Software Architecture

by Victor Rentea
www.VictorRentea

Design Boundaries

- **Complex Logic** <> **Data Structure**
- **Contract** <> **Implementation**
- **Agnostic Domain Logic** <> **Infrastructure** ± Persistence
- **Controller** <> **Service** <> **Persistence** = Layers
- **High-Level** Facade <> **Low-Level** Details = Sep by Layers of Abstraction
- **Hexagonal/Clean Architecture** vs overengineering
- **Use-Cases** <> = VSA aka "anti-layers architecture"
- **User Actions** <> **Generic Update aka** "large PUT"
- **Command** <> **Query** = CQRS
- **Bounded Contexts** (Modular Monolith)
- **Microservices** for NFRs: availability, scalability, deployability..

3 Laws of Architecture

1. Everything is a Tradeoff
2. "Why?" matters more than "How?"
3. Evolution is Inevitable. Measure it!

Debate:

"Two projects should share the same SQL DB instance."

Pros

Cons

Built by 1 or 2 teams?

Debate:

Separate Schemas?

"Two projects should share the same SQL DB instance."

How complex?

High Load?

Volumetry?

Both write 😱?

Pros

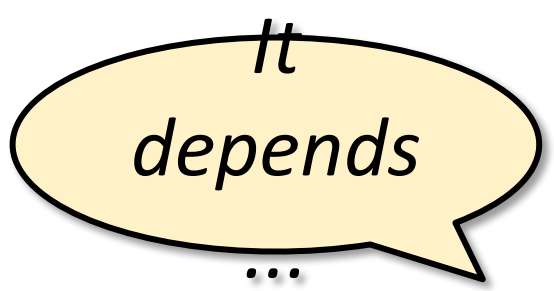
- 🏆 **Single source of truth**
- 😍 **Strong Consistency** (Tx, FK)
- ⏩ **Fast & Simple** (no API calls)
- 💰 **Cheaper** (initially)
- 😊 **Simpler ops** (1 server)

Cons

- **Data Semantic Corruption** 💩
- **Difficult to ALTER tables** ❄️
- **DB race and/or deadlocks** 🔒
- **DB Contention** (bottleneck) 🐌
- **Single Point of Failure** 💥

Everything is a Trade-off

#1



- **Every decision has a con!**
- **Watch out for decisional blind-spots (mental biases) like:**
  - 🤩 **Shinny Object** (*I wanna use cool tech*): Microservices, CQRS,...
  - 👴 **Golden Hammer** (*I'll do it like I've always done it*): PL/SQL, jQuery..
  - 🤕 **Post-Traumatic Stress** (*I'll never use again...*): Monolith, JPA/ORM
  - 🤑 **Sales Pitches** (*Our stack/product is the best!*): BPM, Axon, AWS☁️, ...

= Systematic cognitive errors

Mental Biases

- **Buzzword**: "Everyone is going serverless – so should we"
- **Survivorship Bias**: Netflix uses microservices (ignoring invisible failures)
- **Confirmation Bias**: we over-value evidence supporting our existing beliefs
- **Anchoring Bias**: we over-value the first option we hear (-20% off, 50K€)
- **Golden Hammer**: we want to keep using the familiar tool: $, PLSQL, Redis..
- **Sunk-Cost Effect**: can't abandon, long after it's obvious we should
- **Not Invented Here**: let's build an in-house framework instead of using OSS
- **Planning Fallacy**: we always under-estimate effort (= "optimism")
- **Dunning-Kruger**: amateurs are confident due to their unknown unknowns
- **Authority Bias** of seniors even outside their expertise; think alone first 🔗
- **Post-Traumatic System Design**: instinctual, fear-driven decisions

Project Phoenix — a fictional story

Acme Corp has been running its payments service on a custom in-house framework called *Helix*, built 4 years ago by the CTO, Dana. Helix has worked well, but the team has grown and onboarding new engineers takes 3 months just to understand it.

A senior engineer, Priya, proposes migrating to a well-supported open-source framework. She runs a spike and shows it would cut onboarding time in half and reduce maintenance burden significantly.

In the review meeting:

- **Dana** says: *"Helix was built for our exact needs. I've seen generic frameworks fail at scale — remember the 2021 outage we had with that third-party library? We can't trust external dependencies."*
- **Marcus** (team lead) agrees quickly: *"Yeah, Dana knows this system better than anyone. I think we should trust her judgment on this."*
- **Priya** pushes back with benchmarks, but Marcus cuts the discussion short: *"We've already spent two years optimising Helix. Throwing that away feels wrong. Let's not be rash."*
- A junior engineer, Tom, has concerns about Helix's lack of community support but stays quiet — the last person who challenged Dana's architectural decisions was moved to a different team.
- The team decides to do a *"light modernisation"* of Helix instead. The scope is defined by Dana over a weekend and presented as a done deal on Monday.

-**Buzzword**: Everyone is ..
-**Survivorship Bias**: Netflix uses
-**Confirmation Bias**: ... existing beliefs
- **Anchoring Bias**: the first option...
- **Golden Hammer**: use the familiar ...
- **Sunk-Cost Effect**: can't let go,...
- **Not Invented Here**: in-house ...
- **Planning Fallacy**: under-estimate...
- **Dunning-Kruger**: ..amateurs
- **Authority:** ...outside their expertise
- **Post-Traumatic**: fear-driven

1. What bias you spot?
- Impact on individual, team, process?

2. Single most damaging?

3. Rewrite one moment — how should it have gone instead?

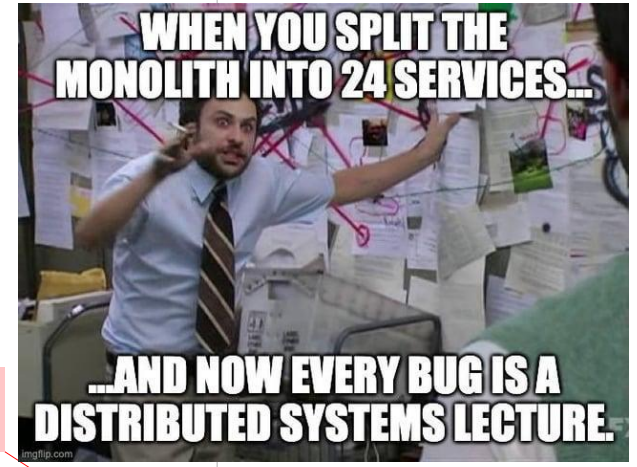
Hype Cycle

[2015 .. 2018] peak hype for microservices

VISIBILITY

- UBER-style

- KRAZAM🎞

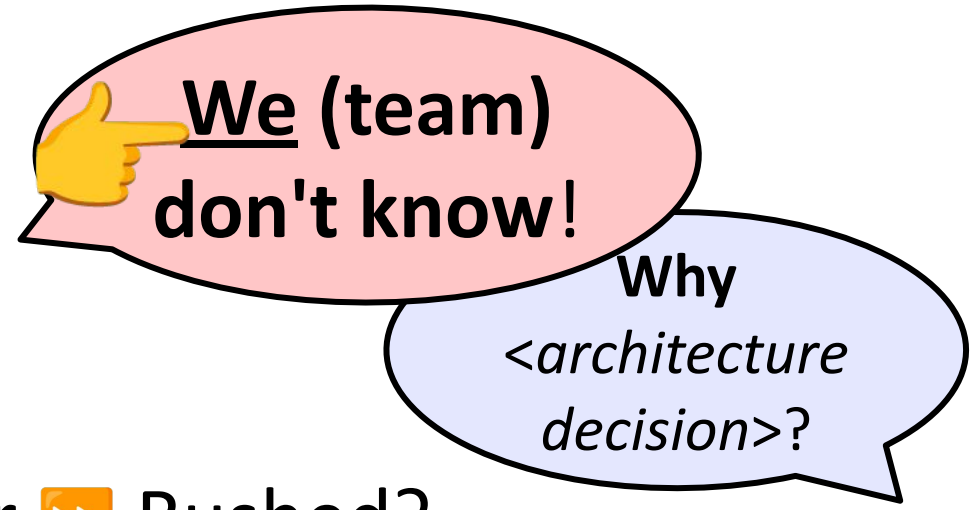


https://www.gartner.com/en/research/methodologies/gartner-hype-cycle

Brainstorming Best Practices

- Make it a **safe** meeting
- Invite skeptics, enthusiasts, pragmatics, dreamers..
- Consider **all** (absurd) **ideas** (+AI)
- Discover all pros/cons (+AI)
- Evaluate in your **Context** 👑

Reasons for



- That's how we got the project 🎁
  - Knowledge Transfer was: 😡 Denied or ⏩ Rushed?
- That's how the ninja developers 🥷 built it
- We copy-pasted from another microservice 🙈
- Following corporate rules - *it's always been done like that*
- They rushed us max ⏩ - we never actually decided ...
- We forgot 😔!

#2 "Why" is More Important than "How"

Today: **What are we missing?** 😬

1. Why did *they* select **Option A**?
2. Why not **Option B**?

⬇️

Write **Architectural Decisions Records** (**ADR**)

Architectural Decision Record (ADR)

Contents:

- **Status:** draft (in review) ⇒ accepted ⇒ superseded (by a later ADR)
- **Context:** expertise, hype, culture, politics... (might change later)
- **Clear Decision** / action point
- **Why**: Pros + Cons, for each **Option Considered**
- **Enforcement**, if it impacts daily practice: code reviews, or tools❤️

⭐All diagrams in ASCII (plantuml/mermaid): AI-ready, pt git diff, nu necesita tool, scapi de OCD de dev

https://github.com/joelparkerhenderson/architecture-decision-record

ADR Scope

- **Strategic ADR**: high-level > stored in Confluence
  - **Risk:** prohibitively long, out of sync, unclear goal
  - **Risk**: lack of marketing => collaborate, mob-read, feed to AI RAG
- **Tactical ADR**: impacting one Git repo > stored in that repo ⭐
  - Developers - *Own your architecture*! ✊
  - Articulating a decision clarifies it for you!
  - Context for AI agent to review/generate code tomorrow
  - Can reference Strategic ADR

https://github.com/joelparkerhenderson/architecture-decision-record

Architecture Decision Records – Use-cases

0) **Pre-Meeting Draft**
   - Sent to all participants before a meeting

1) **New decision** ⭐
   - Written by someone else than the decision promoter= feedback loop ⟲

2) **Archeology**
   - To answer the questions asked by the new colleague.

3) **Migration Guide**
   - Explain refactoring goals & techniques

Tactical ADR on Git: Examples

- **Tradeoffs**: We extract DTOs directly from search SELECT queries
- **Code Structure**: these modules / packages are meant to contain...
- **Role**: Classes Xyz**Access** talk to DB, and don't depend on each other
- **Boundaries**: We don't expose Domain Model as JSON in REST APIs
- **Patterns**: We apply discounts using Chain of Responsibility Pattern
- **Library Selection**: We chose jQGrid as it has server-side pagination

Tactical ADR to Empower Engineers

- Written **for developers**
  - Kept Concise – devs hate rdin lng dcments
- Written **by developers**
  - As a manifesto: What we believe in! ✊
- **Stored on Git**
  - Lower barrier for developers
  - Allows to time-travel code + decisions in Git history
- **Ideally enforced with Automated Tools**
  - Like ArchUnit for code structure

ADR Pitfalls

❌ **Lack of awareness**, if written solo

- Collaborate; keep on project's Git

❌ **Too detailed**, eg 1 ADR for every PR

- Use only for design decisions impacting work of many weeks/months

❌ **Fall out of sync with code**, just like any // comments

- Try (hard) to automate any rules enforced

❌ **Risk to be seen as "carved in stone" by developers**

- Status: Accepted, add: "(⚠️But please don't hesitate to challenge it!)"

What's Wrong with Waterfall?

Long Feedback Loop

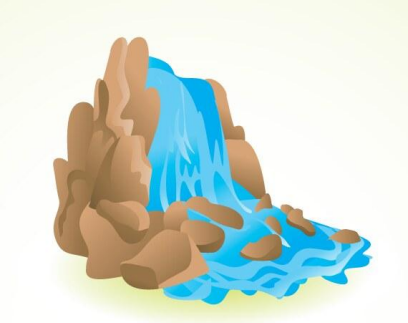
Waterfall = all changes are requested in the last week of the dev year 🤣

⇒ Emergency Patches and Dirty Quick-fixes

⇒ Next year, speculate about future changes/issues => overengineering = waste

⭐ **Agile** = shorten feedback loops and "Embrace change"

#3 Evolution is Inevitable



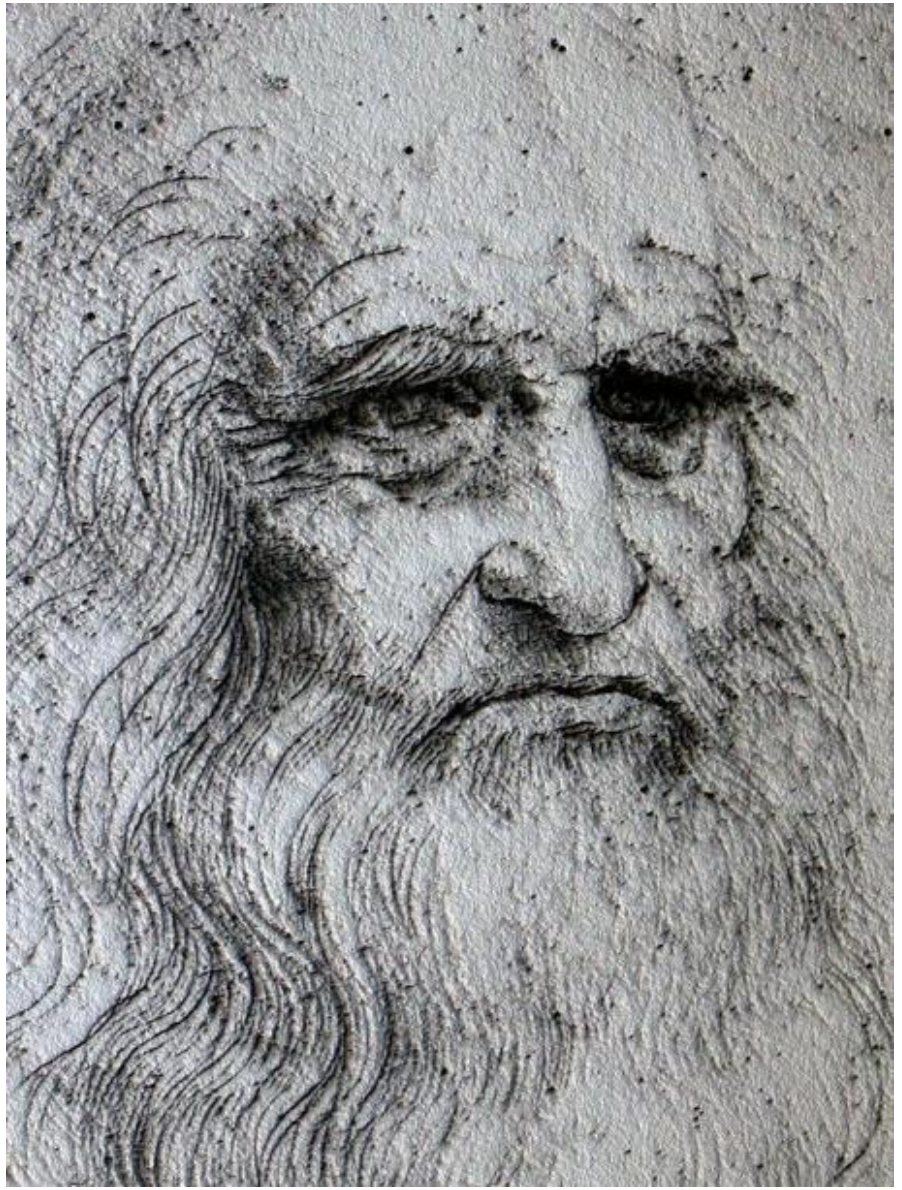
- Instead of **overengineering a futureproof** solution for 3 years
- **Guide its evolution with data**👑 = **architectural fitness functions**
  - **Clean Code**: CI Quality Gates like SonarQube...
  - **Code Structure:** build modules, ArchUnit tests, Checkstyle, Sonargraph
  - **Functional Test Coverage%**: Component > Integration > Unit (Honeycomb)
  - **Code/SBOM Vulnerabilities**: Snyk, ...
  - **Performance** using jMeter, Gatling, K6, JMH on CI🌙; profile production (JFR)
  - **Metrics + Traces + Alarms + Logs** in production; regular time tracing
  - **Resilience Tests** vs bad network (ToxyProxy) or nodes dying (ChaosMonkey)

Keep It Simple, Stupid !!

https://en.wikipedia.org/wiki/KISS_principle

Simplicity is the ultimate sophistication

-- Leonardo DaVinci



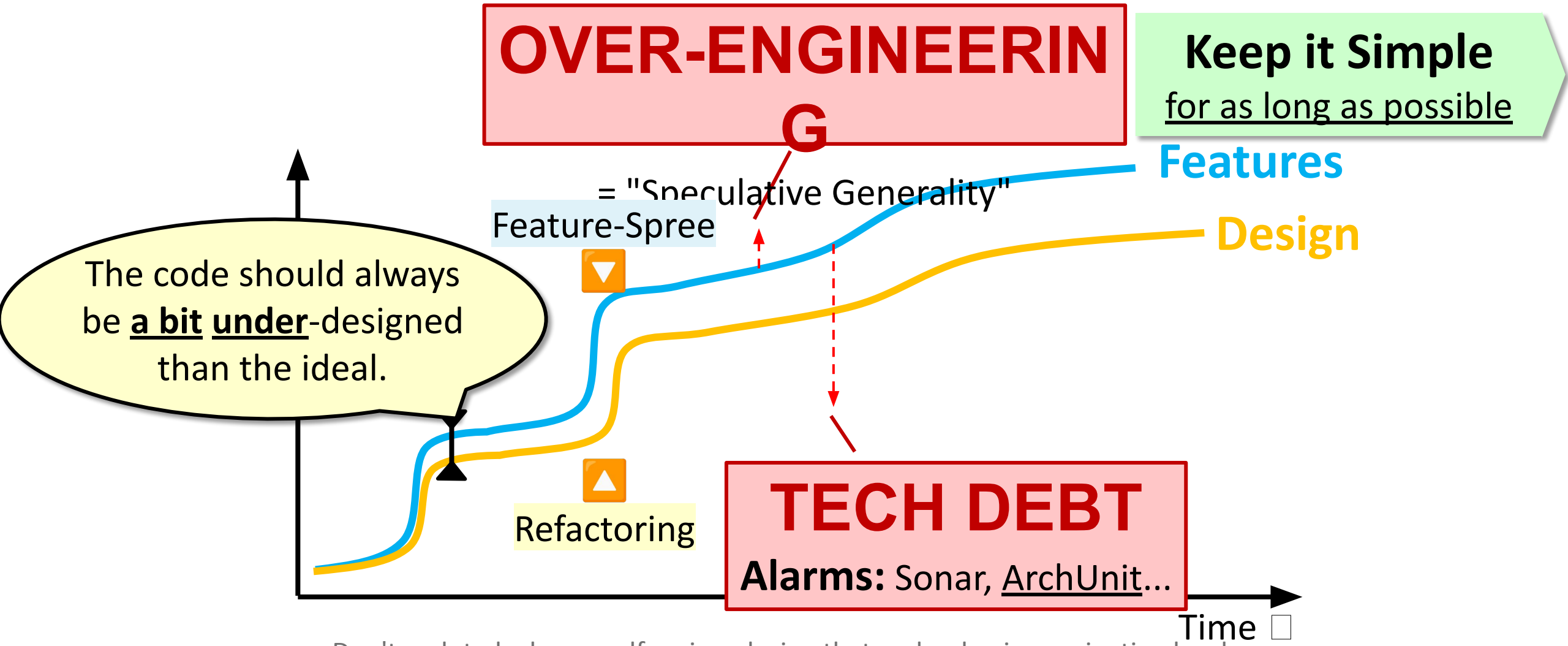
The code should always be **a bit under**-designed than the ideal.

not **heavily**

not **over**-

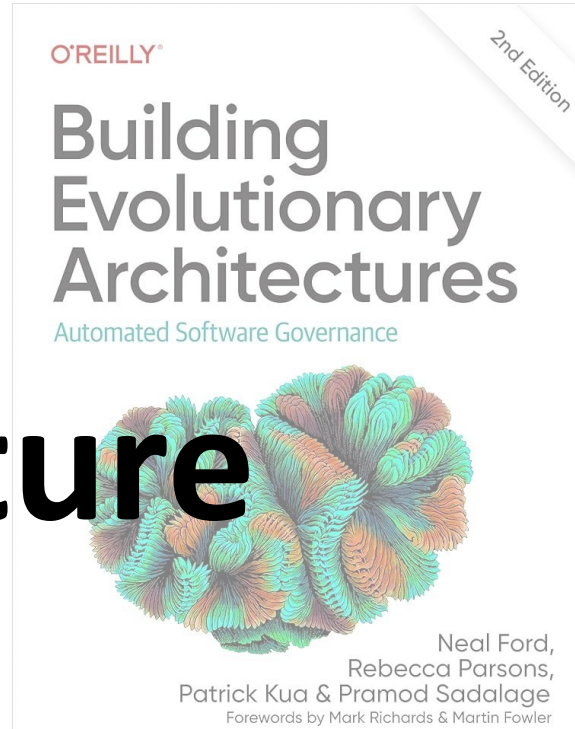
*Don't let "**perfect**" become the enemy of "**good enough**".*

Evolutionary Architecture



Don't rush to lock yourself up in a design that makes business pivoting hard.
The moment in which you know least about a problem is at the beginning.

Evolutionary Architecture



Don't religiously follow a reference' textbook architecture.

Instead, **observe the growing pain points** of the project, and select the most helpful '**design increment**' to apply.

Evolutionary Architecture

Unit-Test earlier with longer, functional tests! (see video: Test-Driven Design Insights)

Refactor continuously to KISS with Quality Gates, Code Review +2, Pairing, ArchUnit

Complex Domain / Tricky Consistency => Rich Domain Model ± DDD Aggregate

Ugly LIB or API => +Adapter ± implementing interface from Domain (Dependency Inversion)

Complex Mapping => switch to manual mapping

Cumbersome or >= 2 input channels => +Input Adapters (eg RestController)

Legacy or Shared DB => +DAO ± dep-inv

High Complexity mixed with Dependencies => Integration / Operation Segregation (IOPS)

Extract Complexity in Pure Functions (video: Functional Core / Imperative Shell)

Large Team => Split Domain for more autonomy (modular monolith)

Non-functional Requirements => Microservices

Some decisions must be taken early

Variable Blueprints ☹️

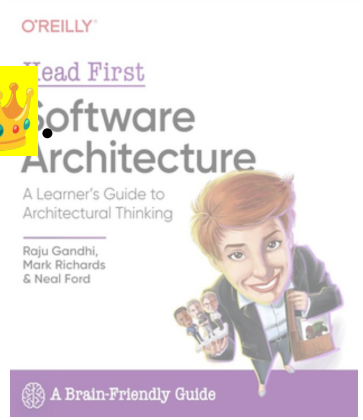
Classify projects in T-Shirt size based on macro-estimate

- ⚫ **Small** projects: can use Python, **golang**, or Java ±**jHipster** (rapid dev)
- ⚫⚫ **Medium** projects: must use **.java** + **Hexagonal**
- ⚫⚫⚫ **Large:** must use .**java** & after 2 year: Modulith preparing for micros?

Laws of Architecture

1. Everything is a Tradeoff

"It depends" - Identify Pros/Cons relevant in Your Context👑.

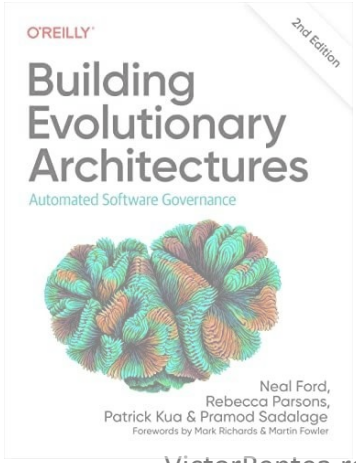


2. Why matters more than How

Document design decisions via ADR in Wiki ± Git...

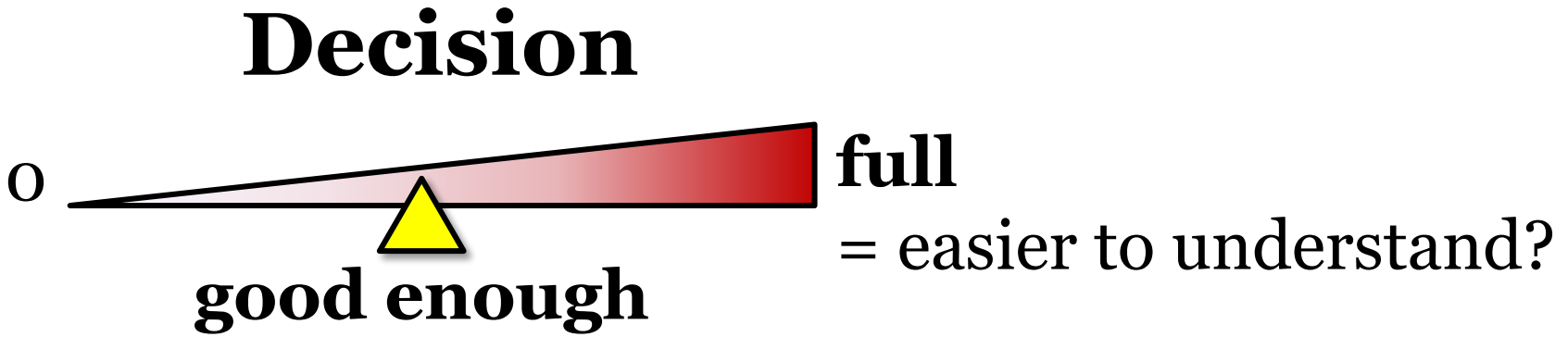
3. Evolution is inevitable

Constrain architecture with fitness functions on data



Design Gradients

In a "dogmatic" Architecture,
developers often complain of **overengineering**



Why this optimization (only) there?

The true Challenge = Finding the best path *there*...

- While shipping features/fixes as usual,

- Take small risk dose every release,

- Address highest pain-points

- Enjoy the journey⚠️.

Usually requires prior **experimentation**.

NFRs

Non-Functional Requirements

Avoid this name: might not seem important enough

Architectural Characteristics

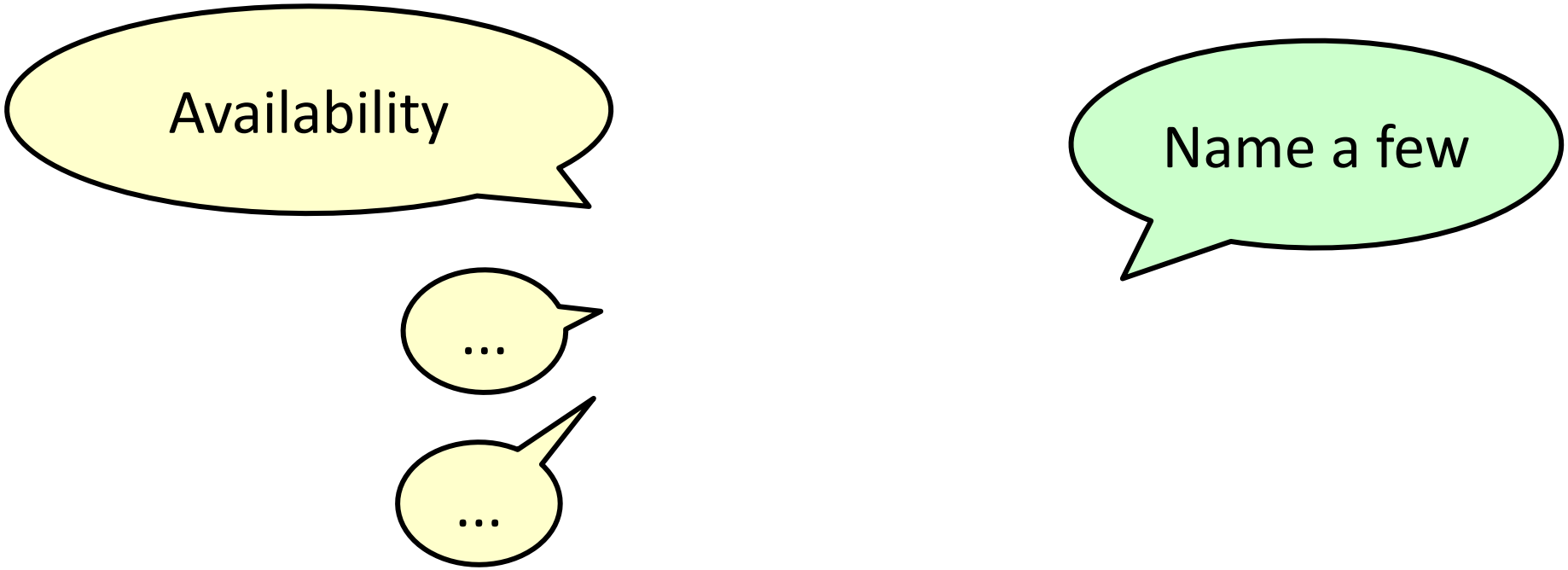
Non-**F**unctional **R**equirements

Software **Qualities**

–ilities ☺

Avoid this name: might not seem important enough

Architectural Characteristics



Avoid "NFR" term as these are often very "functionally" critical

Arhitectural Qualities

- **Availability**: %uptime, fault-tolerance
  - **Reliability**: frequency of incidents 🚄✈️
  - **Recoverability**: MTTR⏬, w/o data loss
  - **Robustness** to bad input/network failures
- **Scalability:** load⏫=>SLA=✅, co$t=liniar↗️
  - **Latency**: response time =50ms 99%th
  - **Throughput**: # messages, calls, lines / Δt
  - **Elasticity:** spinoff-time on load spikes
- **Data Consistency** across APIs
- **Configurability** w/o code changes:feat-flag
- **Usability:** easy to learn/retrain users
- **Accessibility** to color-blind♿, shortcuts

- **Maintainability**: easy to bugfix / evolve
  - **Extensibility** with new features/clients
  - **Testability** (automated)
  - **Observability**💖**:** logs, metrics, traces
- **Feasibility** within cost💰/time📆/scope🥧
- **Portability** Web/iOS, ORA/PG, gcp/aws/azu
- **Interoperability** with SAP,SF,BPM,COBOL...
- **Security**: authentication, confidentiality, integrity, non-repudiation, privacy
- **Compliance**: ISO, GDPR, SOX...
- **Auditability**: legal tracing of actions👨‍⚖️

Exercise: Which two can **come in conflict**?
Exercise: arch design **decision hijacking** each?

Moral?

There's so much more to software than

It works on my machine™

Architecture Characteristics Worksheet

based on sheet from https://www.developertoarchitect.com/resources.html

System/Project: _____ _____

Architect/Team: _____ Date: _____ Next Review: _____

Top 3	Driving Characteristics
☐	
☐	
☐	
☐	
☐	
☐	
☐	

Implicit Characteristics
(promote to driving if x-critical)

- feasibility (cost/time)
- security
- maintainability
- observability

Others Considered

Instructions

- Identify no more than 7 driving characteristics.
- Pick the top 3 characteristics (in any order).
- Implicit characteristics can become driving characteristics if they are *critical* concerns.
- Add additional characteristics identified that weren't deemed as important as the list of 7 to the *Others Considered* list.
- Definitions are on the following pages

Common Architecture Characteristics

performance	data integrity	deployability
responsiveness	data consistency	testability
availability	adaptability	abstraction
fault tolerance	extensibility	workflow
scalability	interoperability	configurability
elasticity	concurrency	recoverability
usability	compliance	portability

<<other, not specified here?>>

a
b
denotes characteristics that are related; some systems only need one of these, other systems may need both

Architecture Characteristics Worksheet

performance

The amount of time it takes for the system to process a business request

responsiveness

The amount of time it takes to get a response to the user

availability

The amount of uptime of a system; usually measured in 9's (e.g., 99.9%)

fault tolerance

When fatal errors occur, other parts of the system continue to function

scalability

A function of system capacity and growth over time; as the number of users or requests increase in the system, responsiveness, performance, and error rates remain constant

elasticity

The system is able to expand and respond quickly to unexpected or anticipated extreme loads (e.g., going from 20 to 250,000 users instantly)

data integrity

The data across the system is correct and there is no data loss in the system

data consistency

The data across the system is in sync and consistent across databases and tables

adaptability

The ease in which a system can adapt to changes in environment and functionality

concurrency

The ability of the system to process simultaneous requests, in most cases in the same order in which they were received; implied when scalability and elasticity are supported

interoperability

The ability of the system to interface and interact with other systems to complete a business request

extensibility

The ease in which a system can be extended with additional features and functionality

deployability

The amount of ceremony involved with releasing the software, the frequency in which releases occur, and the overall risk of deployment

testability

The ease of and completeness of testing

abstraction

The level at which parts of the system are isolated from other parts of the system (both internal and external system interactions)

workflow

The ability of the system to manage complex workflows that require multiple parts (services) of the system to complete a business request

configurability

The ability of the system to support multiple configurations, as well as support custom on-demand configurations and configuration updates

recoverability

The ability of the system to start where it left off in the event of a system crash

feasibility (implicit)

Taking into account timeframes, budgets, and developer skills when making architectural choices; tight timeframes and budgets make this a driving architectural characteristic

security (implicit)

The ability of the system to restrict access to sensitive information or functionality

maintainability (implicit)

The level of effort required to locate and apply changes to the system

observability (implicit)

The ability of a system or a service to make available and stream metrics such as overall health, uptime, response times, performance, etc.

Architectural Kata: **Silicon Sandwiches**

Description

- A national sandwich shop wants to enable online ordering (in addition to its current call-in service).

Users

- Thousands, perhaps one day millions

Requirements

- Users will place their order, then be given a time to pick up their sandwich and directions to the shop (which must integrate with several external mapping services that include traffic information)
- If the shop offers a delivery service, dispatch the driver with the sandwich to the user
- Mobile-device accessibility
- Offer national daily promotions/specials
- Offer local daily promotions/specials
- Accept payment online, in person, or upon delivery

Additional context

- Sandwich shops are franchised, each with a different owner
- Parent company has near-future plans to expand overseas
- Corporate goal is to hire inexpensive labor to maximize profit

See other Architectural Katas >>

Example Performance Goals

- **Availability:** 99.999% (< 5 minutes downtime/year)
- **Response Time:** of GET /x: 99%th=100 ms under 10K requests/s
- **Throughput**: can ingest 1000 messages/second
- **Stability**: can ingest a spike of 1K requests over 10 milliseconds
- **Data Volumes:** max 1M active orders; +100K new orders/day

Label	# Samples	Average	Median	90% Line	95% Line	99% Line	Min	Maximum	Error %	Throughput
HTTP Request	1260913...	114	104	126	144	268	0	44957	0.11%	4958.8/sec

The Art of Diagrams

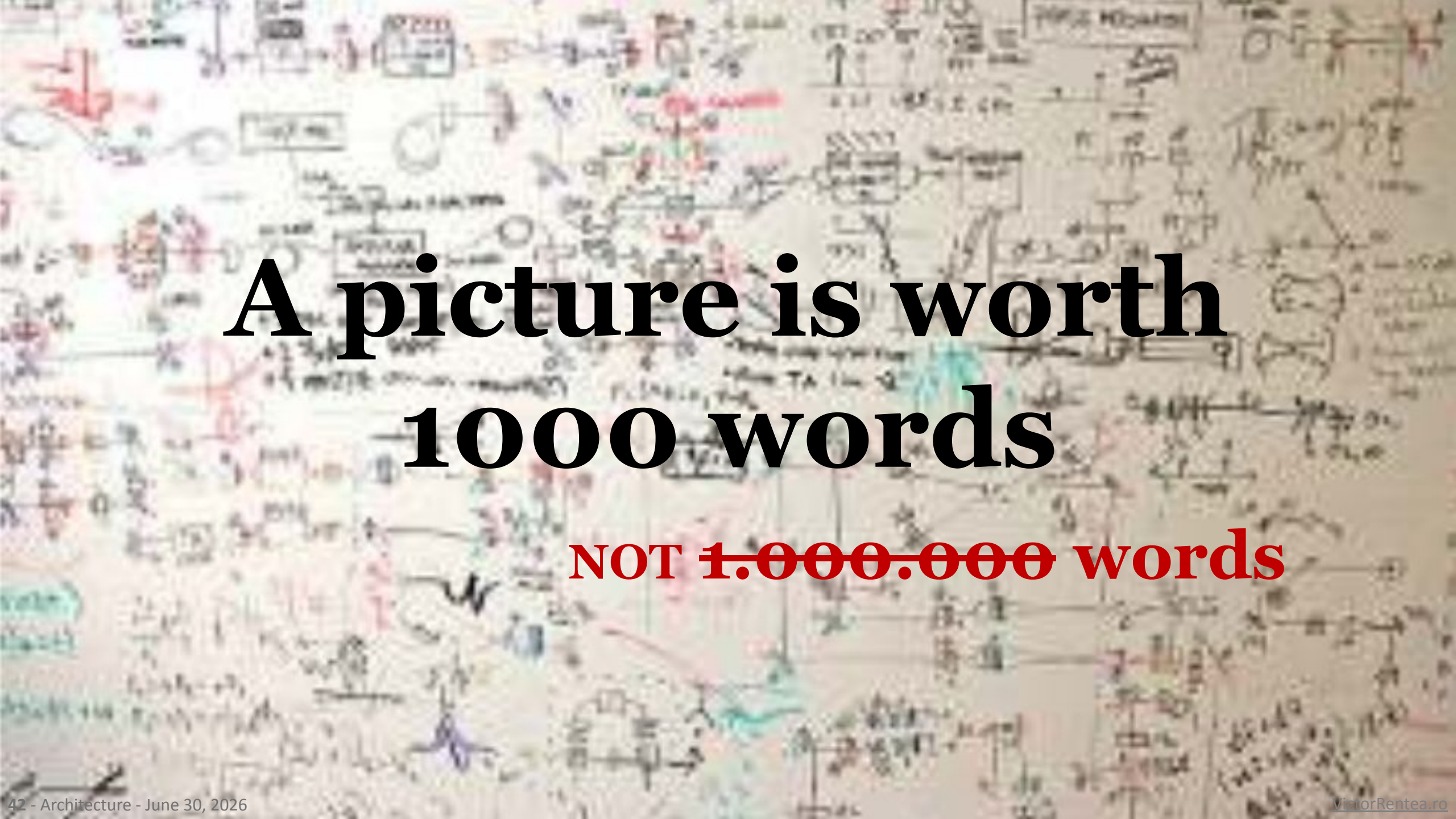
https://nealford.com/downloads/Documenting_and_Presenting_Software_Architecture_by_Neal_Ford.pdf

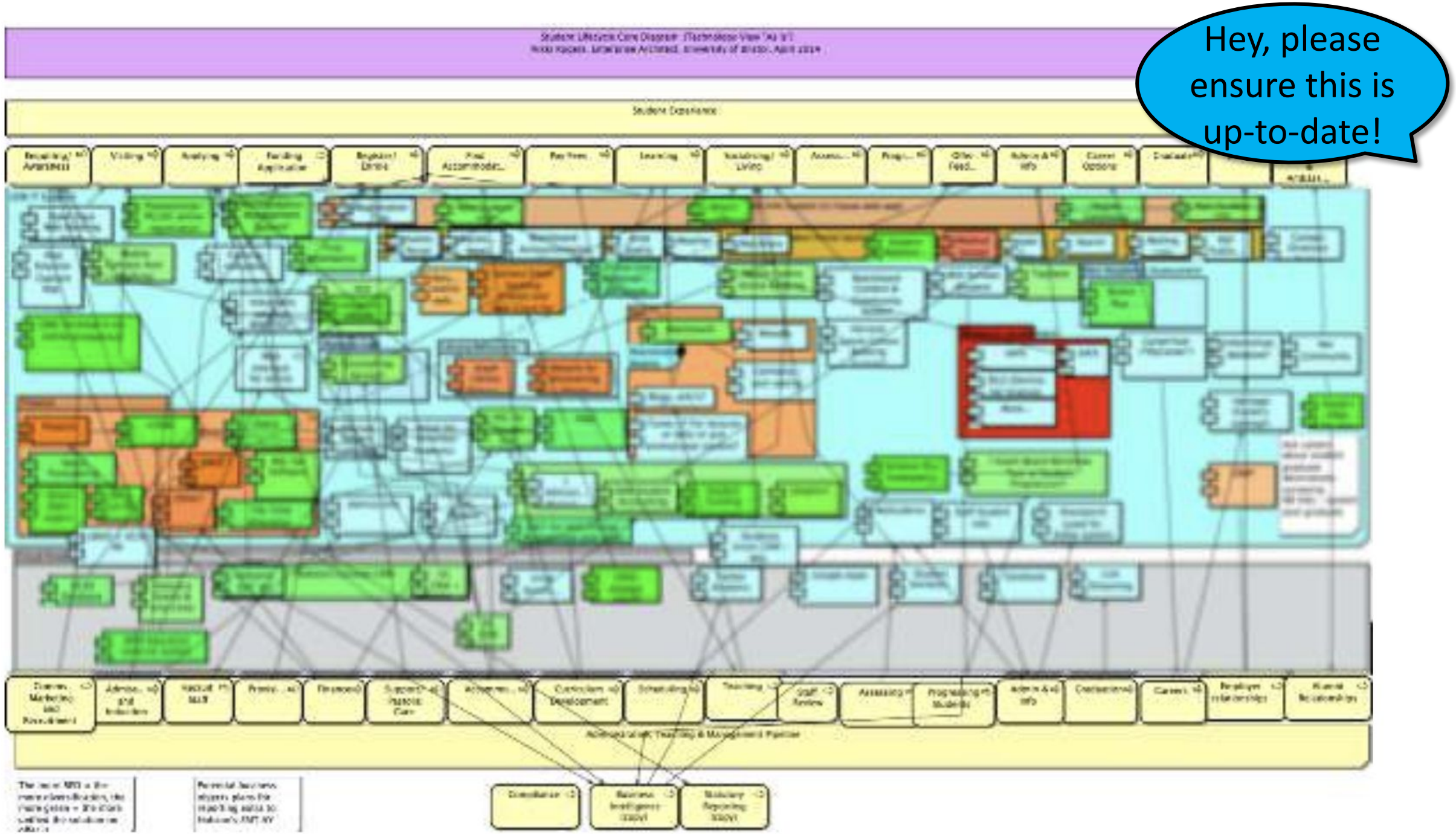
Mark Richards - https://www.youtube.com/watch?v=m2FlZNRELYA

https://www.infoq.com/articles/crafting-architectural-diagrams/

What is the top one most important advice for desinigng a diagram?

