



Ingegneria del Software

Corso di Laurea in Informatica per il Management

Agile software development

Davide Rossi
Dipartimento di Informatica
Università di Bologna



The problem

Efficiency: too much effort spent in overhead (all the activities that are not directly related to the construction of the software system). Improving the process in order to improve the product does not work as well as in other engineering branches.

The problem

The effort spent to guarantee adherence to the plan: contract negotiation, comprehensive (process-related) documentation, risk assessment, etc., could be wasted when the plan is not crystal clear from the beginning. Which is usually the case in software development.

We don't just need our software to be “flexible”, we need our whole development system to be able to adapt to change.



Manifesto for Agile Software Development

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.



Manifesto for Agile Software Development

Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas



Agile software development

Agile software development is NOT:

- a process
- a design method

Agile software development is:

- a collection of **practices** guided by a set of **principles** inspired by **values**

The 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.

The principles

- 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 principles

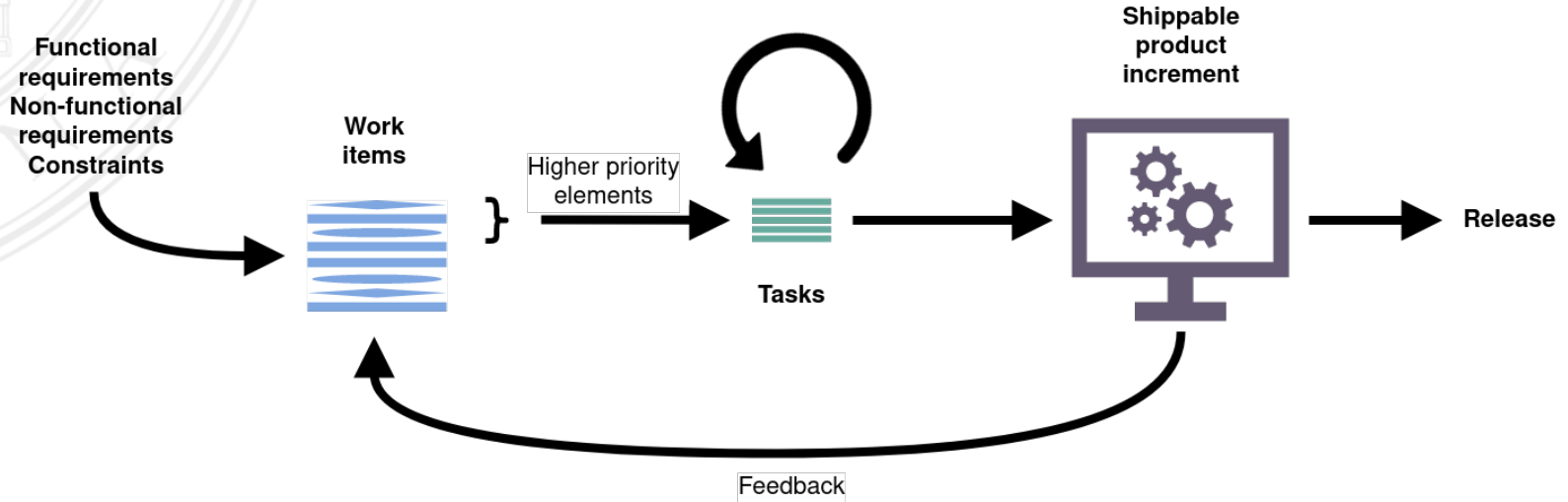
- The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity – the art of maximizing the amount of work not done – is essential.
- The best architectures, requirements, and 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.

Agile methods

In the last few years a growing number of agile-inspired methods have been proposed:

- Agile Modeling
- Agile Unified Process
- Crystal Clear
- Extreme Programming
- Scrum
- ... and others ...

Agile lifecycle



The practices

- Refactoring
- Small release cycles
- Continuous integration
- Coding standard
- Collective ownership
- Planning game
- Whole team
- Daily meetings
- Test-Driven Design
- Code and design reviews
- Pair programming
- Document late
- Use of design patterns
- ...

See <http://guide.agilealliance.org/>

Code review

By code review we usually mean the practice by which new/revised code must pass a review before being committed.

Our study reveals that while finding defects remains the main motivation for review, reviews are less about defects than expected and instead provide additional benefits such as knowledge transfer, increased team awareness, and creation of alternative solutions to problems.

[Bacchelli, Bird]

Pair programming: extreme code review

Pair programming consists of two programmers sharing a single workstation (one screen, keyboard and mouse among the pair). The programmer at the keyboard is usually called the "driver", the other, also actively involved in the programming task but focusing more on overall direction is the "navigator"; it is expected that the programmers swap roles every few minutes or so.

