TDD - Walking

Learning TDD basics while doing mob programming

Kristoffer Steen

This presentation covers..

- How we learned: Pair/mob programming basics and takeaways
- TDD (very) basic process
- How to improve the way the (production) code evolves
- Object Calisthentics
- My takeaways from this course

Week 1: Pair/mob programming basics

The course did not follow the typical structure one often see, with long presentations.

- Instead, we got a short intro on today's main focus, and then the whole team solved real programming exercises together.
- The concept of Mob programming was used, and the instructors served as mentors, serving us the knowledge we needed there and then.

Mob programming roles

- Driver: The (only) person that writes code.
  - Does not think about the solution
- Navigator: Instructs the driver on what to do or how to implement this
  - Takes input from the mob, might also have to keep the mob in line if they disagree
- Mob: The expert panel
  - They intervene if there is a problem. If silent, they agree.
  - They will help the navigator/driver if they are stuck. They can also perform an internet search if needed.
  - They'll spot mistakes in code, logic, language best practice, general best practices, team practices etc

Remember:

- Rotate roles and combinations
- Be humble and respectful.

Pair/mob programming takeaways

I thought i knew something about pair/mob programming – well,.. i basically knew nothing!

- All roles can focus on their individual task, less context switching
  - The navigator does not have to spend brain power batteling the IDE, he can focus on the implementation.
- Instant feedback on the code and implementation
  - Navigator can trust the mob to catch mistakes and to keep him on track and focused on the goal
- Brain scale out and instant help for complex problems
  - The navigator can trust the mob to help out when stuck
- All participants have their own strengths and weaknesses, but get wery powerful when combined
  - This is both humbling and uplifting at the same time

Week 1: TDD (very) basic process

1. Create a failing/RED test
    - Test class and method naming: ClassName + MethodName reads as a sentence.
        - Class: «StringCalculatorShould»
        - Method: «AcceptAnyNumberAsString» or «ThrowIfEnteringNonNummericValue»
    - Start with the Assert-part, then write Act and Arrange
    - Pretend like you allready have the classes/methods you will be testing, and let the IDE handle creation of them instead of doing it manually
    - Ensure the test is failing for the right reason, review the actual test results and don't assume
2. Make the test pass/GREEN
    - Dont write more code than needed to pass that particular test
3. Commit
4. Refactor agressively (mandatory, but wait for things to repeat 3 times before generalizing)
    - Refactoring is only allowed while test is GREEN
    - Also remember to use all features of the IDE while refactoring. It's safer, faster and keeps you focused on the main task
5. Commit again
6. Repeat

Week 2: Evolving the code through transformations

When writing the implementation that makes the test pass, there are 3 ways forward:

1. Fake implementation

   E.g: Hard coding the value needed to make the test pass

2. Obvious implementation
   - Sure of the code needed
   - Used to move forward quickly
3. Triangulation with the next test – «I dont see a pattern yet» or «i see a pattern emerging, but suspect i dont have all the details yet»
   - Don’t refactor yet, instead: add more tests, make them GREEN with fake implementations and see if the pattern becoms clearer.

Transformation priority premise

When one choose to write the «obvious implementation», chose the least complex one

- There is a list of transformations to choose from
- It’s sorted from low to high complexity, so low complexity is at the top
- Pick as as high as you can from this list
- The list is not absolute, and might vary some according to your language, skillset, taste etc

https://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html

Week 3: Object Calisthenics

Kalos and Sthenos is Greek, and means «beauty» and «strength» respectively

Following some simple rules will make your code more easy to read ,maintain and test

- Only one level of indentation per method
  - Focused methods
  - If exceeding, are we breaking the Single Responsibility Principle?
- Don’t use the ELSE keyword
  - One want a single main execution lane
- Wrap all primitives and strings
  - Make code more explicit
  - Make invalid state non-representative
- First class collections (wrap all collections)
  - Don’t let users sort applicants by age if it is not a real use case
- Only one dot per line (does not count for LINQ)
  - Not: ~~dog.Body.Tail.Wag(),~~ but dog. Wag()=> dog.ExpressHapiness()
- No abbreviations, they might not be as obvious for somebody else
  - Maybe the concept or responsibility is misunderstood?
  - Maybe the class or method is doing to much?

Object Calisthenics cont.

- Keep all entities small
    - 10 files per package, 50 lines per class, 5 lines per method, 2 arguments per method
    - If exceeding, are we breaking the Single Responsibility Principle?
- No classes with more than two instance variables
    - Think Orchestrators vs Actuators
- No public getters/setters/properties
    - Objects should call each others methods, and not manipulate each other’s state directly
    - The biggest problem is when other classes are manipulating setters without clear intention or knowledge
        - Implement predefined and named behaviours, exposed as methods, instead

Overall personal takeaways

I’ve changed the way i think about software and development

- Untested code is dangerous (i sort of allready knew that)
- Writing tests up front is not slowing down the process, it’s just ensuring we hit the goal
- Can now recognize patterns in problems better
- Now seconds and minutes away from GREEN, not hours, days or weeks
- Pay more attention to access modifiers, don’t leave classes public by default.

Overall personal takeaways cont.

- I’ve learned ways to avoid doing boring stuff
  - I need to become (even) more lazy, i am not using the full potential of my IDE
    - Code generation
    - Refactoring
    - Hotkeys
  - I’ve found more ways to avoid writing comments
  - I’ve found away to avoid writing documentation, i’ll print a list of my tests instead

Overall personal takeaways cont

- Me and other developers must be less afraid of others opinions and take them as a chance to improve and learn something new

- Same goes for not being afraid to suggest a solution. It does not have to be spot on, but the idea can nudge the team in the right direction.

- Downside: I now hate code reviews even more than before, i’d rather rather wish we wrote the code together *

- Downside: Now i can’t write untested code anymore?

Course review

I think the interactive way of learning helped me learn faster and to make the skills stick.

I did not at any time get bored during the sessions, and after the session, I was tired for the right reason. This is impressive, especially considering it was a online course

I had a great time learning to walk, and I am hungry for more!

«You can’t start a fire without a spark» - Bruce Springsteen

Thank you! Any questions?

Kristoffer Steen

public string EmailAddress { get; set;} = ‘’kristoffer.steen@bouvet.no”; //Oops