- Versioning
- Less is more, Keep it simple
- Keep it simple and expect your diagram to be out of date fast
- ask review kind and constructive > easier to review than a long arch decision record
- keep the same level of abstraction; try C4
- Construct a clean and clear story on it when present it; make a short video of you explaining it. your children will ai-trascribe synthetize yoru video
- if a diagram requires you to explain it, it's not good.
- simple, demo youself; present it in fron of a group
- tailor for audience (security, audit, developers, ops)
- colours are another dimension
- Review / create as DoD on Jira tasks to ensure that you don’t forget to do it: tech spec
- Ask ai to do comic story with actors and dialogs between services = Domain Storytelling
- NO don't mix control and data flow, make 2 diagrams instead
- use labels to clarify data flow assuming arrow mean control flow
- comic is like CRC
- At the end of the day most tools work well, getting people to maintain the diagrams is often a bigger hurdle than the used tool.
- A single goal for the diagram, don't try to to include things that are orthogonal/unrelated


A picture is worth
1000 words
NOT ~~1.000.000~~ words


Hey, please ensure this is up-to-date!

"I spent 3 hours on that diagram and I'm NOT changing it!"

Irrational Artifact Attachment

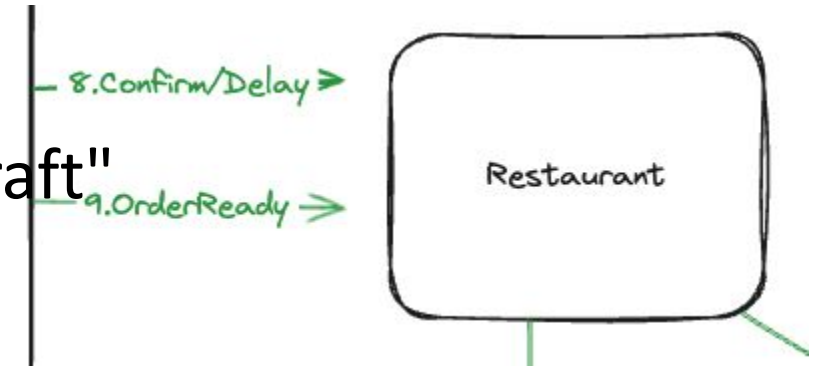
is proportional to how long it took you to produce it.

Don't overly polish your diagrams.

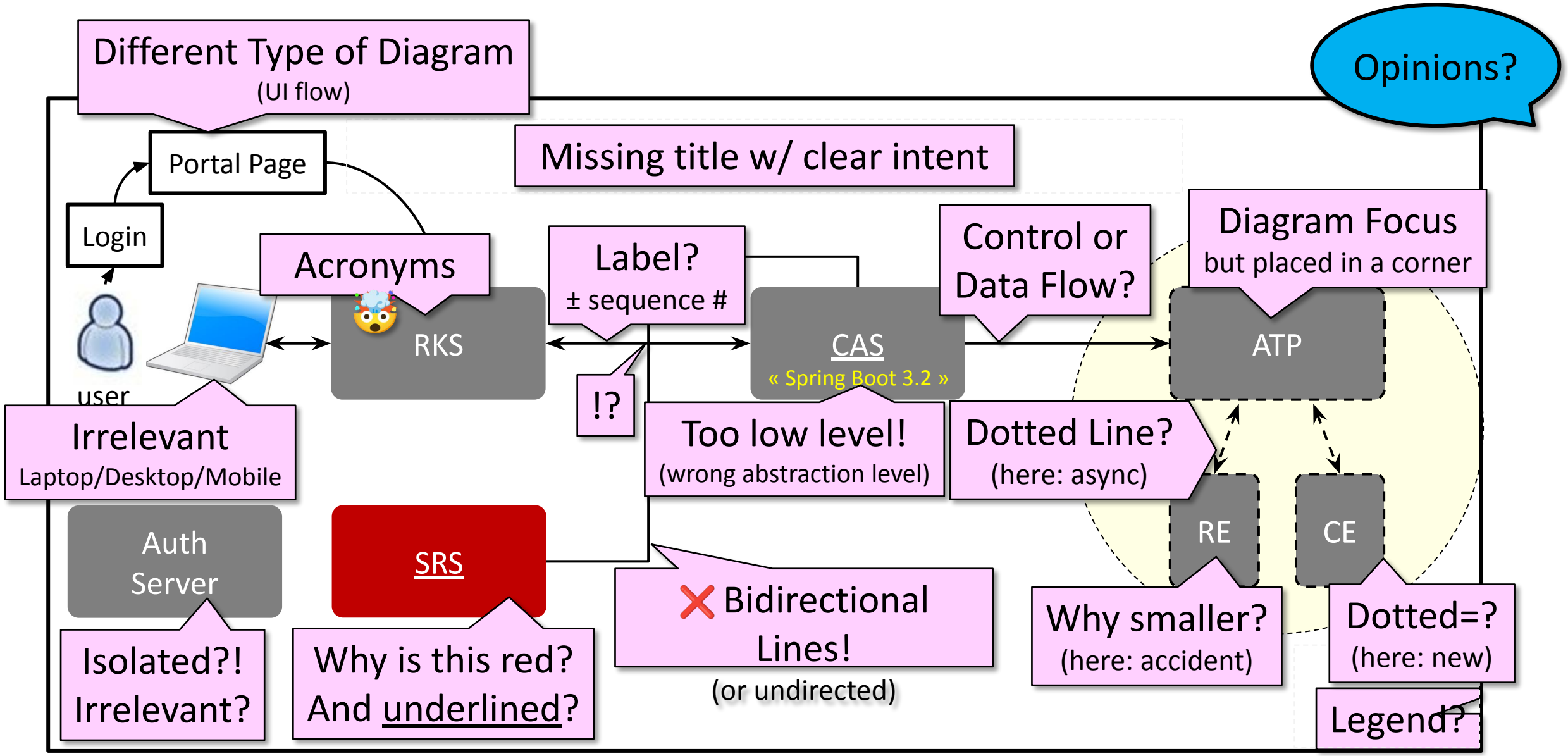
Prohibitively hard to change

=> Out of sync faster.

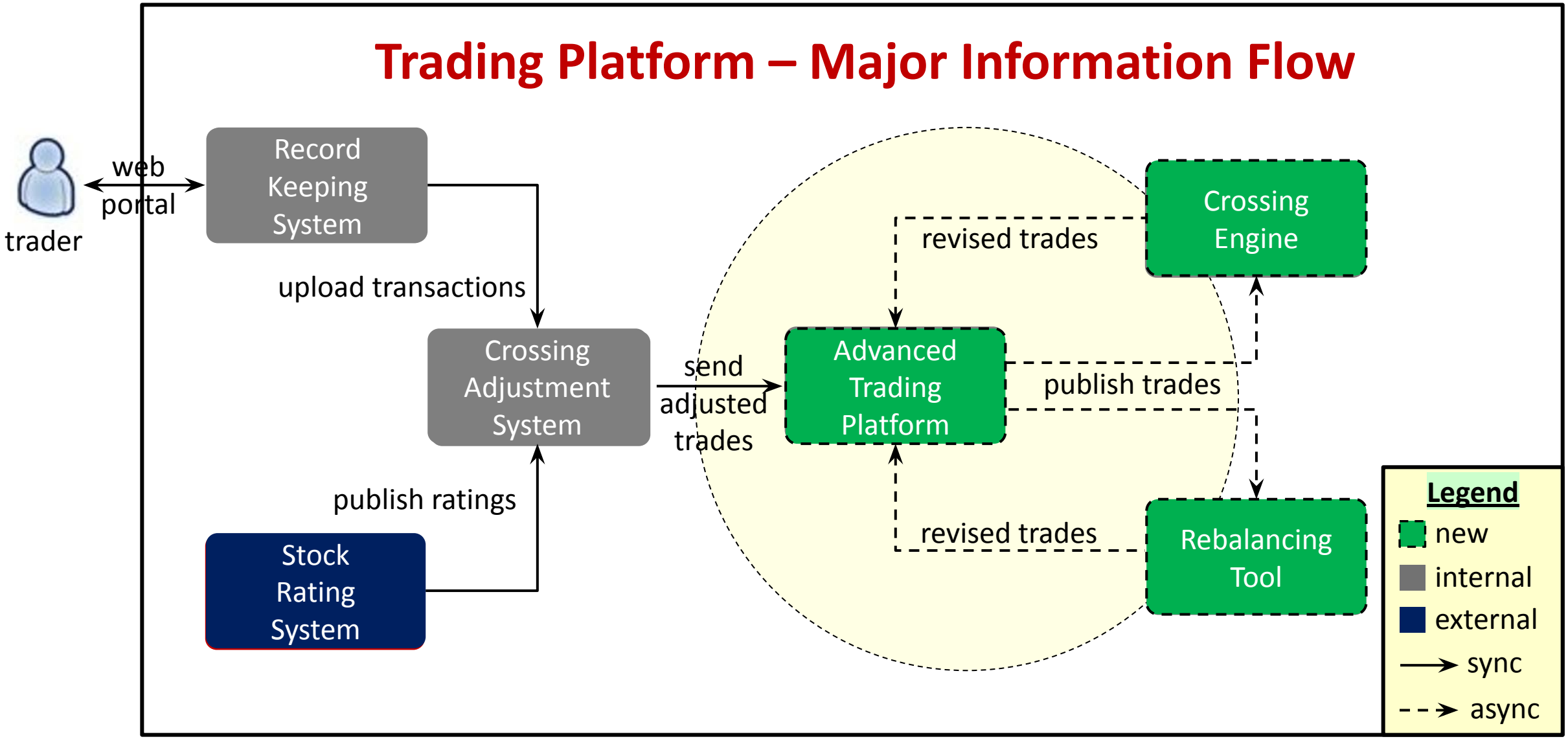
- Keep diagrams small
- Use sketchy lines = "it's a ever-draft"



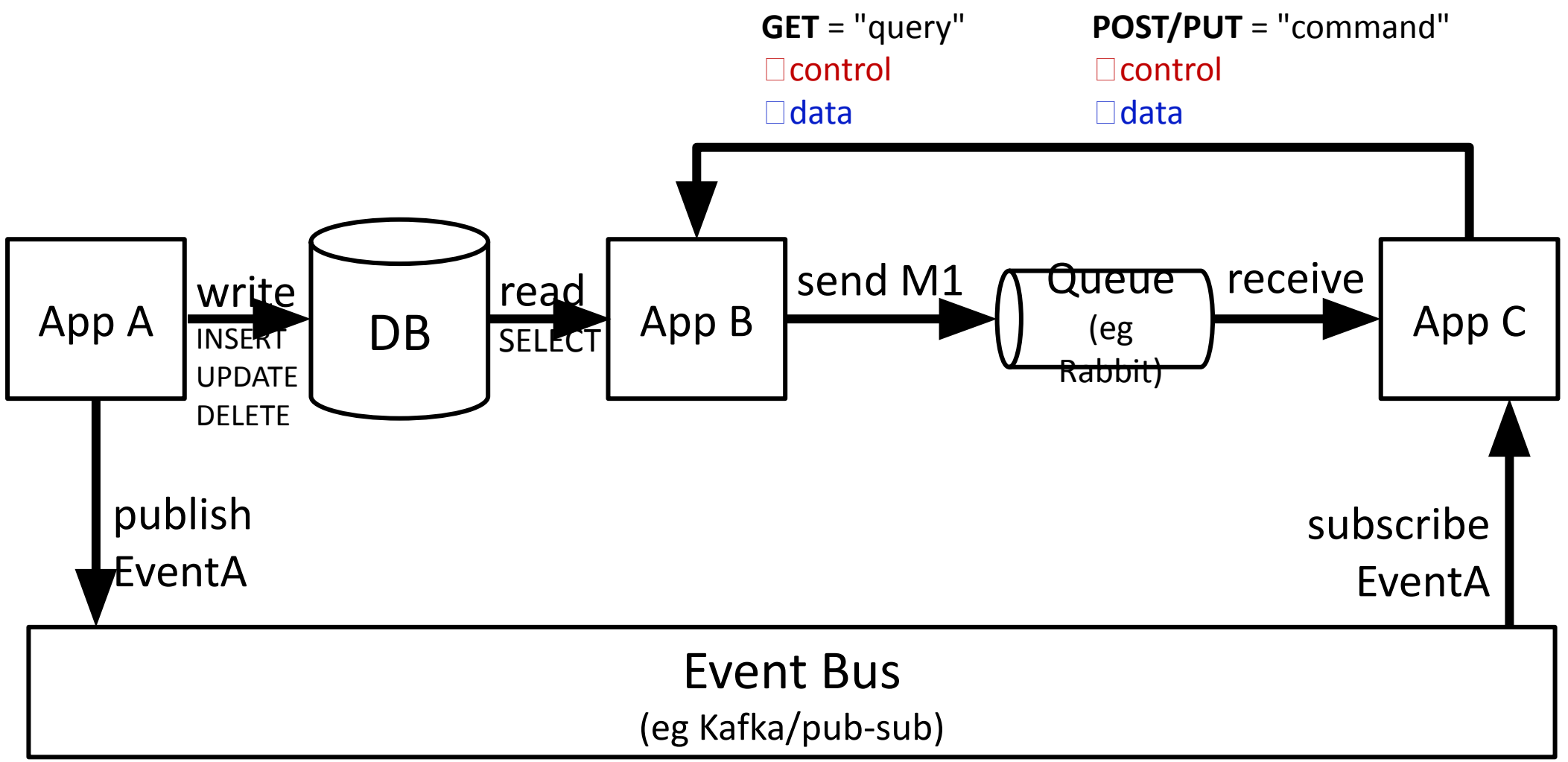
https://nealford.com/memeagora/2008/12/01/irrational_artifact_attachment.html


Different Type of Diagram
(UI flow)
Opinions?
Missing title w/ clear intent
Portal Page
Login
Acronyms
Label?
± sequence #
Control or
Data Flow?
Diagram Focus
but placed in a corner
RKS
CAS
« Spring Boot 3.2 »
ATP
user
!?
Too low level!
(wrong abstraction level)
Dotted Line?
(here: async)
Irrelevant
Laptop/Desktop/Mobile
RE
CE
Auth
Server
SRS
❌ Bidirectional
Lines!
(or undirected)
Why smaller?
(here: accident)
Dotted=?
(here: new)
Isolated?!
Irrelevant?
Why is this red?
And underlined?
Legend?

Trading Platform – Major Information Flow



Shapes and Arrows



Diagrams – Key Points

- Mind its audience: Highlight a single story / delta
- Clear **Title**
- **Legend**
- **Boxes**: uniform style, not isolated
- **Lines**: unidirectional labeled arrows explaining data flow, less 
- **Single level of detail:** at one of the C4 levels
- **Keep it small**: too large => prohibitive to change => out of sync

https://www.infoq.com/articles/crafting-architectural-diagrams/

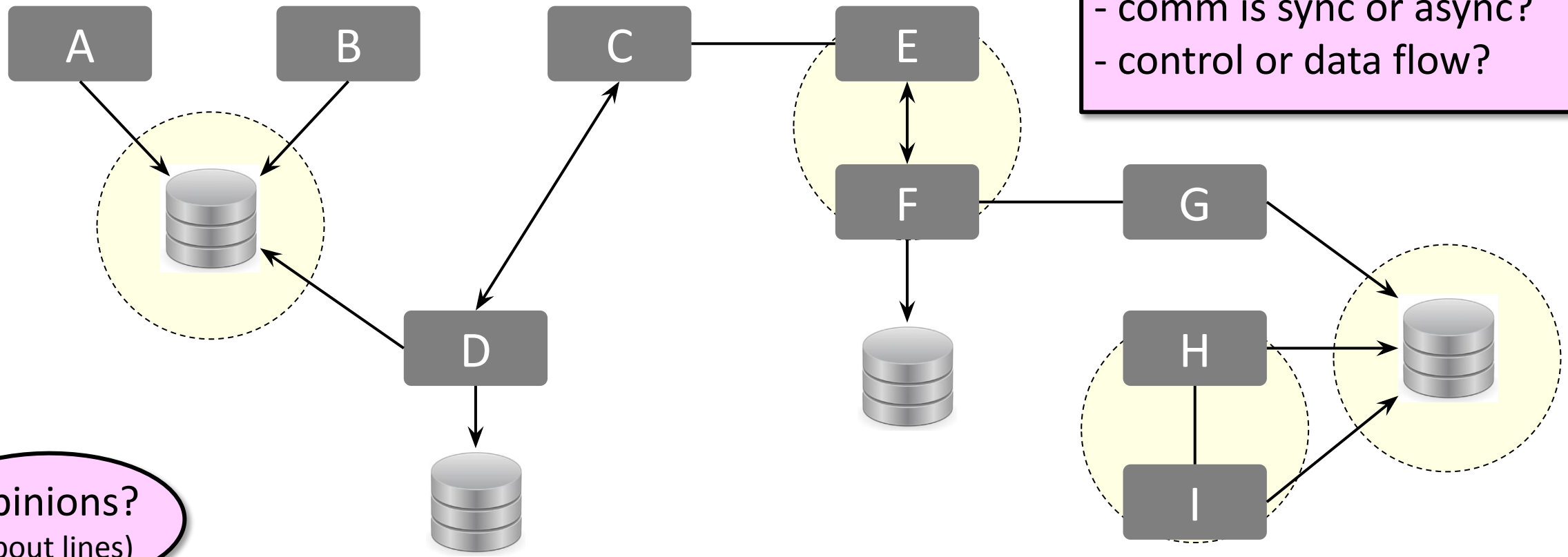
Top Most Important Advice for a Diagram

- Version it
- Less is more, Keep it simple
- Expect your diagram to be out of date fast
- Ask review - much easier than for a long document
- Keep the same level of abstraction; try C4
- Construct a clean and clear story on it then present it; make a short video ~TikTok of you explaining it. your children will ai-transcribe synthetize your video
- Present it in front of a group
- If a diagram requires you to explain it, it's not good.
- Tailor for audience - security, audit, developers, ops
- Colour are another dimension
- Review / create as DoD on Jira tasks to ensure that you don’t forget to do it => tech spec
- Don't mix control and data flow, make 2 diagrams instead±
- Use labels to clarify data flow assuming arrow mean control flow
- At the end of the day most tools work well, getting people to maintain the diagrams is the challenge
- A single goal for the diagram, don't try to to include things that are orthogonal/unrelated

Lines?

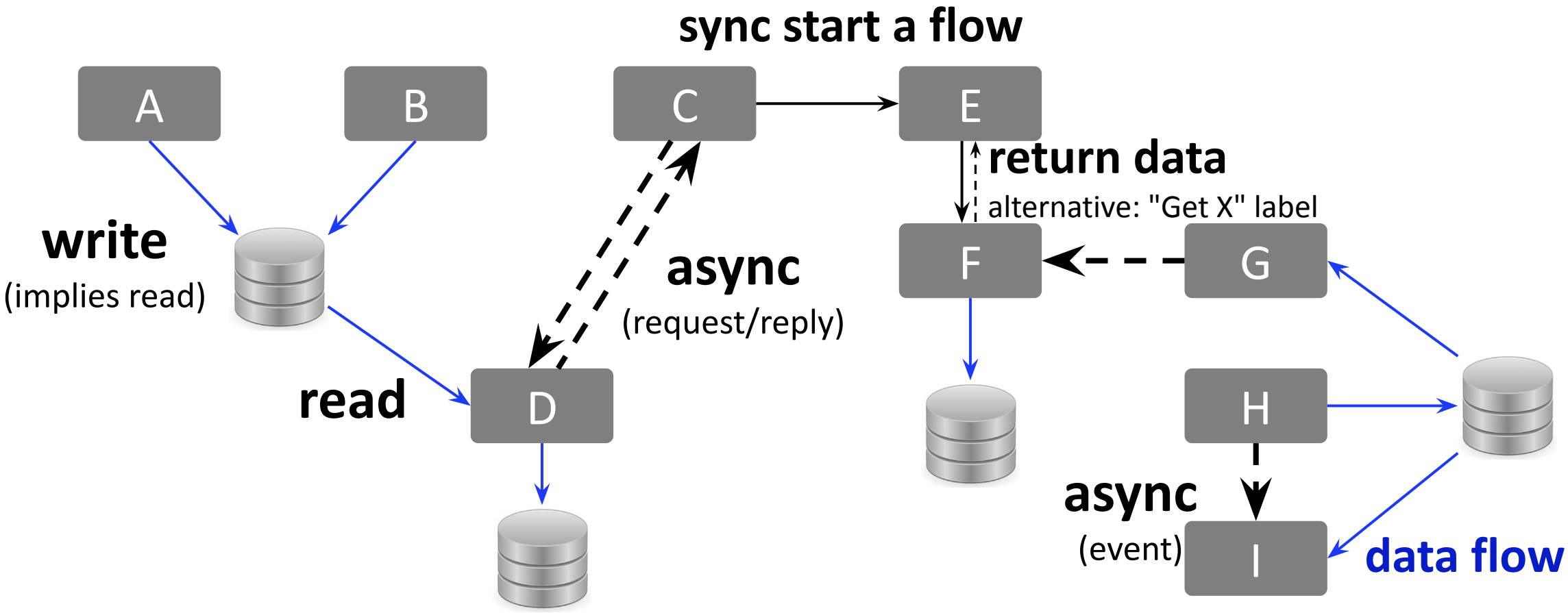
Problems with lines

- undirected lines = ?
- bidirectional arrows = ?
- DB arrows = read or write?
- comm is sync or async?
- control or data flow?



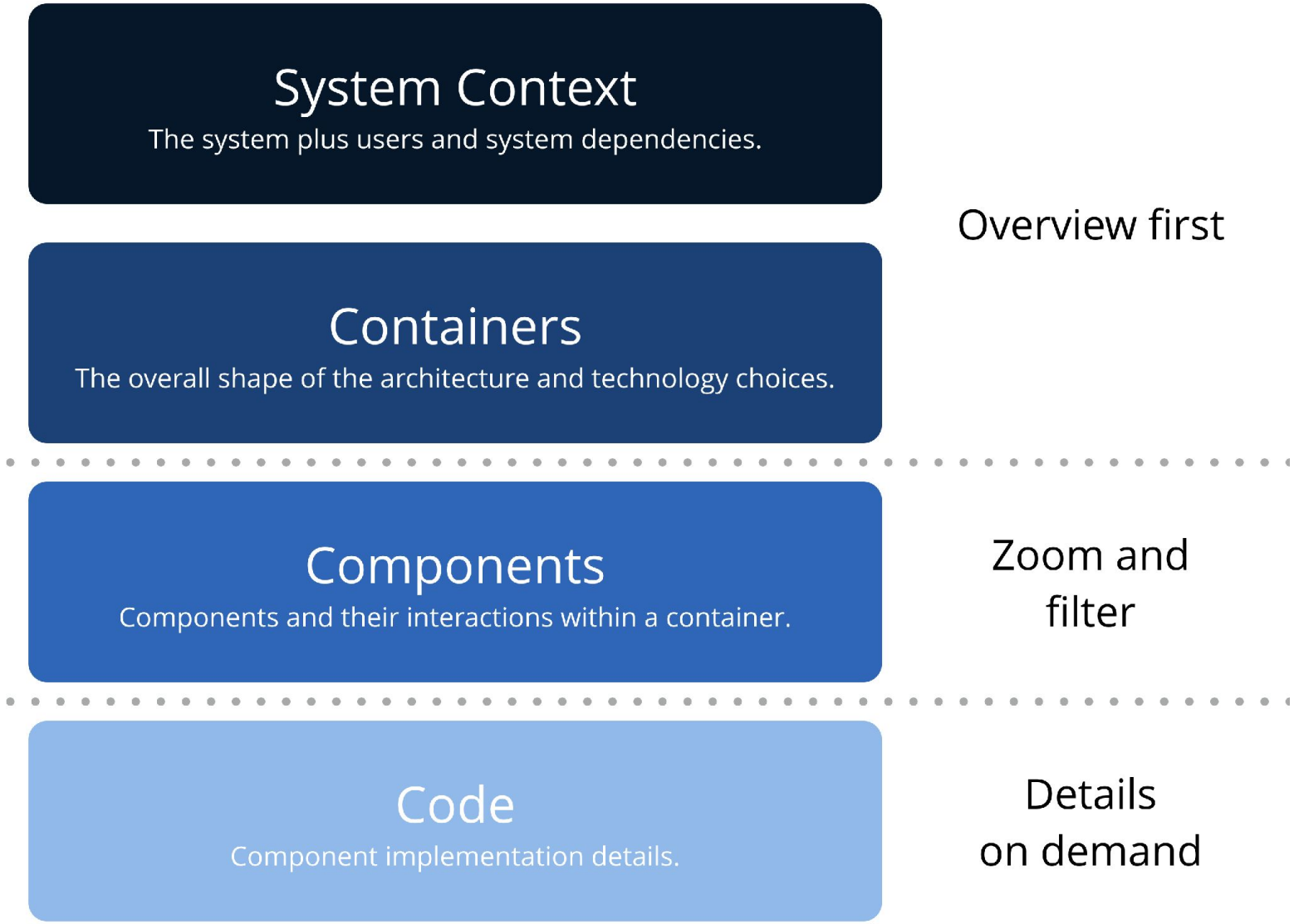
Opinions?
(about lines)

Lines!


sync start a flow
A
B
C
E
return data
alternative: "Get X" label
write
(implies read)
async
(request/reply)
F
G
read
D
H
async
(event)
I
data flow

C4 Model

by Simon Brown



https://c4model.com/

TODO redo anim

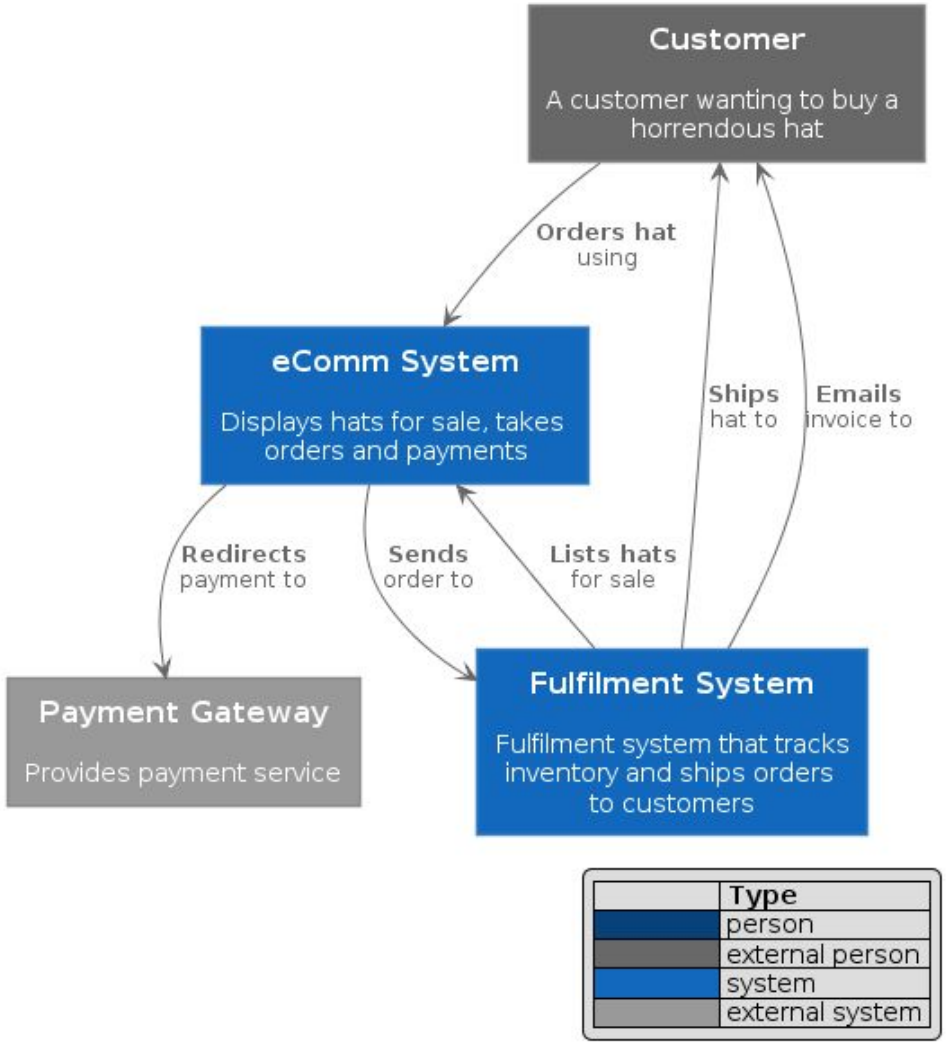
C4Model.com
by Simon Brown

1) Context
2) Container
3) Component
4) Class

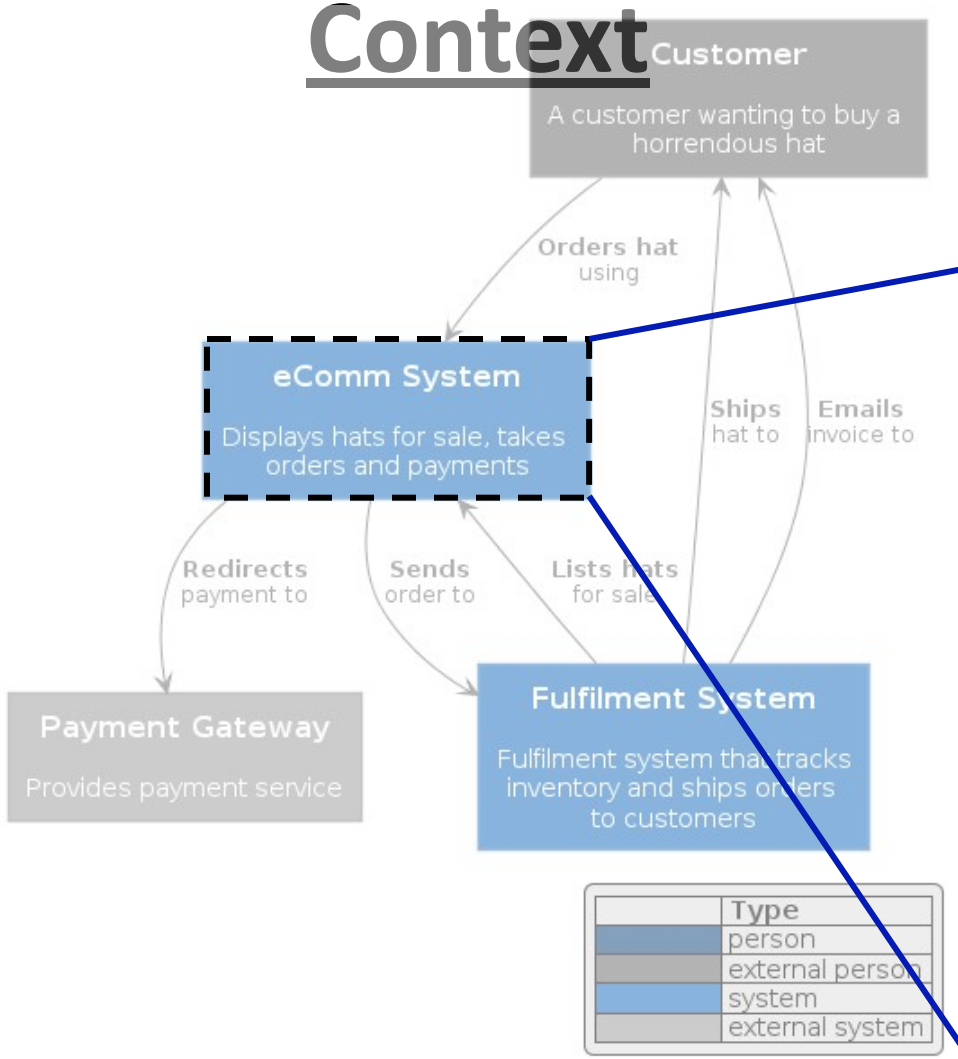
from high⇒low

1)

Actors & External Systems interacting with my system as a black-box

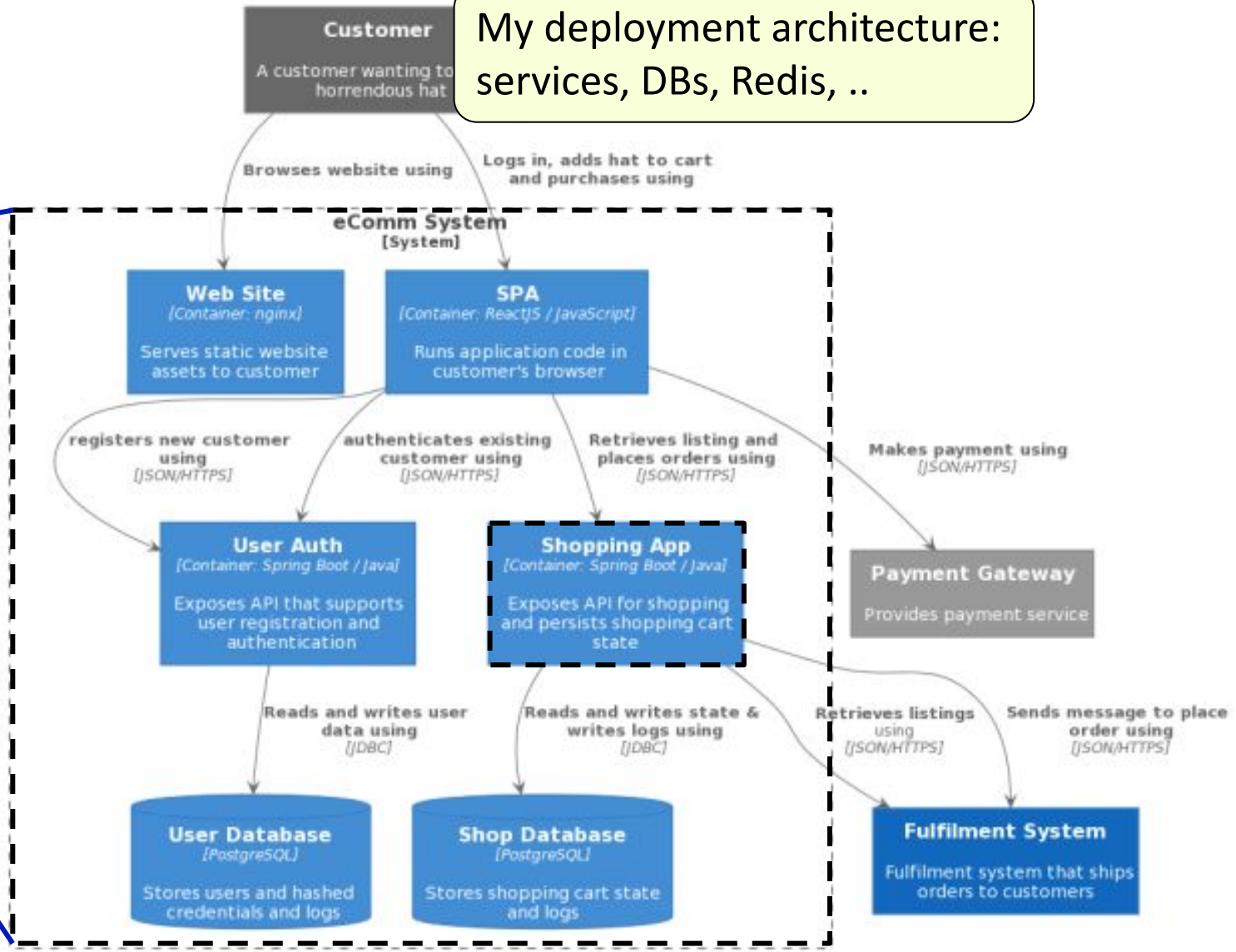


1) Context

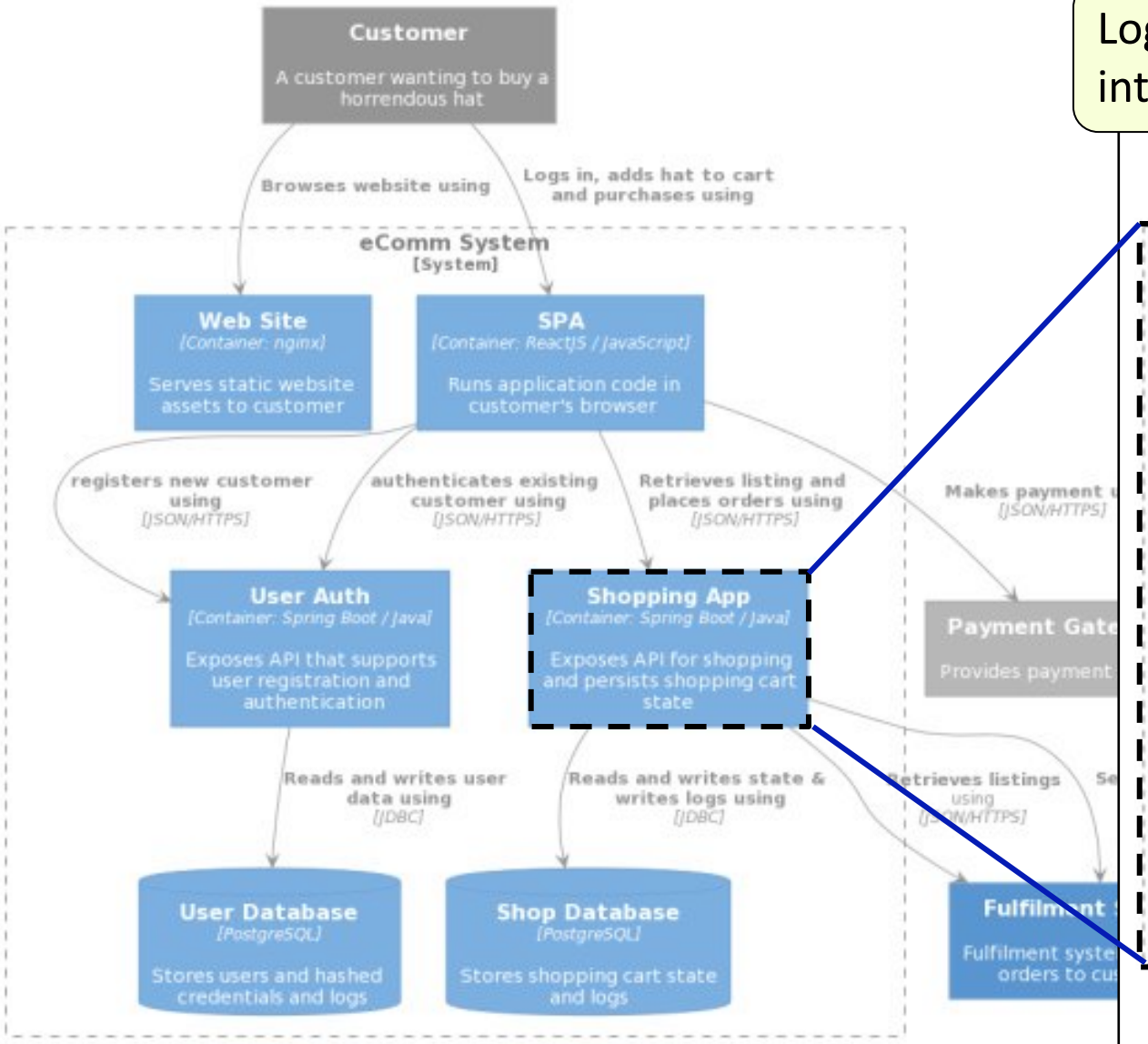


2) Container

My deployment architecture: services, DBs, Redis, ..



