Agile Software Development

L4 – Kanban, XP, and code refactoring

Mariana Falco
mariansa.falco@ut.ee
Agenda

1. Quick recap
2. Organizing User Stories
3. XP and refactoring
An agile program

collections of epics that drive toward a common goal

large bodies of work that can be broken down into a number of smaller tasks (called stories)

short requests written from the perspective of an end user
User Story

• The smallest **unit of work** in an agile framework

• A USs is a general explanation of a software feature written from the **perspective of the end user**

• In Scrum, **USs are added to the sprints** and burned down over the duration of each sprint

• USs are the **building blocks** of larger agile frameworks like epics and initiatives
User Story Structure

As a [persona]
I [want to]
so that [benefit]

Who are we building this for?
Their intent - not the feature they use
What is the big problem that needs solving?

+ acceptance criteria

https://www.atlassian.com/agile/project-management/user-stories
Organizing User Stories

How to envisage the entire product or service as a series of tasks which the user completes?

Two mechanisms:

• Kanban board
• User Story Mapping
Kanban

• A framework to implement ASD
• It requires real-time communication (of capacity) and full transparency of work
• Work items are represented visually on kanban boards
• Kanban promotes continuous collaboration and encourages active, ongoing learning and improving by defining the best possible team workflow.
Kanban: Three Principles

• **Visualize what you do today (workflow):** seeing all the items in context of each other can be very informative  

• **Limit the amount of work in progress (WIP):** this helps balance the flow-based approach, so teams don’t start and commit to too much work at once

• **Enhance flow:** when something is finished, the next highest thing from the backlog is pulled into play
Kanban Board

To visualize work, limit work-in-progress, and maximize efficiency (or flow)

<table>
<thead>
<tr>
<th>Backlog</th>
<th>To Do</th>
<th>In Progress</th>
<th>Testing</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature 10 hrs HIGH</td>
<td>Bug Fix 2 hrs Medium</td>
<td>Research 3 hrs Medium</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update 4 hrs Low</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Content 2 hrs HIGH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kanban Board

https://www.atlassian.com/agile/kanban/boards
Kanban Board and WIP

WIP = Work In Progress → number of work items that you have going at the same time

From KANBAN IN ACTION by MARCUS HAMMARBERG and JOAKIM SUNDÉN, Copyright 2015.
User Mapping
User Story Mapping

The backbone is also the narrative flow!

Priority +

Priority -

https://plan.io/blog/user-story-mapping/

MTAT.03.295 | Agile Software Development | © Mariana Falco 2022
User Story Mapping

Let’s review the assignment!
Release vs Iteration planning

• **A release** is made up of one or more **iterations**

• **Release planning** refers to determining a balance between a projected timeline and a desired set of functionality

• **Iteration planning** refers to selecting stories for inclusion in this iteration

• The customer and developers involved in and iteration planning made of user stories
Release planning

- Product vision drives product roadmap
- Product roadmap drives release plans

Release plan establishes the iterations

Release Plan
- Iteration 0
- Iteration 1
- Iteration 2
- Iteration 3
- Iteration n

Iteration plans schedules feature development
- Prioritized features delivered by user stories
- Tasks created to deliver user stories

User perspective

Developer perspective
Planning a release

• Give **priority** to the user stories, which includes:
  • The desirability of the features (customer)
  • Technical risk, dependencies (developers)
  • Cost  
    • The cost of a story is the **estimate** given to it by the developers
      • How much work can the team complete in one iteration?
    • **Effort estimation**  
      • several techniques!
eXtreme Programming (XP)
eXtreme Programming (XP)

• An ASD framework that aims to produce higher quality software
• It is the most specific of the agile frameworks regarding engineering practices
• The five values are communication, simplicity, feedback, courage, and respect

https://www.agilealliance.org/glossary/xp/
eXtreme Programming (XP)

The practices are:

• **Sit Together** [communication]
• **Whole Team** [people with a need and team]
• **Informative Workspace** [up-to-date]
• **Energized Work** [focused]
• **Pair Programming**
• **Stories**
• **Weekly Cycle** [= iteration]

https://www.agilealliance.org/glossary/xp/
eXtreme Programming (XP)

