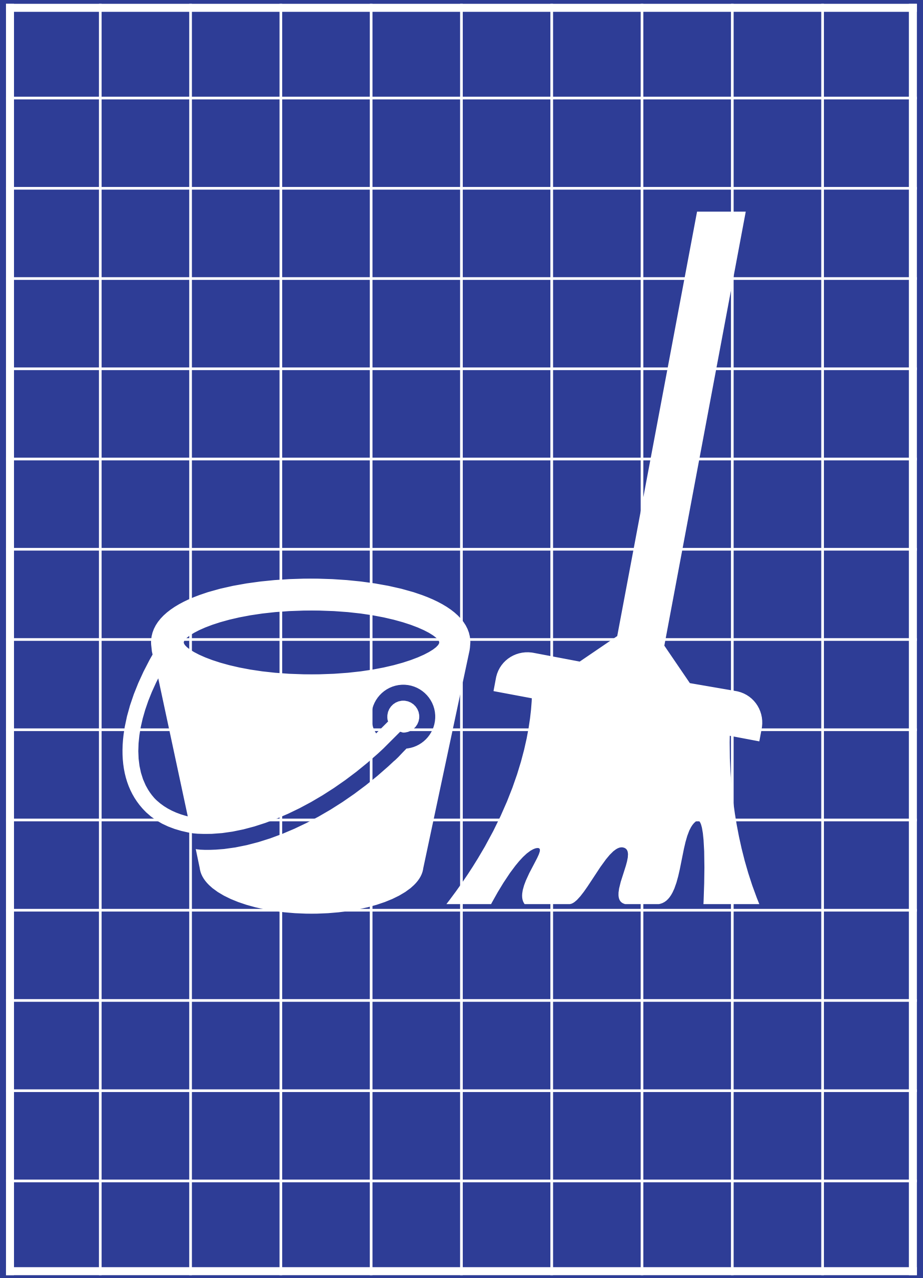
TIDY FIRST?