2) Container



3) Component

Logical architecture: internal packages/modules

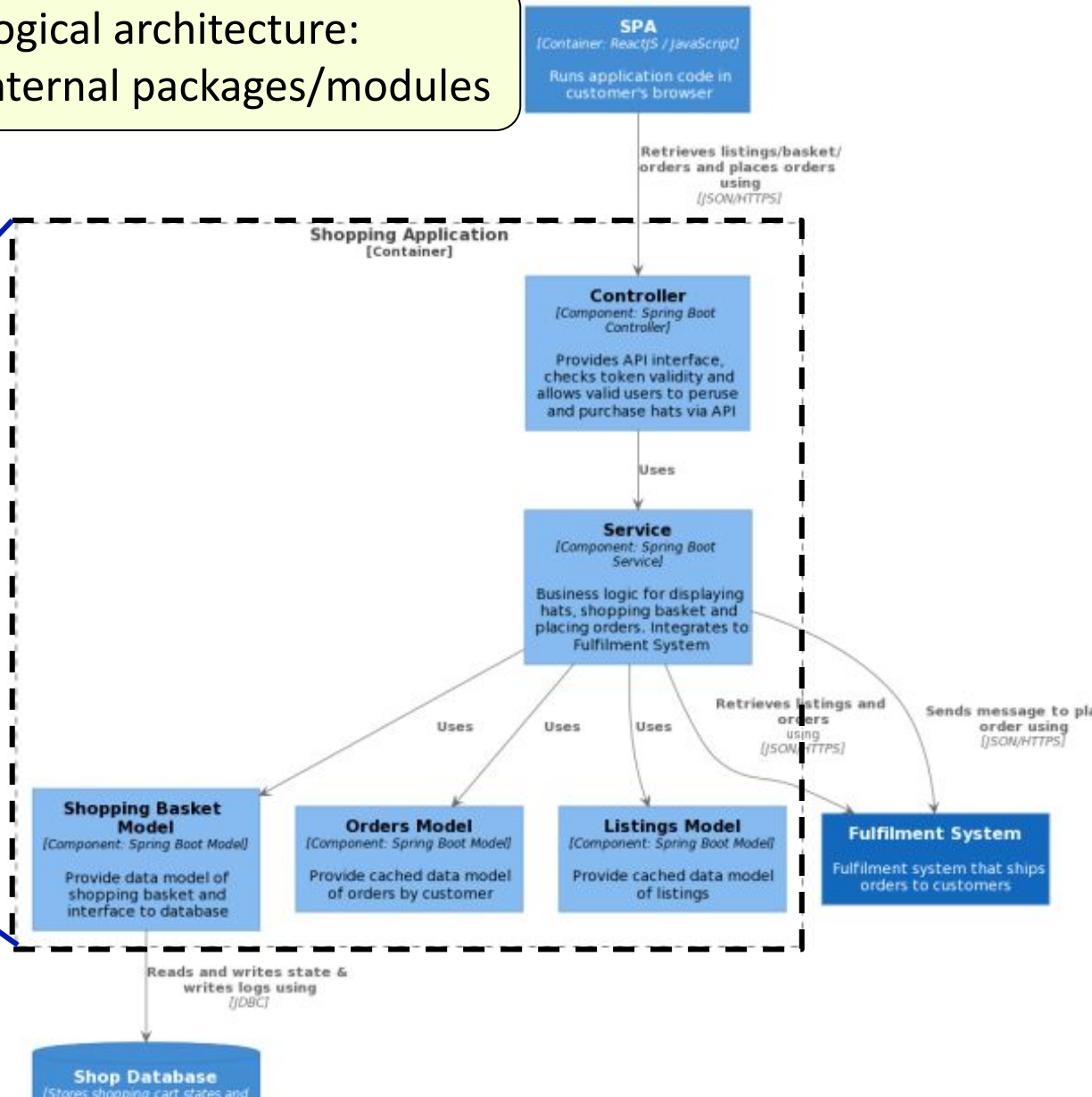


Diagram Tools

💖 Text to diagram: PlantUML.com, Mermaid.JS.org
can be rendered by IDEs, Confluence, Wiki...
best for sequence, state, activity (less aesthetics options)

```
in diagram.puml :
@startuml
class Car
Driver - Car : drives >
Car *- Wheel : have 4 >
Car -- Person : < owns
@enduml
```

⭐ Online Collab draw.io, miro.com, lucidchart.com, ExcaliDraw.com

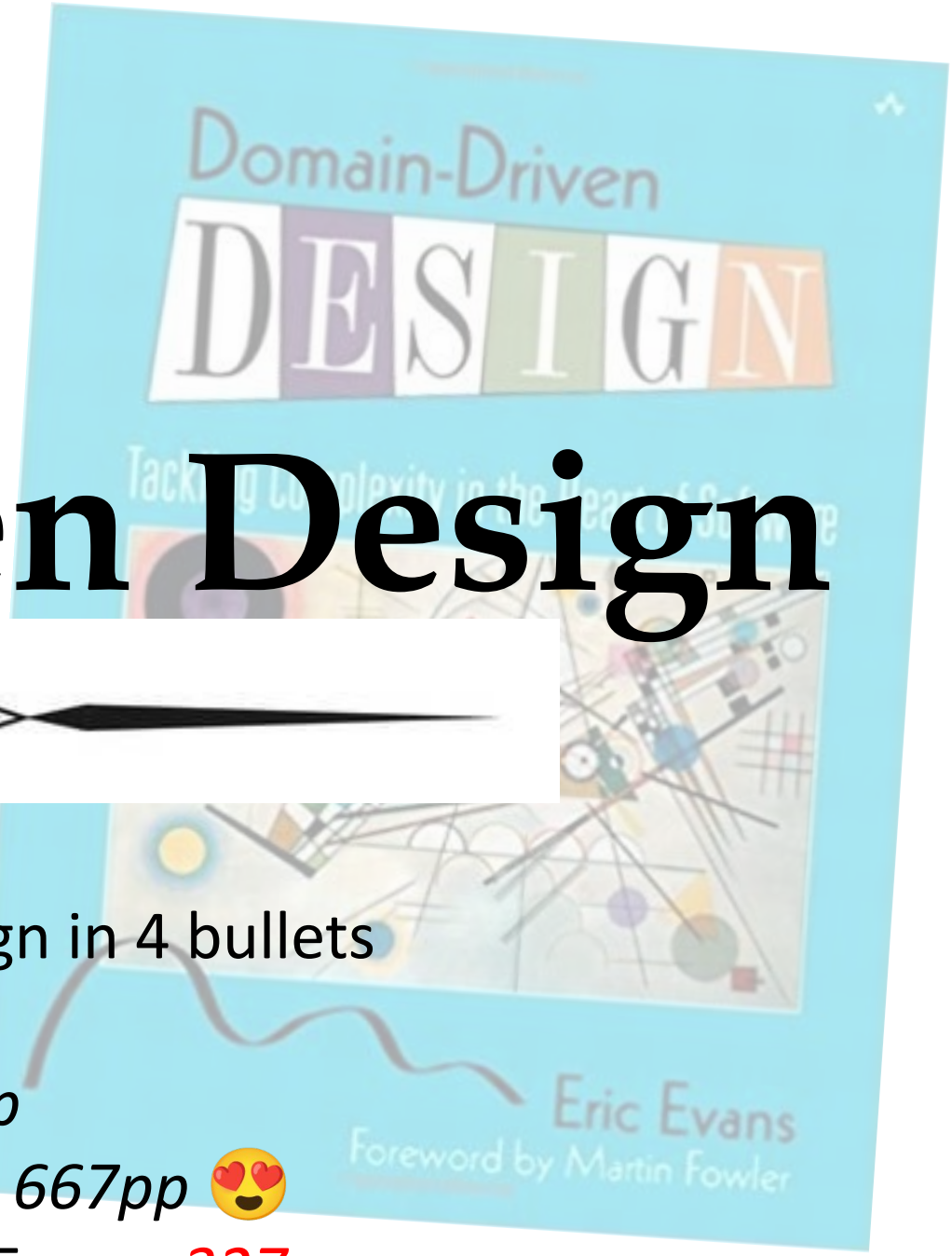
👴 Desktop: Enterprise Architect, MagicDraw

... Yours?

In IntelliJ, right-click a package > Show Diagram

Used to keep code maintainable
in some of the
most complex systems on Earth

Domain-Driven Design



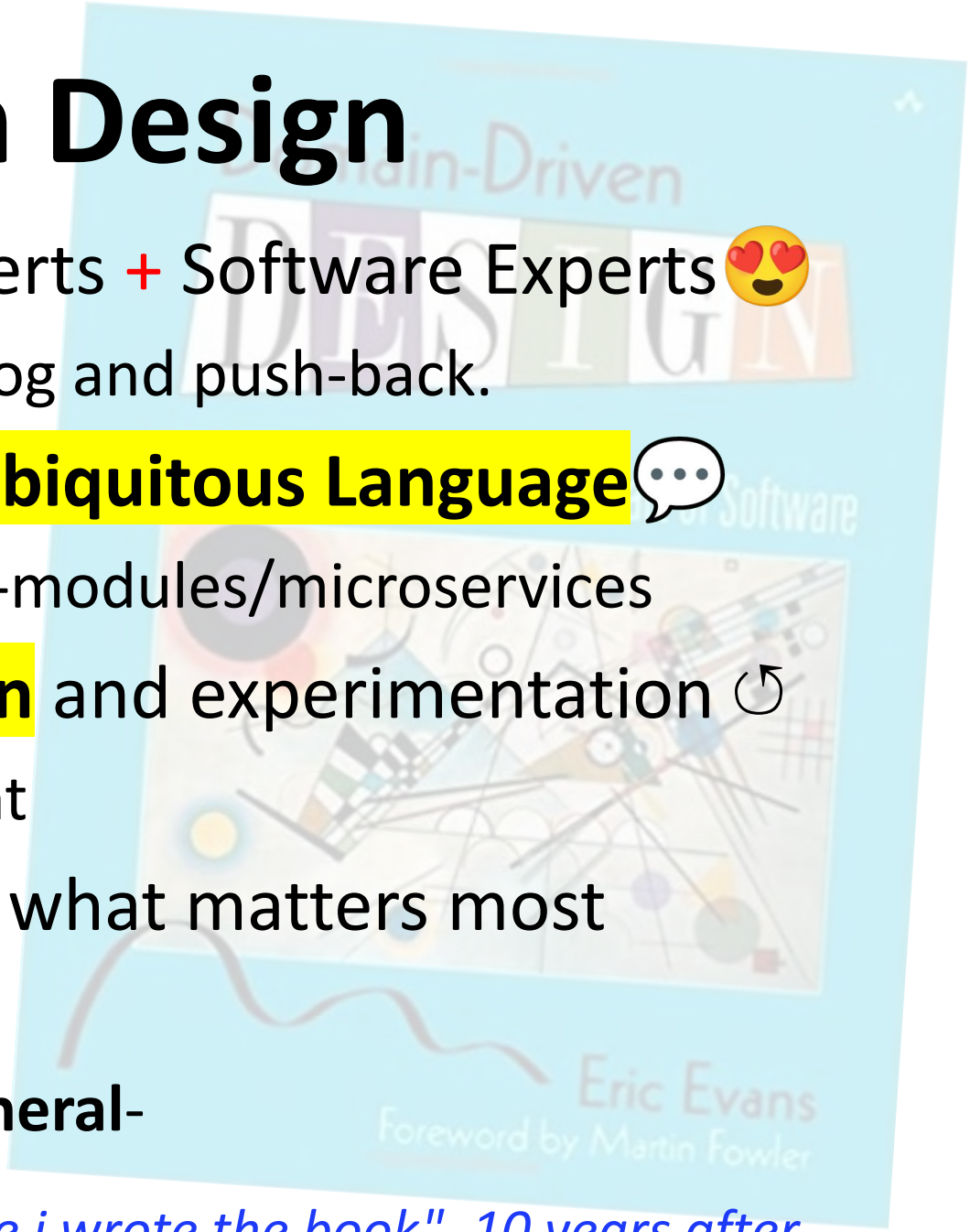
easy

> 📺 Talks @ DDD Europe Conference
> 💬 Hey AI, summarize Domain Driven Design in 4 bullets
> 📖 **DDD Quickly** *106pp*
> 📖 **DDD Distilled** by Vaughn Vernon, *137pp*
> 📖 **Implementing DDD** by Vaughn Vernon, *667pp* 😍
> 📖 **Domain-Driven Design** in 2003 by Eric Evans, *327pp*

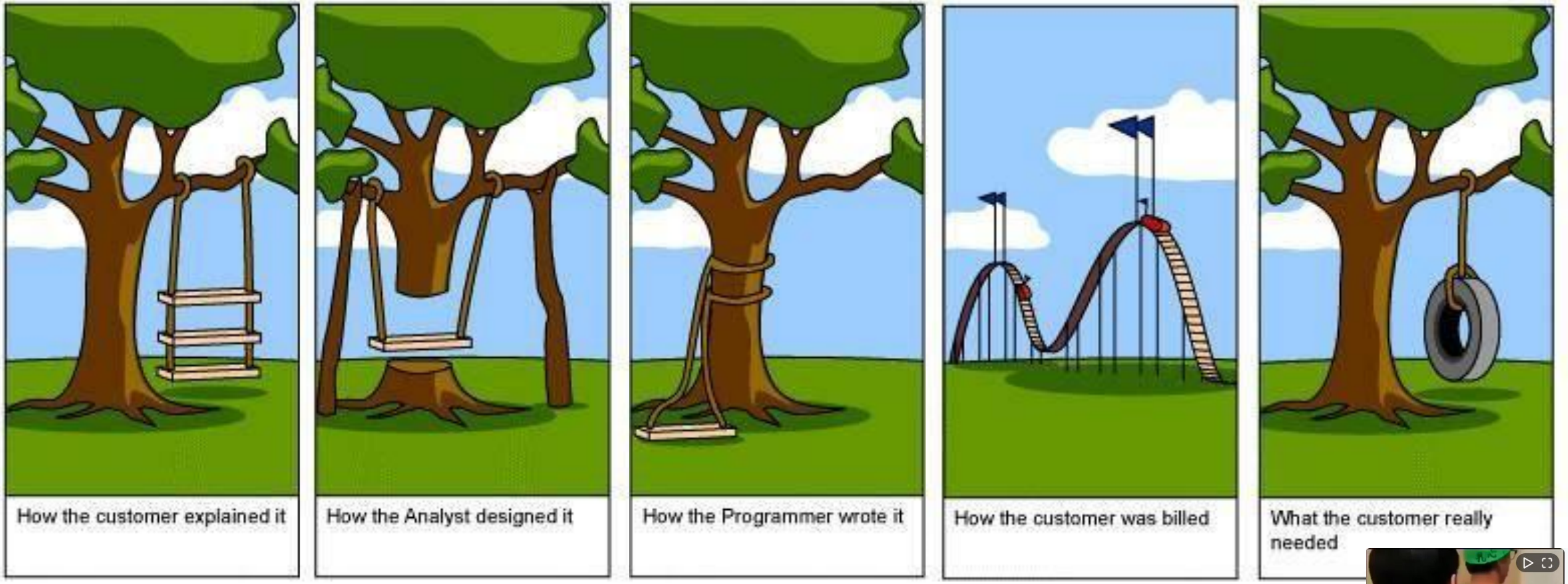
hard

Domain-Driven Design

- **Creative Collaboration** of Domain Experts + Software Experts 😍
  - Barriers: language, human proxies? Dialog and push-back.
- **Bounded Context** speaking different **Ubiquitous Language** 💬
  - Separate functional areas in subdomain-modules/microservices
- **Deepen Domain Model via exploration** and experimentation ⟲
  - Play with code for deeper domain insight
- **Distillation of Core Logic** 💎: declutter what matters most
  - Domain *agnostic* to infrastructure
  - **core-** subdomain vs **supporting**- and **general**-



--- Eric Evans in "What I've learned about DDD since i wrote the book", 10 years after


How the customer explained it
How the Analyst designed it
How the Programmer wrote it
How the customer was billed
What the customer really needed


Code
Developer
Business Analyst
Client

Domain Model

= representation of **concepts in your problem**.

= data structures used in **your most complex logic**.

= often persisted in a DB

⭐ **Kept Internal** => can evolve without breaking clients.

Evolve Domain Model

1. **Deep structures** (not flat) with fewer fields
   - 👉 Customer#shippingAddress: ShippingAddress (=**VO**)
2. **Hosting bits of domain logic**
   - 👉 Customer#canReturnOrders(), #isNaturalPerson()
3. **Protect Domain Invariants**
   - 👉 Customer#validate(username$^{\neq null}$)
   - 👉 @NotNull

Value Object

(Design Pattern)

- **Immutable**, typically small structure
- **With no persistent identity** (PK)
  - Unlike a mutable Entity that *changes*, a VO is *re-created* (new ...)
- **Can be part of domain model** 👑
  - Unlike a DTO that just moves data to/from APIs json/xml
  - Not necessarily persisted

Use to:

❤️ **Enrich semantics:** Money{amount, currency}

❤️ **Shrink large entities:** ShippingAddress{city, street, zip}

vs **Anemic** -
= only getters & setters

DDD terms
Rich Domain Model

Valuable for complex 🧠 domains

```
public class Customer {
  ... few fields⚠️ ...

  // explain the data meaning💖
  public boolean isIndividual() { return this.legalEntityCode == null; }

  // implement business rule💖
  public boolean canReturnOrders() {
    return this.isGold || this.isIndividual();
  }

  // guarding domain invariants💖
  public void validate(String user) {
    if (this.status != DRAFT) throw new IllegalStateException(...);
    this.status = VALIDATED;
    this.validatedBy = requireNotNull(user);
  }
  public void ~~setStatus(status) {..}~~ // prevent unprotected changes
  public void ~~setValidatedBy(user) {..}~~

}
```

tl;dr

put 1-5 lines of (reusable) logic inside the data model

* Unusual? "Implementing DDD" book by V. Vernon will convince you
* Unusual? "Implementing DDD" book by V. Vernon will convince you

Bloated Domain Model

```
public class Contract {
   ... dozens of fields 🤯 ...

   // coupling to data structures ❌
   public void f1(BigStructure20Fields) {..}

   // coupling to behavior ❌
   public void f2(SomeRepo) {..} // DB calls
   public void f3(ApiClient) {..} // API calls
   public void f4(ComplexService) {..} // potential deep effects

   // coupling to presentation ❌ = specific to a single use-case
   public String toXmlForAmazon() / getCreatedDateFormatted() {..}

   // complex logic ❌ □ move to a stateless class
   public .. f() {>10 logic lines} // never @Mock this instance

}
```

Push logic in your data structures to:

✅ simplify complexity

✅ reuse easier

✅ enforce rules

Anything > 1000 lines

GOD CLASS

Written with NO Code Review and NO Unit Tests

Many years ago, in the largest class of my **ISP**:

Contract.php

Internet Service Provider

4,500 lines of code, having

120 fields, out of which:

20 dependencies to query DB and call external API

12 temporary fields set and read by various methods

200 methods (largest: 500 lines of complexity hell)

4 subclasses (for extra fun) 🤯

10% unit-test coverage 😬

Zero refactoring 😱

10 years ago, my first consultancy gig:

InvoiceGenerator.java (extra-risk)

35,000 lines of code, having

1000+ bugs over 18 years of life (C++)

100 fields, out of which:

>10 dependencies

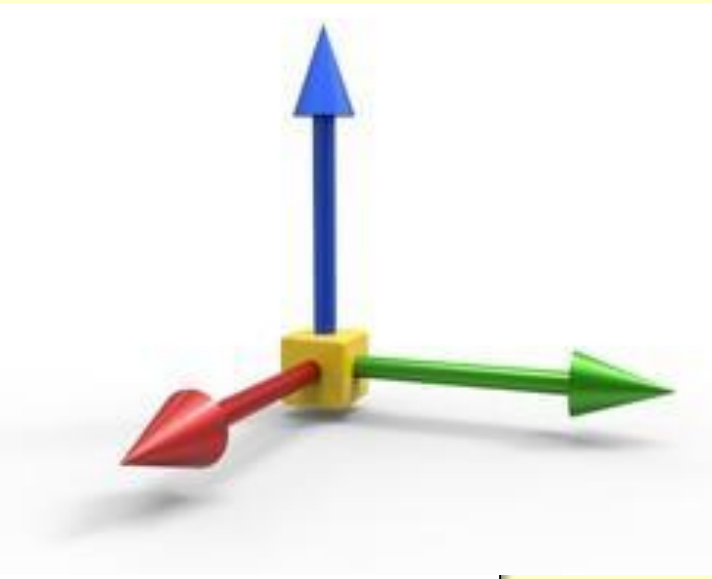
>10 temporary fields

500 methods, largest: ~1000 lines taking 14 params

3 axis of Complexity:

- data semantics
- orchestration
- cyclomatic complexity

3 Axis of Complexi



- **data semantics** ≈ many attributes

```
class Data{a,b,c,d,.....z} =🤯
```

- **orchestration** ≈ many dependencies

```
class Facade {@Autowired d1,d2,..,d20;} =🤯
```

*Overviewed by IntelliJ Dependency Matrix, PlantUML from code Maven Plugin …

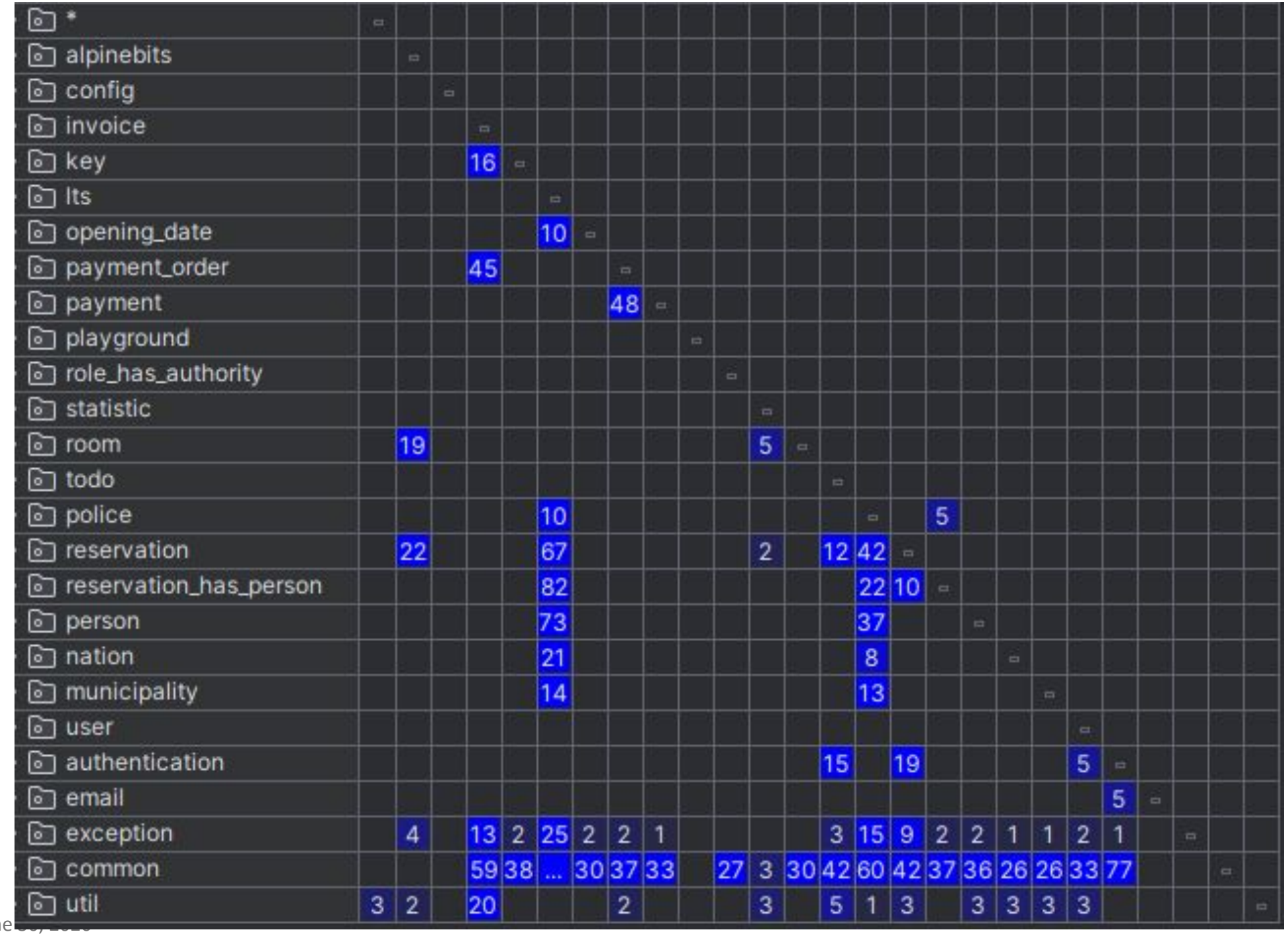
- **cyclomatic complexity** ≈ branching ~ no(tests)

```
try {if {for {if {if {if {switch =🤯
```

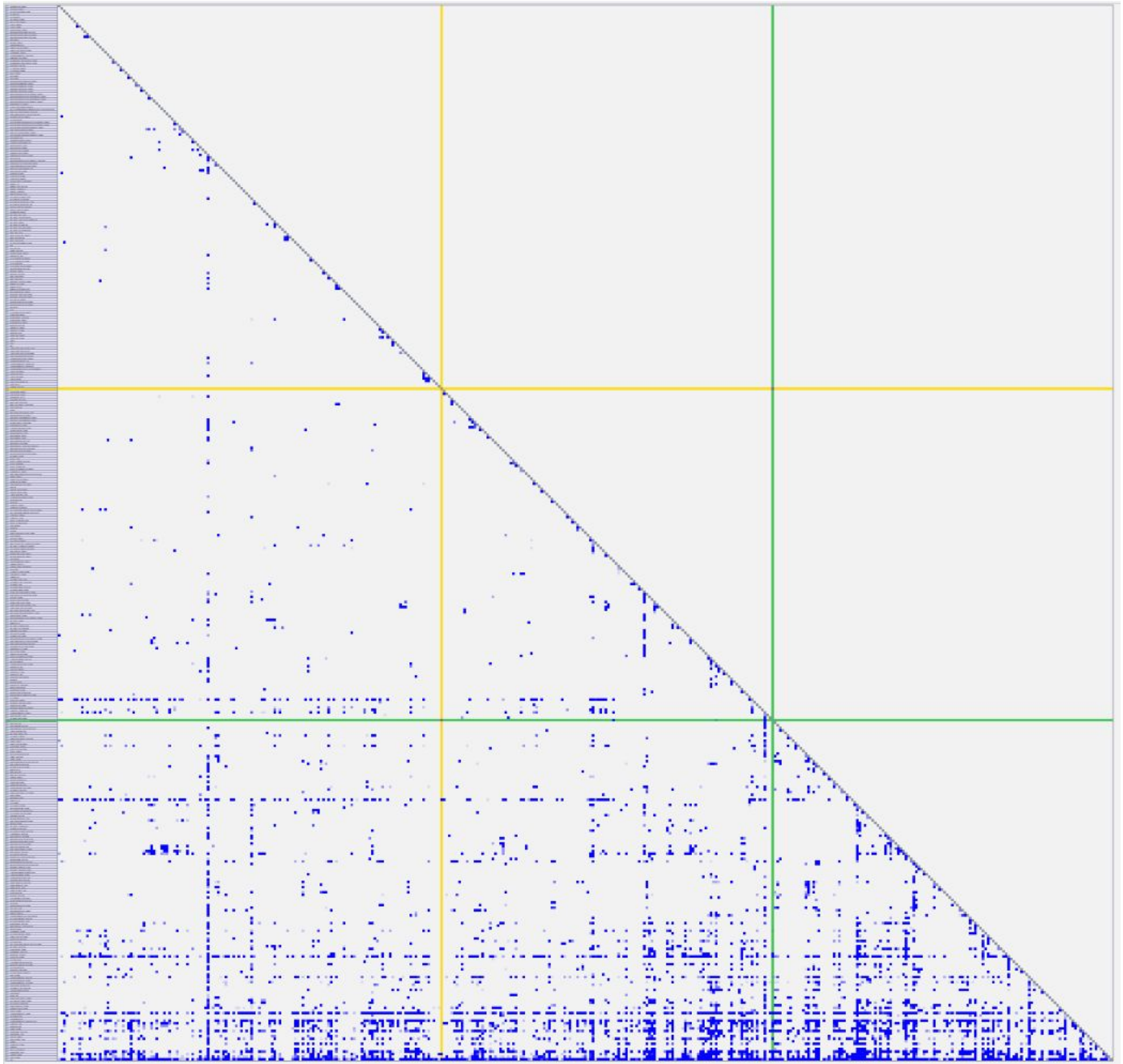
*Can be measured by IntelliJ plugins: Code Complexity😍 and CodeMetrics

The history the 35.000 LOC class (sketch)

- The first 3000 lines of code of this class were written by C++ developers porting an ancient system
- The next 2000 lines were added by some junior contractors
- In 2004 the company internalized their software engineering
- The class was called InvoiceGenerator, and there was a terrible fear to break things, since unit tests was lacking.
- In 2006 massive code was added **quick** to onboard a new customer
- **No mandatory code review/Sonar; No unit-testing until here**
- New code was easier to add inside the 10.000 LOC class ("gravity")


*
alpinebits
config
invoice
key
lts
opening_date
payment_order
payment
playground
role_has_authority
statistic
room
todo
police
reservation
reservation_has_person
person
nation
municipality
user
authentication
email
exception
common
util

In a large project:



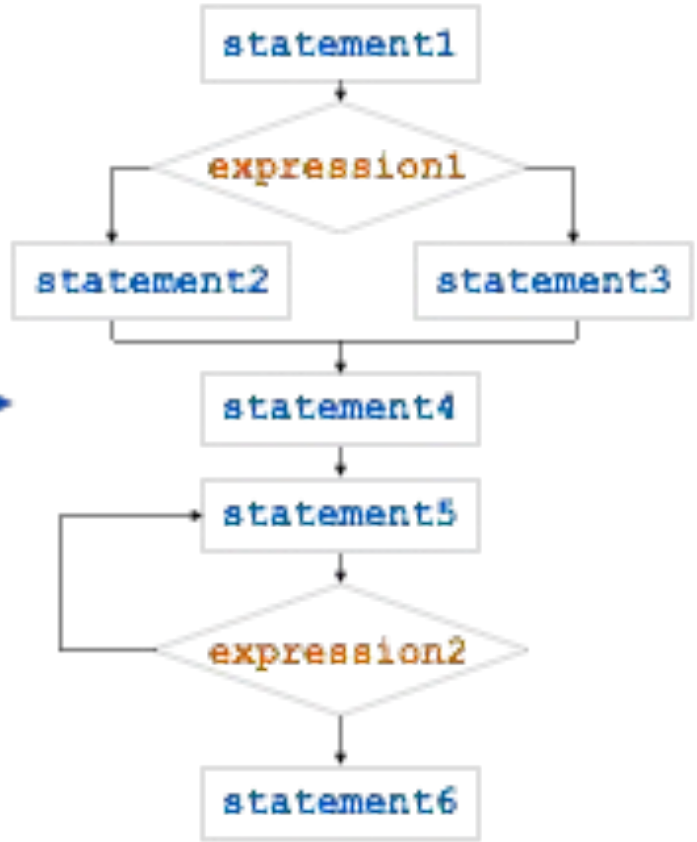
Cyclomatic Complexity

= #edges - #nodes + 2 = 8 – 10 +4 = 2

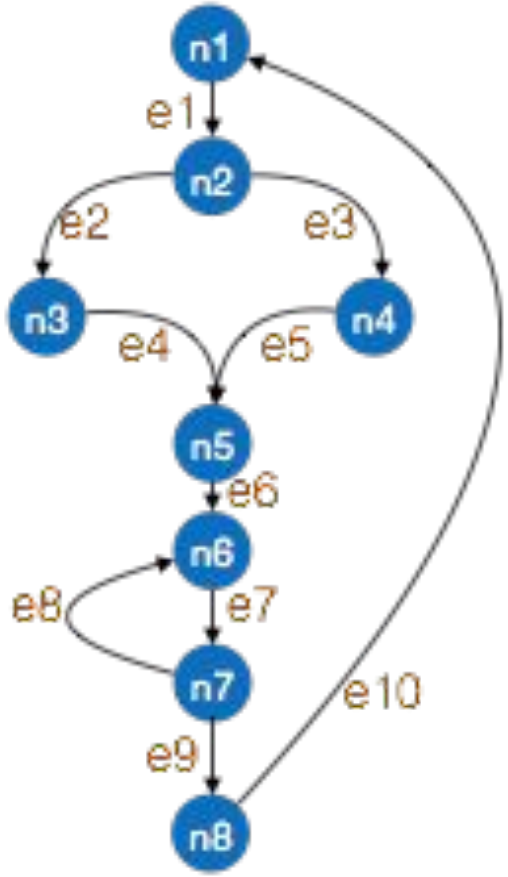
Code

```
statement1

If expression1
        statement2
else
        statement3

statement4

do
        statement5
while expression2

statement6
```

Flow-Chart



Flow-Graph



https://en.wikipedia.org/wiki/Cyclomatic_complexity

`@Service, @Singleton...`

keep in stateless classes that are easy to understand and unit-test

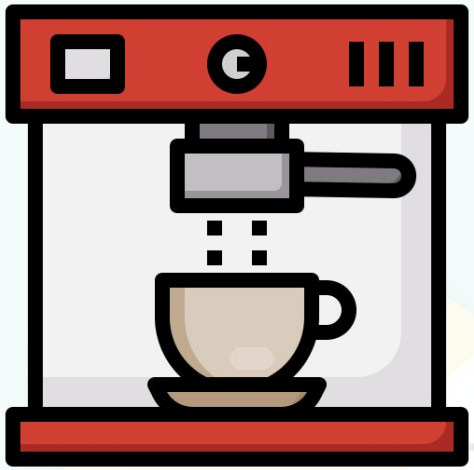
Separate **Complex Logic** from **Stateful Objects**

Validation vs Business Rule

- **Validate any data** entering your system ASAP!
  - Received requests / messages
  - Responses from APIs you call - often neglected ⚠️
- **1) Payload Validation** □400, 💖 best via annotations
  - name is @NotBlank and @Size(min=5) characters, valid @Email...
- **2) Business Rule** (may use DB/APIs) □500
  - no two customer have the same email

Language

Language



... client ...

It depends on who's talking!

Meaning of word "Client"?

User moving the mouse in UI
—Frontend Dev

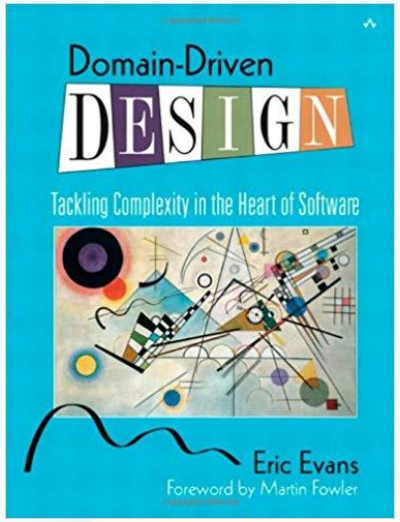
Company that pays us money
—Sales

Application using OAuth
—Security Team

Caller of my API
—Backend Dev

Library used to call a REST API
—Backend Dev

Developer using my library
—Platform Team



Ubiquitous Language

Used by everyone in a team: BE, FE, QA, PO, BA

in requirements, diagrams, emails, Slack

phone, meetings, ☕ break

in code 💖

Word 1 ↕ 1 Concept

What if the correct one is "Client"? 😬

```
Client
Customer customer;
```

Rename

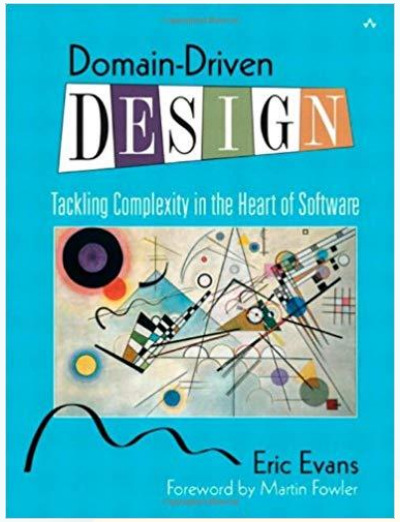
Map to a new VO in my Domain Model:

```
== externalProduct.code()
```

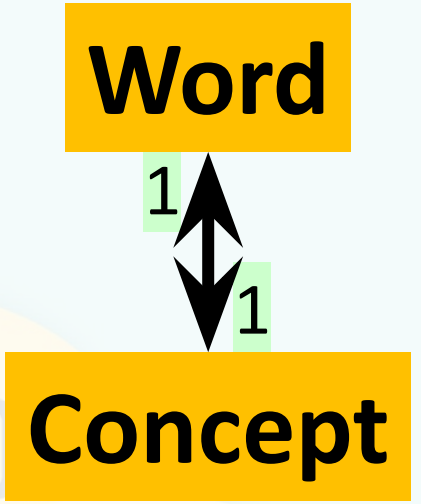
in a 2000 😵‍💫 lines file:

```
if (product.code() == itemDto.materialNumber()..
```

ItemDTO from an API we call — External Corruption



Ubiquitous Language



☠️ Yet Another Naming Horror Story ☠️

"supplier part number" = external PK from supplier

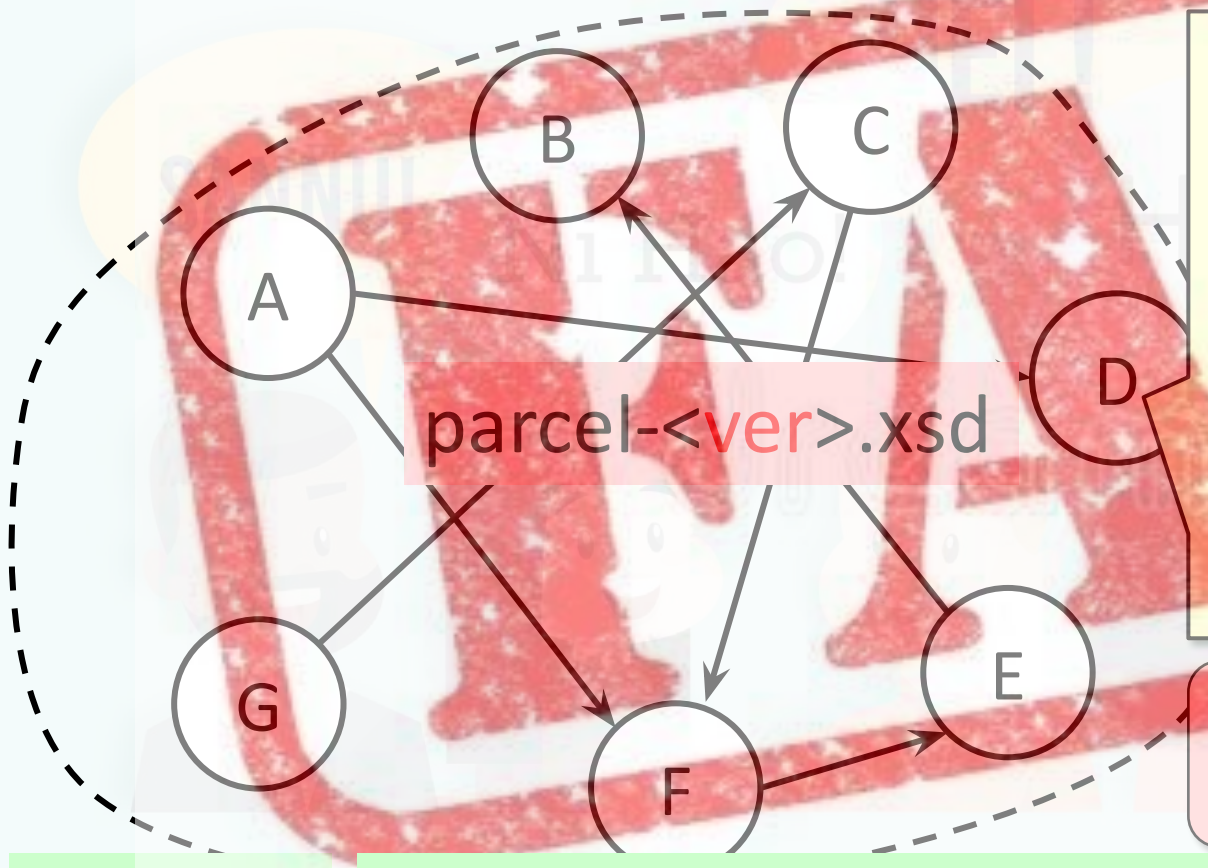
"part number key" = internal PK = "PNK"

var partNumber = f(); 😱😱😱

Ubiquitous Language Cannot be Universal

*In the **Postal** Domain, in 2013, during the **Age of SOA** (.wsdl, .xml, SOAP),*

*trying to **rewrite 15y old legacy** with 7 teams x 5 devs*



Universal Model

shared by 7 teams

- Volatile: parcel-v17.7.xsd in a few weeks
- Huge & dilluted: 120 attributes
- Painful meetings to change it 😭

...until added: metadata: Map<String, ?>

Developers can add any key here
(instead of arguing with 6 teams)

What to do? > Let each team develop their own view about a 'parcel' in their Bounded Context

Ubiquitous Language

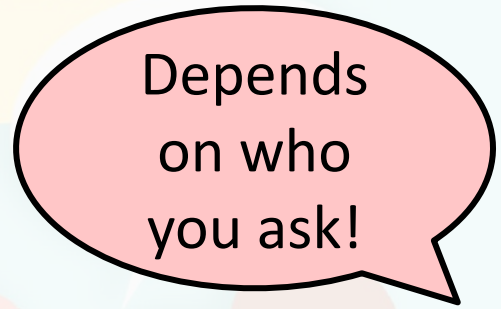
Should be shared by a few people from the same

Bounded Context

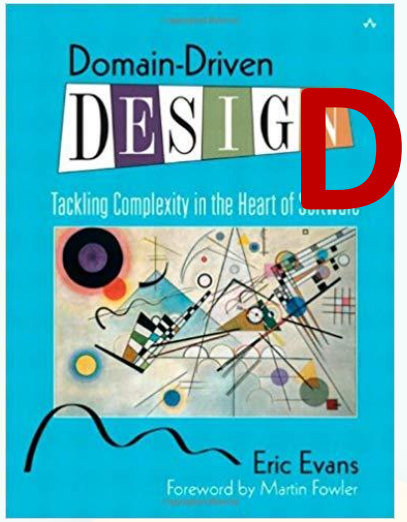
≈ 1-2 team

In an eShop we all agree what a "**product**" means.

