



A Practical Guide to Collaborative Software Design



A Practical Guide to Collaborative Software Design

2025 edition - Session 1



How it was supposed to be



Reality



1 Architecture and Software Design

2 Mindset and practices

Today

3 Models

4 How to Design

5 Next Steps &
Further reading and viewing...



In theory, there is
no difference between
theory and practice; In
practice, there is.



Software Engineering: it's a Craft, should be a Culture and needs to be about People

Manifesto for Software Craftsmanship

Raising the bar.

Aspiring Software Craftsmen we are raising the bar of professional software development by practicing it and helping others learn the craft. Through this work we have come to value:

Not only working software,
but also **well-crafted software**

Not only responding to change,
but also **steadily adding value**

Not only individuals and interactions,
but also **a community of professionals**

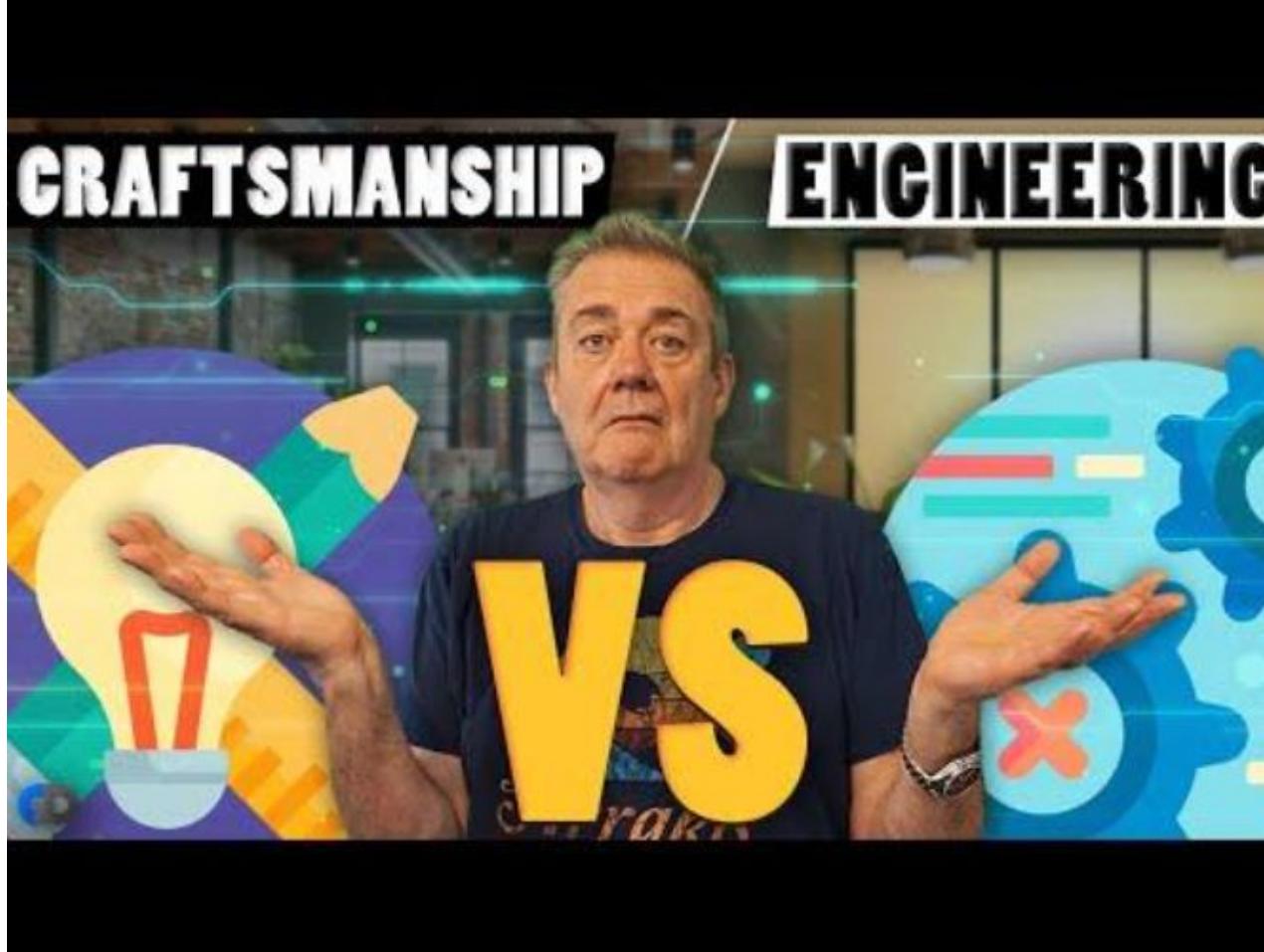
Not only customer collaboration,
but also **productive partnerships**

That is, in pursuit of the items on the left we have found the items on the right to be indispensable.





