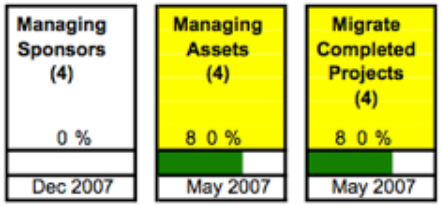
PROJECT MANAGEMENT (PM)

Monitoring Compliance (5)
Tracking Project (30)
Processing Applications (12)
Reporting on Projects (8)
Creating & Maintaining Project Data (15)
Finalizing Project (7)
Generating Documents (8)
Underwriting/Approving Projects (4)
Maintaining Project Notes (2)



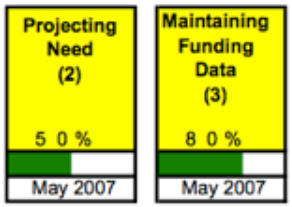
ASSET MANAGEMENT (AM)

Managing Information on Sponsors & Asset Entities (4)
Migrate Completed Projects to Asset Management (3)



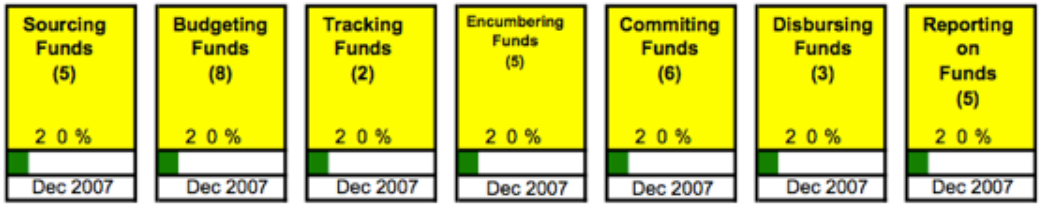
FINANCIAL MANAGEMENT (FI)

Projecting need for General Fund (2)
Collecting & Maintaining data for General Fund Projections (3)



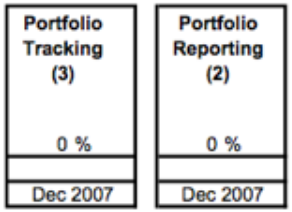
FUND MANAGEMENT (FM)

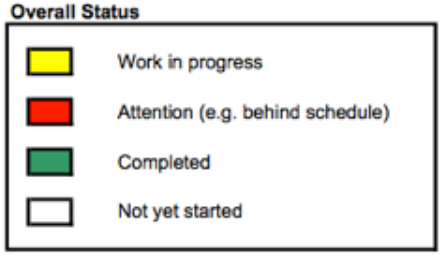
Sourcing Funds (5)
Budgeting Funds (8)
Tracking Funds (2)
Encumbering Funds (5)
Commiting Funds (6)
Disbursing Funds (3)
Reporting on Funds (5)



PORTFOLIO MANAGEMENT (PL)

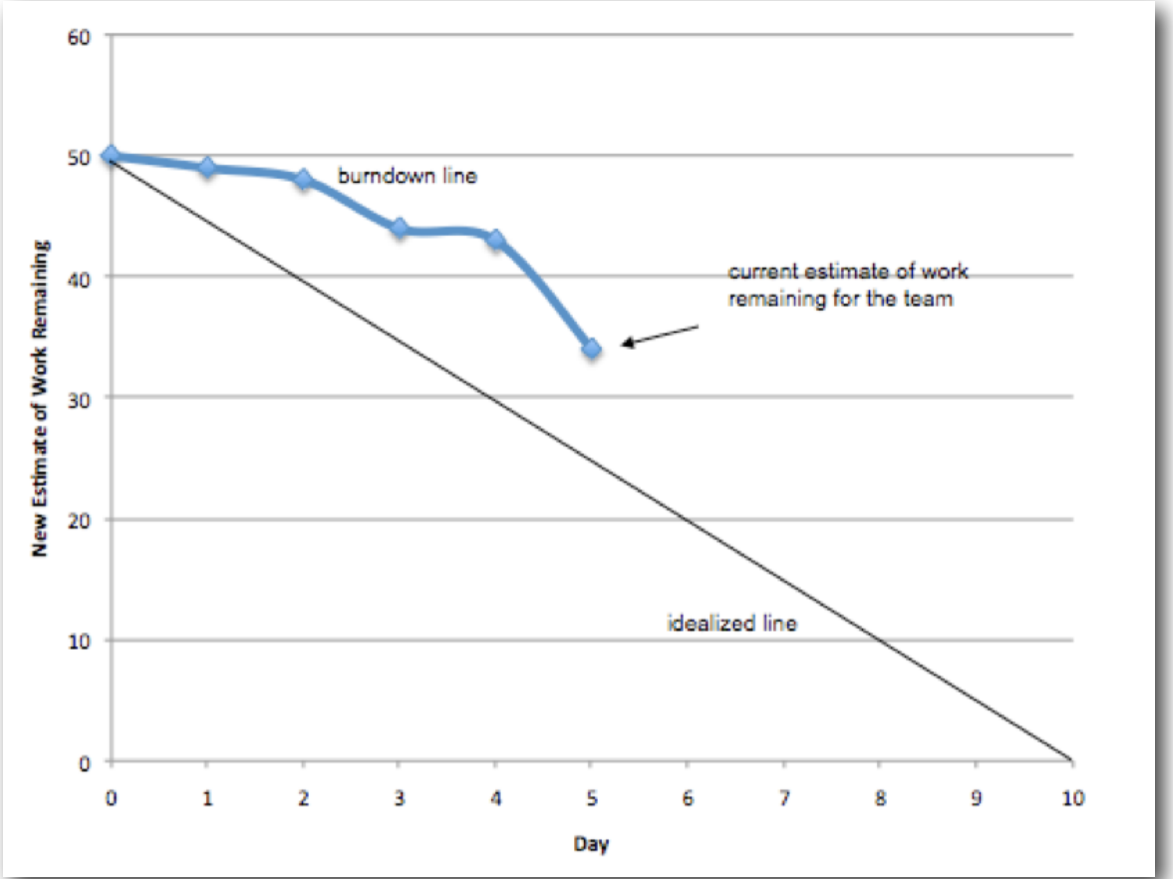
Tracking (3)
Reporting (2)



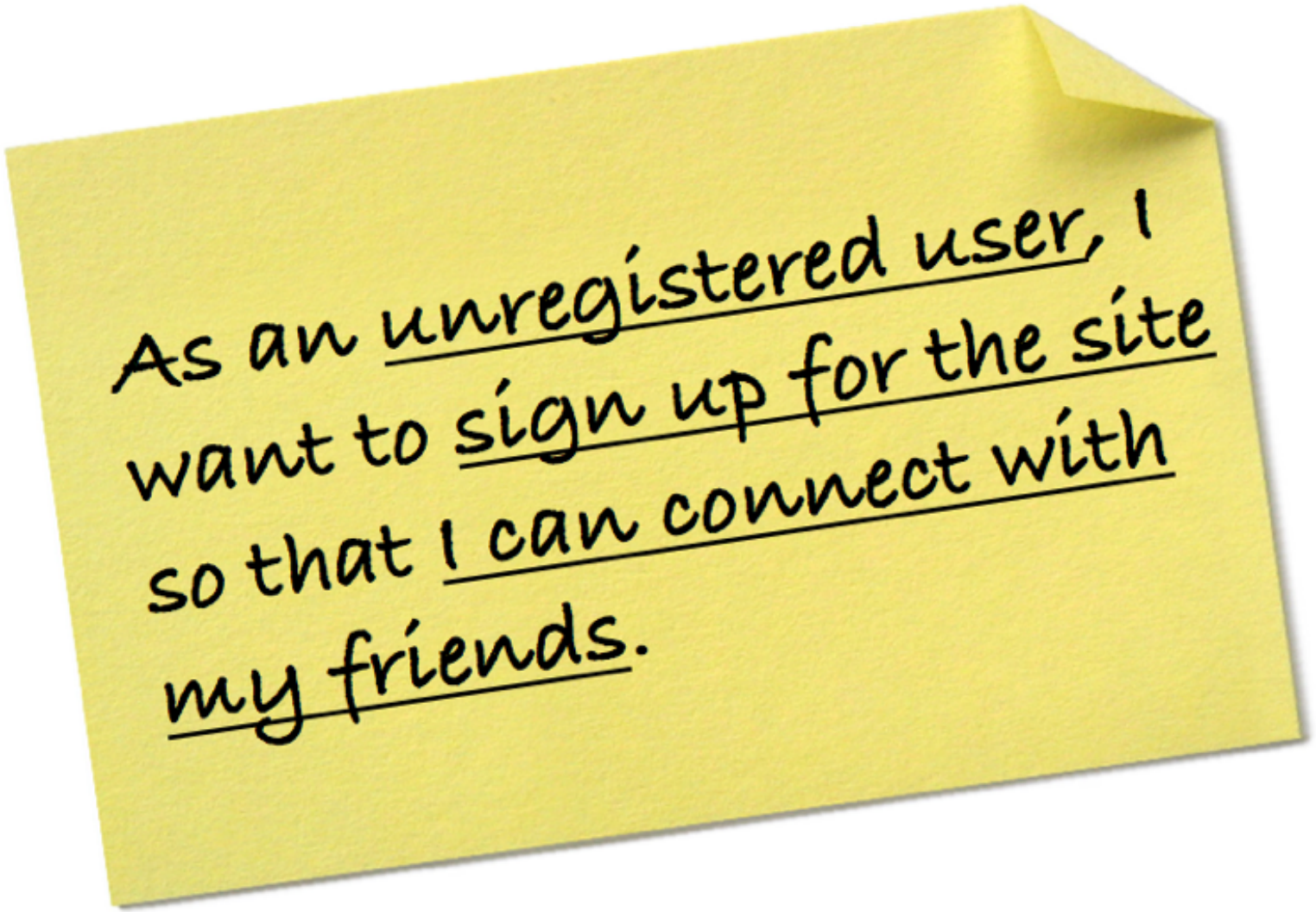


				New Estimates of Effort Remaining as of Day...					
Product Backlog Item	**Sprint Task**	**Volunteer**	**Initial Estimate of Effort**	**1**	**2**	**3**	**4**	**5**	**6**
As a buyer, I want to place a book in a shopping cart	modify database		5						
	create webpage (UI)		8						
	create webpage (Javascript logic)		13						
	write automated acceptance tests		13						
	update buyer help webpage		3						
	. . .								
Improve transaction processing performance	merge DCP code and complete layer-level tests		5						
	complete machine order for pRank		8						
	change DCP and reader to use pRank http API		13						

