Coding dojo

Practice makes perfect

Agenda

Intro - dojo + tdd + warmup kata	30m
Kata - part I	30m
Break	10m
Kata - part II	30m
Feedback	10m
Pizza - if you can/want	

Welcome to the coding dojo

Leverage each other knowledge

Coding Dojo mindset

We usually
train *on the job*

We need a place and time to experiment and *fail spectacularly*

You are here to learn not to build something, no output required

Slow down. Don't focus on getting it done, focus on doing it perfectly

It is necessary to push to the extreme to verify the validity of a technique, hence the dojo

Dojo goals

Iterative-Incremental development

Baby steps

TDD cycle

Disclaimer

The examples are ~~dead simple~~ a bit simplistic on purpose, otherwise we'd be lost trying to understand the code

?

Let's talk about *TDD*

TDD benefits

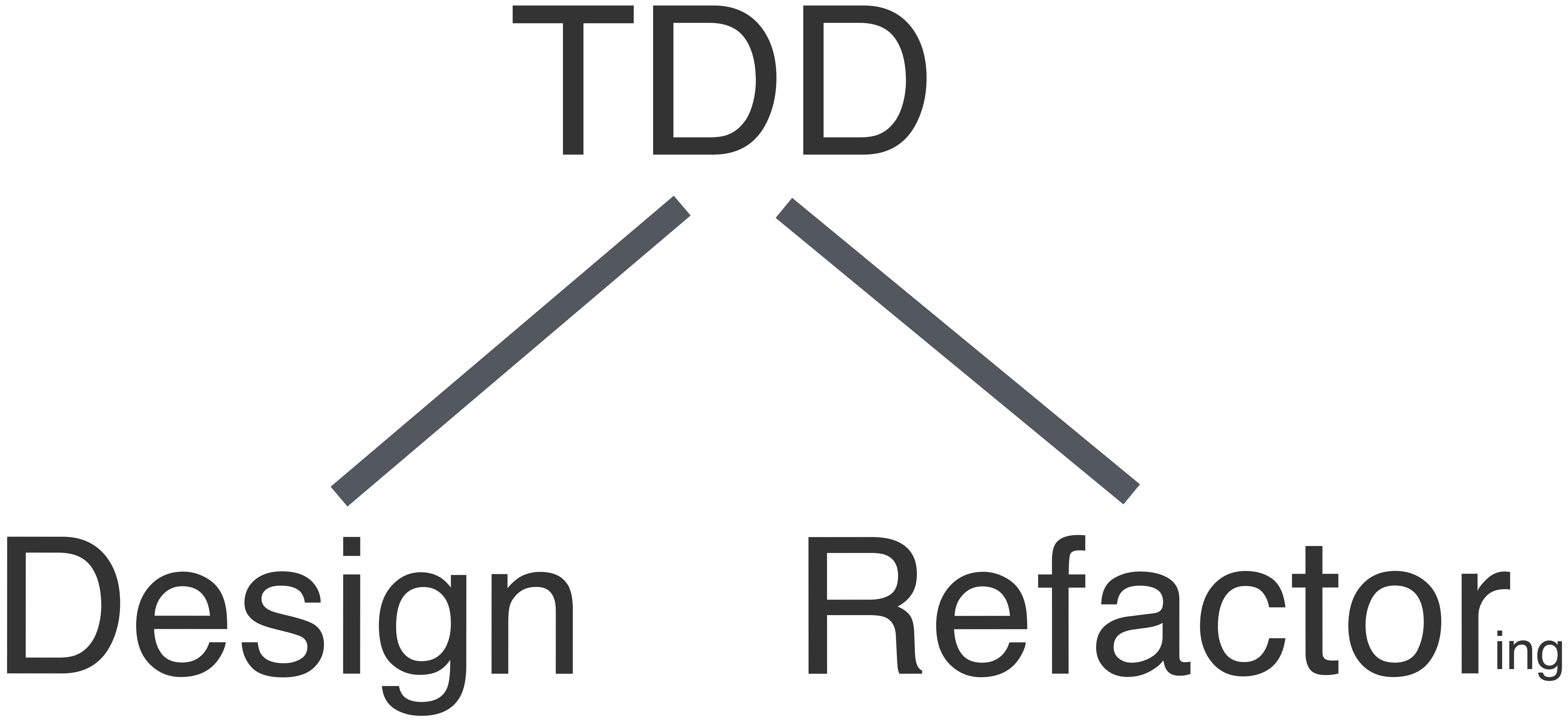
Increase chances to catch errors before production => Lower defects

Encourage more modular design => Easier to change

Increase our confidence to make changes => Enable refactoring

TDD *is hard*

TDD is a *multifaceted* technique


TDD
Design
Refactoring

TDD Cycle

1. Think about the problem
2. Write a test
3. Run it to see it fail
4. Make the test pass
5. Remove duplication

$$\begin{tikzcd}
& \text{Red} \arrow[dr] & \\
\text{Refactor} \arrow[ur] & & \text{Green} \arrow[ll]
\end{tikzcd}$$

Different phases, different thinking

Red	Constrain the problem
Green	Make it run. Speed trumps design
Refactor	Make it right. Evolve your design
here most of the time	the magic happens here!

“TDD doesn't drive good design. TDD gives you immediate feedback about what is likely to be bad design. If a test is hard to write, if a test is non-deterministic, if a test is slow, then something is wrong with the design.”

Kent Beck

Let's talk about *Design*

What is *Design?*

How you factor & arrange *things*

Design is the result of the millions of decisions you've taken over time design =

fold (stream of
everyday
coding
decision)

How can we achieve a ~~good~~ better design?

making the design
easier to change

1. small composable modules

2. make dependencies explicit

effective design doesn't guess the future but leaves space to move

Let's talk about *Refactoring*

Definition:

change implement.
without changing
behaviour

It's not *Refactoring* if

Alter how an object behave

Add/Remove a feature

It's an essential skill
to change the shape
of your codebase

⇩

Emergent design

Refactoring goal is
an economic one

↓

Reduce the marginal
cost of the next feature

a codebase
become legacy
one day at a time

Enough talking...

Warmup exercise: build a stack which we can push, pop and peek

Tonight kata

You're the new hire at Evil corporation which is in the thriving business of censorship

git.io/vHvaq

Rules

Proceed one requirements at a time. Don't cheat

Strictly implement what's asked. Try to don't anticipate design

Regex are forbidden, for your own safety :-)

Slow down. Focus on doing it right