Agile Software Development

L04 – Code refactoring

Ezequiel Scott
ezequiel.scott@ut.ee
Agenda

• Recap
• Organizing User Stories
• Refactoring
• Code Smells
Recap

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

• Recap
• Organizing User Stories
• Refactoring
• Code Smells
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 more sophisticated than "mere" task boards
Kanban Board and WIP

WIP = Work In Progress

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

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

WIP too high = work idle

WIP too low = people idle

From KANBAN IN ACTION by MARCUS HAMMARBERG and JOAKIM SUNDÉN, Copyright 2015.
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 (看板) – **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/
User Story Mapping
https://plan.io/blog/user-story-mapping/
User Story Mapping

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

The **backbone** is also the **narrative flow**!
User Story Mapping

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

The **backbone** is also the **narrative flow**!
User Story Mapping
https://plan.io/blog/user-story-mapping/

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

Priority +

Priority -
User Story Mapping

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

The **backbone** is also the **narrative flow**!
User Story Mapping
https://plan.io/blog/user-story-mapping/

The backbone is also the narrative flow!

Priority +

Priority -

Time

*planio

UNIVERSITY OF TARTU
User Story Mapping
https://plan.io/blog/user-story-mapping/

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

**USER**

**USER TASKS**

**ACTIVITIES** (EPICS)

**NARRATIVE FLOW** (USER STORIES)

**RELEASE SLICE**

**DETAILS**

Priority +

Priority -
Story map – example (1)
Story map – example (2)
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

- **Product vision** drives the product roadmap.
- **Product roadmap** drives release plans.
- **Release plan** establishes the iterations.
- **Iteration plan** schedules feature development.
  - Prioritized features delivered by user stories.
- **Tasks** created to deliver user stories.

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

**Iteration Plan**
- Feature A (User Story 1)
- Feature A (User Story 2)
- Feature B (User Story 3)
- Feature C (User Story 4)
- Feature D (User Story 5)
- Task A (5 hours)
- Task B (8 hours)
- Task C (4 hours)
- Task D (12 hours)

**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, next lecture!
Agenda

• Recap
• Organizing User Stories
• **Refactoring**
• Code Smells
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: " + markAvg);
}
```
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

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))
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

Refactor it

Discard it

It is infeasible to refactor

The detection was wrong
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!!
This week ➔ Homework!

• Homework 1: TDD & Elixir (4 points)

• Description:
  https://courses.cs.ut.ee/2020/ASD/fall/Main/Homework1

• To be completed in pairs

• Submission deadline: Friday 09.10.20 at 23:59