Agile Software Development

L4 – Kanban, XP, and code refactoring

Ezequiel Scott
ezequiel.scott@ut.ee
Homework assignment (HW1)

• HW1: Roll your dice! (4 points)
• Full description on Moodle
• To be completed in pairs
• Submission deadline: Friday 08.10.20 at 23:59
Agenda

• Recap
• Organizing User Stories
• XP
• Refactoring
Recap

✓ Software development processes, agile terminology
✓ Current state of Agile worldwide
✓ Test-driven Development (TDD)
✓ Requirements Management in ASD
✓ Organizing User Stories
  ✓ Kanban Board
  ✓ User Story Mapping
Agenda

• Recap

• Organizing User Stories

• XP

• Refactoring
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 Board

- It visualizes work and the process it goes through
- Generally, it is more sophisticated than “mere” task boards
Kanban Board and WIP

WIP = Work In Progress

- The WIP limit isn’t a strict rule; it’s a trigger for discussions.

WIP → the number of work items you have going at the same time.

WIP too high = work idle

WIP too low = people idle
Kanban Board vs. Kanban

• Kanban Board

is a **practice** to visualize work and the process it goes through

• Kanban

refers to an approach to **manage** and **improve** flow systems for knowledge work

the improvement relies on **visualizing the work** scheduling, managing “**flow**” as the primary measure of performance

gets its name from the use of Kanban – **visual signaling mechanisms** to control work in progress for intangible work products

Kanban is **continuous**, there are no iterations
The 3 principles of Kanban

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

• **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 promotes continuous collaboration and encourages active, ongoing learning and improving by defining the best possible team workflow.
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
User Story Mapping

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

The **backbone** is also the **narrative flow**!

**Priority** +

**Priority** -

**Time**

**Activities** (EPICS)

**Narrative Flow** (USER STORIES)

**Details**

**Release Slice**

**User Tasks**
User Story Mapping: Example (I)
Story map – example (II)
Story map – example (III)
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 team and the developers are **both involved** in release and iteration planning

Made of user stories
Release planning
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!**
Agenda

• Recap
• Organizing User Stories
• XP
• Refactoring
eXtreme Programming (XP)

• The XP customer role is responsible for **writing stories** and **acceptance tests** for each story, and sits with the development team.

• On an XP project the distinction between programmer and tester is **blurred**. Programmers write unit tests of their own code; testers program automated acceptance tests.

• XP projects include a coach and possibly a separate project manager who are responsible for guiding the team and removing obstacles from its way.

• XP includes 12 practices and 4 values

• XP is better applicable in:
  - outsourced or in-house development
  - small- to medium-sized systems
  - where requirements are vague and likely to change.
XP workflow

Extreme Programming Project

User Stories
- Requirements
- New User Story Project Velocity
- Test Scenarios

Architectural Spike
- System Metaphor
- Uncertain Estimates
- Confident Estimates

Release Planning
- Release Plan

Iteration
- Latest Version
- Next Iteration

Acceptance Tests
- Customer Approval

Small Releases
XP Practices

Here is a quick summary of each of the major practices in XP.

Planning game. Customers decide the scope and timing of releases based on estimates provided by programmers. Programmers implement only the functionality demanded by the stories in this iteration.

Small releases. The system is put into production in a few months, before solving the whole problem. New releases are made often—anywhere from daily to monthly.

Metaphor. The shape of the system is defined by a metaphor or set of metaphors shared between the customer and programmers.

Simple design. At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods. This rule can be summarized as, “Say everything once and only once.”

Tests. Programmers write unit tests minute by minute. These tests are collected and they must all run correctly. Customers write functional tests for the stories in an iteration. These tests should also all run, although practically speaking, sometimes a business decision must be made comparing the cost of shipping a known defect and the cost of delay.

Refactoring. The design of the system is evolved through transformations of the existing design that keep all the tests running.

Pair programming. All production code is written by two people at one screen/keyboard/mouse.

Continuous integration. New code is integrated with the current system after no more than a few hours.

When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

Collective ownership. Every programmer improves any code anywhere in the system at any time if they see the opportunity.

On-site customer. A customer sits with the team full-time.

40-hour weeks. No one can work a second consecutive week of overtime. Even isolated overtime used too frequently is a sign of deeper problems that must be addressed.

Open workspace. The team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the center.

Just rules. By being part of an Extreme team, you sign up to follow the rules. But they’re just the rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change.
XP Practices

Here is a quick summary of each of the major practices in XP.

Planning game. Customers decide the scope and timing of releases based on estimates provided by programmers. Programmers implement only the functionality demanded by the stories in this iteration.

Small releases. The system is put into production in a few months, before solving the whole problem. New releases are made often—anywhere from daily to monthly.

Metaphor. The shape of the system is defined by a metaphor or set of metaphors shared between the customer and programmers.

