SOFTWARE ARCHITECTURE HEURISTICS & TRADE OFFS

v1.7.38.20260102

PART 1

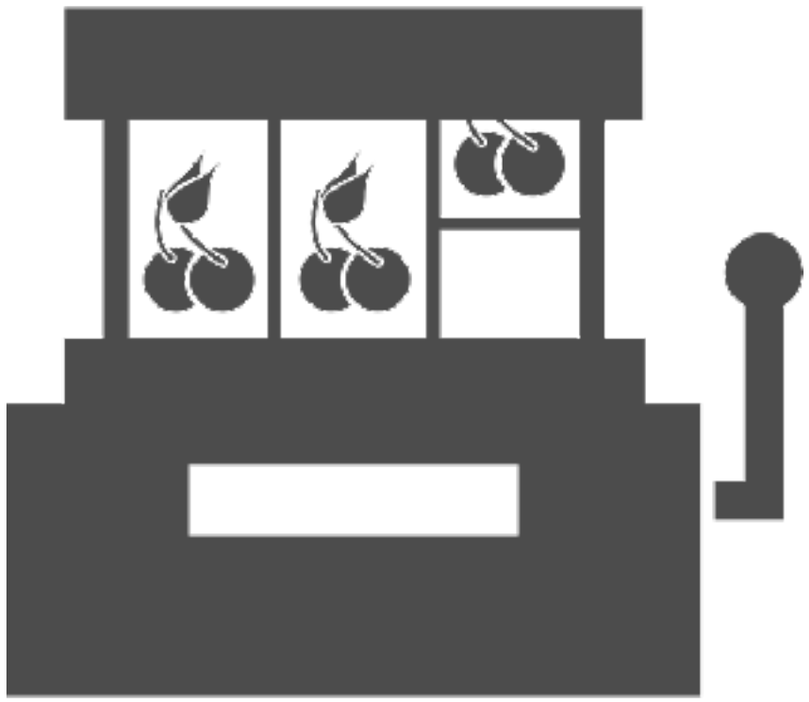

MUSEUM KIRCHERIANUM BONANNI
KIRCHERI DE ARTE MAGNETICA
HARRINGTONS WORKS
BACONI OPUS MAJUS
KIRCHERI OEDIPUS AEGYPTIACUS
STANLEYS HISTORY OF PHILOSOPHY
SEXTUS PRINCIPIA PHILOSOPHIA

HISTORY

LET'S SHARE A FEW THOUGHTS

As Software developers, we try to solve problems

… but most of the times software is seen as the problem, not the solution! … why?



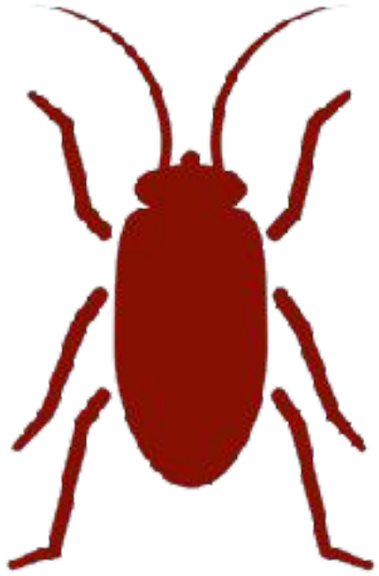
« As long as there were no machines, programming is no problem at all »

« Only a few years ago, the talk about a software crisis was blasphemy. .. and by now, it is generally recognized that the design of any large sophisticated system is going to be a very difficult job. »

« If software were to continue to be the same clumsy and expensive process as it is now, things would get completely out of the balance »

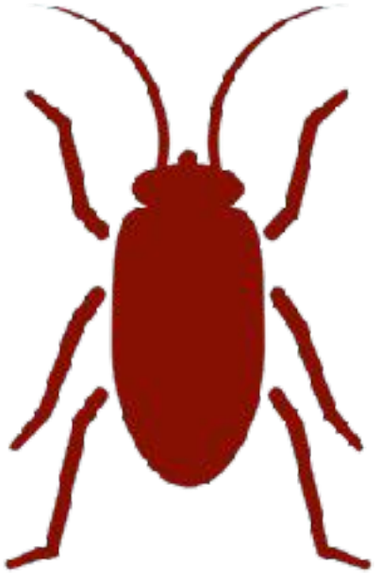
« we must learn to program an order of magnitude more effectively! »

«If you want more effective programmers, they should not waste their time debugging, they should not introduce the bugs to start with.»



« The programmer should let correctness proof and program grow hand in hand. »

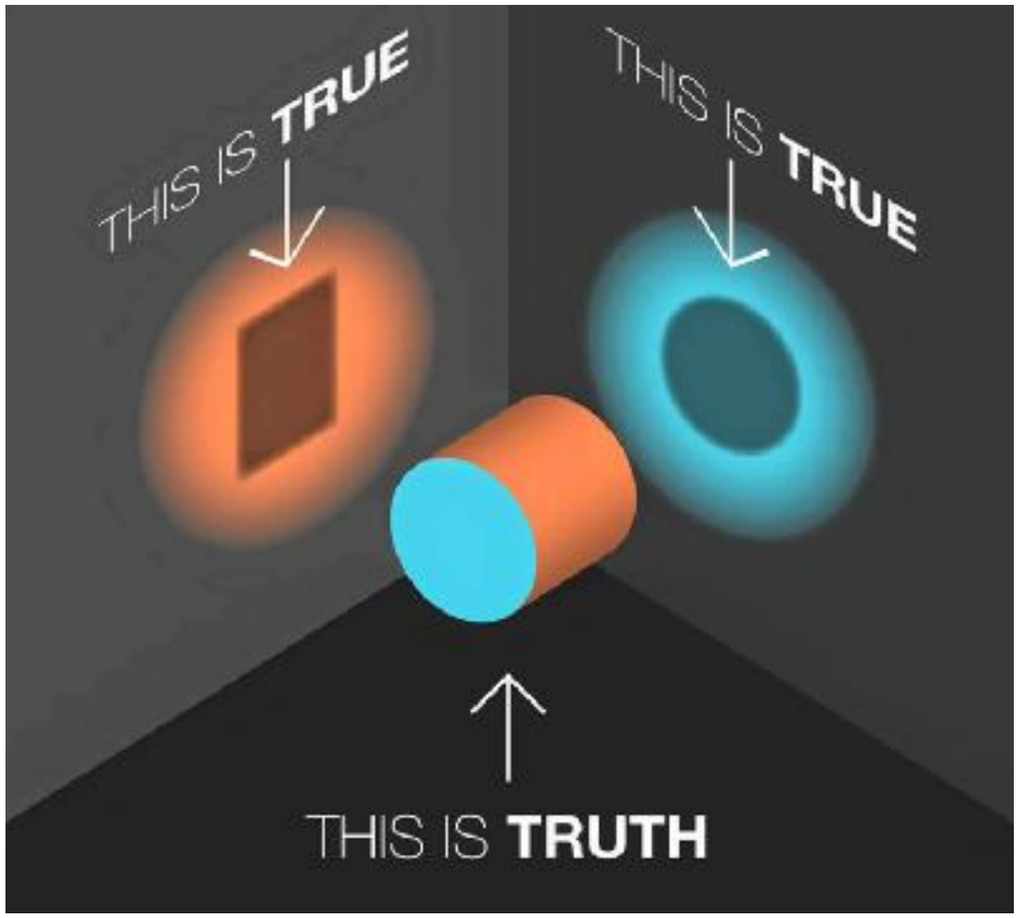
testing before



feedback loops

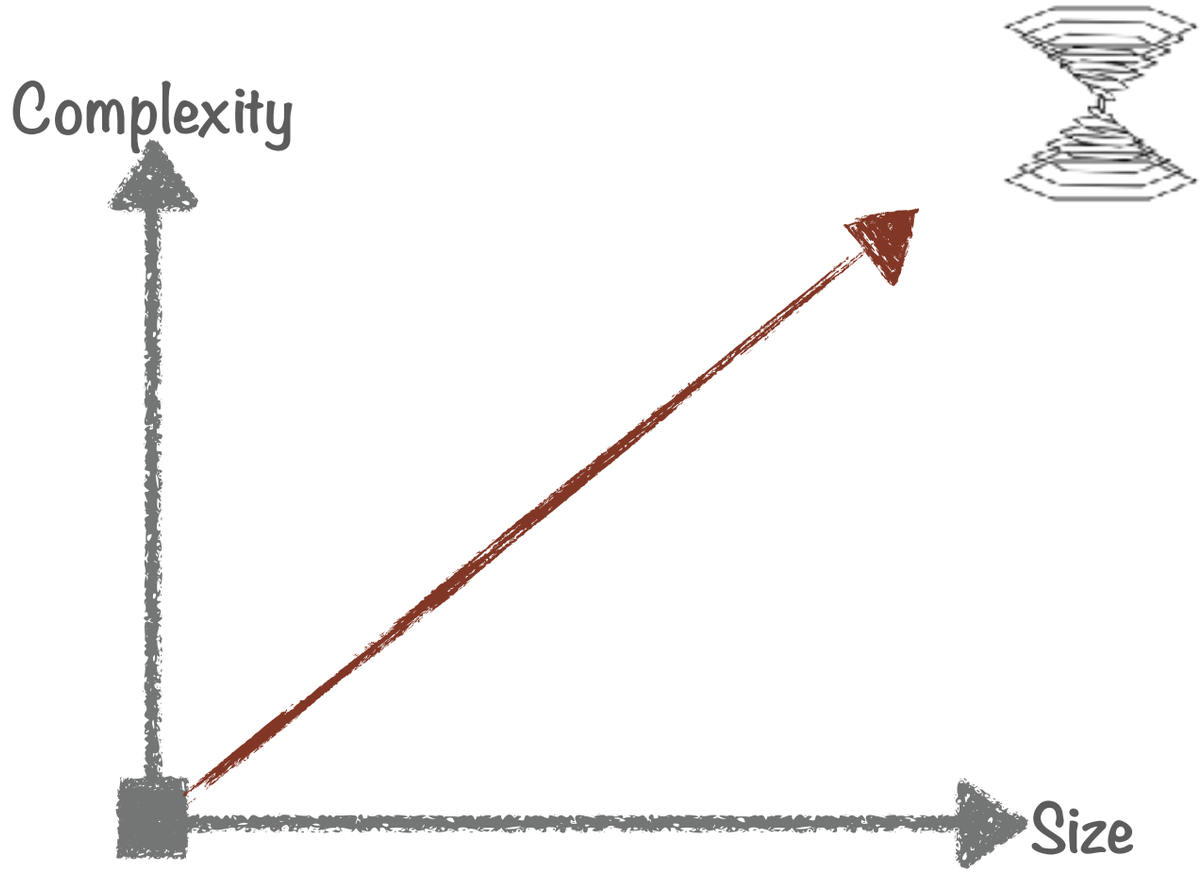
incremental growth

more/better abstractions?



« The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.»

Simplicity!



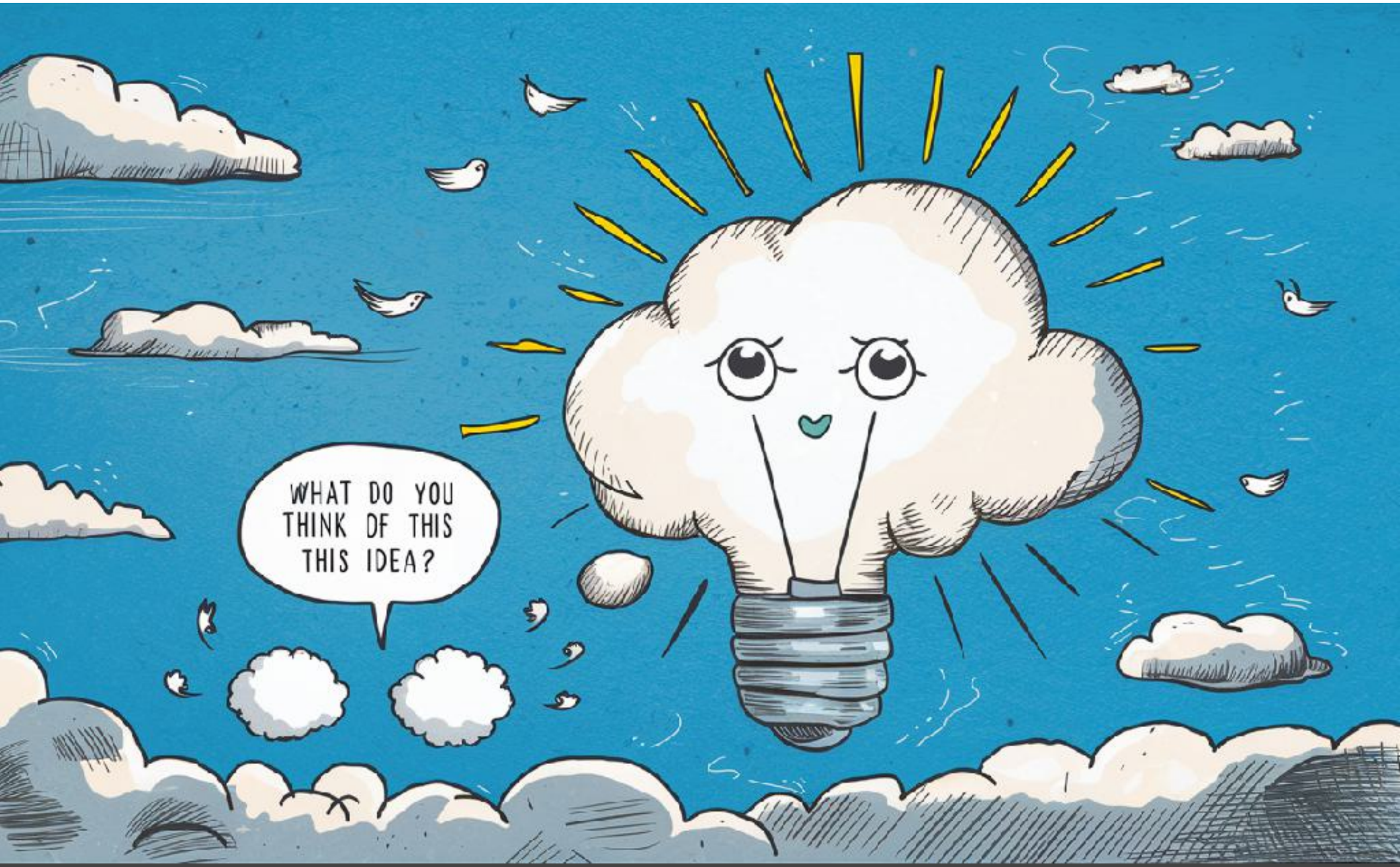
« the intellectual effort needed to conceive or to understand a program need not grow more than proportional to program length »

The Humble Programmers manifesto

- provided that we approach the task with a full appreciation of its tremendous difficulty,
- provided that we stick to modest and elegant programming languages,
- provided that we respect the intrinsic limitations of the human mind,
- and approach the task as Very Humble Programmers

We shall do a much better programming job

The Humble Programmers manifesto



WELL...

THESE WORDS ARE NOT MINE

all this …

« As long as there were no machines, programming is no problem at all »

« Only a few years ago, the talk about a software crisis was blasphemy. .. and by now, it is generally recognised that the design of any large sophisticated system is going to be a very difficult job. »

« If software were to continue to be the same clumsy and expensive process as it is now, things would get completely out of the balance »

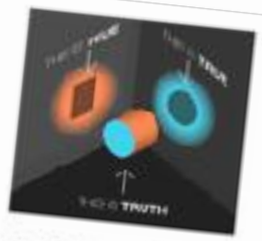
$$$

« we must learn to program an order of magnitude more effectively! »

«If you want more effective programmers, they should not waste their time debugging, they should not introduce the bugs to start with.»

« The programmer should let correctness proof and program grow hand in hand. »

more/better abstractions?



« The purpose of abstracting is not to be vague, but to create a new semantic level in which one can be absolutely precise.»

« the intellectual effort needed to conceive or to understand a program need not grow more than proportional to program length »

The Humble Programmers manifesto

«

- provided that we approach the task with a full appreciation of its tremendous difficulty,
- provided that we stick to modest and elegant programming languages,
- provided that we respect the intrinsic limitations of the human mind,
- and approach the task as **Very Humble Programmers**

We shall do a much better programming job »

ACM Turing Lecture 1972 EWD340 - 0

The Humble Programmer.

by

Edsger W.Dijkstra

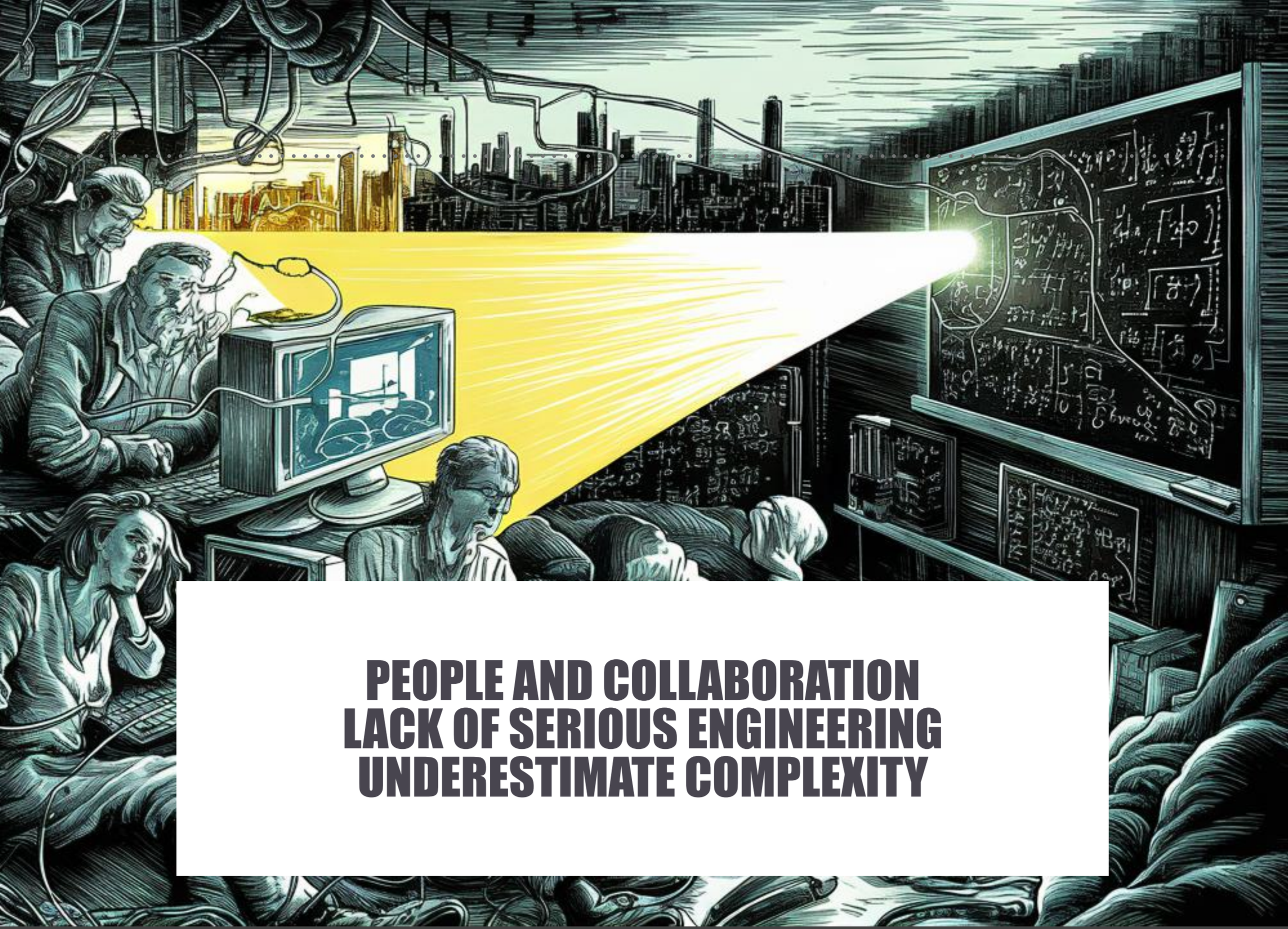
As a result of a long sequence of coincidences I entered the programming profession officially on the first spring morning of 1952 and as far as I have been able to trace, I was the first Dutchman to do so in my country. In retrospect the most amazing thing was the slowness with which, at least in my part of the world, the programming profession emerged, a slowness

Dijkstra / 1972

SO WHAT ?...

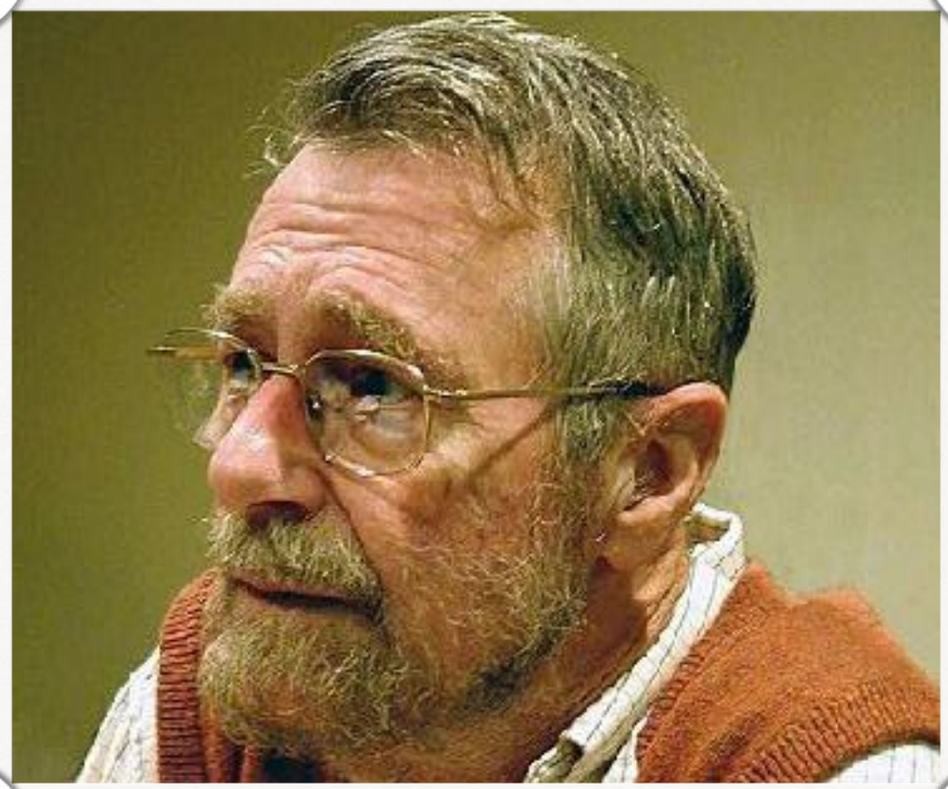
THE PROBLEMS OF TODAY ARE THE SAME AS 50 YEARS AGO WHEREAS THE COMPUTE POWER IS 10^10 TIMES HIGHER

SO, WHERE IS THE PROBLEM ?



PEOPLE AND COLLABORATION
LACK OF SERIOUS ENGINEERING
UNDERESTIMATE COMPLEXITY

EDSGER W. DIJKSTRA



Edsger Wybe Dijkstra

born 11 May 1930, died 6 august 2002

Dutch systems scientist, programmer, software engineer, science essayist, and pioneer in computing science

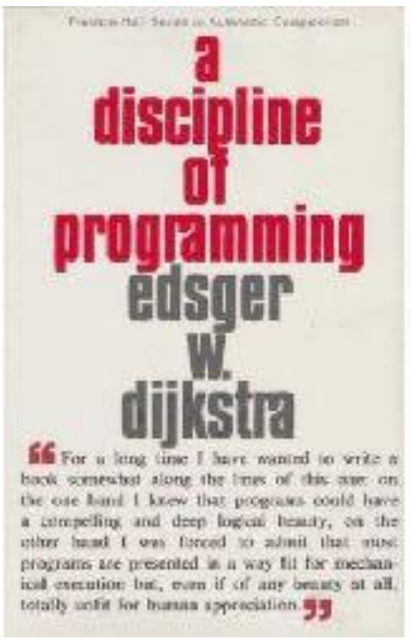
KNOWN FOR

First professional dutch programmer in 1957

Well known for « Goto statement considered harmful » in 1968

The humble programmer in 1972

Dijkstra Algorithm (shortest path between nodes)

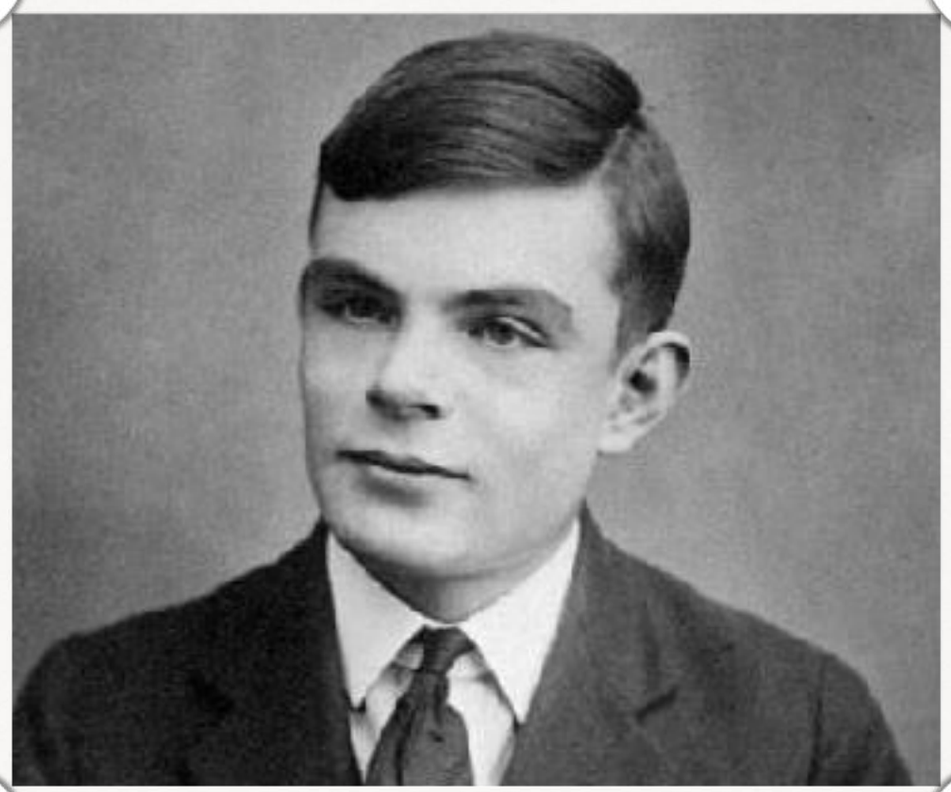


"Testing shows the presence, not the absence, of bugs."

- Edsger Wybe Dijkstra

ALAN TURING

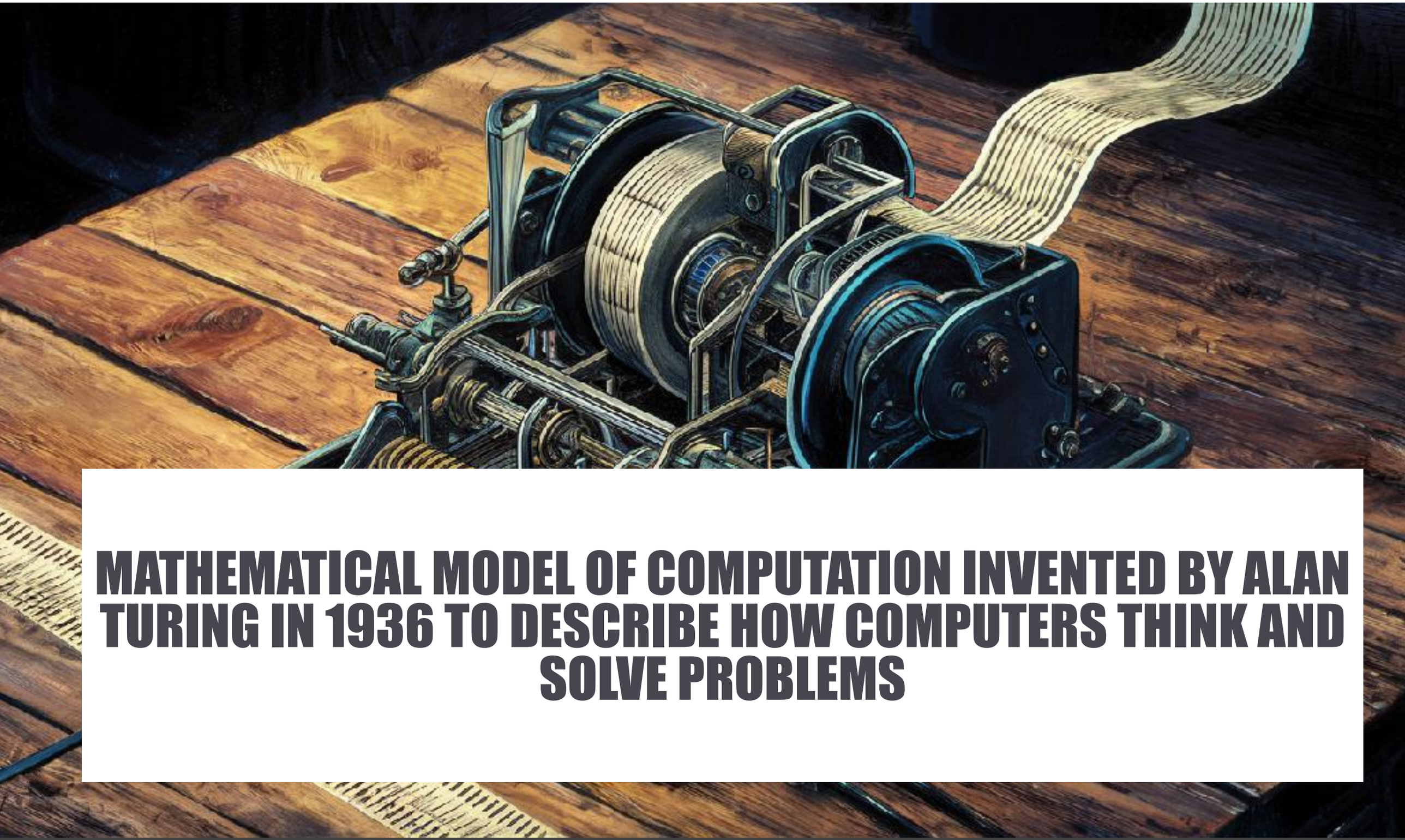
Alan Mathison Turing
born 23 June 1912, died 7 June 1954
English mathematician, computer scientist, logician, cryptanalyst, philosopher, and theoretical biologist.



KNOWN FOR

formalisation of the concepts of algorithm and computation with the Turing machine, father of theoretical computer science and artificial intelligence
Broke German cyphers with Enigma Machine
Automatic Computing Engine (one of the first designs for a stored-program computer)

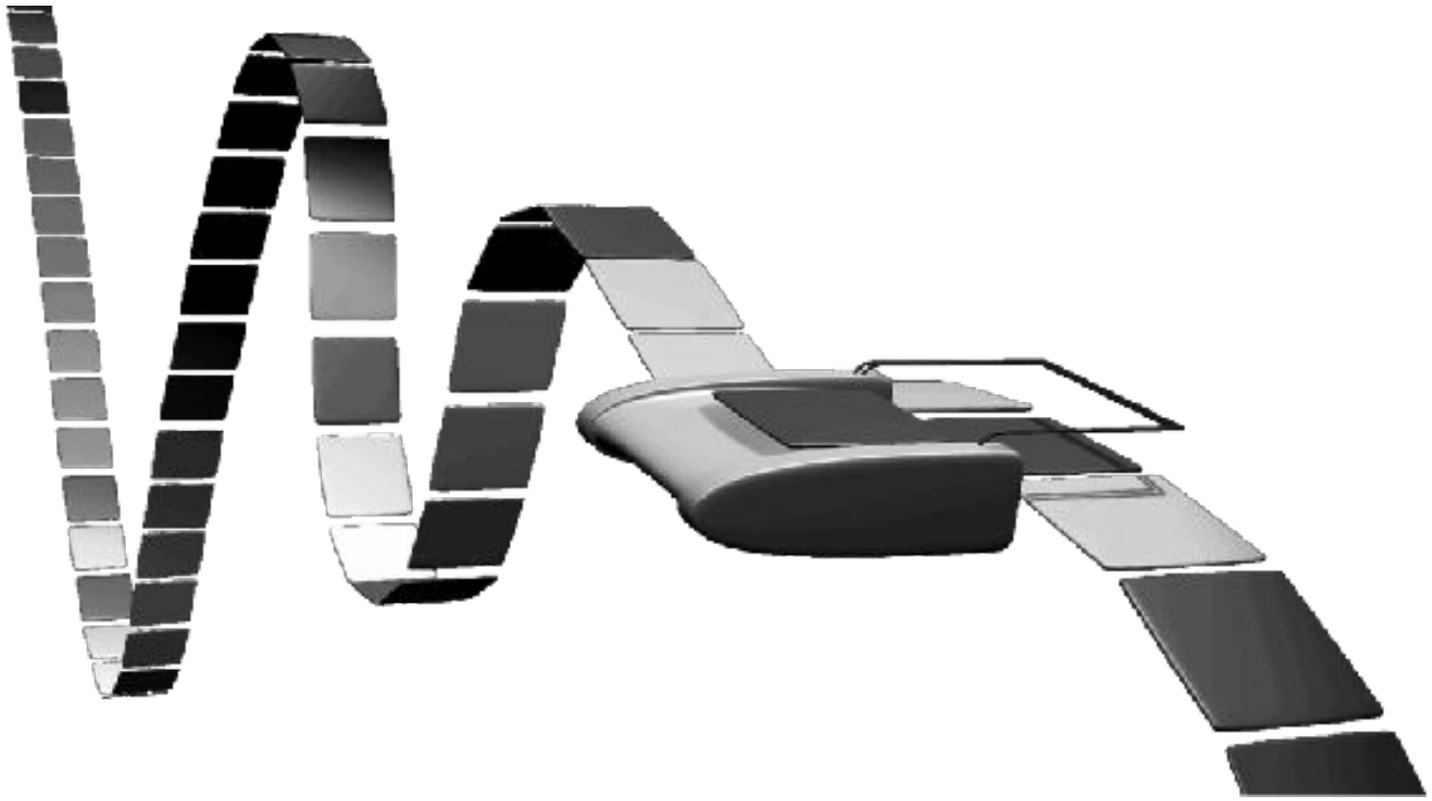
TURING MACHINE



MATHEMATICAL MODEL OF COMPUTATION INVENTED BY ALAN TURING IN 1936 TO DESCRIBE HOW COMPUTERS THINK AND SOLVE PROBLEMS

TURING MACHINE

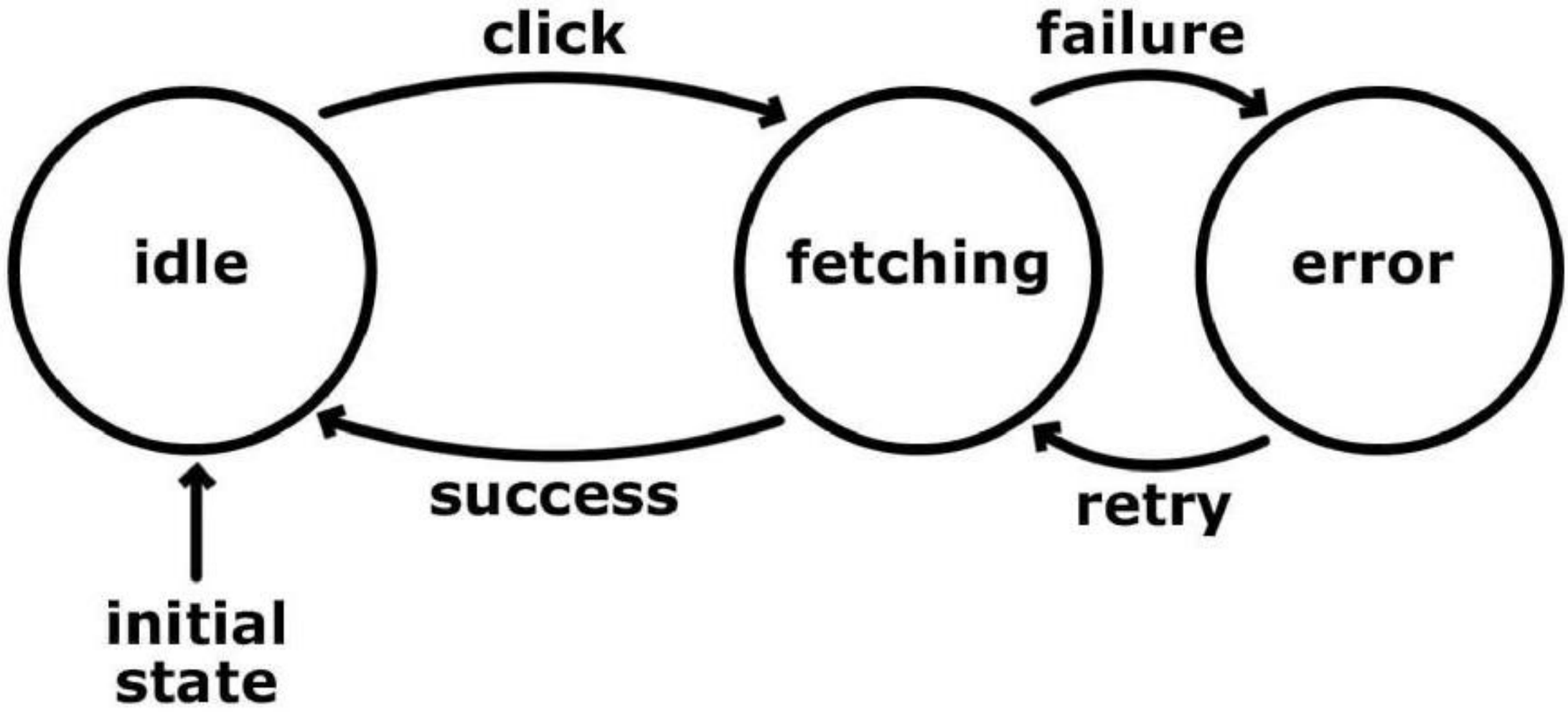
- AN INFINITE TAPE
- A HEAD
- A STATE REGISTER
- FINITE TABLE OF INSTRUCTIONS



- THE INFINITE TAPE HAS CELLS WITH ONE SINGLE VALUE EACH
- THE HEAD READS THE SYMBOL
- APPLY THE RULES OF STATE TRANSITION
  - CHANGE THE SYMBOL
  - MOVE LEFT OR RIGHT OR STAY
- CONTINUE

FINITE STATE MACHINE

(LESS POWERFUL THAN TURING MACHINE, NO MEMORY)



TURING COMPLETE

A programming language is "Turing complete" if it's powerful enough to simulate any algorithm that can be executed by a Turing machine

In order to be considered Turing complete, a programming language needs to have the ability to do the following:

1. Perform basic mathematical operations, such as addition, subtraction, multiplication, and division.
2. Use variables to store and manipulate data.
3. Use conditional statements and loops to control the flow of the program.
4. Have the ability to read and write data to and from a memory source, such as a hard drive.

Not Turing complete :

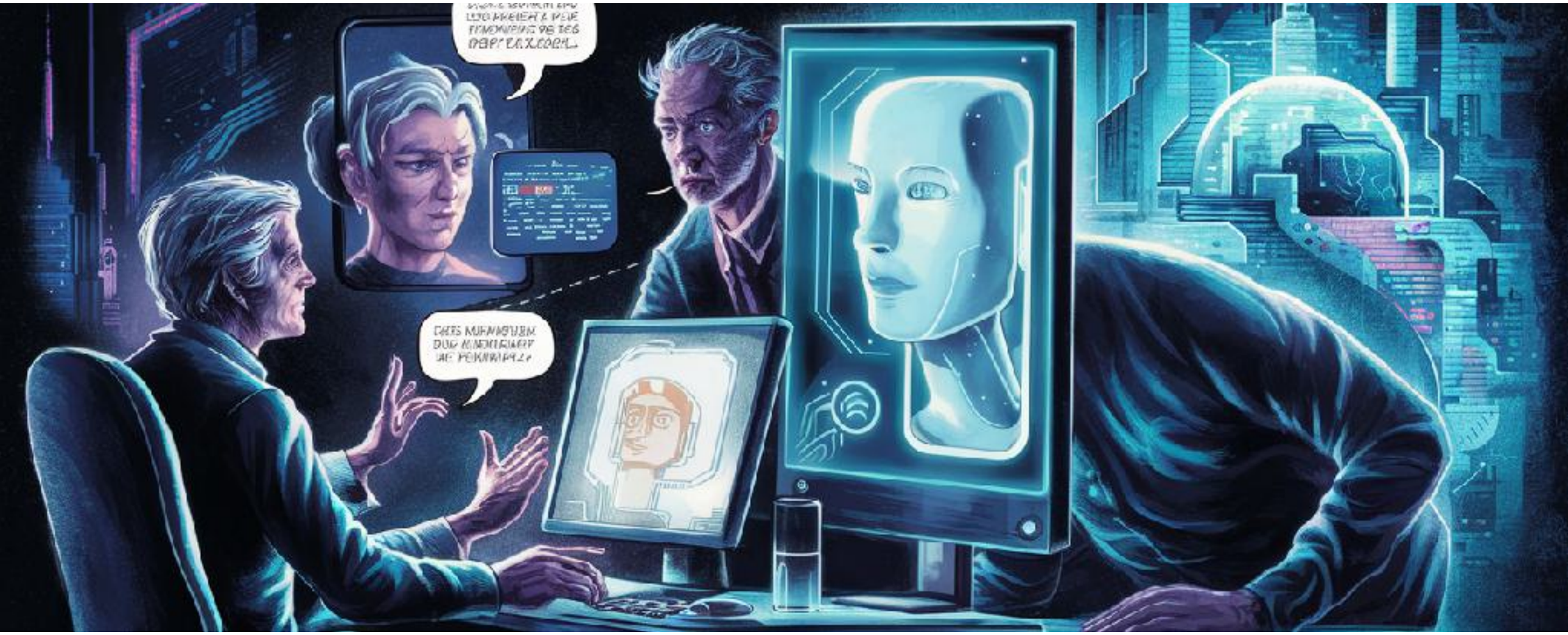
HTML, CSS, SQL, Yaml, Json...

