A Lean and Scalable Requirements Model for Agile Enterprises

By Dean Leffingwell
Lean & Kanban Conference
May 6-8, 2009
Miami, Florida

About Dean Leffingwell

Author
Scaling Software Agility
Managing Software Requirements

Coach
Agile Enterprise Coach
Executive Mentor
BMC Agile Transformation
Cofounder/Advisor
Ping Identity, Roving Planet, Rally Software

Executive
Founder and CEO
ProQuo, Inc., Internet Identity
Senior VP
Rational Software
Responsible for Rational Unified Process (RUP) & Support of UML
Founder/CEO
Requisite, Inc.
Makers of RequisitePro

© 2008-2009 Leffingwell, LLC.
The Agile Team in The Enterprise

There can be a large number of teams in the enterprise

“pods” of 5-10 teams building a feature, component, or subsystem is not unusual

Some product lines require 30-40-50 teams to build

However, the structure of each team is largely the same

---

**THE MODEL FOR AGILE TEAMS**

© 2008-2009 Leffingwell, LLC.
The structure of each team is largely the same

Their work is based on the user story

"As a Gmail user, I can select and highlight a conversation for further action"
“Stories” drive iterations

Stories are implemented by tasks

**Story 51 – As a user, I can highlight a conversation for further action**

- Task 51.1 – Define acceptance test: Juha, Dean, 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

Tasks are used by the team to estimate and coordinate the effort required to deliver a story into the baseline
Tasks are either

- The only reasonable way to:
  - define interdependencies
  - take individual responsibility for work
  - Break the story down into small enough chunks (hours) to be truly estimable
  - track story completion, task by task

- OR
  - Valueless and non-lean administrative overhead

Stories are maintained in the teams backlog

There is only one backlog for the team

All work comes from the backlog

If isn't a user story (defect, etc) it still goes in the backlog
Stories: Card, Conversation, Confirmation

- *Card, Conversation and Confirmation* describe the three elements of a User Story:
- Card represents the 2-3 sentences used to describe the intent of the story.
- Conversation represents fleshing out the details of the intent of the card in a conversation with the customer or product owner.
- Confirmation represents how the team will confirm the story has been implemented correctly.

*Ron Jeffries (co-creator of XP)*

Confirmation - all code is tested code

Stories can’t be considered “done” until they pass an acceptance test
Stories must be integrated into the baseline to be accepted
The Product owner is usually responsible for accepting the story

© 2008-2009 Leffingwell, LLC.
A test and quality-centric approach

Teams perform unit testing and functional testing for every story.

The details of the story go into the functional test, where they are the persistent representation of system behavior.

Stories are temporal (not maintained after implementation).

The model for agile teams is lean and scalable

<table>
<thead>
<tr>
<th>It is Lean</th>
<th>It is Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simplest possible agile artifact types</td>
<td>No limit to the number of teams</td>
</tr>
<tr>
<td>Just-in-time story development eliminates requirements specification and inventory</td>
<td>No limit to the number of stories</td>
</tr>
<tr>
<td>All code is tested code – no untested code inventory either</td>
<td>If all code is tested code, the system will have inherent quality too</td>
</tr>
<tr>
<td>Simple backlog construct facilitates a pull/kanban system</td>
<td>Separation of backlogs simplifies administration and tooling</td>
</tr>
</tbody>
</table>

© 2008-2009 Leffingwell, LLC.
Scaling agile up requires rethinking

- Assume a program requires
  - 200 practitioners, (25 agile teams) to deliver a product
  - The enterprise delivers software every 90 days in five, two week iterations.
  - Each team averages 15 stories per iteration.
  - Number of stories that must be elaborated and delivered to achieve the release objective = 25*5*15= 1,875!

- How is an enterprise supposed to reason about things?
  - What is this new product going to actually do for our users?
  - If we have 900 stories complete, 50% done, what do we actually have working? How would we describe 900 things?
  - How will we plan a release than contains 1,875 things?

- And, what if it took 500 people?
And further

- And, even if I know 100 things that “as a <role> I can <activity> so that <business value>”, can do

what Features does the system offer to its user and what benefits does it provide?