Software Engineering: it's a Craft, should be a Culture and needs to be about People



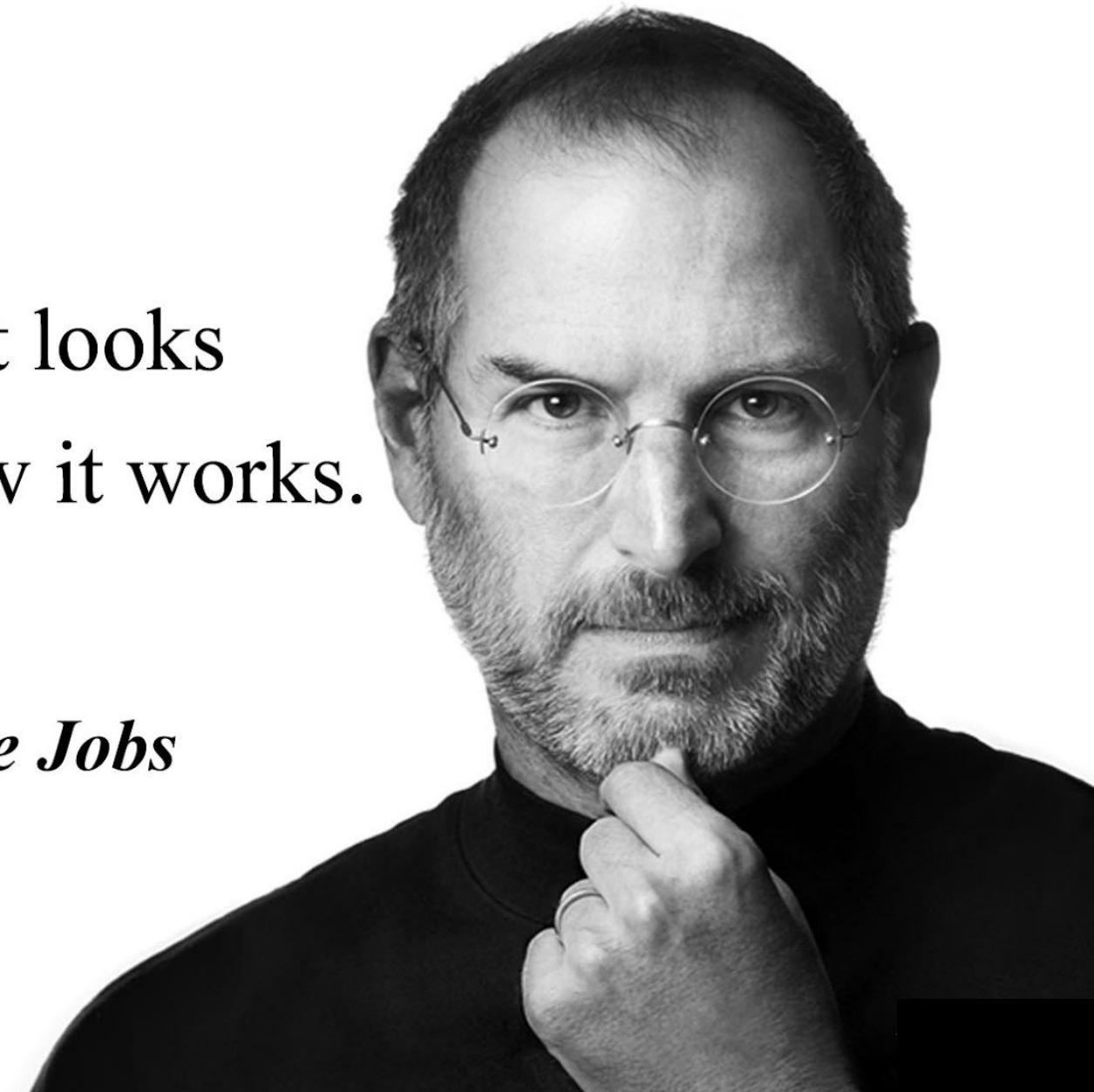
Software Craftsmanship vs Software Engineering

[youtube.com](https://www.youtube.com)



Design is not just what it looks
and feels like. Design is how it works.