				New Estimates of Effort Remaining at end of Day...					
Product Backlog Item	**Sprint Task**	**Volunteer**	**Initial Estimate of Effort**	**1**	**2**	**3**	**4**	**5**	**6**
As a buyer, I want to place a book in a shopping cart	modify database	Sanjay	5	4	3	0	0	0	
	create webpage (UI)	Jing	3	3	3	2	0	0	
	create webpage (Javascript logic)	Tracy & Sam	2	2	2	2	1	0	
	write automated acceptance tests	Sarah	5	5	5	5	5	0	
	update buyer help webpage	Sanjay & Jing	3	3	3	3	3	0	
	. . .								
Improve transaction processing performance	merge DCP code and complete layer-level tests		5	5	5	5	5	5	
	complete machine order for pRank		3	3	8	8	8	8	
	change DCP and reader to use pRank http API		5	5	5	5	5	5	
	. . .	. . .	. . .						
		Total (person hours)	50	49	48	44	43	34	



USER STORY FORMAT


As an unregistered user, I
want to sign up for the site
so that I can connect with
my friends.

SPECIFYING WHO...

Grounds the team in a user-centered mindset

Provides empathy and understanding of target customers

Provides context for what is being built or designed

Allows for trade-offs and focus in implementation

SPECIFYING WHAT...

Solves a problem or takes advantage of an opportunity

Indicates what needs to be designed, built, and tested

Communicates the feature, functionality, or user goal

Provides the implementation as seen by the user

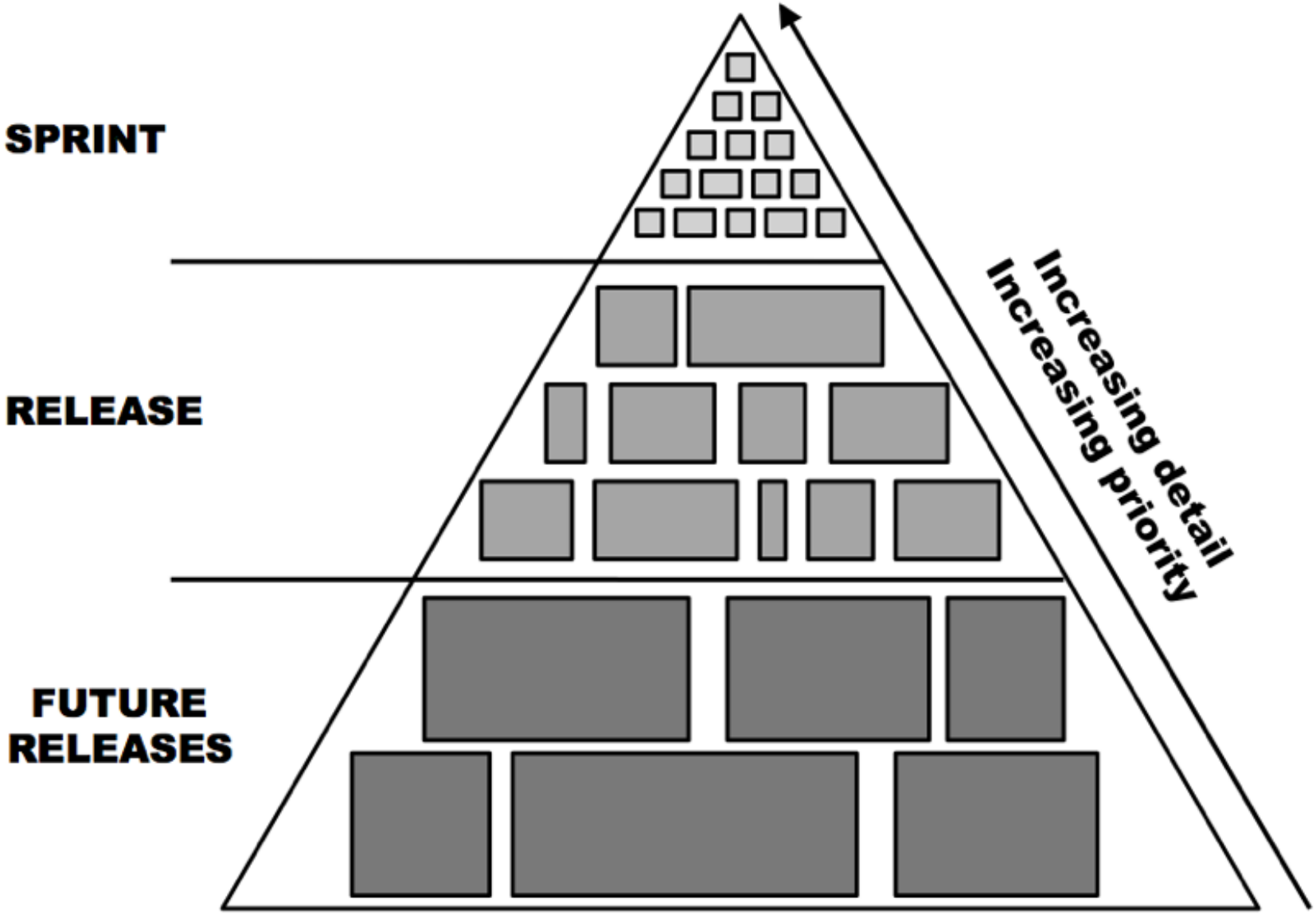
SPECIFYING WHY...

Indicates why this is important

Moves us towards our vision

Lets us understand what's in it for our users

Lets us understand how we are providing value


SPRINT
RELEASE
FUTURE RELEASES
Increasing detail
Increasing priority

SPLITTING STORIES

Research vs. Action
Spike vs. Implementation
Manual vs. Automated
Buy vs. Build
Batch vs. Individual
Single User vs. Multi-User
Simple vs. Complex
Static vs. Dynamic
Errors vs. Error Handling
Transient vs. Persistent
Low Fidelity vs. High Fidelity
Unreliable vs. Reliable
Duplicate vs. Reuse

Slower vs. Faster
Less vs. More
General vs. Specific
Show vs. Hide
Shiny vs. Sketchy
20% vs. 80%
Beginner vs. Expert
Someone vs. Everyone
Polish vs. Feedback
Solid vs. Throwaway
Certified vs. Non-certified
Accessible vs. Constrained
Evolved vs. Primitive

Revenue vs. Investment
and many more.

ACCEPTANCE CRITERIA...

Criteria for acceptance

A clarification of the story

A guide for automated tests

Documentation

A good tool for splitting and negotiation

Behavior Driven Development

Title
The story should have a clear, explicit title.