Test-Driven Design

Is a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be described by the following set of rules:

- write a single unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write just enough code, the simplest possible, to make the test pass
- refactor the code until it conforms to the simplicity criteria
- repeat, accumulating unit tests over time

User stories

- Several agile software development methods adopt user stories to represent requirements.
- User stories provide processable functional description of the system (from the user's viewpoint, as the name suggests) and are processed into **acceptance tests**.

User stories

- User stories are concise sentences, written in the domain language, capturing the expectations of the users.
- User stories are NOT use cases.

User stories templates

- Most usual format:

As a <role>, I want <goal> so that <benefit>.

- As an **employee** I want to **purchase a parking pass** so that **I can drive to work.**
- "Jobs stories", from jobs to be done (JTBD):
When <event>, I want to <goal> so that <benefit>.
 - When I **drive to work**, I want to be able to **purchase a parking pass** to access the company's parking so that **I can be at my desk on time.**

Acceptance tests

- Are integral parts of user stories.
- Often follow the Given-When-Then template.

Given my bank account is in credit, and I made no withdrawals recently,

When I attempt to withdraw an amount less than my card's limit,

Then the withdrawal should complete without errors or warnings

Example

- Story: As a cyclist, I want to follow my friends' cycling routes so that I can join them on rides
- Test: Given that the cyclist wants to ride with friends, when they check the map view, then the routes of other cyclists in their social network should be visible.

Stories are volatile

Stories are not meant to survive their processing.

What persists are the associated tests.

Stories, epics, themes

- Stories describing high level features are usually collected early but are underspecified and are refined as the project progresses
- **Epics** are large user stories; they usually need more than one iteration to be fully developed
- Epics are split in smaller, more detailed stories when they approach the development stage
- **Themes** are collection of related stories

INVEST

An accepted set of criteria, or checklist, to assess the quality of a user story:

- Independent
- Negotiable
- Valuable
- Estimable
- Small
- Testable

Independent

- Stories should not depend on each other as much as possible.
- As a customer I want to buy a good with my AmEx credit card
- As a customer I want to buy a good with my other credit card

Negotiable

- User stories are not contracts.
- User stories are the result of a negotiation and are prone to be re-negotiated at any point in time.
- Often a negotiation phase takes place at estimation time.

Valuable

- Stories must provide a value.
- Note: not always for the end user. A perspective should be taken, and the story written from that perspective.

Estimable

- The team should be able to estimate the level of complexity and the amount of work need for its processing.
- A high level of uncertainty is a good indicator that the scope of the story is too wide or not focused enough.

Small

- A user story should be processed in an iteration. By the end of the iteration, they have to be considered done.
- Iterations in agile methods are usually short and we have more than one developer to be kept busy.

What if the story is not short?

1 PREPARE THE INPUT STORY

Does the big story satisfy INVEST* (except, perhaps, small)?

YES
Combine it with another story or otherwise reformulate it to get a good, if large, starting story.

NO
Is the story size 1/10 to 1/2 of your velocity?

You're done.

Continue. You need to split it.

WORKFLOW STEPS

Can you split the story so you do the beginning and end of the workflow first and enhance with stories from the middle of the workflow?

Can you take a thin slice through the workflow first and enhance it with more stories later?

DEFER PERFORMANCE

Could you split the story to just make it work first and then enhance it to satisfy the non-functional requirement?

Does the story get much of its complexity from satisfying non-functional requirements like performance?

Could you split the story to do that simple core first and enhance it with later stories?

SIMPLE/COMPLEX

Does the story have a simple core that provides most of the value and/or learning?

Could you group the later stories and defer the decision about which story comes first?

MAJOR EFFORT

When you apply the obvious split, is whichever story you do first the most difficult?

Does the story get the same kind of data via multiple interfaces?

Can you split the story to handle data from one interface first and enhance with the others later?

INTERFACE VARIATIONS

OPERATIONS

Can you split the operations into separate stories?

Does the story include multiple operations? (e.g. is it about "managing" or "configuring" something?)

BUSINESS RULE VARIATIONS

Can you split the story so you do a subset of the rules first and enhance with additional rules later?

Does the story have a variety of business rules? (e.g. is there a domain term in the story like "flexible dates" that suggests several variations?)

VARIATIONS IN DATA

Does the story do the same thing to different kinds of data?

Can you split the story to process one kind of data first and enhance with the other kinds later?

BREAK OUT A SPIKE

Are you still baffled about how to split the story?

Can you find a small piece you understand well enough to start?

Write that story first, build it, and start again at the top of this process.

Can you define the 1-3 questions most holding you back?

Write a spike with those questions, do the minimum to answer them, and start again at the top of this process.