Turing complete:

generic languages like c, python,...,
Game of life, Minecraft, Magic...

TURING TEST

a measure of a machine's ability to exhibit intelligent behaviour equivalent to, or indistinguishable from, that of a human



1950, paper ‘Computing Machinery and Intelligence’

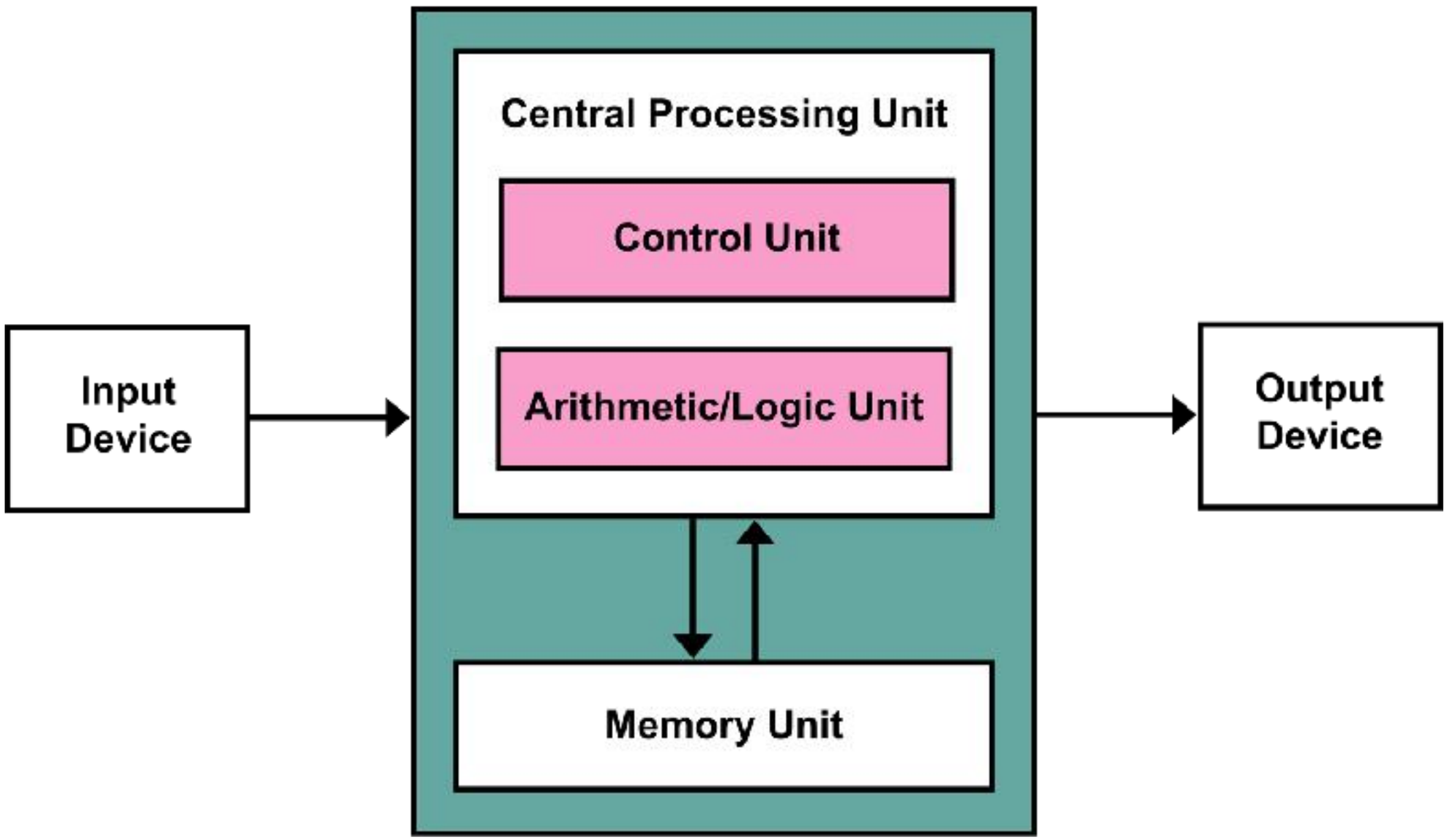
JOHN VON NEUMANN

Neumann János Lajos
born December 28, 1903,
died February 8, 1957
Hungarian-American mathematician, physicist,
computer scientist, and polymath

KNOWN FOR

Participation in the Manhattan Project, late 30's
Merge sort algorithm in 1945
The general and logical theory of automata
Philosophy of artificial intelligence
Many contributions to Maths, weather systems, physics

VON NEUMANN ARCHITECTURE

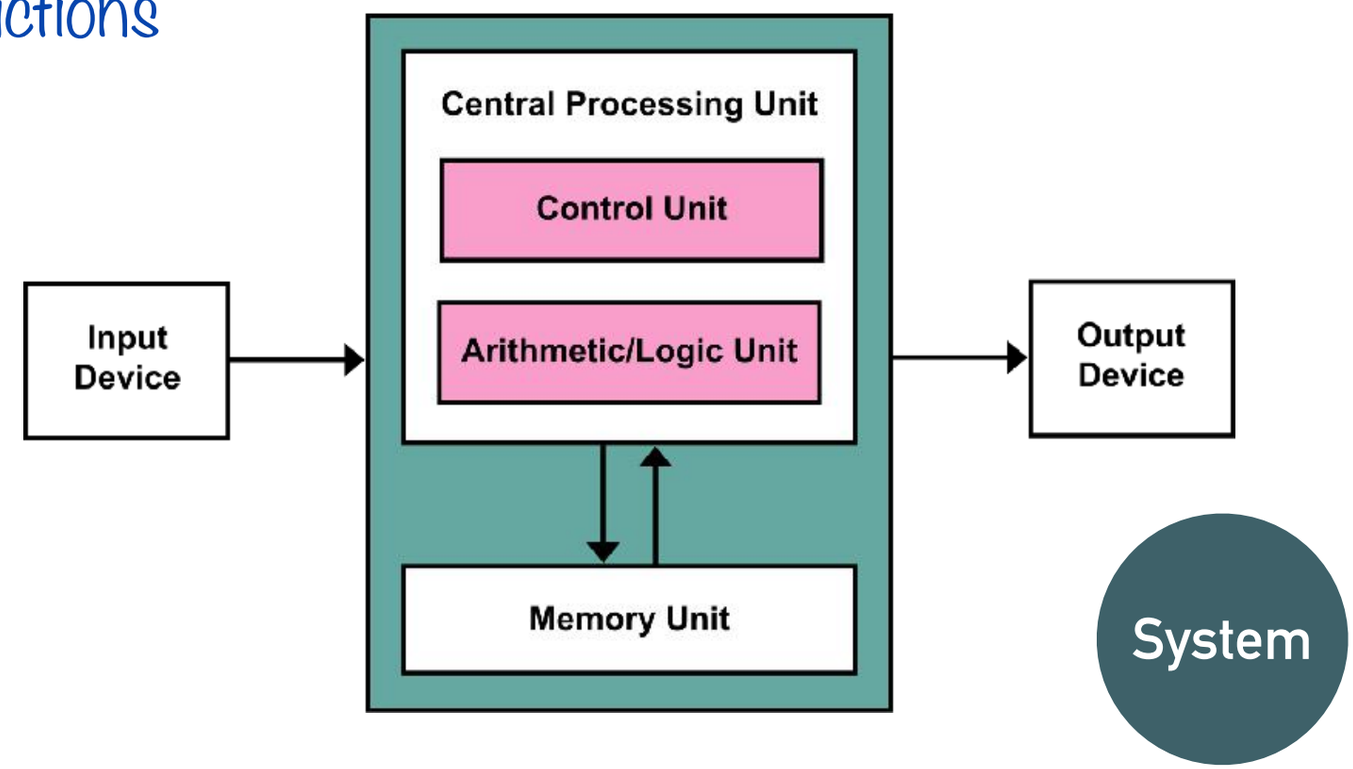


System

VON NEUMANN ARCHITECTURE

is a computer architecture based on a 1945 description by the mathematician and physicist John von Neumann that describes a design architecture for an electronic digital computer with these components:

- A processing unit that contains an arithmetic logic unit and processor registers
- A control unit that contains an instruction register and program counter
- Memory that stores data and instructions
- External mass storage
- Input and output mechanisms



WHAT HAVE WE LEARNED FROM HISTORY ?

WHAT DID WE LEARNT FROM HISTORY?

Programming is hard

Programs need structure

Tests are only known failures

Despite technical progress, the lack of developers create the same problems **today as 60 years ago**

Most « new » shiny stuff today was invented 10’s of years ago!

We’re still driven by Von Neuman architecture

WHAT DID WE LEARNT FROM HISTORY?

Architecture

Is

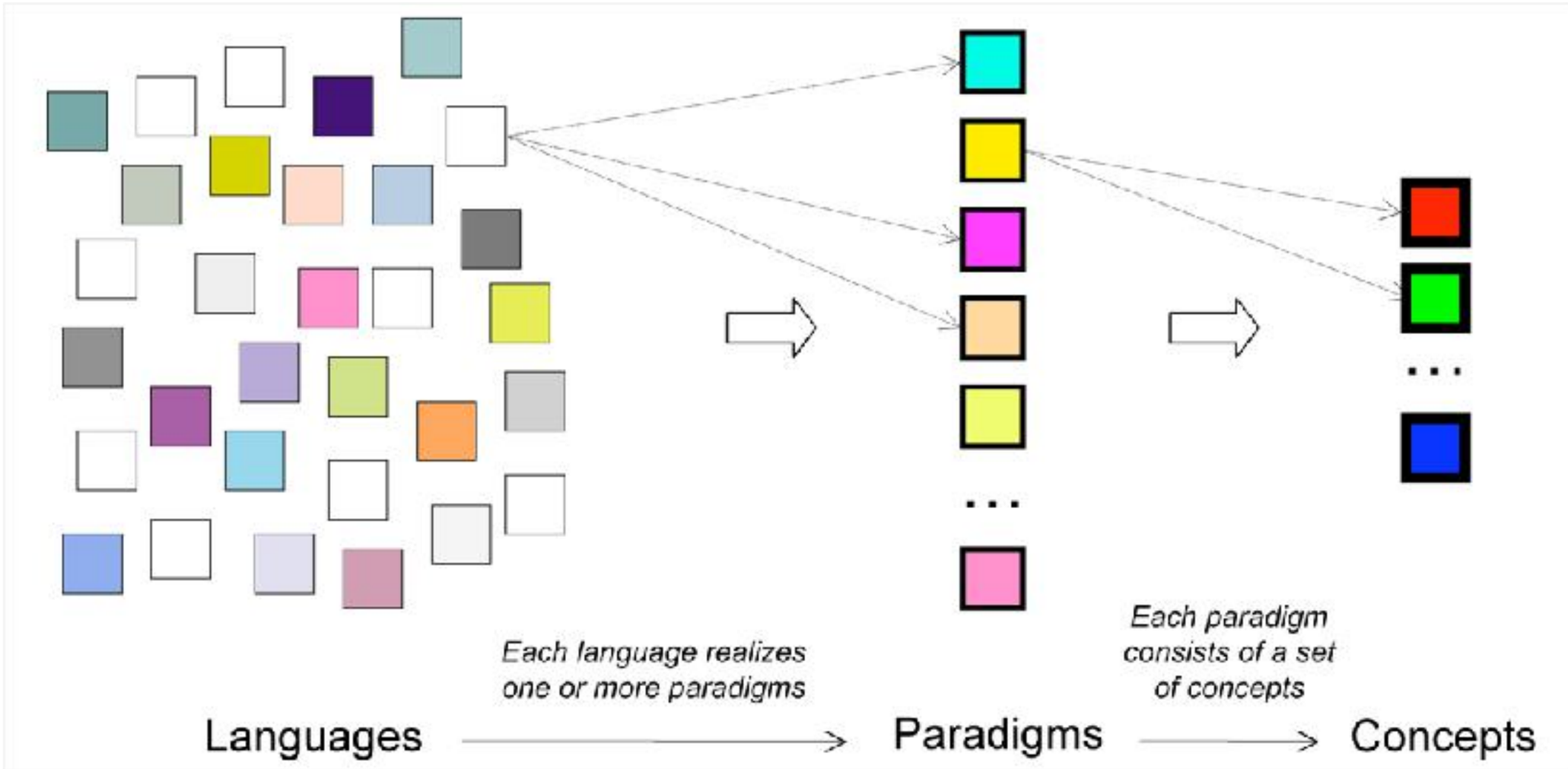
Structure with purpose

HISTORY II

Paradigms & why

BEFORE ARCHITECTURE COME THE PARADIGMS

LANGUAGES, PARADIGMS AND CONCEPTS



The medium used to interact with the machine

The Style or approach that adheres to a specific set of principles

The Foundational ideas and constructs

IMPERATIVE PROGRAMMING

We started as we speak,
one instruction after another

Assembly

Not invented by someone, it emerged as each machine has it's own version

(And the crapy code that works we write line by line)

PROCEDURAL PROGRAMMING

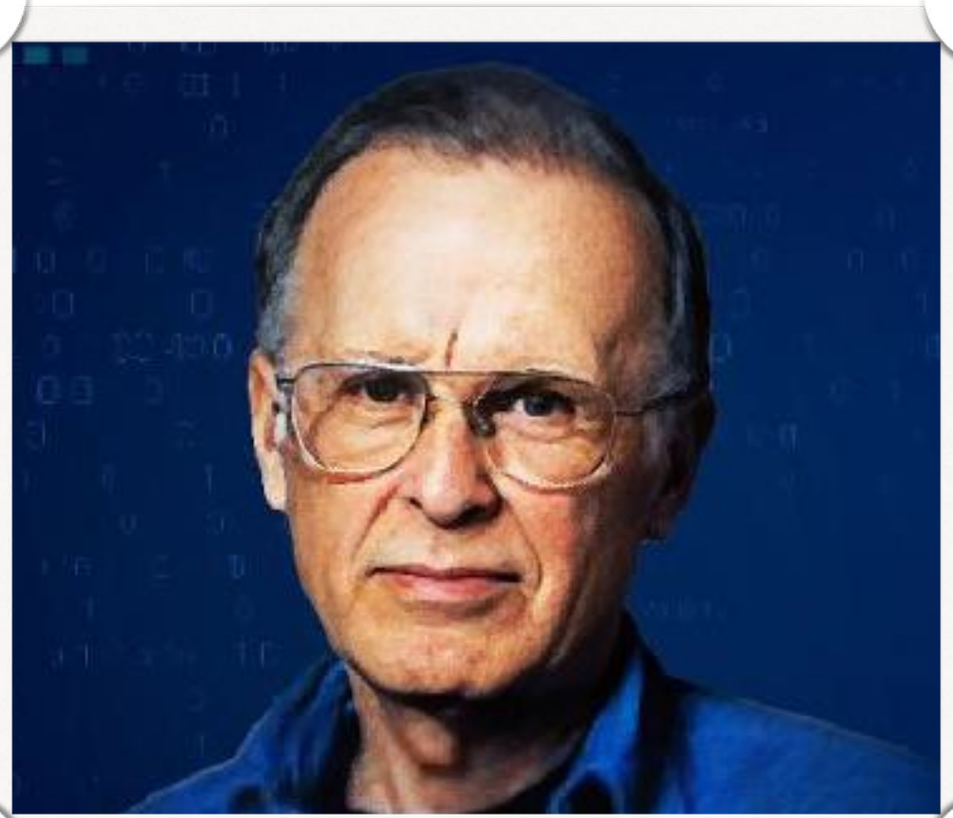
As programs evolved, we felt the need to have some abstractions, break complexity and create more human readable code

Fortran (1957)
(Backus)

Algol (1958)
(Backus and others)

JOHN BACKUS

John Warner Backus
born December 3, 1924,
died March 17, 2007
American computer scientist



KNOWN FOR

Created FORTRAN the first high-level programming language

Worked on ALGOL

Invented BNF (Backus-Naur form)

Received the Turing Award in 1977

```
<postal-address> ::= <name-part> <street-address> <zip-pa

     <name-part> ::= <personal-part> <last-name> <opt-suf
<personal-part> <name-part>

 <personal-part> ::= <first-name> | <initial> "."

<street-address> ::= <house-num> <street-name> <opt-apt-n

      <zip-part> ::= <town-name> "," <state-code> <ZIP-co
```

STRUCTURED PROGRAMMING

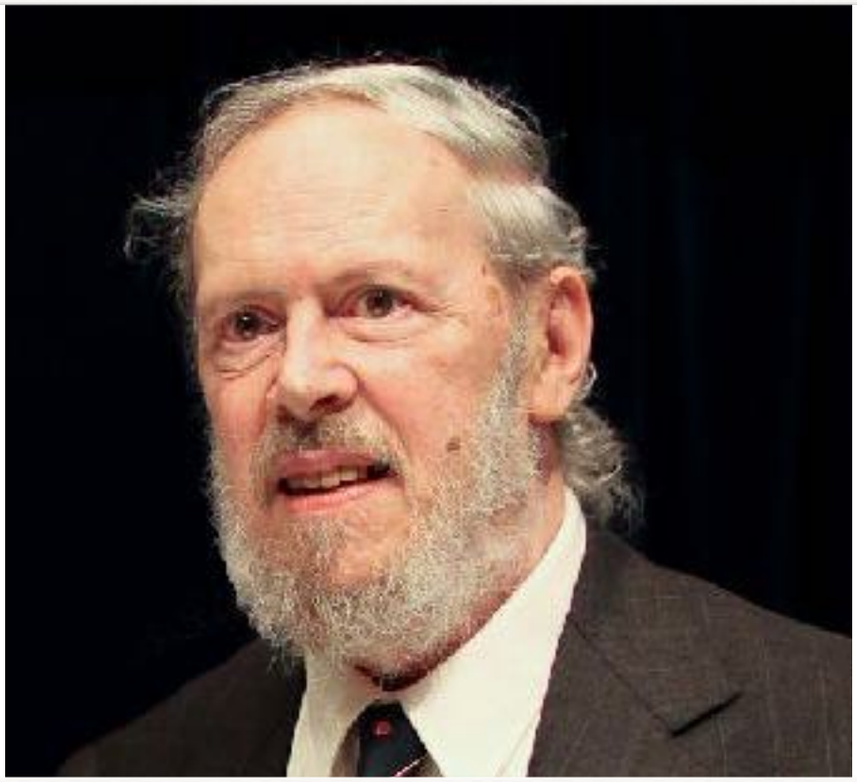
A clarification and improvement of procedural programming

ALGOL 60 (1960)
Backus and others

C (1972)
Dennis Ritchie

Pascal (1970)
Niklaus Wirth

DENNIS RITCHIE

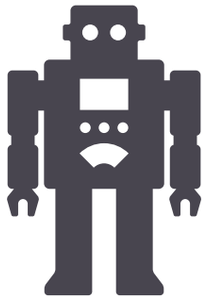


Dennis MacAlistair Ritchie
born September 9, 1941,
died October 12, 2011
American computer scientist

KNOWN FOR

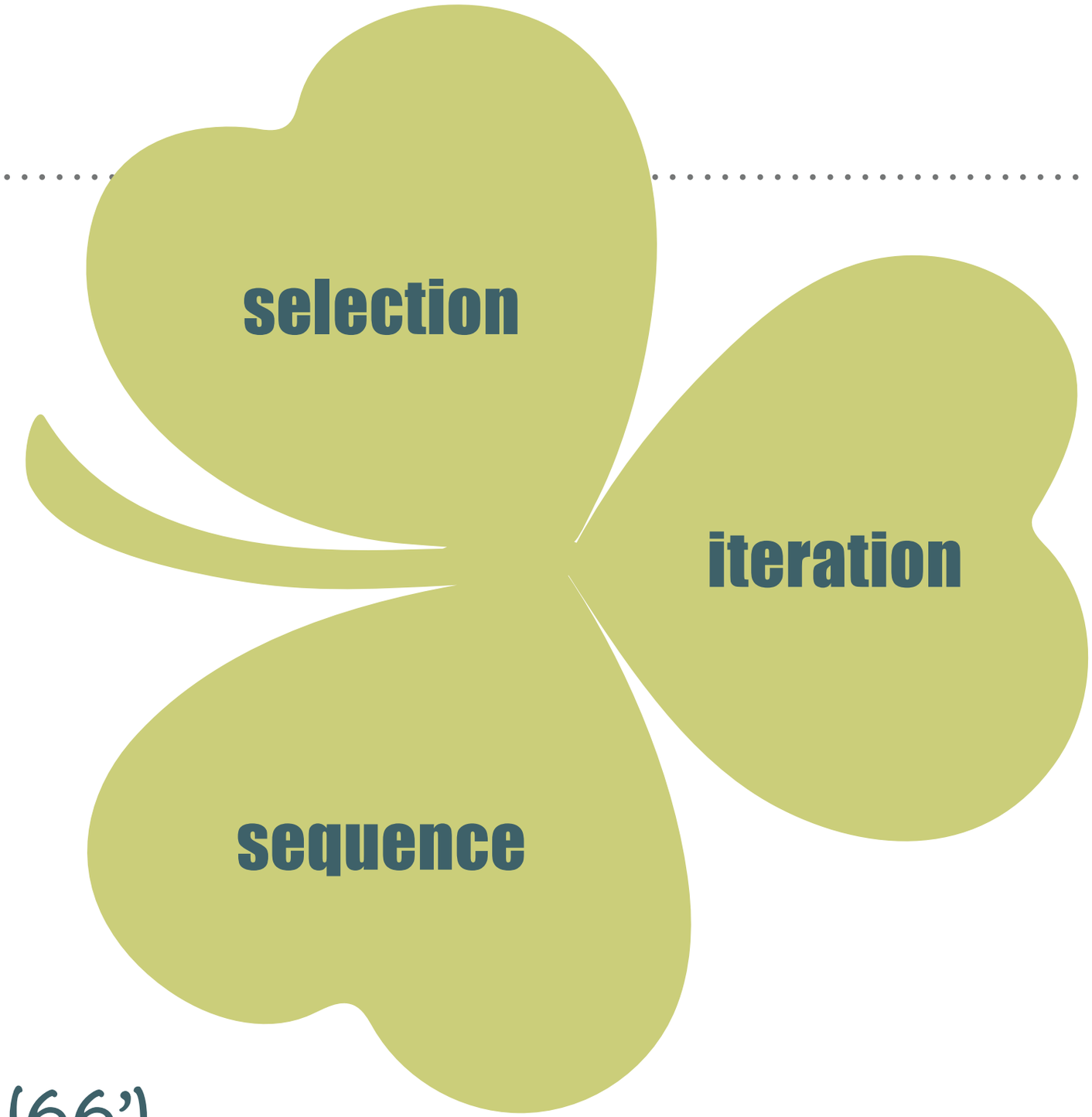
Created C programming language
Co-Created UNIX with Ken Thompson
Received the Turing Award in 1983

SOFTWARE



program

any computer program can be written using just three structures

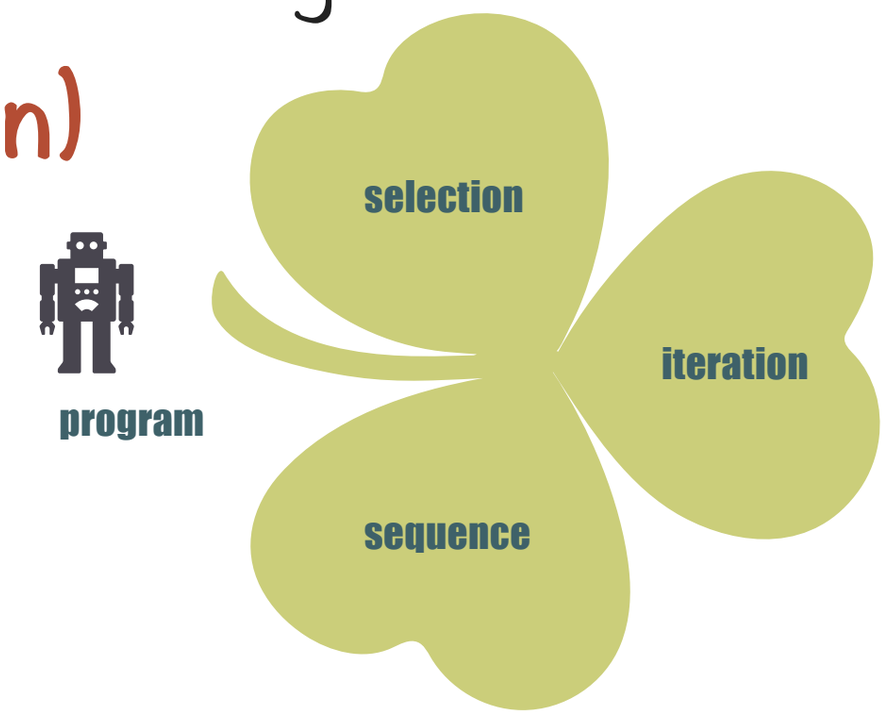


Böhm & Jacopini theorem (66')

→ Then popularised by Dijkstra,

SOFTWARE

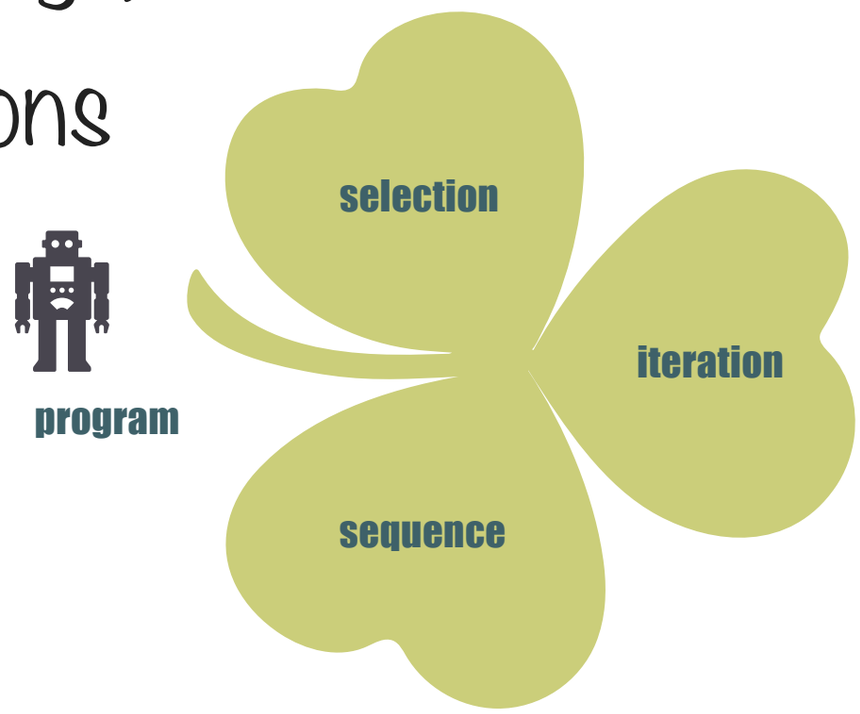
1. Executing one subprogram, and then another subprogram **(sequence)**
2. Executing one of two subprograms according to the value of a boolean expression **(selection)**
3. Repeatedly executing a subprogram as long as a boolean expression is true **(iteration)**



Böhm & Jacopini theorem (67')

STRUCTURED PROGRAMMING HOPE & LEARNINGS

1. Writing code one line after another like imperative is not enough
2. Programming is hard, so we need to break it in smaller pieces
3. But breaking in procedures is not enough, we need the manage the flow of control and iterations



OBJECT ORIENTED PROGRAMMING

It imposes discipline on indirect transfer of control

SIMULA 67 **Smalltalk** **Java***

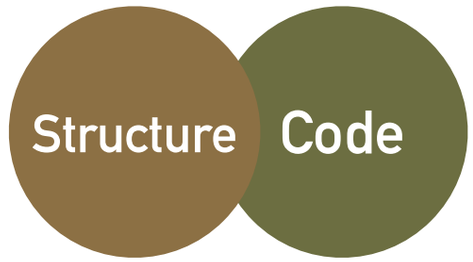
OO WAS INVENTED AROUND 1966 WHEN DAHL & NYGAARD MOVED THE FUNCTION CALL STACK FRAME TO THE HEAP

(DURING THEIR WORK ON ALGOL THEN SIMULA)

THE GOAL OF SIMULA WAS TO REFLECT THE END USER WORLD MODEL IN THE SYSTEM DESIGN

That key principle has been lost in decades of methodology and programming language obfuscation

OBJECT PROGRAMMING HOPE AND LEARNINGS



1. Break programs into smaller pieces is not enough, these pieces should have meaningful properties and behaviours
2. An object is a black box for the others and only communicates on what's necessary
3. It will be hard to simulate the world with objects
4. Actual POO is far from original idea

FUNCTIONAL PROGRAMMING

It imposes discipline upon assignment

Lisp (1958)

John McCarthy

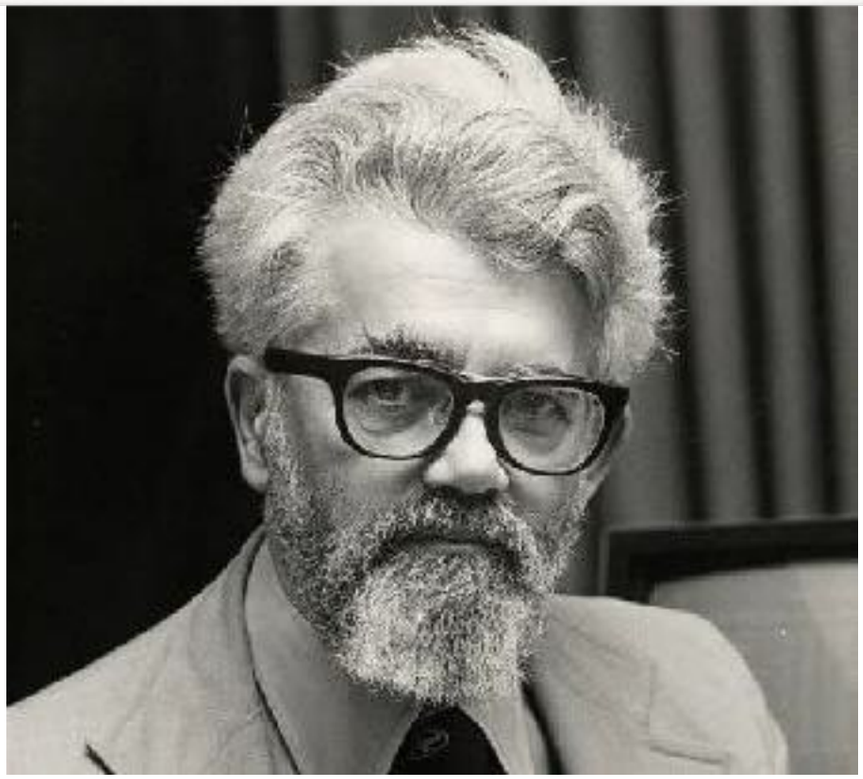
ML (1973)

Haskell (1990)

IS THE DIRECT RESULT OF THE WORK OF ALONZO CHURCH, WHO IN 1936 INVENTED LAMBDA CALCULUS

LAMBDA CALCULUS IS THE FOUNDATION OF THE LISP LANGUAGE, INVENTED BY JOHN MCCARTHY IN 1958

JOHN MCCARTHY

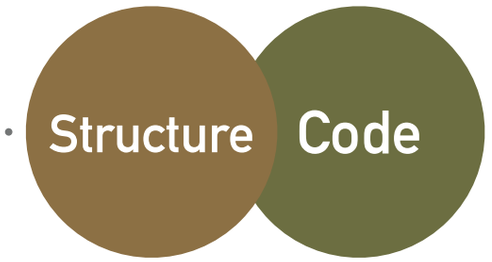


John McCarthy
born September 4, 1927,
died October 24, 2011
American computer scientist

KNOWN FOR

Created LISP in 1958
Invented « garbage collection »
Coined the term « Artificial Intelligence » and founded the discipline
Received the Turing Award in 1971

FUNCTIONAL HOPE AND LEARNINGS



1. If we think about small units of computation as mathematical functions it will be easier to ensure it's correct and to compose
2. We shouldn't tell the computer how to do the things but what we want
3. Powerful but simple concepts have been proven helpful to produce good software: simplicity of functions, immutability, functions as variables, purity and no side effects, strong type system

SO WHAT ?

WHAT WE LEARNT IN THE LAST HALF CENTURY IS NOT WHAT SOFTWARE DESIGN IS, BUT ONLY WHAT NOT TO DO

There are two ways of constructing a software design:

One way is to make it so ***simple*** *that there are* ***obviously no deficiencies***

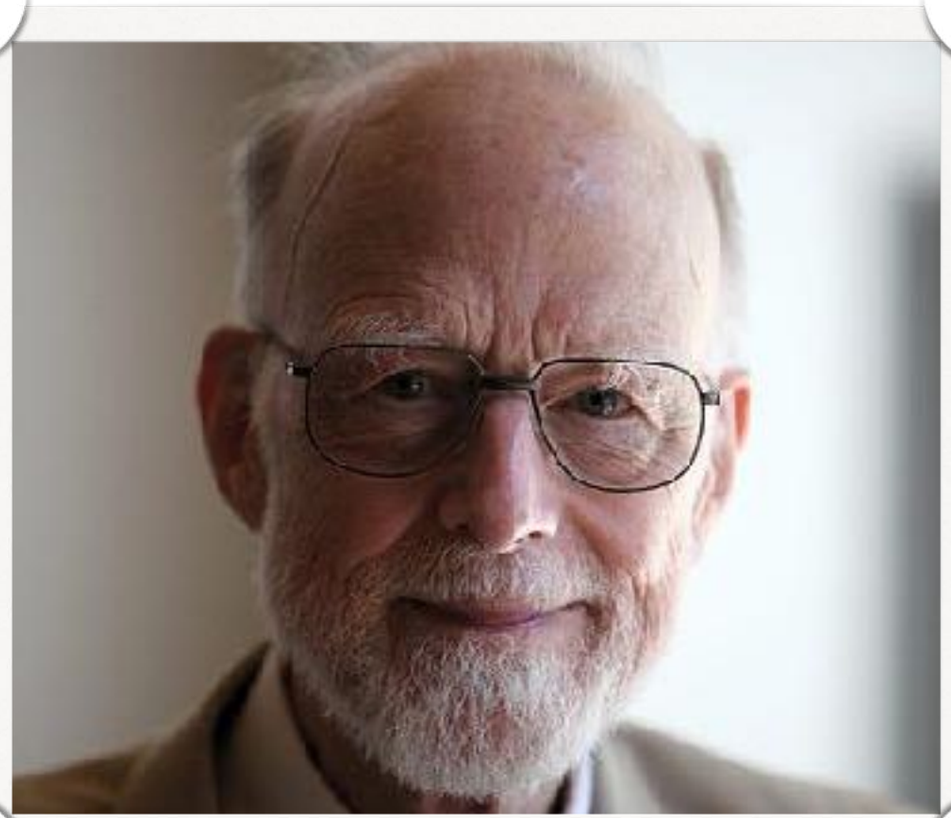
and the other is to make it so ***complicated*** *that there are* ***no obvious deficiencies****.*

-Tony Hoare

1991, Turing Award Lecture

TONY HOARE

Sir Charles Antony Richard Hoare
born 11 January 1934
British computer scientist.



KNOWN FOR

Sorting algorithm quicksort in 1959/1960.
Hoare logic for verifying program correctness,
Formal language communicating sequential processes (CSP) to specify the interactions of concurrent processes
Inspiration for the occam programming language

WHAT DID WE LEARN ABOUT PARADIGMS ?

The process of isolation, called abstraction, is emerging as a principal activity of programmers

–Kent Beck

PROGRAMS NEED STRUCTURE ALLOWING PREDICTABLE OUTPUT

```
START
  1. DO A
  2. Output X
  3. Do B
  4. Input Y
  5. Go to 7
  6. Do C
  7. Go to 5
  8. Output ‘lost in time’
END
```

STYLES OF PROGRAMMING AND TYPES MAKE DIFFERENCE

```
//js style, dynamic


function isAdult(i) {

   if(isNaN(i)) {

      throw ‘Parameter is not a
      number!’;

      }

   if(i < 0 || I > 120) {

       throw ‘age should be a
      number between 0 and 120’;

       }

   if(i>18) {

        return ‘adult’;

        } else {

            return ‘not an adult’;

        }

}
```

```
//c#-java style


string IsAdult(int age) {

      if(age < 0 || age > 120) {

        throw ‘age should be a
      number between 0 and 120’;

        }

      if(age>18) {

         return ‘adult’;

         } else {

             return ‘not an adult’;

         }

}
```

```
//strong typed language

Type Age : int [0..120]

string IsAdult(Age age) {

      if(age>18) {

         return ‘adult’;

         } else {

             return ‘not an adult’;

         }

}
```

STYLES OF PROGRAMMING AND TYPES MAKE DIFFERENCE

```
//js style, dynamic


function isAdult(i) {

   if(isNaN(i)) {

      throw ‘Parameter is not a
      number!’;

      }

   if(i < 0 || I > 120) {

      throw ‘age should be a
     number between 0 and 120’;

      }

  if(i>18) {

        return ‘adult’;

        } else {

            return ‘not an adult’;

        }
}
```

```
//c#-java style


string IsAdult(int age) {

      if(age < 0 || age > 120) {

        throw ‘age should be a
      number between 0 and 120’;

        }

    if(age>18) {

       return ‘adult’;

       } else {

           return ‘not an adult’;

       }
}
```

```
//strong typed language

Type Age : int [0..120]

string IsAdult(Age age) {

    if(age>18) {

       return ‘adult’;

       } else {

           return ‘not an adult’;

       }
}
```

Business code

STYLES OF PROGRAMMING AND TYPES MAKE DIFFERENCE

```
//js style, dynamic

function isAdult(i) {
   if(isNaN(i)) {
      throw ‘Parameter is not a
      number!’;
      }
   if(i < 0 || I > 120) {
      throw ‘age should be a
     number between 0 and 120’;
      }
   if(i>18) {
        return ‘adult’;
        } else {
            return ‘not an adult’;
        }
}
```

```
//c#-java style

string IsAdult(int age) {
    if(age < 0 || age > 120) {
      throw ‘age should be a
     number between 0 and 120’;
      }
    if(age>18) {
       return ‘adult’;
       } else {
           return ‘not an adult’;
       }
}
```

```
//strong typed language

Type Age : int [0..120]

string IsAdult(Age age) {
    if(age>18) {
       return ‘adult’;
       } else {
           return ‘not an adult’;
       }
}
```

Plumbery

Business code

RECAP

- A program needs a **structure**
- Behaviour should be **encapsulated** and **isolated**
- Stronger types gives less flexibility but enforces correctness
- Whatever the paradigms, **Naming** is the key

ARCHITECTURE, WHAT ?

WHAT IS AN ARCHITECT ?

WHAT IS AN ARCHITECT ?

SOMEONE WHO CREATES ARCHITECTURE

> *the shared understanding that the expert developers have of the system design.*
>
> **-Ralph Johnson**

"the decisions you wish you could get right early in a project"

-Ralph Johnson

"For a **developer** to become an **architect**, they need to be able to recognize what elements are important, recognizing what elements are likely to result in serious problems should they not be controlled."

-Martin Fowler

MARTIN FOWLER

born 1963,
British software developer, author and international public speaker on software development.