The practices are:

- **Quarterly Cycle** [= release]
- **Slack** [low priority tasks that can be dropped if the team gets behind on more important tasks]
- **Ten-Minute Build** [to automatically build the whole system and run all test in 10 min]
- **Continuous Integration** [code changes are immediately tested when they are added to a larger code base]
- **Test-First Programming** [Write failing automated test → Run failing test → develop code to make test pass → run test → repeat]
- **Incremental Design** [perspective then details + refactoring]

https://www.agilealliance.org/glossary/xp/
eXtreme Programming (XP)

• The four most common roles are:
  
  - **The XP Customer** [all of the business decisions: features, acceptance criteria, available budget, in what order?]
  
  - **The Developer**
  
  - **The Tracker** [often one of the developers who keep track of relevant metrics]
  
  - **The Coach** [outside consultant or someone who has used XP before, mentoring]
When should XP be used?

Extreme Programming (XP) was created in response to problem domains whose requirements change. Your customers may not have a firm idea of what the system should do. You may have a system whose functionality is expected to change every few months. In many software environments dynamically changing requirements is the only constant. This is when XP will succeed while other methodologies do not.

XP was also set up to address the problems of project risk. If your customers need a new system by a specific date the risk is high. If that system is a new challenge for your software group the risk is even greater. If that system is a new challenge to the entire software industry the risk is greater even still. The XP practices are set up to mitigate the risk and increase the likelihood of success.

XP is set up for small groups of programmers. Between 2 and 12, though larger projects of 30 have reported success. Your programmers can be ordinary, you don’t need programmers with a Ph.D. to use XP. But you can not use XP on a project with a huge staff. We should note that on projects with dynamic requirements or high risk you may find that a small team of XP programmers will be more effective than a large team anyway.

XP requires an extended development team. The XP team includes not only the developers, but the managers and customers as well, all working together elbow to elbow. Asking questions, negotiating scope and schedules, and creating functional tests require more than just the developers be involved in producing the software.

Another requirement is testability. You must be able to create automated unit and functional tests. While some domains will be disqualified by this requirement, you may be surprised how many are not. You do need to apply a little testing ingenuity in some domains. You may need to change your system design to be easier to test. Just remember, where there is a will there is a way to test.

The last thing on the list is productivity. XP projects unanimously report greater programmer productivity when compared to other projects within the same corporate environment. But this was never a goal of the XP methodology. The real goal has always been to deliver the software that is needed when it is needed. If this is what is important to your project it may be time to try XP.

http://www.extremeprogramming.org/when.html
When should XP be used?

Extreme Programming (XP) was created in response to problem domains whose requirements change. Your customers may not have a firm idea of what the system should do. You may have a system whose functionality is expected to change every few months. In many software environments dynamically changing requirements is the only constant. This is when XP will succeed while other methodologies do not.

XP was also set up to address the problems of project risk. If your customers need a new system by a specific date the risk is high. If that system is a new challenge for your software group the risk is even greater. If that system is a new challenge to the entire software industry the risk is greater even still. The XP practices are set up to mitigate the risk and increase the likelihood of success.

XP is set up for small groups of programmers. Between 2 and 12, though larger projects of 30 have reported success. Your programmers can be ordinary, you don’t need programmers with a Ph.D. to use XP. But you can not use XP on a project with a huge staff. We should note that on projects with dynamic requirements or high risk you may find that a small team of XP programmers will be more effective than a large team anyway.

XP requires an extended development team. The XP team includes not only the developers, but the managers and customers as well, all working together elbow to elbow. Asking questions, negotiating scope and schedules, and creating functional tests require more than just the developers be involved in producing the software.

Another requirement is testability. You must be able to create automated unit and functional tests. While some domains will be disqualified by this requirement, you may be surprised how many are not. You do need to apply a little testing ingenuity in some domains. You may need to change your system design to be easier to test. Just remember, where there is a will there is a way to test.