SOFTWARE ARQUITECTURE
GROUP 3

MARCO LORENZO
CARLOS LAVILLA
MARCOS LOSADA
DIEGO MARTÍNEZ

Escuela de Ingeniería Informática
Universidad de Oviedo



OUTLINE

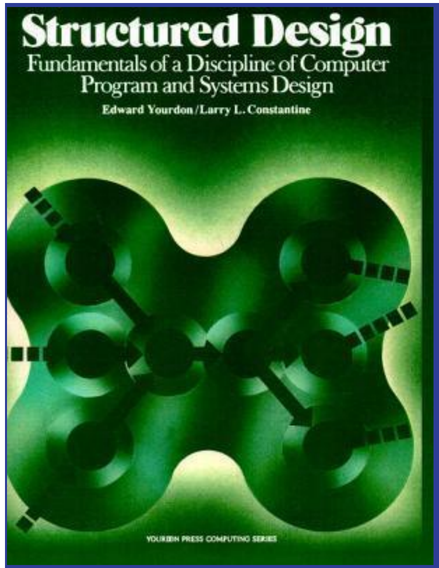
MOTIVATION
&
KENT BECK

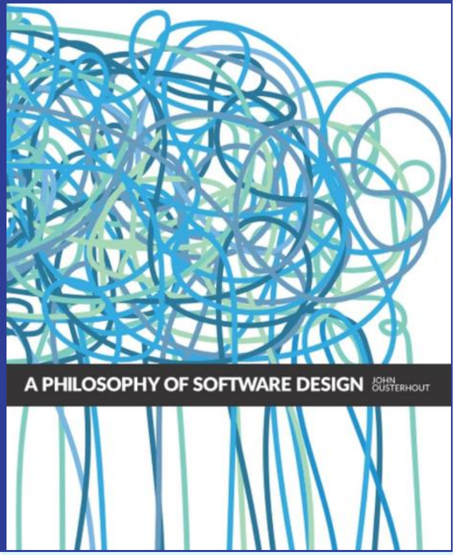
STRUCTURAL
VS
BEHAVIORAL

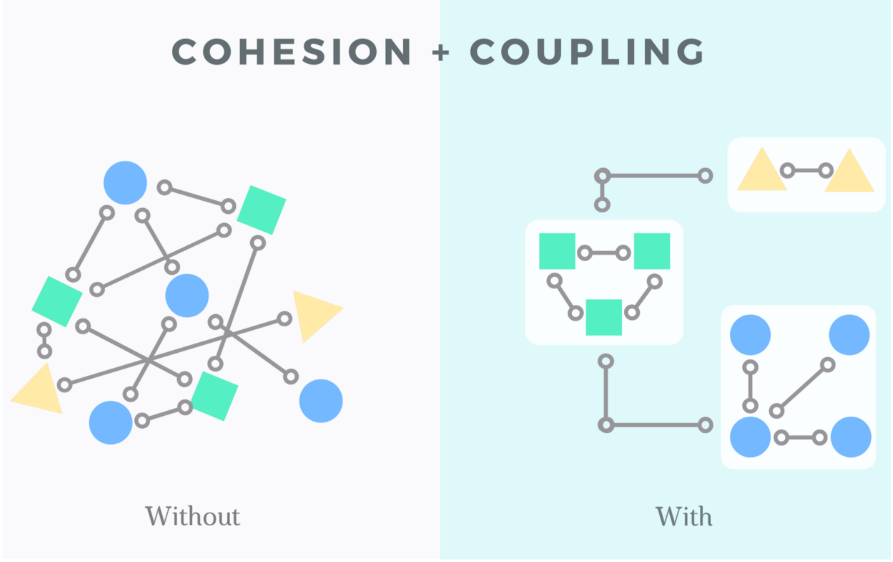
GLOSSARY

MAIN
CONCEPTS

WHEN / HOW
TO TIDY


Structured Design
Fundamentals of a Discipline of Computer Program and Systems Design
Edward Yourdon/Larry L. Constantine
YOURDON PRESS COMPUTING SERIES





MOTIVATION

Modernized version of
*“**Structured Design**”*
by *Edward Yourdon*
Larry L.. Constantine

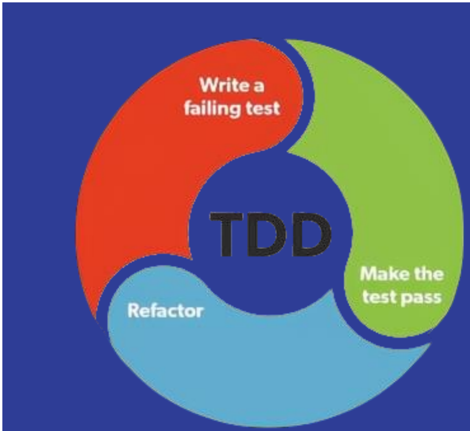
Redefinition of important concepts
Cohesion, coupling, ...