KNOWN FOR

helped create the Manifesto for Agile Software Development
Refactoring: Improving the Design of Existing Code book in 1999
Planning Extreme Programming book in 2000
Patterns of Enterprise Application Architecture in 2002
martinfowler.com bliki

SOFTWARE ARCHITECTURE STARTED AS FRED BROOKS'S VISION OF A GOOD METAPHOR FOR HOW WE DO SOFTWARE

FRED BROOKS

Frederick Phillips Brooks Jr
born 1931, died 2022
Computer Scientist, Software Engineer, and Author

KNOWN FOR

Led the development of IBM's System/360 family of computers
Managed the development of IBM's OS/360 operating system
"The Mythical Man-Month: Essays on Software Engineering" (1975)
"No Silver Bullet: Essence and Accidents of Software Engineering" (1986)

Adding manpower to a late software project makes it later.

- Fred Brooks

> *Nine women can't make a baby in one month.*

- Fred Brooks

WRITING CODE THAT WORKS LINE AFTER LINE IS NOT PROFESSIONAL

YOU NEED TO THINK AND MAKE DECISIONS ABOUT FORM AND STRUCTURE TO START SAYING YOU'RE ARCHITECTURING AN APP

EVERY LINE OF CODE YOU WRITE IS A DESIGN DECISION THAT IMPACT THE FUTURE OF THE PROJECT

ARCHITECTURE, WHY ?

OUR IDEA OF GOOD DESIGN IS A MIXED FEELING ABOUT OUR OWN MENTAL MODEL, AESTHETICS, THEORY, THE REAL WORLD

BY DEFINITION, IT'S HARD ...

MENTAL MODEL, — Totally subjective & personal

AESTHETICS, — Totally subjective & personal

THEORY, — Limited & complex

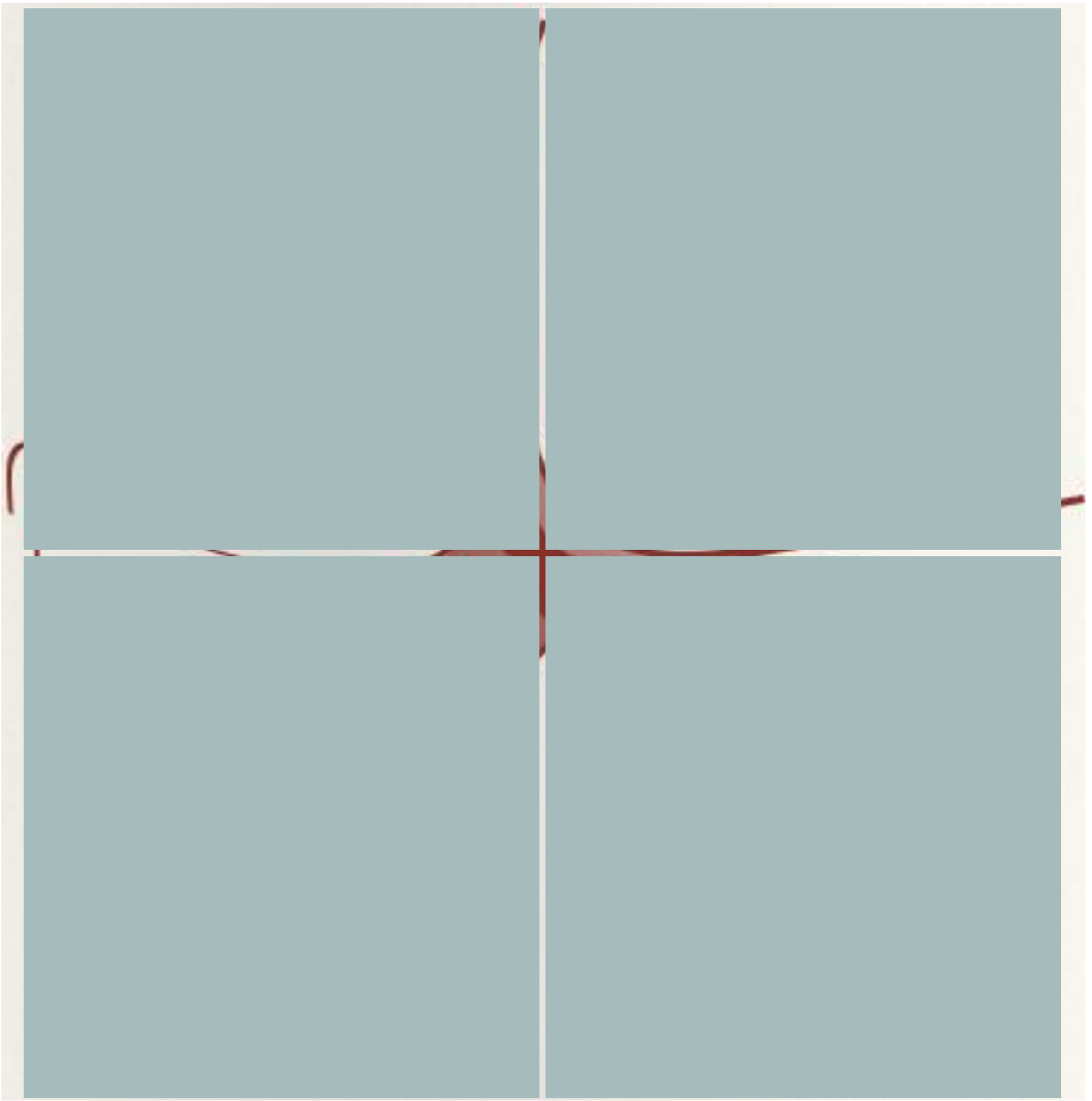
THE REAL WORLD — Infinite & complex

KEEPING OUR PROGRAMS UNDERSTANDABLE AND OUT OF THE CHAOS IS ALWAYS COMPLICATED, OFTEN COMPLEX, ALMOST NEVER SIMPLE

But what does these words really mean ?...

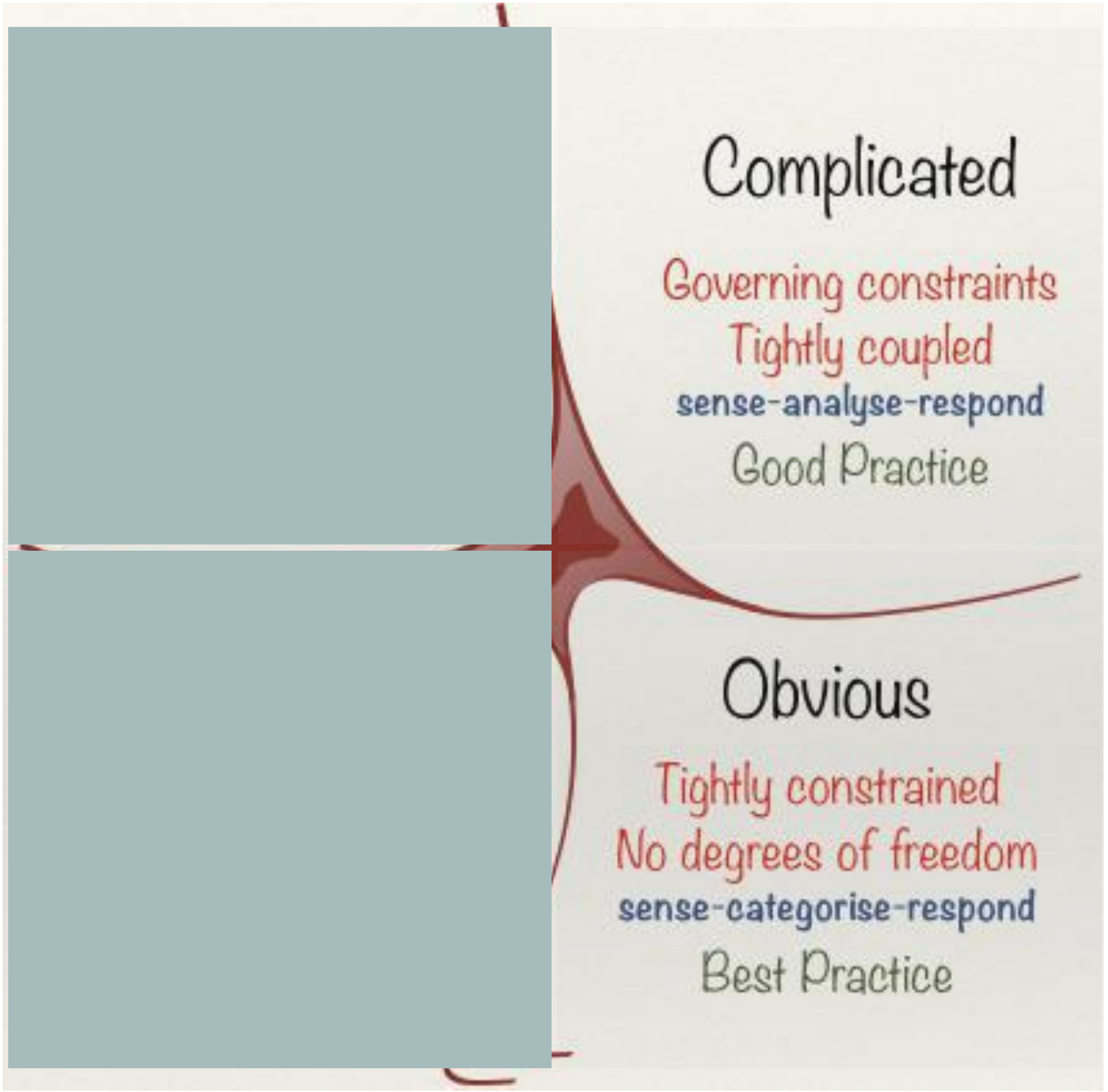
COMPLEXITY

Cynefin framework

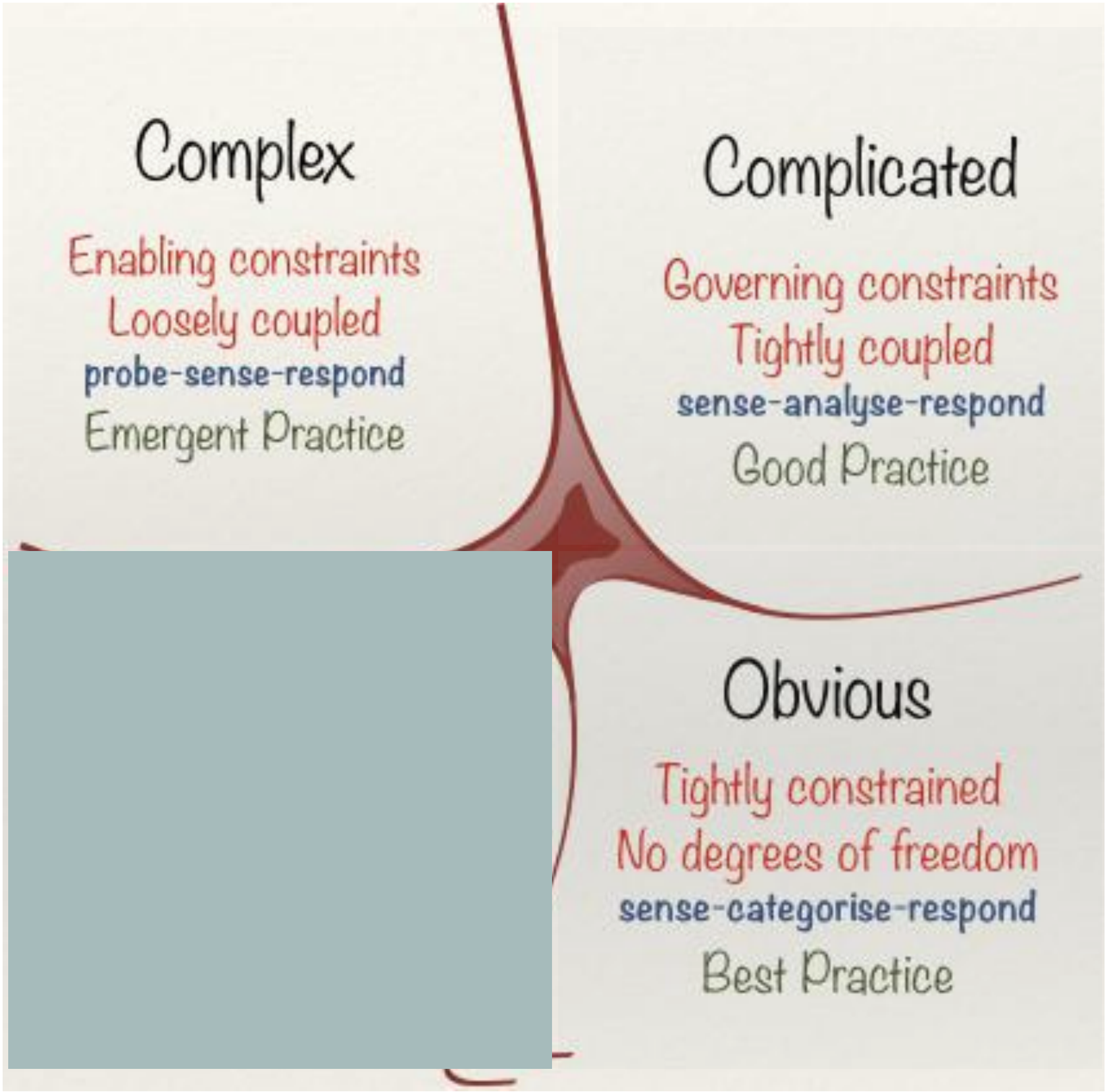


Cynefin framework

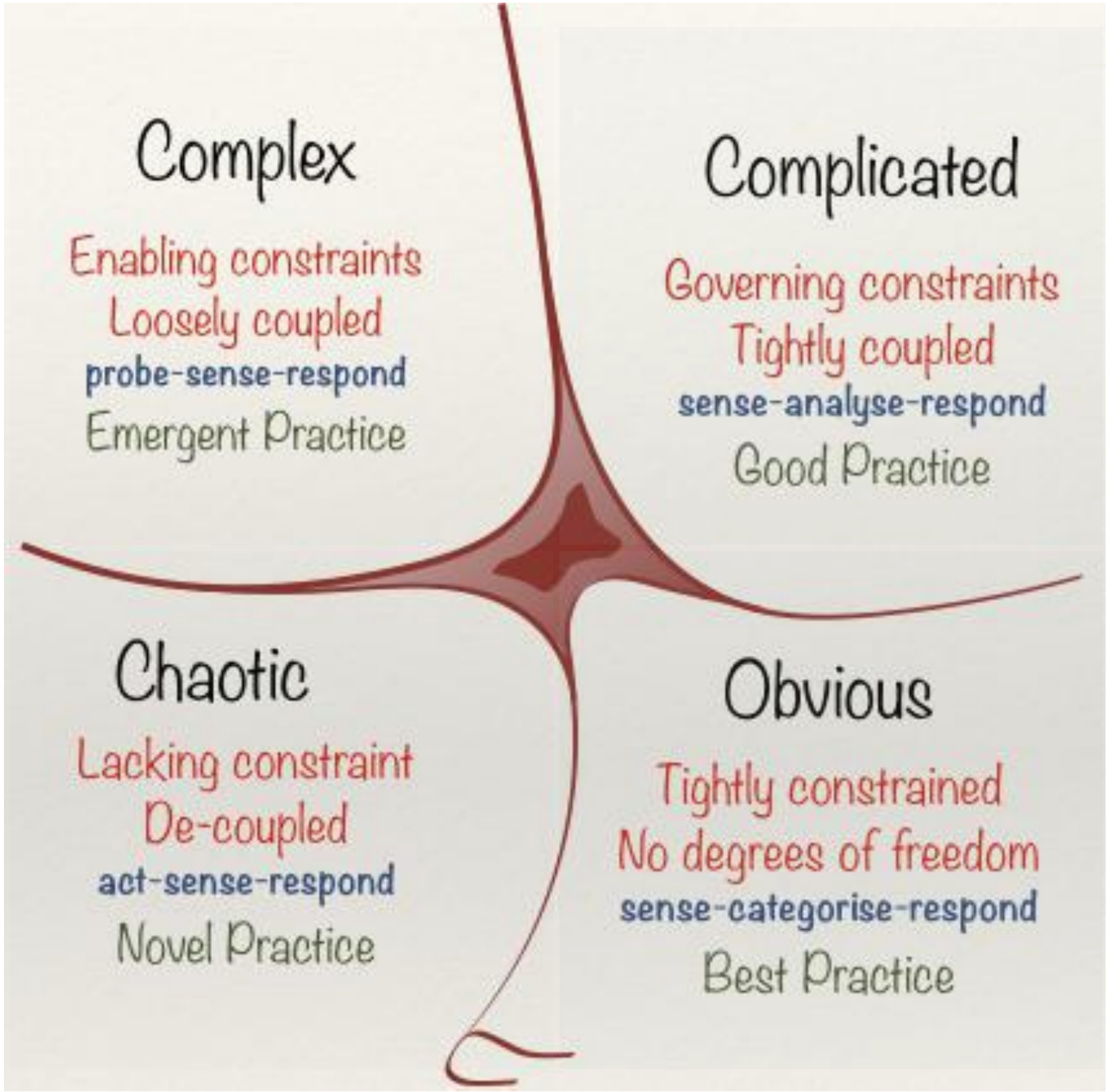
Cynefin framework



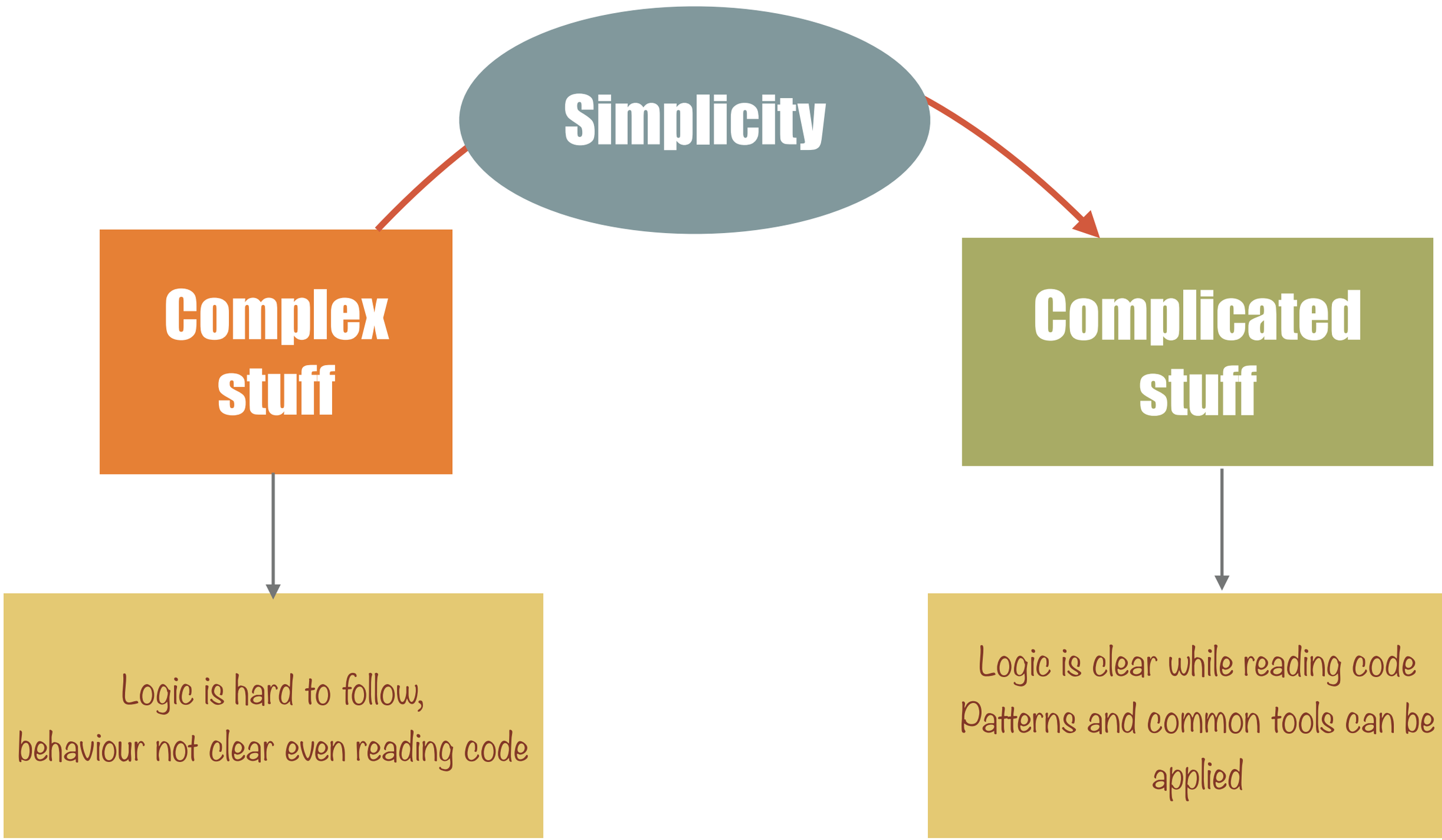
Cynefin framework



Cynefin framework

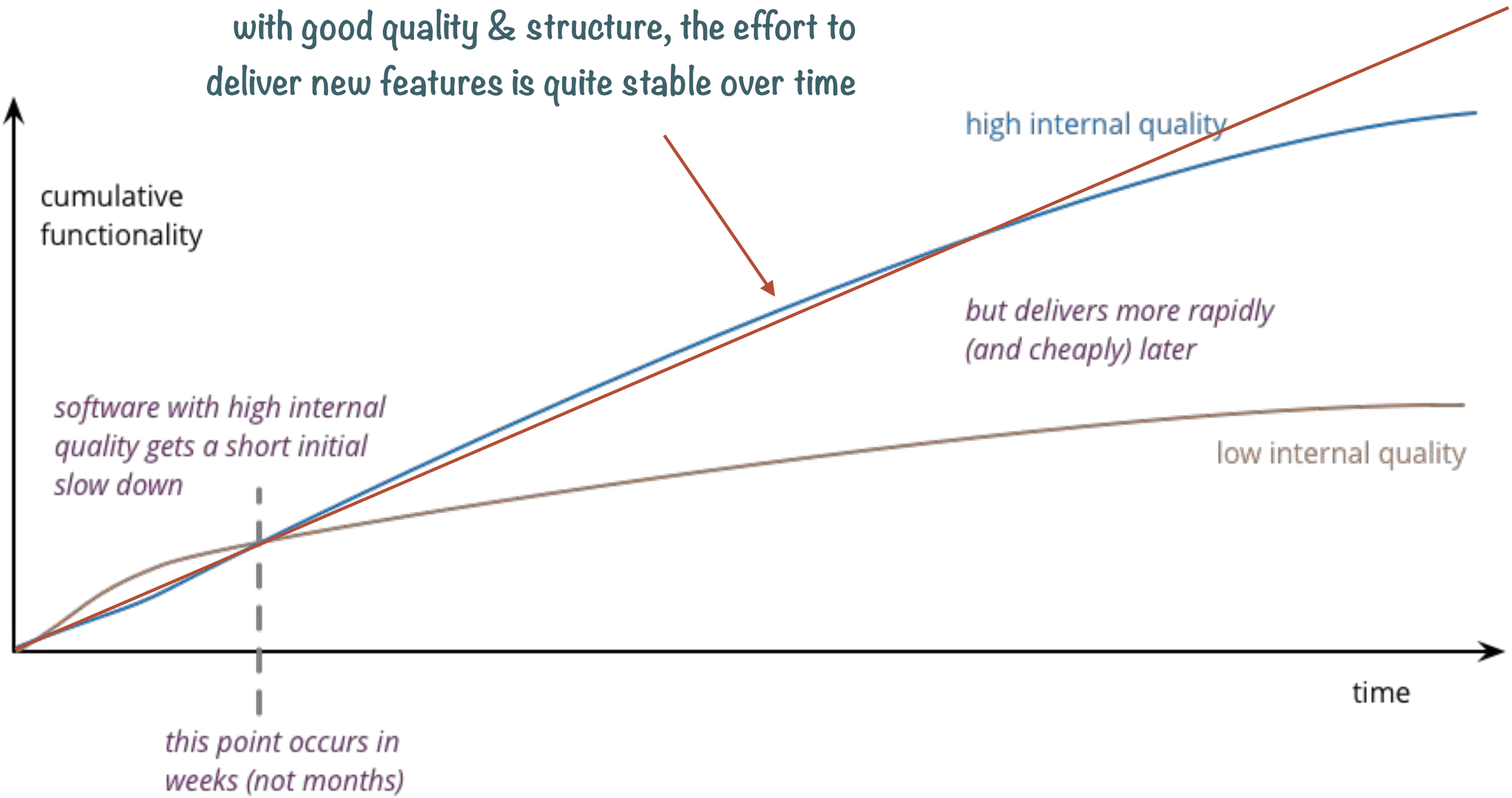


IMPACT ON ARCHITECTURE ?



QUALITY

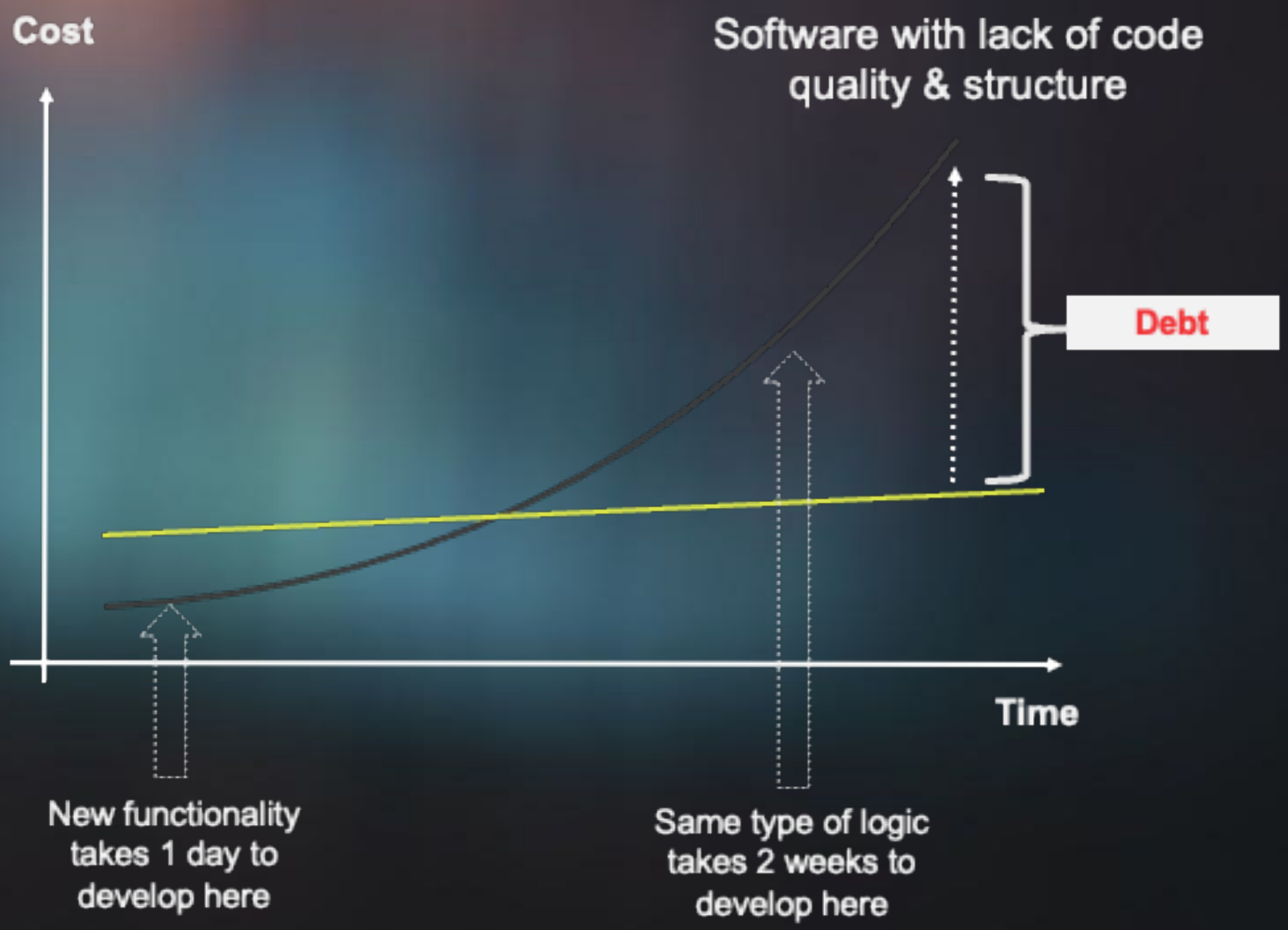
COST OF QUALITY OVER TIME



QUALITY

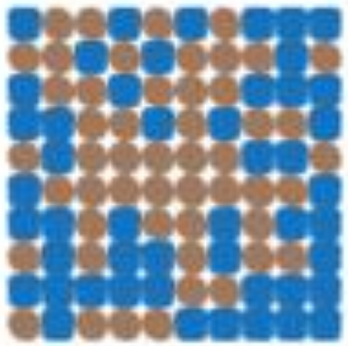
A Definition :

adding equivalent features
should be constant over time



COST OF QUALITY OVER TIME

If we compare one system with a lot of cruft...



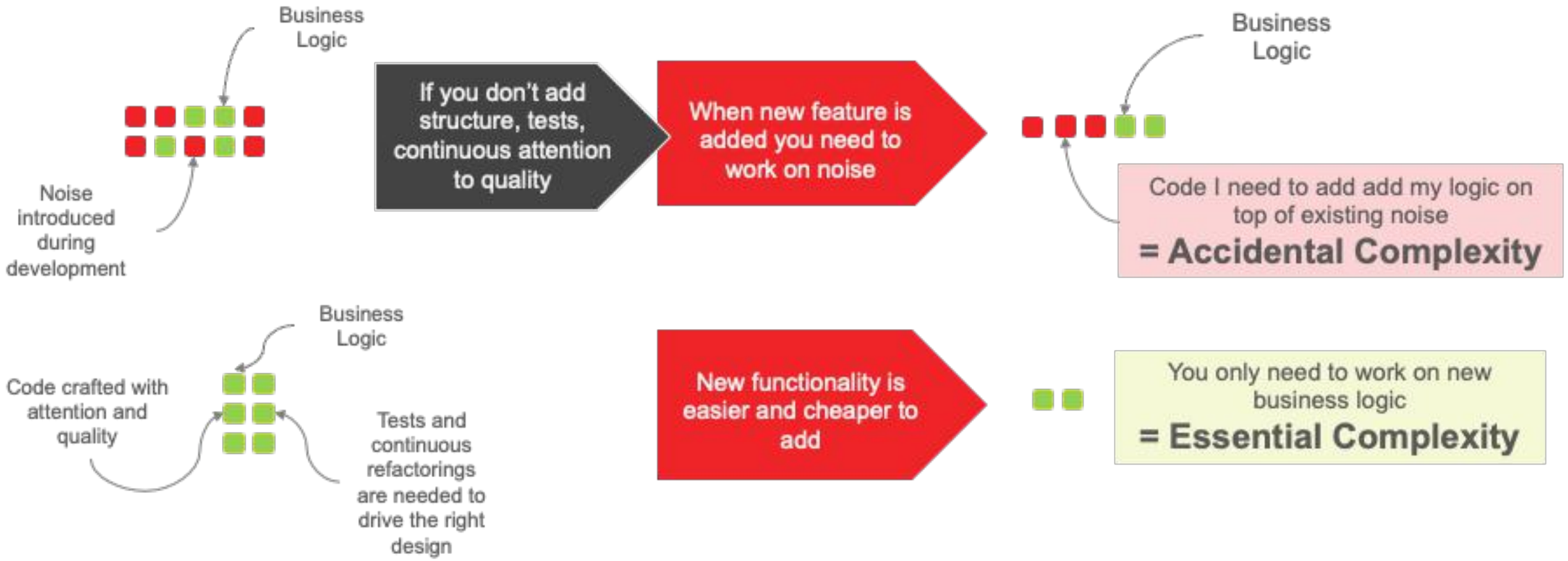
the cruft means new features take longer to build

this extra time and effort is the cost of the cruft, paid with each new feature

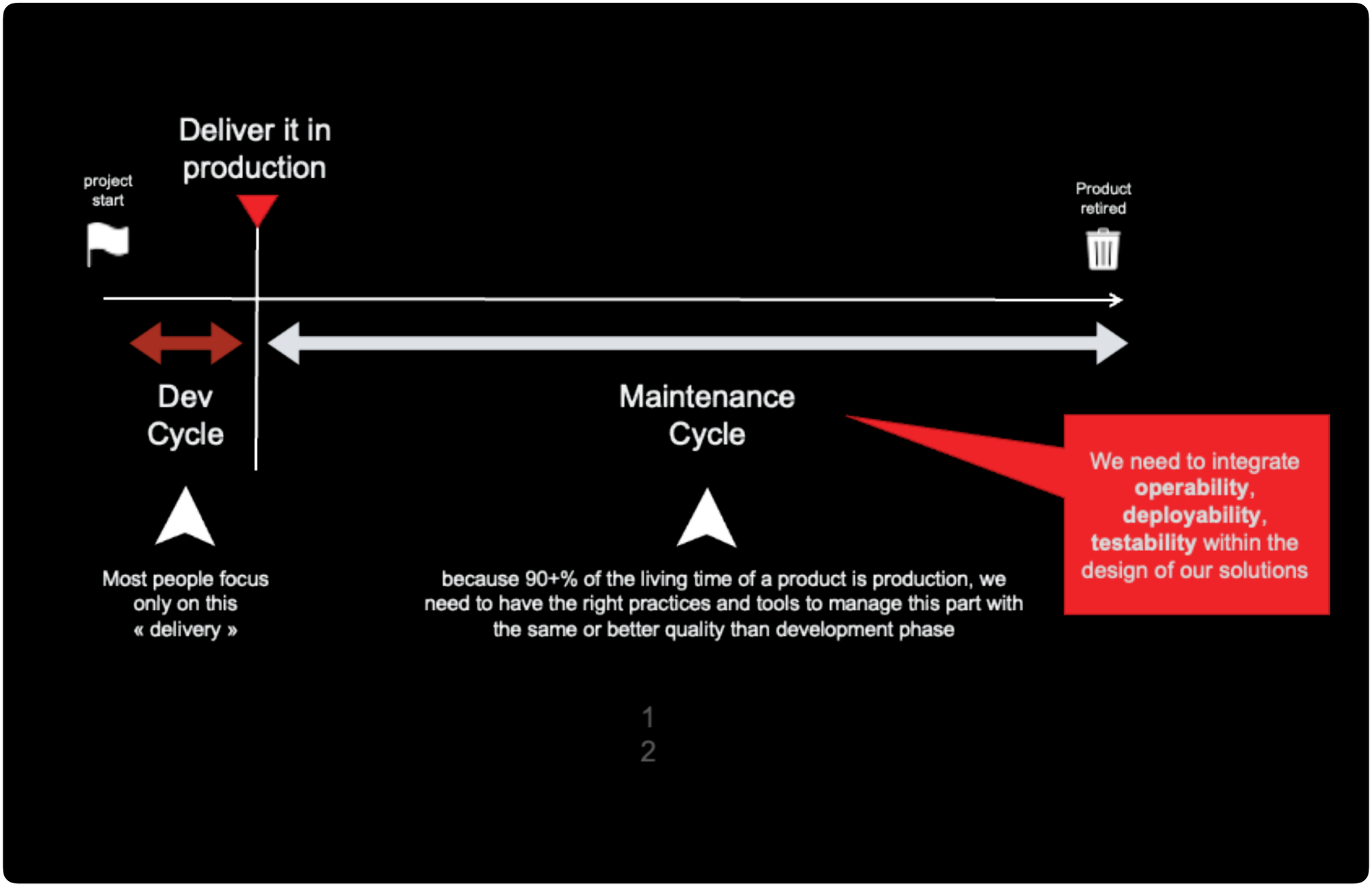
...to an equivalent one without

free of cruft, features can be added more quickly

POOR QUALITY SLOWS YOU DOWN VERY BADLY



OPERABILITY IS ESSENTIAL FOR SUSTAINABILITY



NEED FOR ARCHITECTURE

XP

WE DO ARCHITECTURE:

Kent Beck (1999)

1. To capture stakeholders' perspectives that affect design;
2. To embrace change and to reduce the cost of solving problems;
3. To create a shared vision across the team and the stakeholders;
4. To smooth the decision-making process

PART 2

ARCHITECTURE, HOW ?

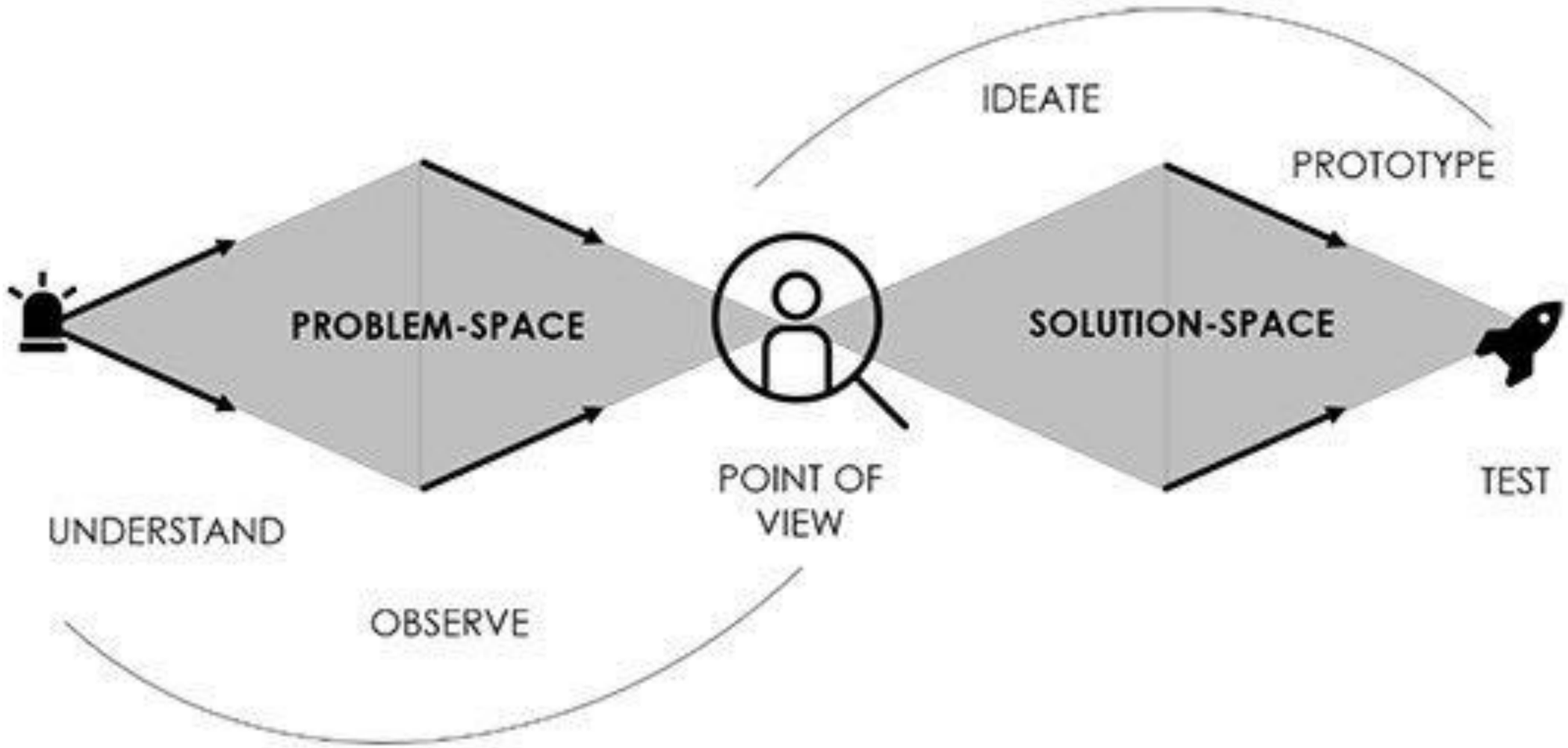
DEFINE THE PROBLEM

Problem :

difference between the current state and the desired state

-Jerry Weinberg

DESIGN THINKING



A GOOD ARCHITECTURE ?