But:

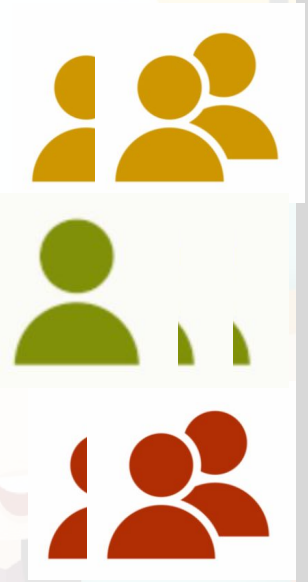


What attributes should a product have?



Domain Model Dilution

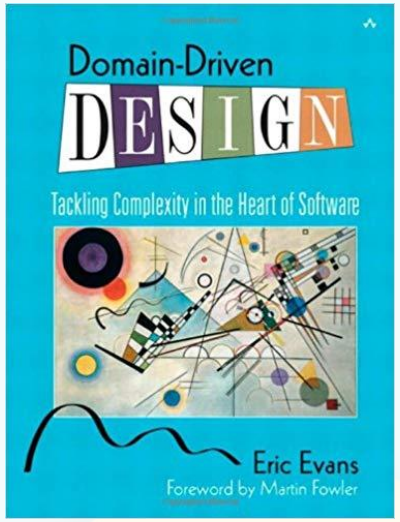
Developers specialized in parts of the code



```
class Product {
    id
    name
    description
    attributes
    price
    sku
    quantity
    locationCode
} // 70 attributes!!
```

Which are required?

😱



Split the Domain

~~class Product { 70 attributes ☠ }~~

Bounded Context

Catalog

```
class Product {
   id
   name
   description
   attributes
   photo +7 more
}
```

Pricing

```
class PricedProduct {
   id
   price
   offers, +5 more...
}
Bundle {..}, Offer{...}
```

Inventory

```
class Article {
   id
   name
   sku
   quantity + 4...
   locationCode
}
```

Agnostic Domain

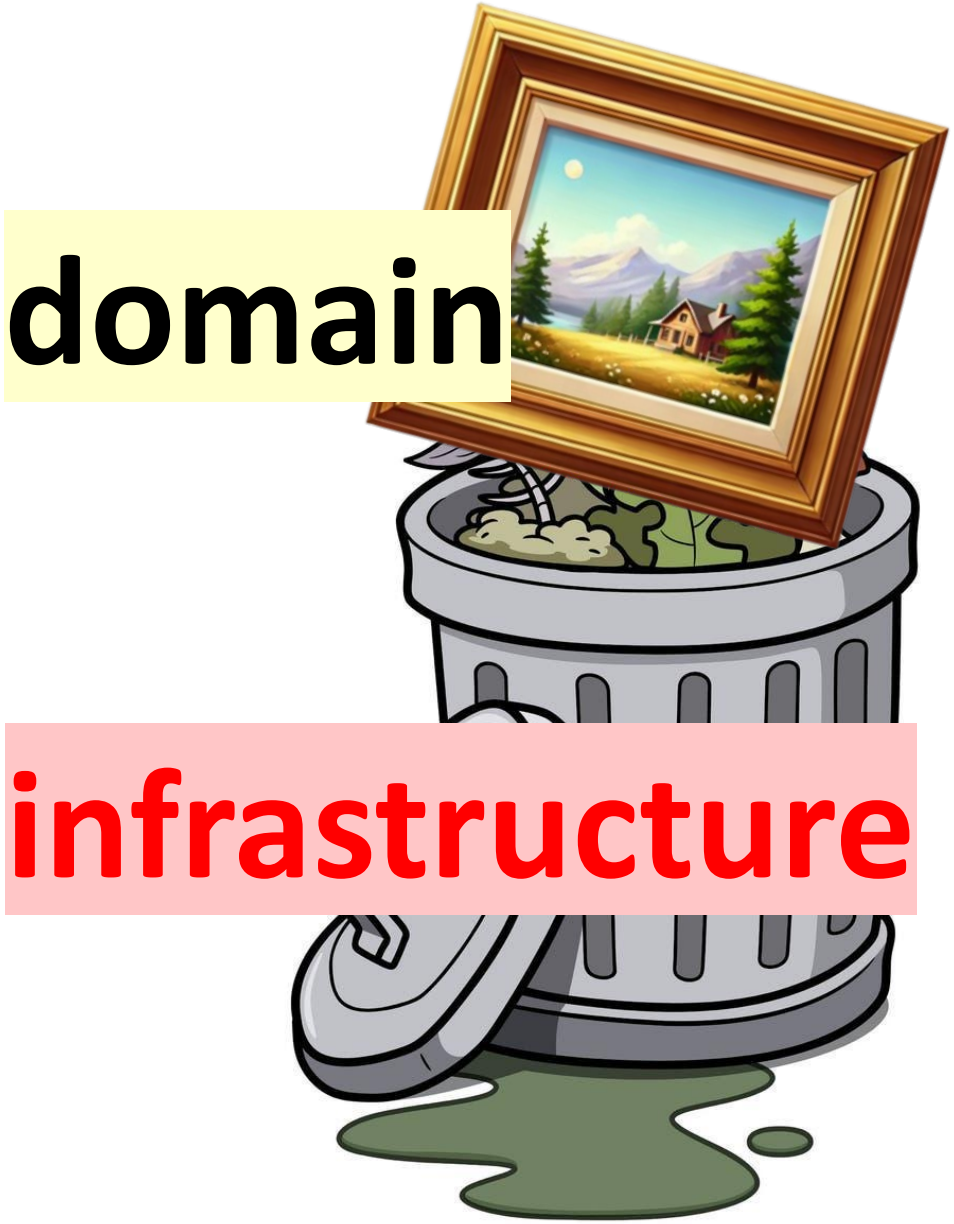
dependency inversion

What is the main **purpose** of our application?
Reason for its budget 💰?

Let's call it "**domain**"

Separate
1) Intrinsic complexity of
the problem we are trying to solve

from
2) Accidental complexity of
calling an API, exporting XML...



*Architecture = the **art of drawing boundaries***

Protect Core Complexity
against the External Corruption

Debate

domain

Domain Logic
(core complexity)

Infrastructure

("accidental complexity")

parse JWT
send email over SMTP
call an ugly API
import a file
monitor a batch job
start jobs once/cluster
...

Stuff I could copy-paste to another project
=> use an OSS library 😍

Foreign DTOs*

are Evil (by default)

*Data Transfer Objects – data structures that move across APIs

Beware of Foreign Data Structures

External DTOs (=APIs of others) are:

- **Different perspective** 🐔 (foreign bounded context)
- **Bloated** with more fields than we need
- **Invalid data**: nulls, doesn't follow my constraints
- **Anemic (holding no logic),** if in client.jar / generated
- **Mutable** without annotations or tricks
- **May change in time** (imagine v2.0😬)
- I could **change the provider** tomorrow

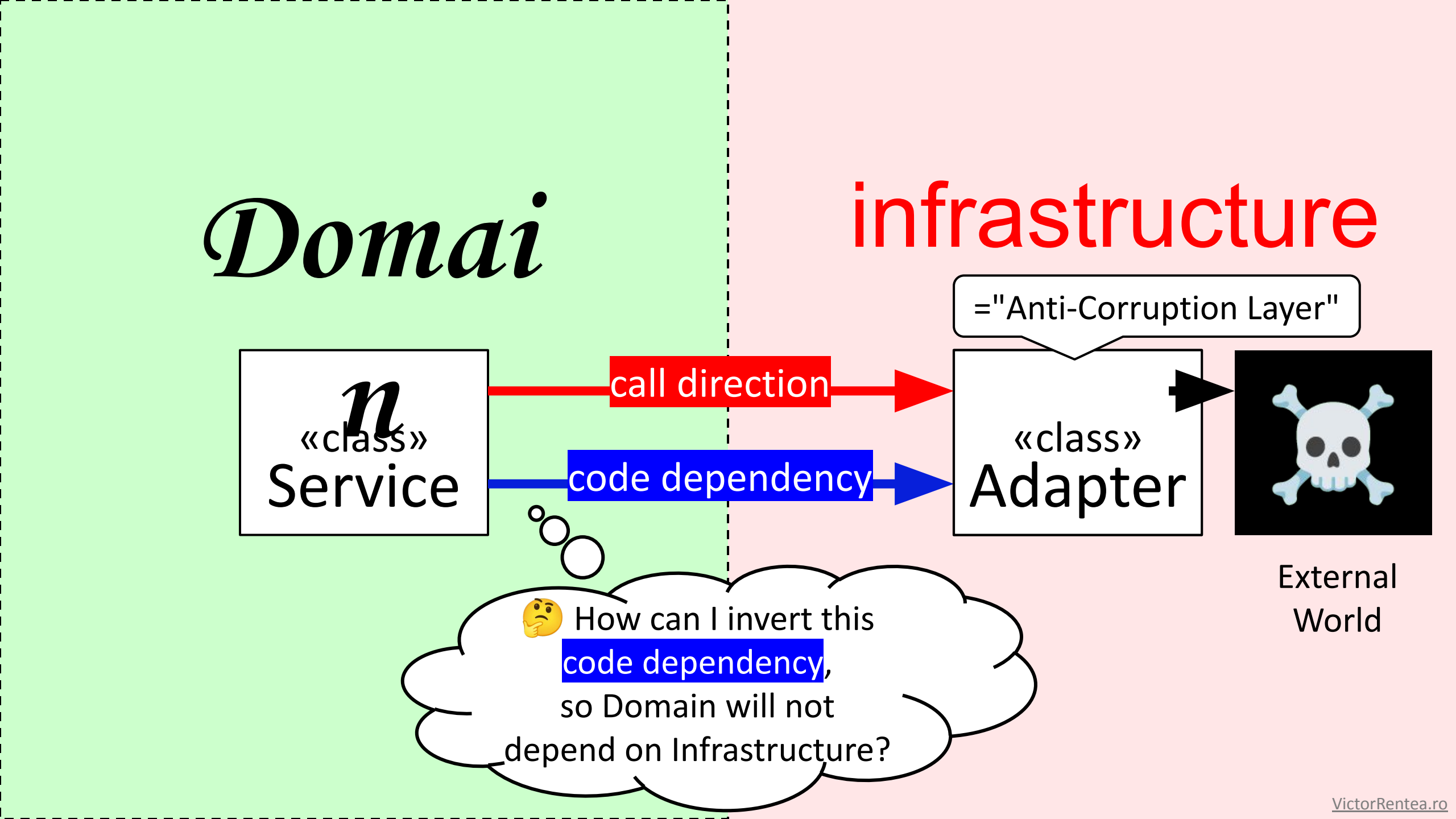
In complex logic we want:

- 💗 Our ubiquitous **language**
- 💗 **Small** data structures
- 💗 Objects that guard their **invariants**
- 💗 **Rich** objects with behavior (OOP)
- 💗 **Immutable** objects
- 💗 **Stable** structures
- 💗 **Contract stays similar**

Agnostic Domain

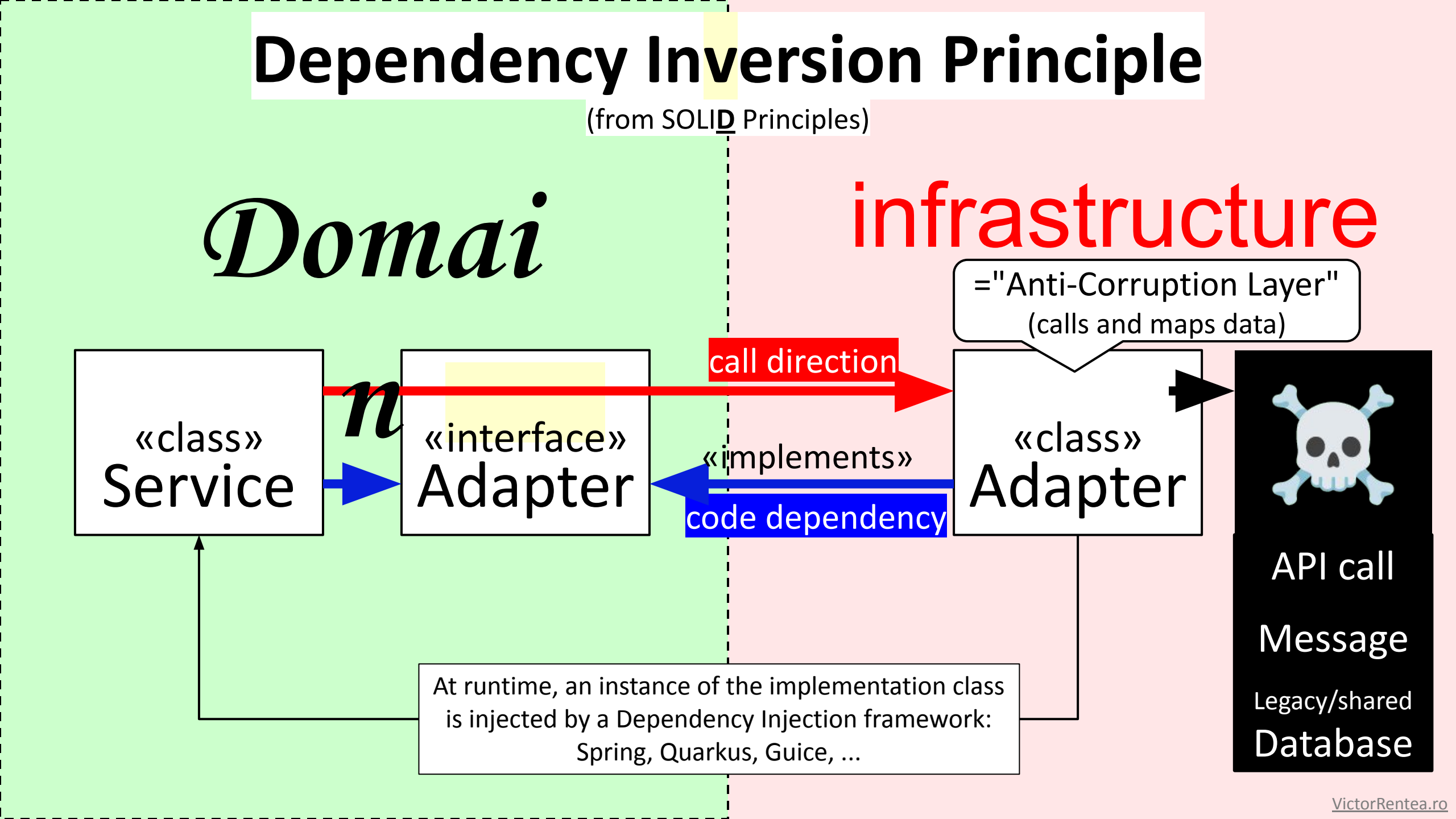
should not depend on the

EXTERNAL CORRUPTED WORLD



Dependency Inversion Principle

(from SOLI**D** Principles)



Keep core Complexity inside

Separate your **Agnostic Domain**

from **External Corruption**

= Infrastructure

Enforcing Code Dependencies

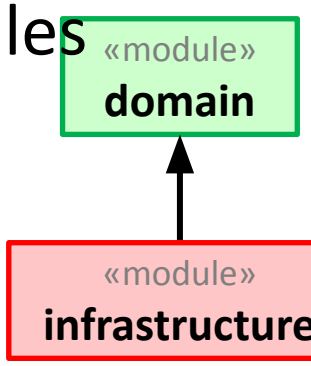
1) Separate build units: eg Maven/Gradle modules

✅ Visibility++, good for big/junior team

✅ No one can @Disabled/Edit the @Test (use CODEOWNERS🤔)

☹️ Longer build time

😫 All-or-nothing: can't track refactoring progress




```
<artifactId>clean-domain</>
...
```

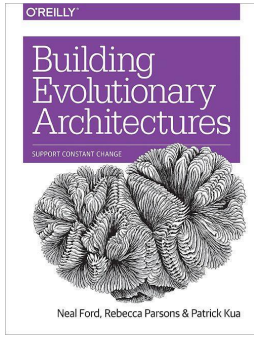
```
<artifactId>infrastructure</>
<dependencies>
  <dependency>
    <groupId>victor.training</>
    <artifactId>clean-domain</>
  </dependency>
  ...
```

See here how to check Maven dependencies with ArchUnit

2) ArchUnit @Test doing static code analysis

C#: https://www.ndepend.com/
PHP: https://github.com/qossmic/deptrac
TypeScript: https://github.com/MaibornWolff/ts-arch

```
@Test
public void dependencyInversionTest() {
  ArchRuleDefinition.noClasses().that().resideInAPackage("..domain..")
    .should().dependOnClassesThat().resideInAPackage("..infra..")
    .check(new ClassFileImporter().importPackages("my.corp.app"));
}
```



Architecture Fitness Function can measure how far you are from goal

Architecture Fitness Function

```
@Test // running on CI, written by devs, CODEOWNERS-reviewed by architects
void test() {
  var violations = rule.evaluate(classes).getFailureReport().getDetails();

  // (1a) Track progress towards goal
  assertThat(violations).hasSize(123); // initially
                        .hasSize(30); // +2 months of occasional refactoring
                // ❌ fails with 31🔼? => investigate 🕵️
                // ❌ fails with 29🔽? => change to 29 + enjoy🎉
                        .hasSize(0); // 1 year later: done =>
🎉🥳🍾🍻🍻🍻🍻🍻

  // (1b) Freeze rule vs /src/test/resources/archunit-current/* 💡
  rule.check(classes); // or (2) throw on any violation
```

https://www.archunit.org/

ArchUnit Assignment

```
<dependency>
  <groupId>com.tngtech.archunit</groupId>
  <artifactId>archunit-junit5</artifactId>
  <version>1.4.0</version>
  <scope>test</scope>
</dependency>
```

1) AI Prompt: Write me an ArchUnit @Test that checks <rule>, excluding tests
2) Run the @Test on your project (adding archunit dep)
3) If the test already **PASSed**✅, make it **FAIL**❌ by breaking the rule in src/main/

PRO tip: if you can't break the rule w/o breaking compilation, select another finer-grained rule

Example <rule>:

- **Layers**: 'repository..' should not depend on 'service..'
- **Strict Layers**: 'controller..' package should not depend directly on 'repository..'
- **No cycles**: top-level subpackages of 'com.example.(*)' don't form any dependency cycle
- **No APIs in Domain**: 'domain..' should not use any DTOs
- **Standalone util**: the 'util' package doesn't depend on any other package than java/javax
- **Agnostic Domain**: '..domain..' is agnostic to everything else (only depends on itself)
- **Package definition**: All @RestController classes are in ..api.. package
- **No deprecation**: No class uses @Deprecated code
- **JPA-Free:** 'core..' module must not depend on jakarta.persistence
- **Mandate Metrics:** Classes containing a RestTemplate field/@KafkaListener must be annotated with @Timed
- **Controllers** may only access Spring, java, converters, and services.
- **Decent domain classes**: no classes in package 'service' should have > 300 lines

ArchUnit Resources / Ecosystem

Resources:

* User Guide: https://www.archunit.org/userguide/html/000_Index.html
* https://www.baeldung.com/java-archunit-intro
* https://www.archunit.org/getting-started
* https://github.com/enofex/taikai
* https://github.com/rweisleder/archunit-spring
* https://reflectoring.io/enforce-architecture-with-arch-unit
* https://github.com/xmolecules/jmolecules
* Sample ArchitectureTest.java in github.com/victorrentea/clean-architecture

Goal of all this?

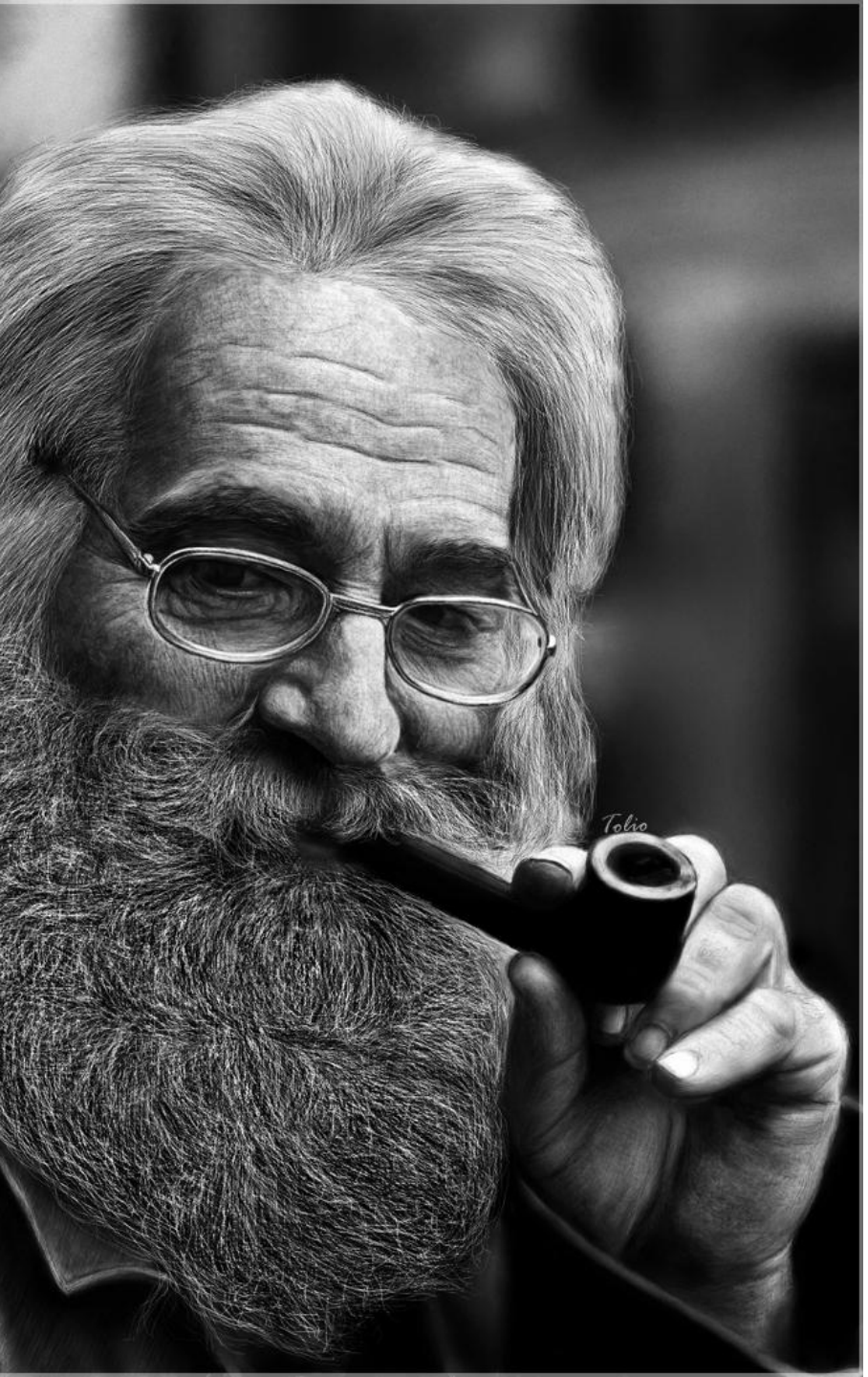
independent of the outside world

An Agnostic Domain allows you to focus on your complex problem

Paranoia?


FOR
MICROSERVICES
YOU RARELY NEED
DEPENDENC
Y

>



Elders Wisdom

Worths defending?

Don't implement complex logic

on **foreign** data structures.

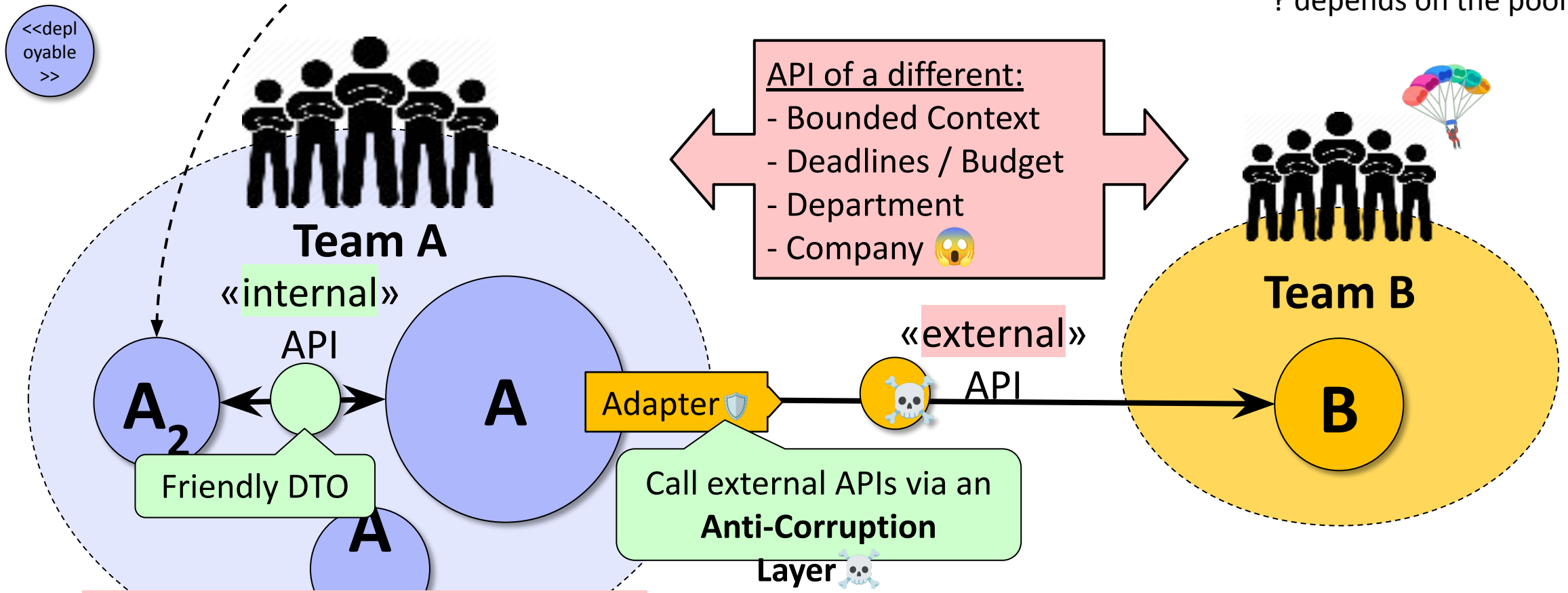
Large gap from my system

Then Dependency-Invert that call via an Adapter

No need for Dependency Inversion in:

Small Satellite System

Developer fungibility ↻: 💖
+ increase bus factor
+ there's work for everyone
- cognitive load on dark parts
? depends on the pool size



Risk ⚠️ : *Internal* becomes *External* if:
- Another team calls A2 directly
- The team grows and splits 💔

Adapt if gap vs the called interface is large 🤔

⚠️ "Jeff Bezos API Mandate" says All service interfaces [...] should be externalizable ... -- Amazon CTO, in 2002

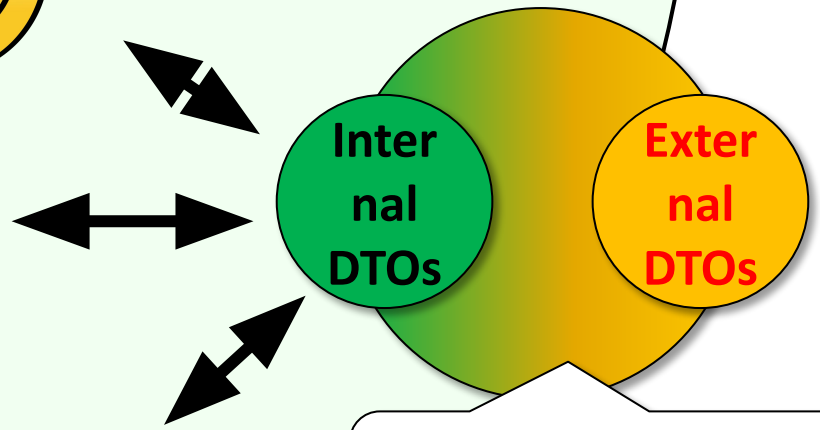
No need for Dependency Inversion in:

Microservice Ecosystem

Happy Internal Systems

Service Proxy

an Adapter system encapsulating interaction with an outside system



Lacks an internal Domain.

aka *Open-Host Service (DDD)*

Unfriendly System

SAP
SalesForce
BPM
AS400,
Siebel
zOS
COBOL,
Fortran
...

Microservice or Library?

↓

Microservice vs Shared Library

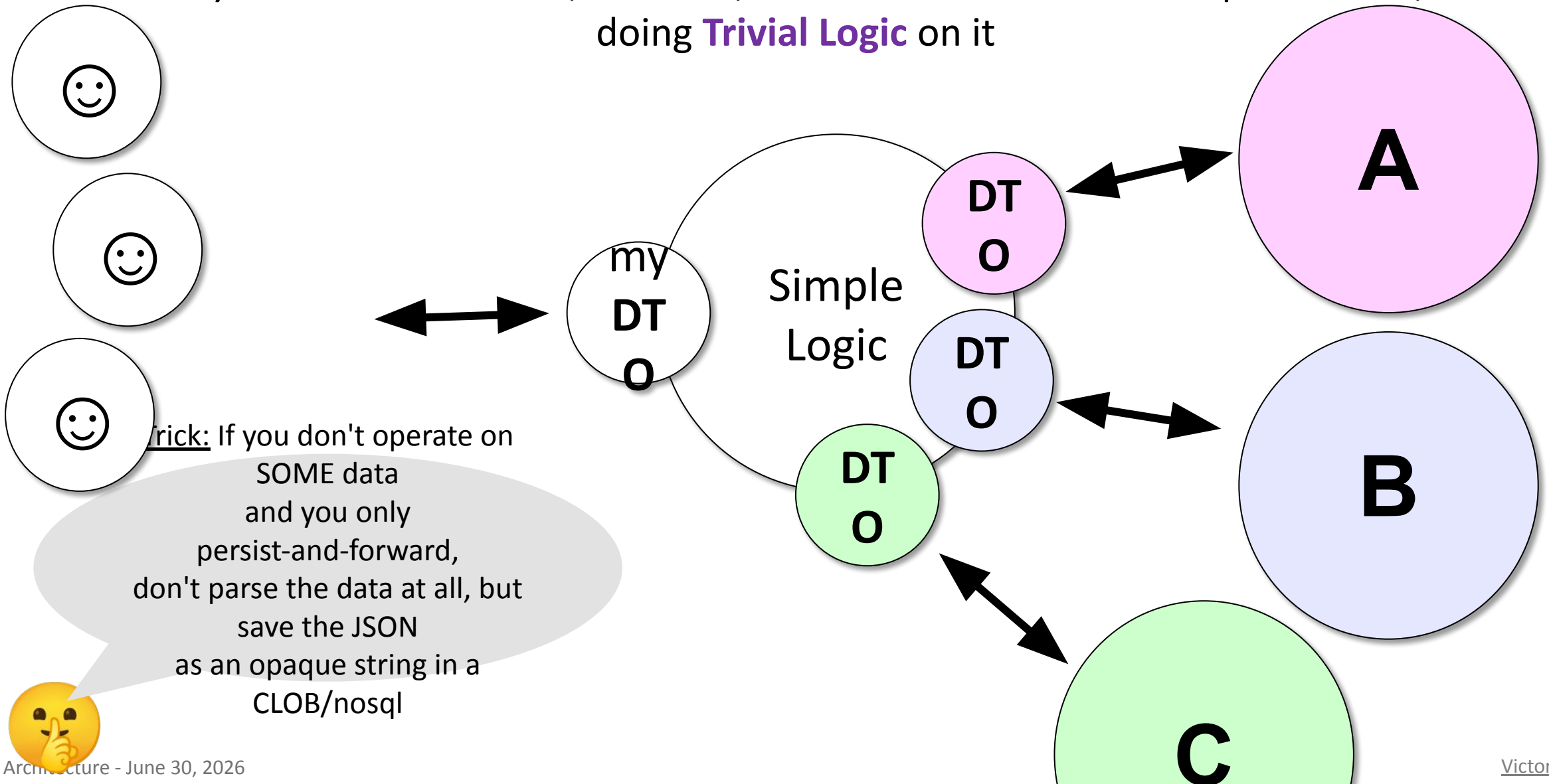
eg: sap-proxy-service vs sap-client-v1.32.jar

+ **centralized**:
  + security: access control, 3rd party authentication
  + cache
  + resilience: throttle, rate limit, retries...
  + metrics & alarms
  + can crash alone (eg OOME); can be a SPOF
  + change contention: clients don't have to upgrade
- **network call** overhead + risk
- **cost** develop + operate
+ clear **ownership** by a dedicated team
+ can make sure external crap 💩 doesn't leak in any of my systems
+ **tech stack** freedom: called from C#, no transitive lib vers collisions

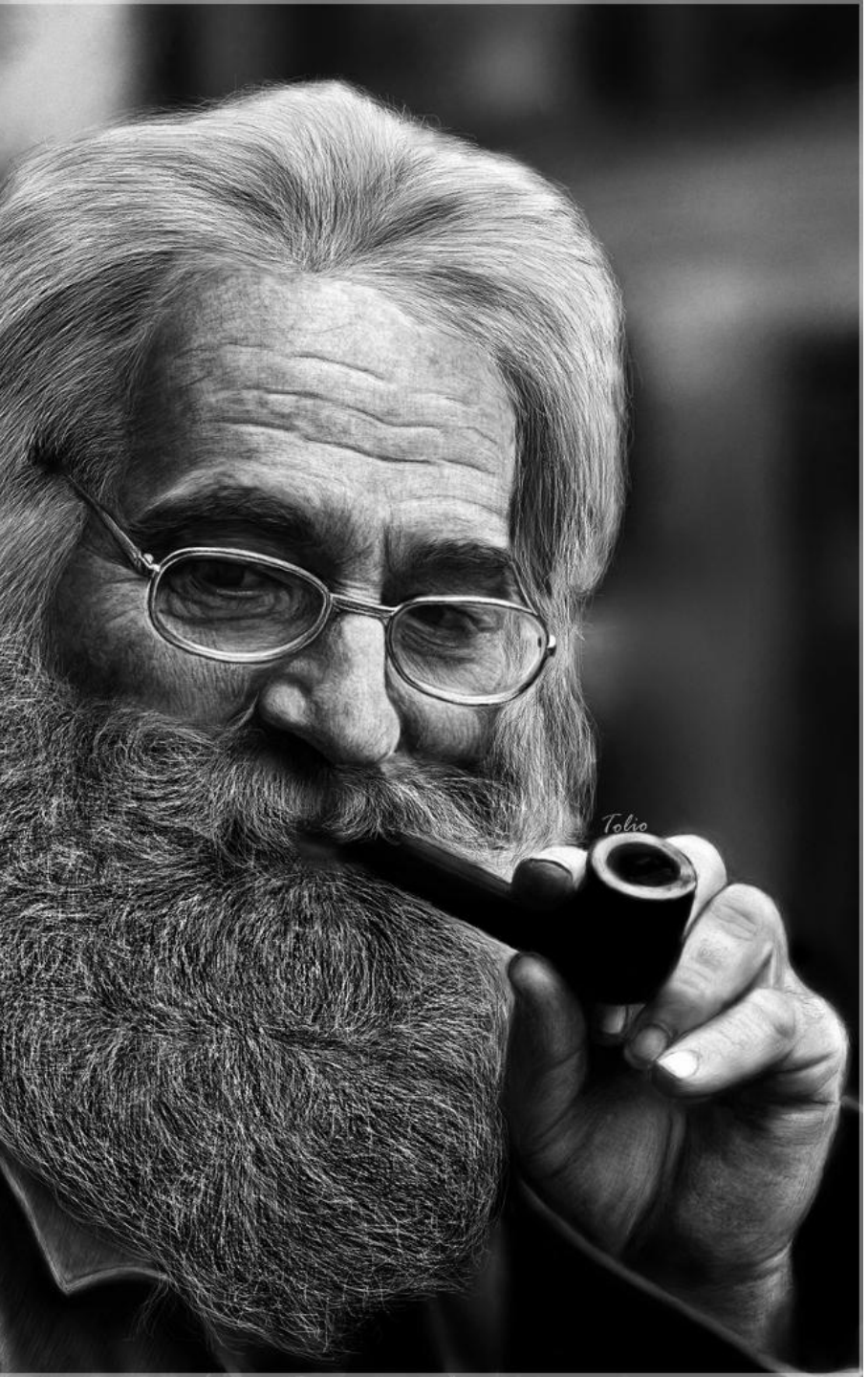
No need for Dependency Inversion in:

Aggregator

a system that combines, enriches, transforms data from multiple sources,
doing **Trivial Logic** on it



Trick: If you don't operate on SOME data and you only persist-and-forward, don't parse the data at all, but save the JSON as an opaque string in a CLOB/nosql



Elders Wisdom

Don't implement complex logic

on **foreign** data structures

agnostic domain

Separate **Your Core Complexity**

from **External APIs** & **LIBs**

infrastructure

PRO

Marshalling Value Objects

== Pragmatic Variation ==

Consider if the gap between your Domain and the consumed API is small

Manually hand-craft Value Objects whose:
- private fields are mapped ⇔ external JSON (reflection)
- public methods expose a nice API used by the Domain

You can make these objects:

❤️ only expose used attributes

❤️ can host behavior (OOP)

❤️ speak your domain language

❤️ immutable (no setters)

❤️ can be validated eg with @NotNull... + @Validated on a Feign Client

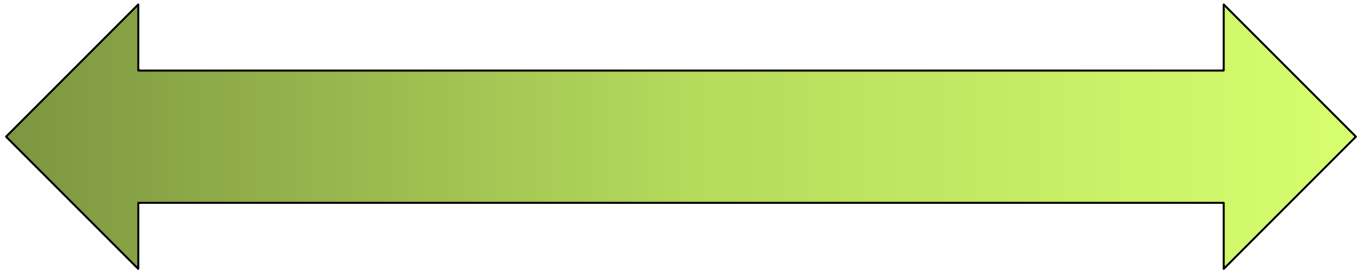
Compared to classic Adapter/VO:
❤️ less data structures, no Adapter
😬 No OpenAPI generated objects 😬 Requires contract tests

PRO

Degrees of Protection

Domain vs External APIs

Less Code,
More Pragmatic

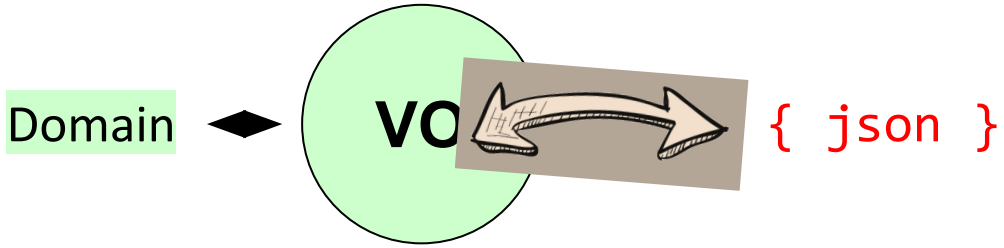


Cleaner &
Easier to Test

No Protection
(DTOs are free to enter Domain)

Marshalling Value Objects
Unmarshalling in VO 👇

Enforced Dependency Inversion
(via Adapters outside an 'Agnostic Domain')



Evolutionary Architecture

Simplify Design

Dummy Controller Anti-Pattern

```
@RestController
public class CustomerApi {
  private CustomerService customerService;
  @PostMapping("customers")
  public CustomerDto create(@RequestBody CustomerDto dto) {
    return customerService.create(dto);
  }
  // all other methods like this
}
```

"Middle Man" code smell

(aka boilerplate method)

If controllers contain roughly no code at all,

Merge the Controller with the 1st Service 🧠 it calls

🛑 **UNLESS** the same feature is exposed on ≥ 2 channels: REST+gRPC+SOAP...