Practical viewpoint of
*“**A Philosophy of Software Design**”*
by *John Osterhout*

KENT BECK

CONTRIBUTIONS

Original signatory of
AGILE MANIFESTO



Creator of
EXTREME PROGRAMMING

TEST-DRIVEN DEVELOPMENT
pioneer

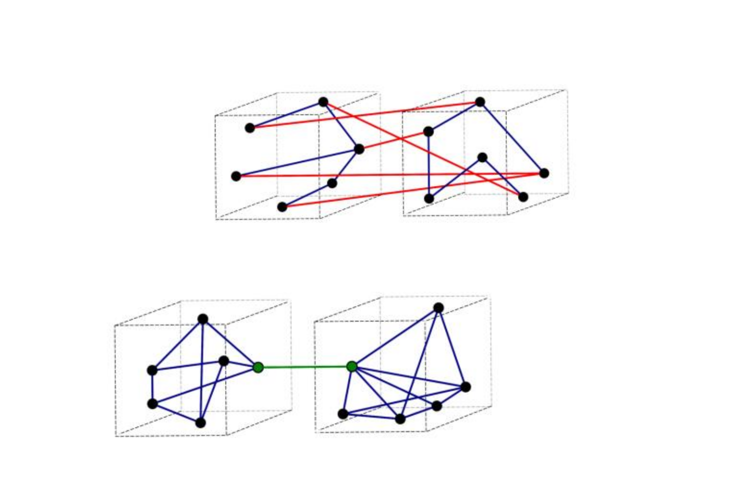
MAIN CONCEPTS

Three main concepts of the book:

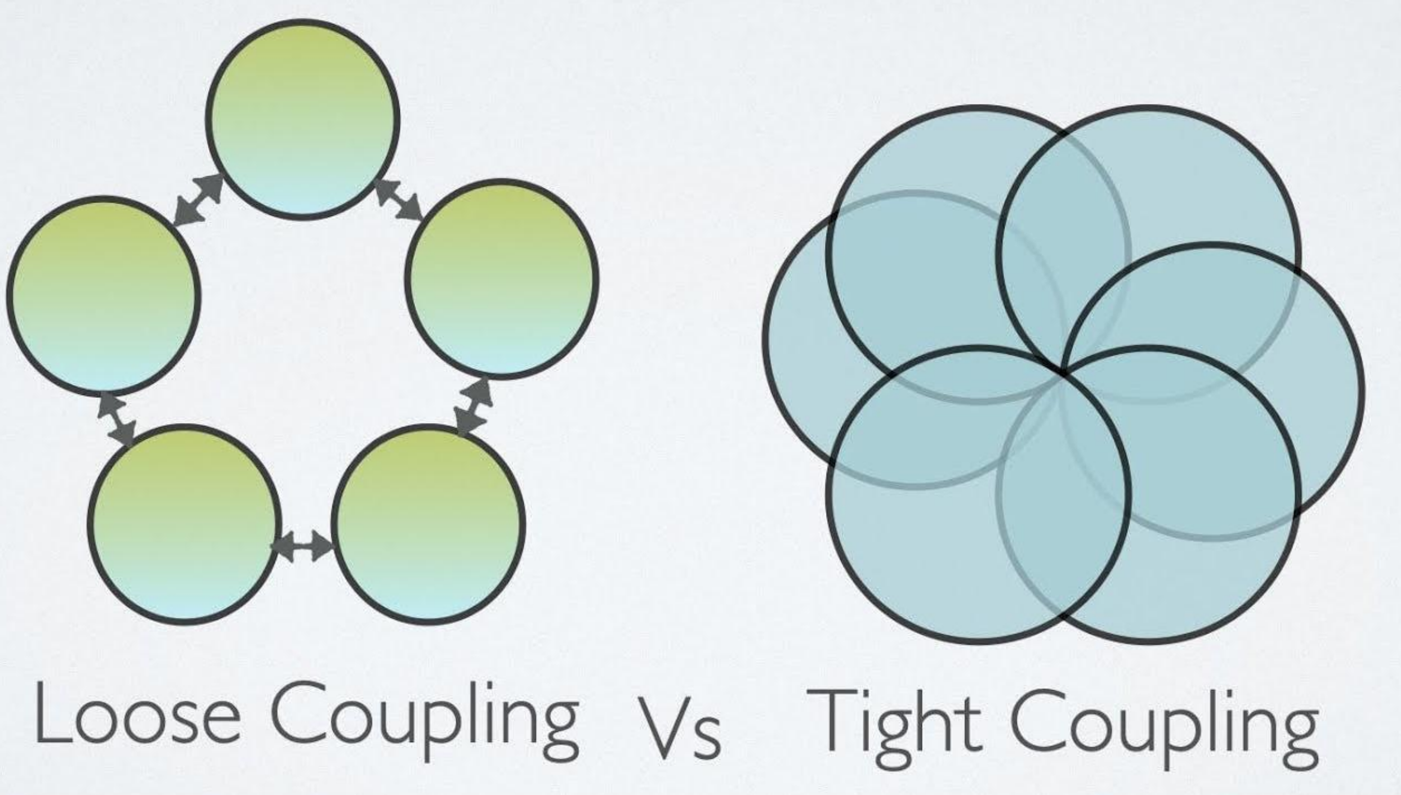
COUPLING

COHESION

TIDYING



Beck will write a book for each concept.
"Tidy first?" is the first one because tidying is the smallest skill of design, so the readers can practise with it first.



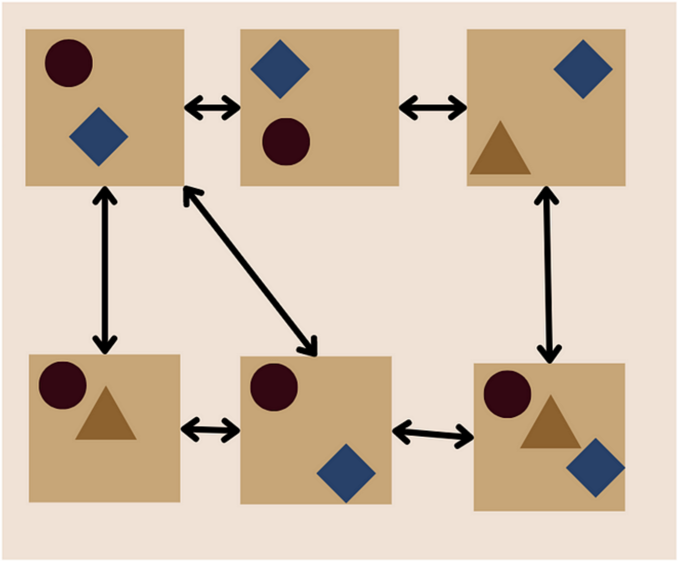
COUPLING

Elements are coupled when changing one of the elements implies changing the other one.

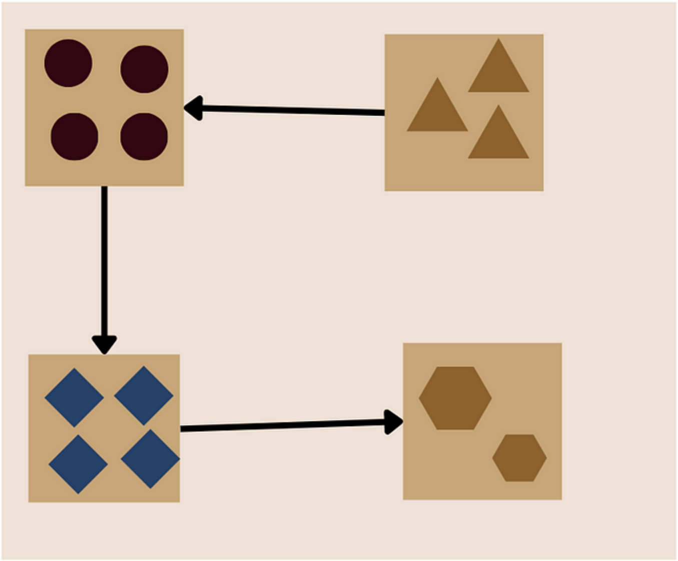
This is **coupling with respect to a change**.

Example: a function calling another is coupled with respect to changes of the name.

It is important to have **low coupling**.