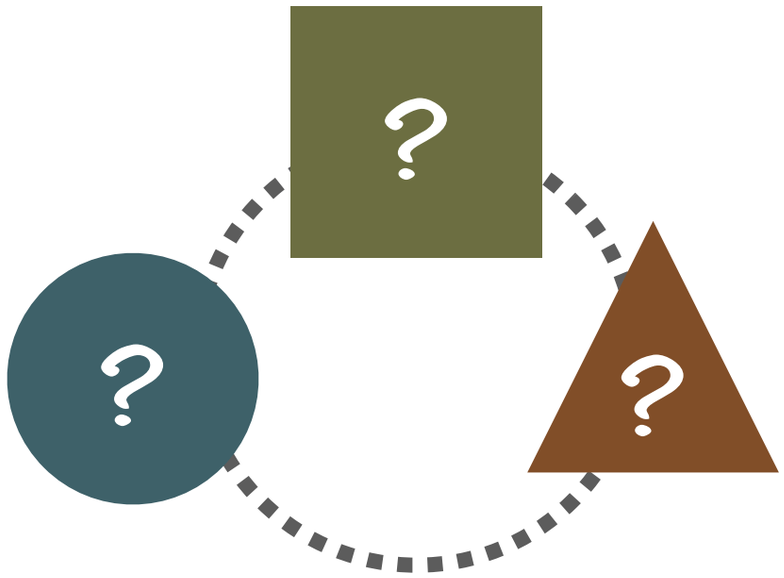
A GOOD ARCHITECTURE

* Leave options open
* Has decoupled Layers
* Has decoupled Use cases
  * visible over technical structure
  * easy to follow
* Permits independent developability
* Permits independent deployability

MORE HEURISTICS

CONSIDER ALTERNATIVES

CONSIDER ALTERNATIVES



Always consider different solutions

Then annotate your decisions through ADRs*

*ADR : Architecture Decision Record

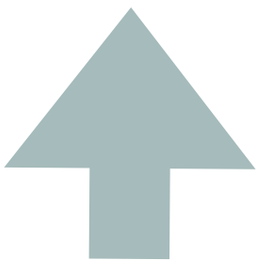
EMERGENT DESIGN

EMERGENT DESIGN

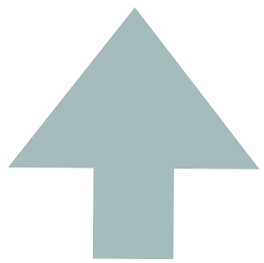
Architectures should be driven by the underlying technical requirements of the system, rather than speculative planning for a future that may change

COUPLING & COHESION

	COHESION	COUPLING
CONCEPT	internal	external
RELATIONSHIP	between modules	within module
REPRESENTS	functional strength	dependence among modules
BEST SOFTWARE	Highly Cohesive	loosely coupled

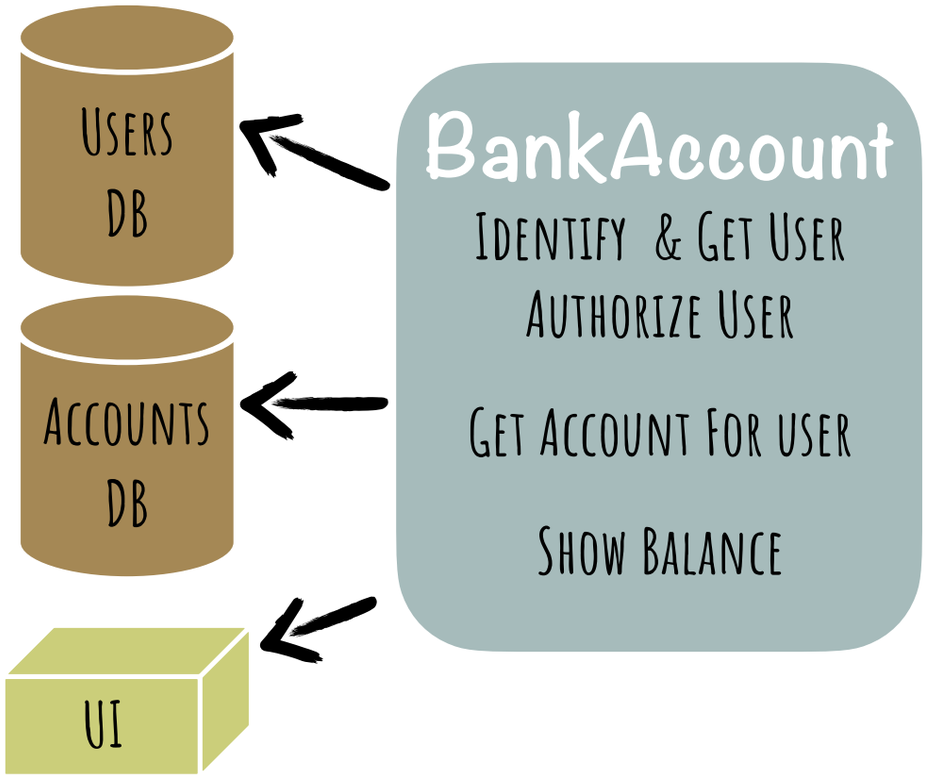


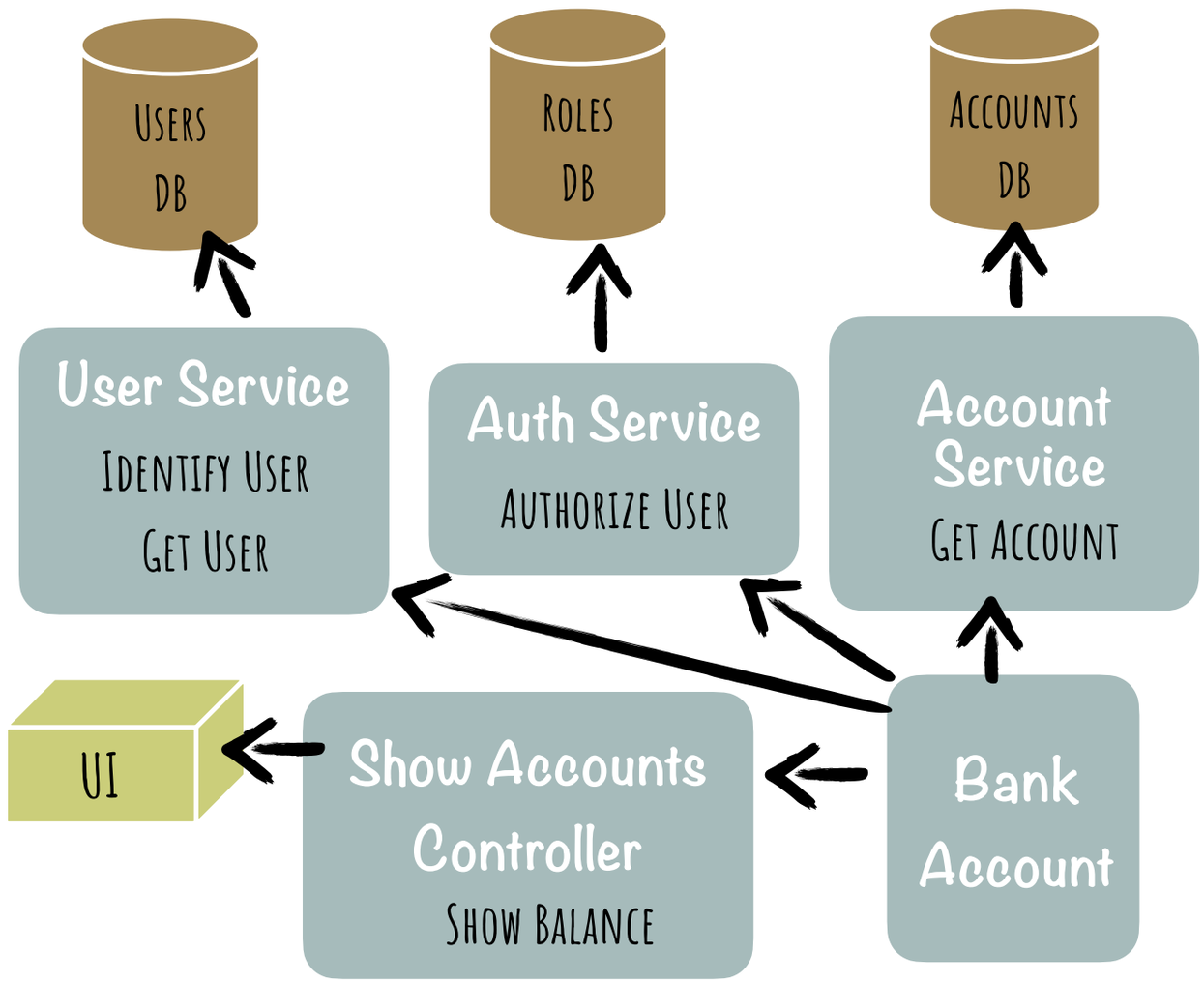
more of that



less of that

COUPLING AND COHESION





Low cohesion:
too many different things in the same component

High coupling:
Direct Dependencies of many components

BAD

GOOD

High cohesion:
Everything inside each component make sense together

Low coupling:
Every service is indenpendent and tied only to its own DB
Composition is made at the end, at the use case level

ARCHITECTURE SMELLS

What should sting you when you see it

MULTI TIERS LAYERS

Coupling behind the scenes

MULTI-TIERS ARCHITECTURES

« As quoted in wikipedia »

« In software engineering, **multitier architecture** (often referred to as **n-tier architecture**) or **multilayered architecture** is a client–server architecture in which presentation, application processing, and data management functions are physically separated. » - *wikipedia*

The most widespread use of multitier architecture is the **three-tier architecture**.

N-tier application architecture provides a model by which developers can **create flexible and reusable applications**. *By segregating an application into tiers, developers acquire the option of modifying or adding a specific layer, instead of reworking the entire application. A three-tier architecture is typically composed of a presentation tier, a domain logic tier, and a data storage tier.*

CLASSICAL TO TIERS

Presentation tier

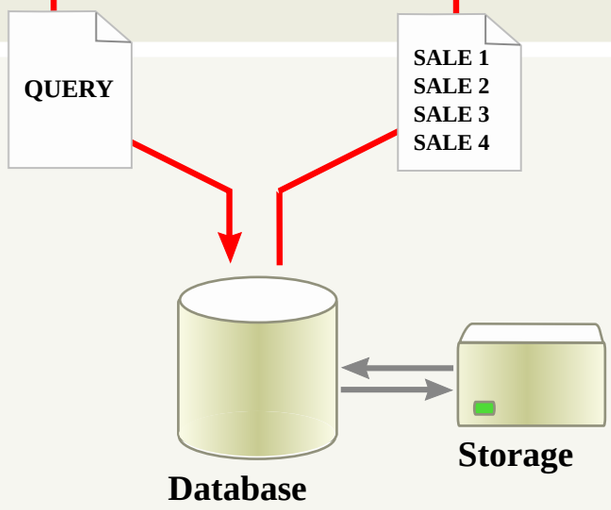
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



N-TIERS ARCHITECTURE FALLACIES : False de-coupling

Presentation tier

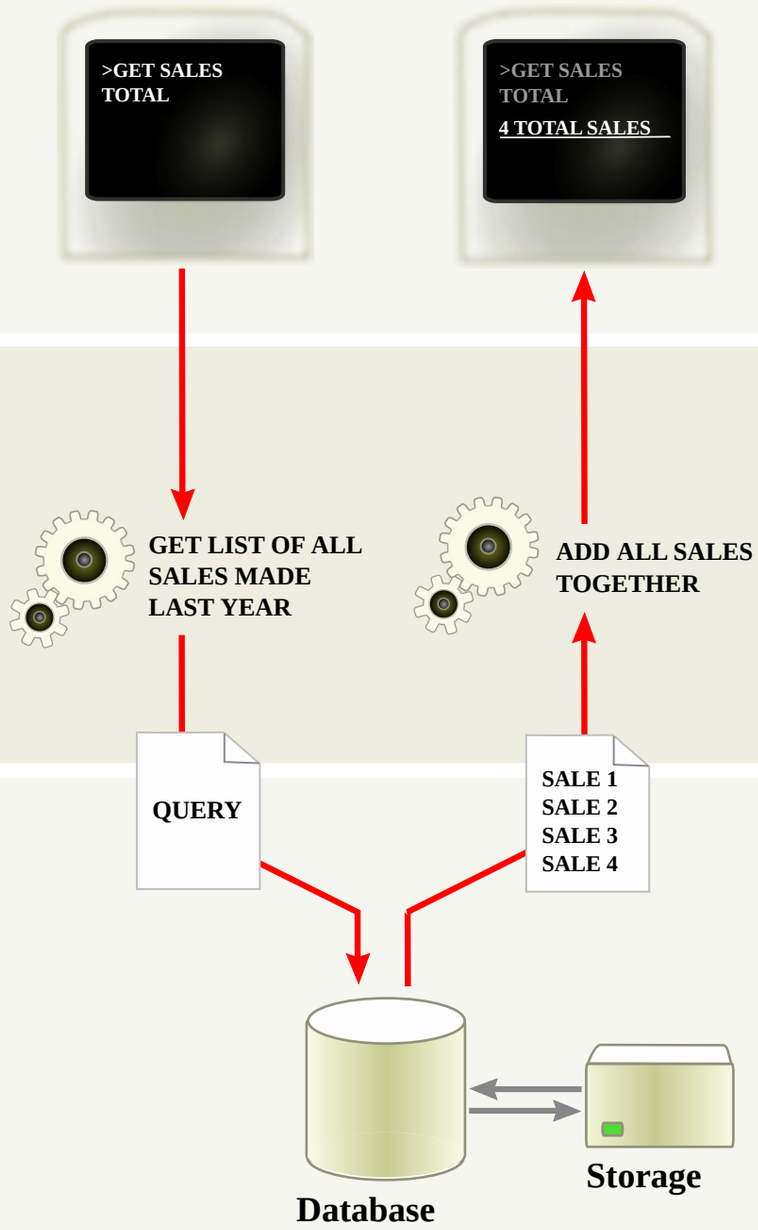
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



Layers based on technical Separation of responsibilities

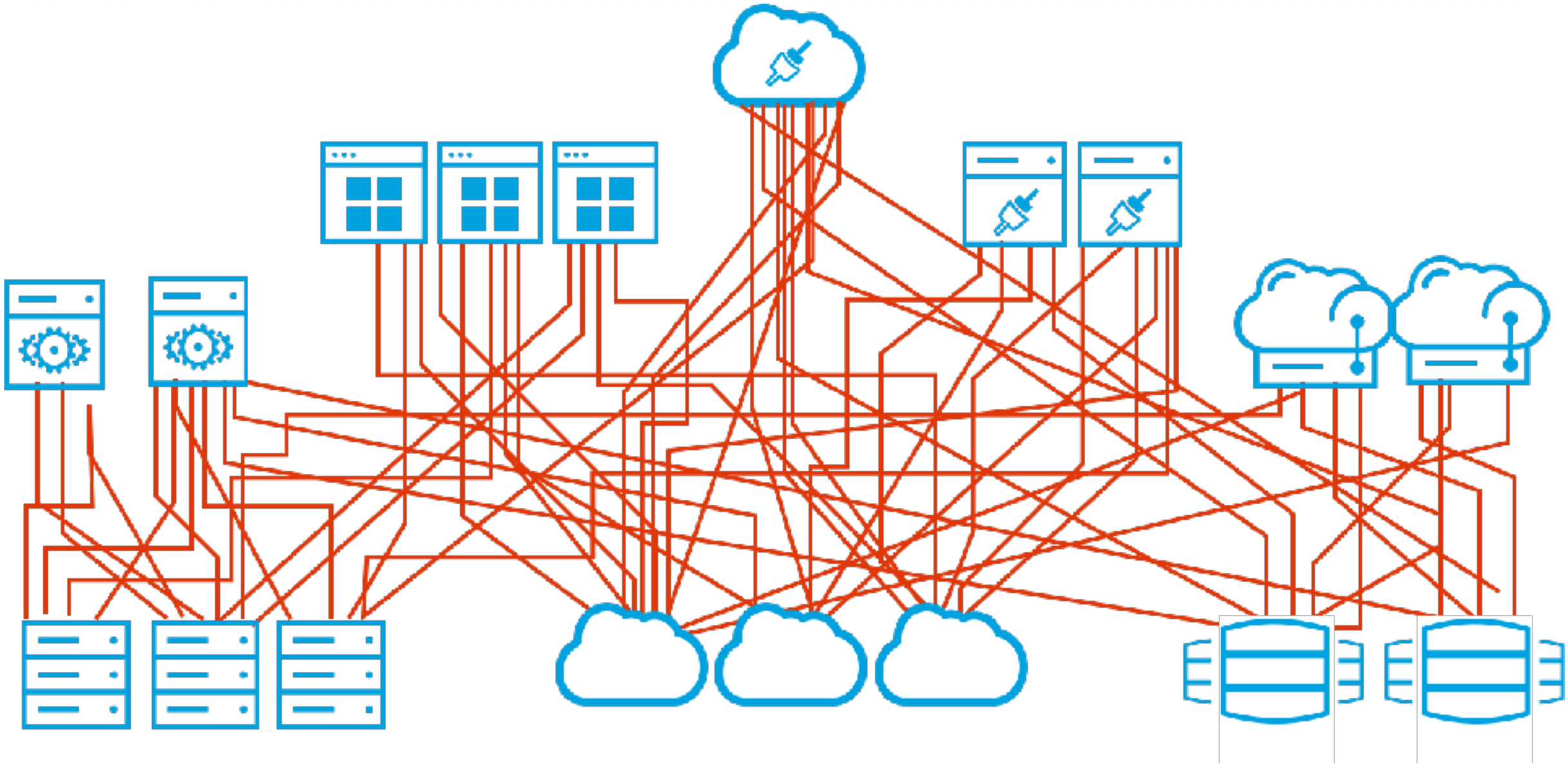
Usually different teams are involved for each layer, creating strong social dependencies

Business features are mixed across all layers making it costly to change or just follow « the flow »

SPAGHETTI INCIDENT

Use it anywhere you need it

USE WHATEVER YOU WANT, EVERYWHERE YOU WANT...



IF YOU WANT TO QUICKLY TURN YOUR CODE INTO AN IMPOSSIBLE MESS

ANEMIC DOMAIN MODEL

No domain

FRAMEWORK SILVER BULLET

Easy to start

TO RESUME

Feel the smell when:

- it's hard to find the entry point
- it's hard to follow the flow
- it's hard to change
- there is many layers without any behavior or value added
- When you see direct calls to other modules in the middle of nowhere
- When you spend more time configuring your framework to fit your needs than writing valuable production code

ARCHITECTURE STYLES

Different styles for different purposes

DESIGN IS ABOUT CHOICES

CODE

Compiled by a machine

+ performance

+ constraints

Optimized for what ?

Code

Read & written by a Human

+ Maintainability

+ Quality

Runtime

Interactions with end users

+ usability

+ value

WE'LL SPLIT THIS IN 3 MAIN CATEGORIES

Layered

Prescriptive

+ Separation of concerns
+ Standardised code organisation
+ Good Predictability
- Cognitive overhead
- Strong constraints
- Dangerous naïve use

COMPLEX

improved simplicity

+ Usually very robust
+ Scale well
+ Best quality over time
- Less obvious
- Hard to start with

Others

Last resort

+ Seem Simple
+ Or Mandatory
+ Work well in POCs or scripts
- Usually not a good idea

OTHERS

NO ARCH

Why should I Care ?

WRITING IN THE FLOW IS A CHOICE THAT YOU MUST BE AWARE OF!

(AND IN RARE MOMENTS IT'S TOTALLY ACCEPTABLE)

AS “PROFESSIONAL DEVELOPERS” AND NOT HOBBYISTS

IT IS YOUR WORK TO BRING STRUCTURE AND COHESION

SUNK COST DRIVEN ARCHI

One to rule them all

A SUNK COST

IS A SUM PAID IN THE PAST THAT IS NO LONGER RELEVANT TO DECISIONS ABOUT THE FUTURE

< PAST DECISIONS SHOULDN'T INFLUENCE OUR BEHAVIORS IN THE FUTURE >

WE PAID FOR IT
YOU'LL USE IT!
(EVEN IF DON'T FIT YOUR NEEDS)

DON'T DO THIS!

ALWAYS QUESTION YOUR CHOICES

USE ADR TO DOCUMENT THAT

MICRO SERVICES

Exlode it all

A MICRO SERVICE IS A UNIT OF AGGREGATION TO ALLOW LOOSE COUPLING AND HIGH COHESION

USE WHATEVER ARCHITECTURE INSIDE THAT WORKS FOR YOU

BUT KEEP IT SIMPLE!

IT MUST BE INDEPENDENTLY DEPLOYABLE (YES INCLUDING DATA)

Think about the difference between your physical architecture and your logical architecture

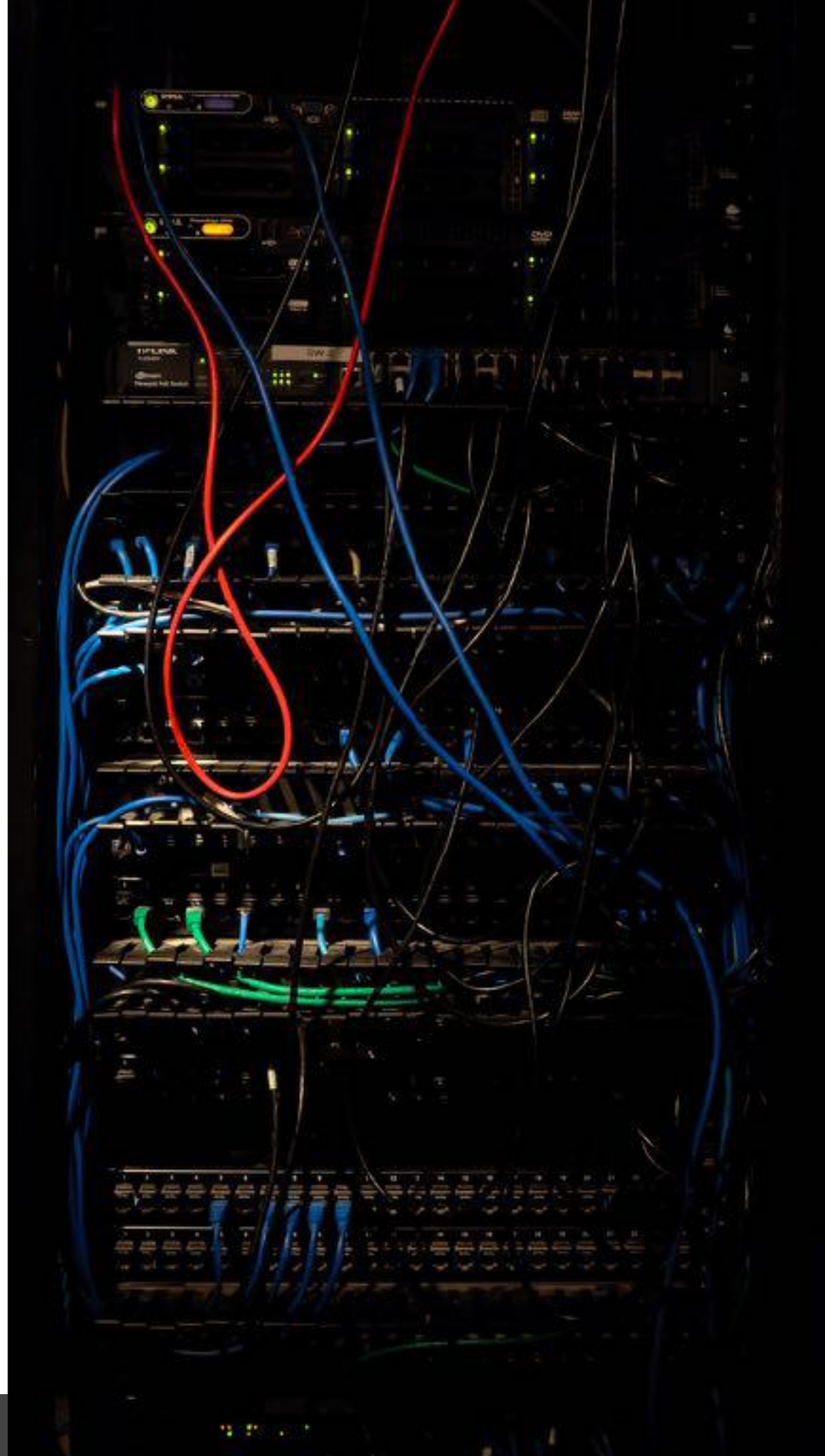
- Me

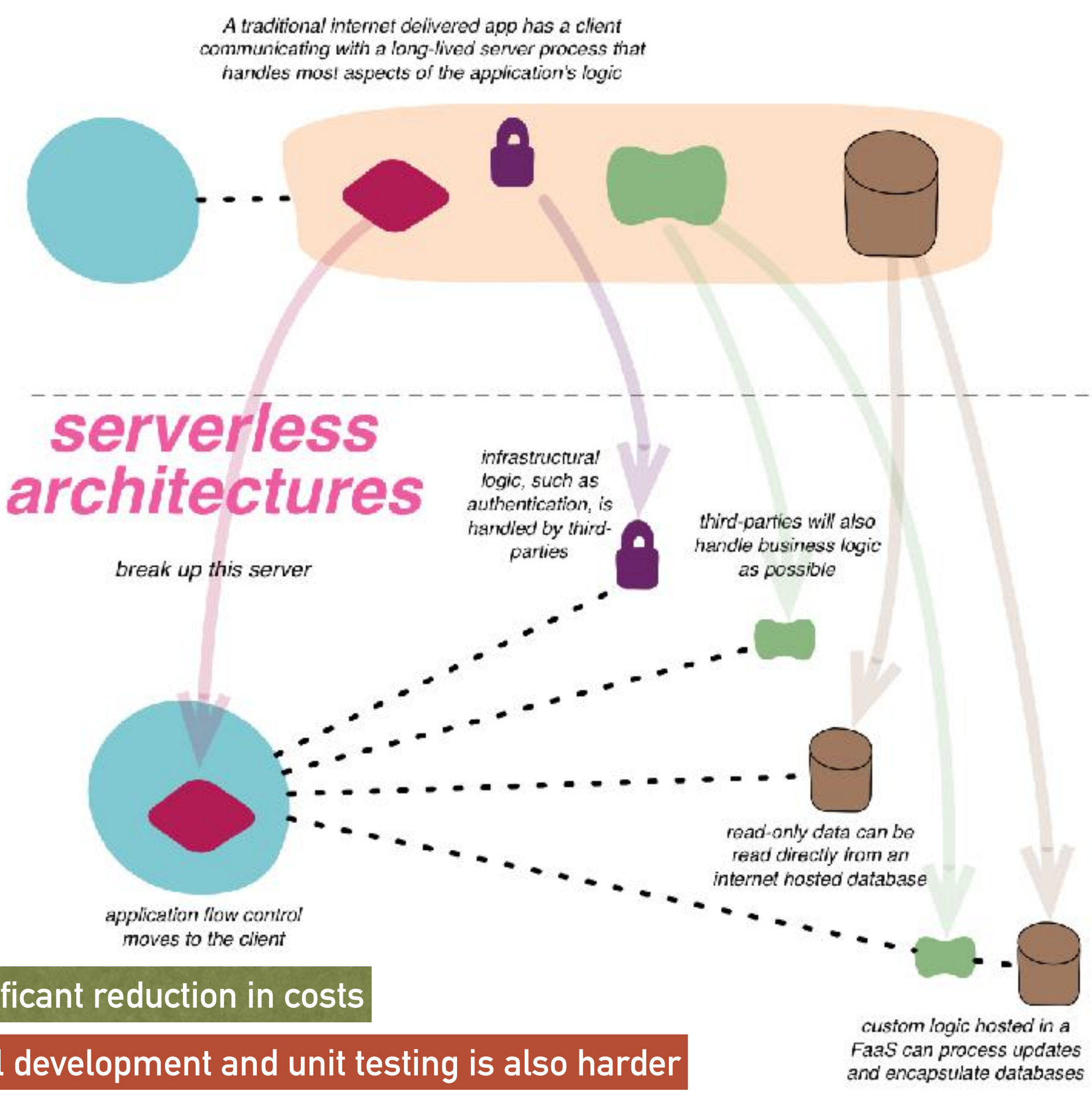
"If you can't build a well-structured monolith, what makes you think you can build a well-structured set of microservices?"

-Simon Brown

SERVERLESS

No architecture ?





significant reduction in costs

Local development and unit testing is also harder

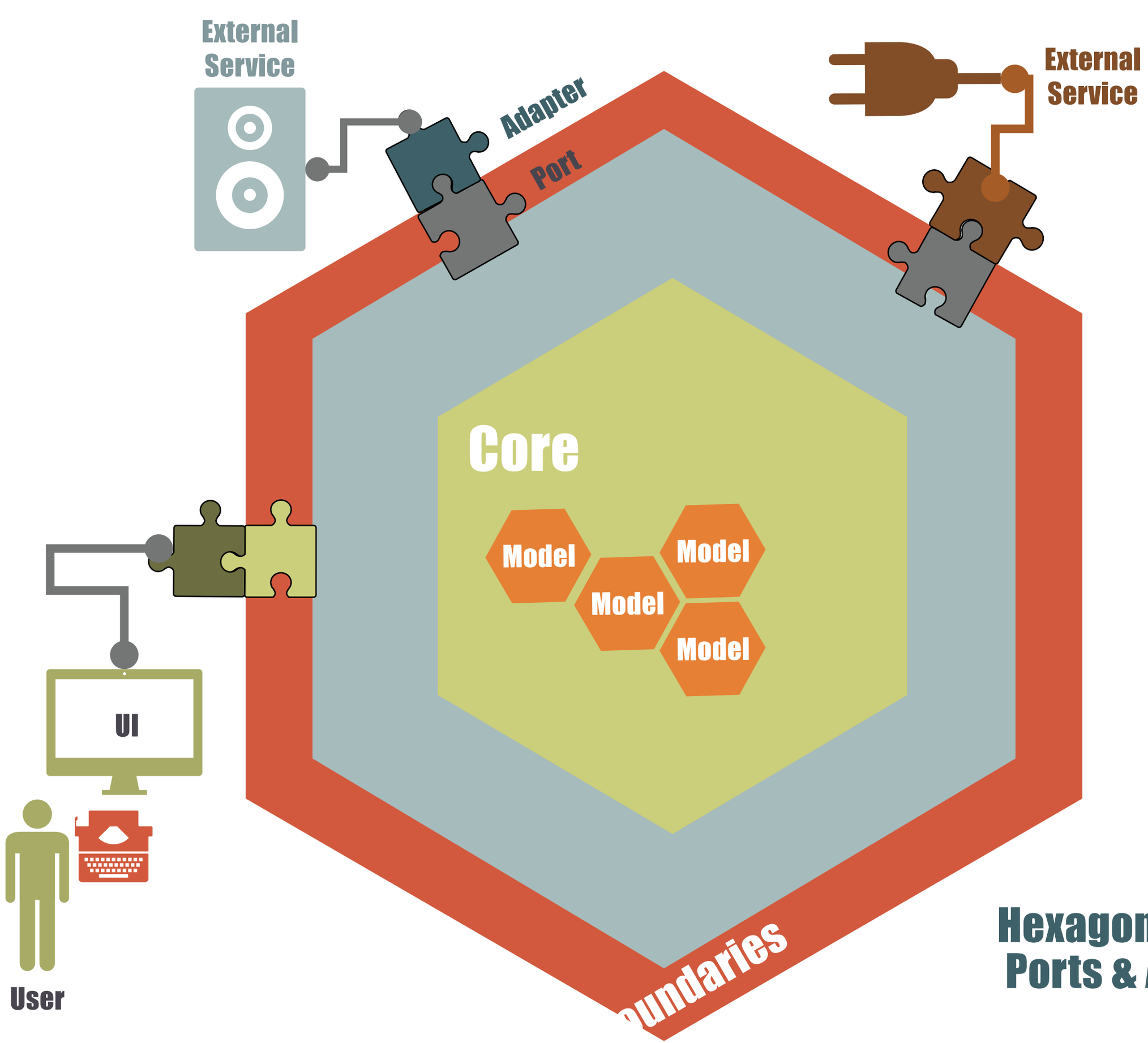
LAYERED ARCHITECTURES

HEXAGONAL - PORTS & ADAPTERS

Protect your domain

Hexagonal architecture is about protecting the domain from external dependencies

Ports & Adapters reflects the gates and protocols to communicate with the outside world



Hexagonal Core + Ports & Adapters

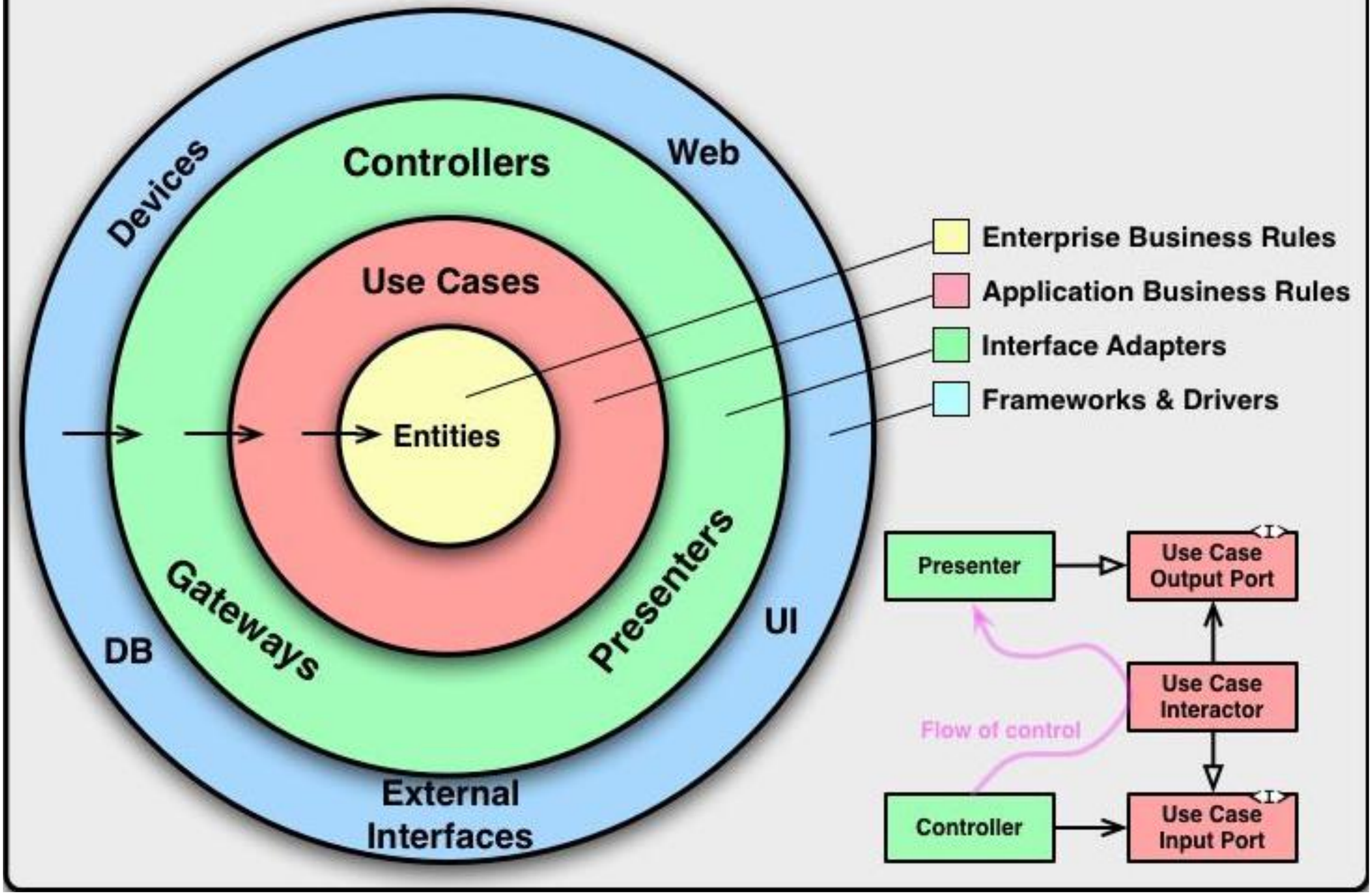
CLEAN ARCHITECTURE

Mix them all

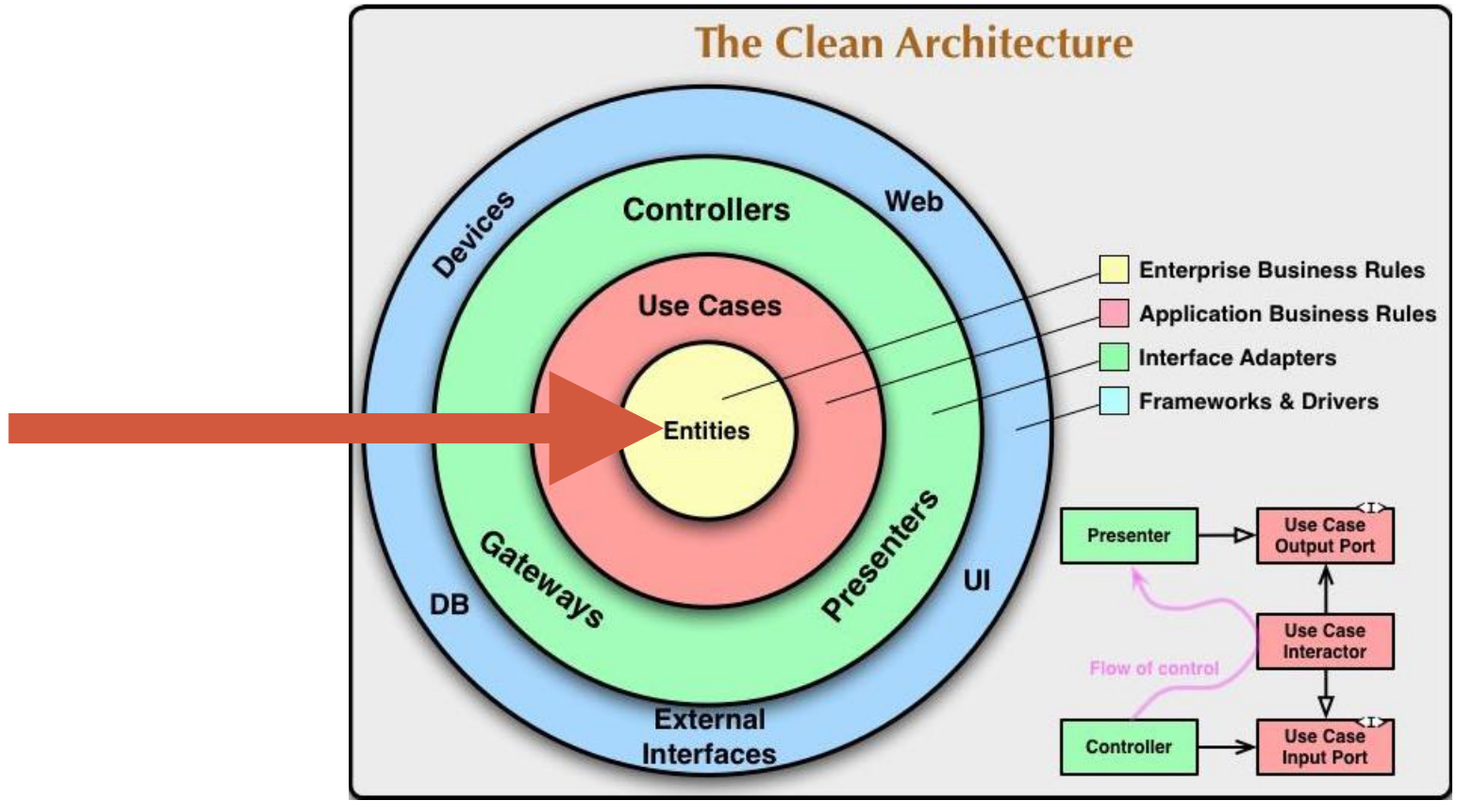
COMMON IN ALL MAJOR ARCHITECTURES

- Independent of Frameworks
- Testable
- Independent of UI
- Independent of Database
- Independent of any external agency