Narrative

A short, introductory section that specifies

- who (which business or project role) is the driver or primary stakeholder of the story (the actor who derives business benefit from the story)
- which effect the stakeholder wants the story to have
- what business value the stakeholder will derive from this effect

Acceptance criteria or scenarios

a description of each specific case of the narrative. Such a scenario has the following structure:

- It starts by specifying the initial condition that is assumed to be true at the beginning of the scenario. This may consist of a single clause, or several.
- It then states which event triggers the start of the scenario.
- Finally, it states the expected outcome, in one or more clauses.

```
Story: Returns go to stock

In order to keep track of stock
As a store owner
I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock.
When he returns the garment for a replacement in black,
Then I should have three blue garments in stock
And two black garments in stock
```

http://en.wikipedia.org/wiki/Behavior-driven_development

http://dannorth.net/introducing-bdd/

we are not so good at

estimating

we are good at

comparison

1, 2, 3, 5, 8, 13

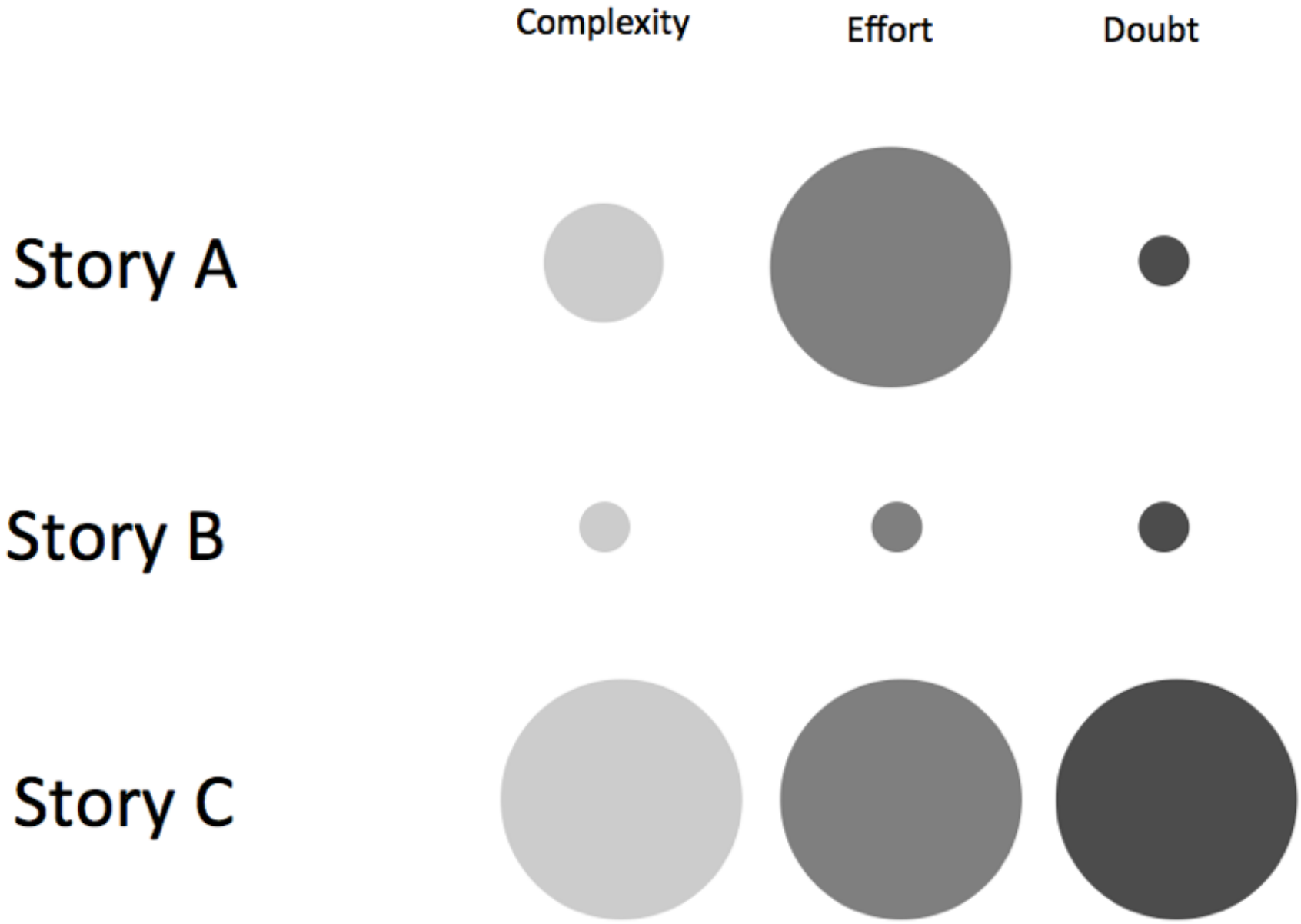
20, 40, 100

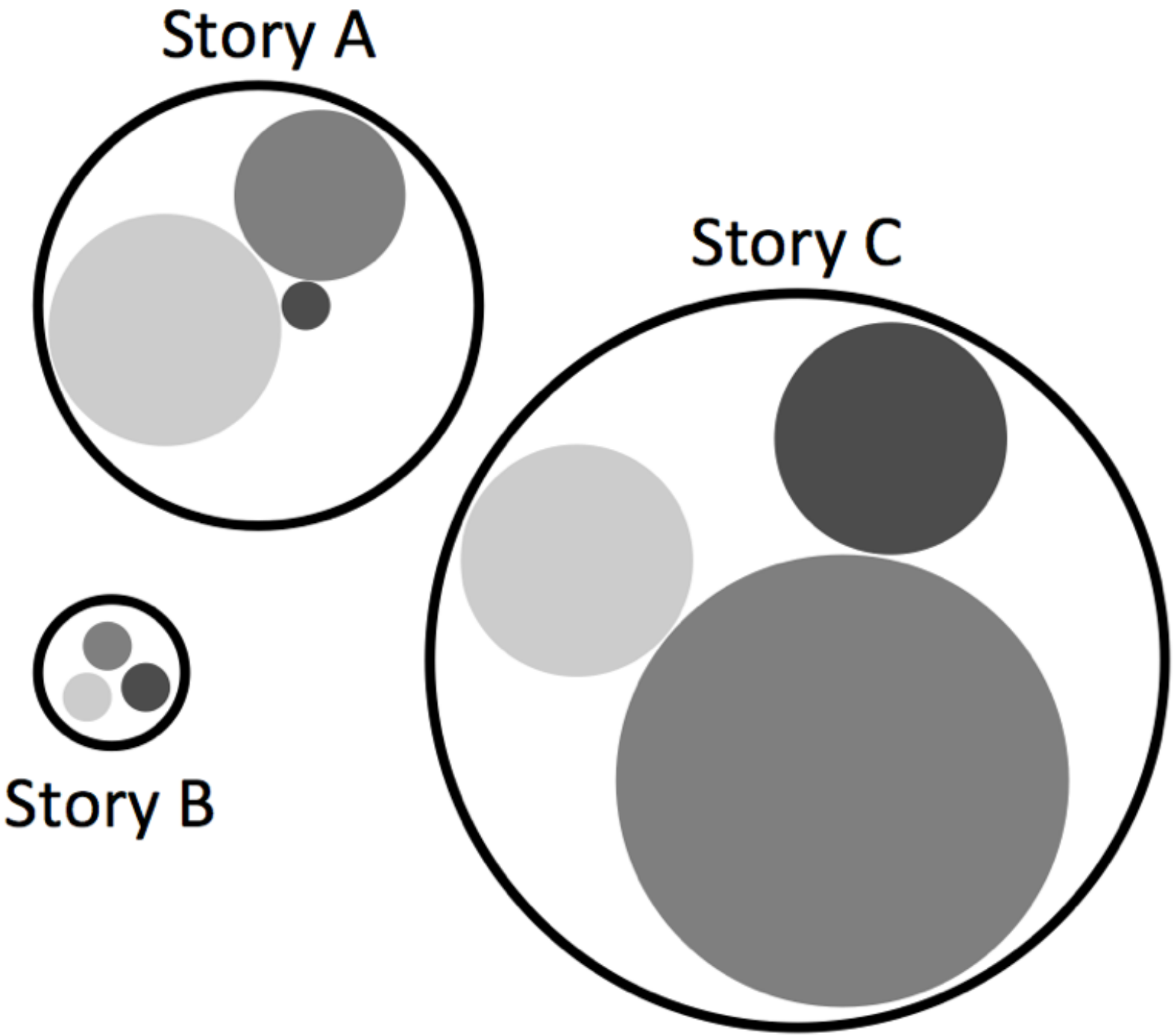
STORY **POINTS**

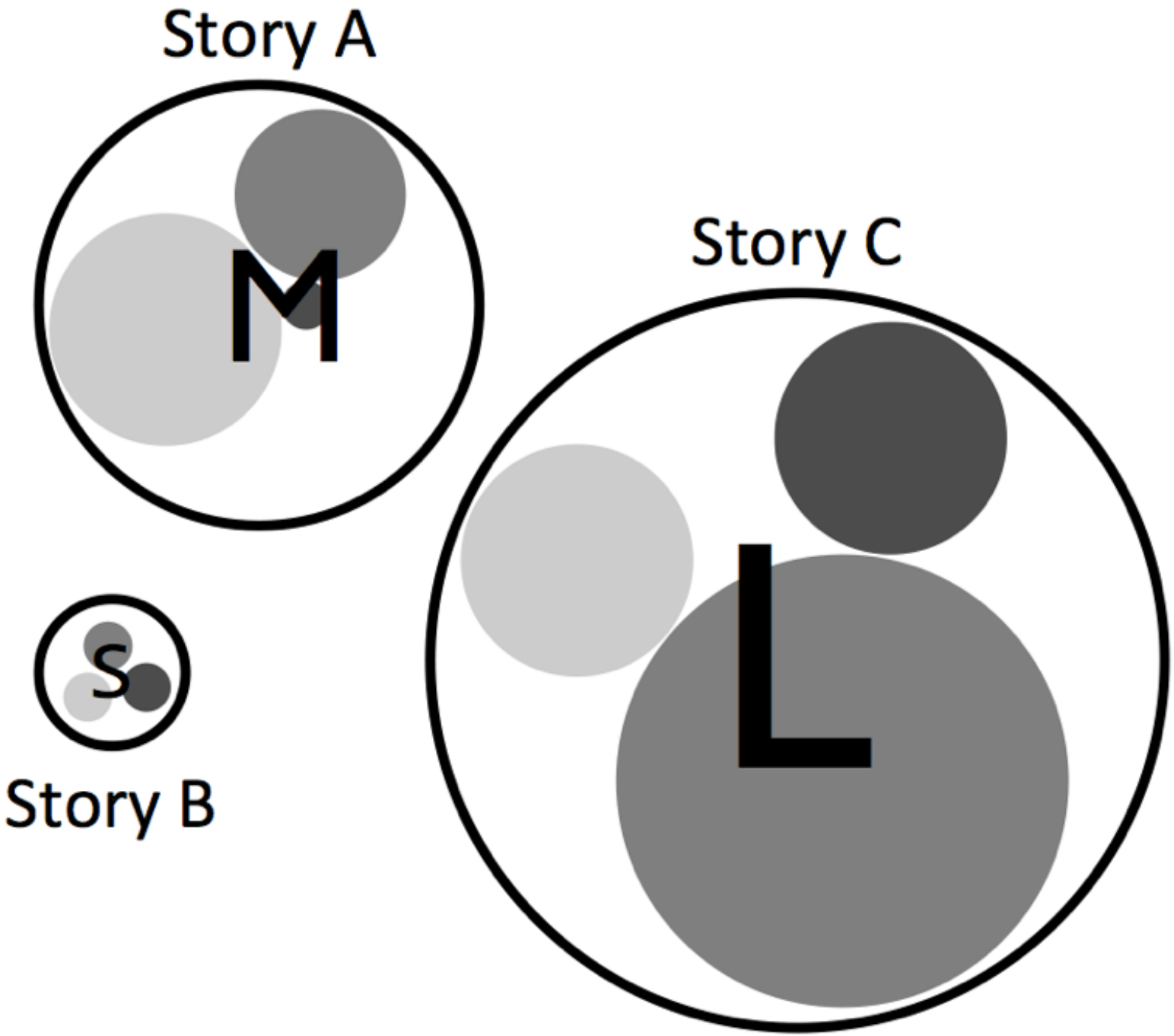
Complexity

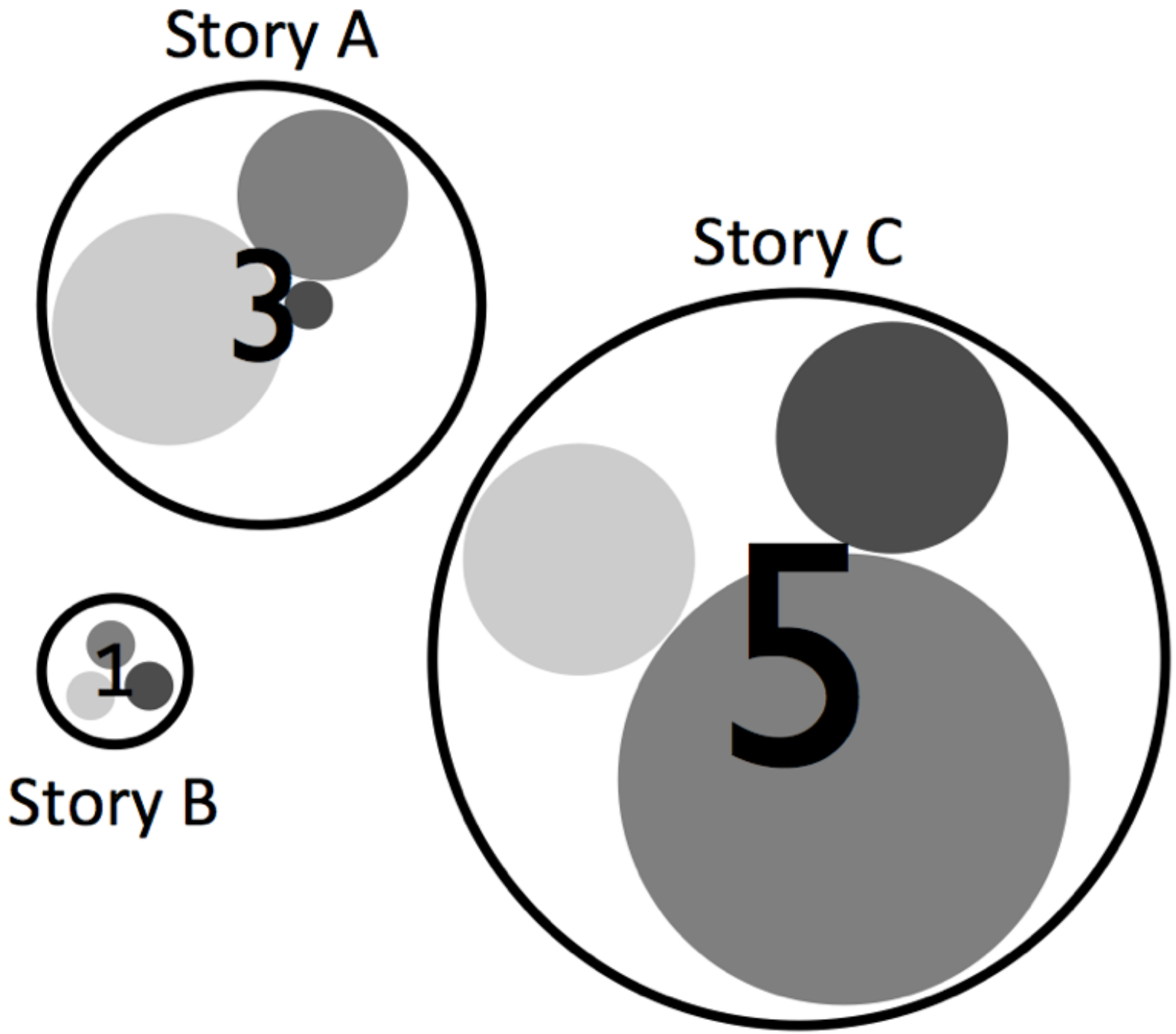
Effort

Doubt


Complexity
Effort
Doubt
Story A
Story B
Story C


Story A
Story C
Story B

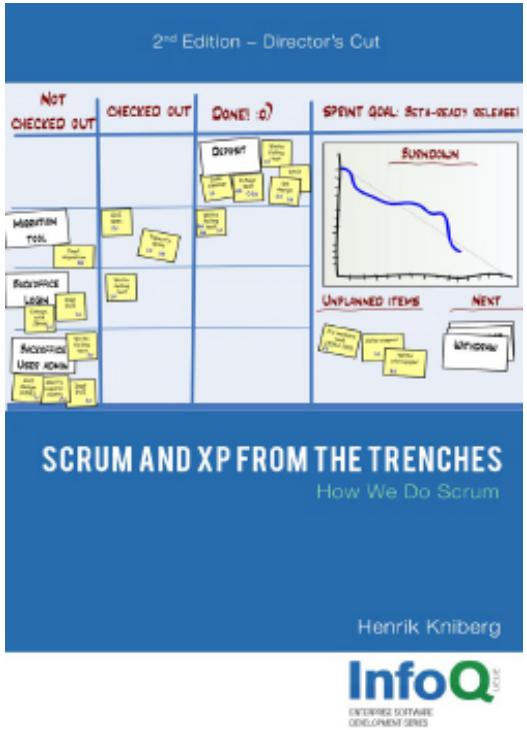

Story A
M
Story C
L
S
Story B


Story A
3
Story C
5
1
Story B

Resources

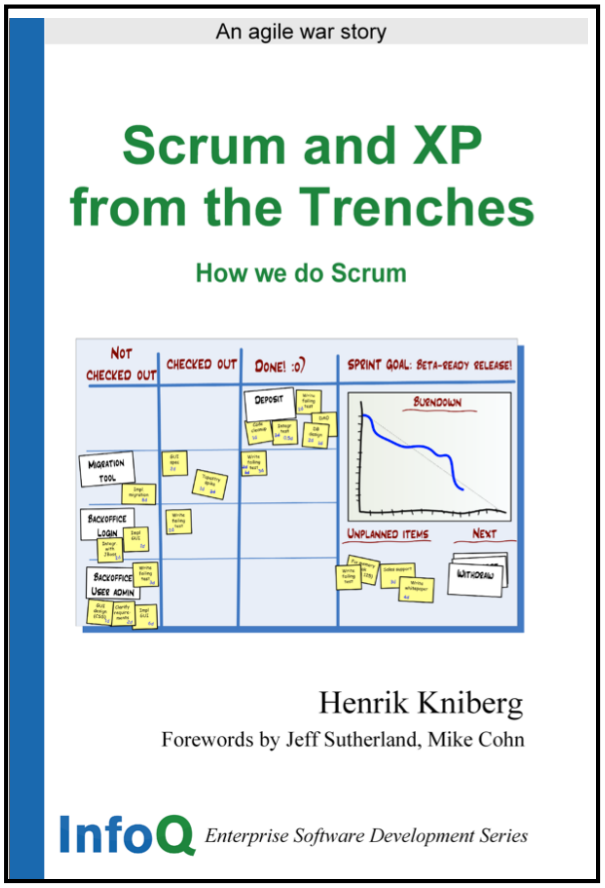
Readings

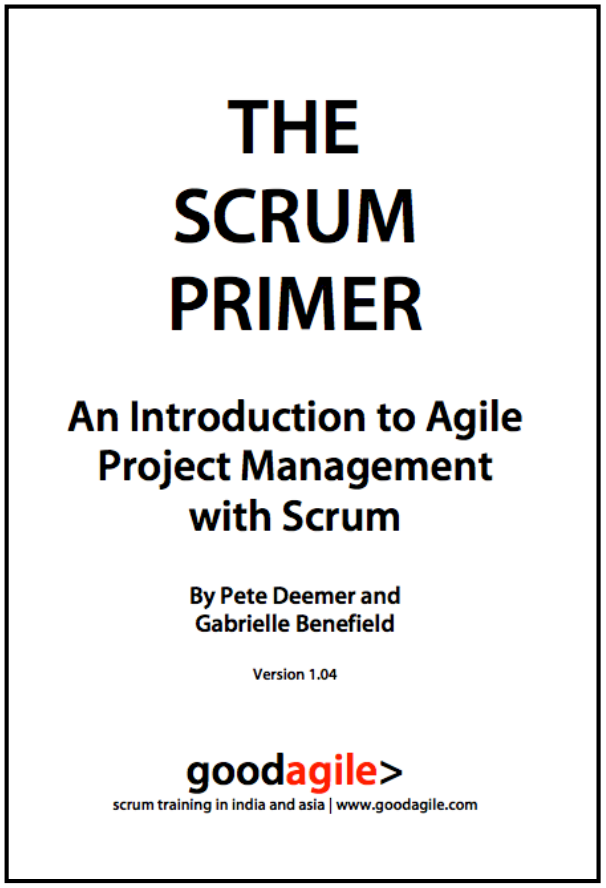