Steve Jobs



If you are in a hurry.....The lost art of software design by Simon Brown



DEVOXX
BELGIUM

Continuous attention to
technical excellence and
good design enhances agility.

Principle 9 of the Manifesto for Agile Software Development

Mininlayer (i)

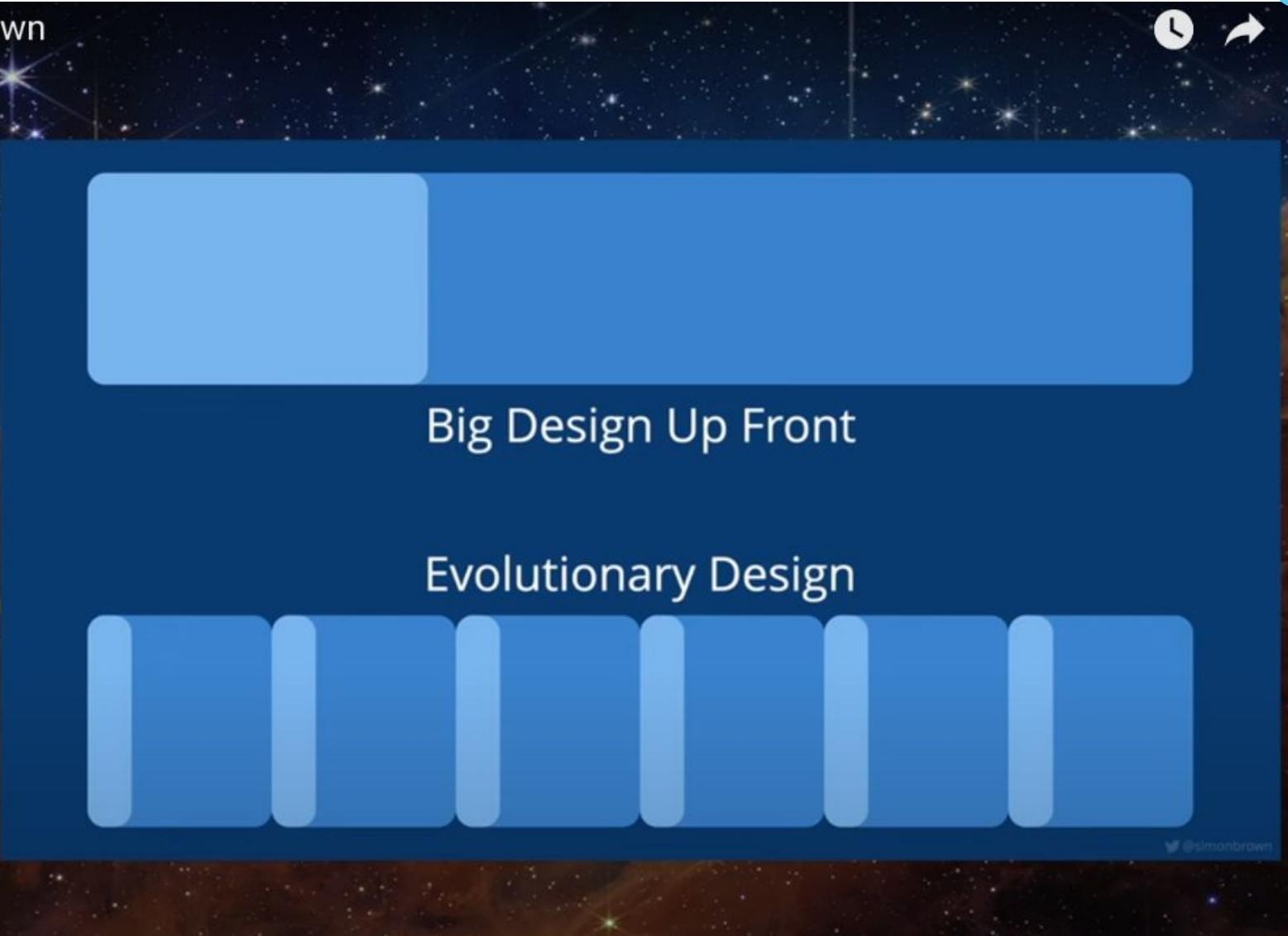


<https://youtu.be/36OTe7LNd6M>

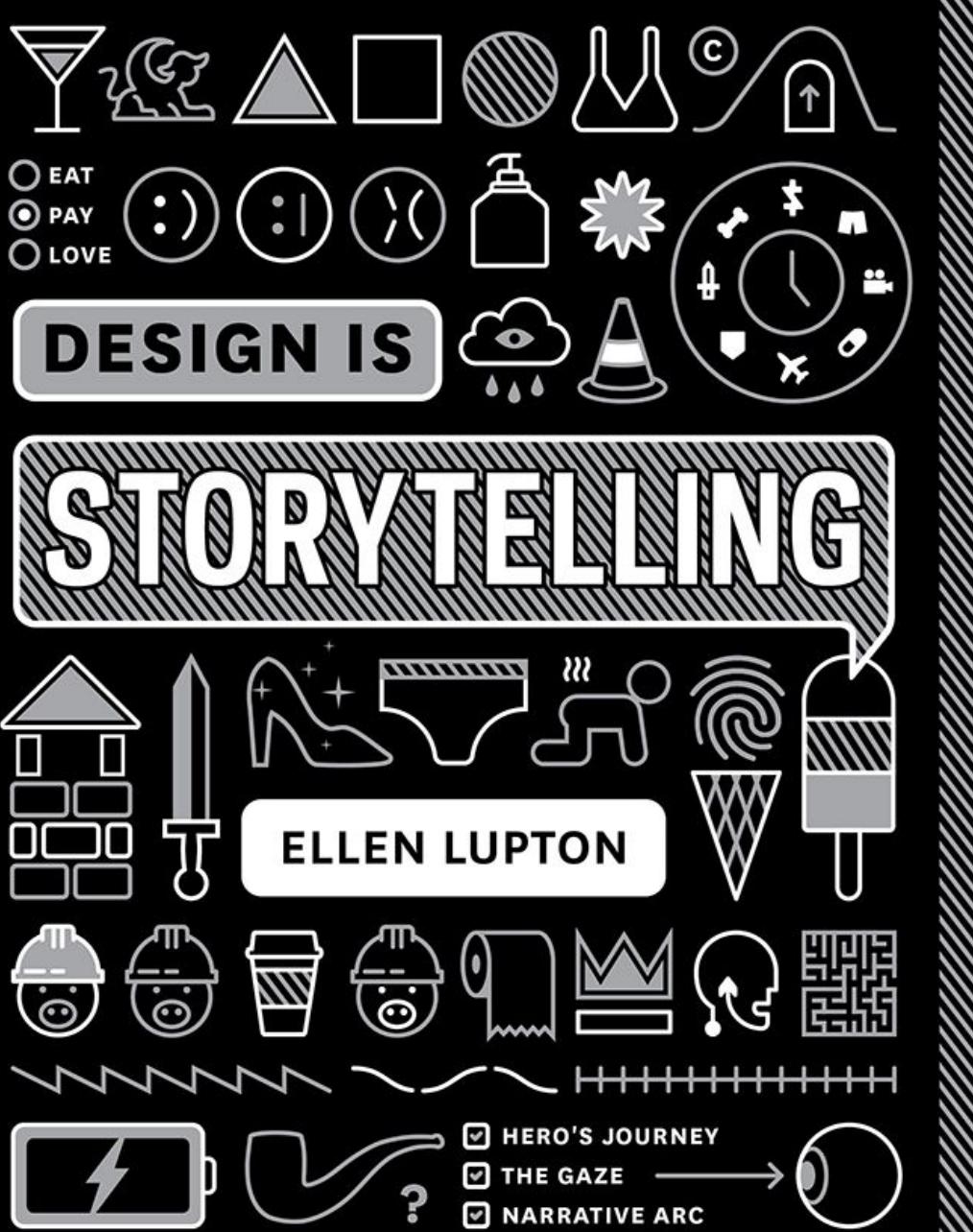
If you are in a hurry.....The lost art of software design by Simon Brown