👍 Extract noisy Swagger annotations to an implemented interface

👍 Convert exceptions to HTTP in a global handler (@RestControllerAdvice)

I hope some of you disagreed.

Honest disagreement is often a good sign of progress.

- Mahatma Gandhi

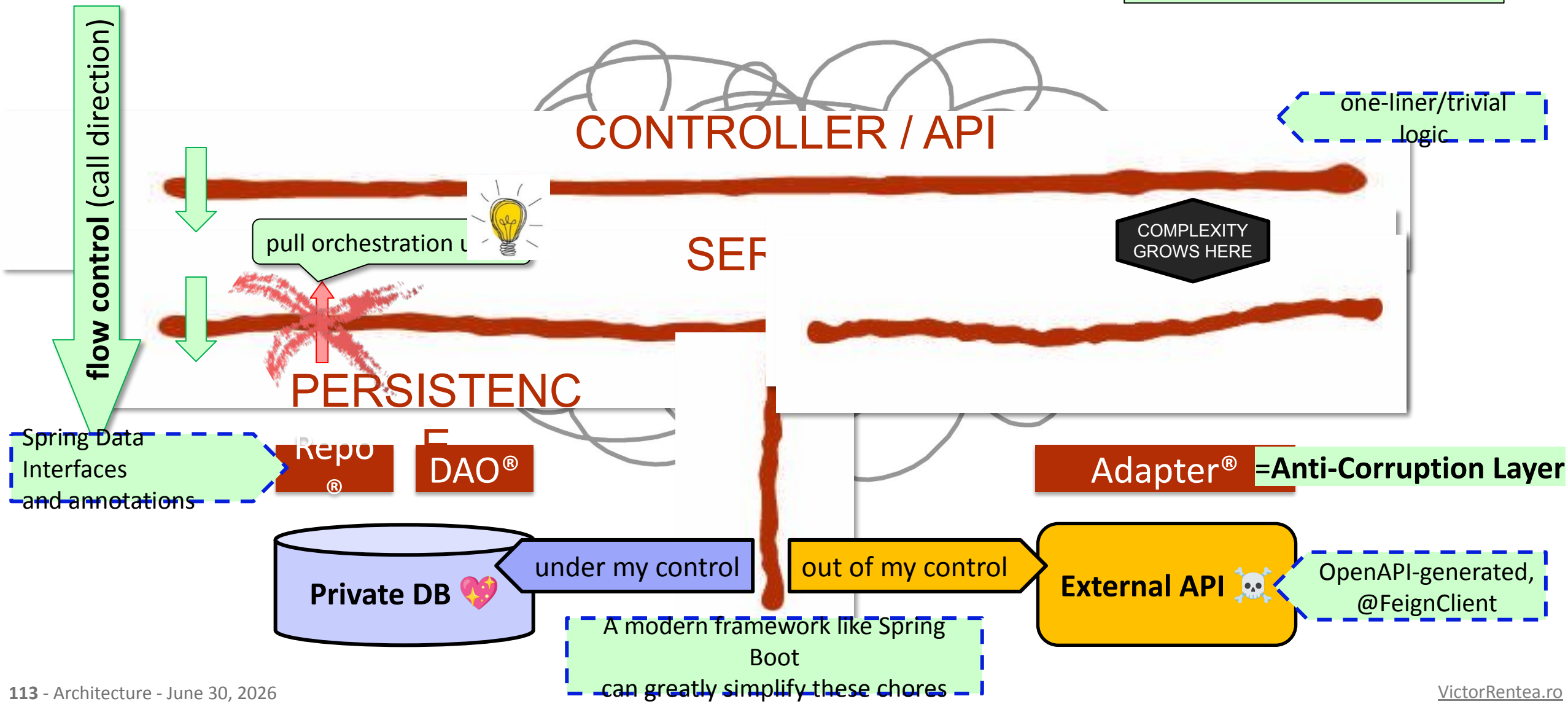
Simplify Design

/

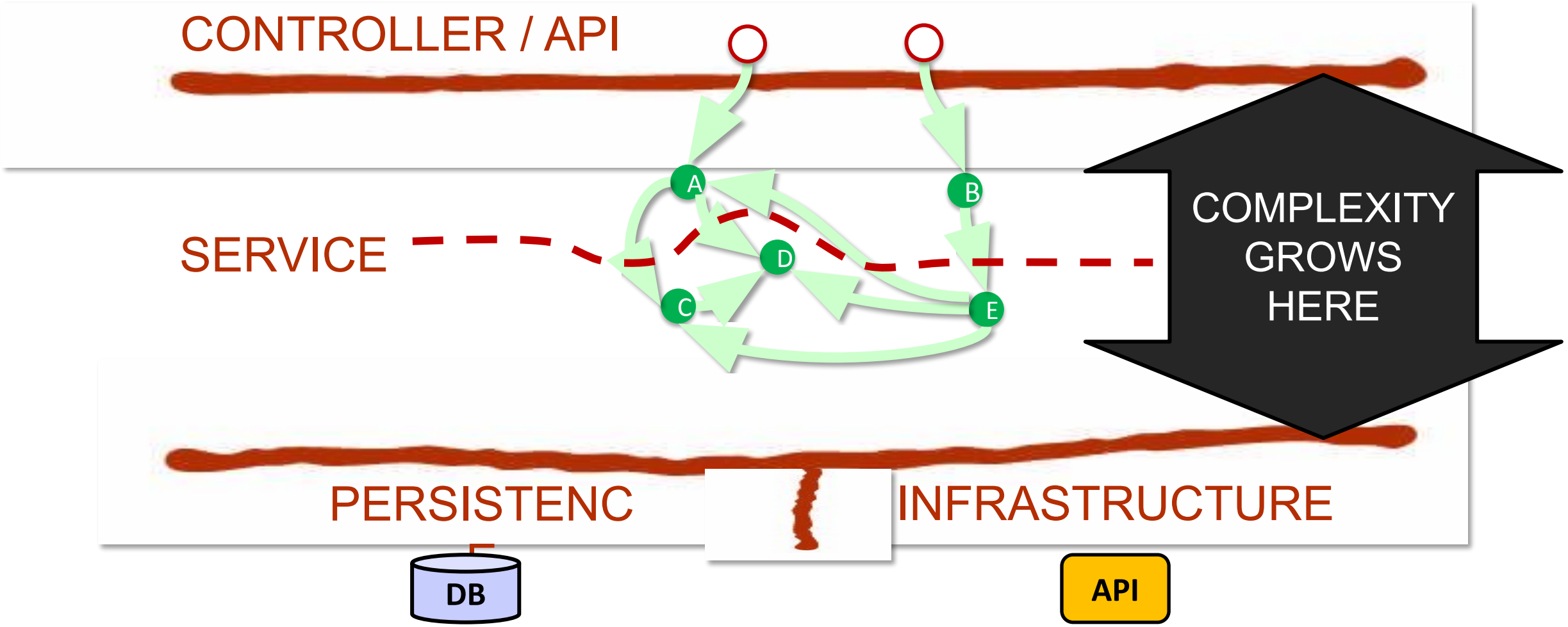
Expand Design

Layered Architecture

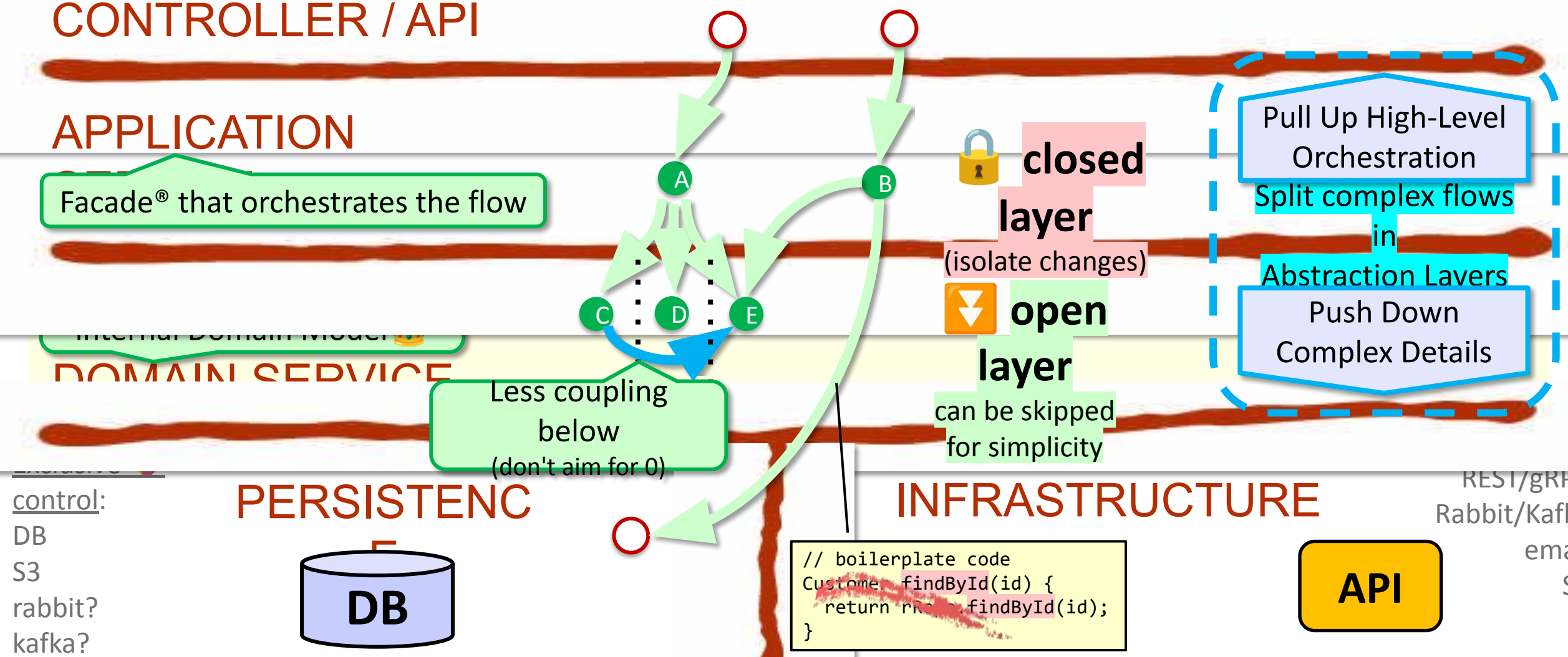
3 layers could be enough for a Small-Medium app
eg. < 3 dev x 1 year



Complexity Accumulates in Service Layer



Introduce a new Layer of Abstraction



Application Service Roles

- **Convert** DTOs ⇔ Entities
  - inline, via DTO constructors/methods or via [Auto-]Mappers
- **Validate** inputs
  - @NotNull, `if`s, or self-guarding domain objects 💪
- **Orchestrate** the workflow of a use-case = "**Facade**" Design Pattern
  - Push complexity in lower-level components, to keep the Facade thin! 👍
- Manage **Transactions** when required
  - Prefer finer-grained transactions in high Transaction Per Second (TPS) systems
- [pragmatic] Expose endpoints
  - eg. Expose REST API (+authorization)

Logical Architectural Styles

Layers

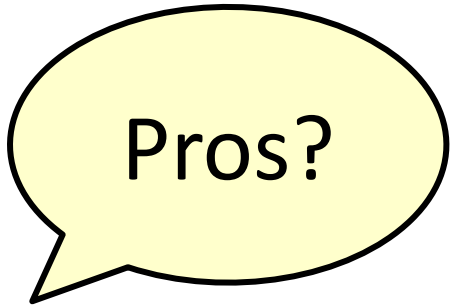
Hexagonal / Onion

VSA ("Anti-Layers")

Modular Monolith

Layered Architecture

aka The Junior Architecture



☺️ Very easy to understand & follow

controller

```
@RestController // Spring
class UserController {
```

service

```
@Service // Spring
class UserService {
```

repository

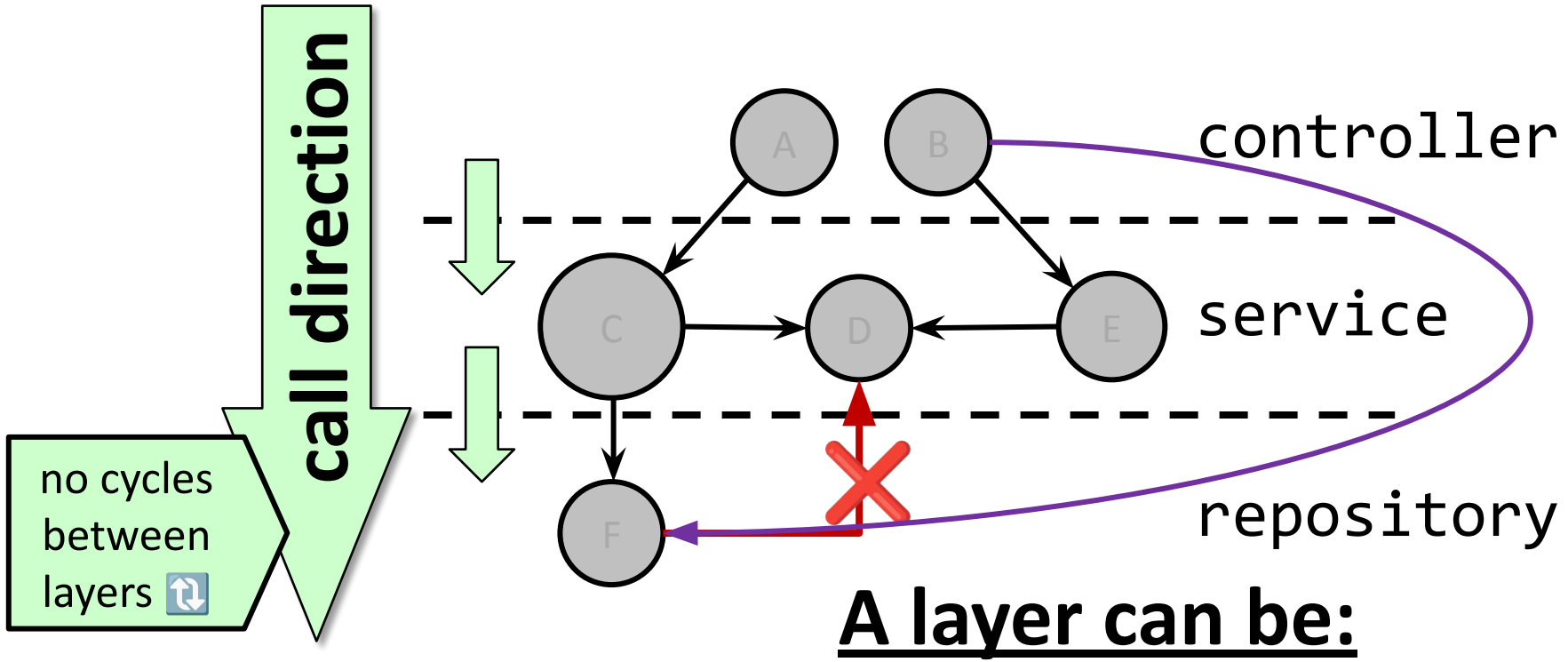
```
@Repository // Spring
class UserRepository {
```

Layered Architecture

```
controller
service
repository
```

- **Easy to follow and enforce**
- **Separation of Concerns ≈ Technologies**
  - REST <vs> SQL <vs> WSDL <vs> gRPC <vs> Kafka ...
- **Testability**
  - Service vs Repo = natural mocking seam
- **Decoupling**
  - A layer can only call layers below = no cycles between layers
  - A closed 🔒 layer hides any layers below
- **Split by Layer of Abstraction** (SLAb)
  - **High-Level** policy vs **Low-Level** details (for complex flows)

Layered Architecture



During "SOA times":
- customer-frontend.ear (JSP/F)
- customer-logic.ear (EJB)
- customer-persistence.ear (JDBC/JPA)

❌ sync changes, tight coupling

A layer can be:

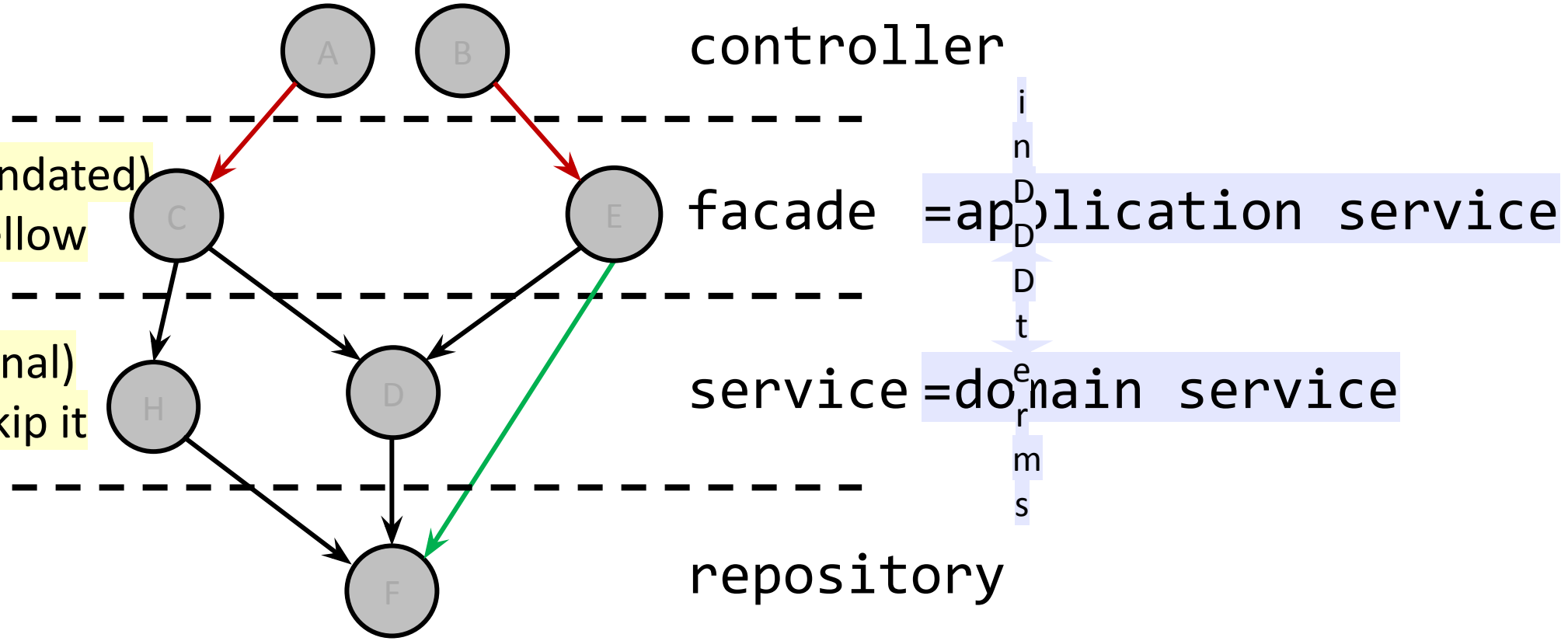
CLOSED Layer = mandatory
eg Controller must call via Service to use Repo
☺️ encapsulation ☺️ homogenous ☺️
reuse++

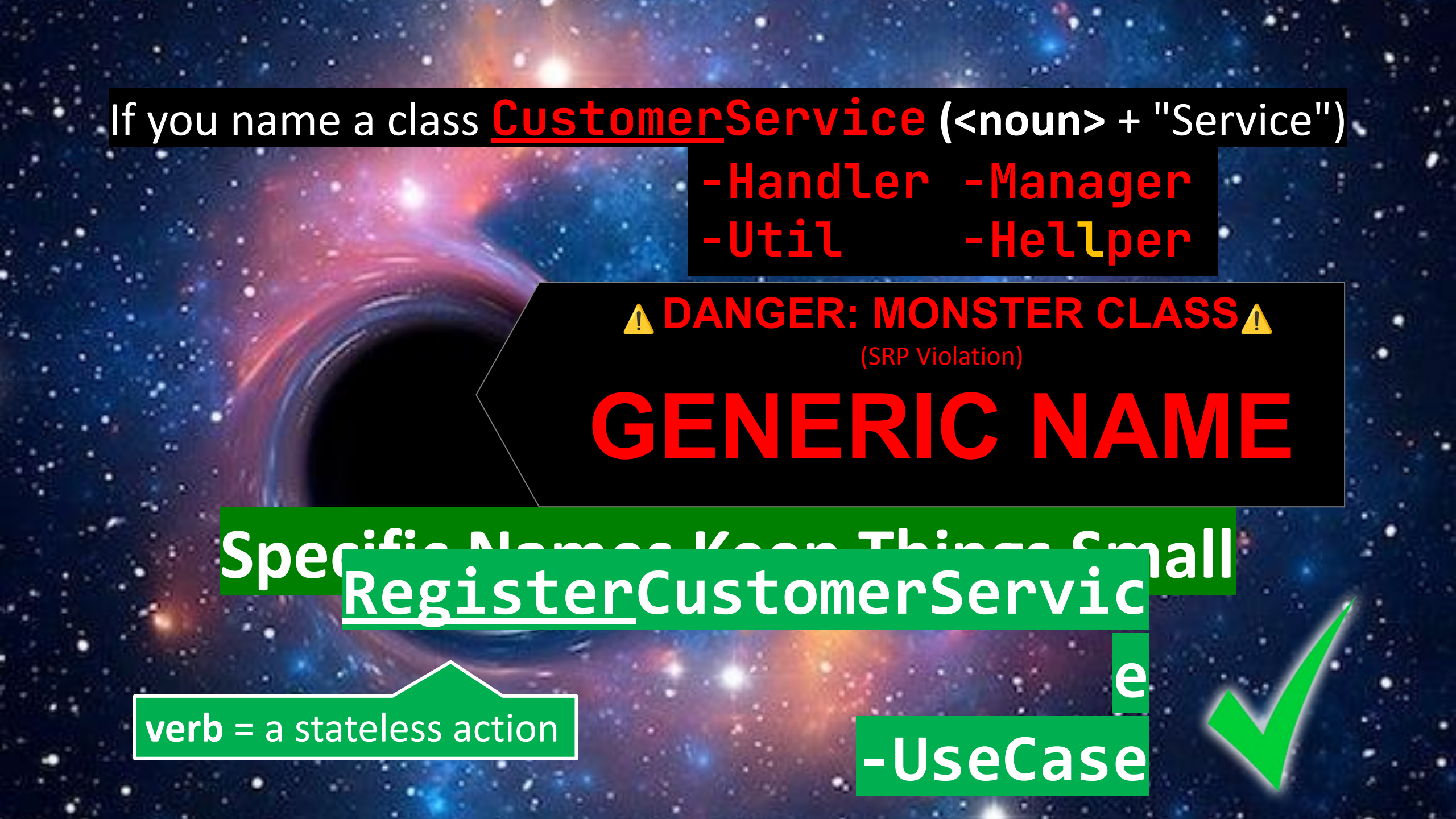
OPEN Layer = optional
eg Controller can directly use Repo
☺️ pragmatic (less stupid code)

Layered Architecture

CLOSED Layer (Mandated)
Hides the layers bellow

OPEN Layer (Optional)
Simple flows can skip it




If you name a class CustomerService (<noun> + "Service")
-Handler -Manager
-Util -Hellper
⚠ DANGER: MONSTER CLASS ⚠
(SRP Violation)
GENERIC NAME
Specific Names Keep Things Small
RegisterCustomerServic
e
-UseCase
verb = a stateless action

Hey ChatGPT, draw me the CTO of a multinational company washing his socks in the hotel sink.

Caller (eg Controller)

```
class OrderService {
  public placeOrder() {
    > 20% of lines
    (including its private methods)
  }
  public getOrder() {15 lines}
  public cancelOrder() {20 lines}..
} > 300 lines
```

Side-to-Side Split
(same level of abstraction)

Caller

```
class PlaceOrderService {
  public placeOrder() ...
} < 150 lines
```

```
class OrderService {
  public getOrder()
  public cancelOrder()
} < 200 lines
```

Alternative: split from start:
have 1 public method/class

class PlaceOrderUseCase

class GetOrderUseCase

class

too many tiny classes?

Caller

```
class PlaceOrderService (UseCase) {
  public placeOrder() ...
  = the only public method
  <reserve stock>
  <pay>
} > 300 lines
```

High/Low-Level Split
push down complexity;
orchestrate from above

class PlaceOrderService

Coupling

Cyclomatic

Cyclomatic

Coupling

class ReserveStockService

class PaymentAuthorizer

Identify a clear responsibility = hard.
Avoid meaningless suffixes: *-Util, -Hellper...*

Separation By Layers of Abstraction (SLAb)

"Application Service" (DDD)
acting as a "Facade" (GoF)

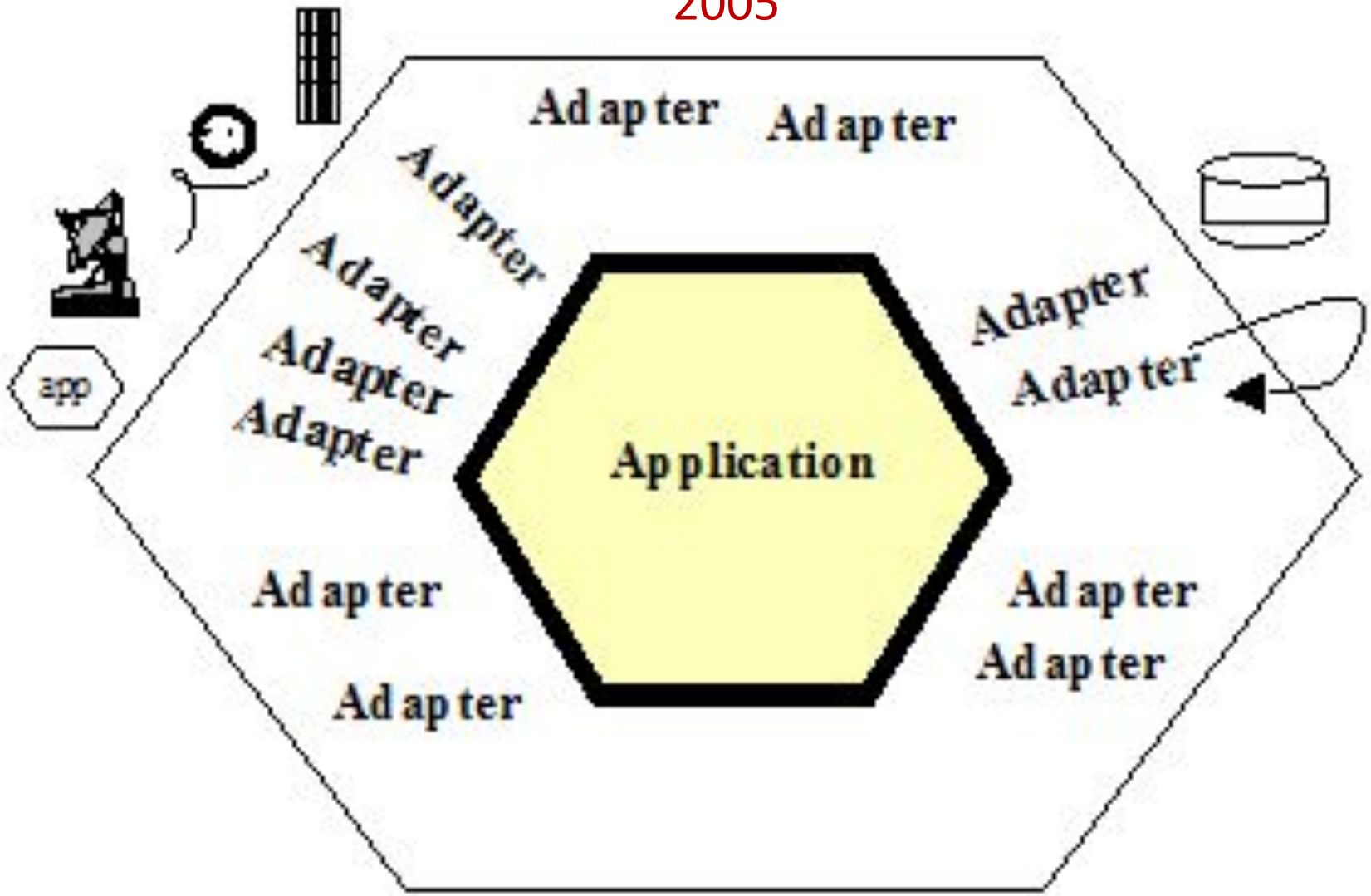
Separate **High-Level Orchestration**

from **Low-Level Details**

a "Domain Service"
(DDD) if possible

Hexagonal Architecture

2005



https://alistair.cockburn.us/hexagonal-architecture

Hexagonal Architecture by Julien Topcu (dogmatic form)

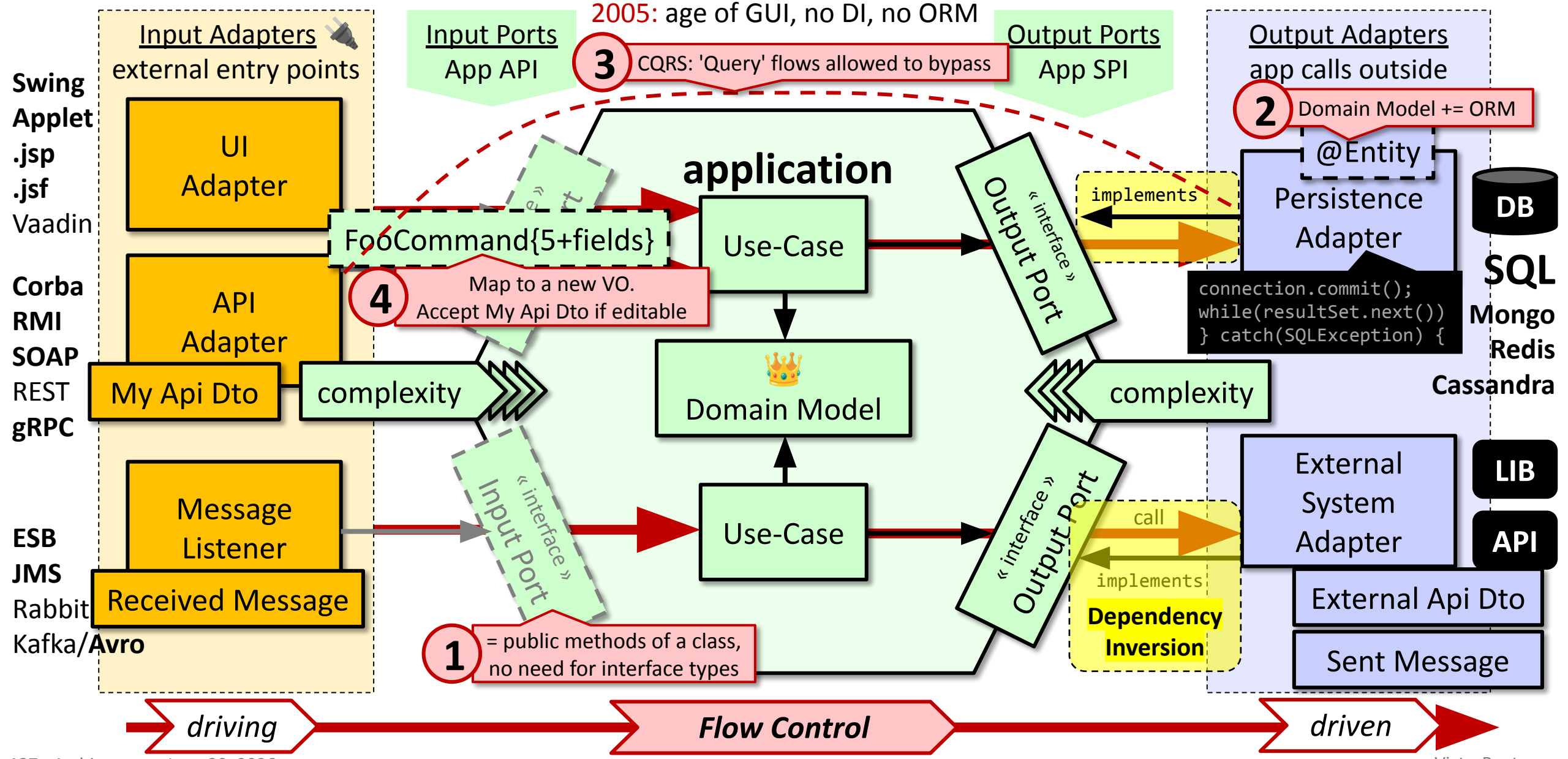
(Alistair loved this shape)

https://alistair.cockburn.us/hexagonal-architecture/

Hexagonal / Ports-Adapters Architecture

? Simplifications for common>aggressive

2005: age of GUI, no DI, no ORM

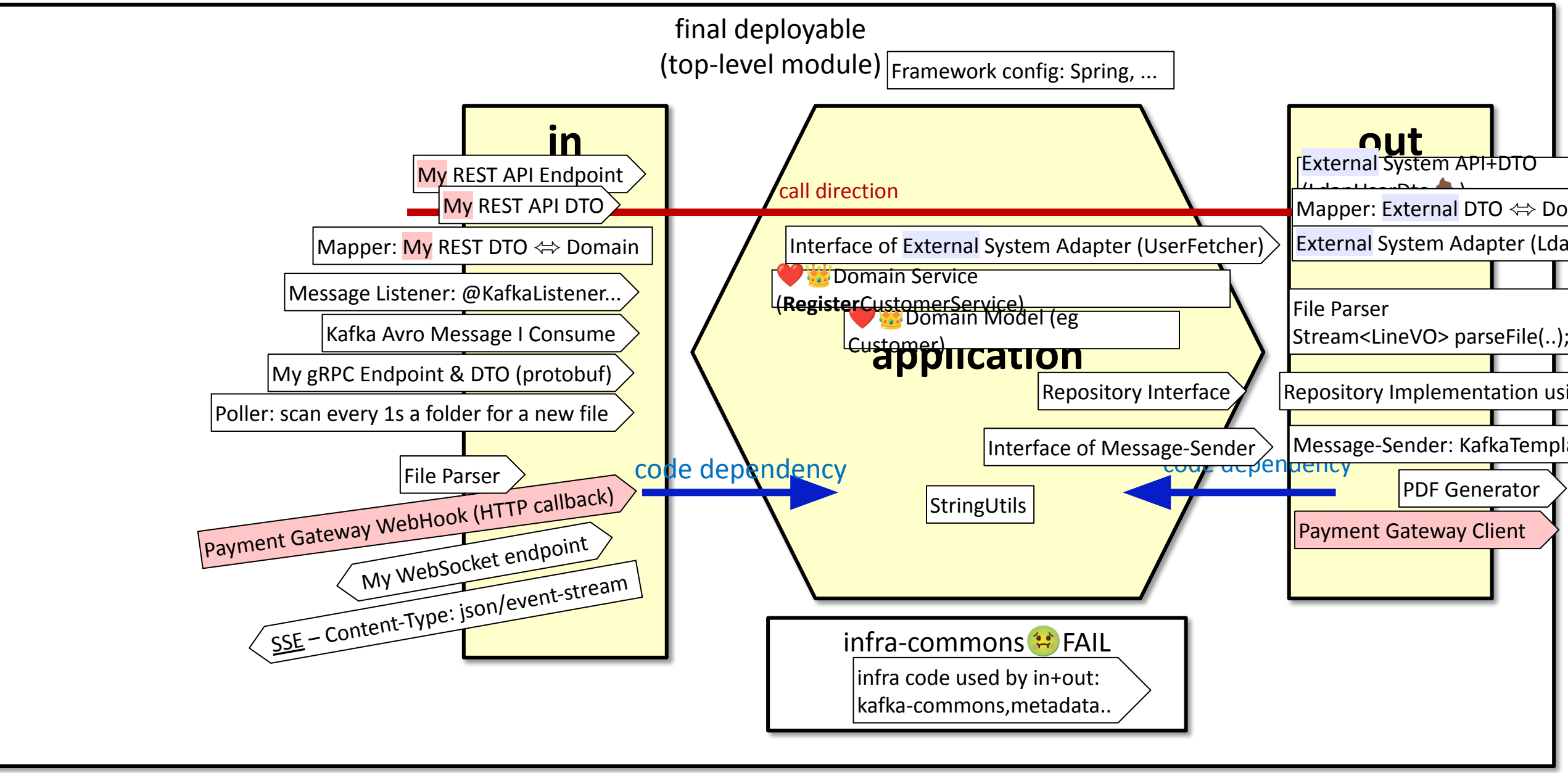


Hexagonal Tough Questions/Debates:

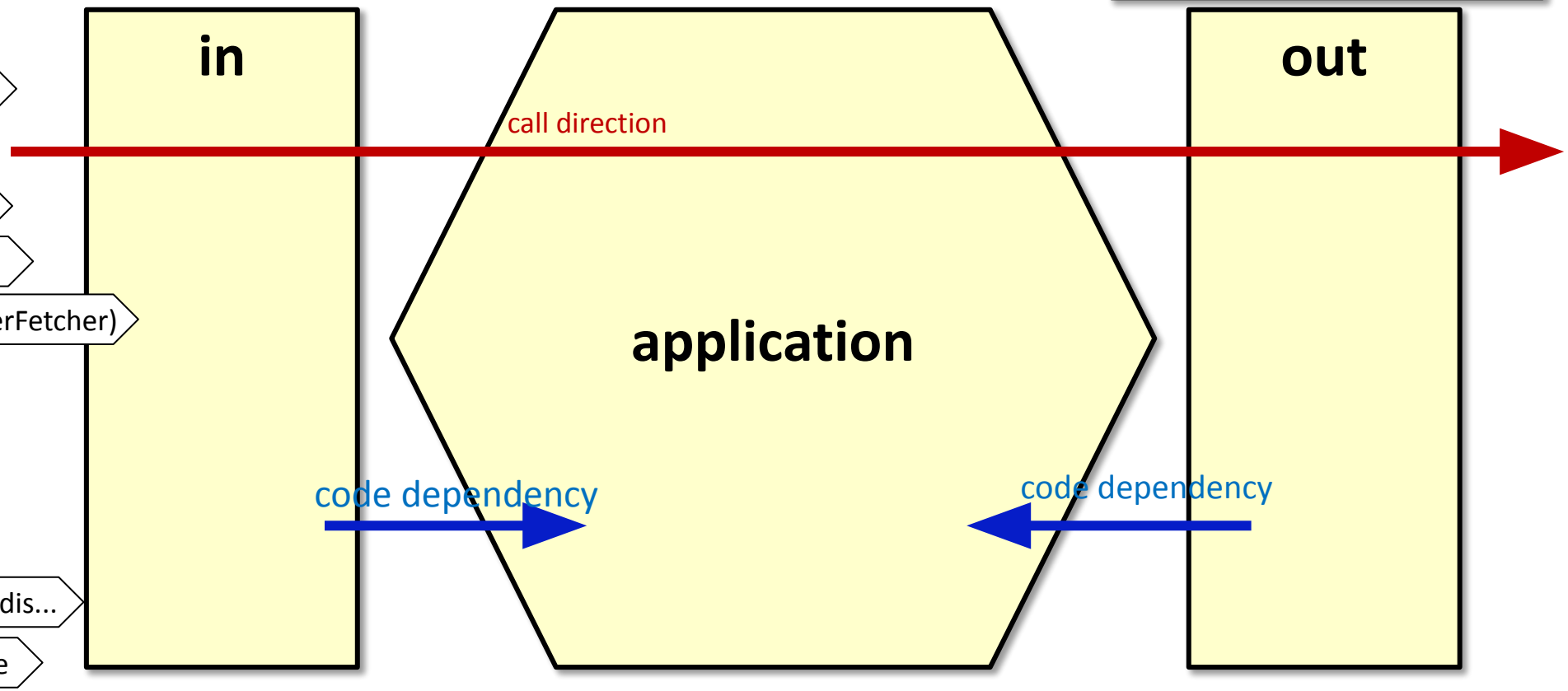
- A remote API call consists of a sequence of 3 calls. Orchestrate them from Adapter or from App?
- How to pass metadata (correlationId, ..) from an input adapter (REST) to output adapter (REST)
- Relax vs API out adapter between 2 microservices in the same bounded context?
- Relax vs DB out adapter if DB is private & decent?
- Relax vs API in adapter (accept my rest api in app) if DTOs are friendly?
- Bypass core for read flows (CQRS) for performance & simplicity
-

TODO rename application to 'core' in drawings

Hexagonal Puzzle



Hexagonal Puzzle



Domain Model (eg Customer)

Domain Service (RegisterCustomerService)

My REST API Endpoint

My REST API DTO

Mapper: My REST DTO ⇔ Domain

External System API+DTO (LdapUserDto)

Mapper: External DTO ⇔ Domain

Message-Sender: KafkaTemplate.send(...