Low cohesion and high coupling



High Cohesion and low Coupling

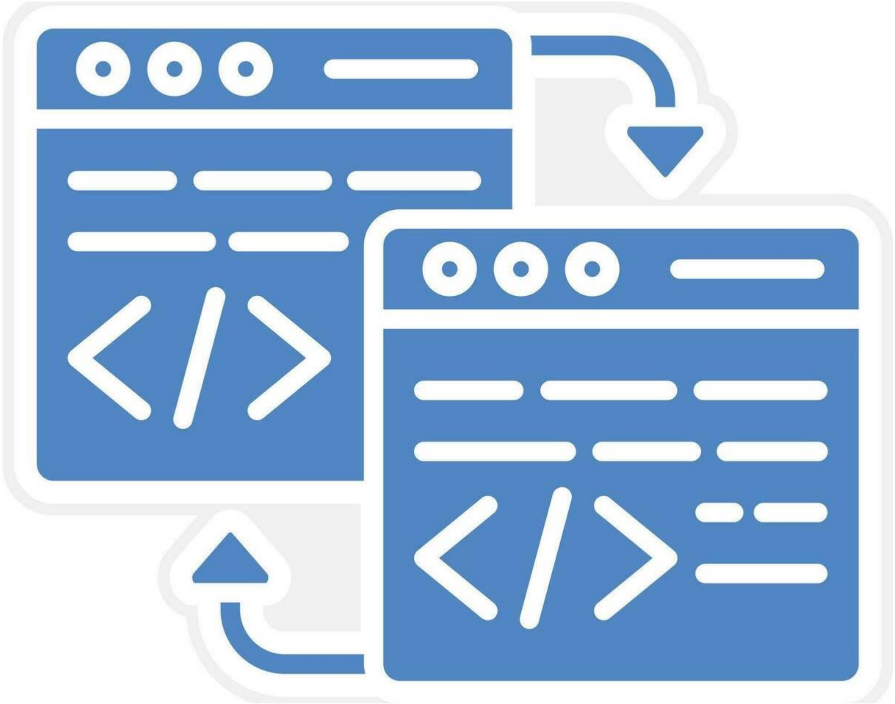
COHESION

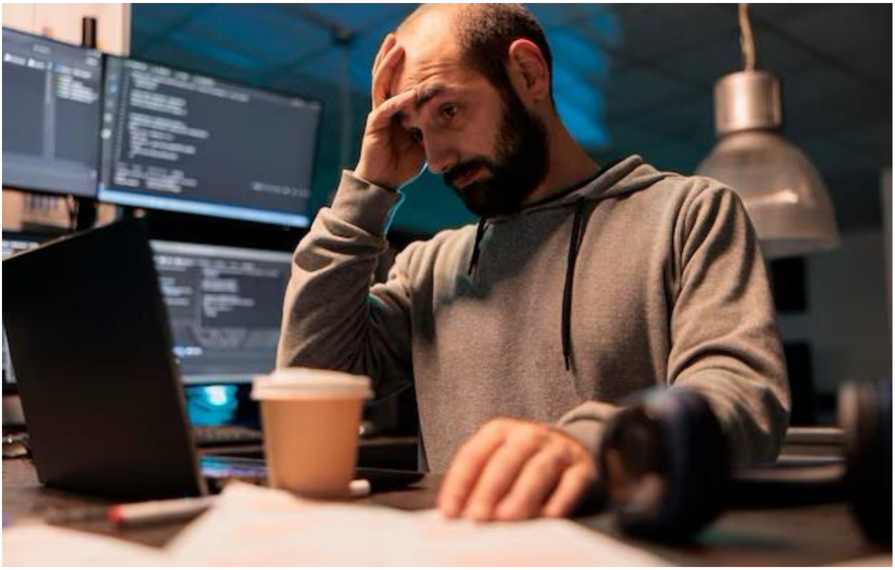
Cohesion is about keeping all the coupling together into an element. An element is cohesive if its internal elements are **coupled between them**.

Example: a file is cohesive when changing one of its functions implies changing all the other functions of the file.

The intention is to keep all the related functionality together.

It is important to have **high cohesion**.





TIDYING

Very common situation for programmers.