The lost art of software design by Simon Brown



<https://youtu.be/36OTe7LNd6M>

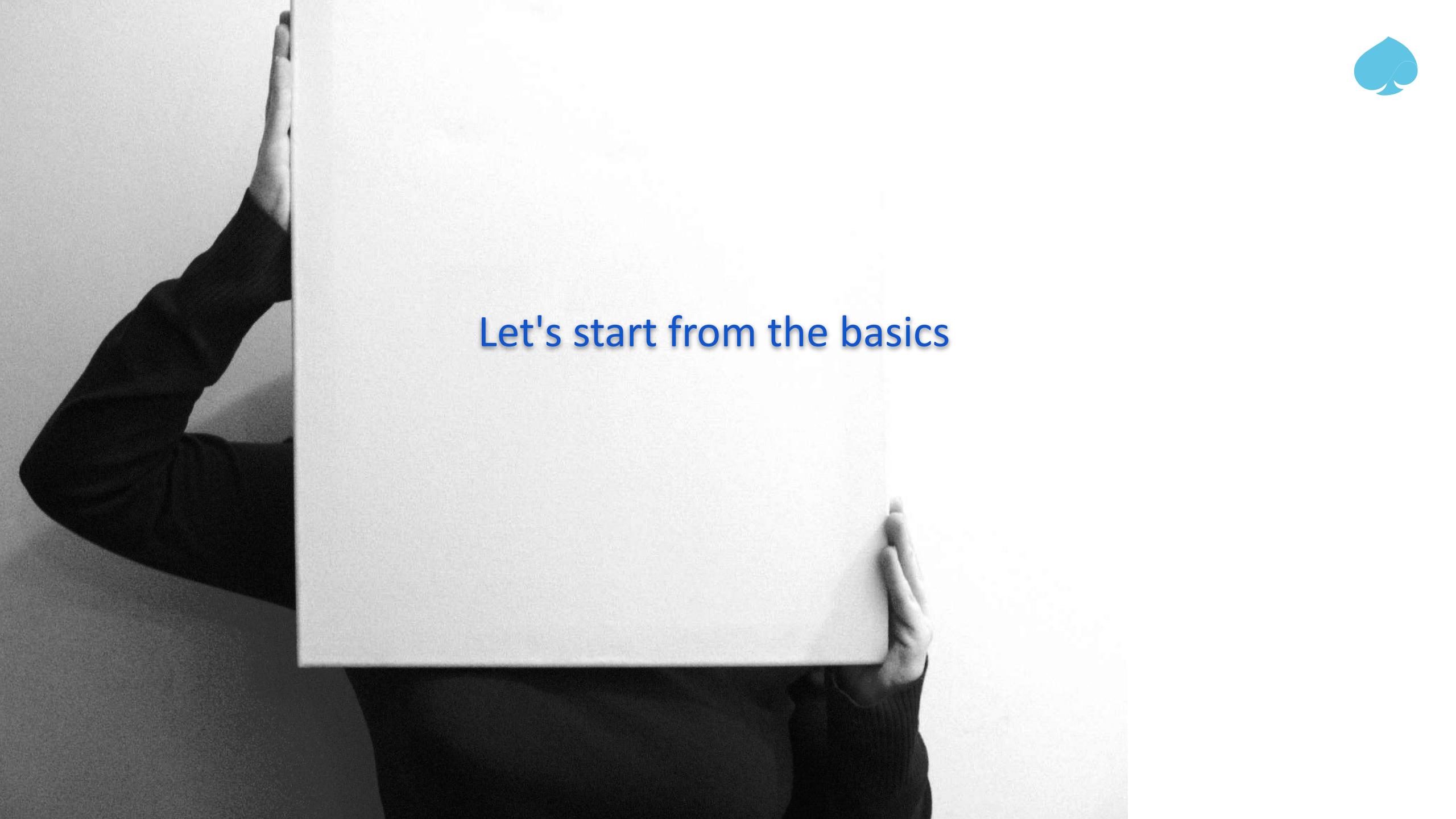


Design is describing
- telling a story -
about How “it” works

Lupton's *Design is Storytelling* reminds software engineers that **humans connect with systems and interfaces through stories**—whether explicit or implicit.