External System Adapter (LdapApiClient)

Interface of External System Adapter (UserFetcher)

Message Listener: @KafkaListener...

Kafka Avro Message I Consume

My gRPC Endpoint & DTO (protobuf)

Repository Interface

Repository Implementation using SQL, Redis...

Poller: scan every 1s a folder for a new file

Scheduler: start reports job Sun at 00:00

final deployable
(top-level module)

infra-commons

Interface of Message-Sender

File Parser

StringUtils

Payment Gateway Client

Payment Gateway WebHook (HTTP callback)

infra code used by in+out: Kafka commons,metadata..

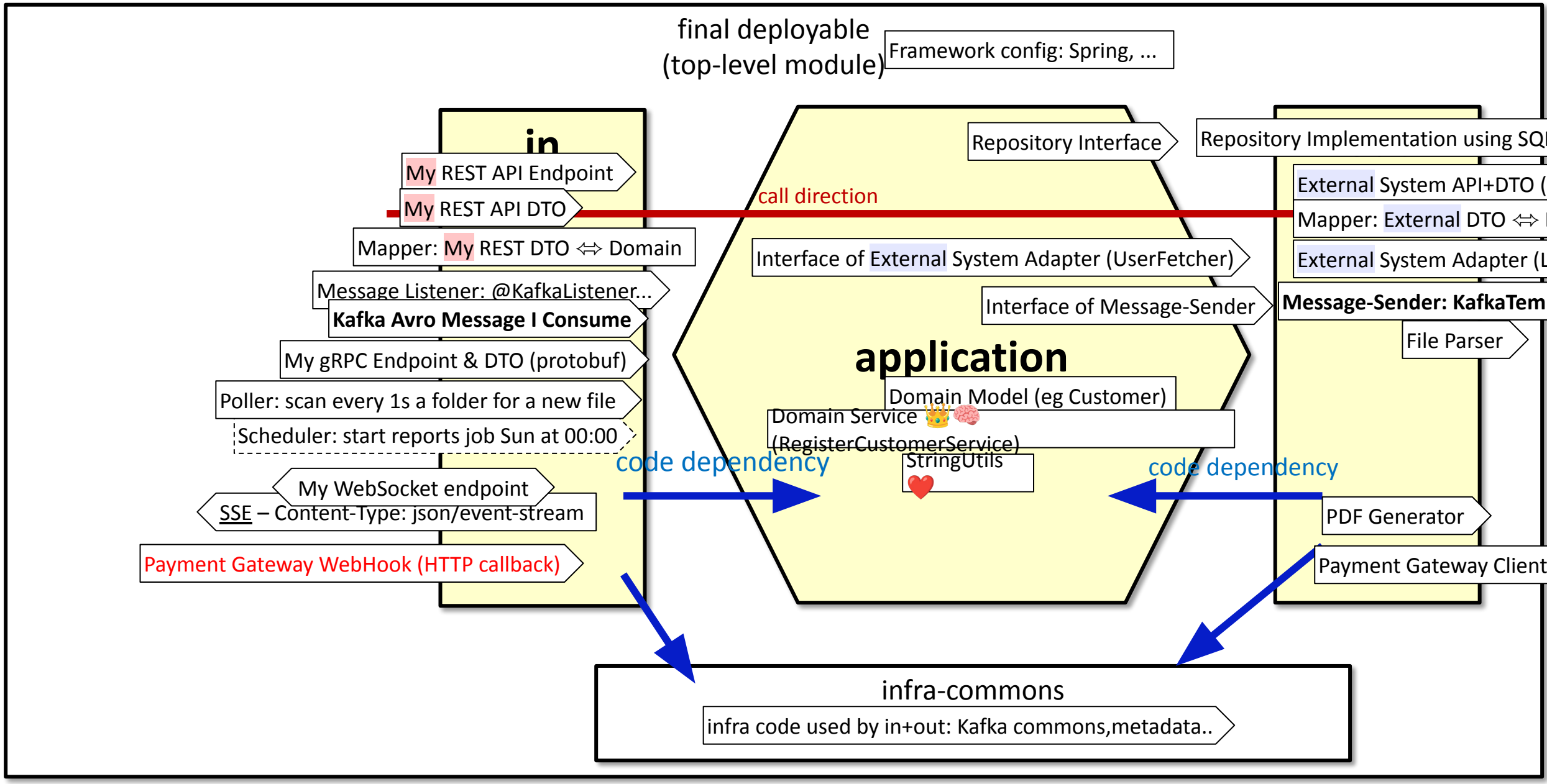
My WebSocket endpoint

SSE – Content-Type: json/event-stream

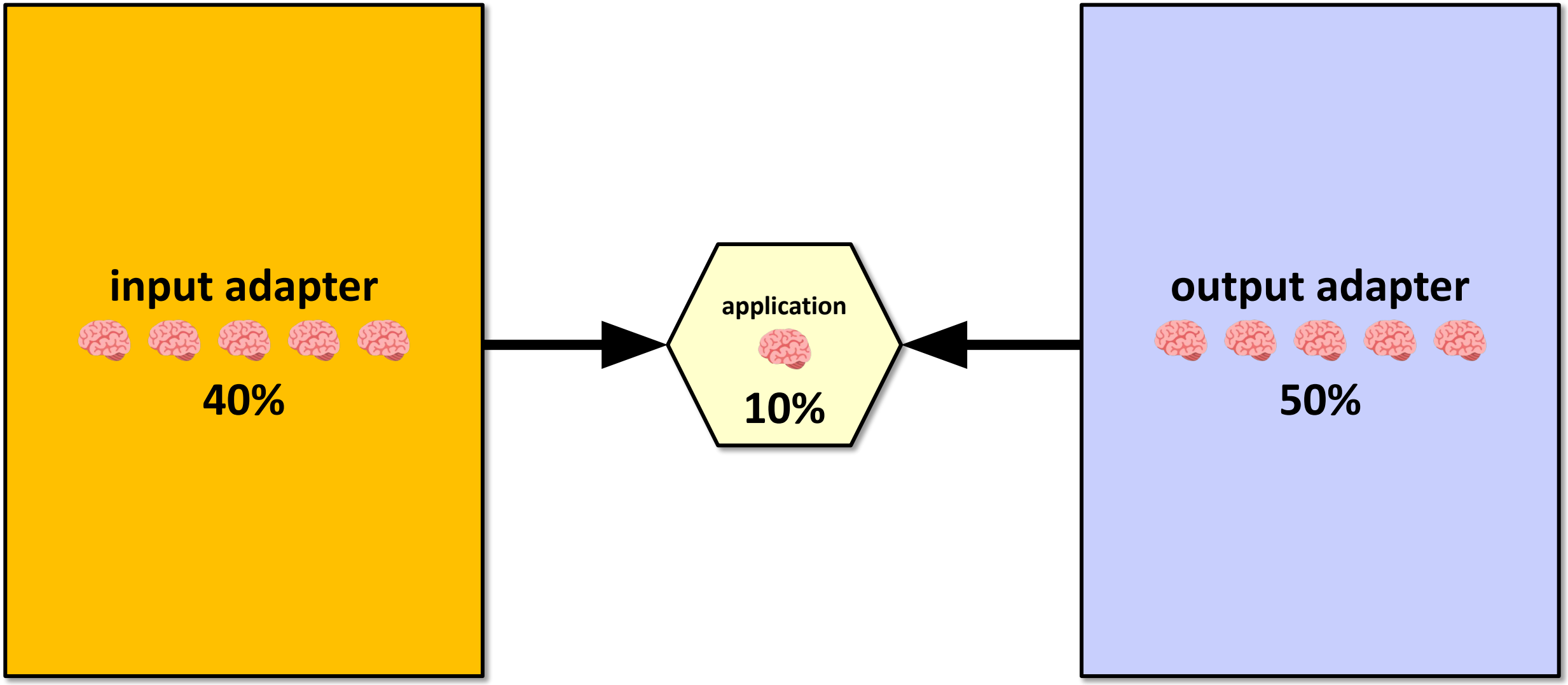
PDF Generator

Framework config: Spring, ...

Hexagonal Puzzle

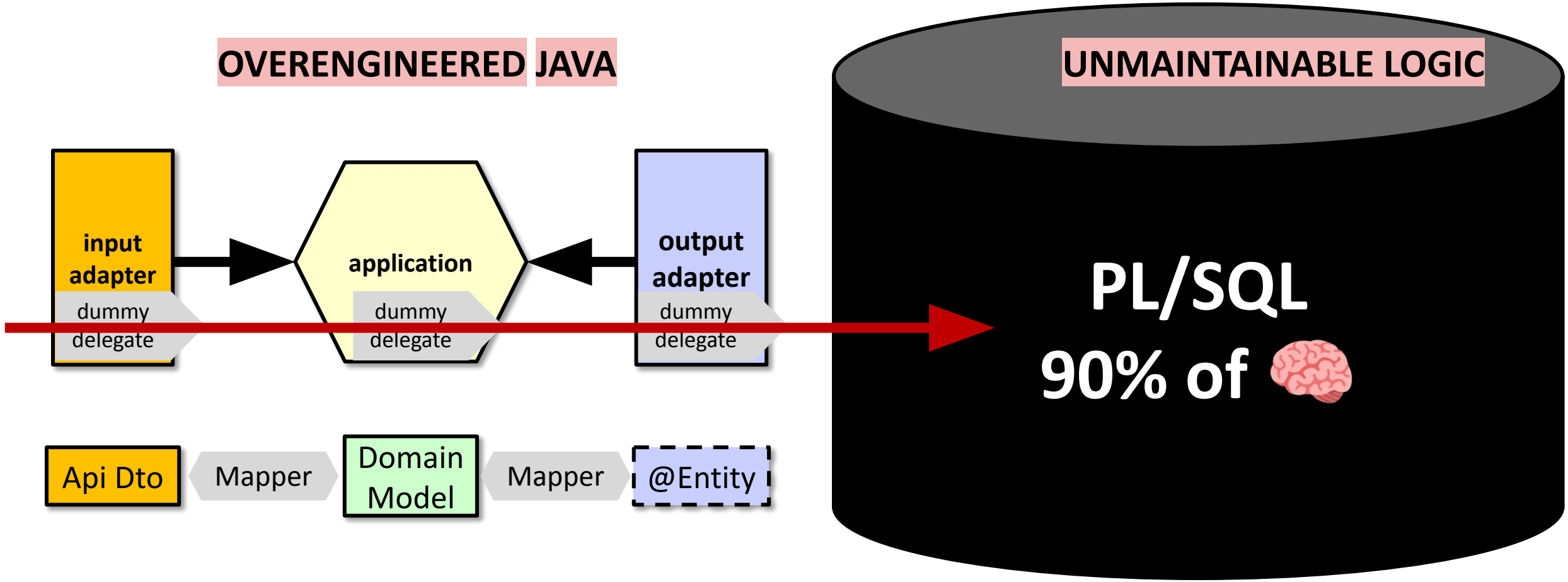


Hexagon Fail #1: Domain Logic outside 'app'



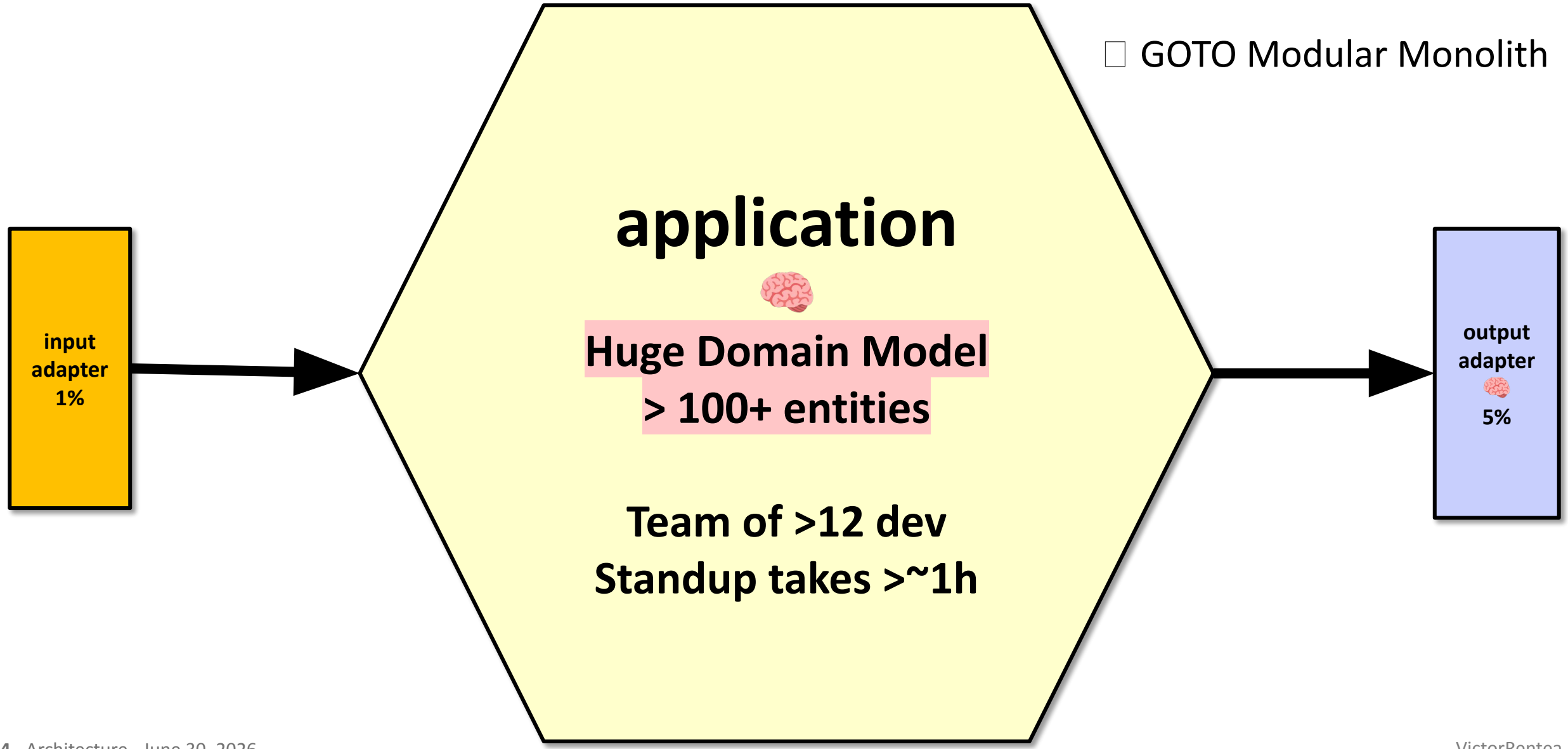
Legend: 🧠 = business logic

Hexagon Fail #2: All Logic in PL/SQL



Legend: 🧠 = business logic

Hexagon Fail #3: Growing into a BBoM



Layers vs Hexagonal

```
com.org.app
.controller
.service
.repository
.entity
.ldap
.dto
.mappers
+10 more…😢
```

```
com.org.app
.in.rest.dto
        .mappers
   .kafka.«consumer»
.app.usecase
    .model
    .port(out)-interfaces
.out.db
    .ldap
    .kafka.«sender»
```

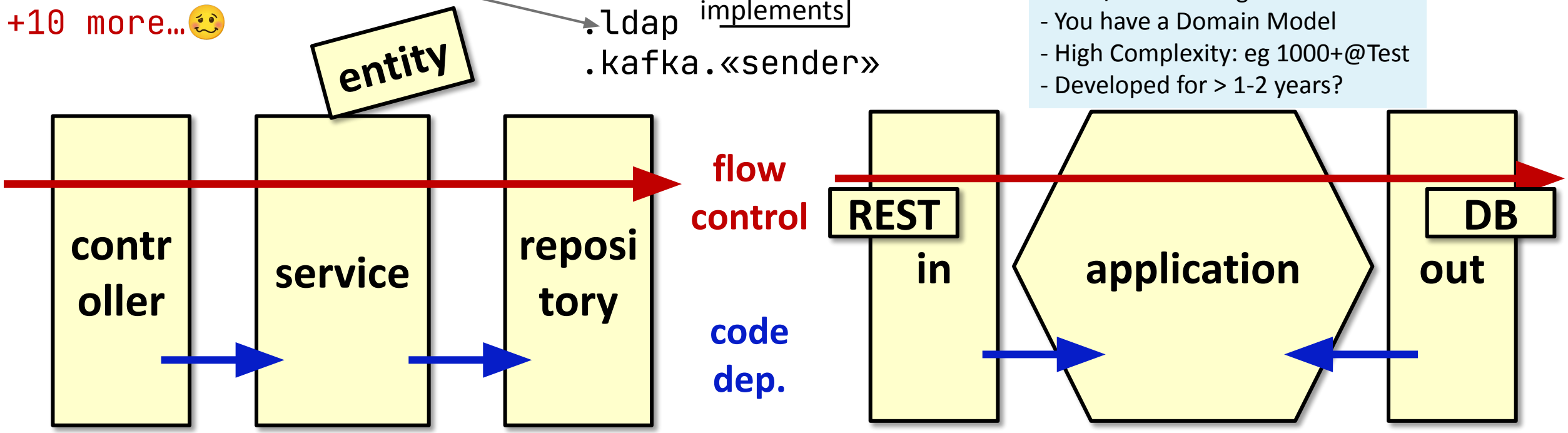
implements

⭐ **Highlights control flow**: in=>app=>out
+ Easy to **swap impl of an** in/out **adapter**

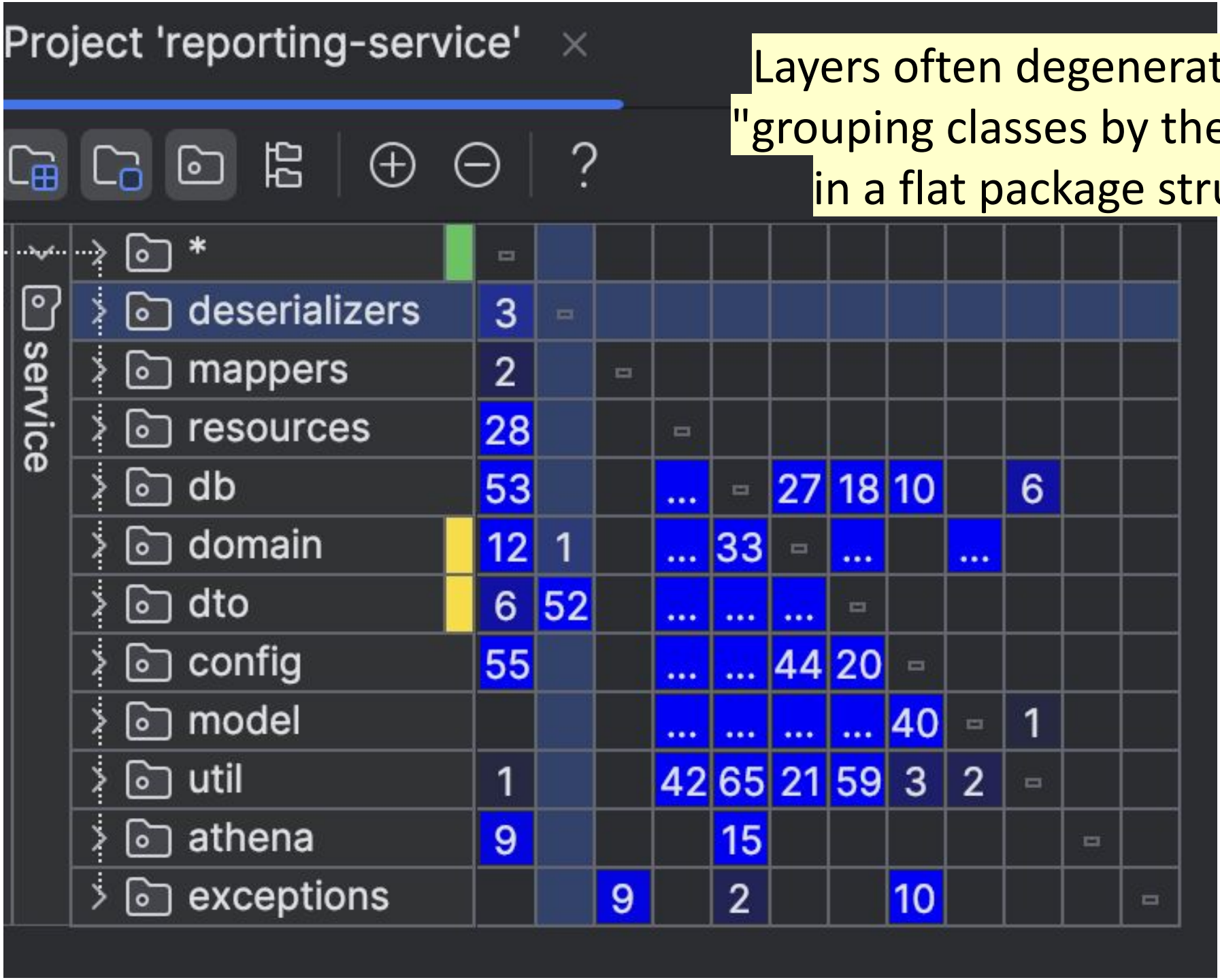
⭐ **Protects core complexity** ⭐
✅ Kept **agnostic** to external "corruption"
✅ **Cleaner to unit test core logic** 👑

⭐ **Deeper structure**: in.rest.dto
more space to spread classes

When/Should I migrate? □
- You have a Domain Model
- High Complexity: eg 1000+@Test
- Developed for > 1-2 years?



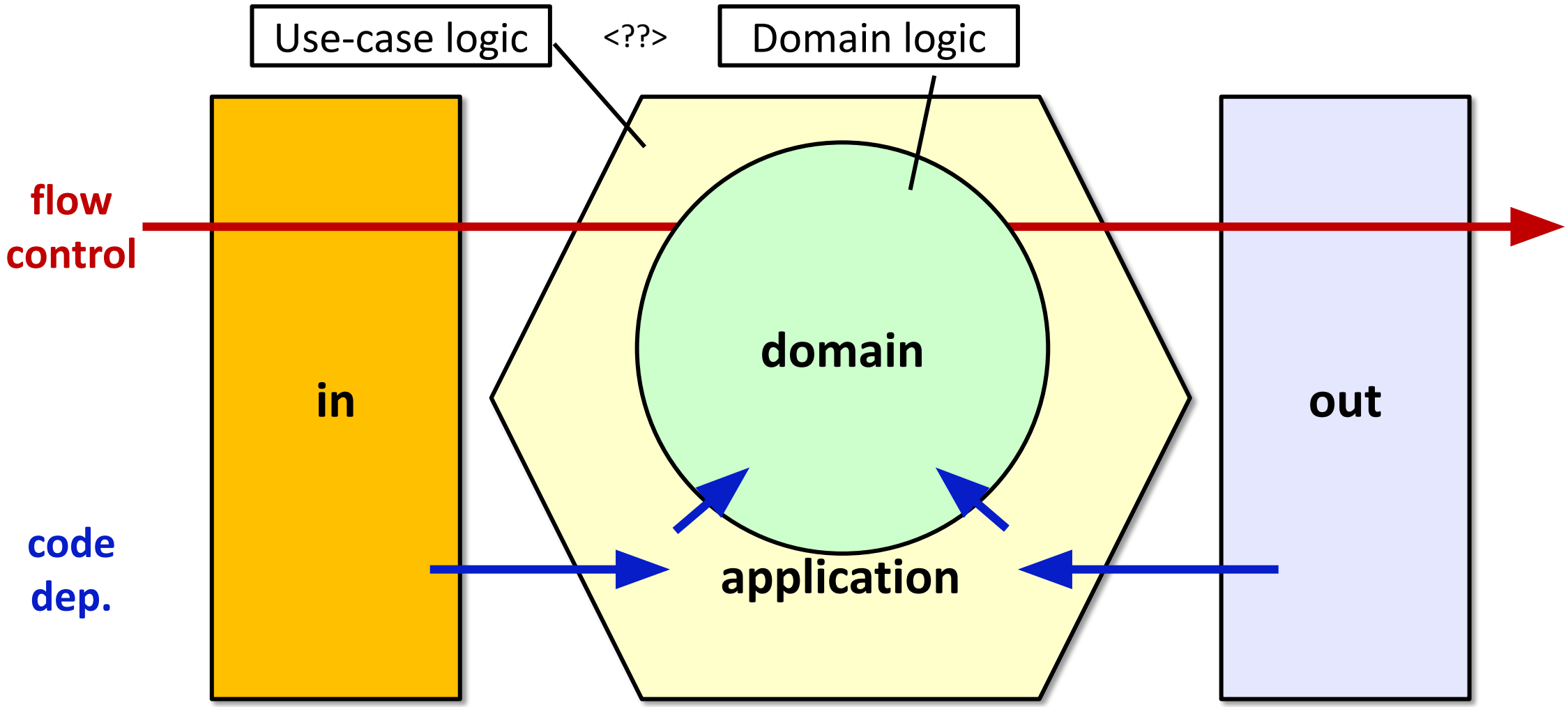
Layers often degenerates in just "grouping classes by their nature" in a flat package structure



tl;dr – Hexagonal is like Layers, but:

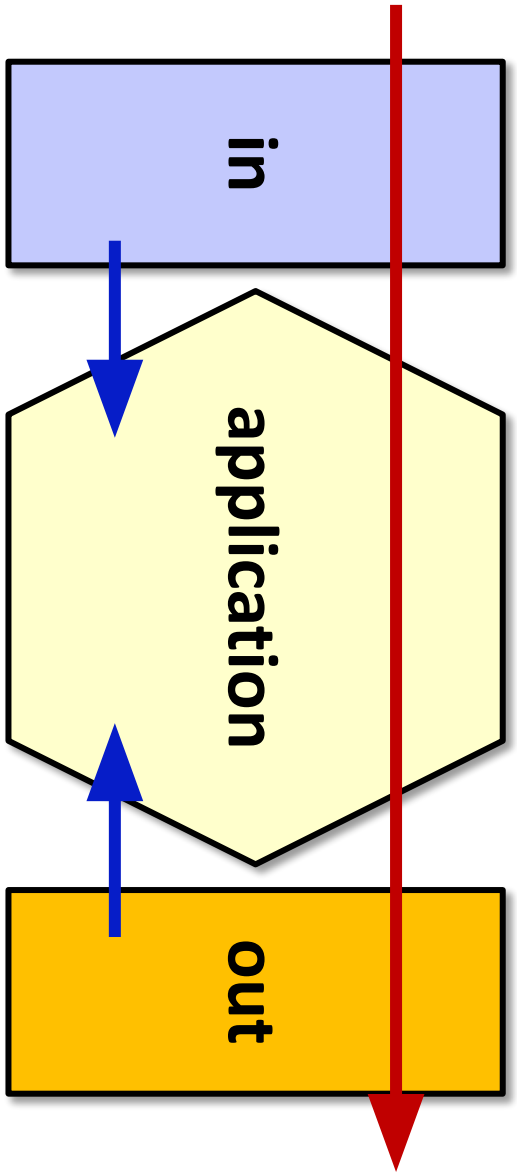
- But Out = **Repos + Adapters** calling outside,
- They **depend backwards** to the center (dependency inversion)
- The center is **agnostic** to the external world (DTOs, APIS, DB)

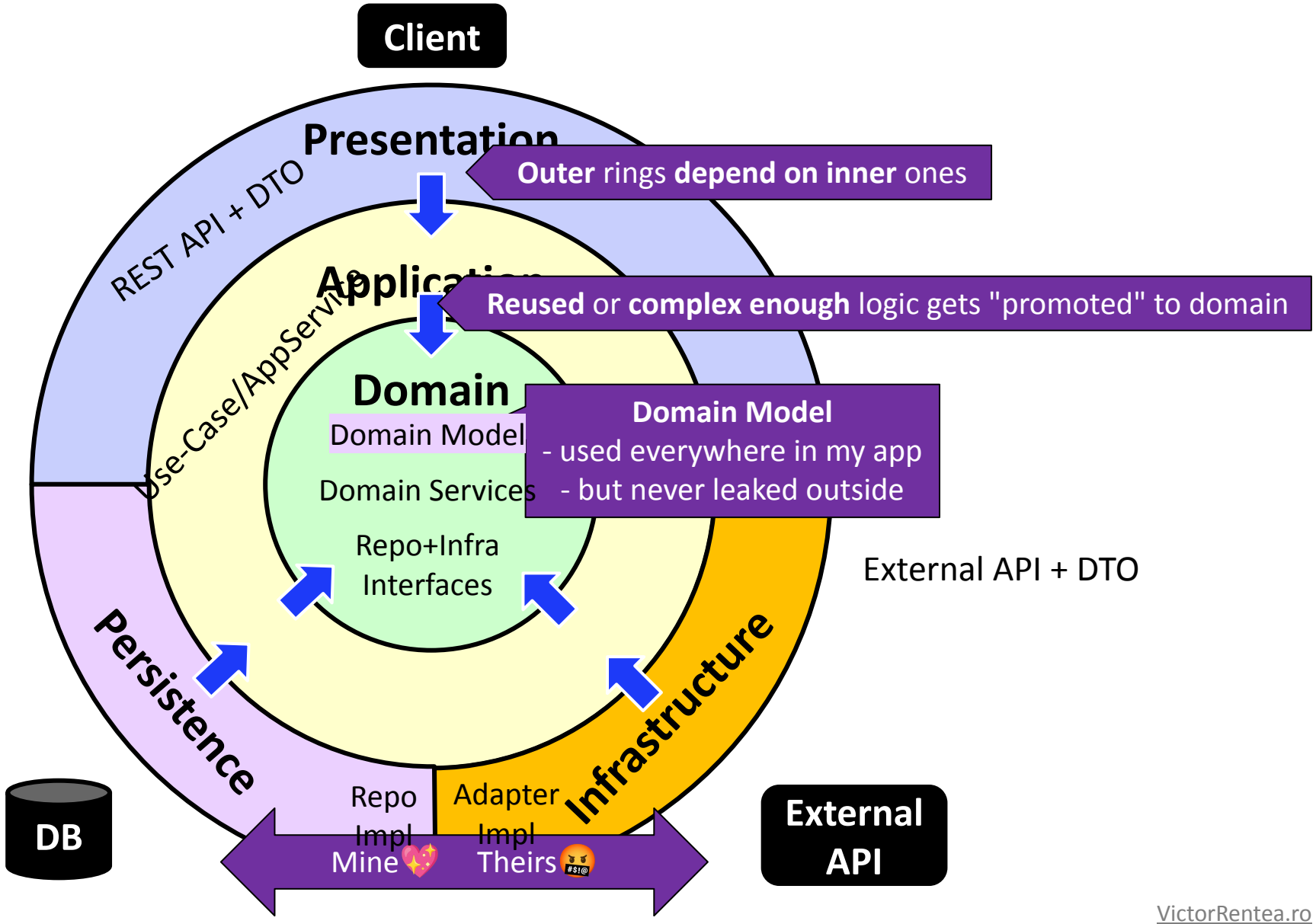
Hexagonal to Onion



Onion Architecture

by Jeffrey Palermo in 2008





Simplifying the Onion

Common Simplifications:

A) Merge Application Service with Rest API
If use-case exposed only **on 1 channel** = REST
Debate: what if exposing **gRPC**, **Avro**, **GraphQL**, Swagger-generated Dtos..

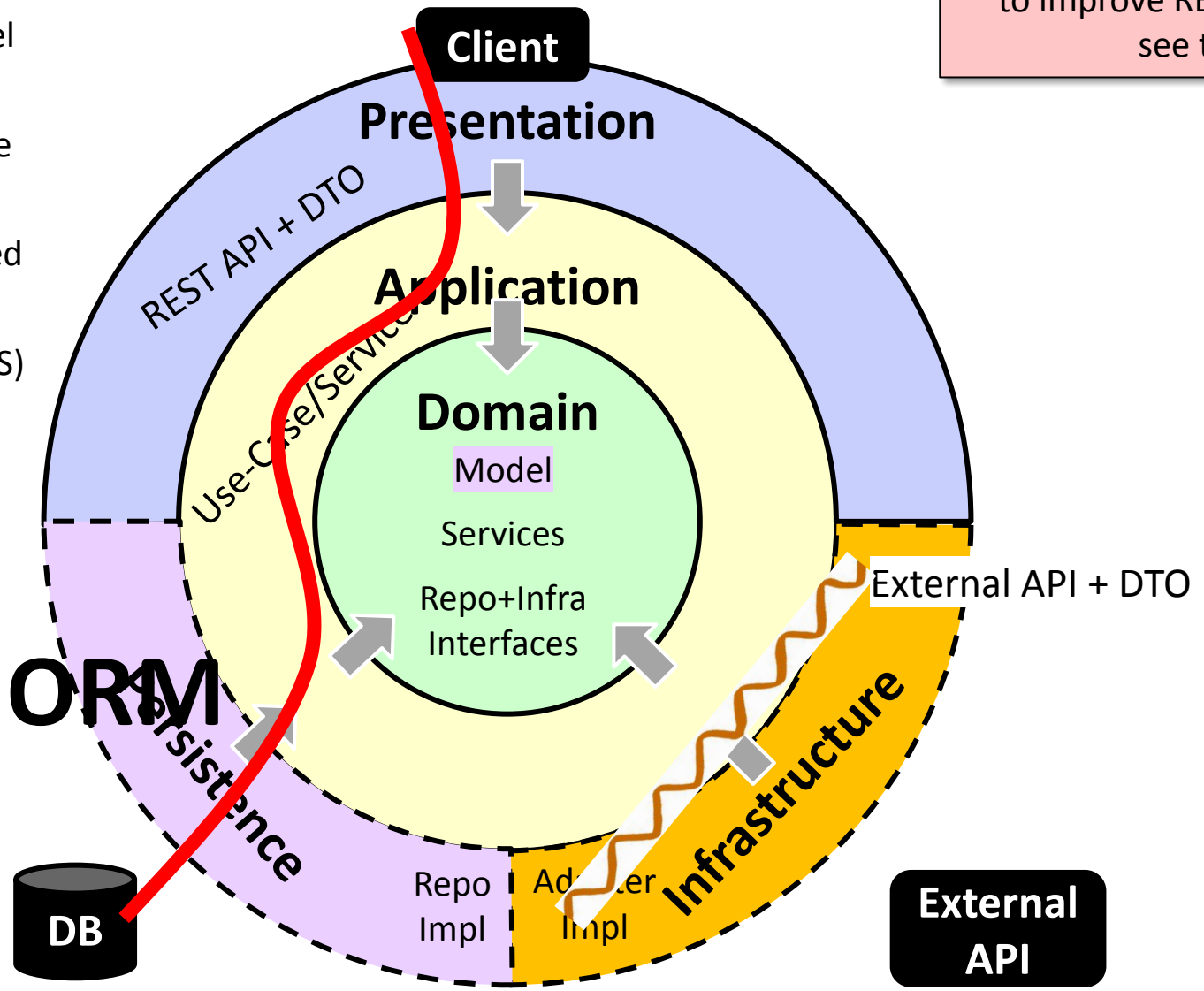
B) Allow ORM to manage Domain Model
If SQL DB is clean and private.

C) Merge Application with Infrastructure
If gap to External API is decent.

D) Output Adapters might not be needed when calling friendly APIs

E) Bypass Domain for Query flows (CQRS)
Extract DTOs directly from persistence (the adr)

Use-case-Optimized Search
SELECT DTOs directly from DB
to improve READ performance
see the ADR



Use-Case Optimized Searches

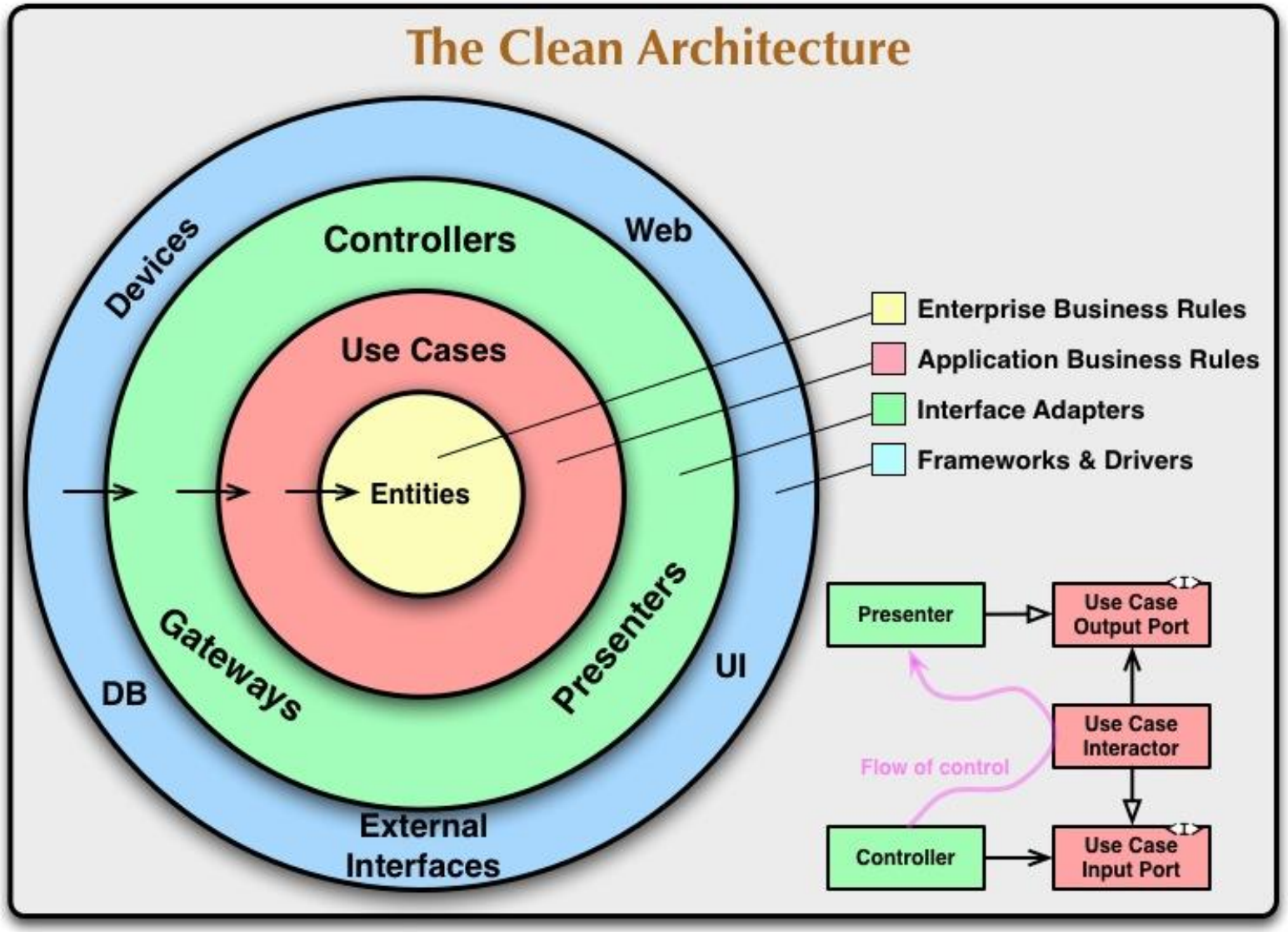
For serious complexity,

you need a

Clean Architecture

Clean Architecture

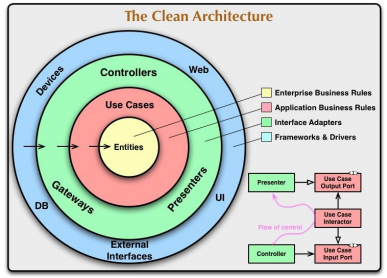
by Robert C. Martin ("Uncle Bob") in 2012



https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Clean Architecture

by Robert C. Martin (aka "Uncle Bob") in 2012



Core Logic should be... 1. Agnostic to User Interface (MVC)

mobile, **web** (jsp, jsf, GWT, Vaadin…), **desktop** (Swing, SWT…), or **reporting** (PDF, XLS,…)

Presenta ; and too screen-specific.