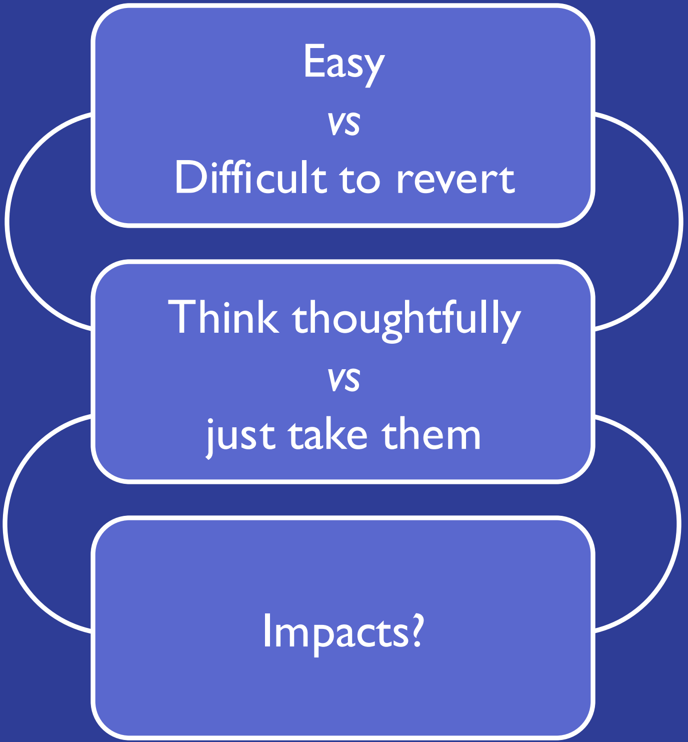
I have to make changes in a **messy code**, should I tidy first?

Tidings are **simple changes**, such as changing the name of a function so it is easier to understand.

Tidings are structural changes, they **do not change behaviour**.

BRIEF DESCRIPTION OF TYPE 1 & 2 DECISIONS

KEY ASPECTS



Easy
vs
Difficult to revert

Think thoughtfully
vs
just take them

Impacts?

STRUCTURAL *VS* BEHAVIOURAL CHANGES

MAIN ISSUES

What is the aim of these changes?

Organization and clarity against functionality

Are they reversible? Implications of reversibility?

Mostly reversible *vs* irreversible

Relationship with Type 1 & 2 decisions

Do them together?

NEVER

High risk of mistakes, reduced team confidence and increased team anxiety

OPTIONS TO HANDLE STRUCTURAL AND BEHAVIOURAL CHANGES IN PRs

Option 1
Separate **different type of changes in different pull** requests

Option 2
Separate **different type of changes in different commits** within a pull request

What if I have mixed them?

Option 1. Discard it and redo it applying the previous recommendations
Option 2. Discard it and redo it so it can be explained as a direct path from A to B