The Clean Architecture



DEPENDENCY RULE FROM EXTERNAL TO INTERNAL



ENTITIES ENCAPSULATE WIDE BUSINESS RULES

It's the most stable part of your app

USE CASES
ORCHESTRATE THE FLOW OF DATA TO AND FROM THE ENTITIES

This defines interactions, behavior of entities in a context

INTERFACE ADAPTERS

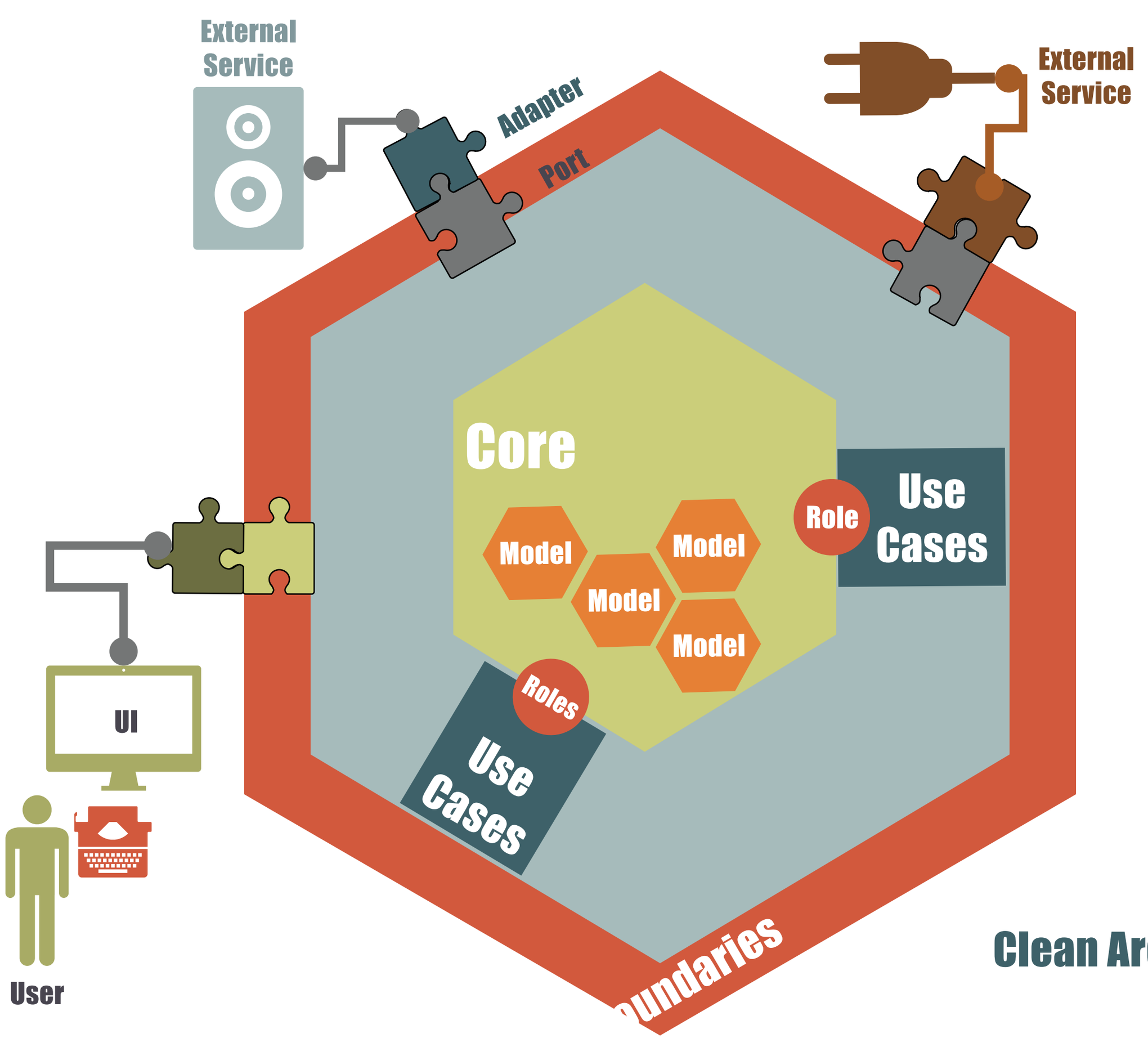
CONVERT DATA FROM THE FORMAT MOST CONVENIENT FOR THE USE CASES AND ENTITIES, TO THE FORMAT MOST CONVENIENT FOR SOME EXTERNAL AGENCY

They are the ones allowing the communication with the outside world

FRAMEWORKS & DRIVERS

THIS LAYER IS A GLUE TO CONCRETE IMPLEMENTATION OF TOOLS WE WANT TO USE

This should be a thin layer that configures and links these external dependencies

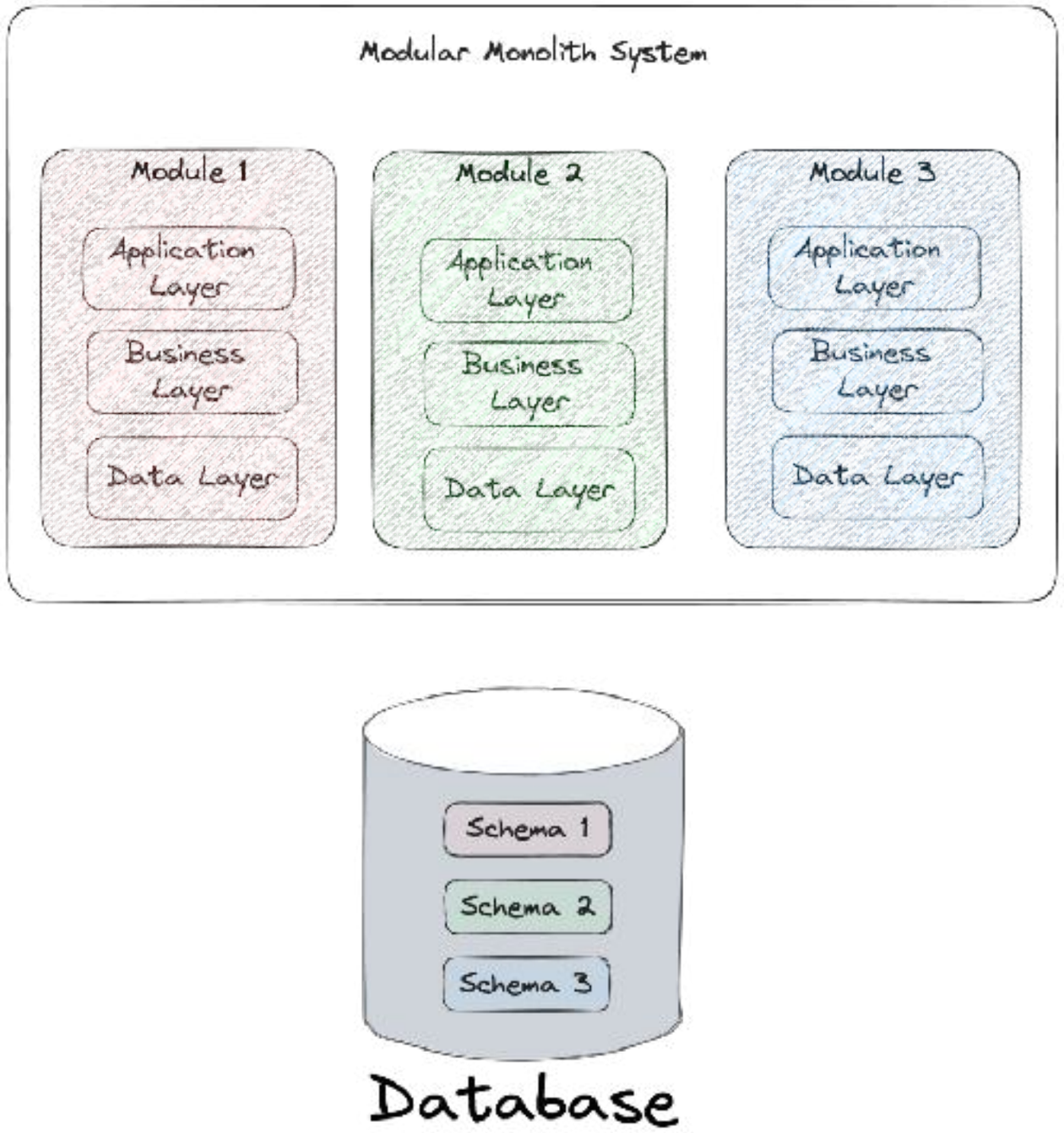


Clean Architecture

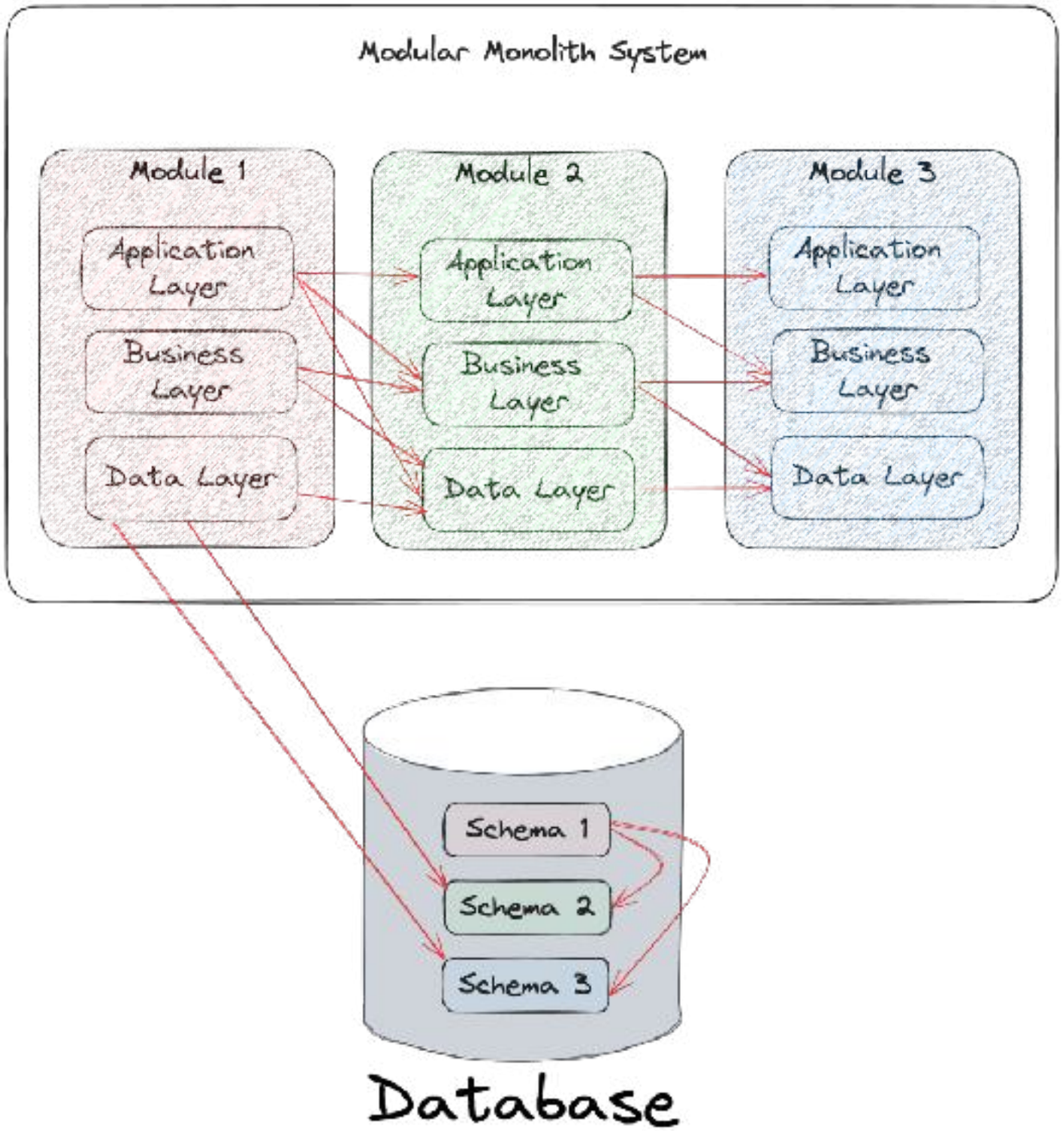
MODULAR MONOLITH

Privilege logical architecture over physical

A SIMPLE OVERVIEW



THE RISK IS ABOUT USING EVERYTHING THAT IS EASY AVAILABLE ...



THINK ABOUT HOW YOUR MODULES WORK AND INTERACT LOGICALLY OVER HOW THEY ARE PHYSICALLY ORGANISED

BUILD UPON SMALL LOGICAL AND INDEPENDENT MODULES

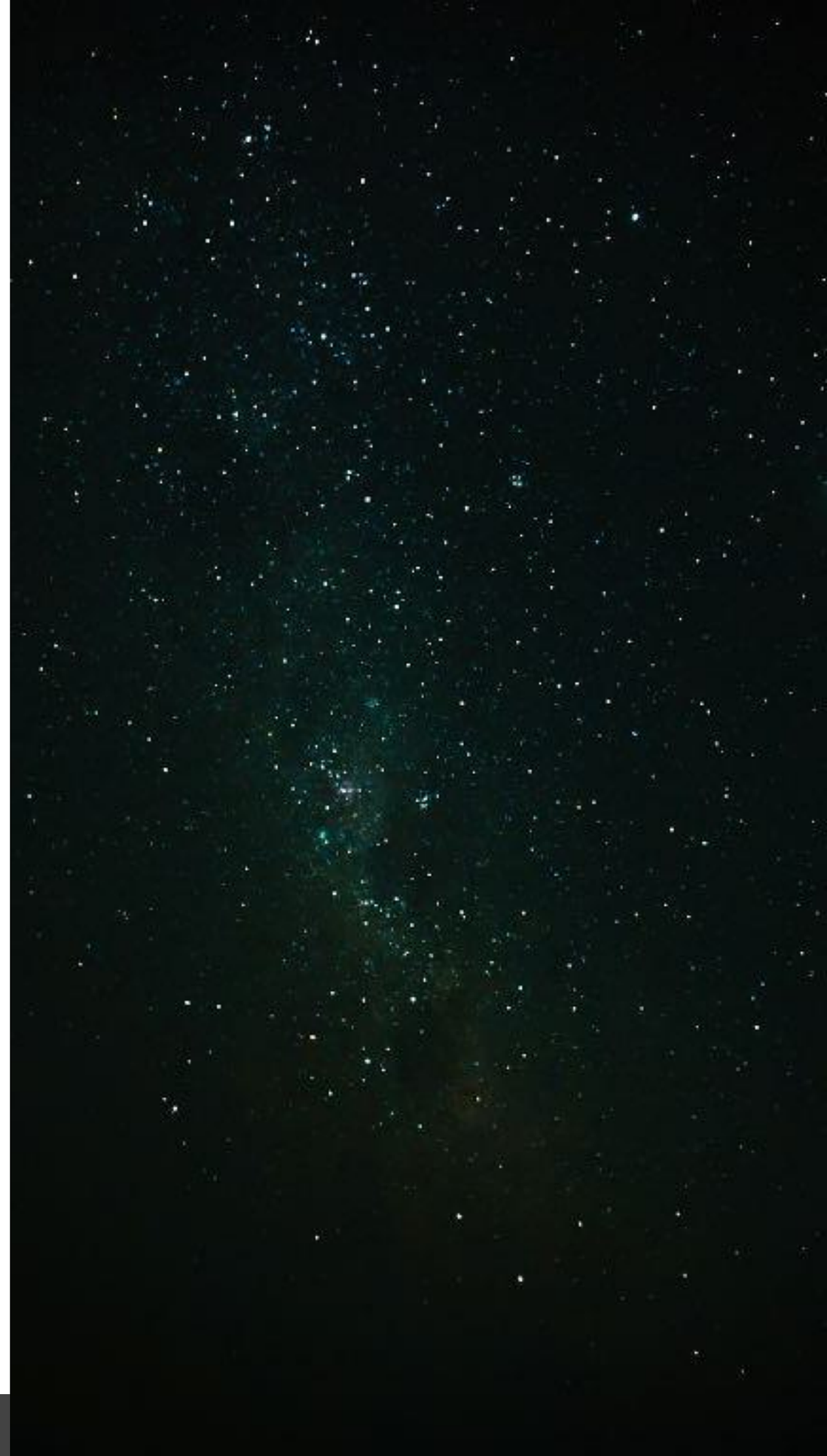
THINK ABOUT HOW THEY MUST COMMUNICATE ON THEIR HIGHER LEVEL OF ABSTRACTION

NEVER CROSS BOUNDARIES AND USE AN INTERNAL KNOWLEDGE OF OTHER MODULE

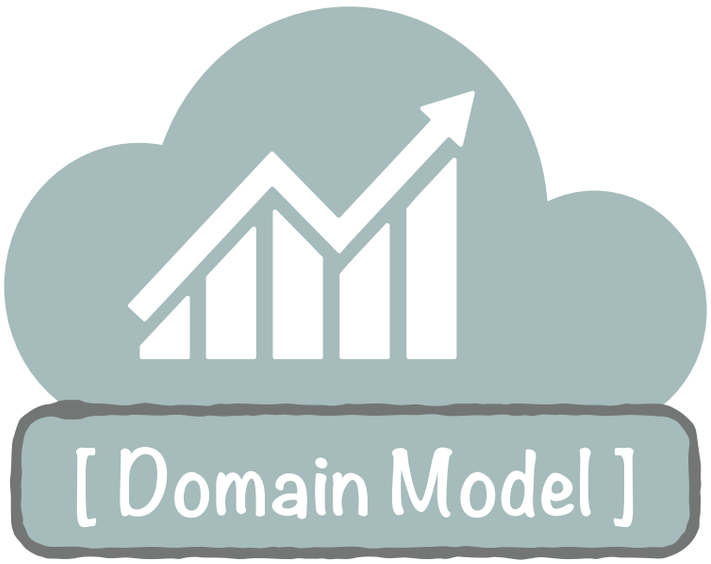
COMPLEX ARCHITECTURES

EVENT SOURCING

Your log is your app



EVENT SOURCING

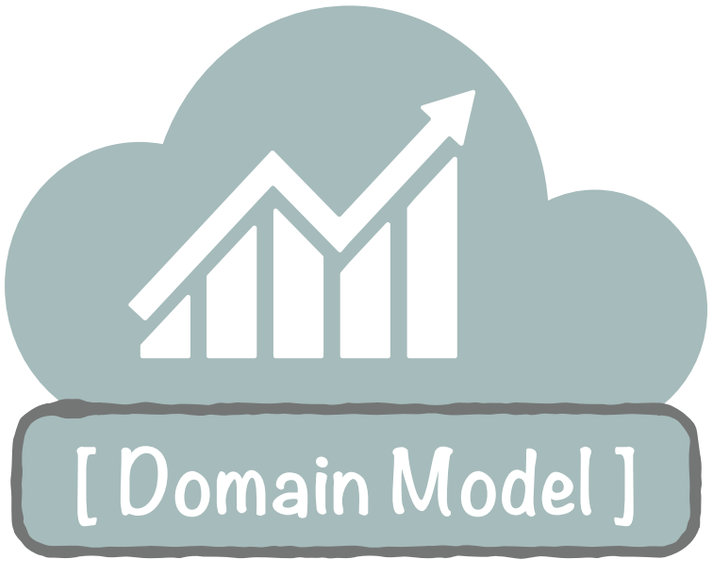


You can persist your domain model into a database

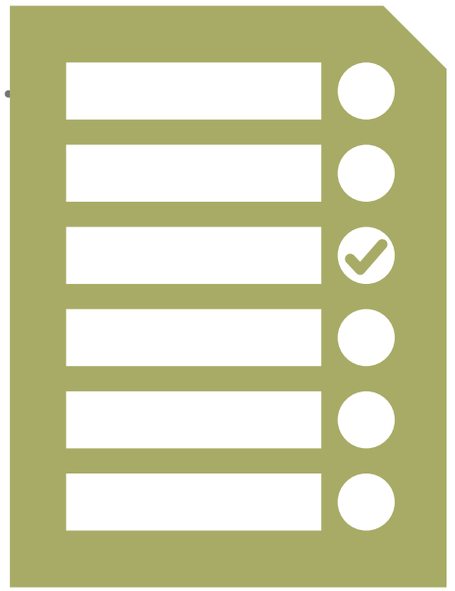
This is a limited snapshot of the truth

EVENT SOURCING

Instead of persisting a model



Store the transactions



This is a limited snapshot of the truth

And build the projections of models you need when necessary

HOW DOES IT WORKS ?

Structure

Classic database models

BankAccount
Id
Owner
TotalBalance

Event sourced models

Command Store
Command
Date
Context

Records

BankAccount
1
Paul
10$

Updated Account, removes previous record

BankAccount
1
Paul
22$

In your app, you read the state, do account movements, and store the actual new state

Command Store
CreateBankAccountForOwner(1, "Paul")
AddMoneyToAccount(10, 1)
AddMoneyToAccount(20, 1)
RemoveMoneyFromAccount(8, 1)

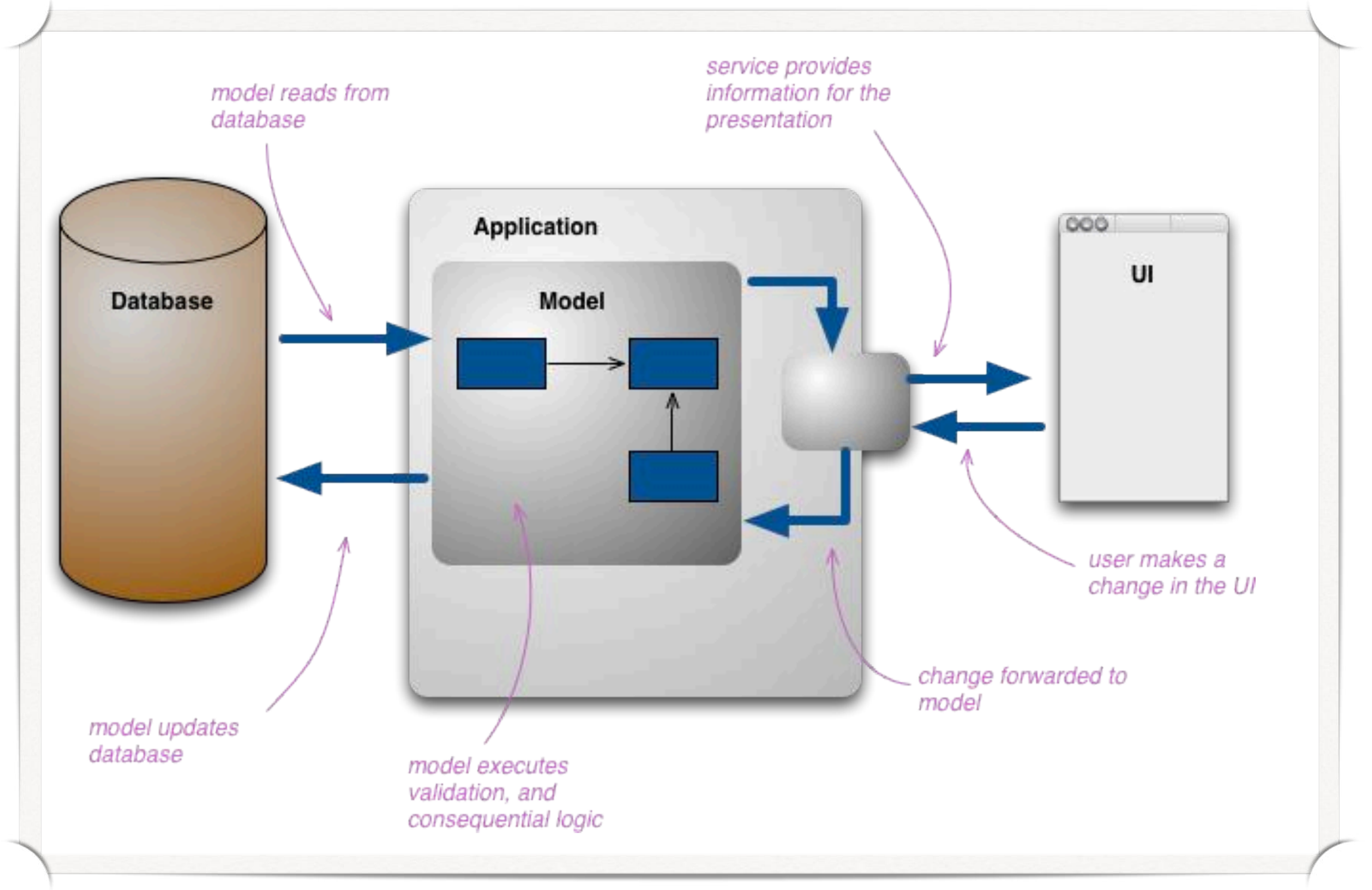
In your app, store the commands when happen

VIEW : In your app, create new object and apply the commands, then you'll have the new state

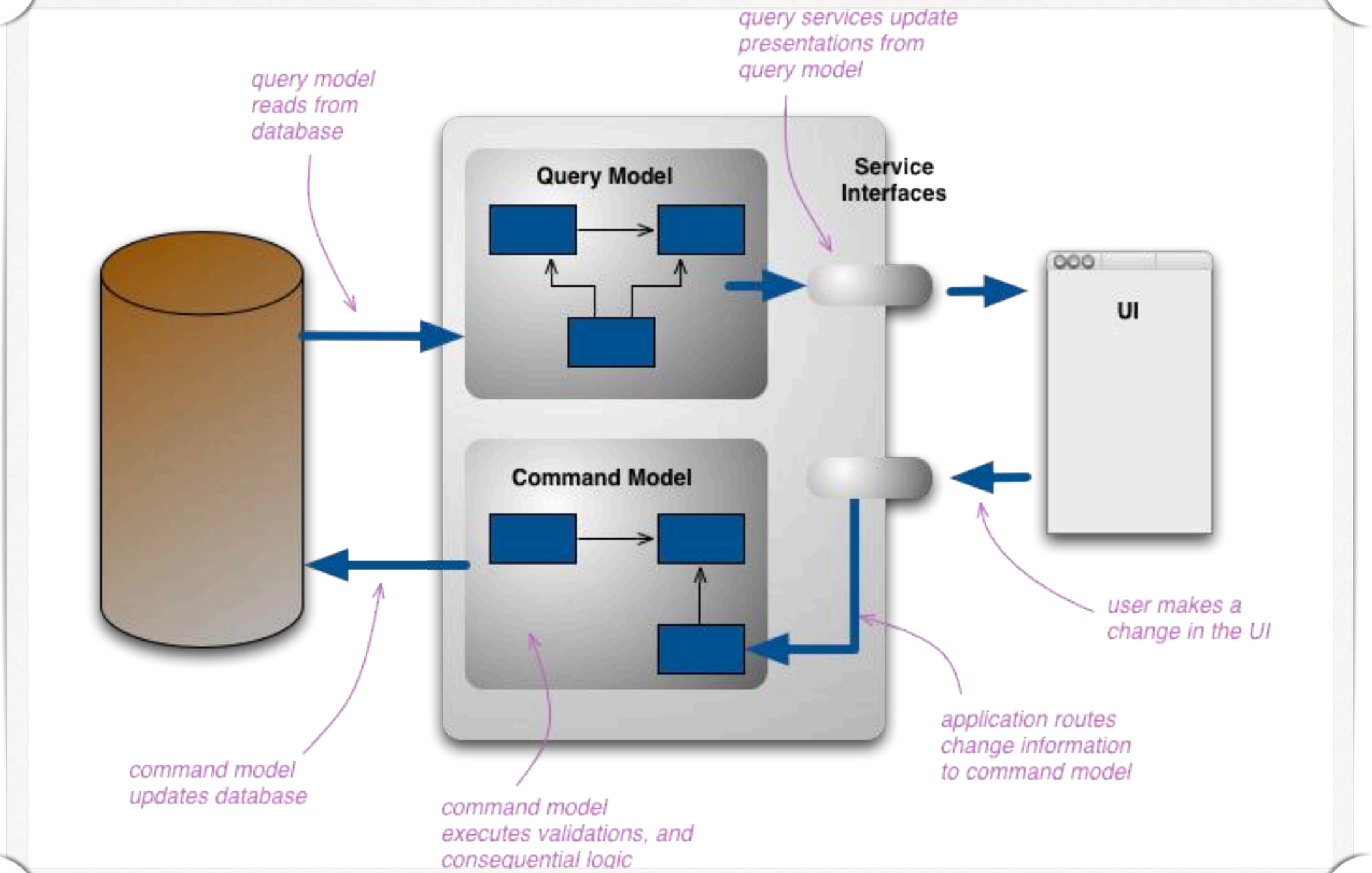
CQRS

Segregate all the things

(CLASSIC) SINGLE MODEL (contains all data interactions)

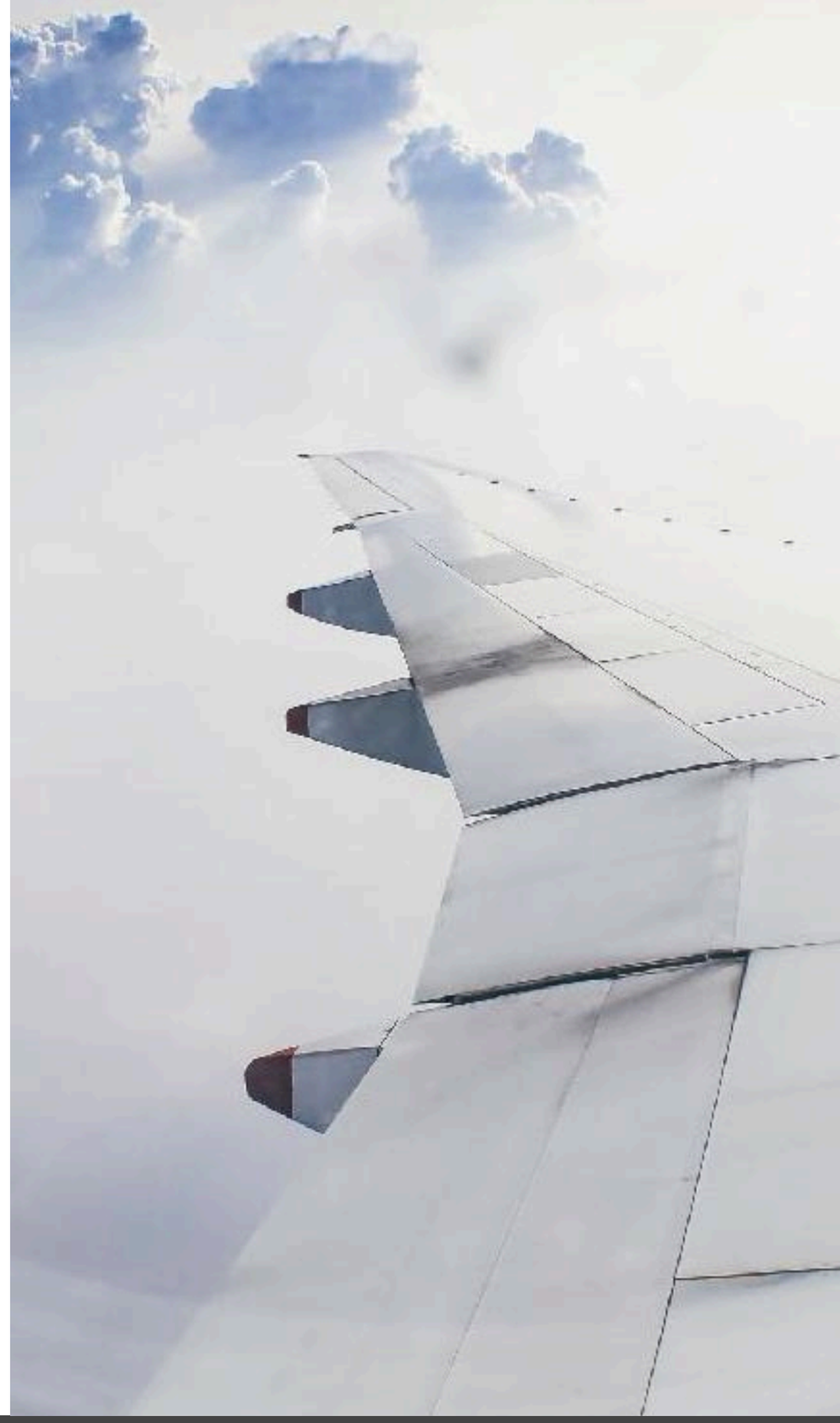


SEGREGATED MODEL



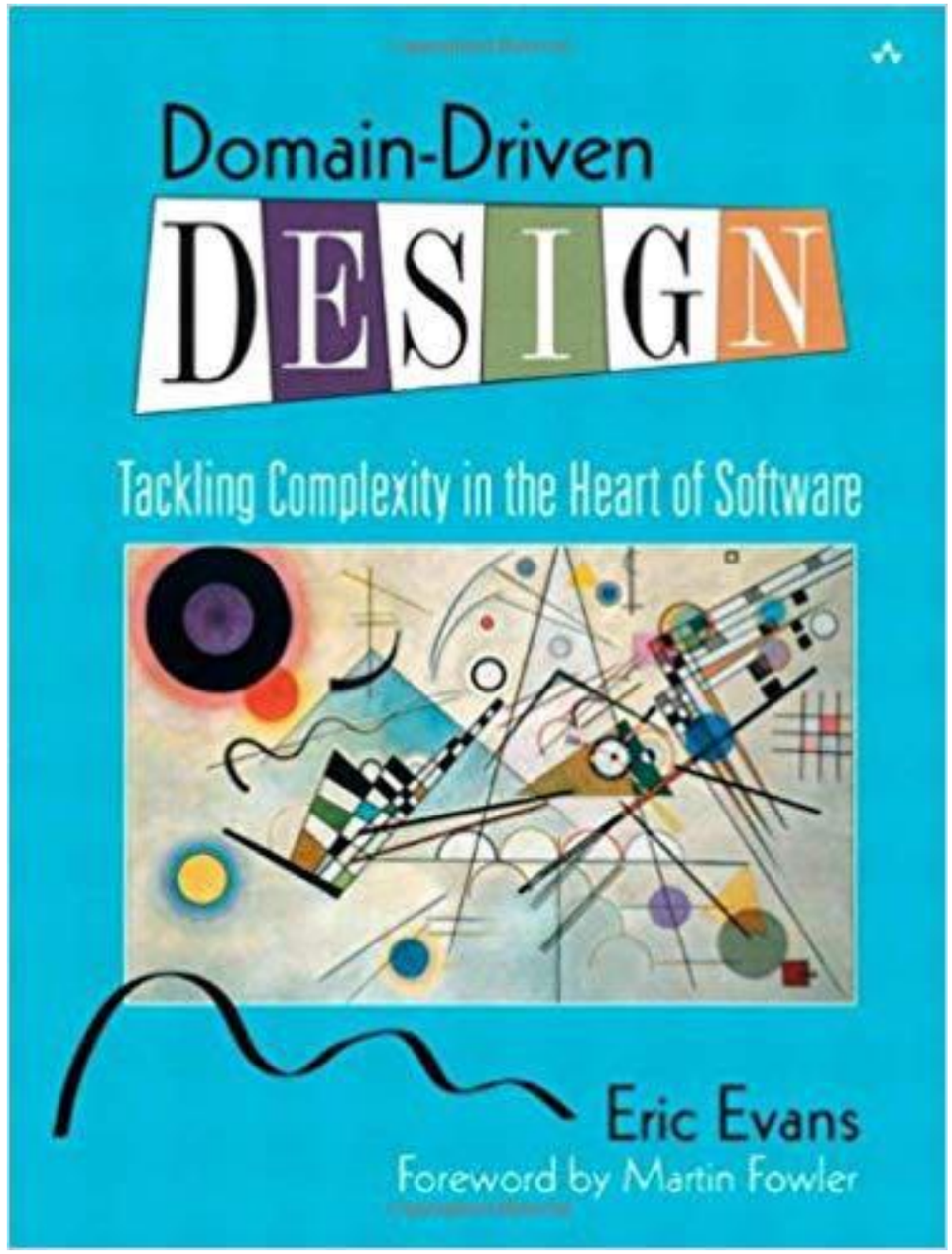
DDD

More a behavior than a style of architecture

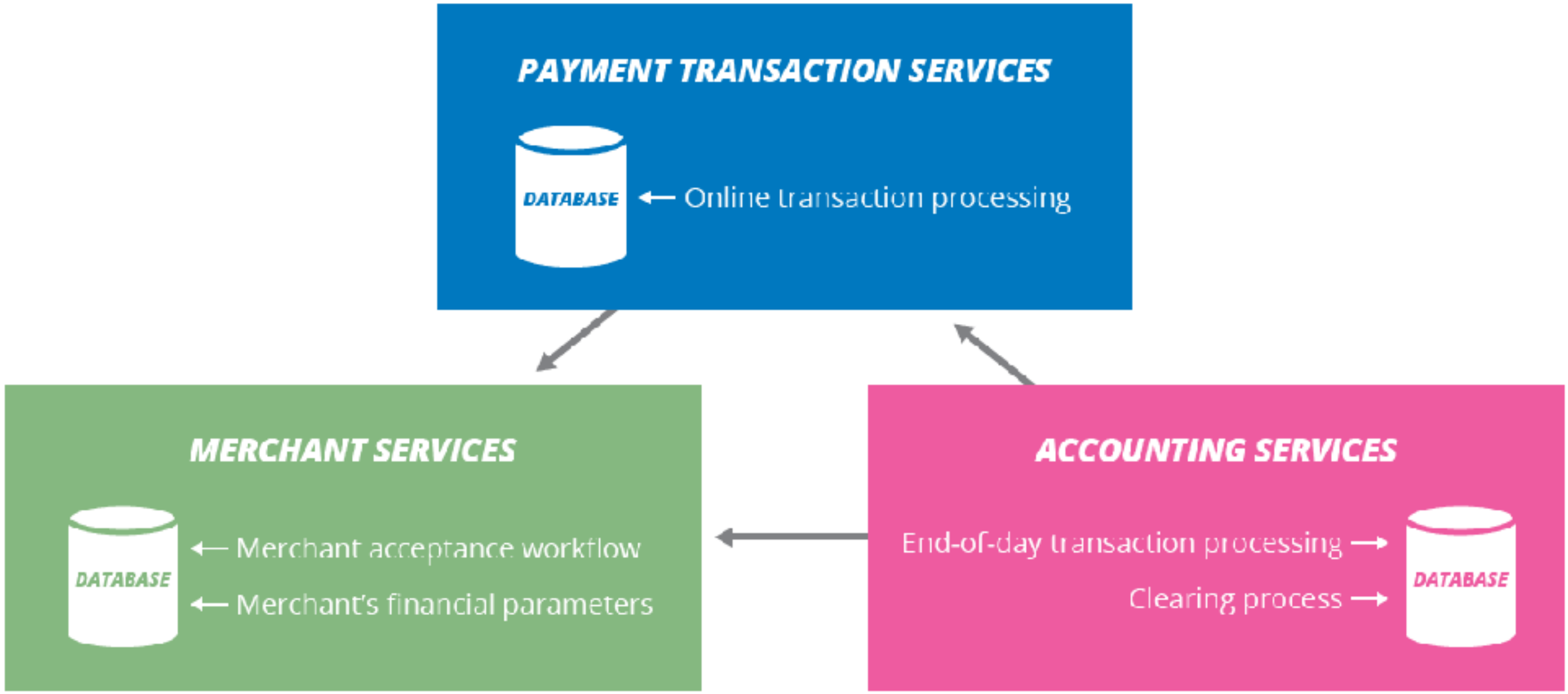


DOMAIN DRIVEN DESIGN Eric Evans (2004)

is an approach to developing software for complex needs by deeply connecting the implementation to an evolving model of the core business concepts.



DDD APPROACH



DOMAIN DRIVEN SERVICES INTEGRATED VIA SERVICES

DEFINITIONS IN DDD WORLD

domain

A sphere of knowledge, influence, or activity. The subject area to which the user applies a program is the domain of the software.

model

A system of abstractions that describes selected aspects of a domain and can be used to solve problems related to that domain.

ubiquitous language

A language structured around the domain model and used by all team members within a bounded context to connect all the activities of the team with the software.

context

The setting in which a word or statement appears that determines its meaning. *Statements about a model can only be understood in a context.*

bounded context

A description of a boundary (typically a subsystem, or the work of a particular team) within which a particular model is defined and applicable.

