# Domain Driven Design

THEORY AND HANDS ON ,

MAHMOUD FADDO

## Course Agenda

- Why Domain Driven design?
- Language and context
- Event storming
- Designing the model
- Implementing the model
- Acting with commands
- Consistency Boundary
- Aggregate persistence
- And much More ..!! (keep you excited )

# First session agenda

- Why Domain Driven design?
- Language and context
- Event storming
- Designing the model
- Implementing the model
- Acting with commands
- Consistency Boundary
- Aggregate persistence
- CQRS
- And much More ..!! (keep you excited )

## Why Domain driven design

- For a long time we used to change our architecture and methods, so why we need to use DDD , what we gain by using it.
- Lets shock you with some numbers (between 2011 and 2015)
  - the percentage of successful IT projects is only  22%.
  - Over 19% of projects failed.
  - The rest stuck in middle !!.
- So , the first and most important cause of failing any software is understanding the problem (software requirements). Developers and analysts can easily miss understand the problem and this would lead us by default to fail the project.

## Why Domain driven design

- DDD explains how people from the software industry can build an understanding of their users' needs and create software systems that solve the problem and make an impact.

## Problem space and solution space

- humans solve problems by searching for a solution in the *problem space*.
  - The problem space describes the initial and desired states
  - specific constraints and rules that define the context of the problem
  - In the software industry, people operating in the problem space are usually customers and users.
- Each real problem demands a solution, and if we search properly in the problem space, we can outline which steps we need to take to move from the initial state to the desired state .This outline and all the details about the solution form a *solution space*.

## Why is that matter

- Why do we talk about problem and solution ?
  - As a developers we are solving problems every day.
  - If we solved those problem wrong we will get a very cost solution.
- Example :
  - space-exploring nations realized that the usual ballpoint pens wouldn't work in space due to the lack of gravity. NASA then spent a million dollars to develop a pen that would work in space, and the Soviets decided to use the good old pencil, which costs almost nothing.

## Result!

- thinking about solutions prevents our brain from thinking about the problem. Instead, we start going deeper into the solution that first came to our mind, adding more levels of detail and making it the most ideal solution for a given problem.

## Fixation VS exploratory

- fixating all your attention on one particular solution, which might not be the best one at all but it was the first to come to mind, based on your previous experiences, your current understanding of the problem, and other factors (fixating is very dangerous)
- The exploratory approach to find and choose solutions involves quite a lot of work to try out a few different things, instead of concentrating on the iterative improvement of the original *good idea*. (more safe but needs to study alot)

## What is the problem of requirements?

- The requirement / user stories : describes the solution only!!.
  - Example :
    - *"Every day, the system shall generate, for each hotel, a list of guests expected to check in and check out on that day.“*
    - *"As a warehouse manager, I need to be able to print a stock-level report so that I can order items when they are out of stock.“*
  - The only describe the solution , which is risky because they describe the solution by point of view of person who wrote them.
- However, lean and agile methodologies embrace more direct communication between developers and end users. Understanding the problem by all stakeholders, from end users to developers and testers, finding solutions together, eliminating assumptions, building prototypes for end users to evaluate—all these things are being adopted by successful teams

## Dealing with complexity:

- Realizing what kind of complexity we are dealing with when creating software thus becomes very important.

## Categorize of problem (cynefin)

- Cynefin divides all problems into five categories.
  - **Simple.**
    - you have problems that can be described as *known knowns*, where best practices and an established set of rules are available, and there is a direct link between a cause and a consequence.
    - The sequence of actions for this domain is *sense-categorize-response*. Establish facts (sense), identify processes and rules (categorize), and execute them (response).

## Categorize of problem (cynefin)

- **Complicated**.
  - Here, you find problems that require expertise and skill to find the relation between cause and effect, since there is no single answer to these problems. These are *known unknowns*.
  - The sequence of actions in this realm is *sense-analyze-respond*.
  - When a thorough analysis is done, the risk of implementation failure is low. Here, it makes sense to apply DDD patterns for both strategic and tactical design, and to the implementation, but you could probably avoid more advanced exploratory techniques such as EventStorming. Also, you might spend less time on knowledge crunching, if the problem is thoroughly understood.

# Categorize of problem (cynefin)

- **Complex**.
  - Here, we encounter something that no one has done before. Making even a rough estimate is impossible. It is hard or impossible to predict the reaction to our action, and we can only find out about the impact hat we have made in retrospect. The sequence of actions in this domain is *probe-sense-respond*.
  - There are no right answers here and no practices to rely upon. Previous experience won't help either. These are *unknown unknowns*, and this is the place where all innovation happens (ALL DDD concepts is needed in such problem)

## Categorize of problem (cynefin)

- **Chaotic**.
  - This is where hellfire burns and the Earth spins faster than it should. No one wants to be here. Appropriate actions here are *act-sense-respond*, since there is no time for spikes. It is probably not the best place for DDD since there is no time or budget for any sort of design available at this stage.

## Categorize of problem (cynefin)

- **Disorder**.
  - right in the middle. The reason for it is that when being at this stage, it is unclear which complexity context applies to the situation. The only way out from disorder is to try breaking the problem into smaller pieces that can be then categorized into those four complexity contexts and then deal with them accordingly.

## Domain Knowledge

- **Domain knowledge is knowledge about the domain in which you are going to operate with your software.**
- This all comes to getting into the problem space. If you are not able to at least understand the terminology of the problem space, it would be hard (if not impossible) to even speak to your future users.

## Gain Domain Knowledge

- There are several ways to obtain domain knowledge, and here are some of them:
  - Conversations
  - Observation
  - Domain
  - ***EventStorming***

## What is next !! Keep excited

**Ubiquitous Language and context**

# Thank you

SESSION 1