http://www.infoq.com/minibooks/scrum-xp-from-the-trenches-2



http://www.infoq.com/minibooks/kanban-scrum-minibook

How to do Scrum



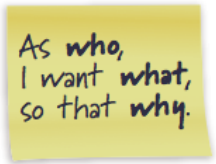


RALLY SOFTWARE

User Stories Guide

What are they?

User stories are value-focused units of delivery that are typically used in Agile projects. Written from the customer or stakeholder's perspective, user stories share what is needed and why.



Who?

The "who" in a user story is typically someone with a particular role or title, or it could be from the perspective of a *persona*: a fictitious, sample user's behaviors and needs, spelled out in detail. When choosing the "who", avoid being too general (e.g. "As a user...") which defeats the purpose of pinpointing the value for your target audience. Write all stories with the "who" included, even those stories that are very technical.

What?

The "what" in a user story specifies the need, feature, or functionality that is desired by the "who". This is what the team will build into the software or service.

Why?

The "why" in a user story specifies the value, keeping the needs of users and customers front and center. Agile's focus is on value-based delivery, so it is extremely important to understand why the story should be delivered. If you can't write a user story with the "why" included, then maybe it's not needed.

Template

The user story template is designed to help Product Owners and others tell stories with a clear "who", "what" and "why".

As a registered user, I want to reset my password, so that I can get back into the site if I forget my password.

As an unregistered user, I can sign up for the site, so that I'm able to have a personalized experience.