DDD PATTERN LANGUAGE OVERVIEW

DOMAIN DRIVEN DESIGN

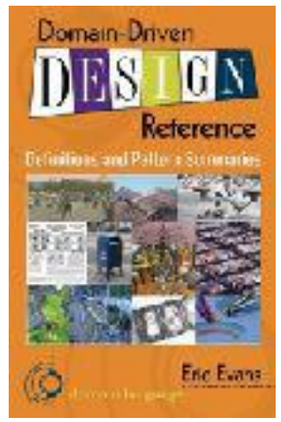
As a whole approach, it's not technically specific, it's not an "architecture"

Implementing it's concepts could lead us to Hexagonal architecture, event sourcing and CQRS

The most important contribution for architecture is the notion of

Bounded Contexts

https://domainlanguage.com/ddd/reference/



FUNCTIONAL CORE, IMPERATIVE SHELL

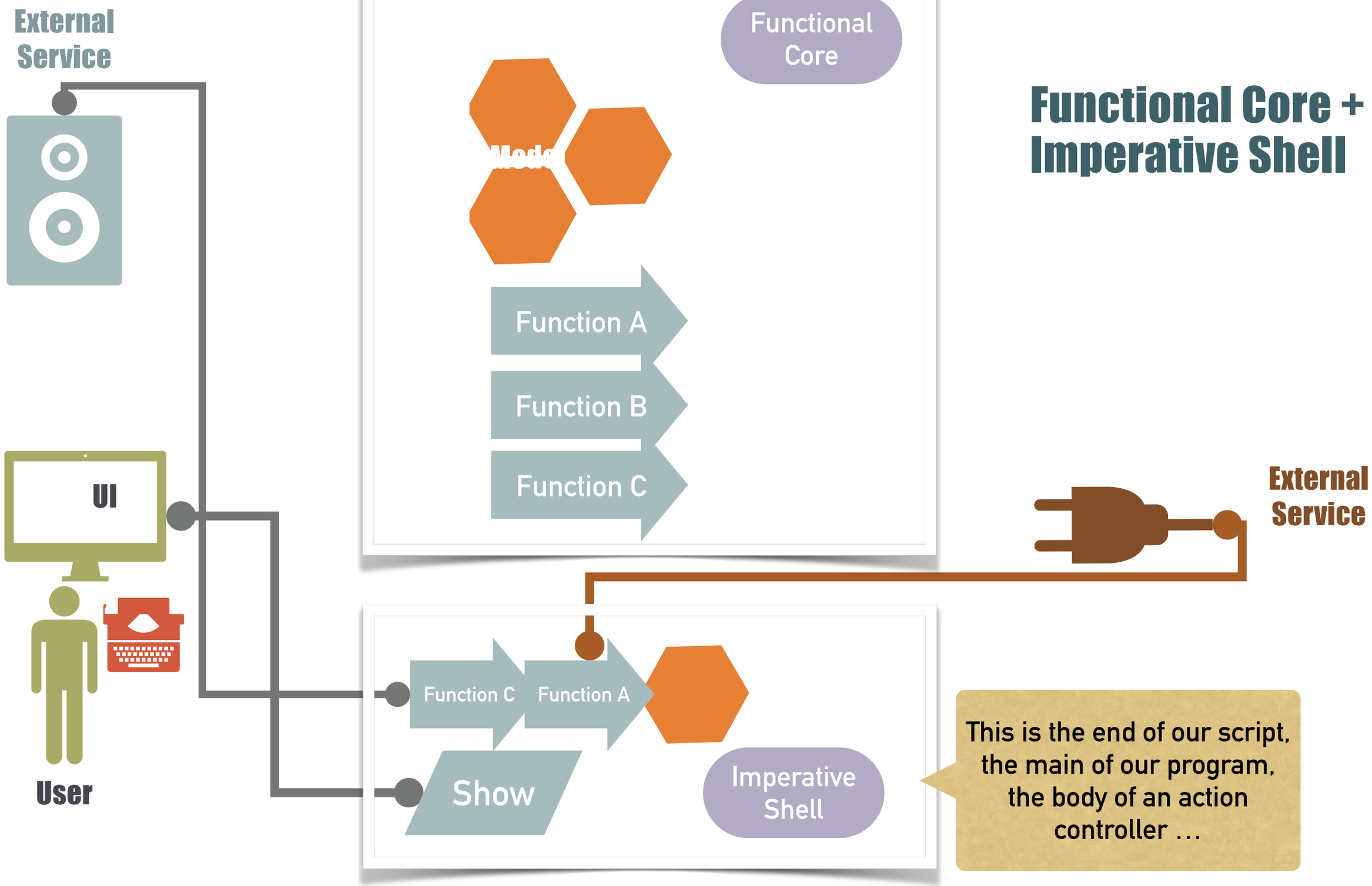
Protect your domain

FUNCTIONAL CORE, IMPERATIVE SHELL

Functional core is the place where you organise your business logic with a purity and simplicity in mind

Imperative shell is the ultimate top layer where you orchestrate everything

Functional Core + Imperative Shell



THIS MODEL IS NOT OBVIOUS BUT IT'S ONE OF THE BEST TO KEEP YOUR DOMAIN PURE AND TESTABLE

BUILD UPON SMALL LOGICAL AND INDEPENDENT MODULES, FUNCTIONS, MODELS

THINK ABOUT HOW THEY MUST COMMUNICATE ON THEIR HIGHER LEVEL OF ABSTRACTION

NEVER CROSS BOUNDARIES AND USE AN INTERNAL KNOWLEDGE OF OTHER MODULE

FUNCTIONAL CORE, IMPERATIVE SHELL

Gist Code

Repl Code

DCI

Back to the roots of Object Oriented programming

Beauty of asymmetry

OBJECTS ARE DIFFERENT THINGS IN DIFFERENT CONTEXTS, WHILE STILL KEEPING ITS IDENTITY

Data Context Interaction (DCI) describe how the evolving use cases integrates with the models within the domain

If we can directly capture key end-user mental models in the code, it radically increases the chances the code will work

-Jim Coplien

HOW DOES IT WORKS ?

SAMPLE

JS GIST

DESIGN PRINCIPLES

Before the big picture ensure you have solid roots

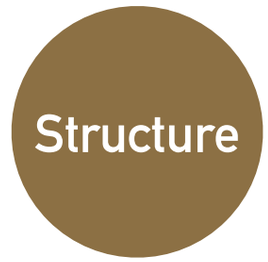
The software industry styles itself on architecture and construction, but rarely discusses aesthetics

-Jim Coplien

PRINCIPLES

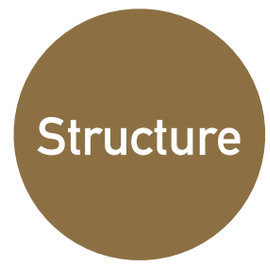
SOLID PRINCIPLES

(at a higher level)



SOLID PRINCIPLES

- Tolerate change
- Easy to understand
- Solid foundations



SRP : SINGLE RESPONSIBILITY

Each software module
should only have one
reason to change

Structure

SRP : SINGLE RESPONSIBILITY

Code

A class should have
one, and only one,
reason to change.

SRP : SINGLE RESPONSIBILITY

Each software module should only have one reason to change

A module should be responsible to one, and only one, actor

Notes

« Cohesion is the force that binds together the code responsible to a single actor »

Structure

LEVELS OF SINGLE RESPONSABILITY

Single Responsibility Principle	Classes & Functions
Common Closure Principle	Components & Modules
Axis Of Change	Architecture

SRP VIOLATION SAMPLE

```
Class Student {
   ShowProfile()
   Save()
   GetAgenda()
   MailParents()
}
```

SRP VIOLATION SAMPLE

```
Class Student {
    ShowProfile()
    Save()
    GetAgenda()
    MailParents()
}
```

User Management

Technical backup

Logistics

Administration

OCP : OPEN CLOSED

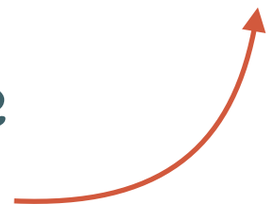
For software systems to be easy to change, they must be designed to allow the behavior to be changed by adding new code and rather than changing existing code

Structure

OCP : OPEN CLOSED

For software systems to be easy to change, they must be designed to allow the behavior to be changed by adding new code and rather than changing existing code

Must be extensible without requiring change



Structure

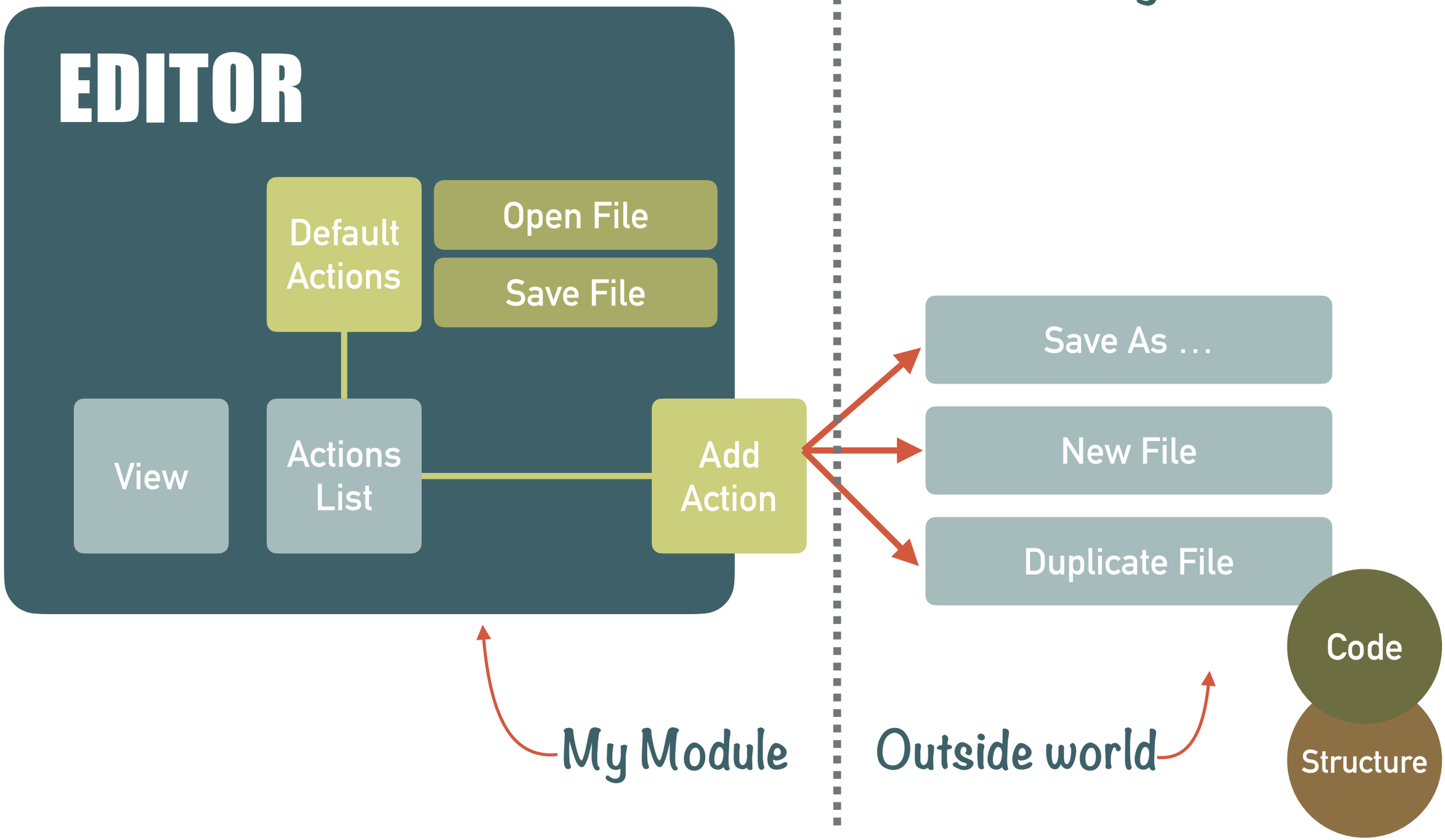
OCP : OPEN CLOSED

Code

You should be able to extend a classes behavior, without modifying it.

OCP : OPEN CLOSED

Must be extensible without requiring change



LSP : LISKOV SUBSTITUTION

In order to build a system with interchangeable parts, those parts must adhere to a contract that allow those parts to be substituted one for another

Structure

LSP : LISKOV SUBSTITUTION

In order to build a system with interchangeable parts, those parts must adhere to a contract that allow those parts to be substituted one for another

Structure

LSP : LISKOV SUBSTITUTION

In order to build a system with interchangeable parts, those parts must adhere to a contract that allow those parts to be substituted one for another

It should guide the use of inheritance and more widely guides the design of interfaces and implementations

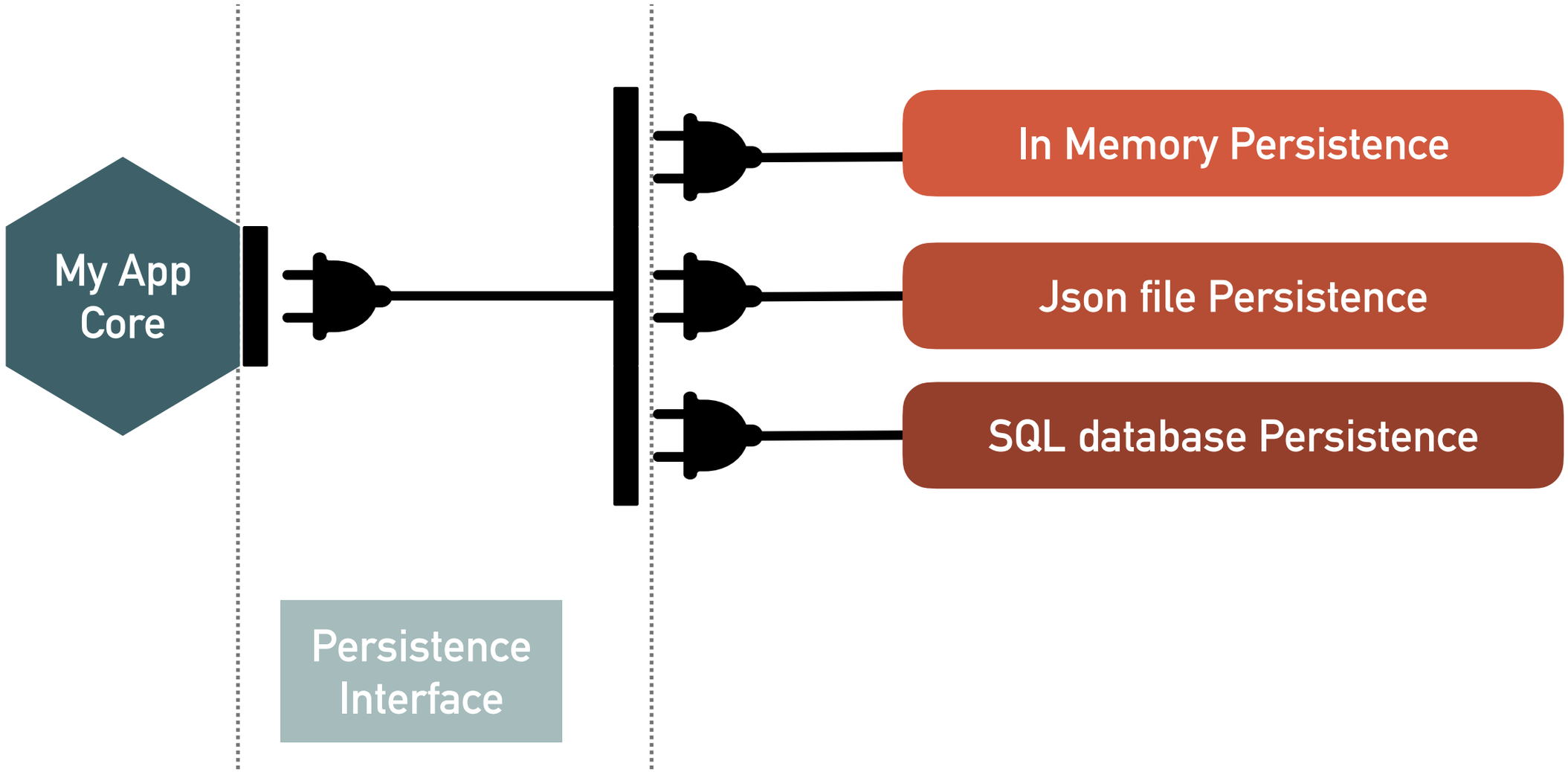
Structure

LSP : LISKOV SUBSTITUTION

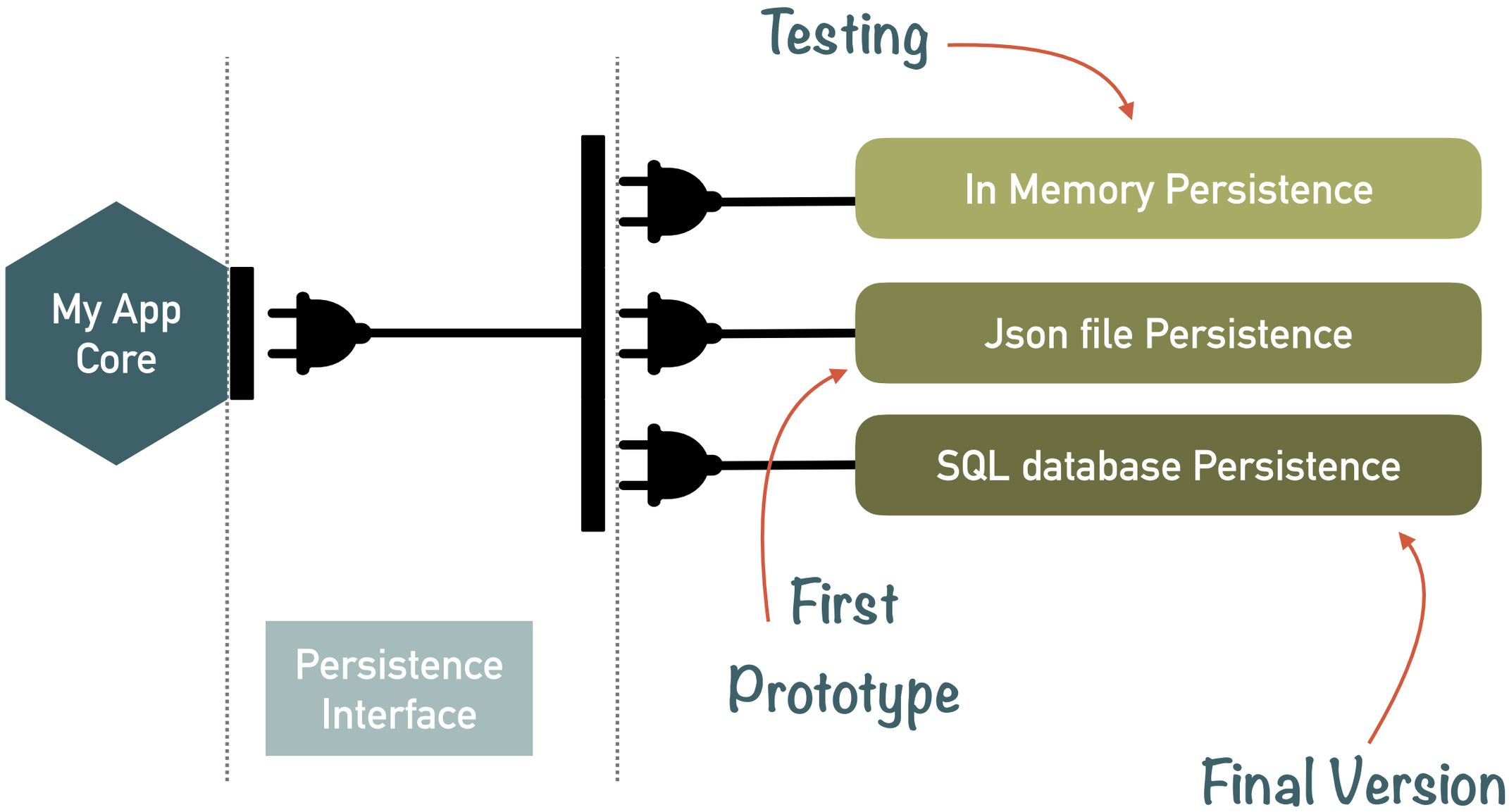
Code

You should be able to extend a classes behavior, without modifying it.

LSP : LISKOV SUBSTITUTION



LSP : LISKOV SUBSTITUTION



A VIOLATION OF LSP WILL BRING A LOT OF ACCIDENTAL COMPLEXITY

ACCIDENTAL COMPLEXITY OVER TIME

```
class StuffOfMyDomain
    {
        public List<Stuff> DoInterestingStuffInDb()
        {
            if(Environment == "test")
            {
                var persistence = new InMemoryPersistence();
                return persistence.GetStuff();
            } else if(Environment == "development")
            {
                var persistence = new JsonFilePersistence("/path/to/my/file.json");
                persistence.LoadFile()
                return persistence.GetStuff();
            } else {
                if(Apptype == "mobile")
                {
                    persistence =  new SqliteStuffProvider("./localfile.db");
                    persistence.Connect();
                    return persistence.GetStuff();

                } else {
                    persistence = new SqlStuffProvider("connectionstring for my db");
                    persistence.Connect();
                    return persistence.GetStuff();
                }
            }
        }
    }
```

MITIGATE ACCIDENTAL COMPLEXITY

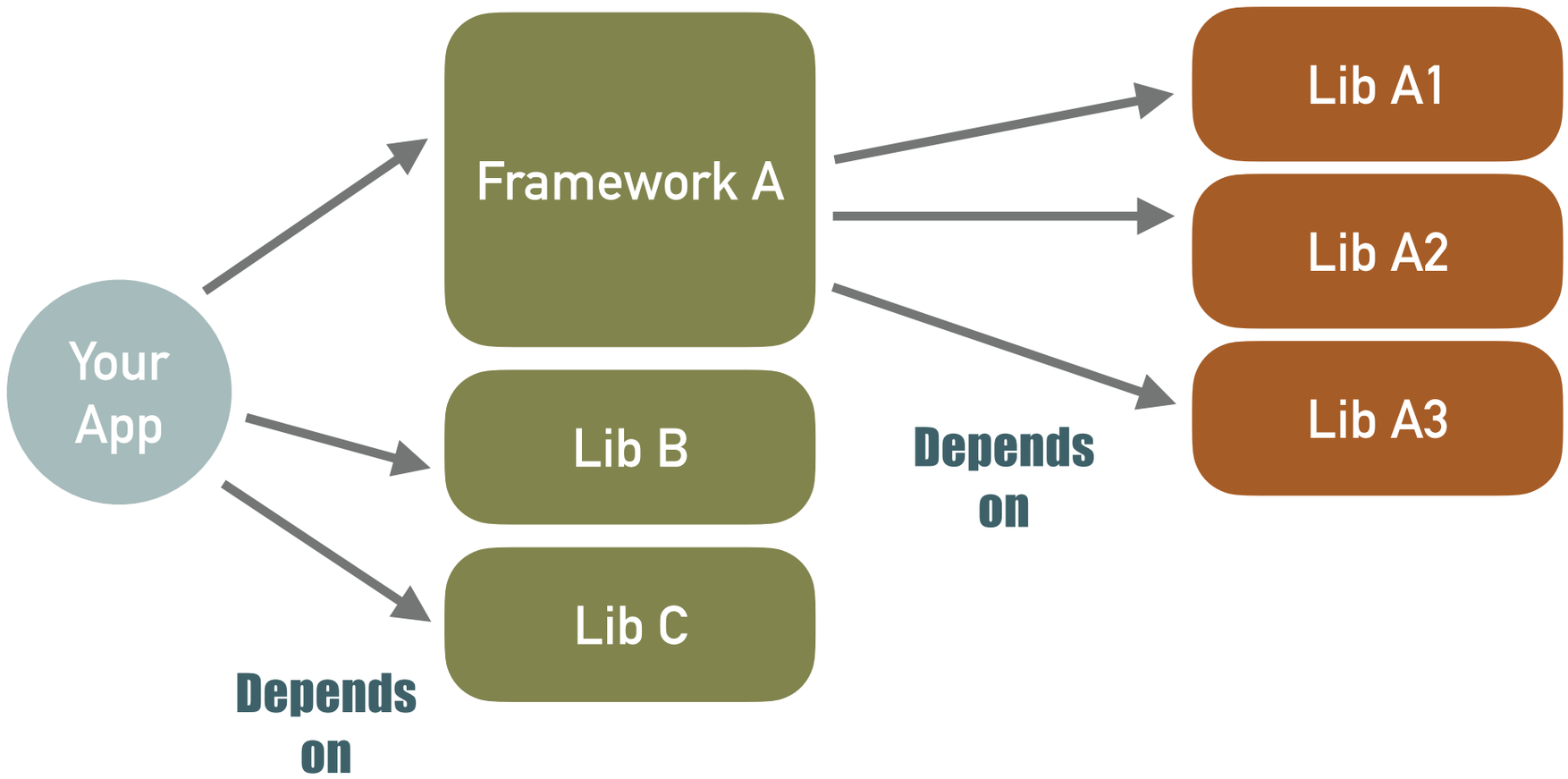
```
class StuffOfMyDomain
{
   IStuffPersistence persistence;

   public StuffOfMyDomain(IStuffPersistence persistence)
      => this.persistence = persistence;

   public List<Stuff> DoInterestingStuffInDb()
      => persistence.GetStuff();
}
```

ISP : INTERFACE SEGREGATION

You shouldn't depend on things you don't use

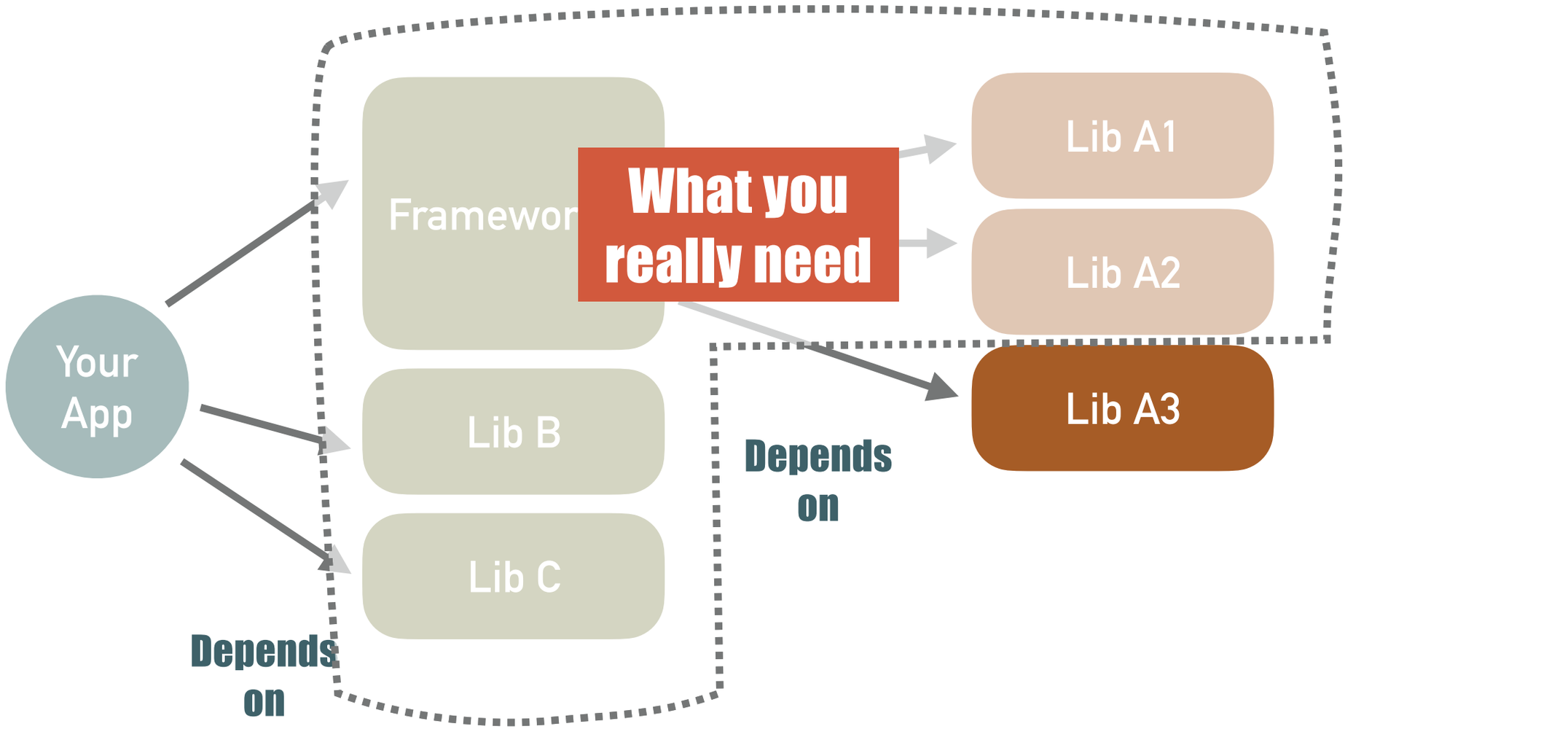
Structure

ISP : INTERFACE SEGREGATION



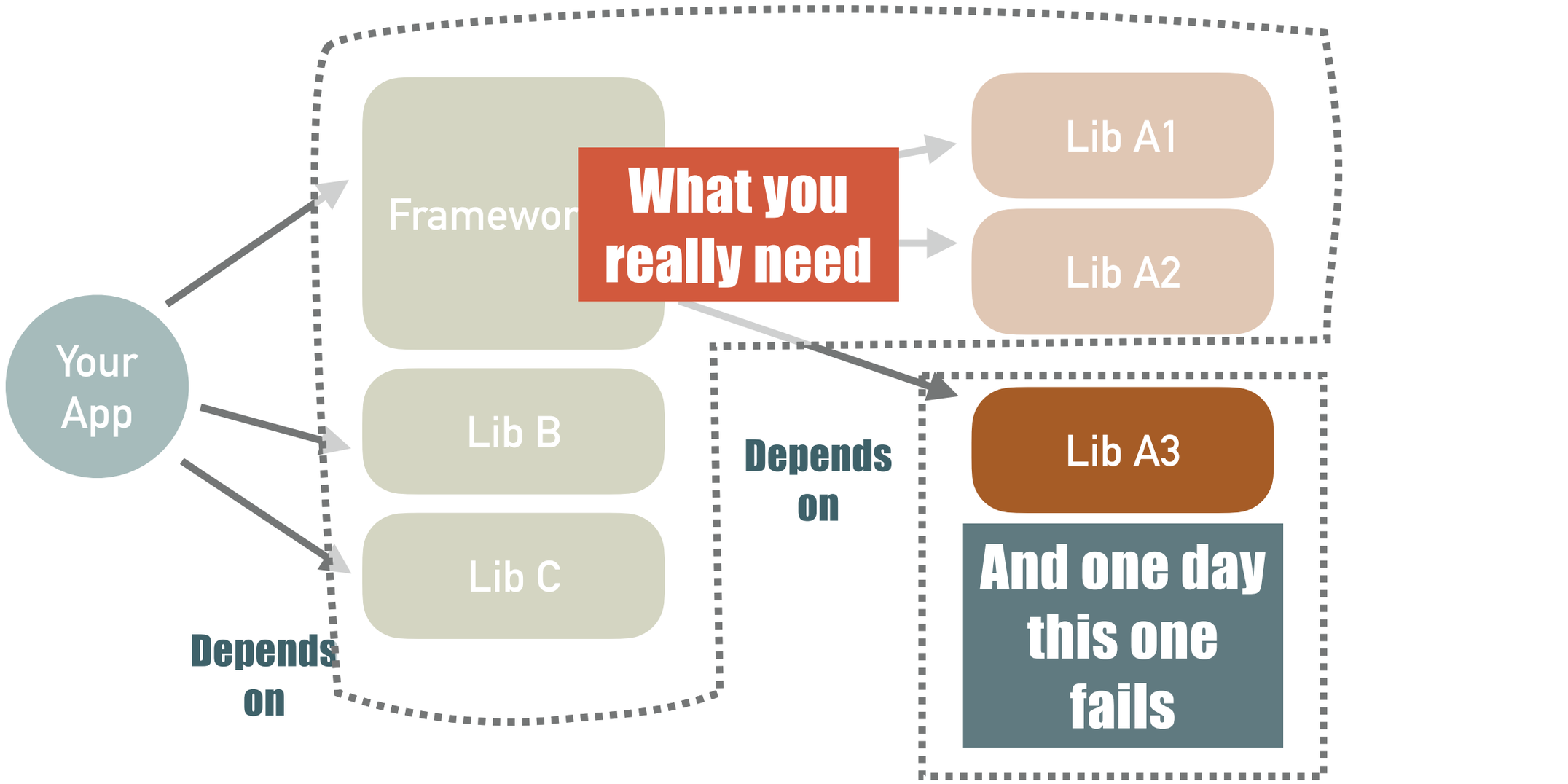
Structure

ISP : INTERFACE SEGREGATION



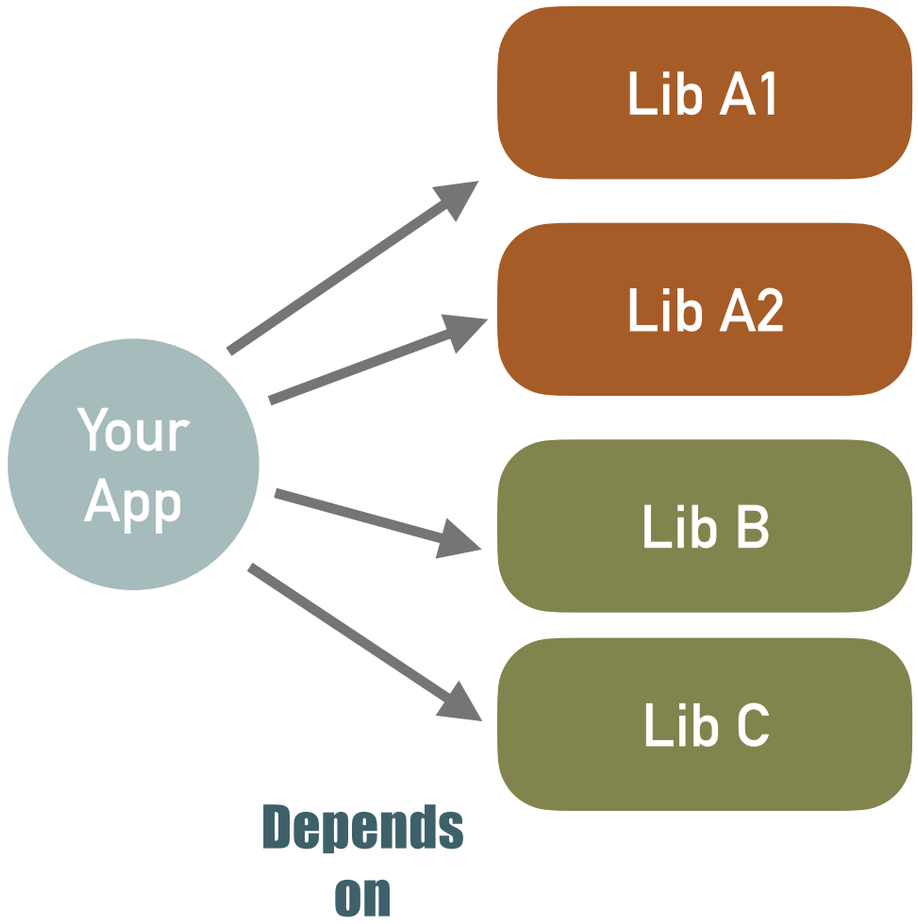
Structure

ISP : INTERFACE SEGREGATION



Structure

ISP : INTERFACE SEGREGATION



Use SMALL direct dependencies

Structure

ISP : INTERFACE SEGREGATION

Code

You should be able to extend a classes behavior, without modifying it.

INSTEAD OF

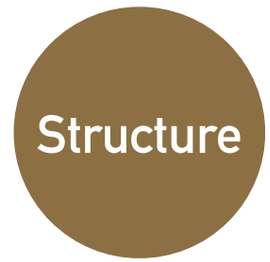
```
interface IRepository
{
    public IEnumerable<Student> GetAllStudents();
    public Student GetOne(string id);
    public bool Save(Student student);
    public bool Remove(Student student);
}

class StudentRepository : IRepository
{
  …
}
```

DO

```
interface IStudentQueries
{
   public IEnumerable<Student> GetAllStudents() {  ... }
   public Student GetOne(string id) { ... }
}
interface IStudentCommands
{
   public bool Save(Student student);
   public bool Remove(Student student);

}
class StudentRepository : IStudentQueries, IStudentCommands
{
  …
}
```

DIP : DEPENDENCY INJECTION

High level structures of code should not depend on the code that implement low level details



DIP : DEPENDENCY INJECTION

Code

Dependencies should be provided when needed

DIP : DEPENDENCY INJECTION

Code

You should depend on

stable abstractions

Don't
- Refer to
- Derive to

→ **volatile** concrete classes

- Override **concrete** functions
- Mention the name of anything concrete and volatile

DIP : DEPENDENCY INJECTION

Code

You should depend on

stable abstractions

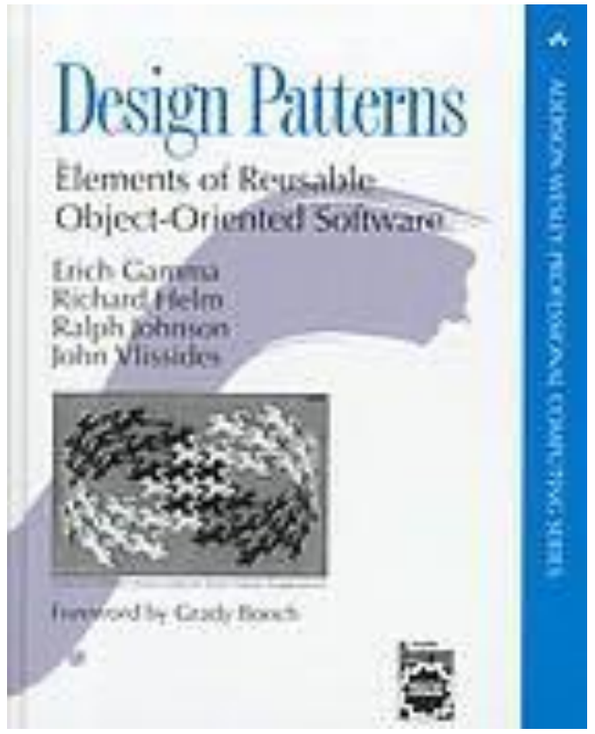
In most OO languages a common practice is to use **Abstract Factories**

PATTERNS

(of object oriented programming)

DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE (1994)

- software engineering book describing **software design patterns**
- written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known as "**Gang of Four**")
- exploring the capabilities and pitfalls of object-oriented programming
- **23** classic software design patterns



Program to an 'interface', not an 'implementation'

Use of an interface also leads to dynamic binding and polymorphism, which are central features of object-oriented programming

Favor 'object composition' over 'class inheritance'

Because inheritance exposes a subclass to details of its parent's implementation, it's often said that 'inheritance breaks encapsulation

Dynamic, highly parameterized software is harder to understand and build than more static software

Distinguish

'**Aggregation**' (object X is or part of Y) and
'**Acquaintance**' (object X merely knows about Y)

DESIGN PATTERNS BOOK

Creational

Abstract factory groups object factories that have a common theme.
Builder constructs complex objects by separating construction and representation.
Factory method creates objects without specifying the exact class to create.
Prototype creates objects by cloning an existing object.
Singleton restricts object creation for a class to only one instance.

Structural

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
Bridge decouples an abstraction from its implementation so that the two can vary independently.
Composite composes zero-or-more similar objects so that they can be manipulated as one object.
Decorator dynamically adds/overrides behaviour in an existing method of an object.
Facade provides a simplified interface to a large body of code.
Flyweight reduces the cost of creating and manipulating a large number of similar objects.
Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral

Chain of responsibility delegates commands to a chain of processing objects.
Command creates objects which encapsulate actions and parameters.
Interpreter implements a specialized language.
Iterator accesses the elements of an object sequentially without exposing its underlying representation.
Mediator allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
Memento provides the ability to restore an object to its previous state (undo).
Observer is a publish/subscribe pattern which allows a number of observer objects to see an event.
State allows an object to alter its behavior when its internal state changes.
Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object .

DESIGN PATTERNS BOOK

Creational

Abstract factory groups object factories that have a common theme.
Builder constructs complex objects by separating co[illegible]ion and re[illegible]
[illegible]ects without [illegible]rying the exact class to create.
Prototype creates objects by cloning an existing object.
Singleton restricts object creation for a class to only one instance.

How to create

Structural

Adapter allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
Bridge decouples an abstraction from its implementation so that the two can vary independently.
Composite composes zero-or-more similar objects so th[illegible] can be manipulate[illegible]t.
Decorator [illegible] ove[illegible]isting me[illegible]
Fac[illegible]es a simplified inter[illegible]e to a large body of code.
Flyweight reduces the cost of creating and manipulating a large number of similar objects.
Proxy provides a placeholder for another object to control access, reduce cost, and reduce complexity.

How to compose

Behavioral

Chain of responsibility delegates commands to a chain of processing objects.
Command creates objects which encapsulate actions and parameters.
Interpreter implements a specialized language.
Iterator accesses the elements of an object sequentially without exposing its underlying representation.
Mediator allows loose coupling[illegible]n classes by being the onl[illegible] detailed knowled[illegible]
Memento[illegible]tore an ob[illegible](undo).
Ob[illegible]n/subscribe pattern which allo[illegible]ber of observer objects to see an even[illegible]
State allows an object to alter its behavior when its internal state changes.
Strategy allows one of a family of algorithms to be selected on-the-fly at runtime.
Template method defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
Visitor separates an algorithm from an object structure by moving the hierarchy of methods into one object .