The last thing on the list is productivity. XP projects unanimously report greater programmer productivity when compared to other projects within the same corporate environment. But this was never a goal of the XP methodology. The real goal has always been to deliver the software that is needed when it is needed. If this is what is important to your project it may be time to try XP.
XP workflow

https://www.agilealliance.org/glossary/xp/
MTAT.03.295 | Agile Software Development | © Mariana Falco 2022
Refactoring
Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior

(noun) a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior

(verb) to restructure software by applying a series of refactorings without changing its observable behavior

Martin Fowler,
Kent Beck (2012)
Example I

```javascript
function logStart() {
    title = 'starting';
    message = 'started';
    console.log(title);
}
logStart();
document.title = title;
```

```javascript
function logStart() {
    title = 'started';
    console.log(title);
}
```
Example II

```javascript
function sqrt(value) {
    if (typeof value !== 'number' || value < 0) {
        return NaN;
    }

    // the magic happens here
}

function sqrt(value) {
    if (isNotNumber(value) || value < 0) {
        return NaN;
    }

    // the magic happens here
}

function isNotNumber(value) {
    return typeof value !== 'number';
}
```
Example of refactoring – Elixir
Adapted from https://nickjanetakis.com/blog/refactoring-elixir-code-if-cond-and-pattern-matching

def initials(name) do
  cond do
    name == nil or name == "" ->
      "?"
    String.contains?(name, " ") ->
      split_name = name |> String.split(" ")
      first_letter = split_name |> List.first() |> String.slice(0, 1)
      last_letter = split_name |> List.last() |> String.slice(0, 1)
      [first_letter, last_letter]
    true ->
      name |> String.slice(0, 1)
  end
end

def initials(nil), do: "?"
def initials(""), do: "?"
def initials(name), do: name |> String.split(" ") |> Enum.map(&String.at(&1, 0))
When to refactor

Rule of Three

✔ When you’re doing something for the first time, just get it done.

✔ When you’re doing something similar for the second time, cringe at having to repeat but do the same thing anyway.

✔ When you’re doing something for the third time, start refactoring.

https://refactoring.guru/refactoring/
When to refactor

When adding a feature

✔ Refactoring helps you understand other people’s code. If you have to deal with someone else’s dirty code, try to refactor it first. Clean code is much easier to grasp. You will improve it not only for yourself but also for those who use it after you.

✔ Refactoring makes it easier to add new features. It’s much easier to make changes in clean code.

https://refactoring.guru/refactoring/
When to refactor

When fixing a bug

✔ **Bugs in code behave just like those in real life:** they live in the darkest, dirtiest places in the code. Clean your code and the errors will practically discover themselves.

✔ **Managers appreciate proactive refactoring** as it eliminates the need for special refactoring tasks later.

https://refactoring.guru/refactoring/
When to refactor

During a code review

✔️ The code review **may be the last chance** to tidy up the code before it becomes available to the public.

✔️ **It’s best to perform such reviews in a pair with an author.** This way you could fix simple problems quickly and gauge the time for fixing the more difficult ones.

https://refactoring.guru/refactoring/
How to refactor

✔ The code should become cleaner
✔ New functionalities shouldn’t be created during refactoring
✔ All existing tests must pass after refactoring

☐ There are two cases when tests can break down after refactoring:
  • An error was made during refactoring
  • Your tests were too low-level
  • A nice way to avoid this is to write BDD-style tests

https://refactoring.guru/refactoring/
Refactoring – expected benefits

- Refactoring improves objective attributes of code (length, duplication, coupling and cohesion, cyclomatic complexity) that correlate with ease of maintenance.
- Refactoring helps code understanding.
- Refactoring encourages each developer to think about and understand design decisions, in particular in the context of collective ownership / collective code ownership.
- Refactoring favors the emergence of reusable design elements (such as design patterns) and code modules.
Refactoring – some keys

• Refactoring lowers the cost of enhancements
• Refactoring is a part of day-to-day programming
• Automated tools are helpful, but not essential