As Tom, I want to only see updates from close friends, so that I can view relevant updates during my time online.

Another Template

Some stories benefit from more than one "who". In those cases, you might want to include multiple who-why pairs.

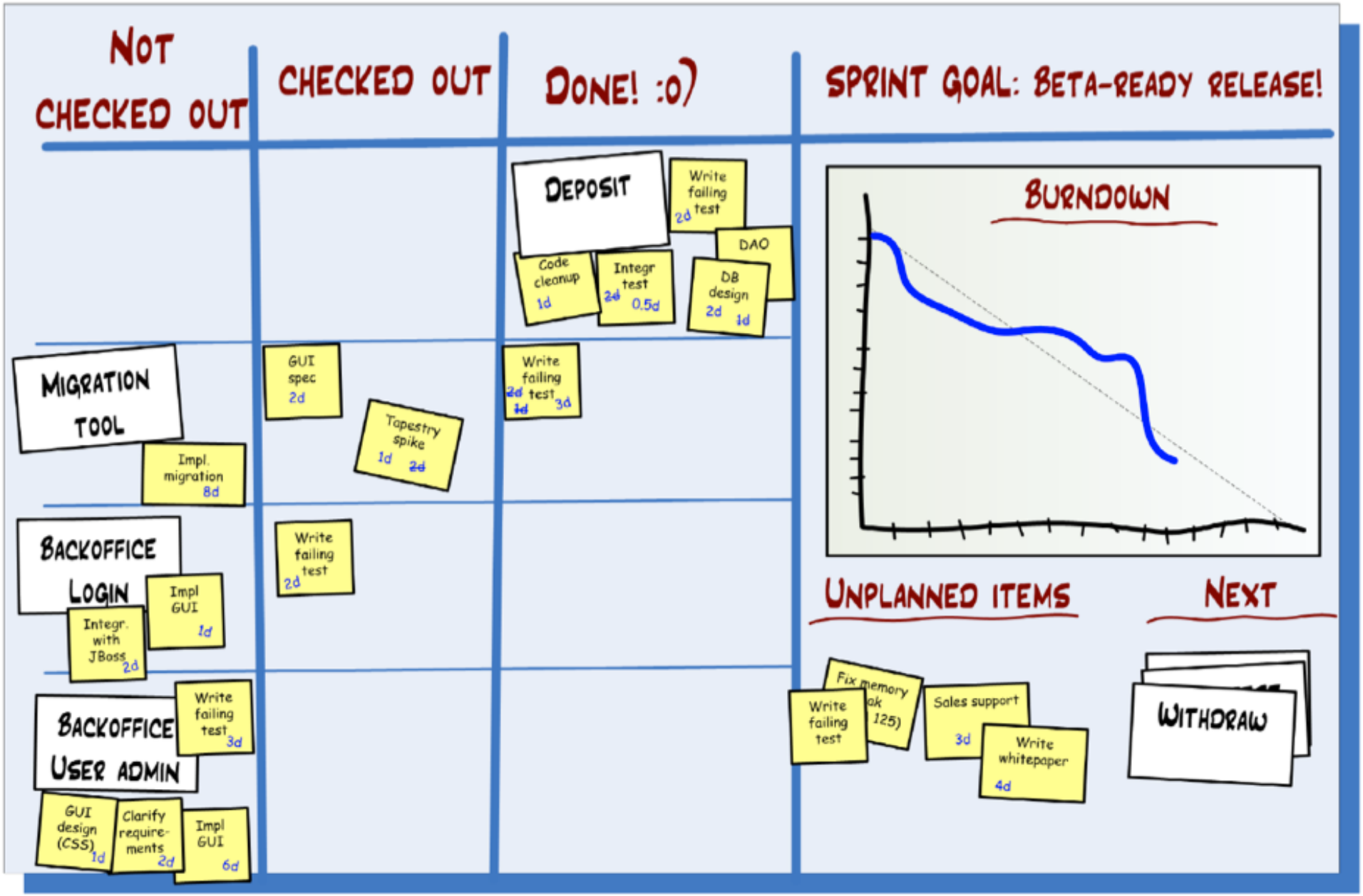
what: Track history of which products registered shoppers have viewed.

who: Registered Shoppers
why: So I can go back and find again (and buy) what I've previously researched

who: Marketing
why: So we can serve ads that are relevant to the interests of individual registered shoppers.

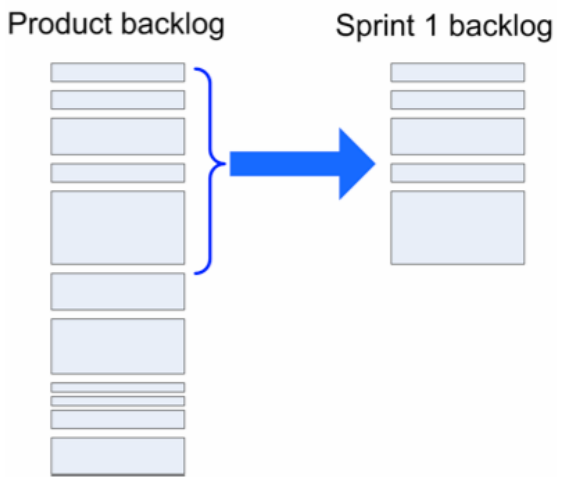
Continued

Tracking

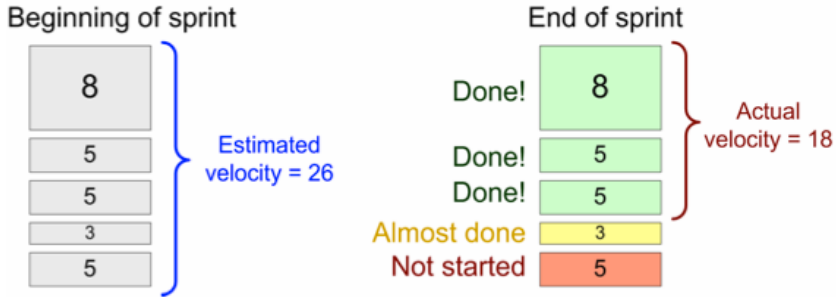

NOT CHECKED OUT
CHECKED OUT
DONE! :o)
SPRINT GOAL: BETA-READY RELEASE!
DEPOSIT
Write failing test 2d
DAO
Code cleanup 1d
Integr test 2d 0.5d
DB design 2d 1d
BURNDOWN
MIGRATION TOOL
GUI spec 2d
Tapestry spike 1d 2d
Write failing test 2d 1d 3d
Impl. migration 8d
BACKOFFICE LOGIN
Write failing test 2d
Impl GUI 1d
Integr. with JBoss 2d
BACKOFFICE USER ADMIN
Write failing test 3d
GUI design (CSS) 1d
Clarify require-ments 2d
Impl GUI 6d
UNPLANNED ITEMS
NEXT
Write failing test
Fix memory leak (125)
Sales support 3d
Write whitepaper 4d
WITHDRAW

Product Backlog

PRODUCT BACKLOG (example)					
ID	**Name**	**Imp**	**Est**	**How to demo**	**Notes**
1	Deposit	30	5	Log in, open deposit page, deposit €10, go to my balance page and check that it has increased by €10.	Need a UML sequence diagram. No need to worry about encryption for now.
2	See your own transaction history	10	8	Log in, click on "transactions". Do a deposit. Go back to transactions, check that the new deposit shows up.	Use paging to avoid large DB queries. Design similar to view users page.