2. Agnostic to Database

the

"DB is a detail"

– Uncle Bob in Clean Architecture 📖

a) **Complex interaction** (without a framework): SQLException, .commit()

b) **Logic in DB** in [PL/]SQL = duplicated, focus-demanding, no unit tests, no refactoring, lock-in 🔒

c) **Complex schema** (600 tables * 5..80 columns) => messy Domain Model ±ORM

3. Agnostic to Foreign APIs

= other Bounded Contexts: LdapApi

4. Easy to Test 👑

only using my domain model

Scaling Promises (next-Neflix?)
Rapid Development
Design Principles
Cool Features

5. Agnostic to Intrusive? Frameworks

Learning Curve
Difficult to Maintain
Pay 80%$ for 20% tweaks
Lock-in Risk ▶️

Vert.x and Reactive (WebFlux), Kafka Stream, Camel, Axon, Akka, ±ORM

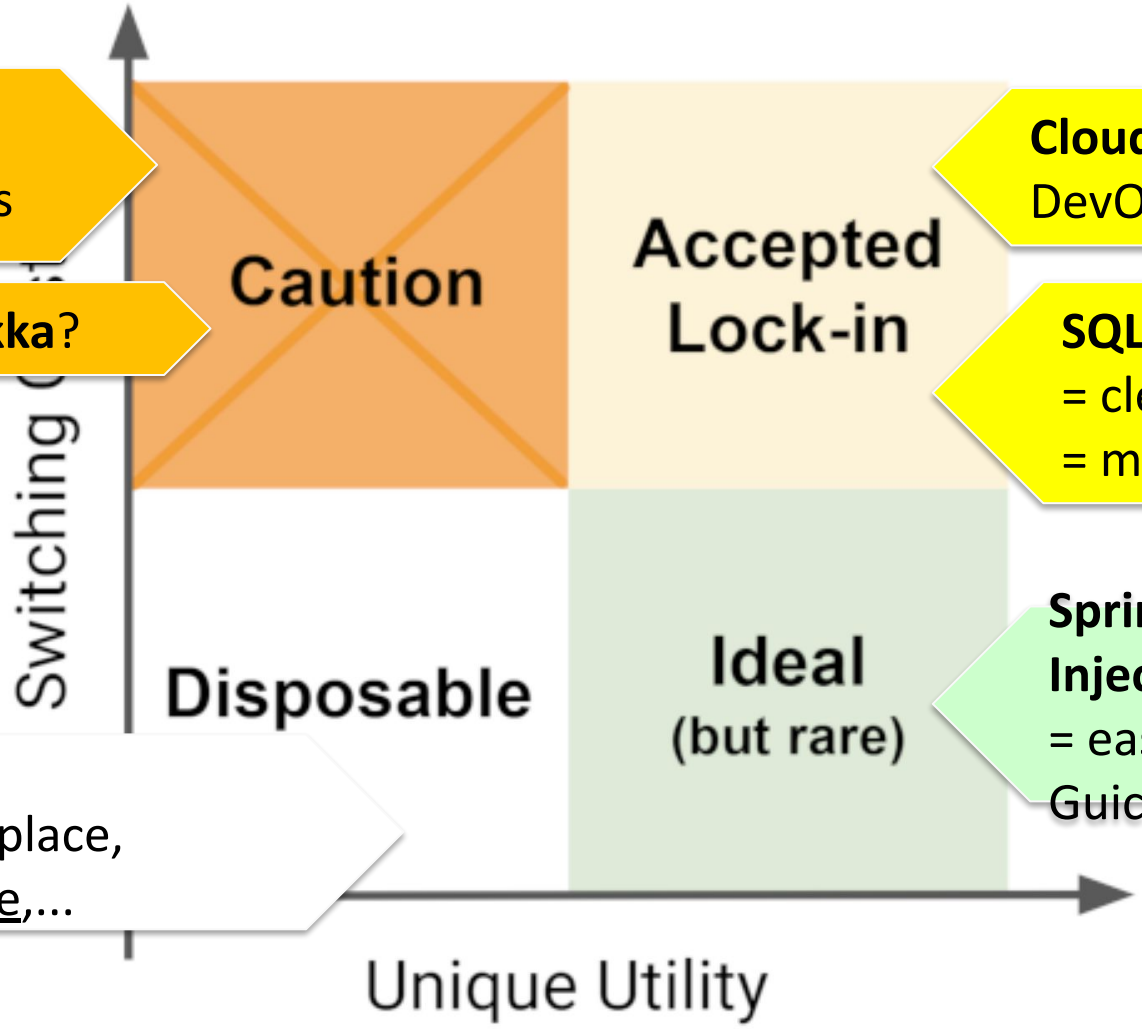
jHipster, FaaS, Vibe-coding, in-house-built framew💩rk(eg Cougar framework)

https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html

Don't get locked up into avoiding Lock-in

WebFlux for High API Throughput
= not unique: ∃ Virtual Threads
= very hard to migrate out

Akka?



Cloud: AWS, GCP, Azure
DevOps Teams / Hard to switch ☹

SQL interaction with Hibernate
= clean code/less boiler plate ☺
= magic/perf/no alternative ☹

Spring only for Dependency Injection
= easy to switch to 10+ alternatives
Guice, Quarkus, Micronaut, CDI...

StringUtils.*isBlank*(s) (Apache)
= trivial to migrate out, using replace,
Structural Replace, OpenRewrite,...

Inner Platform Effect
Building something worse
than the framework you're on

https://martinfowler.com/articles/oss-lockin.html

Overengineering in Hexagonal Architecture

Too many interfaces
"any layer must only expose and consume interfaces"

Delete interfaces with only 1 implementation, unless Dep Inversion

Mandated layers
"a Controller MUST call a Service that MUST call a Repo"

Extract complexity into Domain Services; challenge Controllers

Mock-full tests 🤢
"test each layer by mocking the layer below"

More integration tests (honeycomb) or squash layers

Separate Domain from Persistence 😭
"CustomerModel should be mapped ⇔ CustomerEntity in Repo layer"

Only required if legacy/enemy DB/no ORM

Application-layer structures
"Application layer should have its own data structures ≠ Domain, ≠ Dto"

Only required if UC exposed via 2+ channels

Rigid dependency management
"domain MUST NEVER depend on anything outside"

ArchUnit allows exceptions 😬

Monolithic in nature
"the (one) Domain is in the center"

Gulp... 😬 Indeed

Domain Model

vs

Persistence

Database Type

- SQL DB == **Shared** || **Legacy** (**667 tables**) => 🤬+😱
  => Decouple Domain Model from DB structure via DAO (typically using SQL*)
- SQL DB == **Private** => 💖
  => Domain Model == ORM @Entity/data-jdbc: use same set of classes for both
  - Don't fear ORM, **understand it**! (click & understand all boxes of next slide)
- **Variable structure**🤔**:** document store like **Mongo**...
- **Instant Access** (~Cache☢️): **Redis**$$$ (≤1ms access), **Hazelcast**...
- **Heavy writes**: **Cassandra**...
- **Big data (PB)**: **Hadoop**, Spark...
- **Search-optimized, full-text search:** Neo4j, Elastic Search...
- **AI DB**: vector store

*Java SQL Libraries: JdbcTemplate, CrudRepository, MyBatis, jOOq, JDBi, Exposed..

Click any box🔽 to learn more about **ORM's magic, traps and myths**

Dirty-check Auto-save

```
@Transactional
f() {
  var e = repo.find(id);
  e.setField("change");
}
```
□UPDATE w/o repo.save

Write-behind

```
@Transactional
f() {
  repo.save(new E());
}
```
□ INSERT after method exit ,before transaction COMMIT

Lazy-Loading

```
f() {
  var parent = repo.find(id);
  parent.children().stream()
}
```
at 1st access □ SELECT

Rollback

```
@Transactional
f() {
  repo.save(new A())
}
```
□ ROLLBACK on any runtime exception

Default Constructor

🚫Myth: Entity must have a public no-arg constructor and setters

@Version

= optimistic locking to detect concurrent changes to data

Transaction Propagation

```
@Transactional
f() {
  this.g(); // ❌ no on local call
  other.g(); // ⚠️ on same thread
}

@Transactional(propagation=...
g() {..}
```
□ uses **f** transaction?

merge()

of an Entity ± children

HashCode/Equals

Change after @Id is assigned. Use Natural key.

@Embedded

🚫Myth: Object fields must match table columns

Session in view

Lazy loading still works after @Transactional end in a REST api call

fetch=EAGER

Loads children every time you load a Parent

Rich Entities

🚫Myth: Entities must be anemic (have no brains)

Search for full Entites

It's inefficient to SELECT large entities for only a few fields

N+1 Queries

= JPA can send repeated identical queries to DB

Sequence PK

@SequenceGenerator brings a page of 50 IDs

Example: JPA/Hibernate/Entity Framework/Nhibernate/Doctrine...

ORM Cons

- **Noisy Annotations** — **Tolerate** — 🚫 **DON'T move to .xml**
  - @Column, @AttributeOverrides, ...
- **Magic Features:** — **Learn**
  - Lazy loading, auto-flush changes, write-behind... *(see next slide)*
- **Performance Traps:** — **Understand**
  - N+1 queries, full entities for search, ... *(see next slide)*
- **Myth: ~~ORM Requires anemic POJO~~:** — **False**
  - No-arg constructor + getter + setters + no extra methods... *(see next slide)*
- **Storage-first mindset** — **Fight**
  - OOP Objects with Behavior allow deeper modeling than Tables and FKs

ORM allows us to craft
a better-quality Domain Model,
by **easing experimentation** and
shielding from SQL noise.

But, like any powerful tool,
it's not easy to handle.

Hey Chat, compare SQL, ElasticSearch, Redis, Cassandra, Hadoop, Mongo, Neo4j DBs in TikTok style

Redis:
🔥 **Speed Demon** – “Cache me outside, how ’bout dat!”
🚀 In-memory key-value store, perfect for real-time leaderboards, sessions, and quick lookups.
⚡ If you need it fast and simple, Redis got your back.

Cassandra:
🌍 **Scale God** – “Big Data? Hold my cluster!”
💪 Write-heavy champion, distributed, fault-tolerant.
📈 Bestie for apps like Netflix, but hates complex queries.

Hadoop:
🐘 **Data Mammoth** – “Store it, process it, ask me later.”
🛠️ Batch processing king, loves big analytics but takes its sweet time.
💾 Perfect for massive offline data crunching.

MongoDB:
📚 **Flex Queen** – “Schema? Never heard of her.”
🍕 JSON-like docs, dynamic AF, and scales well horizontally.
👩‍💻 Great for fast-moving startups & apps needing flexibility.

Neo4j:
🔗 **Graph Guru** – “Relationships? Let’s talk.”
🧠 Loves graphs, ideal for social networks, fraud detection, or knowledge graphs.
🤓 Query with Cypher, think in nodes & edges.

Elasticsearch:
🔍 **Search Sorcerer** – “Type a word, and I got you!”
⚡ Real-time, full-text search and analytics wizard.
📈 Loves logs, metrics, and anything you want to sea through *fast*.

SQL Databases:
📊 **Structured Bae** – “Rules, rows, and relationships
🗄️ Perfect for structured data, complex queries, and ACID compliance.

"Blasphemy"
- 15 old architect

≠ Modular Monolith
it's finer-grained than that

Vertical Slice Architecture (VSA)

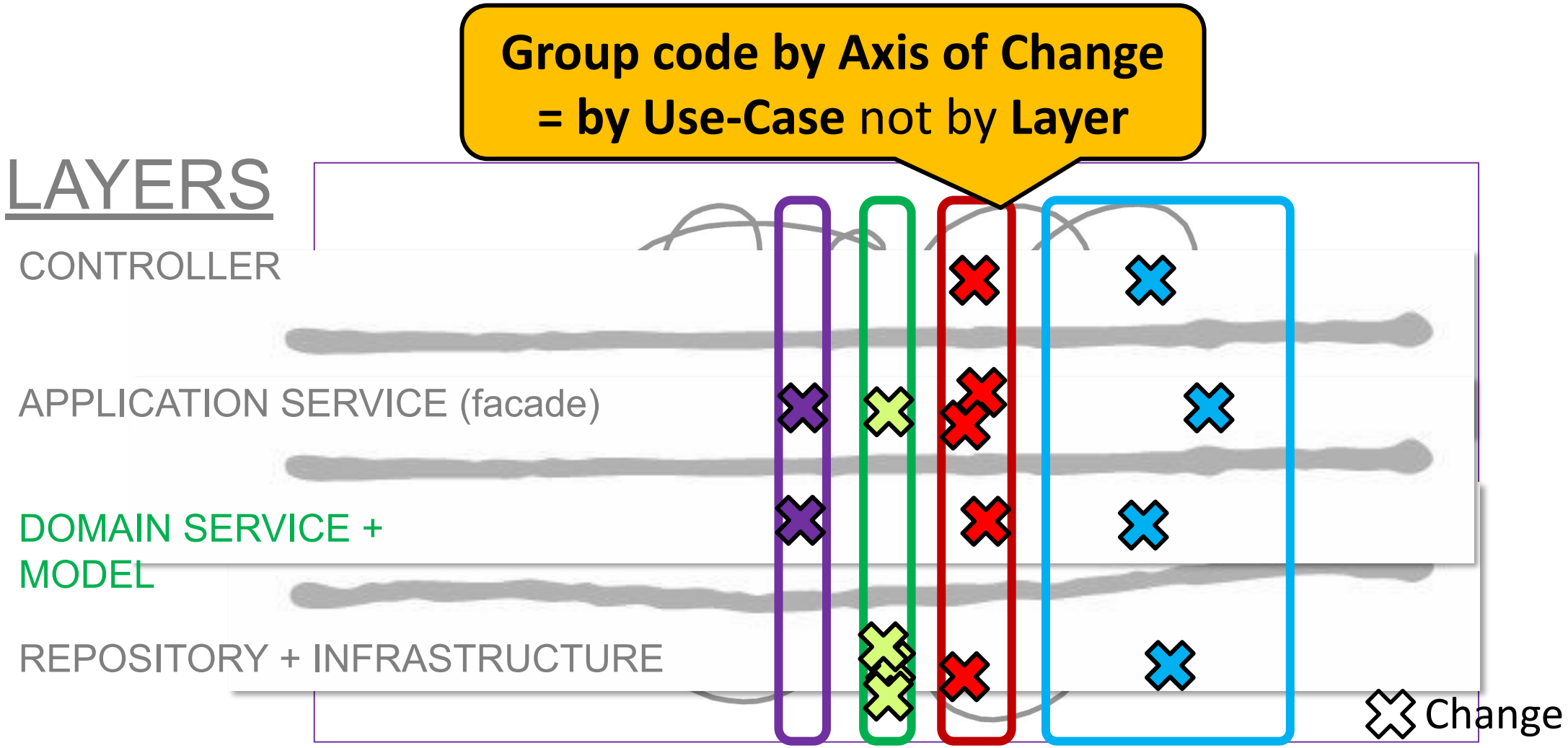
aka "Anti-Layers" Architecture

by Jimmy Bogard in 2018,
only for microservices with limited complexity ⚠️

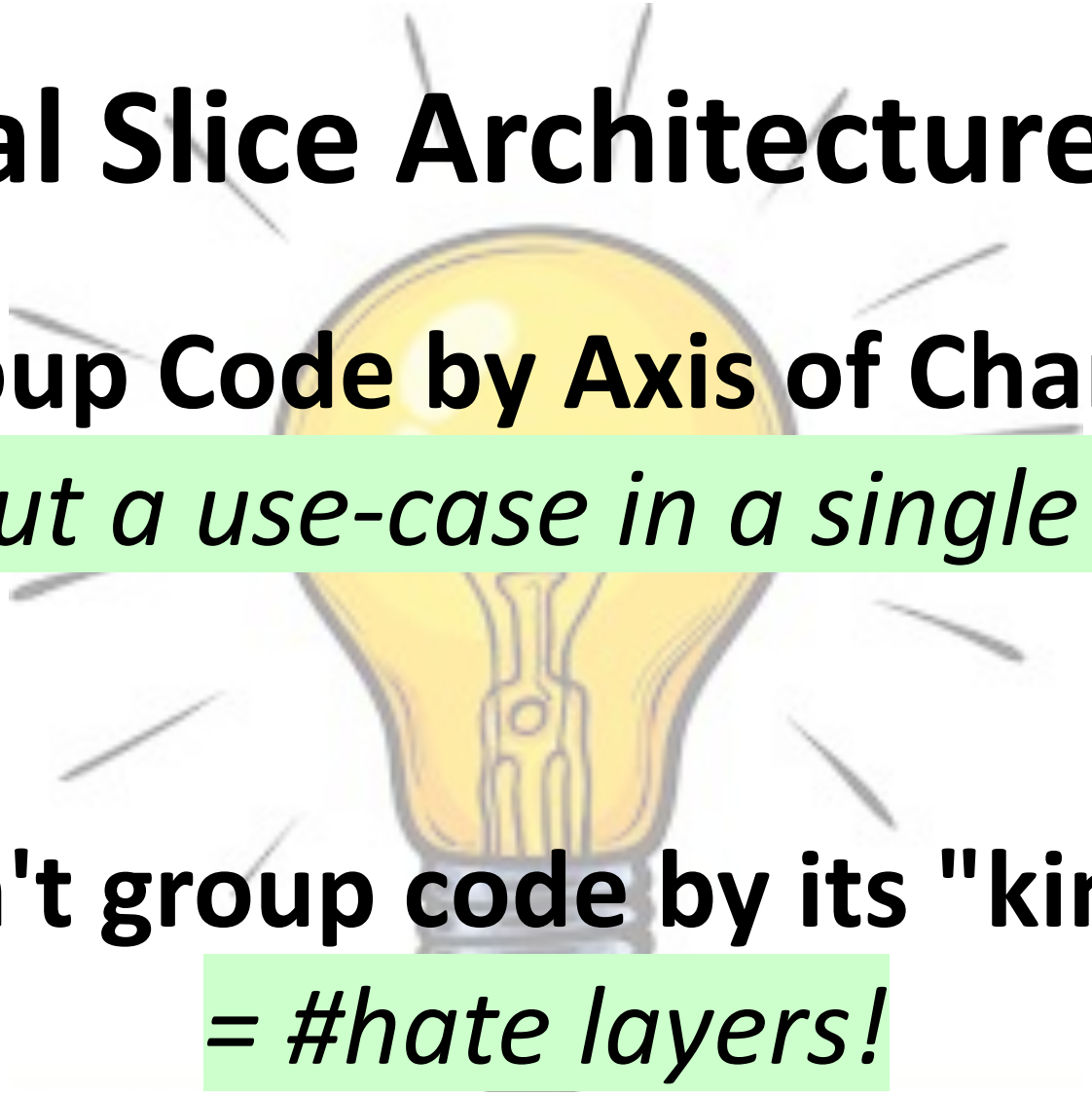
More pragmatic than 🍅 Tomato Architecture

Article: https://jimmybogard.com/vertical-slice-architecture
Talk: https://youtu.be/5kOzZz2vj2o

Vertical Slice Architecture (VSA)



Vertical Slice Architecture (VSA)



Group Code by Axis of Change

= put a use-case in a single file

Don't group code by its "kind"!

= #hate layers!

Vertical-Slices In Practice

Pragmatic Low-dogma Architecture

Babel of Technologies 😵

Simplified by frameworks today

What can go wrong?

```
class GetOrderByIdUseCase { // CQRS
  REST,SQL,JPA,Avro,Swagger...
}

class CancelOrderUseCase { // CQRS
  20 lines xyzabcabc
}

class PlaceOrderUseCase {
  400 lines xyzabcadc
}
```

call

Duplicated Logic, worse reinvented

Coupling++, Unclear SLAb

Monster Class 😈

core

Extract **Shared Logic**

DRY

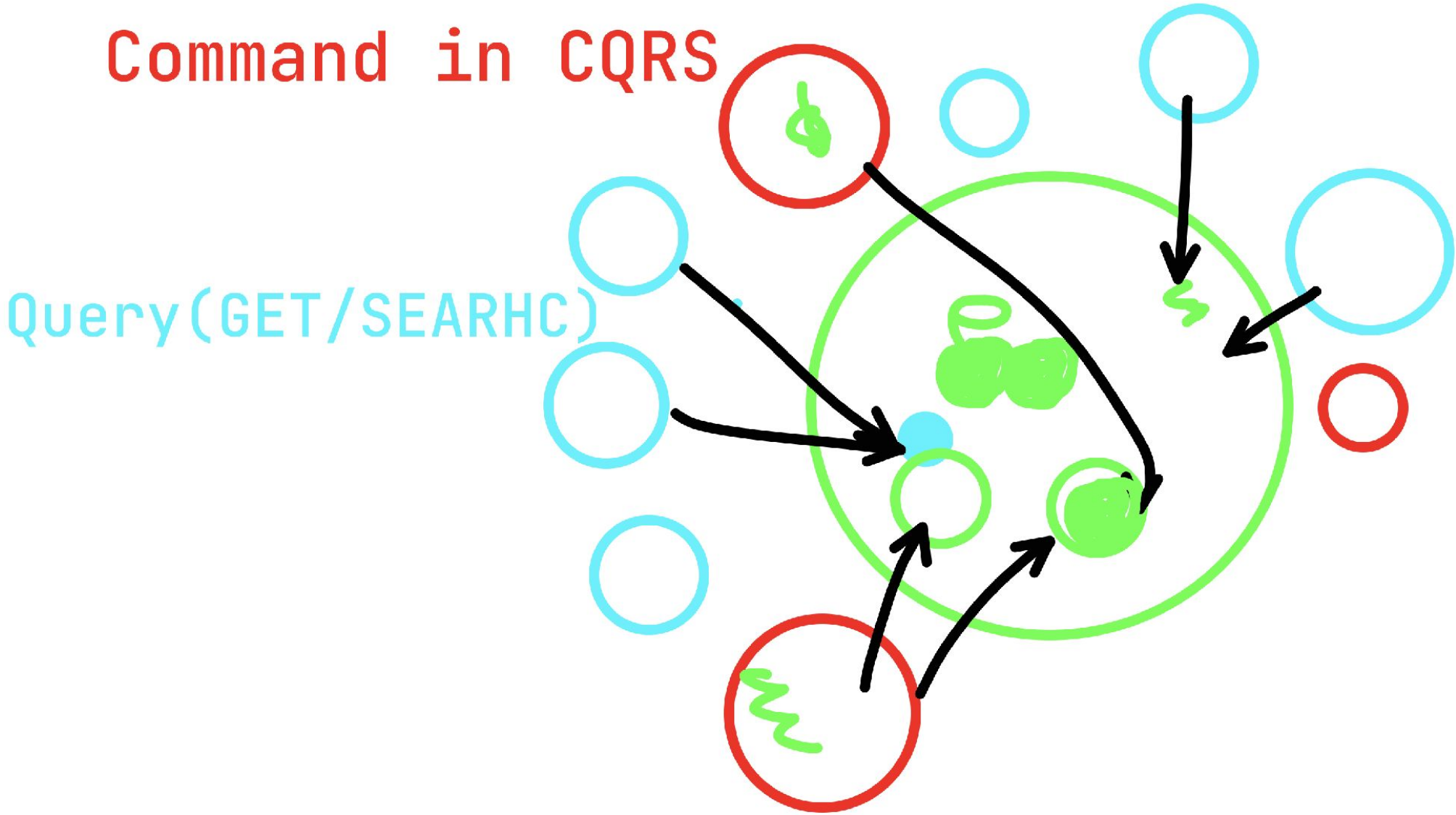
Promote to Core (Domain) Model and/or Service 👑

Make logic only use internal Domain Model

Explain **Complexity**

SRP

Katarina's Slide



Example VSA Use-Case

```
class PlaceOrderUseCase {
  @VisibleForTesting// used only by tests, not by any other Use-Cases
  static class PlaceOrderRequest { ... }
  static class PlaceOrderResponse { ... }

  private final Dependency1 dep1;... // only deps required by this UC❤️

  // 1 single public method
  @PostMapping(..) or @MessageListener(..)
  public PlaceOrderResponse handle(PlaceOrderRequest request) {...}

  // some private methods as necessary

} // class kept small⚠️ (eg < 200 lines)
```

Query Use-Cases 👁️

```
class GetOrderByIdUseCase {
    static class Response { ....<large>..... }
    @GetMapping
    public Response handle(@PathVariable long id request) {...}
}
```

- Can select read projections directly in Dtos = different READ Model (CQRS)
- Can avoid layers/rings and use anything necessary raw SQL,...
- If ≥2 Query UCs share logic (rarely) □ extract it to DRY

Command Use-Cases

Perfect Fit for Task-Based UI

```
class PlaceOrderUseCase {
    static class Request { ....<large>.... } // validation: @NotNull, @Size..
    static class Response { ... }
    @PostMapping
    @Transactional
    public Response handle(@Validated @RequestBody Request request) {
       *read stuff*
       *interesting domain logic to extract*
       *save stuff*
    }
}
```

- Can grow very complex and overlapping
- Continuously refactor:
  - Extract Methods
  - Extract Classes
  - Move Methods (into a **Domain** Model Object or a **Domain** Service)

Application Service Layer/Ring

= Multiple entry points / class

```
class OrderApplicationService {
  public void getOrderById(...) { ...
}
  public void placeOrder(...) { ... }
  ...
}
```

simpler to follow

risk: grows large fast

V vs. Λ

Vertical Slice Architecture

= One use-case / class

```
class GetOrderByIdUseCase{ ... }

class PlaceOrderUseCase { ... }
```

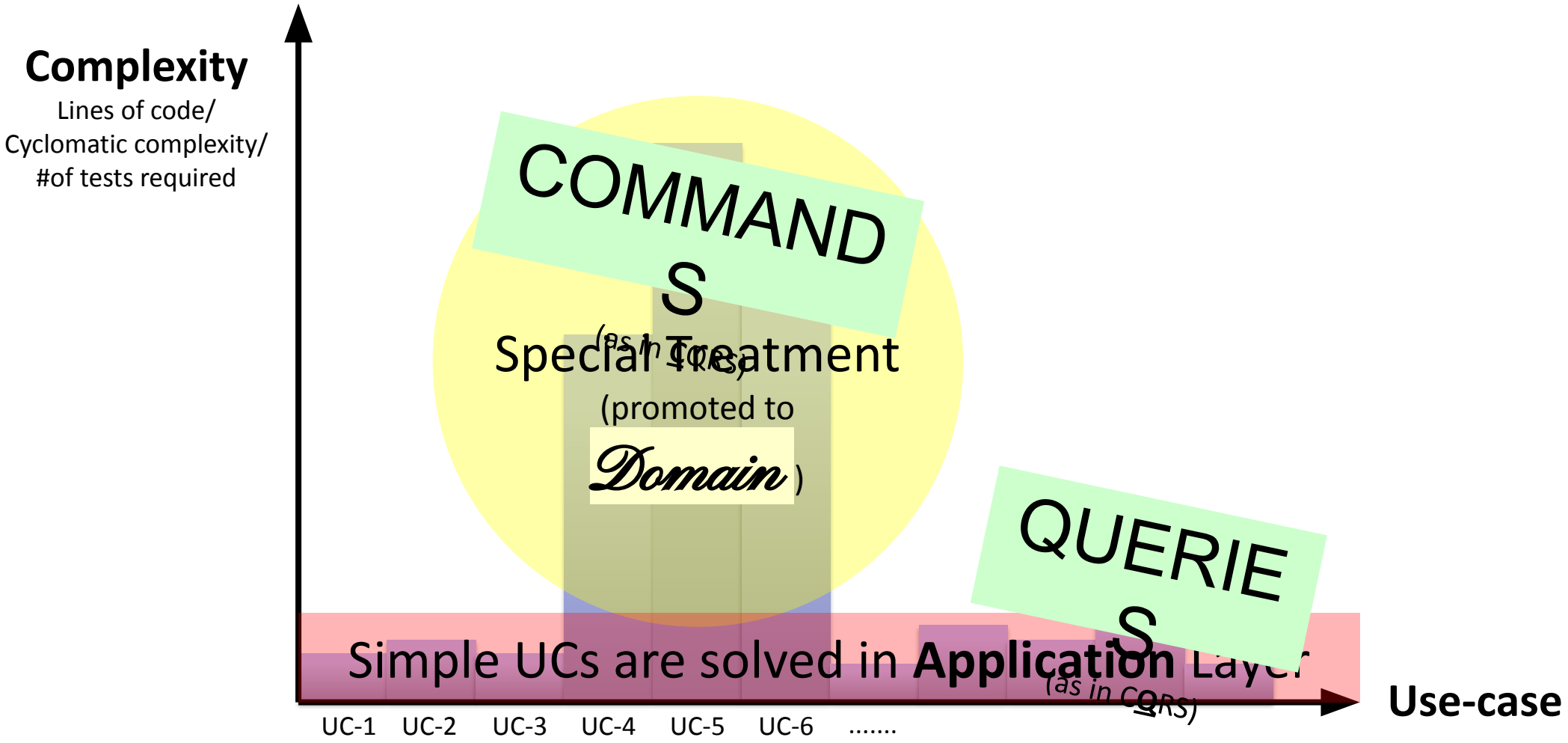
narrow change scope

better decoupling

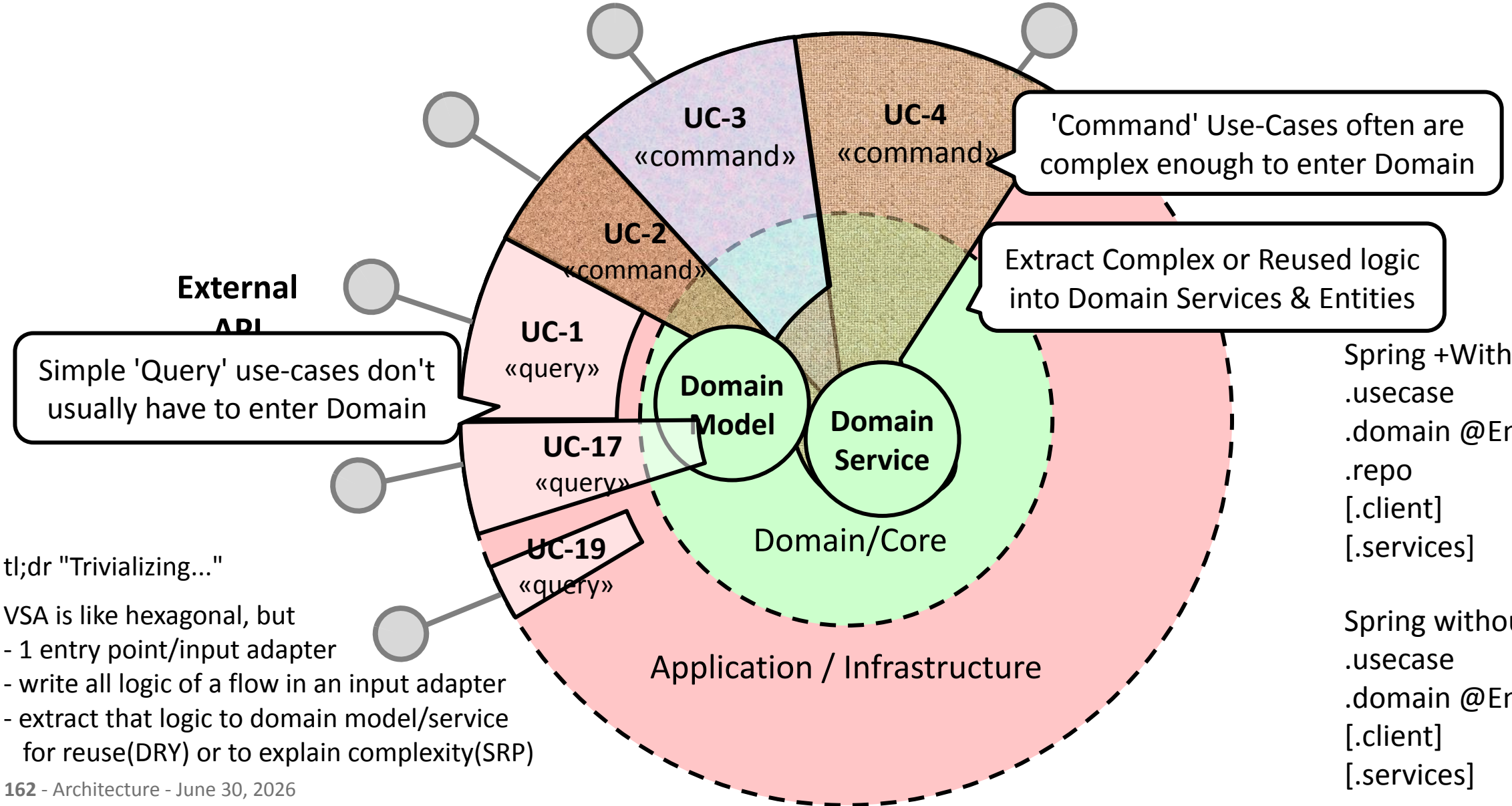
more space for 1 flow

Uneven distribution of complexity per use-case

(typical in backend systems)



VSA Converges to Concentric Architecture



Spring +With ORM:
.usecase
.domain @Entity+VO
.repo
[.client]
[.services]

Spring without ORM
.usecase
.domain @Entity+VO
[.client]
[.services]

tl;dr "Trivializing..."

VSA is like hexagonal, but
- 1 entry point/input adapter
- write all logic of a flow in an input adapter
- extract that logic to domain model/service
  for reuse(DRY) or to explain complexity(SRP)

Vertical Slice Architecture - Summary

- **= Group code by axis of change** = by Use-Case, not by Layer
  - ✅ Smaller blast radius to add/change/delete a feature
  - ✅ Clearer what to integration-/unit- test: less dependencies, no boilerplate
  - ✅ Less coupling: entry points start off isolated
  - ✅ More space to spread your (command) use-case in an entire file
- **Requires Continuous Refactoring** ☣️
  - by **Experienced Teams** familiar with layers ± hexagonal, but postponing it
  - **Pair Programming** to prevent **Monster Classes** 😈 and **Duplicated Code**
  - ... and **Increase Awareness** of entire codebase
- **Very Pragmatic** ⭐

tl;dr: merge code top-down and not have >1 entry point in a file

Imitate Others > Learn the rules > Master them > Then learn when to break them

SHU

FOLLOW THE RULES

HA

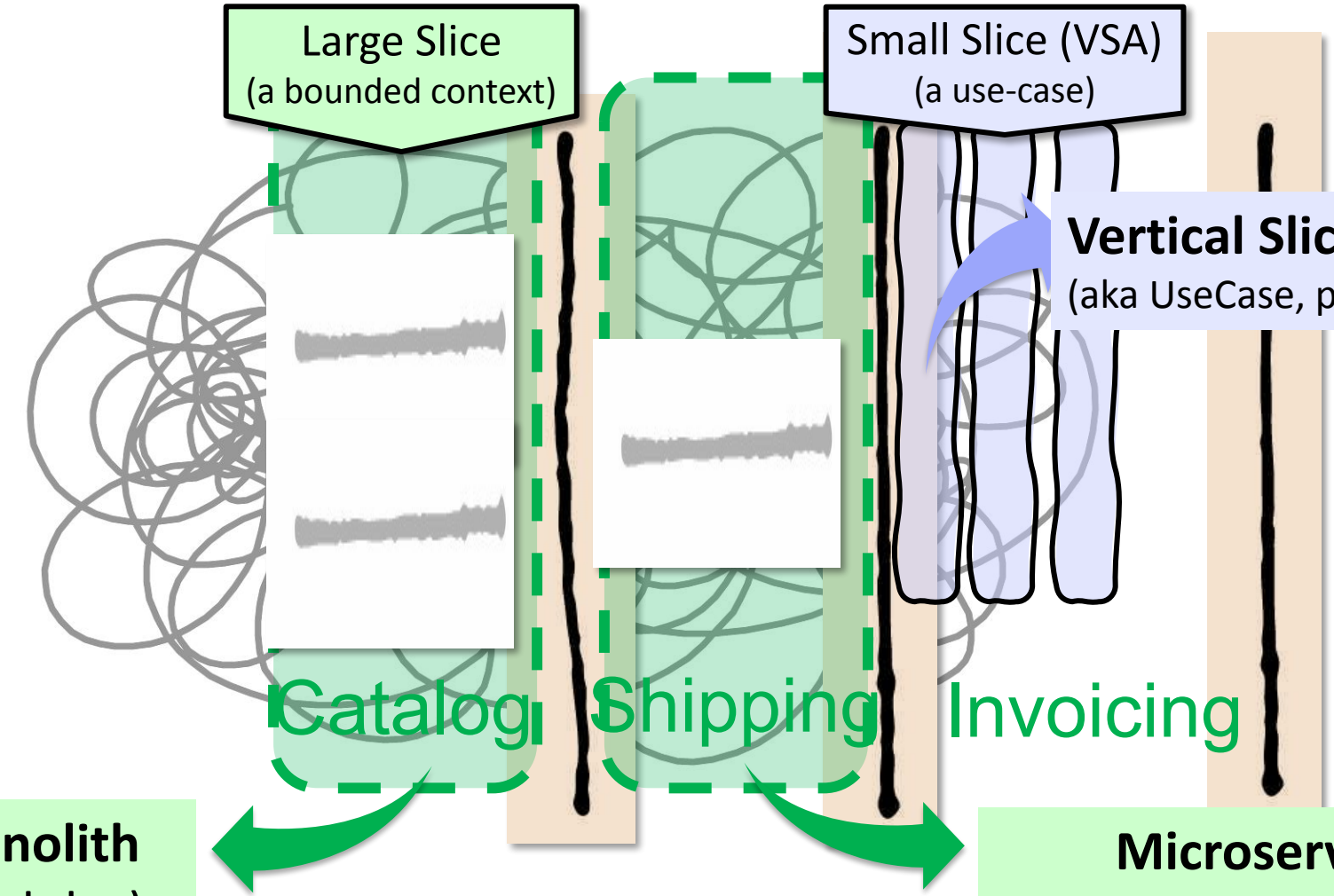
BREAK THE RULES

RI

TRANSCEND THE RULES

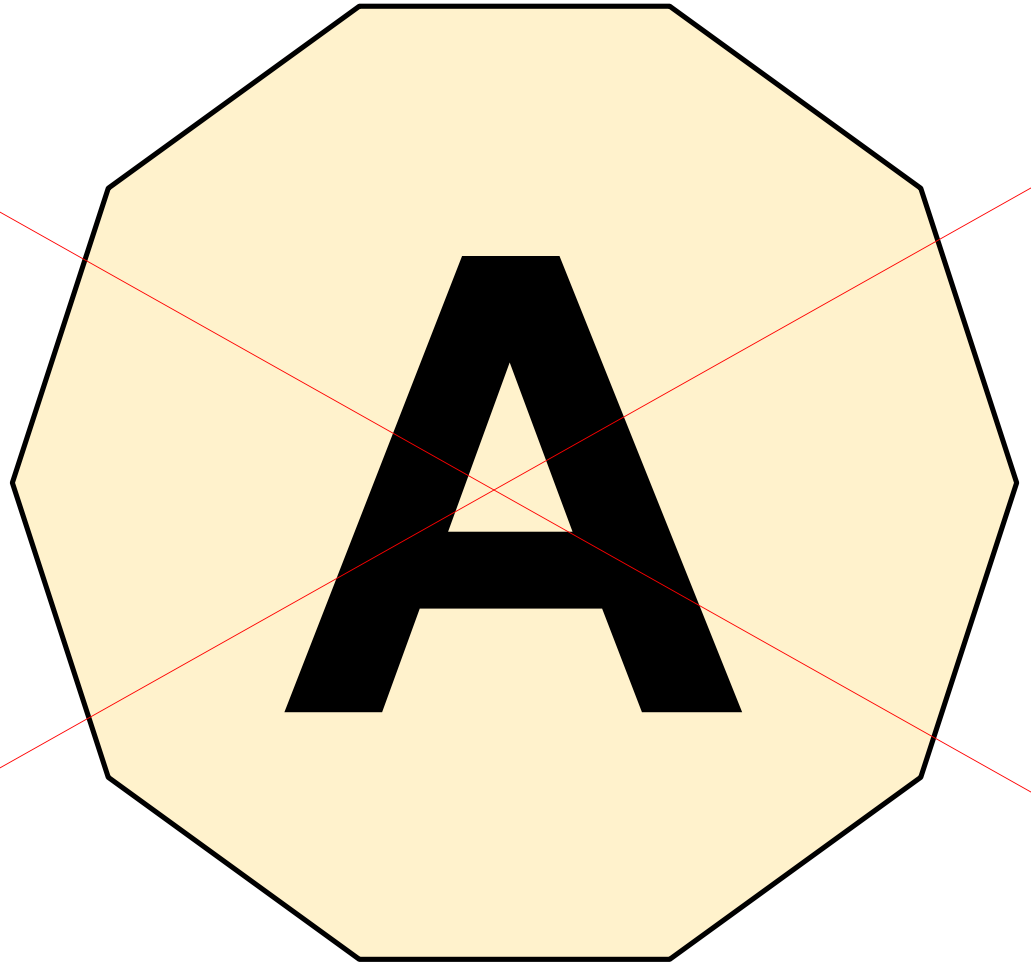
GROUPING CODE IN VERTICALS

What Size?