<table>
<thead>
<tr>
<th>Feature</th>
<th>Benefit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stars for conversations</td>
<td>Highlight conversations of special interests</td>
</tr>
<tr>
<td>Colored label categorization</td>
<td>Easy eye discrimination of different types of stories (folder like metaphor)</td>
</tr>
<tr>
<td>Smart phone client application</td>
<td>Faster and more facile use for phone users – ease adoption</td>
</tr>
</tbody>
</table>

So we need two levels of planning

Features

Product & Release Cycle

Release Vision

Release Planning

Release Scope and Boundaries

Drives

Stories

Iteration Cycle

Iteration Planning

Review & Adapt

Develop & Test

Feedback - Adjust
Which creates an iteration and release pattern

So we need to extend the information model

Features are another kind of Backlog Item

Introduce Gmail “Labels” as a “folder-like” conversation-organizing metaphor.

Or:

As a modestly skilled user, I can assign more than one colored label to a conversation so that I can see a conversation from multiple perspectives
Which creates a bigger “Big Picture”

Teams of teams may be organized by technology domain, product, or product line, system or subsystem

Features also require testing

And maybe a new team …..

Features typically span many teams

Sometimes, a special team is dedicated for the purpose of testing system level features
What about non-functional requirements?

- Features and user stories express functional requirements
- But other requirements (NFRs) determine system quality as well:
  - Performance, reliability and security requirements
  - Industry and Regulatory Standards
  - Design constraints, such as those that provide common behavior across like components
- Typically, these system level qualities
  - Span multiple components/products/applications/services/subsystems
  - Can often only be tested at the system level

NFRs can be considered as constraints on new development

“When we add labels to conversations, we still have to meet the accessibility standards.”
Which must also be tested

Often requires specialty skills and tools

May also be province of system team

Summary for Agile Programs:
Is the model still lean and scalable?

<table>
<thead>
<tr>
<th>It is Still Lean</th>
<th>It is Quite Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teams apply only the project level story abstractions</td>
<td>Features are implemented by stories, and can be traced with tooling</td>
</tr>
<tr>
<td>Features provide a higher level abstraction for program management</td>
<td>Higher abstraction simplifies reasoning and assessment for large programs</td>
</tr>
<tr>
<td>Just-in-time feature elaboration eliminates too early requirement specification inventory</td>
<td>One Feature backlog can drive Stories for many teams</td>
</tr>
<tr>
<td>Feature backlog construct facilitates system level pull/kanban system</td>
<td>Separation of backlogs simplifies administration and tooling</td>
</tr>
</tbody>
</table>
SCALING THE MODEL TO AGILE PORTFOLIO MANAGEMENT

The Agile Enterprise Big Picture

In 2009 Leffingwell, LLC.

Inspired by collaboration
Leffingwell, LLC & Symbian Software Ltd.
At the enterprise portfolio level, even system features may be too fine grained

- There may be dozens of concurrent programs
- Each delivering dozens of features to market
- How do portfolio managers communicate the sweeping, larger scale initiatives that drive all those programs?
- We use the word “Epic” to describe this content type

Epics drive programs with features

Epics are key value propositions that create competitive advantage

Epic may be implemented over long periods, even years

Abstract, high level, visionary
Themes represent investment priorities

Themes are key product value propositions that provide marketplace differentiation and competitive advantage.

Themes are allocated, not prioritized

- Introduce voice and video chat from within mail
- Desktop client integrations
- Mail for Mobile 2.0
- Group chat from within mail

Why not forced-rank prioritization?

- Themes are designed to be addressed on “a percentage of time to be allocated basis.”
  - Different from user stories and features
  - example: lowest priority story on an iteration backlog may not be addressed at all in an iteration and yet the iteration could be a success (meet its stated objectives).
- However, if the lowest priority (smallest investment mix) theme is not addressed over time, the enterprise may ultimately fail in its mission by not making its actual investments based on longer term business priorities.
FINALLY, A FULLY EXPANDED MODEL FOR AGILE PORTFOLIO MANAGEMENT

A fully elaborated model for the agile enterprise
Summary – Is the model still lean and scalable, really?

<table>
<thead>
<tr>
<th>It is Still Lean</th>
<th>It is Scalable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portfolio planners need only two, light-weight, high level abstractions.</td>
<td>Portfolio focus on business case and investment mix, rather than system requirements</td>
</tr>
<tr>
<td>There are only a few investment themes at any one time</td>
<td>Lighter weight, higher level artifacts simplifies reasoning about large numbers of programs</td>
</tr>
<tr>
<td>There are just a few epics of interest in play at the various program release boundaries</td>
<td>Epic-to-feature hierarchy assures investment follows strategic objectives across the full enterprise</td>
</tr>
</tbody>
</table>

In conclusion - caveats

- This is only a model; there is no UML for agile “things”
- If you don’t need it all, don’t use it all
- In any case, always do the “simplest thing that can possibly work”
END