How to communicate

Attention:

Programs full of visible design patterns could be a sign of trouble

Over engineering

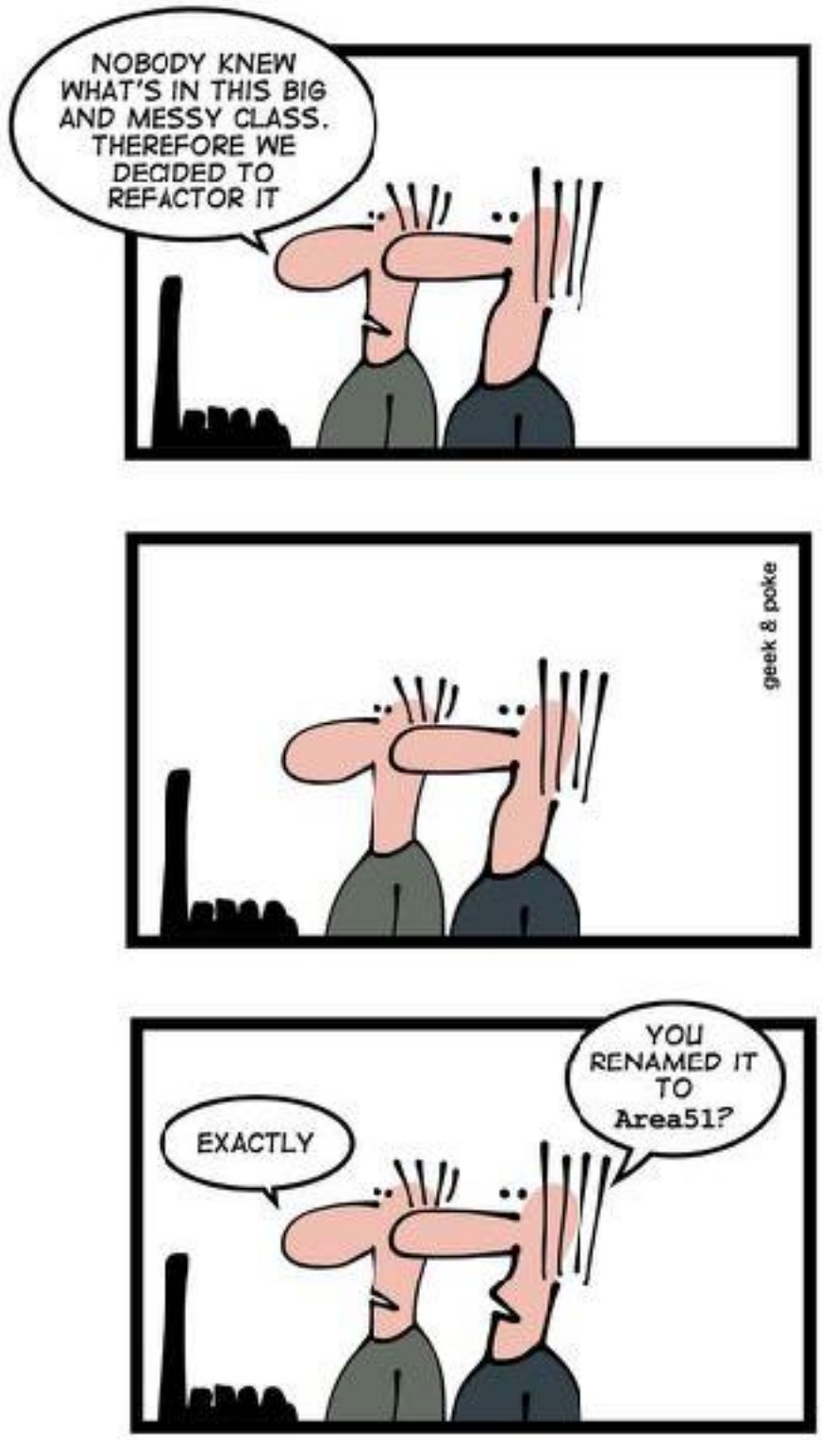
Technical solution over functional design

Accidental complexity

REFACTORING

Keep it clean

REFACTORING IS KEY

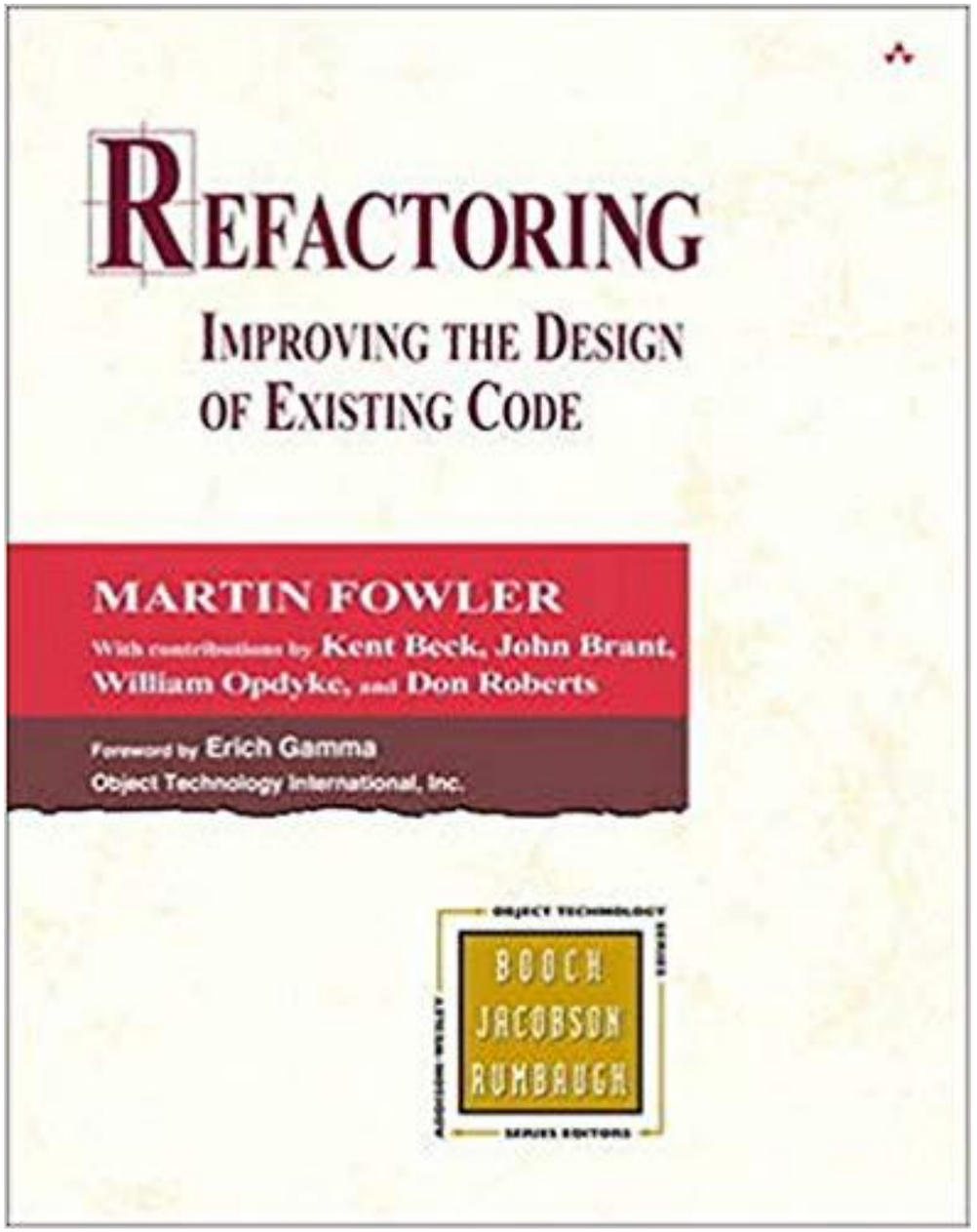


CHANGING THE ARRANGEMENT OF CODE WITHOUT CHANGING IT'S BEHAVIOR

WE BEGIN WITH THE EXISTING CODE AND CHANGE IT OVER TIME AS NEEDED

1. How to identify code smells
2. How to apply refactorings
3. Which refactorings eliminate which code smells

DEEP DIVE ON REFACTORING

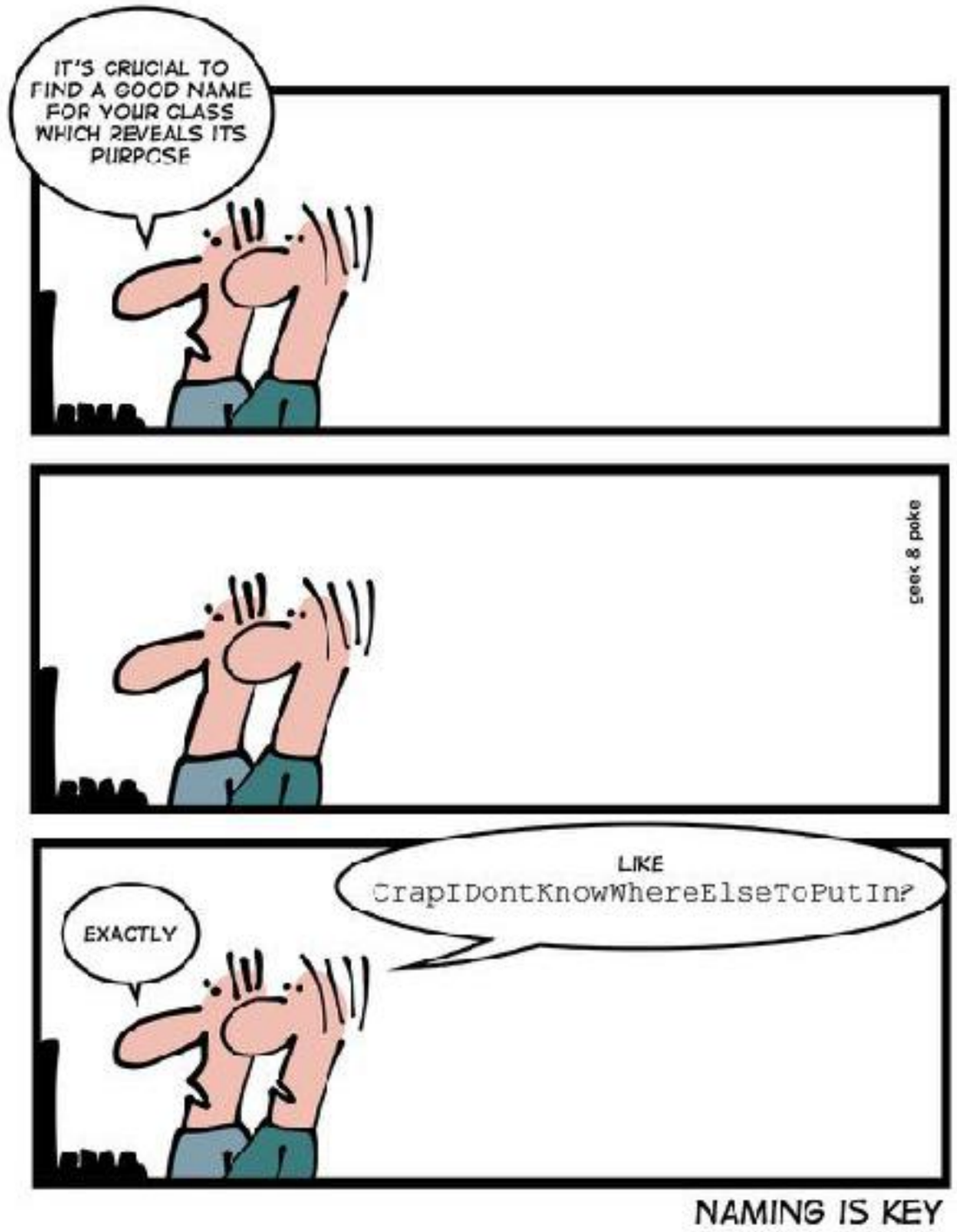


RULES OF SIMPLE DESIGN

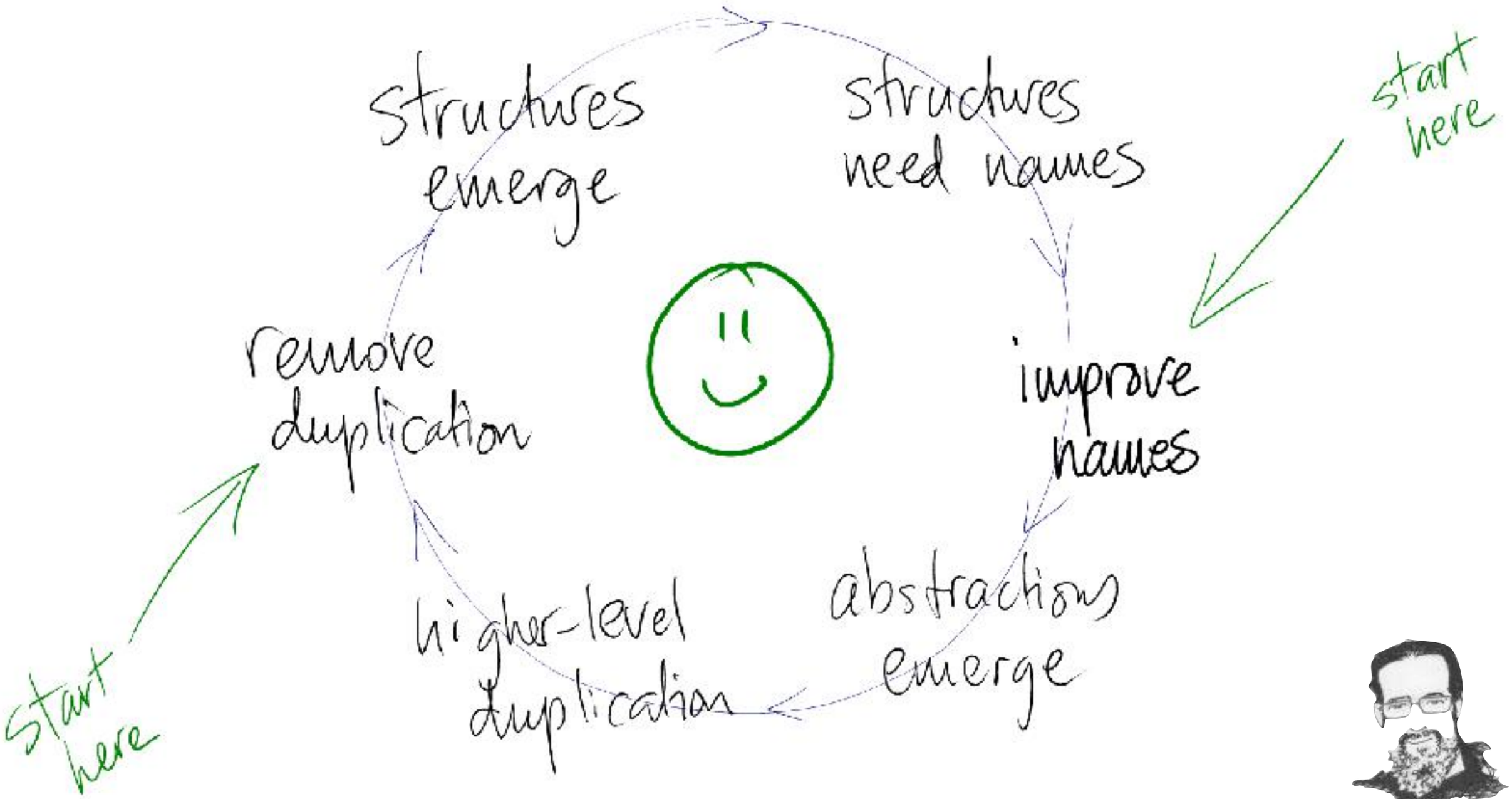
RULES OF SIMPLE DESIGN XP version, by Kent Beck

1. Run all the tests
2. Contain no duplicate code
3. Express all the ideas the author wants to express
4. Minimize classes and methods

NAMING IS KEY

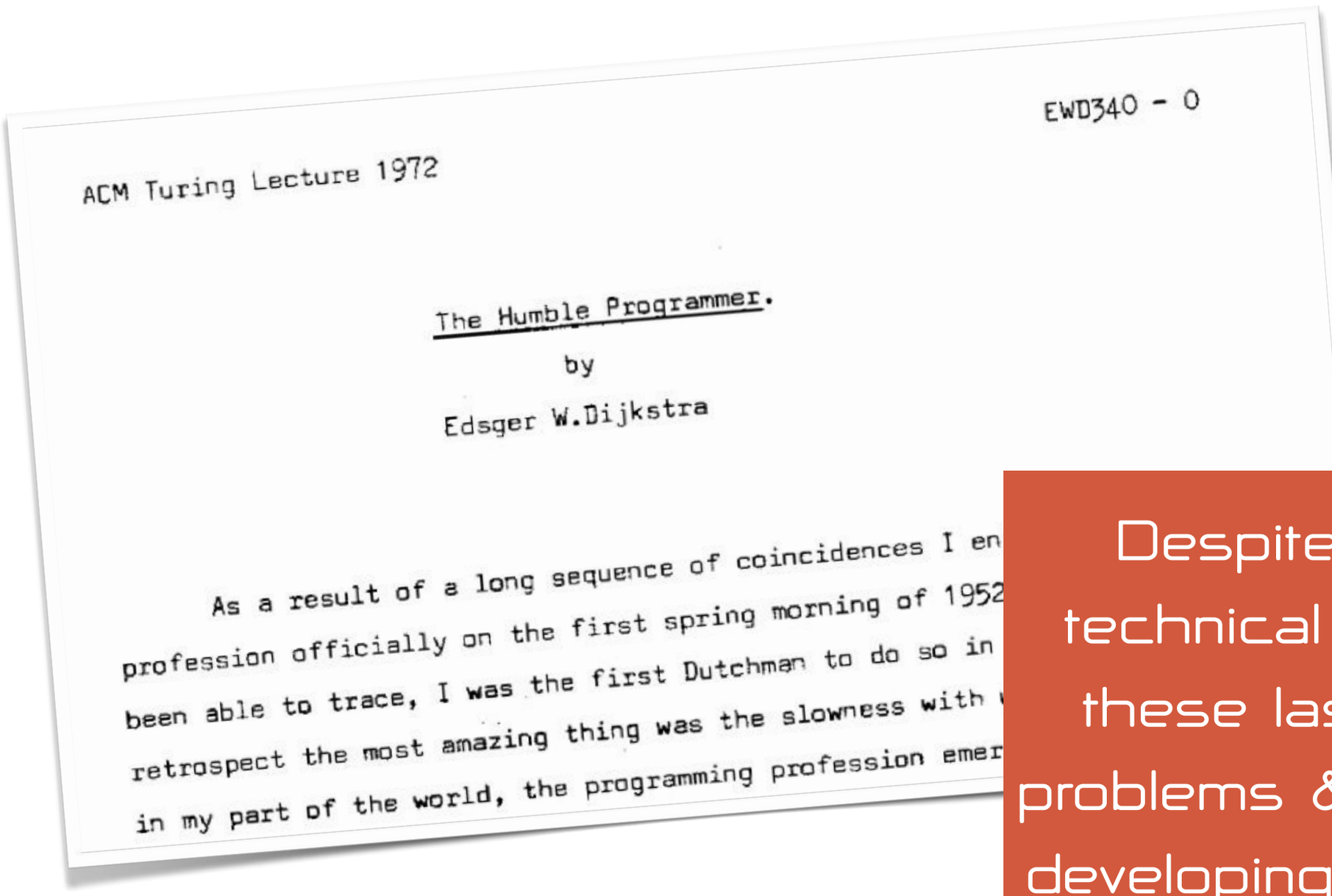


THE SIMPLE DESIGN DYNAMO™



@jbrains

QUICK SUMMARY



ACM Turing Lecture 1972

EWD340 - 0

The Humble Programmer.

by

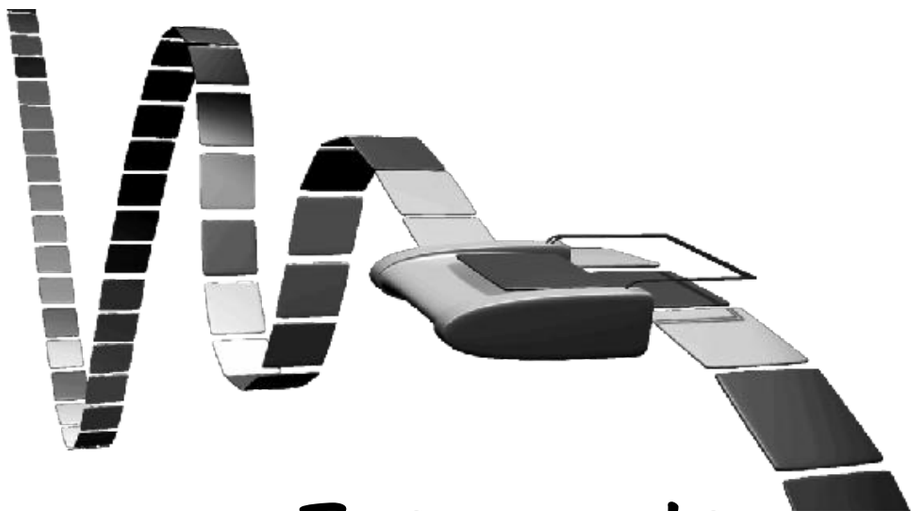
Edsger W.Dijkstra

As a result of a long sequence of coincidences I en
profession officially on the first spring morning of 1952
been able to trace, I was the first Dutchman to do so in
retrospect the most amazing thing was the slowness with
in my part of the world, the programming profession emer

Despite the huge technical progress of these las 60 years, problems & difficulties of developing software are exactly the same

Dijkstra / 1972

VON NEUMANN ARCHITECTURE

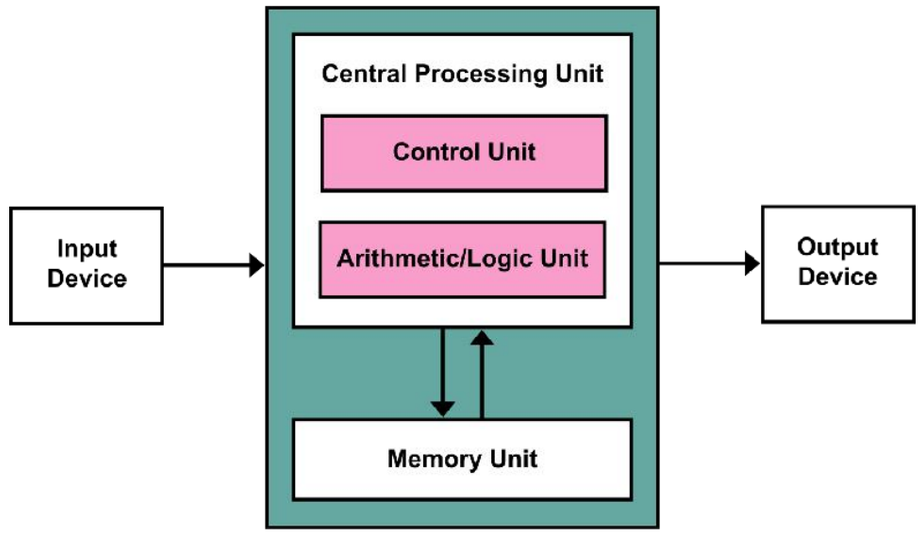


Turing machine

PARADIGMS

Functional Object Strucured

At different levels, all these ‘oldies’ still have an impact on structures of programs



Von Neumann architecture

System

THINK ARCHITECTURE

OUR IDEA OF GOOD DESIGN IS A MIXED FEELING **ABOUT**
OUR OWN MENTAL MODEL,
AESTHETICS,
THEORY,
THE REAL WORLD

BEING AN ARCHITECT MEANS CREATING ARCHITECTURE WHICH MEANS TAKE CONSCIOUS DECISIONS ABOUT THE STRUCTURE AND FORM

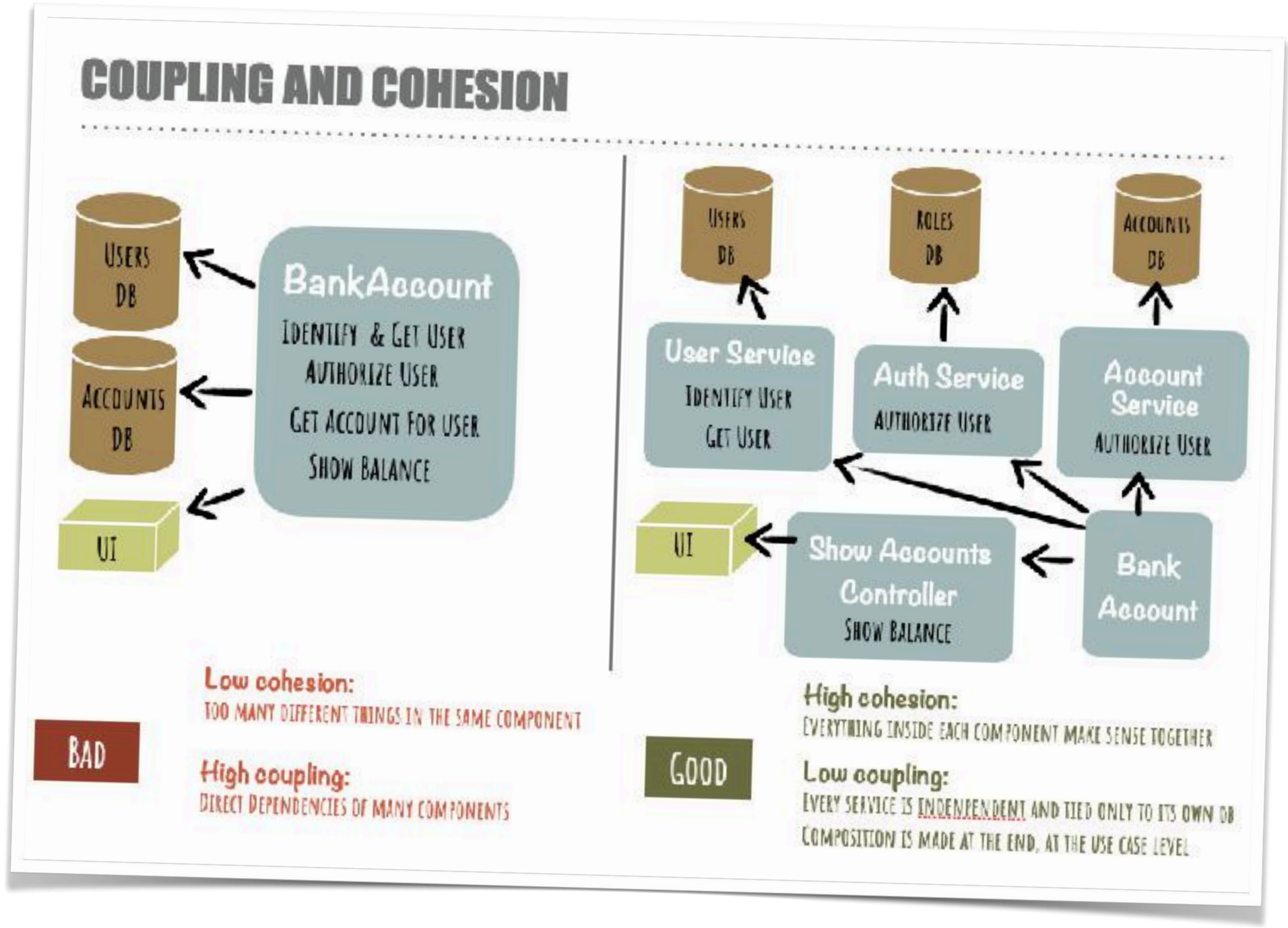
COUPLING & COHESION

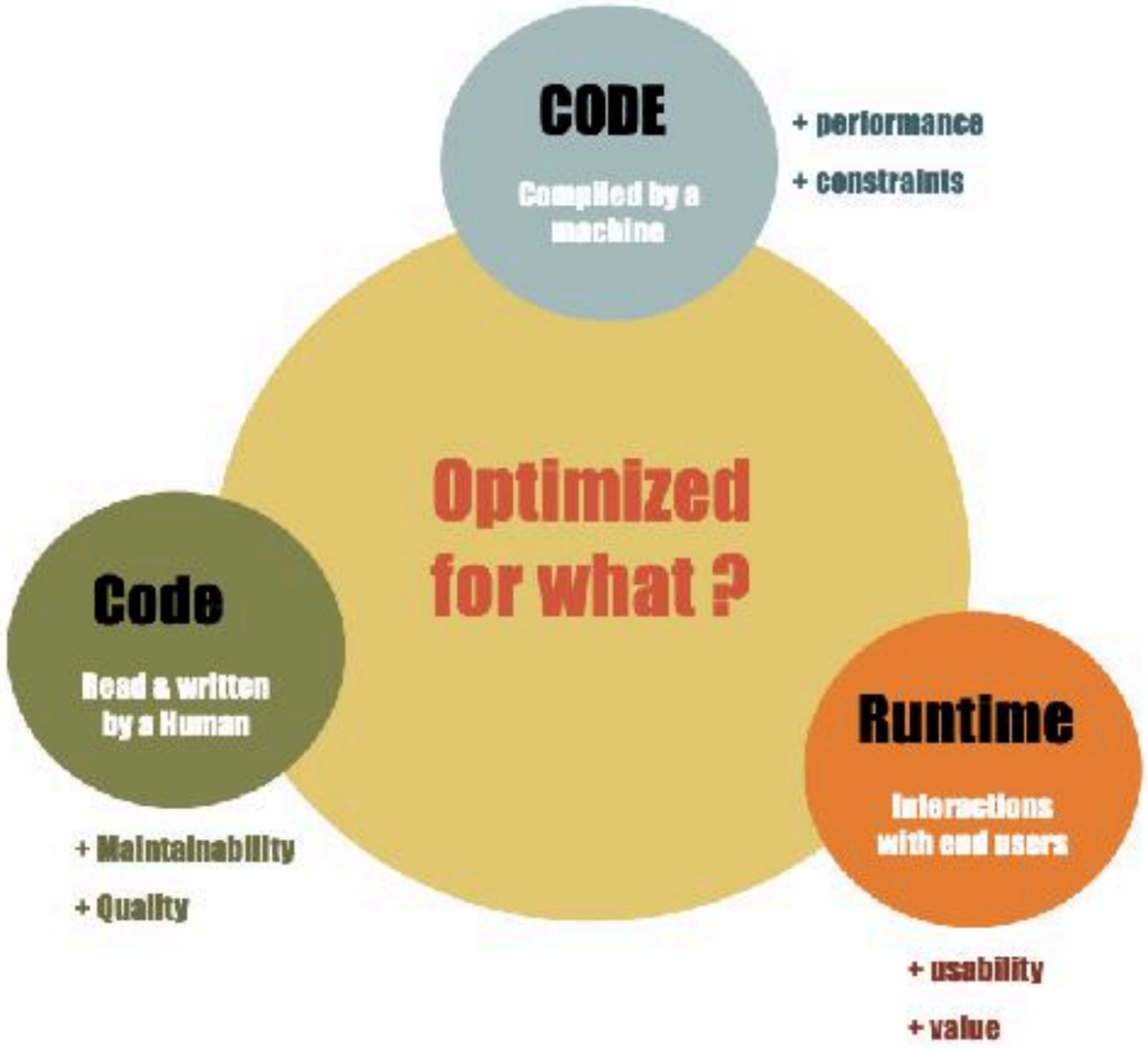
	COHESION	COUPLING
CONCEPT	internal	external
RELATIONSHIP	between modules	within module
REPRESENTS	functional strength	independence among modules
BEST SOFTWARE	Highly Cohesive	Loosely coupled

more of that

less of that

OVERALL THE MOST IMPORTANT :

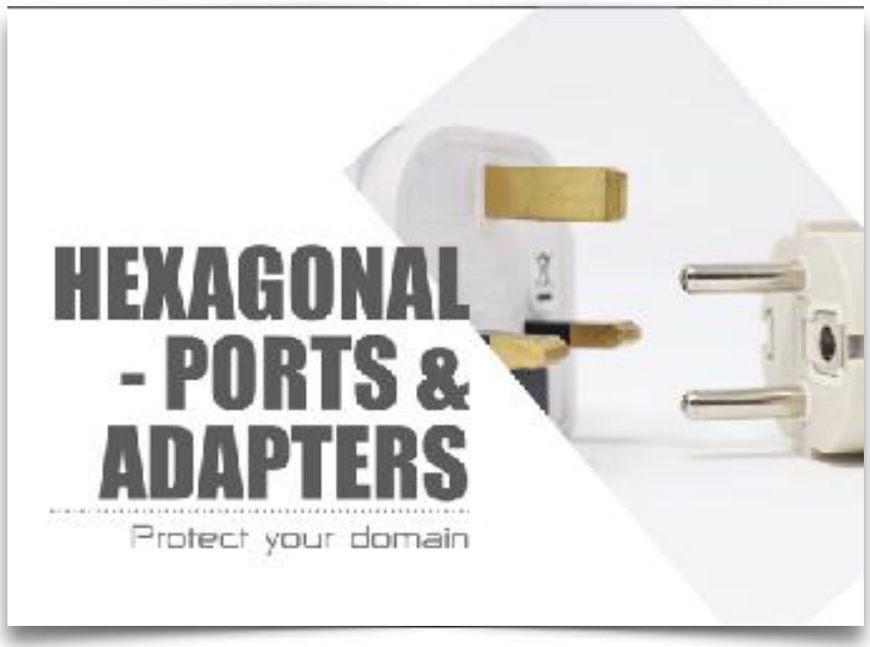




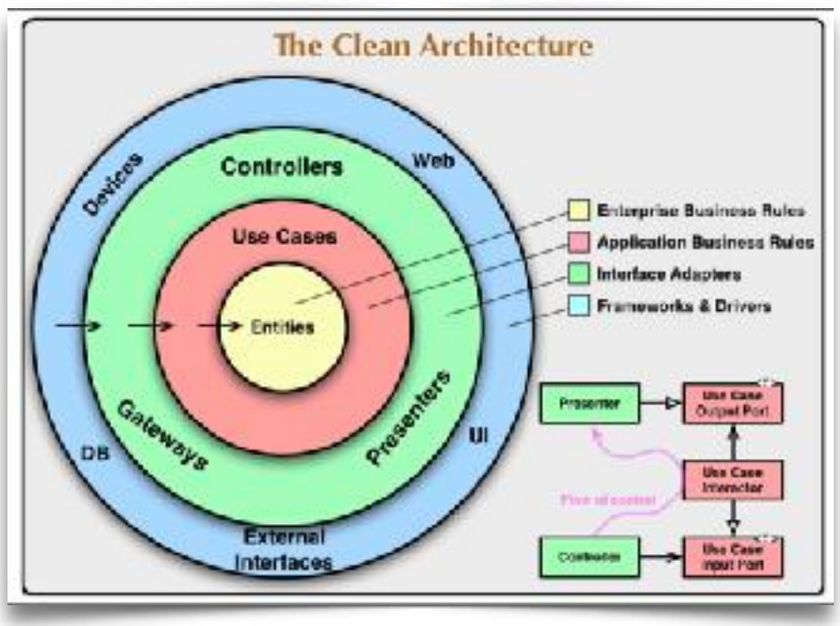
ARCHITECTURE STYLES

Different styles for different purposes

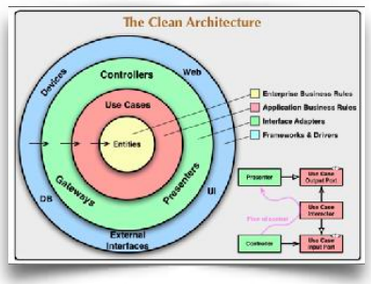
ARCHITECTURE STYLES

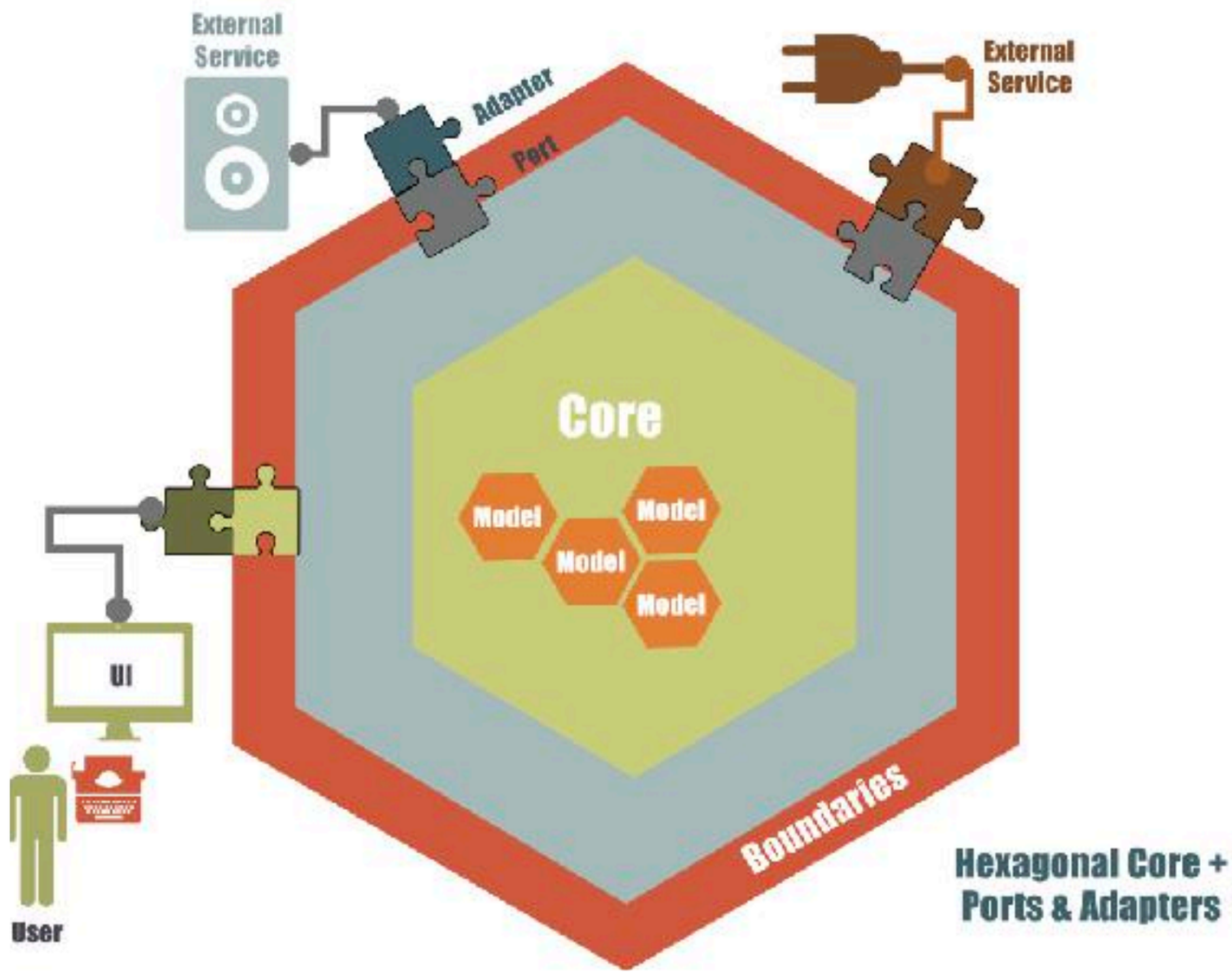


ARCHITECTURE STYLES

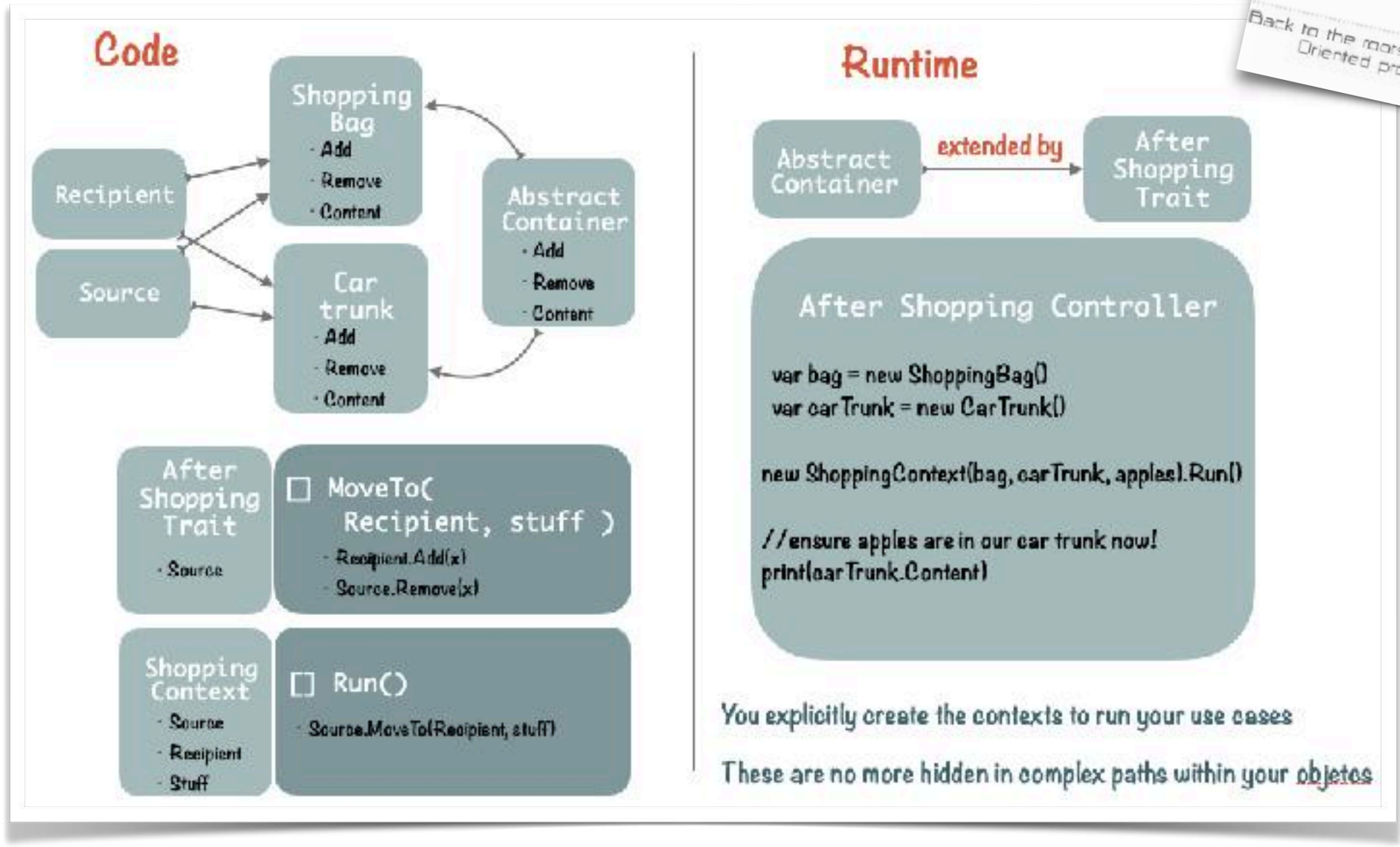


ARCHITECTURE STYLES



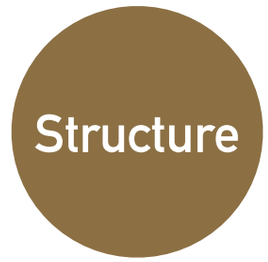


FOCUS ON USE CASES



SOLID PRINCIPLES

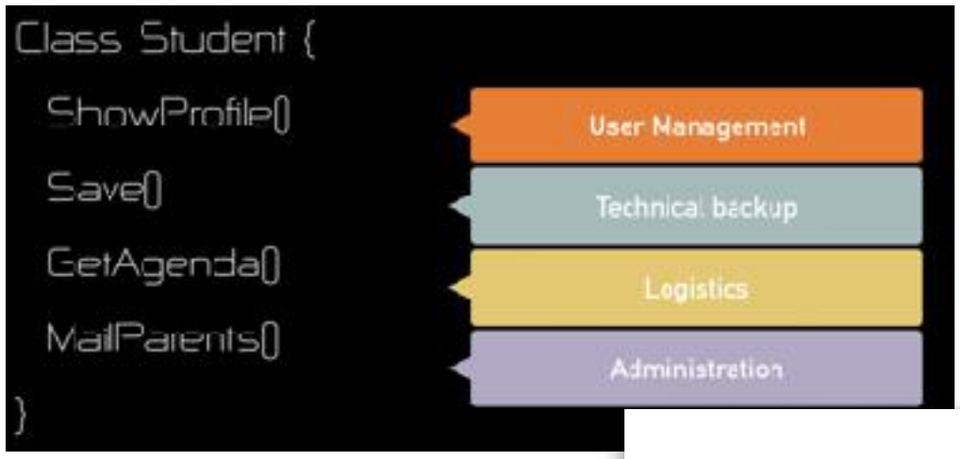
(at a higher level)