For example: In software, user flows are like narratives. They take users on a journey with a beginning (onboarding), middle (tasks), and end (success or feedback).

By treating software experiences as narratives, engineers and designers can create products that are not only functional but also intuitive, memorable, and engaging.

A black and white photograph showing a person from the chest up. They are wearing a dark long-sleeved shirt. Their hands are visible at the bottom, holding a large, blank sheet of paper in front of their face, completely obscuring it. The background is a plain, light color.

Let's start from the basics



Architecture and Software Design



Defining the terms...

A Practical Guide to

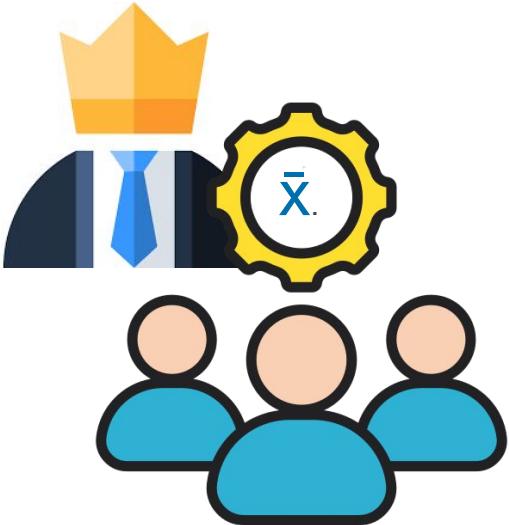
Collaborative

Software Design

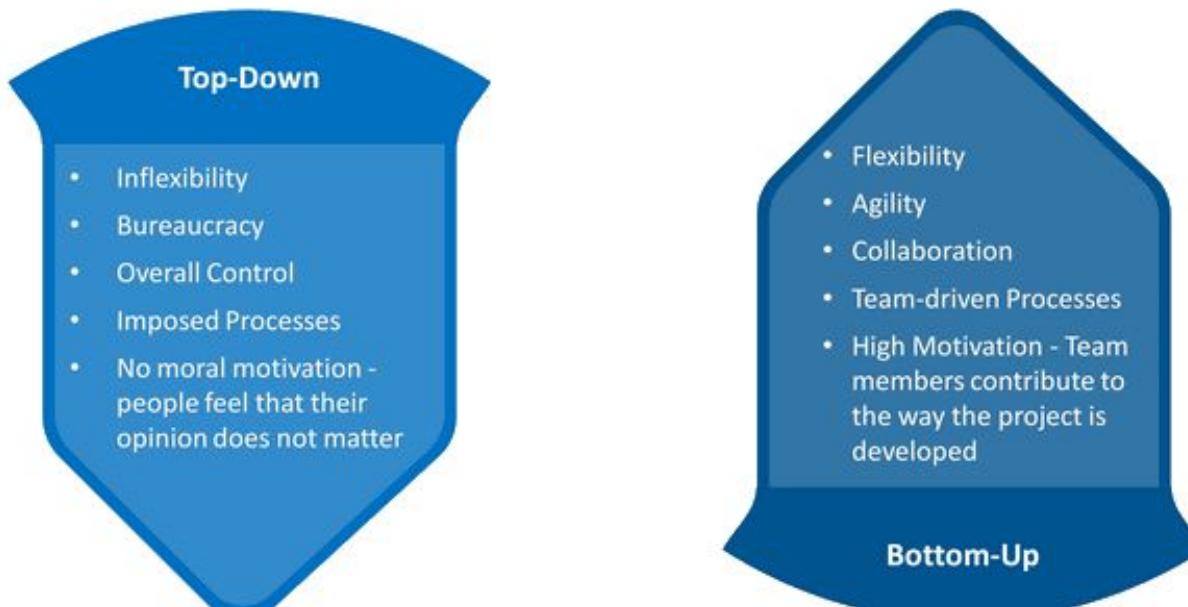


Collaborative?

TOP-DOWN VS BOTTOM-UP APPROACH



An strongly led, average, baseline team will suffice



Requires a strong, motivated team, capable and inclined to work peer to peer

Control and
Collaboration

Clarity of Project goals and visibility
of internal organization processes

Coordination and
Collective Intelligence



The Team should (be able to) work as one



Defining the terms...