Simple design. At every moment, the design runs all the tests, communicates everything the programmers want to communicate, contains no duplicate code, and has the fewest possible classes and methods. This rule can be summarized as, “Say everything once and only once.”

Tests. Programmers write unit tests minute by minute. These tests are collected and they must all run correctly. Customers write functional tests for the stories in an iteration. These tests should also all run, although practically speaking, sometimes a business decision must be made comparing the cost of shipping a known defect and the cost of delay.

Refactoring. The design of the system is evolved through transformations of the existing design that keep all the tests running.

Pair programming. All production code is written by two people at one screen/keyboard/mouse.

Continuous integration. New code is integrated with the current system after no more than a few hours.

When integrating, the system is built from scratch and all tests must pass or the changes are discarded.

Collective ownership. Every programmer improves any code anywhere in the system at any time if they see the opportunity.

On-site customer. A customer sits with the team full-time.

40-hour weeks. No one can work a second consecutive week of overtime. Even isolated overtime used too frequently is a sign of deeper problems that must be addressed.

Open workspace. The team works in a large room with small cubicles around the periphery. Pair programmers work on computers set up in the center.

Just rules. By being part of an Extreme team, you sign up to follow the rules. But they’re just the rules. The team can change the rules at any time as long as they agree on how they will assess the effects of the change.
Agenda

• Recap
• Organizing User Stories
• XP
• 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 1

Which code segment is easier to read?

**Sample 1:**

```java
if (markT>=0 && markT<=25 && markL>=0 && markL<=25)
    { 
        float markAvg = (markT + markL)/2; 
        System.out.println("Your mark: " + markAvg); 
    }
```

**Sample 2:**

```java
if (isValid(markT) && isValid(markL))
{ 
    float markAvg = (markT + markL)/2; 
    System.out.println("Your mark: " + mark); 
}
```
Example II

```java
void printOwing() {
    printBanner();
    // print details
    System.out.println("name: "+_name);
    System.out.println("amount "+getOutstanding());
}

Refactored to

void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}
void printDetails(double outstanding) {
    System.out.println("name: "+_name);
    System.out.println("amount "+outstanding);
}
```

Figure 1: Example of Refactoring
Example – Long Methods

https://makolyte.com/refactoring-the-long-method-code-smell/
Example of refactoring – Elixir
Adapted from https://nickjanetakis.com/blog/refactoring-elixir-code-if-cond-and-pattern-matching

```elixir
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
```

```elixir
def initials(nil), do: "?"
def initials(""), do: "?"
def initials(name), do: name |> String.split(" ") |> Enum.map(&String.at(&1, 0))
```
Refactoring – When?

✅ Refactor when you **add a function**

✅ Refactor when you need to **fix a bug**

✅ Refactor as you do **code review**

❌ Do not refactor when it is **easier to start from the beginning**

❌ Do not refactor when you are **close to the deadline**
Refactoring opportunities – How?

Refactoring opportunities can be motivated by:

- Casual inspection
- Code reviews → Peer review
- Tools (JDeodorant, SonarQube)
- Bad smells
Life-cycle of a smell

- Smell Introduced
- Smell Detected
- No action
- Discard it
- It is infeasible to refactor
- The detection was wrong
- Refactor it
Bad smells (examples)

• Code duplication

• Class / method organization
  • Large class, Long method, Long parameter list, Lazy class, Data class...

• Lack of loose coupling or cohesion
  • Inappropriate intimacy, Feature envy, Data clumps...

• Too much or too little delegation
  • Message chains, Middle man...

• Comments

• ...

https://refactoring.guru/
https://refactoring.com/catalog/
Refactoring – common pitfalls

• Refactoring **does not** mean:
  • rewriting code
  • fixing bugs
  • improve observable aspects of software such as its interface

• Refactoring in the absence of safeguards against introducing defects (i.e. violating the “behaviour preserving” condition) is **risky**

• Safeguards include aids to regression testing including automated unit tests or automated acceptance tests
Does refactoring work?
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
But…

In 2010, Soetens and Demeyer found surprisingly little correlation between refactoring episodes, as identified by version control logs, and decrease in cyclomatic complexity

• methodological issues, or

• gap btw research and common practices ???

Studying the Effect of Refactorings: a Complexity Metrics Perspective

 Quinten David Soetens and Serge Demeyer
 Department of Mathematics and Computer Sciences
 University of Antwerp
 {quinten.soetens, serge.demeyer}@ua.ac.be
Automated tools

• Many languages have IDEs which automate many common refactorings

• Such tools aren't essential - I often work in programming languages without tool support, in which case I rely on taking small steps, and using frequent testing to detect mistakes.

-- Martin Fowler
Problems with code smells

• Only a good recipe book and nothing more
• It is not always easy
• It is not always useful
• Most of them are specific to Object Oriented Programming (Data Class, Gold class)
  • Declarative/functional programming? → Handout

Practice #3!!