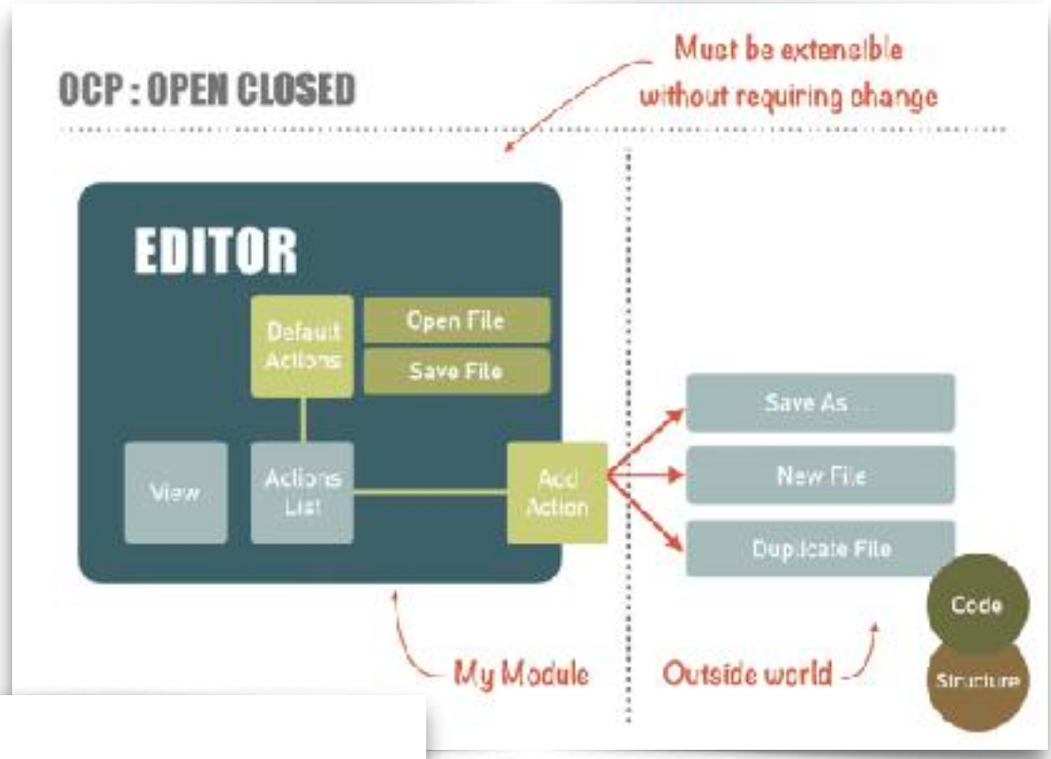


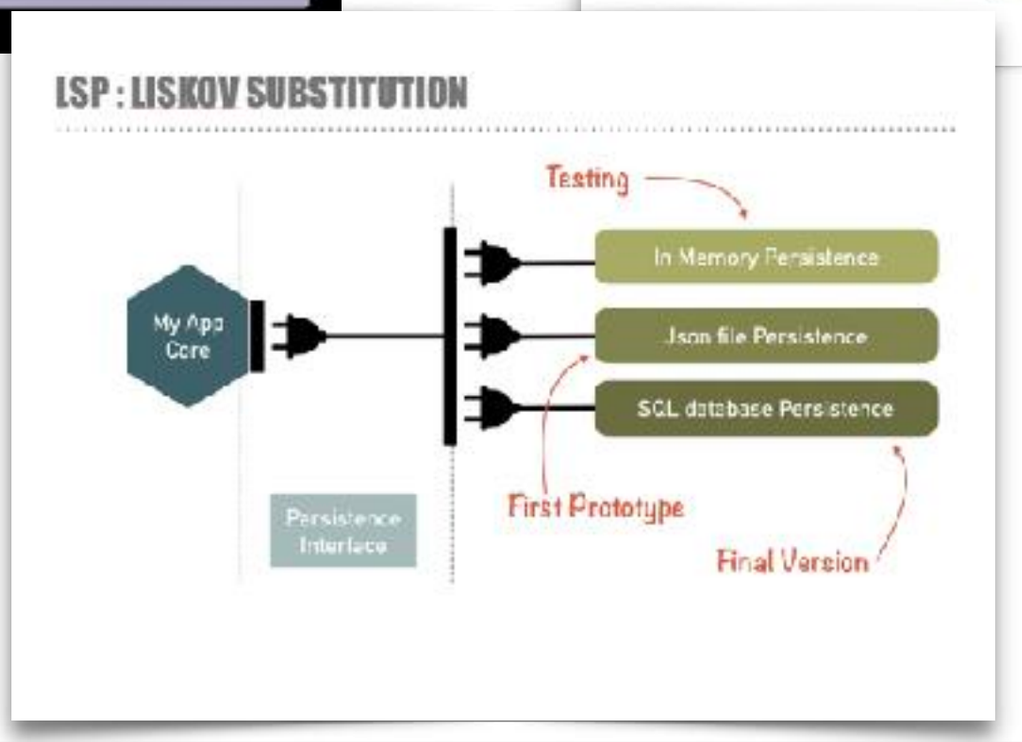
SOLID PRINCIPLES

A class should have one, and only one, reason to change.

SRP

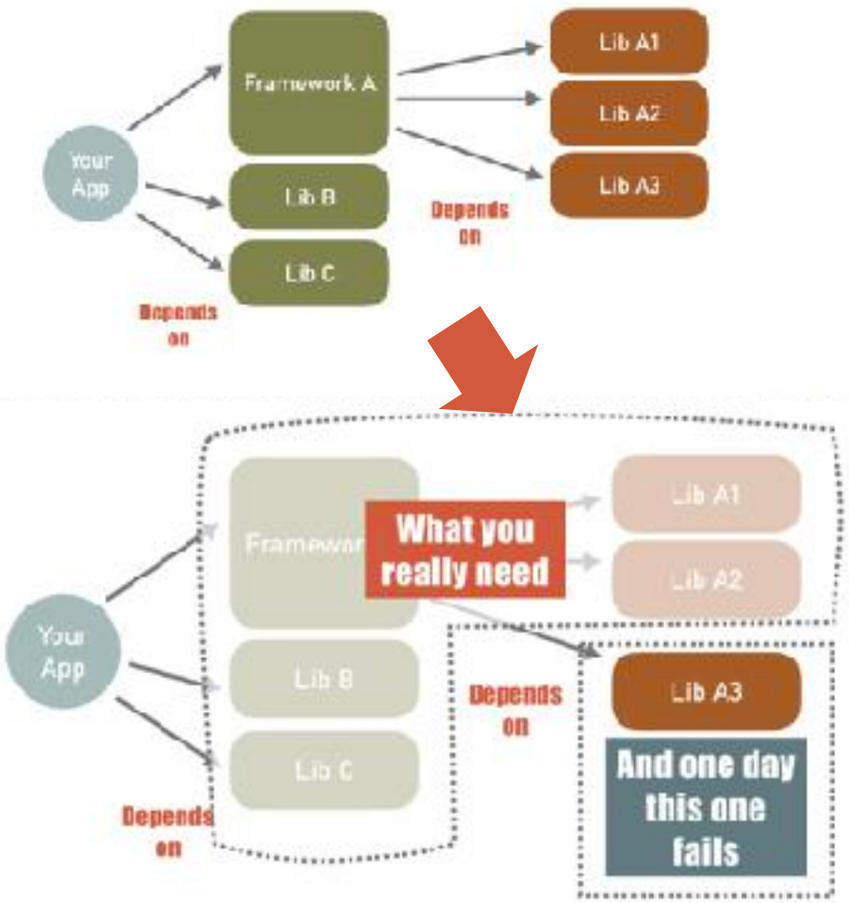






SOLID PRINCIPLES

ISP : INTERFACE SEGREGATION



INSTEAD OF

```
interface IRepository
{
   public IEnumerable<Student> GetAllStudents();
   public Student GetOne(string id);
   public bool Save(Student student);
   public bool Remove(Student student);
}

class StudentRepository : IRepository
{
  …
}
```

DO

```
interface IStudentQueries
{
   public IEnumerable<Student> GetAllStudents() {  ... }
   public Student GetOne(string id) { ... }
}
interface IStudentQueries
{
   public bool Save(Student student);
   public bool Remove(Student student);

}
class StudentRepository : IStudentQueries, IStudentQueries
{
  …
}
```

SOLID PRINCIPLES

DIP : DEPENDENCY INJECTION

Code

You should depend on

stable abstractions

Don't
- Refer to
- Derive to

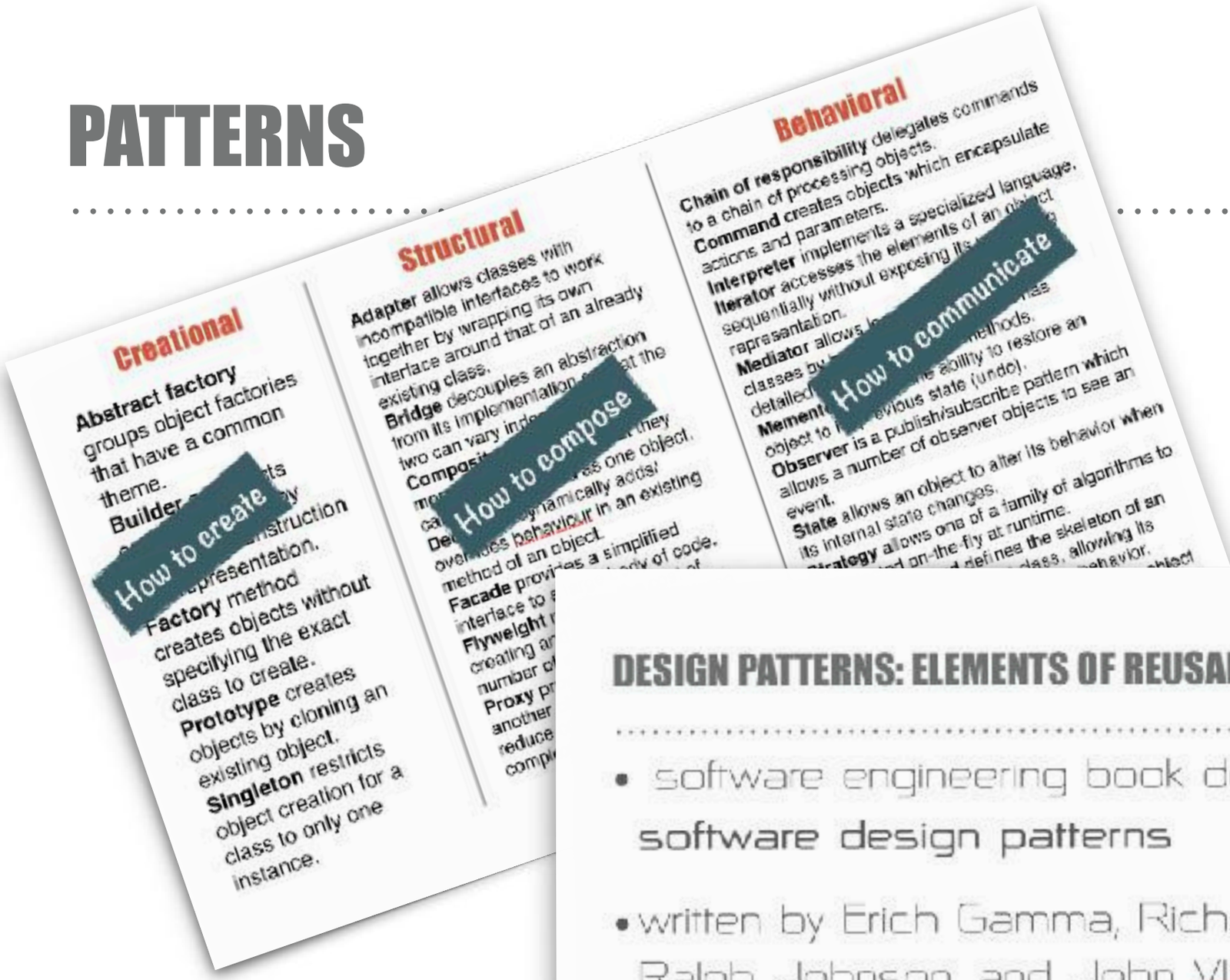
volatile concrete classes

- Override **concrete** functions
- Mention the name of anything concrete and volatile

In most OO languages a common practice is to use **Abstract Factories**

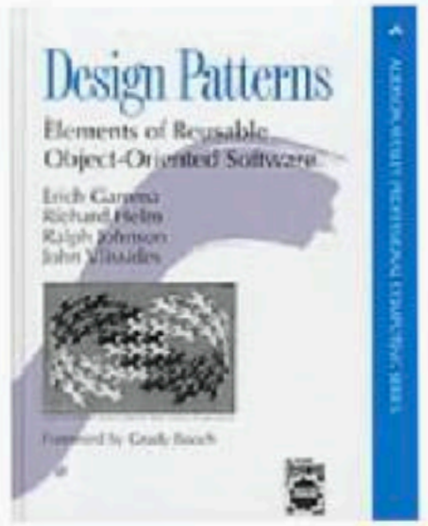
Dependencies should be provided when needed

PATTERNS

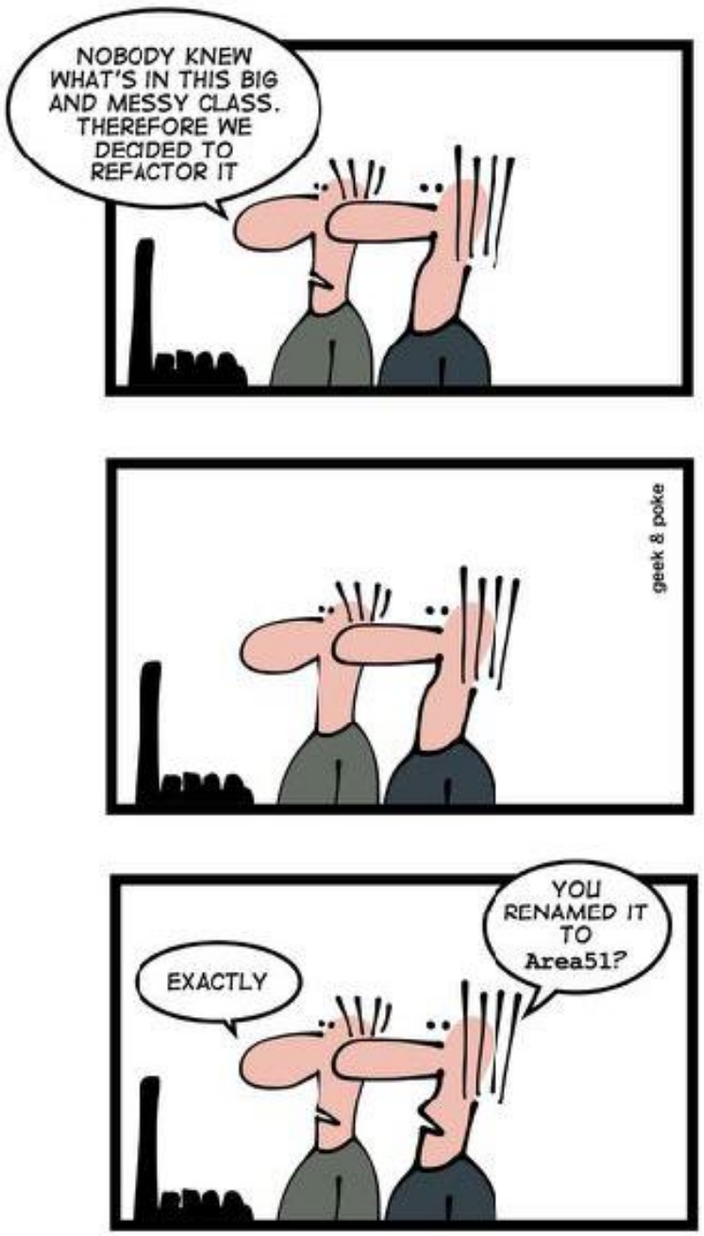


DESIGN PATTERNS: ELEMENTS OF REUSABLE OBJECT-ORIENTED SOFTWARE (1994)

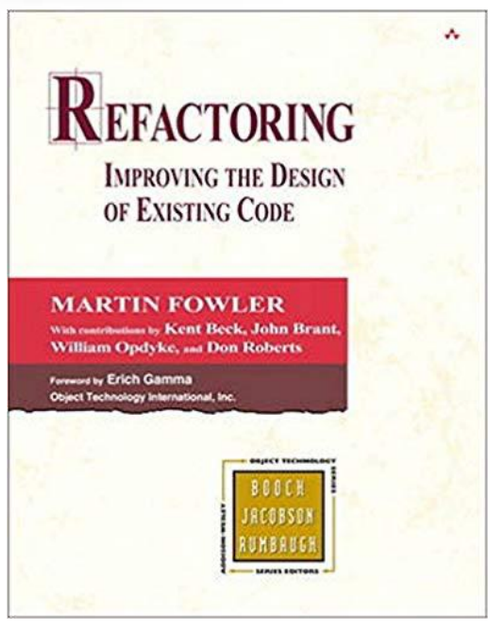
- software engineering book describing **software design patterns**
- written by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (known as "**Gang of Four**")
- exploring the capabilities and pitfalls of object-oriented programming
- 23 classic software design patterns



REFACTORING

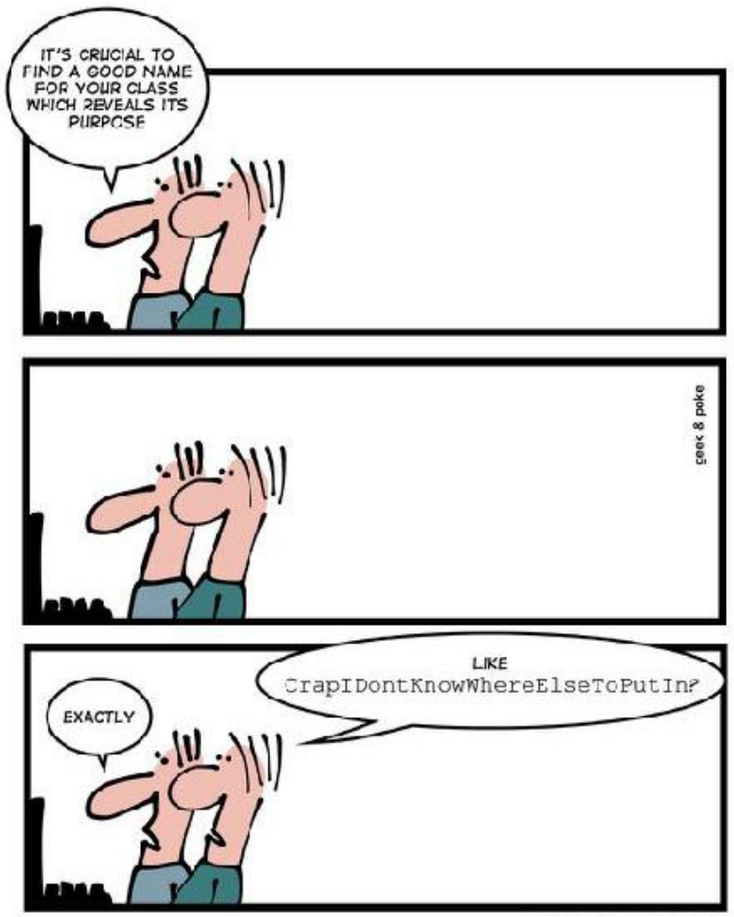


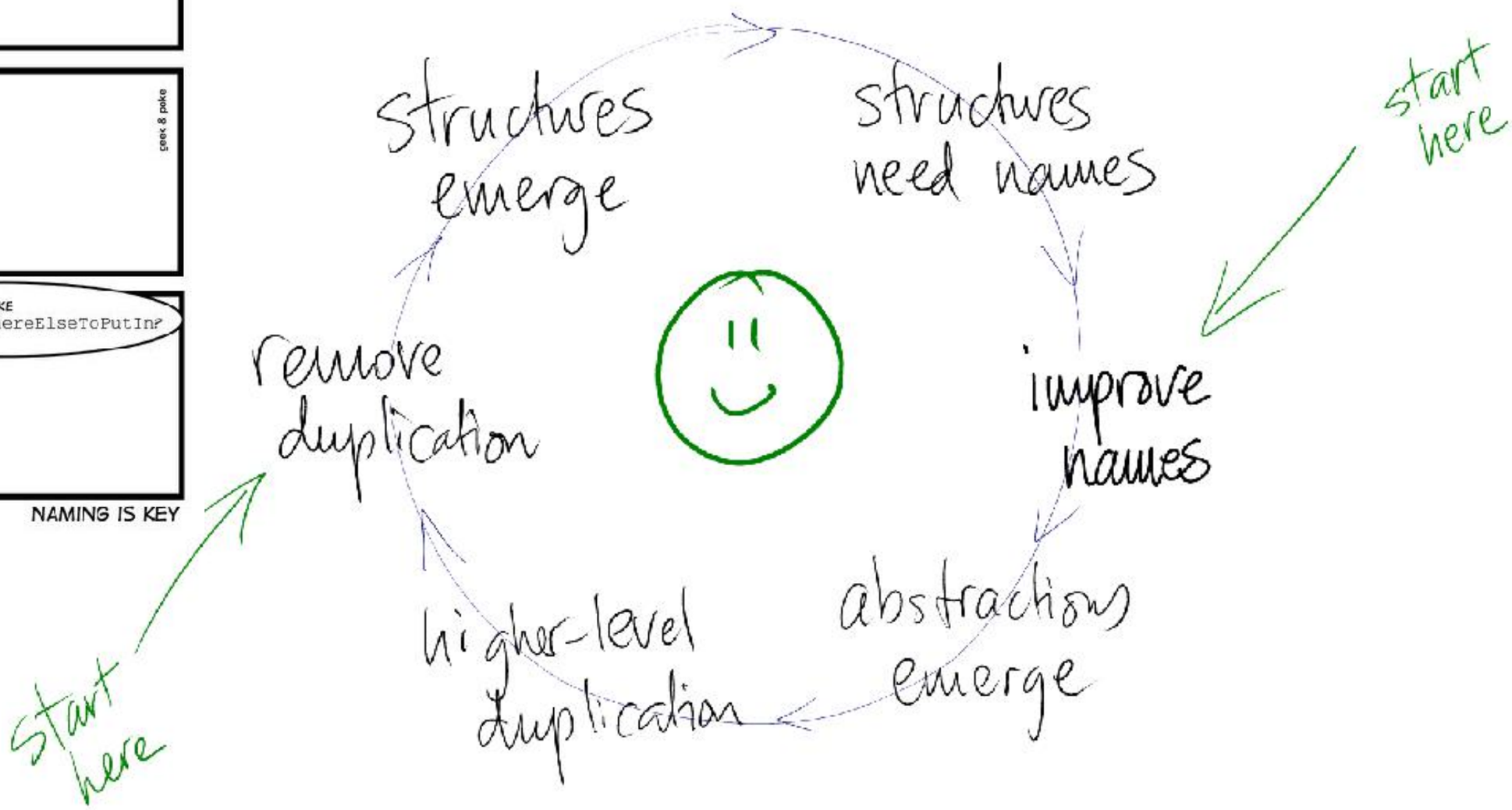
CHANGING THE ARRANGEMENT OF CODE WITHOUT CHANGING IT'S BEHAVIOR



1. How to identify code smells
2. How to apply refactorings
3. Which refactorings eliminate which code smells

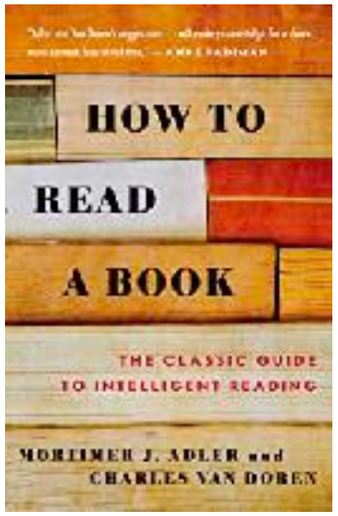
RULES OF SIMPLE DESIGN





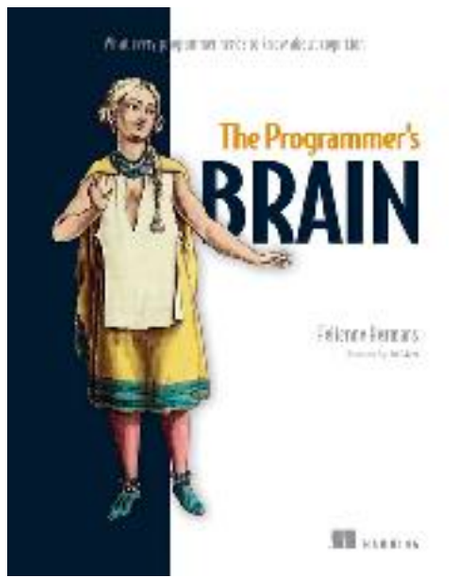
BOOKS

BOOKS TO READ 1 : PREPARE YOUR MIND



How to learn, how to use books and information to create our knowledge

How to use notes as the extension of our mind to create a meaningful personal knowledge system

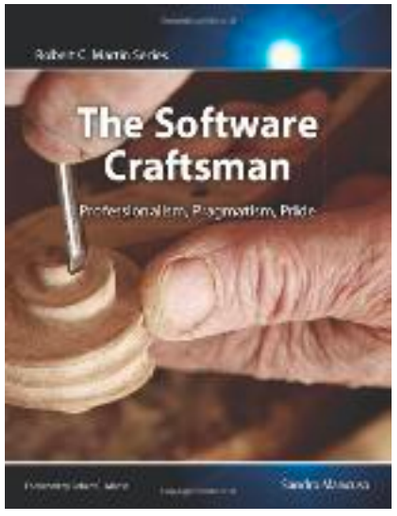


How our brain works, our limitations and how to make software fit in our heads

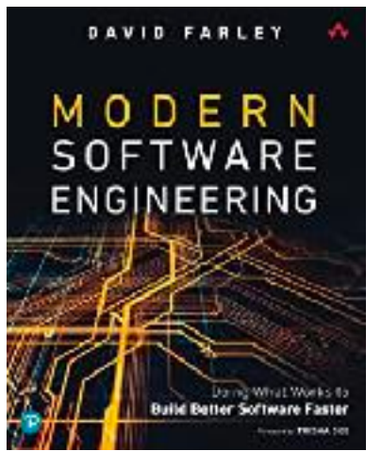
History of people building software since the beginning, give perspective on why things work that way today

BOOKS TO READ 2 : THE ESSENCE OF DEVELOPMENT

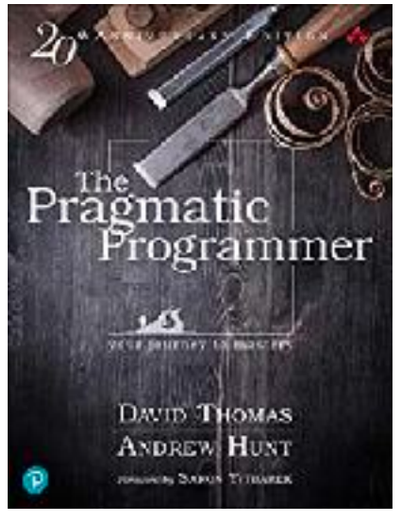
Simple and clear perspective on the nature of software by Ron Jeffries



How to be a professional developer, our career and have pride and joy in our work



A modern view on the essence of software engineering and what matters most for building software

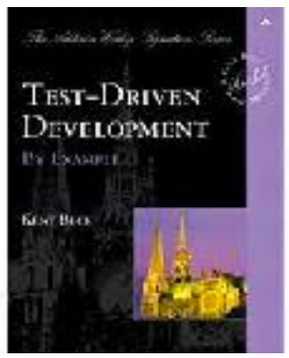


The first book describing what is our job, the nature of development and important practices

BOOKS TO READ 3 : PRACTICES

Kent Beck book that changed the way we see building software and introduced technical agility, tdd and much more....

The most complete book on how to test things with code



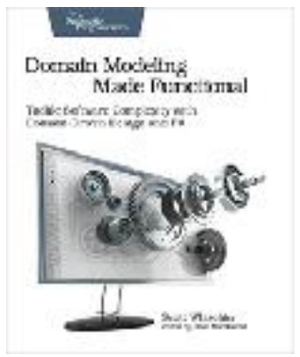
A conversation with Kent beck showing how he use TDD to write code

The fundamental book about refactoring techniques by Martin Fowler

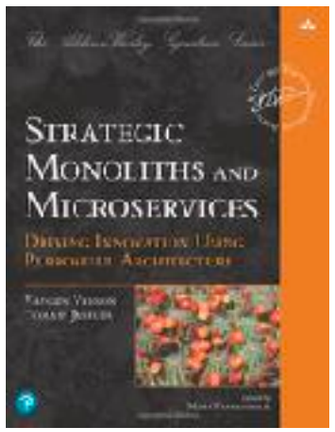
BOOKS TO READ 4 : HIGHER LEVEL

A novel about software and devops

A very fundamental book on agile architecture by the pioneer of patterns and agility Jim Coplien



Best practical book on DDD with lots of nice ideas from functional programming



A modern and complete approach of designing software based on DDD, socio-technical team topologies...

QUOTES

(temp)

*It doesn't make sense to hire **smart people** and tell them what to do; we hire **smart people** so they can tell us what to do.*

-Steve Jobs

LINKS

LINKS

- https://en.wikipedia.org/wiki/Multitier_architecture
- https://martinfowler.com/bliki/CQRS.html
- https://martinfowler.com/tags/application%20architecture.html
- http://ub-prod01-imgs.uio.no/arkiv/dns/hopl78/ (Nygaard talk on Simula in 78')
- https://blog.jbrains.ca/permalink/refactoring-where-do-i-start

- https://github.com/malk/the-kebab-kata

PEOPLE

You should know about

TODO ::

The first three package principles are about package cohesion, they tell us what to put inside packages:

REP
The Release Reuse Equivalency Principle
The granule of reuse is the granule of release.
CCP
The Common Closure Principle
Classes that change together are packaged together.
CRP
The Common Reuse Principle
Classes that are used together are packaged together.

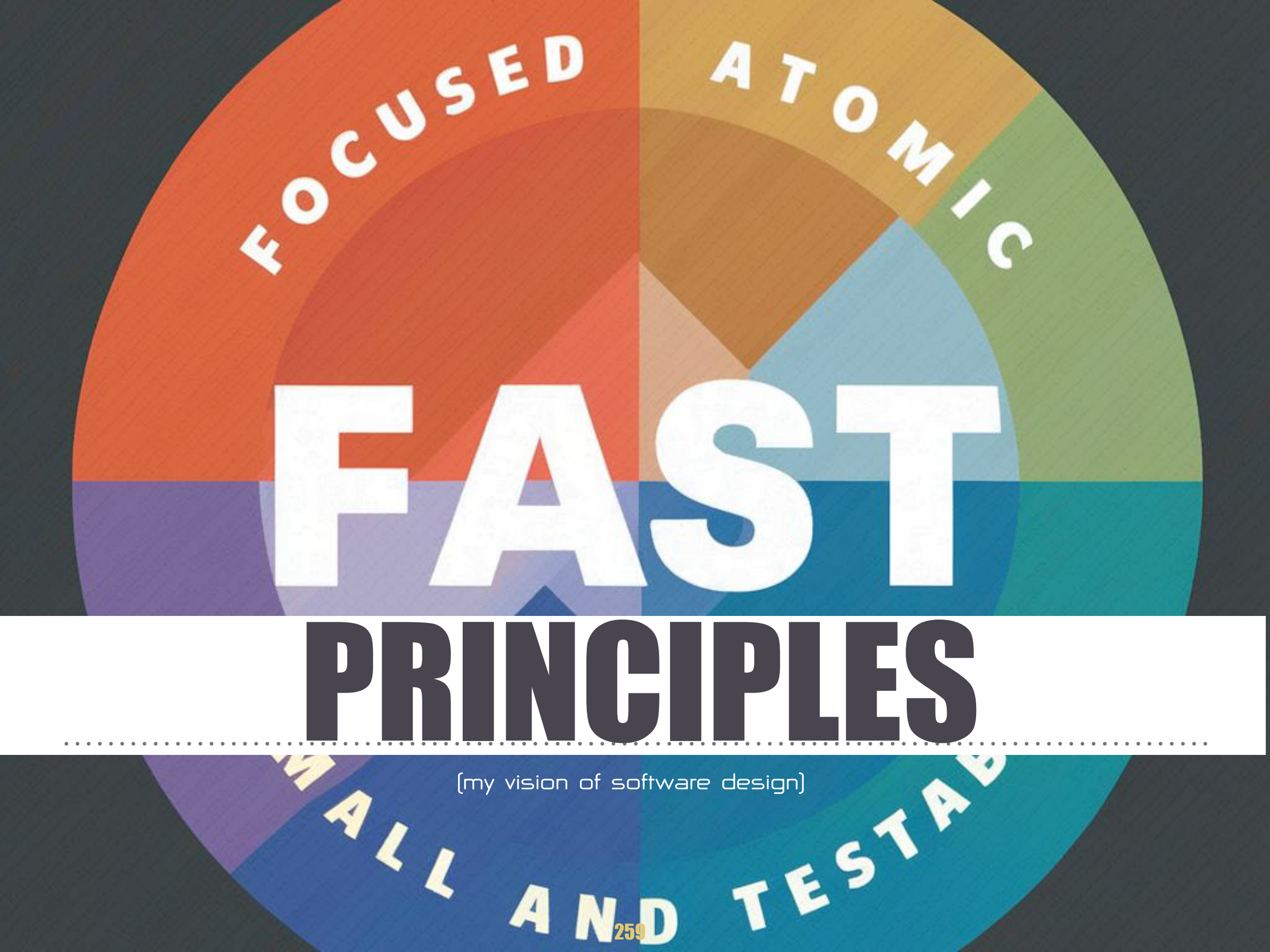
The last three principles are about the couplings between packages, and talk about metrics that evaluate the package structure of a system.

ADP
The Acyclic Dependencies Principle
The dependency graph of packages must have no cycles.
SDP
The Stable Dependencies Principle
Depend in the direction of stability.
SAP
The Stable Abstractions Principle
Abstractness increases with stability.

TODO ::

Extract function
Golden master
Approval tests

Fist Principles


FOCUSED
ATOMIC
FAST
PRINCIPLES
(my vision of software design)
ALL AND TESTAB

FAST PRINCIPLES

We believe in a set of distilled core rules of that helps us writing and keeping better code that matters and ages well.

We believe in these concepts as core values that drive our decisions in design and implementation.

We believe in values of SOLID, FIRST, KISS, YAGNI but feel the need for a set of more common principles that tie them all together

FAST is an acronym that stands for:

- Focused
- Atomic
- Small
- Testable

FAST PRINCIPLES

Focus

The Focus principle means that a code should always have a purpose, a use case.

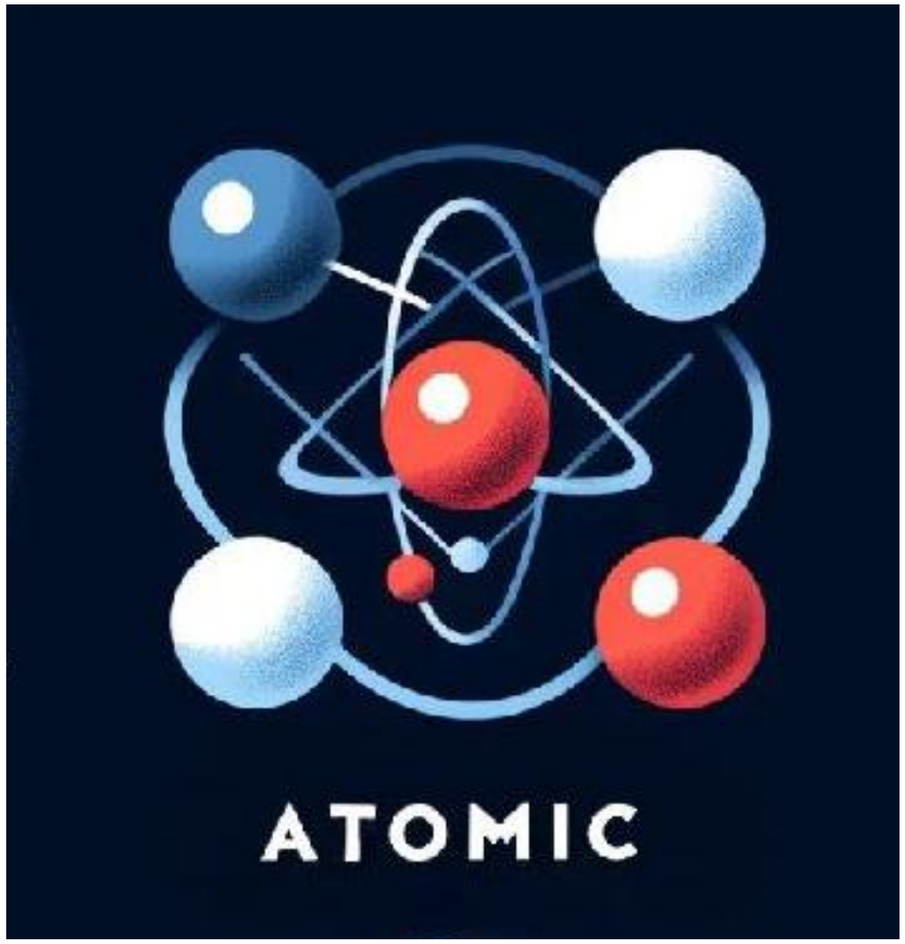
We don't write generic or technical code without a good business reason or a reason that emerge from refactoring. We don't design for reuse, reuse is a consequence of good design. The intent should be clear and simple as possible.

FAST PRINCIPLES

Atomic (& Isolated)

Elements of software should always be, at all levels, isolated from other parts of the systems and cohesive in the internals as much as possible.

This is true from code level to package level. We favor duplication of technical code over creating an unjustified dependency. We don't want future changes to harm the existing code. At least, we want to minimize it and make it easy through continuous refactoring. Let parts know about each others as less as possible.



FAST PRINCIPLES

Small (& Simple)

Complexity is everywhere, keeping things small is the only way to keep a sane projection of reality that you can explain to everyone.

Composition of small parts is always simpler to manage than a bigger one that hides some concepts. Less code you have, less bugs you'll generate. We'll always try to follow simplicity over easiness through the creation of small parts that contains the essence of a concept, a use case. Try to make things fit on our screen.

FAST PRINCIPLES

Tested

Elements of software should always be independently testable in the easiest and fastest possible way.

Keeping things testable is a guarantee for confident evolution. It means there is no hidden part having external effects. It is a glue for other values that helps proving, at least, some quality. It is a design concept; that brings, at least, the proof of absence of known defects