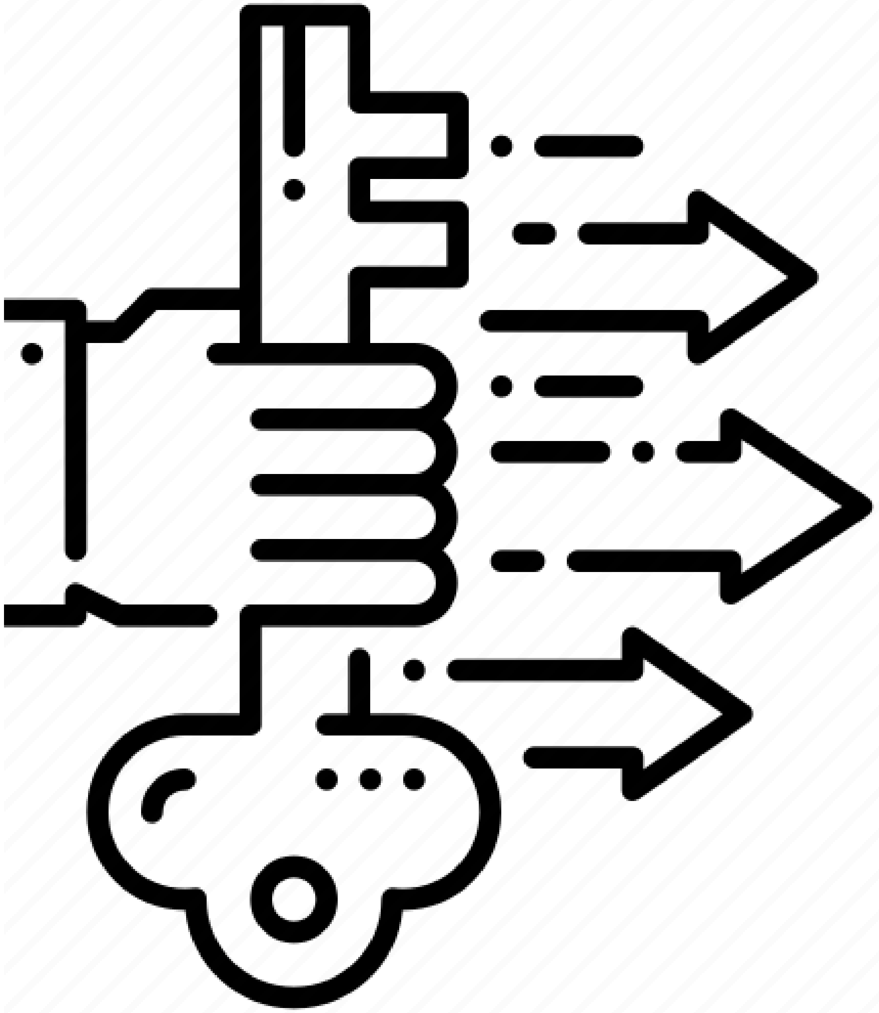
THE BIG QUESTION **TIDY FIRST?**

WHEN & HOW TO USE TIDYINGS IN OUR CODE?

Which **KEY FACTORS** influence the yes or no decision of tidying our code?

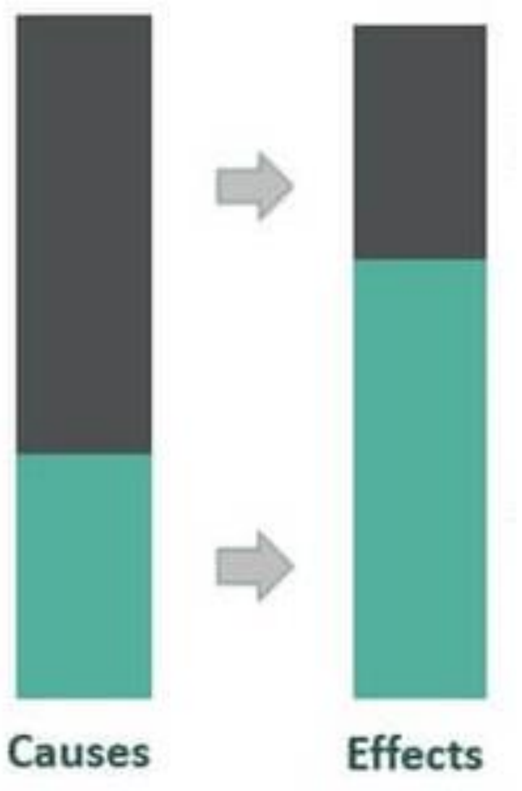
Which **GUIDELINES** can we follow to choose wisely when (not) to tidy our code?

Which **STRATEGY** to follow for knowing how to tidy our code the best way?