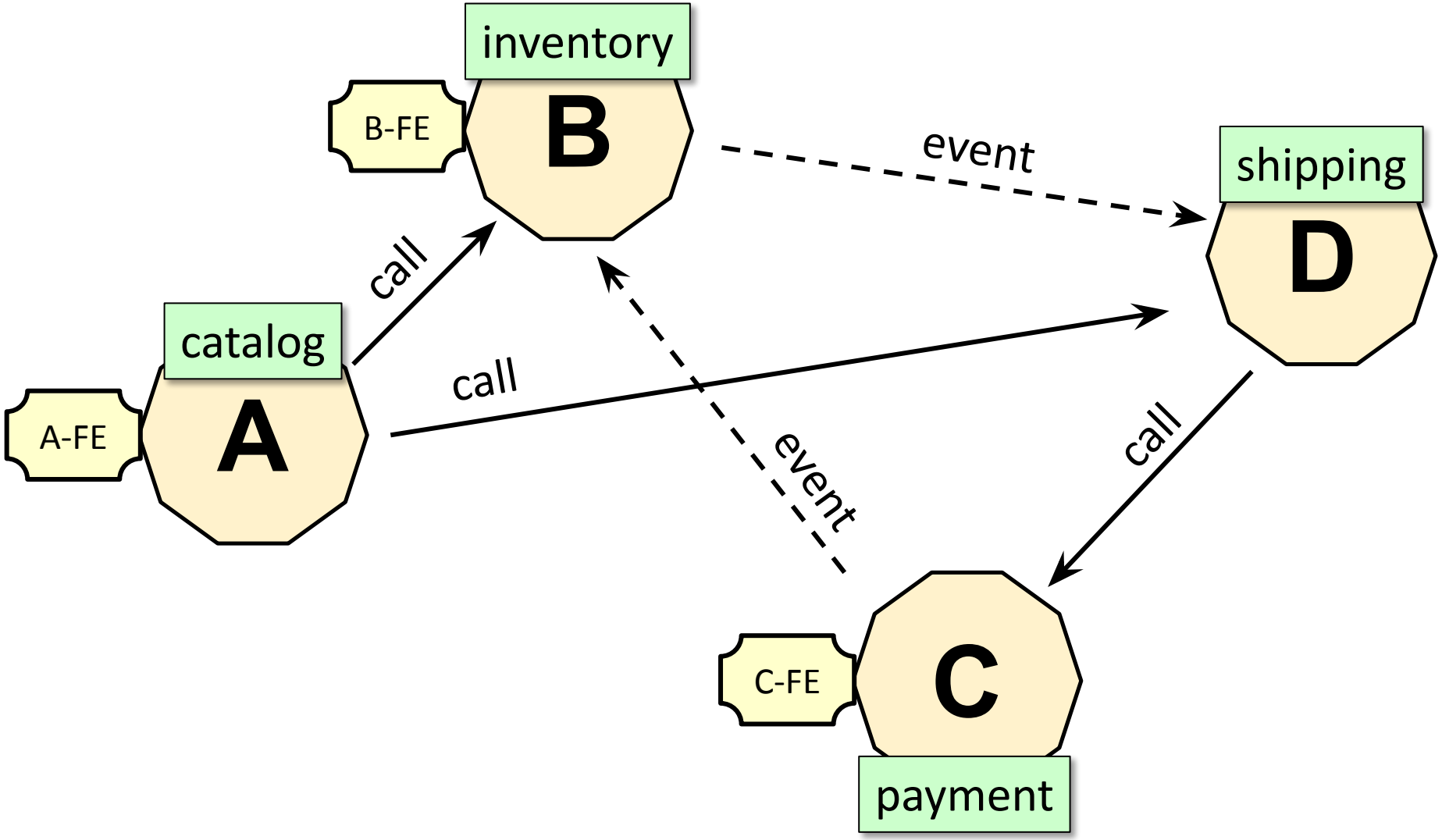


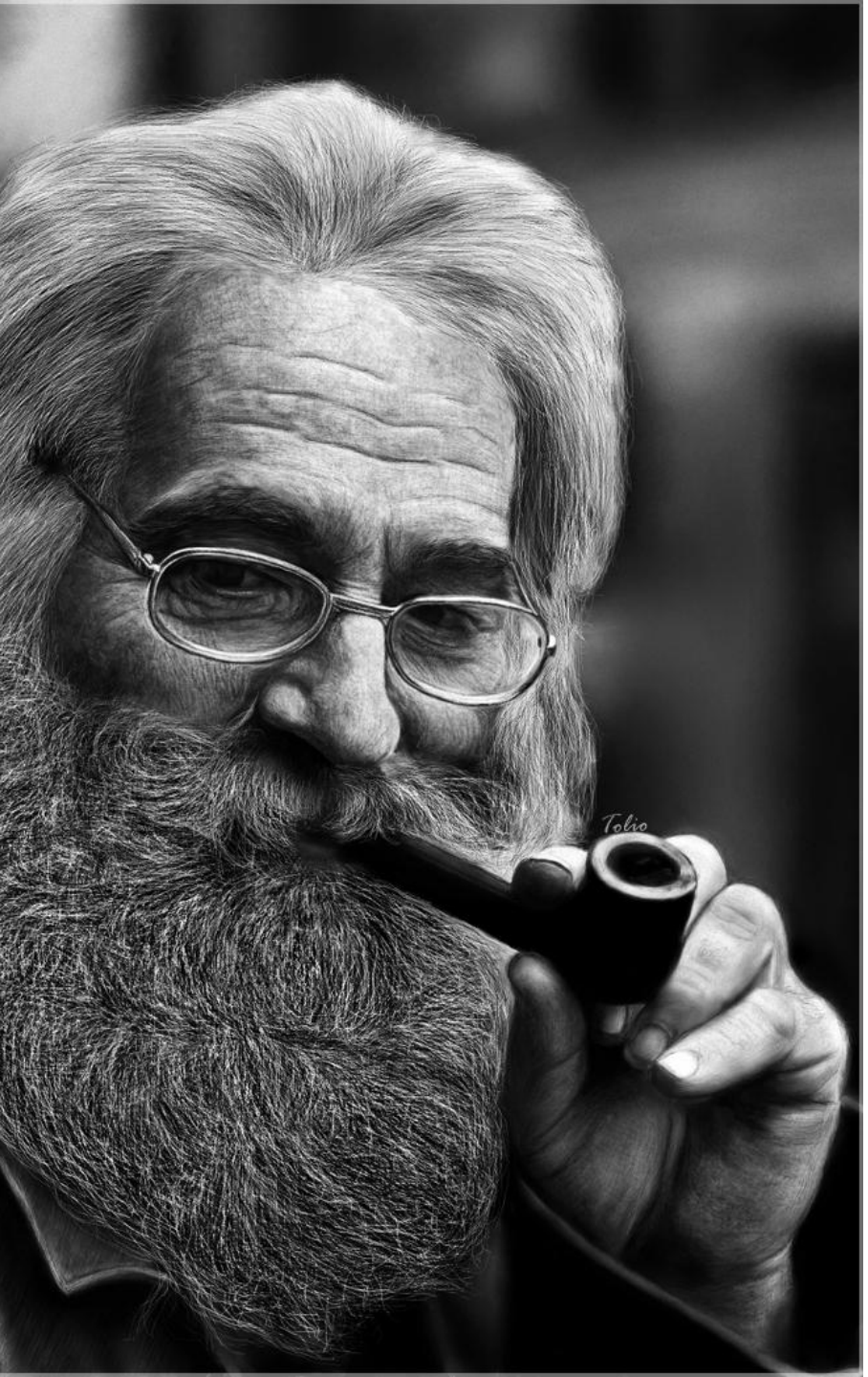
Modular Monolith
(separate modules)

Microservices
(separate deployments)



Split in Modules / Microservices





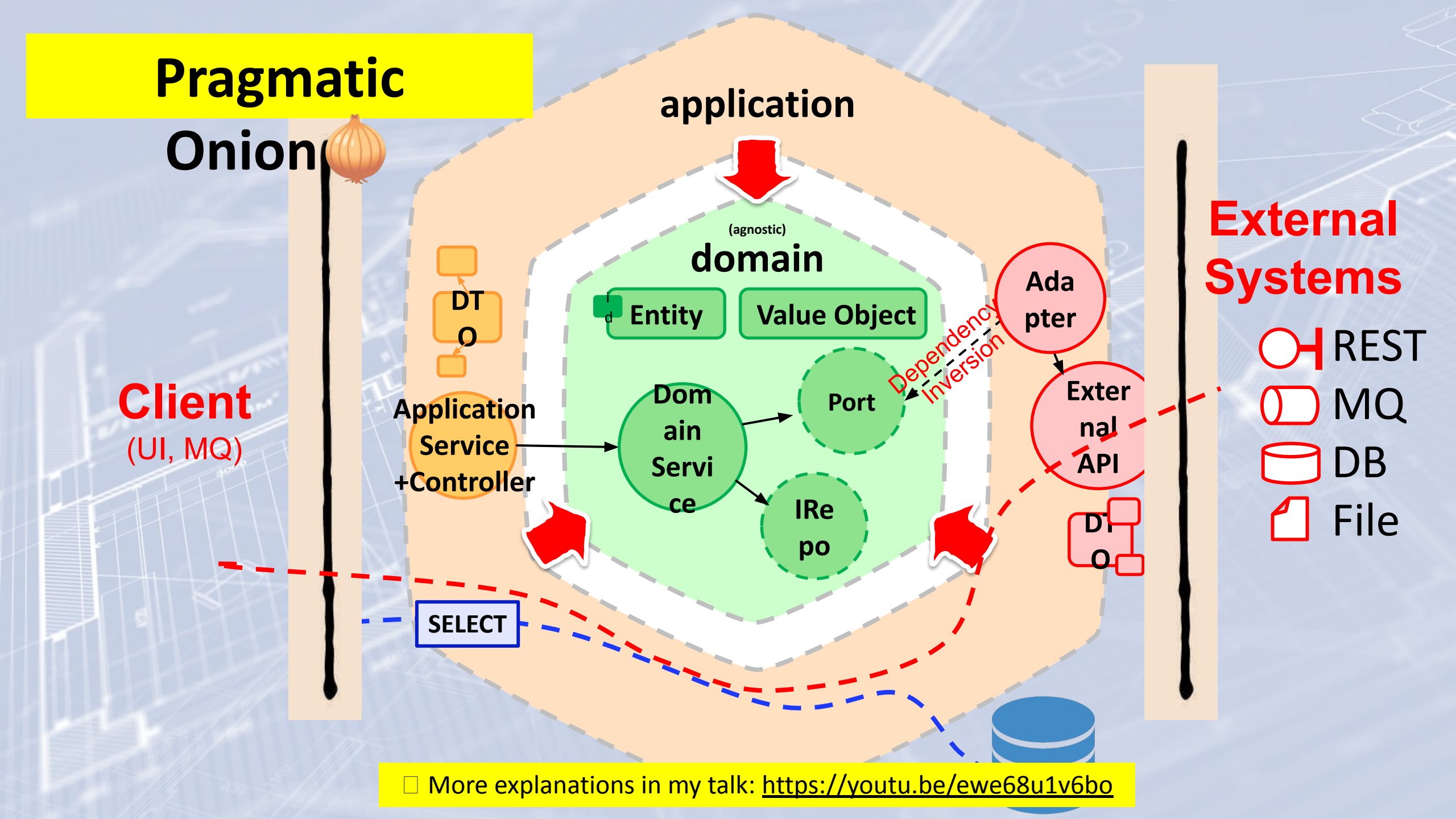
Elders Wisdom

A large project struggles more with **internal coupling**
(= domain tension)

than **coupling to external APIs and Libraries**

Focus of
Concentric Architectures
(Clean, Onion, Hexagonal)

Pragmatic Onion

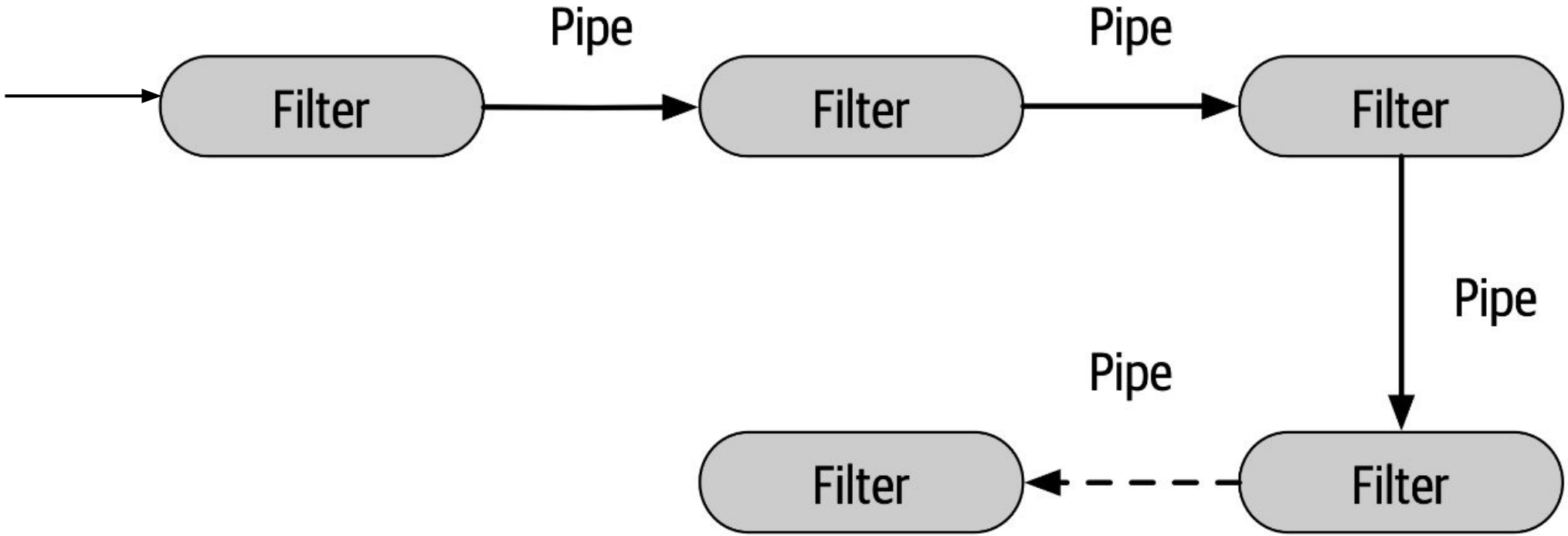


More explanations in my talk: https://youtu.be/ewe68u1v6bo

Pipeline Architecture

Possible Design Goals

- Decouple filters; ability to swap/in/out/combine different client
- Order of filters can matter => temporal coupling =>🤯
- Protect my data from accidental overwrites by a later filter => favor (partly) immutable data
- KISS: don't overengineer postpone create those interfaces you see
- Performance: can we run any independent filters in parallel?
- If filters connect bring data from outside, protect internal data structures "enricher"




```
cat /var/log/syslog | grep "error" | sort | uniq -c | sort -nr | head -10
```

Architecture Evolution

- **Layers** (jr)

⬇️ Rich Domain

- **Hexagonal / Onion**

⬇️ Domain Tension (separate Bounded Contexts)

- **Functional Java Packages** (modulith)

⬇️ Large team

⬆️ Realign Boundaries

- **Functional Maven Modules** (modulith)

⬇️ NFRs (Scalability, Availability, Tech Upgrade, ...)

⬆️ Nano-services, Performance, Consistency, Change-Coupling..

- **Separate Deployments** (Service-Based) / **Microservices**

Quiz

Remember: ***it depends***

1) The three Laws of Architecture:

⚠️ Select **3**

a) The Architect is always right

b) Why is more important than how

c) Design extensible code to reduce effort/redesign tomorrow

d) **Everything is a tradeoff**

e) Evolution is innevitable

What to do about them?

2) What is a Rich Domain Model in DDD?

⚠️ Select 3

not "Anemic"

eg Customer.java

a) Entities hav

b) Entities contain bits of decoupled business logic based on its fields

c) **Entities refer to money**

d) Entities guard domain constraints

e) The domain contains more than 100 Entities

f) Explicitate domain concepts as Value Objects

3) A Value Object = ...

⚠️ Select **2**

record ShippingAddress

a) Is Mutable (has setters)

b) Is Immutable

c) **Has a PK (persistent identity)**

d) Should NOT contain logic

e) Is typically small

4) Ubiquitous Language:

⚠️ Select **1**

a) Is used everywhere in the entire company

b) Is created by developers

c) **Is created by non-technical people**

d) Is used by a few people within a Bounded Context

e) **Requires a dictionary to translate domain terms to code**

5) Three Axis of Complexity in implementation:

⚠️ Select **3**

a) Data semantics (Multiple Bounded Contexts)

b) Long boilerplate code (eg. in a Mapper)

c) Writing logic in the data structures

d) Dependencies (Orchestration)

e) Cyclomatic (Execution Branches ~= @Tests)

6) Layered Architecture Requires ...

⚠️ Select **ONE**

a) Packaging layers in separate compilation modules

b) Having 3 layers, for example: Controller, Service, Repository

c) Equal distribution of complexity between layers

d) Groups of classes which depend on each other with NO cycles

e) Each layer must have its own data structures

7) ❌ INCORRECT in a Layered Architecture ...

⚠️ Select ONE

a) Have more than 3 layers

b) Skip an open layer, if calling in the same direction

c) A layer can use the data structures of another layer

d) A layer can call back into the caller layer

8) CQRS (Command/Query Segregation) says:

⚠️ Select **ONE**

a) Segregate pure functions from side-effecting ones

b) Have different data models to read and update data

c) Migrate changes asynchronously to a read projection

d) Use message queues to read and write data

e) Use different DB Connection Pools to read and write

9) Separation by Layers of Abstraction Principle

⚠️ Select **TWO**

a) Decompose high complexity in decoupled subproblems

b) Don't mix interfaces with concrete classes in a package

c) Have abstract classes extend other abstract classes

d) Don't mix high-level policy with low-level details

10) An Application Service, acting as a "Facade":

⚠️ Select **TWO**

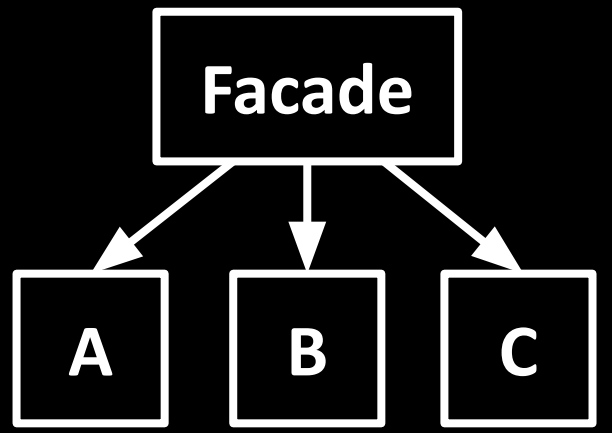
a) Orchestrates a use-case

b) Is decoupled from other components

c) Should implement core complexity

d) Allows downstream components to be less coupled to each other

e) Contains a single public method



Name A+E

11) Dependency Inversion Principle means...

a) A container injects instances of classes you declare you need

⚠️ Select **TWO**

b) High-Level policy should not depend on Low-Level Details

c) Get rid of an unwanted library dependency

d) Call direction and code dependency are reversed

e) You don't call the framework; the framework calls you

12) In a Hexagonal Architecture, the Use Cases:

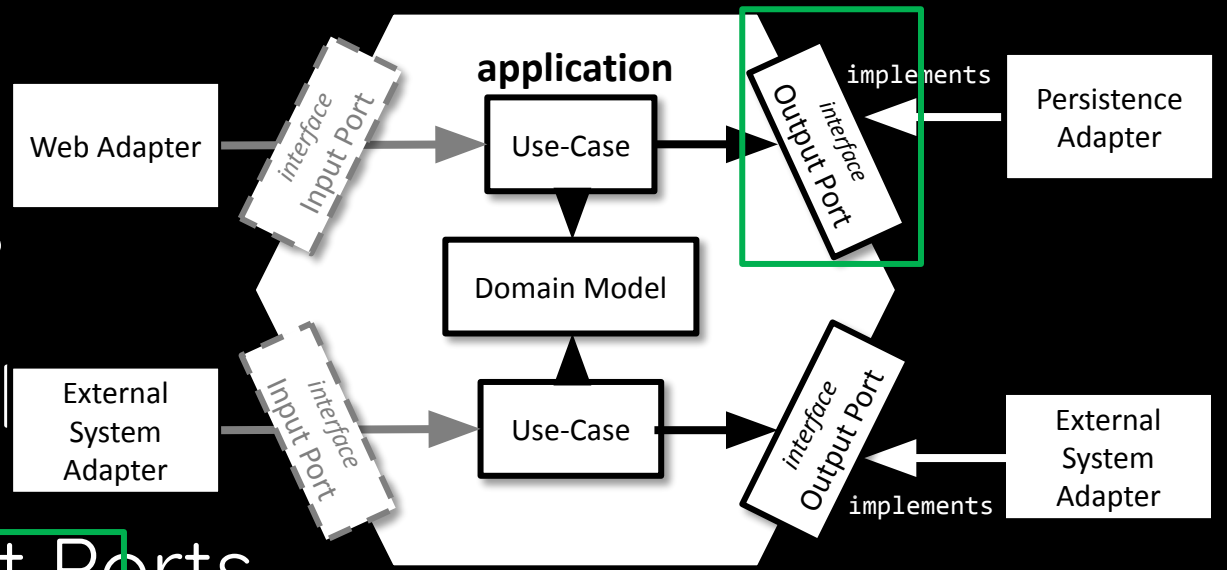
Select **ONE**

a) are called via Output Ports

b) talk to the database direct

c) call outside APIs via Output Ports

d) are isolated from Domain Model via Adapters



13) Role of an Output Adapter

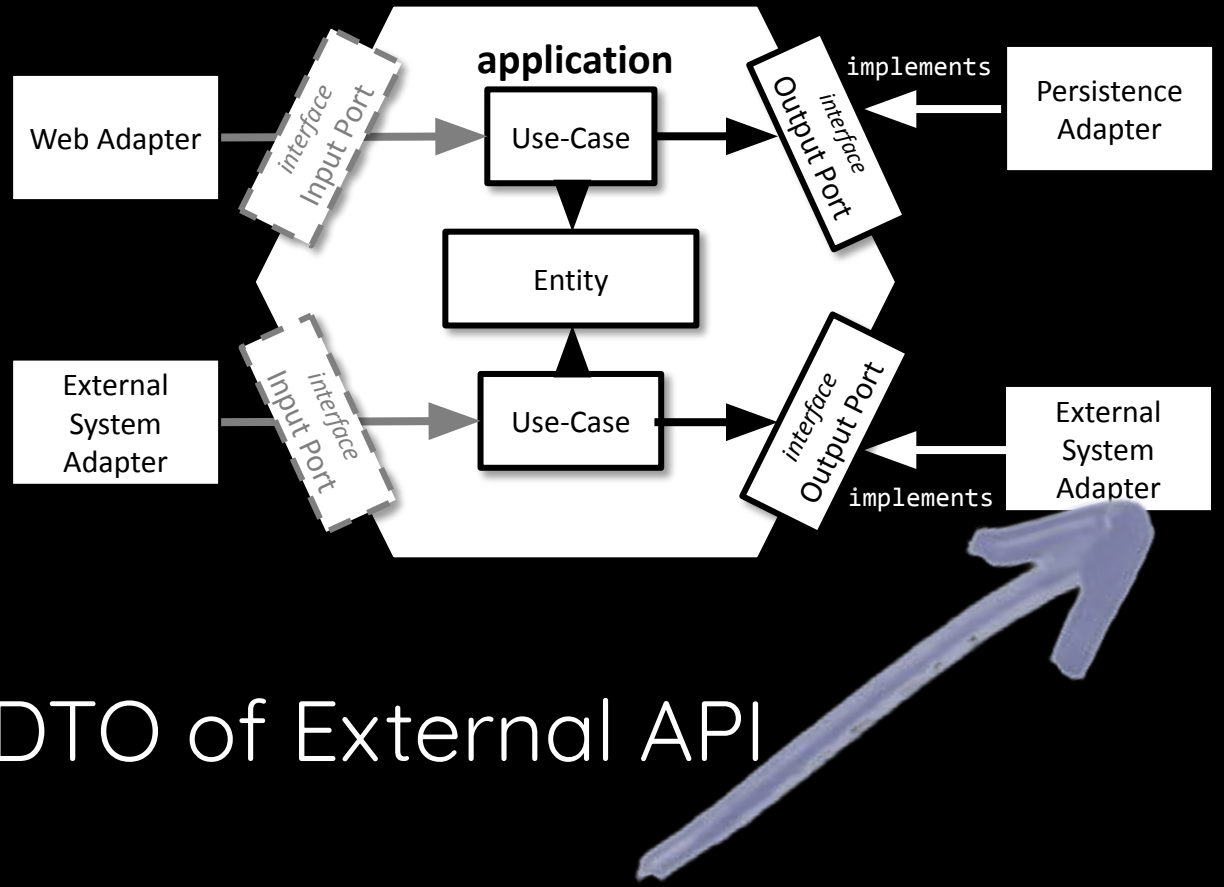
⚠️ Select **TWO**

a) Implement Business Logic

b) Call an ugly API

c) Orchestrate a Use Case

d) Convert data Domain <=> DTO of External API



26) Main goals of Hexagonal Architecture?

⚠️ Select **TWO**

a) Separate read and write models using CQRS

b) Achieve high performance by using non-blocking IO

c) Isolate the domain logic from external concerns like DBs or UIs

d) Enforce layered architecture with strict access from top-bottom

e) Segregate code by direction of flow-control

14) In Onion Architecture, what can "import"/"see" what:

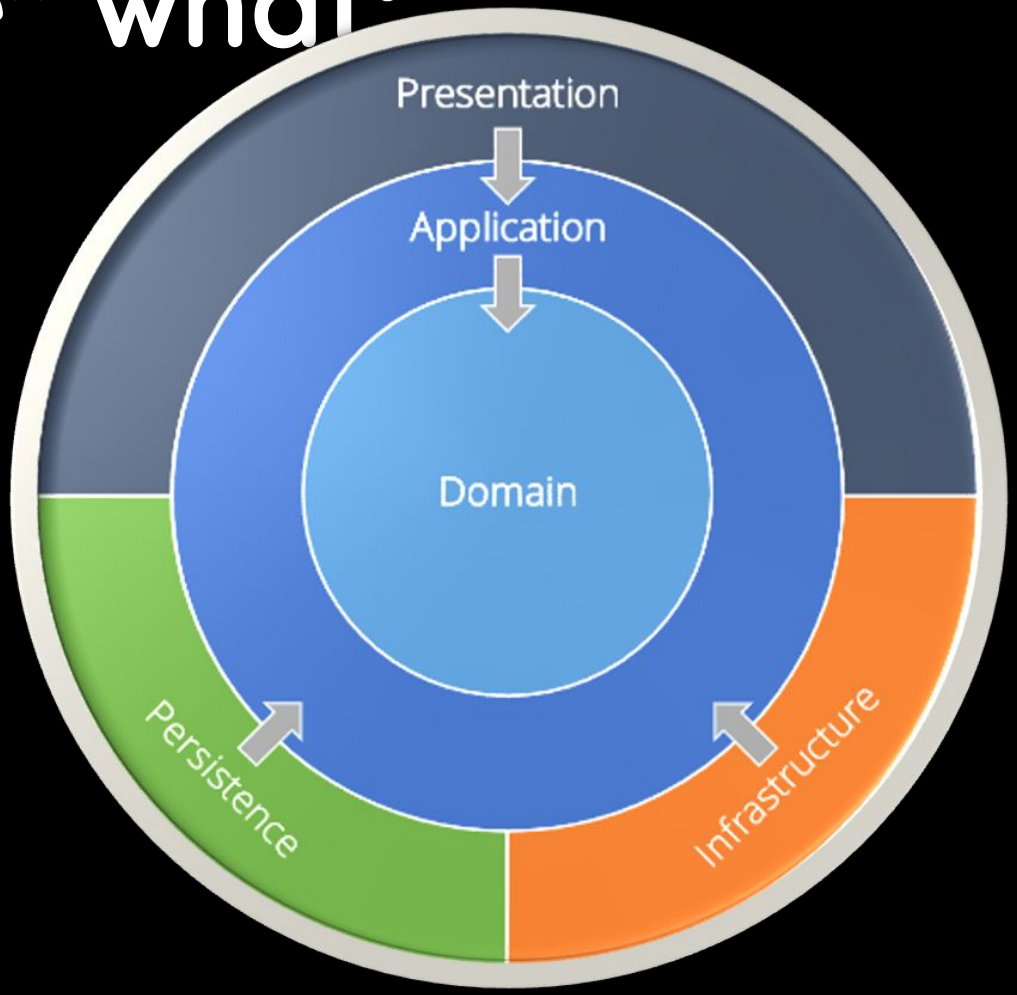
Select **ONE**

a) Application → Infra

b) Infra → Domain

c) Application □ Persistence

d) Infra □ Presentation



15) A Use-case Optimized Query ...

`SELECT new ..Dto(user.id, user.name, ...) ...`

Select **ONE**

a) Must always go through the Domain

b) Can update data directly in the DB

c) Requires a dedicated Read DB updated via events

d) Selects strictly necessary data from DB to improve performance

16) Rules of Clean Architecture by Uncle Bob

a) ⚠️ Simplify unit-testing of Domain Logic

"agnostic domain"

b) Use fewer interfaces

c) Align Domain Model with API model (DTO)

d) Keep Domain Agnostic of DB, UI, Bad Libraries, and APIs

e) Completely separate all Logic from Data Structures

17) Vertical Slice Architecture (VSA) means ...

"anti-layers"

Select **ONE**

a) Developing software driven by .feature files

b) Group code by Axis of Change, not by Layer

c) Separating code in Modules, by functional area

d) Writing the entire implementation of a feature in a single file

18) What architecture:

Select **ONE for each app**

a) onion

b) hexagonal / ports-adapters

c) layers

```
com.org.foo.          (1)
    controller
    entity
    service
    repository

com.org.gee.          (2)
    application
    input
    output

com.org.bar.          (3)
    application
    domain
    infrastructure
    persistence
```

19) Modules in a Modular Monolith ...

Select **ONE**

a) Can use any class in the other modules

b) Share the same database tables

c) Should not form dependency cycles

d) Must communicate only via Events

e) Are deployed separately

23) Minimal Steps from Module to Microservice

Select **MULTIPLE**

a) Method calls to/from other modules → network requests (REST)
b) Events to/from other modules → via message broker:Kafka, Rabbit..
c) Remove FK and JOINS to/from other modules
d) Separate its deployment
e) Transactions shared with other modules → distributed transactions (2PC)
f) Move code to a separate Git repo
g) Create a separate CI pipeline

??) Ways to fix a module dependency cycle?

a) events

b) SPI interface

c) orchestrator

d) merge

e) separation of APIs

f) shared

20) An Event

Select **ONE**

a) Has to provide all data required by listener

b) Requires some behavior in its listener(s)

c) Its name is a noun (a thing)

d) Its name includes a verb in past tense

e) Has a single consumer

21) Using events to replicate data is called:

Select **ONE**

a) Event Notification

b) Event Sourcing

c) Event-Carried State Transfer

d) Domain Events

e) Aggregate Event

22) Consumer receives a copy of current state

Select **ONE**

a) Integration Event

b) Domain Event

c) Notification Event

??) Design Techniques to decouple Modules

Few words

a) Application-level

b) Db-Level:

) Solutions to enforce design boundaries

Few words "this component should not know of that area"

) Modulith to Service-Based

= Deploy Modules as separate deployables

a) Why: indep_deploy, fault-tolerance, diff resources,kt,audit/regul

b) TODOs: calls->REST, event->Kafka..,

??) Service-Based to Microservices

= Split the database

a) Why: shared tx±, overload, crash

b) TODOs: rem FK, VIEWs, ShareTx

24) How to Maintain Distributed Consistency

In a microservice architecture: Select **MULTIPLE**

a) Human intervention: support ☎️

b) Compensating transactions = "undo" on a later failure

c) Two-Phase Commit (2PC) = distributed transaction DB_1+DB_2

d) Reservations before irreversible steps

e) Infinite retry of a call with a for loop

f) Convert a REST call into a message

25) Is Idempotent to Reprocess:

Select **MULTIPLE**

a) GET /order/{id}

b) POST /stock/{productId}/add {quantity: 3}

c) OrderCreatedEvent {orderId, orderLines:[], ..} on a topic

d) SendNotification {email, message} on a queue

e) POST /order/{id}/status {newStatus: "ACTIVE"}

Correct Answers:

1. 3 Laws: bde
2. Rich DM: bdf
3. VO: be
4. Ubiquitous: d (few=sharper DM)
5. 3 Axis Complexity: ade
6. Layered: d
7. Layered2: d
8. CQRS: b
9. SLAb: ad
10. Facade: ad
11. DIP: bd
12. Hexagonal: c
13. Output Adapter: bd
14. Onion: b
15. UCOO: d

Broad Conclusions

- Split complexity in loosely coupled pieces
- Specialize names to keep things small
- Separate high level orchestration vs complex details (SLAb)
- Separate stable contract from evolving implementation
- Dependency invert domain-irrelevant technicalities
- Split large interfaces/UIs in feature sets (ISP)
- Keep it Simple (KISS), be pragmatic, challenge religious architectures
- Split concepts in Bounded Contexts to keep Domain Model small
- Enrich your Domain Model with bits of logic, rules and Value Objects

Architecture Styles

- Big-Ball-Of-Mud Monolith ☠️
  - "It works > Ship it now, refactor *later*" for 5-25 years
- Domain-Driven Design (Eric Evans, 2003) link
  - Design Rich Domain Model for complex problems;
  - Split solution in Bounded Context
- Onion/Clean Architecture (Jeffrey Palermo, Uncle Bob 2008) link
  - Keep core Domain Logic agnostic from UI, APIs, DB, frameworks (by dep. inversion)
- Pragmatic Onion link
  - Merge application with infrastructure, evolve layers as needed, tolerate ORM on Domain Model
- Microservices (2011)
  - Loosely coupled, independent business capabilities deployed separately
- Distributed Monolith ☠️ link
  - Tightly coupled microservices due to premature, too fine-grained breakdown
- Vertical Slice Architecture (Jimmy Bogard, 2018) link
  - Group code by feature, not by technical layer
- Modular Monolith, aka Modulith (Simon Brown, 2018) link
  - Implement business capabilities as loosely coupled modules in a single deployment unit

More Reading

  - ⭐ Nice overview + picture ⭐ https://herbertograca.com/2017/11/16/explicit-architecture-01-ddd-hexagonal-onion-clean-cqrs-how-i-put-it-all-together/
  - Comparison of Layered, Onion, vs Vertical Slices vs Components, *Simon Brown*: https://youtu.be/5OjqD-ow8GE
  - Architecture Decision Records (ADR): https://www.agilealliance.org/resources/experience-reports/distribute-design-authority-with-architecture-decision-records/
- Hexagonal Architecture aka. Ports-and-Adapters
  - Colorful explanation, *8th light*: https://8thlight.com/insights/a-color-coded-guide-to-ports-and-adapters
  - Original article *by Alistair Cockburn*, 2005: https://alistair.cockburn.us/hexagonal-architecture/
- Onion Architecture
  - Clean Architecture *Uncle Bob*: https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
  - Onion Architecture Primer: https://marcoatschaefer.medium.com/onion-architecture-explained-building-maintainable-software-54996ff8e464
  - Original article, *Jeffrey Palermo*, https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- CQRS
  - CQRS overview and variations: https://martinfowler.com/bliki/CQRS.html
  - CQRS flavors: https://learn.microsoft.com/en-us/azure/architecture/patterns/cqrs
  - hard-core CQRS https://udidahan.com/2009/12/09/clarified-cqrs/
- Vertical Slice Architecture (aka -UseCase)
  - Feature slicing *Jimmy Bogard* talk: https://youtu.be/5kOzZz2vj2o and short intro article: https://jimmybogard.com/vertical-slice-architecture/
  - Heuristics to find boundaries: https://www.dddheuristics.com/design-heuristics/
  - Organize by features, not by entities, *Adam Ralph*: https://youtu.be/tVnIUZbsxWI
- Modular Monoliths (Modulith)
  - Majestic Modular Monolith, *Axel Fontaine*: https://youtu.be/BOvxJaklcr0
  - Local Complexity vs Global Complexity in microservices: https://vladikk.com/2020/04/09/untangling-microservices/
  - Spring Modulith https://www.infoq.com/news/2022/11/spring-modulith-launch/
  - Demo repo C# https://github.com/kgrzybek/modular-monolith-with-ddd
- Task-based UI:
  - https://codeopinion.com/decomposing-crud-to-a-task-based-ui/
  - https://cqrs.wordpress.com/documents/task-based-ui/

Read an IT Book 📖

(vs a random SO/article)

+ **Explains fundamentals**
+ **Guided journey** + **overview**.
+ **Single vision** of 1 person, **No conflicting opinions** 🧘
± You can polish/tweak that vision later.
- **Waste of time** if irrelevant😢.

Best way to read (IT) books in 2025

*Hey AI, what are 3 takeaways from **SRE Handbook?***

(I already read book X and Y.) =>

Expand on bullet #3! =>

⭐ How to grow your Architecture Skills? ⭐

- **Redo all changes to code I did** with friend(s), following Git history;
- Take the quiz at the end of this slide deck.
- **Change mindset** from **code typist** to a **problem solver** (±coding)
- **Have team design meetings:** /quarter or /sprint in new project [in bytesize format]
- **Articulate design** in concise **ADRs**, Wikis & Diagrams (.puml, draw.io..)
- Learn to ask the right **questions**, **listen empathetically**, understand **context**
- **Brainstorm decisions & tradeoffs**, workout your communication skills
- **Be available for anyone** seeking design advice/debate🤗 ⭐
- **Expose to other projects/teams**: talk, pair, request a [temporary] rotation↺
- **Keep high-level**, until a "sufficient" level of detail; don't be a control-freak!
- **Defend simplicity**, w/o hijacking foreseeable extensibility
- **Practice an** Architecture Kata with a friend: nealford.com/katas

Premium Words & Concepts

to impress your boss/interviewer with a deep voice

- **Agnostic** Domain = core logic is kept isolated from external world
- **Anti-Corruption Layer =** protects my precious logic from the evil outside
- **Level of Abstraction** = level of detail, granularity
- **SRP** = Single Responsibility Principle
- **DRY** = Don't Repeat Yourself
- **KISS** = Keep It Simple, Stupid; go from specific to generic
- **Cyclomatic Complexity** = # of independent execution paths through some code ~ # of tests
- **Ubiquitous Language** = terms have clear meaning inside the **Bounded Context** (dev+biz+test)
- **Orchestration** = tell others what to do (like-a-bossa🎶)
- **Segregation** = ~~separation~~ -- Architecture is the art of drawing lines
- **CQRS** = Command(Write) Query(Read) Responsibility **Segregation**
- **DDD** = Domain-Driven Design = craft your code around the problem
- **Domain Model** = representation of data objects involved in core complexity; kept internal
- **Code Volatility** = rate of change
- **Cohesion** = what changes together should stick together -> VSA
- **CDC**: Change-Data-Capture: transactions auto-published as Kafka Messages
- **Conway's Law** = code will reflect organizational communication structures

Further Reading

- Coding Basics
  - Clean Code⭐ - coding practices/principles
  - Head-First: Design Patterns – Classic OOP "GoF" patterns: Adapter, Facade, Observer ...
- Architecture
  - Free Video Lessons by Mark Richards: 200 x ~10 min videos, perfect for commuting 🚆🚗🚋🚌
  - Head-First Architecture – A gentle intro to arch
  - Architecture: Fundamentals (=NFRs: Scalability,...) and The Hard Parts (=Distributed Systems)
  - Domain Driven Design – Distilled = DDD for complex domains
  - martinfowler.com = uncontestable articles❤️
  - Awesome System Design Articles + System Design Interview Book
  - Site Reliability Engineering by Google (free) - Ops
  - Designing Data-Intensive Applications⭐ by Martin Kleppmann
  - Release It by Michael Nygard
  - Building Microservices by Sam Newman
- Certifications: Cloud (AWS, GCP ...), General Architecture (isaqb)
- Tech Culture, Practices, Teamwork, Psychology of Geeks
  - Extreme Programming💖 – geek-friendly Agile approach
  - The Software Craftsman – how/why to fight for code quality
  - Pragmatic Programmer – dev culture, productiv ity
  - Apprenticeship Patterns – mentoring and continuous growth
  - Presentation Patterns – shaping and presenting your ideas

- Keep up to date
  - Oskar Dudycz Substack
  - Kent Beck Substack
  - Vlad Mihalcea Newsletter
  - InfoQ Architecture Newsletter
  - dzone refcards