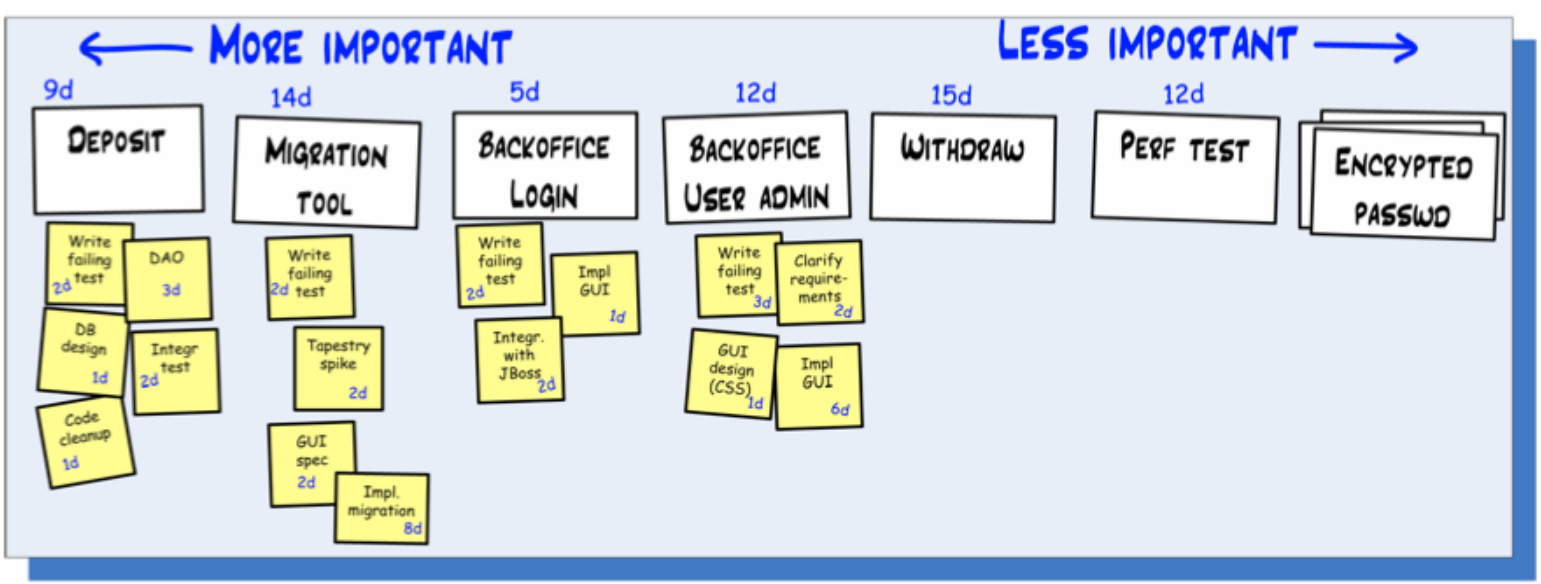


- **Importance** – the product owner's importance rating for this story. For example 10. Or 150. High = more important.
  - I tend to avoid the term "priority" since priority 1 is typically considered the "highest" priority, which gets ugly if you later on decide that something else is even *more* important. What priority rating should *that* get? Priority 0? Priority -1?
- **Initial estimate** – the team's initial assessment of how much work is needed to implement this story compared to other stories. The unit is story points and usually corresponds roughly to "ideal man-days".



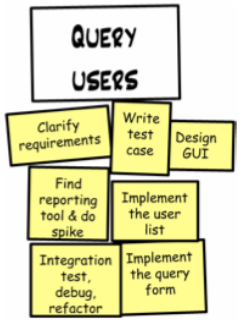
Note that the actual velocity is based on the *initial* estimates of each story. Any updates to the story time estimates done during the sprint are ignored.

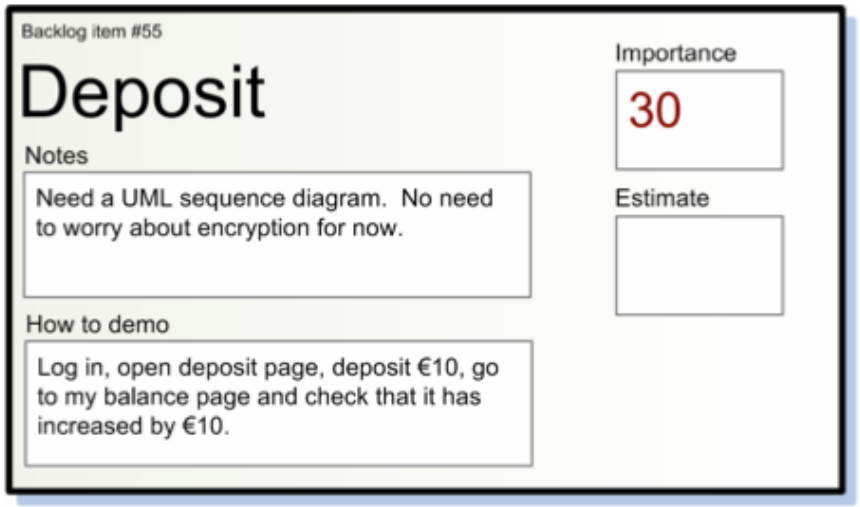
Sprint Planning



Example of breaking down a story into smaller stories:

Example of breaking down a story into tasks:





Starting a Sprint

Dreamteam, Sprint #1

Sprint Goal

Working Prototype to Demo

Sprint Backlog (estimates in parenthesis)

- Deposit (3)
- Migration Tool (8)
- Backoffice Login (5)
- Backoffice User Admin (5)