THE KEY FACTORS

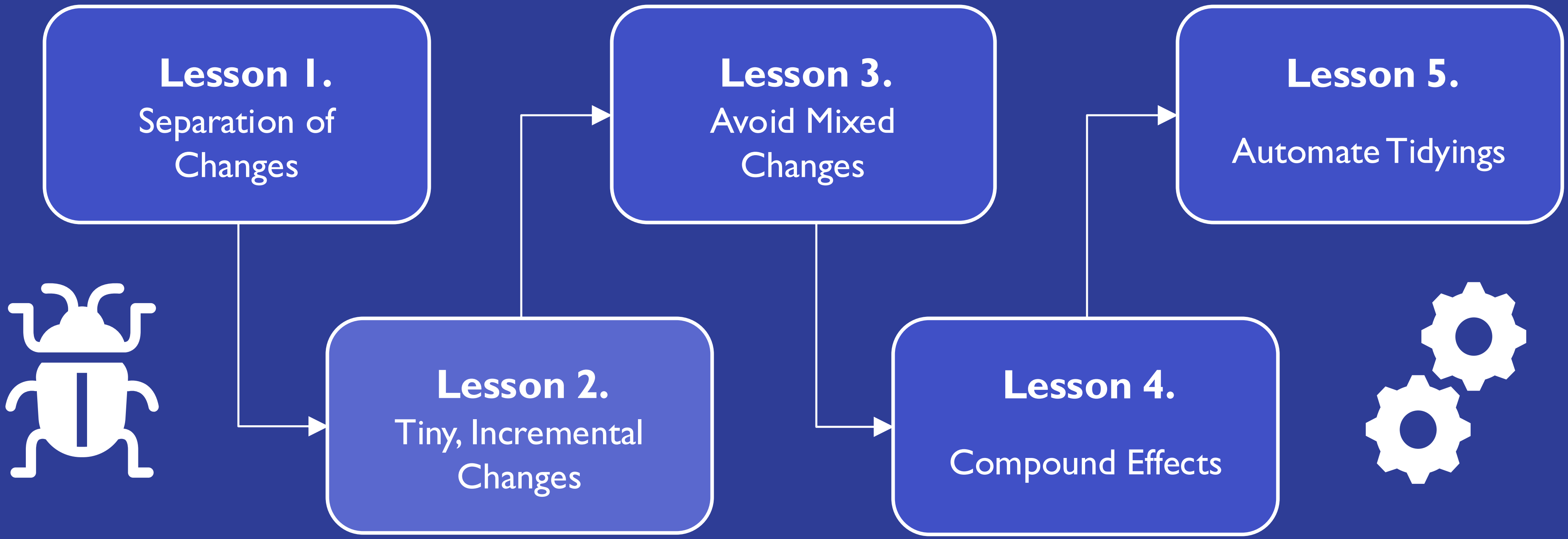
- **Cost and Benefit Analysis:** Evaluate if tidying now saves future maintenance or not
- **Coupling and Cohesion:** Consider the impact on coupling (*dependencies between code elements*) and cohesion (*how closely related functionalities are grouped*).
- **Economic Forces:** Prioritize feature delivery over code cleanliness in early product stages.



THE GUIDELINES

Short Term Benefit	Cost-Effective Tidying	Proximity to Changes "Pareto's principle"
VS	VS	
Long Term Impact	Over-Tidying	

THE STRATEGY



Lesson 1.
Separation of Changes

Lesson 2.
Tiny, Incremental Changes

Lesson 3.
Avoid Mixed Changes

Lesson 4.
Compound Effects

Lesson 5.
Automate Tidyings

GLOSSARY

POWER LAWS

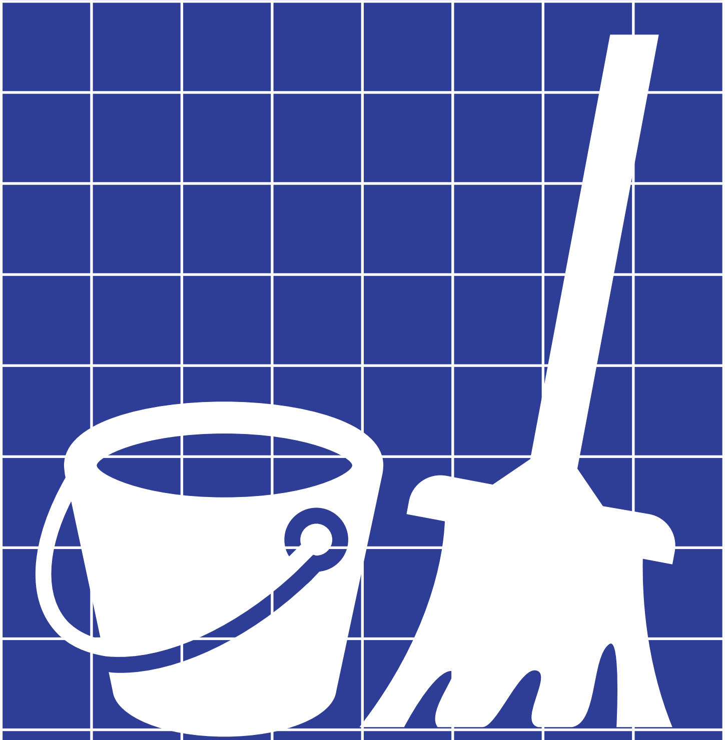
"80% of the changes in 20% of the code" – Pareto

EMPIRICAL SOFTWARE DESIGN

Shaping design on real-world data, not theory.

NET PRESENT VALUE (NPV)

Revenue early = MORE revenue later



SOFTWARE ARQUITECTURE
GROUP 3

THANK YOU