Take a break and try again.

3 EVALUATE THE SPLIT

Are the new stories roughly equal in size?

YES

Is each story about 1/10 to 1/2 of your velocity?

NO

Try another pattern on the original story or the larger post-split stories.

Do each of the stories satisfy INVEST?

Try another pattern.

Are there stories you can deprioritize or delete?

Try another pattern. You probably have waste in each of your stories.

Is there an obvious story to start with that gets you early value, learning, risk mitigation, etc.?

Try another pattern to see if you can get this.

You're done, though you could try another pattern to see if it works better.

2 APPLY THE SPLITTING PATTERNS

start here

last resort

* INVEST - Stories should be:
Independent
Negotiable
Valuable
Estimable
Small
Testable

Testable

- In most agile methods, a story is **done** only when the corresponding features pass the acceptance tests.
- No tests -> no stories done.
- Some argue for test first approaches (makes sense, but not so easy to implement in practice).

Agile and evolution

- Development is just a part of software lifecycle, evolution is just as important
- In agile approaches, knowledge is shared implicitly and very little documentation is produced: what happens when maintenance is handed over to a different team?
- That should be *document later* but often times it becomes *document never*.

Extreme programming - XP

- Four activities: coding, testing, listening, and designing.
- Five values: communication, simplicity, feedback, courage, and respect.
- Three principles: feedback, assuming simplicity, embracing change.

XP practices

Practices in XP are split in four groups:

- fine scale feedback
- continuous process
- shared understanding
- programmer welfare

Fine scale feedback

- Pair programming
- Planning game
- Test driven development
- Whole team

Continuous process

- Continuous integration
- Design improvement
- Small releases



Shared understanding

- Coding standard
- Collective code ownership
- Simple design
- System metaphor



Programmer welfare

- Sustainable pace

Programmer welfare



When do changes induce fixes?

Full Text:  PDF  [Get this Article](#)

Authors: [Jacek Śliwerski](#) International Max Planck Research School, Saarbrücken, Germany
[Thomas Zimmermann](#) Saarland University, Saarbrücken, Germany
[Andreas Zeller](#) Saarland University, Saarbrücken, Germany

Published in:



- **Proceeding**
MSR '05 Proceedings of the 2005 international workshop on Mining software repositories
Pages 1-5
[ACM](#) New York, NY, USA ©2005
[table of contents](#) ISBN: 1-59593-123-6 doi>[10.1145/1083142.1083147](#)
- **Newsletter**
ACM SIGSOFT Software Engineering Notes [Homepage](#)
Volume 30 Issue 4, July 2005
Pages 1-5
[ACM](#) New York, NY, USA
[table of contents](#) doi>[10.1145/1082983.1083147](#)

Programmer welfare

When Do Changes Induce Fixes?

(On Fridays.)

Jacek Śliwerski

International Max Planck Research School
Max Planck Institute for Computer Science
Saarbrücken, Germany

sliwers@mpi-sb.mpg.de

Thomas Zimmermann Andreas Zeller

Department of Computer Science
Saarland University
Saarbrücken, Germany

{tz, zeller}@acm.org

ABSTRACT

As a software system evolves, programmers make changes that sometimes cause problems. We analyze CVS archives for *fix-inducing changes*—changes that lead to problems, indicated by fixes. We show how to automatically locate fix-inducing changes by linking a version archive (such as CVS) to a bug database (such as BUGZILLA). In a first investigation of the MOZILLA and ECLIPSE history, it turns out that fix-inducing changes show distinct patterns with respect to their size and the day of week they were applied.

Which change properties may lead to problems? We can investigate which properties of a change correlate with inducing fixes, for instance, changes made on a specific day or by a specific group of developers.

How error-prone is my product? We can assign a *metric* to the product—on average, how likely is it that a change induces a later fix?

How can I filter out problematic changes? When extracting the

Other notable events on Fridays



Fisheries Research 36 (1998) 149–157

**FISHERIES
RESEARCH**

Assessing perceptions: Do Catalan fishermen catch more shrimp on Fridays?

F. Sardà*, F. Maynou

Institut de Ciències del Mar (CSIC). Plaça del Mar, s/n., 08039 Barcelona, Spain

Received 5 November 1997; accepted 23 February 1998

Planning game

It's the planning process in XP, it's based on user stories.

It's held before each iteration. Two parts: **Release Planning** (includes customers) and **Iteration Planning** (developers only).

Customers sort stories by value (critical, significant business value, nice to have), programmers by risk.

The user stories that will be finished in the next release will be picked.



www.dilbert.com scottadams@aol.com

11-26-07 ©2007 Scott Adams, Inc./Dist. by UFS, Inc.