Estimated Velocity: 21

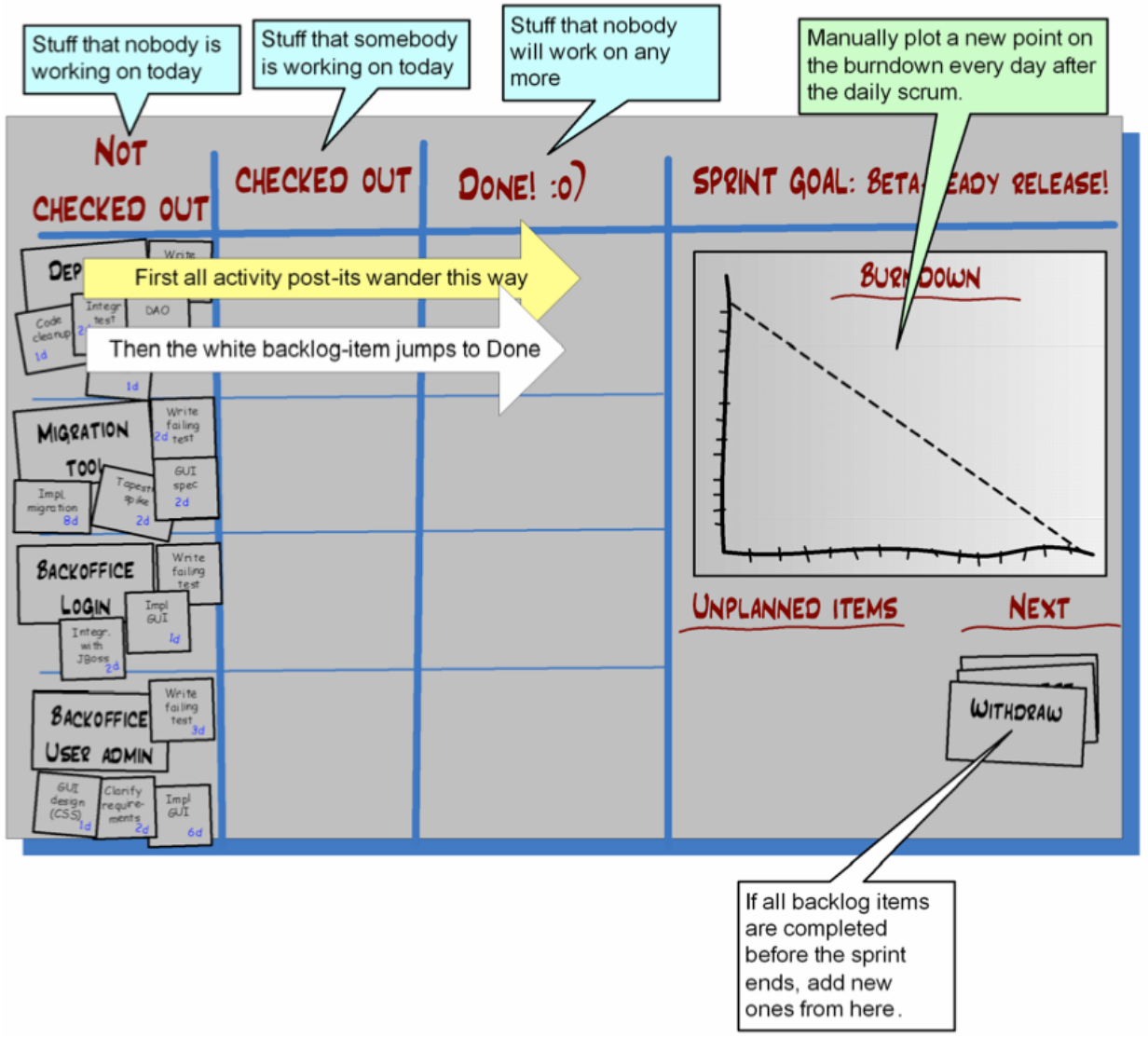
Schedule

- Sprint period: 2006-11-06 to 2006-11-24
- Daily Scrum: 9:30 – 9:45 in the team room
- Sprint Demo: 2006-11-24, 13:00, in the cafe

Team

- Jim
- Erica (scrum master)
- Tom (75%)
- Eva
- John

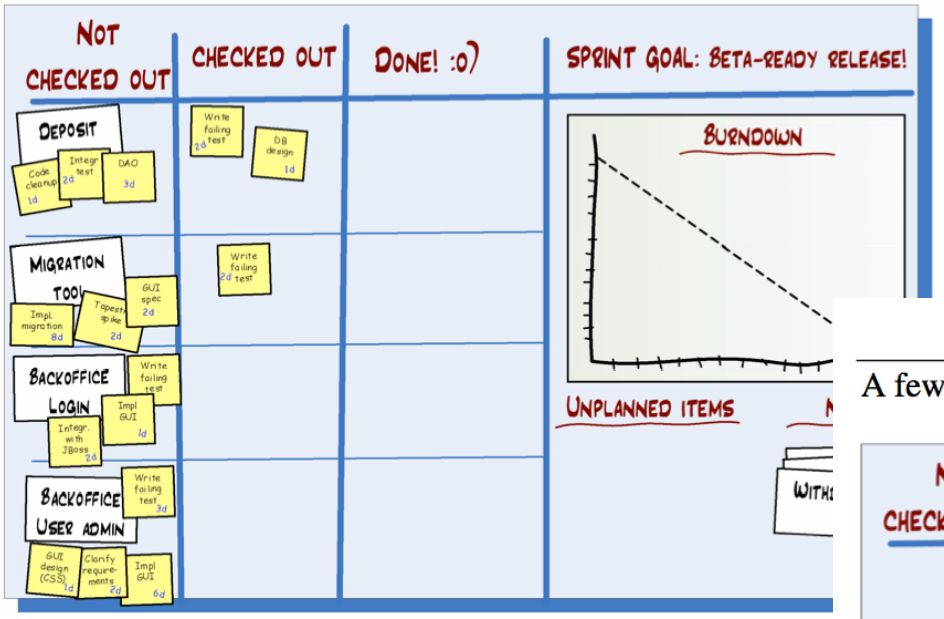
The Task Board



Showing Progress

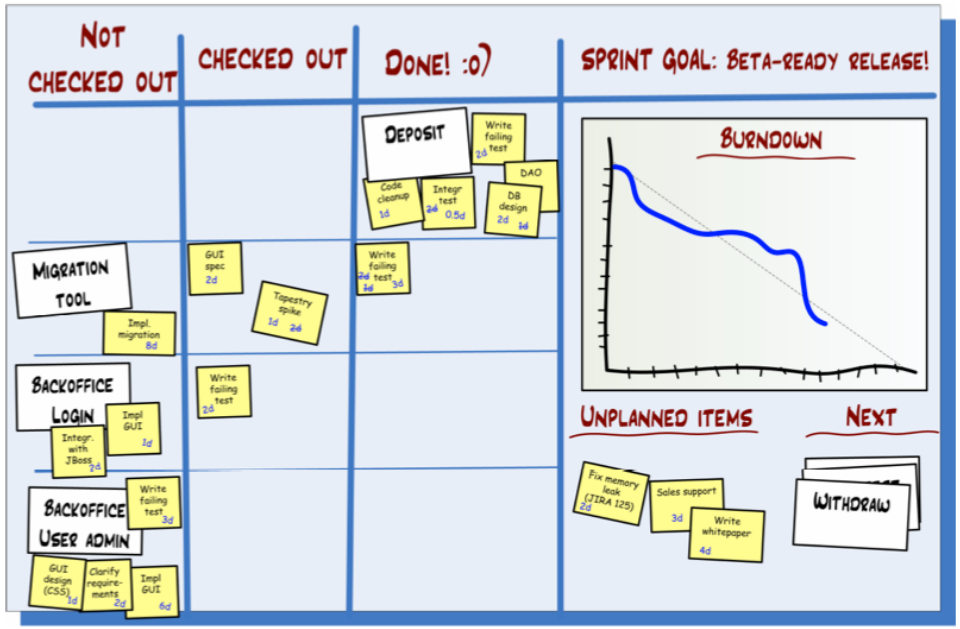
Example 1 – after the first daily scrum

After the first daily scrum, the taskboard might look like this:



Example 2 – after a few more days

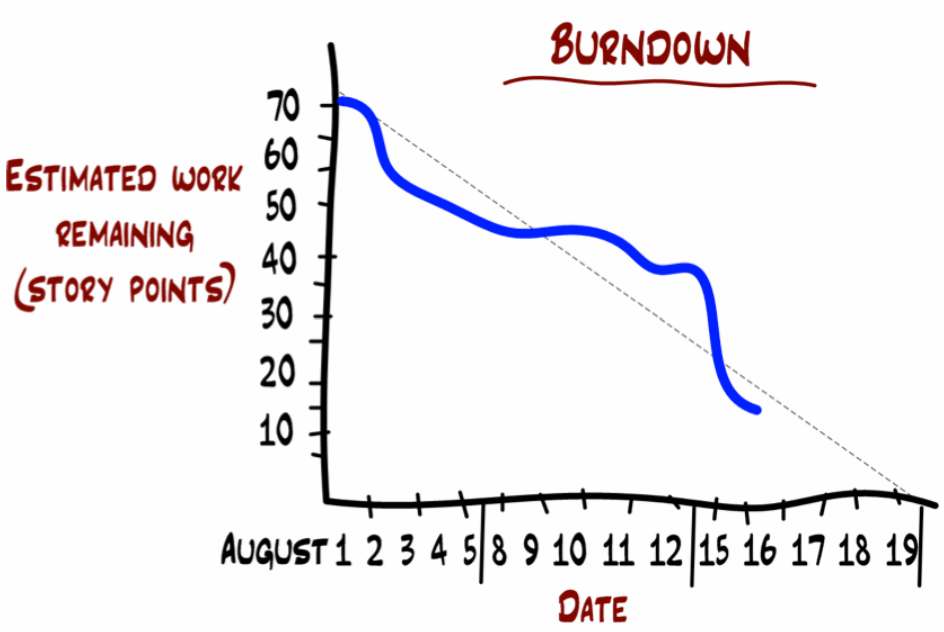
A few days later the taskboard might look something like this:



The Burndown

How the burndown chart works

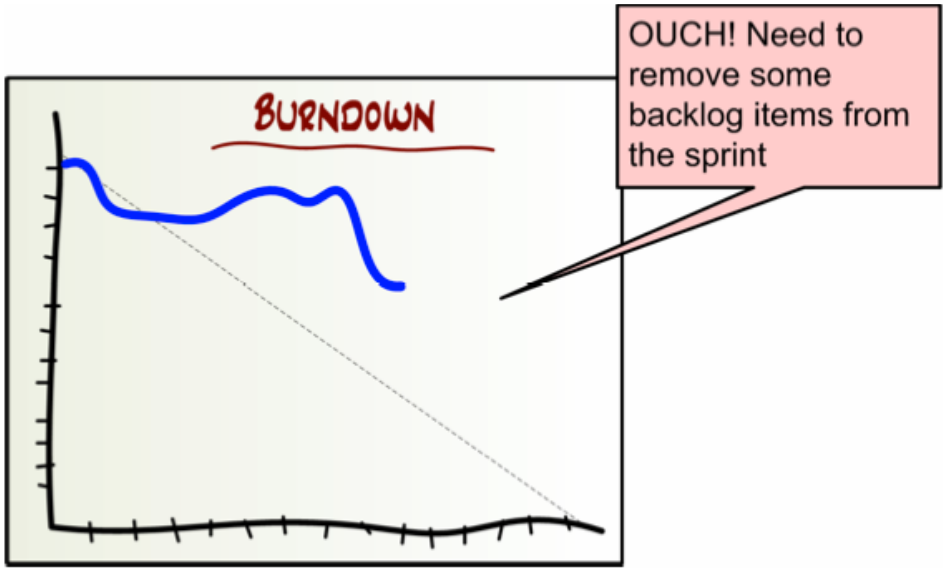
Let's zoom in on the burndown chart:

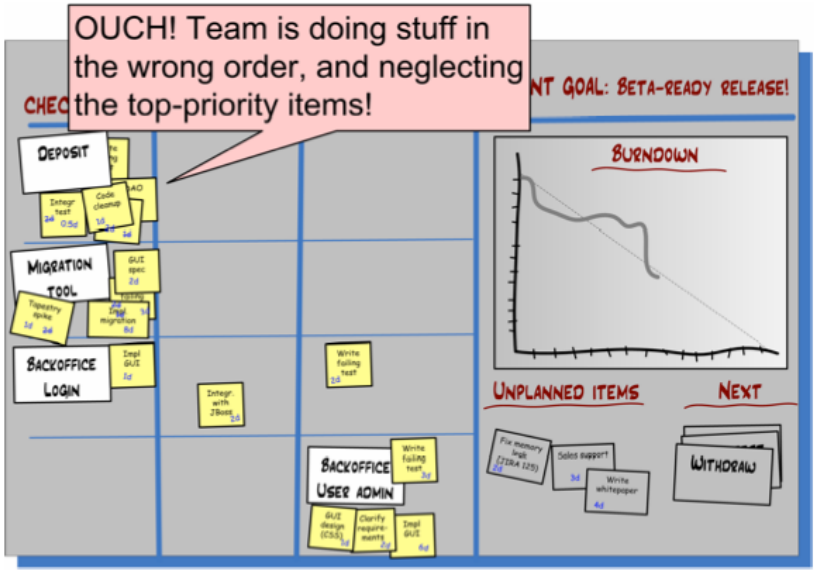


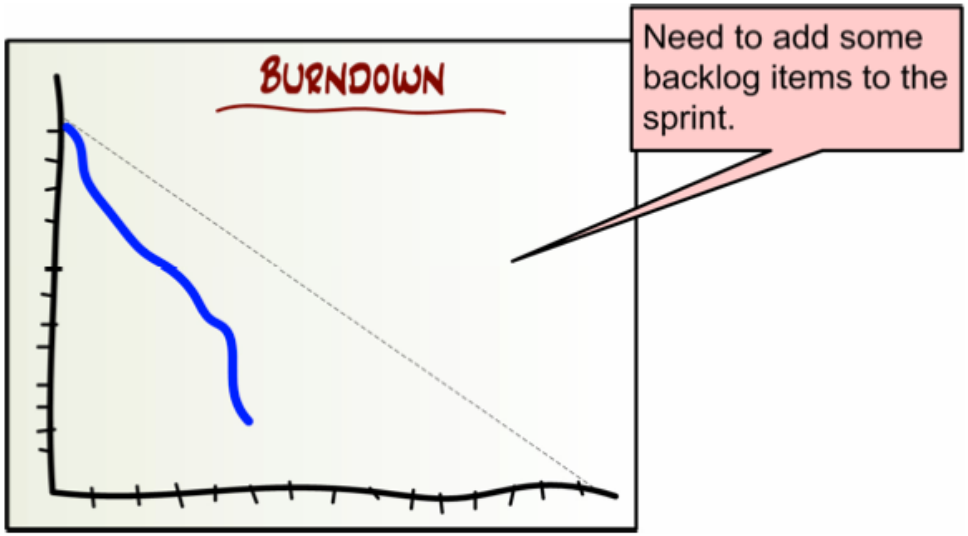
This chart shows that:

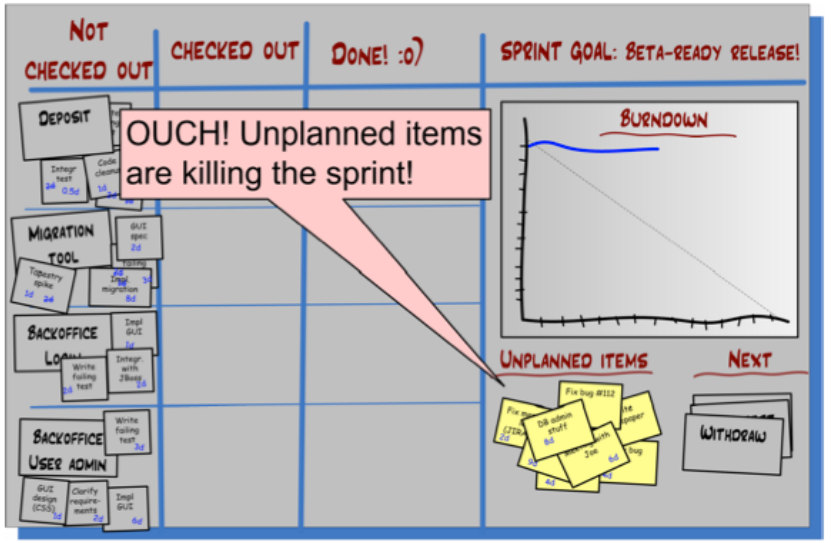
- On the first day of the sprint, august 1, the team estimated that there is approximately 70 story points of work left to do. This was in effect the *estimated velocity* of the whole sprint.
- On august 16 the team estimates that there is approximately 15 story points of work left to do. The dashed trend line shows that they are approximately on track, i.e. at this pace they will complete everything by the end of the sprint.

Trouble?


BURNDOWN
OUCH! Need to remove some backlog items from the sprint


OUCH! Team is doing stuff in the wrong order, and neglecting the top-priority items!
DEPOSIT
MIGRATION TOOL
BACKOFFICE LOGIN
BACKOFFICE USER ADMIN
BURNDOWN
UNPLANNED ITEMS
NEXT
WITHDRAW


BURNDOWN
Need to add some backlog items to the sprint.


NOT CHECKED OUT
CHECKED OUT
DONE! :o)
SPRINT GOAL: BETA-READY RELEASE!
OUCH! Unplanned items are killing the sprint!
DEPOSIT
MIGRATION TOOL
BACKOFFICE USER ADMIN
BURNDOWN
UNPLANNED ITEMS
NEXT
WITHDRAW

Daily Scrum + Burndown Chart

Team Name, Sprint #1

Team Member Name

John Smith

What I did since the last daily scrum:

- Draw UML Class Diagram (done)
- Draw Sequence Diagram (not done, est. 2 more hours)

What I plan to do today:

- Draw Sequence Diagram
- Write Unit Tests

What blockers I have:

- I am waiting on the interface definition for my FooBar class. We need to define this ASAP.

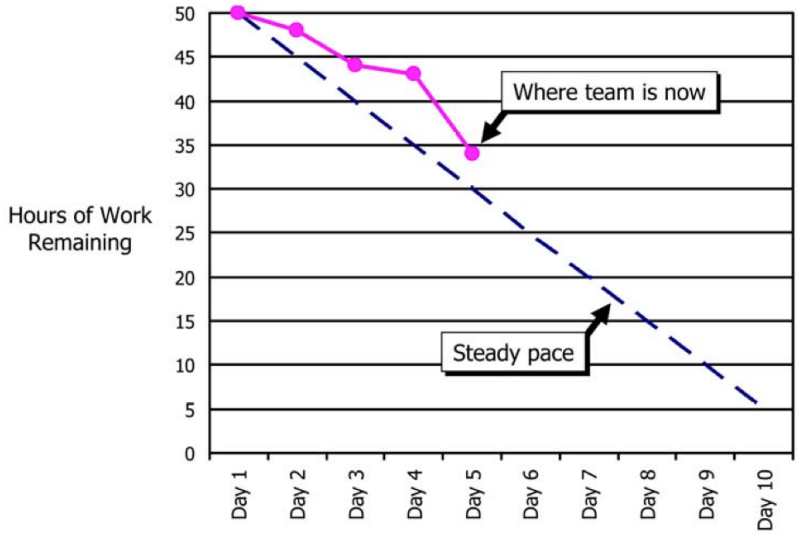


Figure 6. Burndown Chart

				Hours of Work Remaining on Each Day of the Sprint									
Backlog Item	**Task**	**Task Owner**	**Initial Estimate**	**Day 1**	**Day 2**	**Day 3**	**Day 4**	**Day 5**	**Day 6**	**Day 7**	**Day 8**	**Day 9**	**Day 10**
Enable all users to place book in shopping cart	Design business logic	Sanjay	4	4	3	3	1	0					
	Design user interface	Jing	2	2	1	1	1	1					
	Implement back-end code	Philip	6	6	2	5	2	0					
	Implement front-end code	Tracy	4	4	3	2	2	2					
	Complete unit testing	Sarah	4	4	3	3	3	3					
	Complete regression testing	Sarah	2	2	3	3	3	3					
	Write documentation	Sam	3	3	4	2	0	0					
Upgrade transaction processing module (must be able to support 500 transactions /sec)	Merge DCP code and complete layer-level tests	Jing	5	5	2	2	1	0					
	Complete machine order for pRank	Jing	4	4	2	0	0	0					
	Change DCP and reader to use pRank http API	Tracy	3	3	3	2	2	2					
		Total	**50**	**50**	**48**	**44**	**43**	**34**					

Figure 5. Daily Updates of Work Remaining on the Sprint Backlog