A Practical Guide to

Collaborative

Software Design

An important side note:



Software Design is not separate from User Experience Design

USER EXPERIENCE IS...

LOOK + FEEL + USABILITY



UX Design is PART OF Software Design, part of the process as well as the final artifacts. It is NOT a separate process in the same way that designing the Data Model is NOT a separate process.



What *is* Software Design? Is it “Architecture”? What *is* “Architecture”?



The poster is red with a black border. At the top left is the O'REILLY OSCon logo. Below it, text reads "PORTLAND, OR JULY 20-24, 2015 oscn.com #oscon". A cartoon illustration of an owl wearing glasses and a bow tie is on the left. In the center, a dark gray rectangular box contains the title "Making architecture matter" and the speaker information "Martin Fowler ThoughtWorks".

<https://www.youtube.com/watch?v=DngAZyWMGR0>

Ralph Johnson (one of the Design Patterns Gang of Four (GoF) – as paraphrased by Martin Fowler:

“....Expert developers’ shared understanding of the system design

...

very much a social activity

...

The set of design decisions that are hard to change

...

Which boils down to”

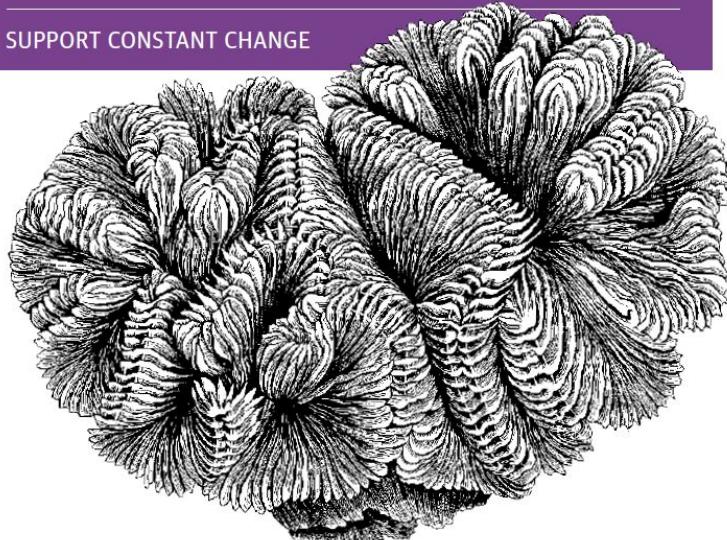


Software Architecture is

O'REILLY®

Building Evolutionary Architectures

SUPPORT CONSTANT CHANGE



Neal Ford, Rebecca Parsons & Patrick Kua

“.....the important stuff
(whatever that is).....”

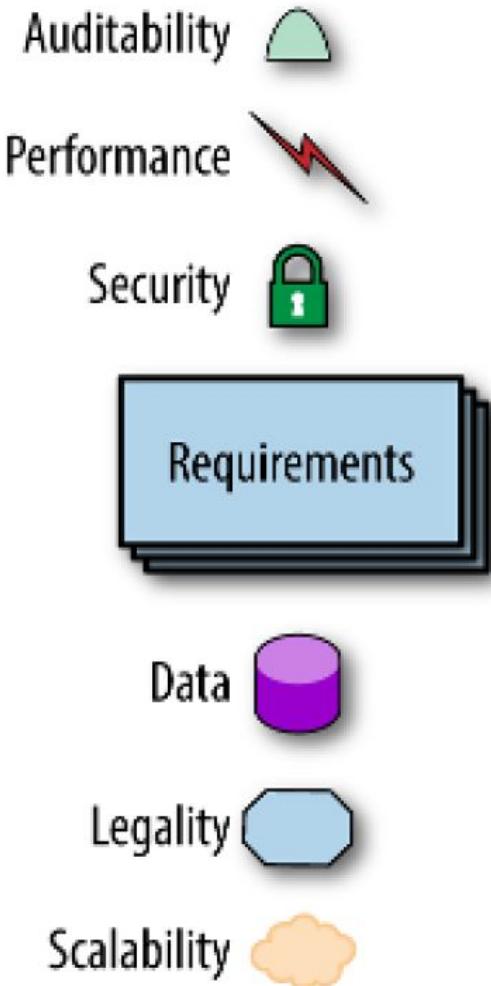
The architect's job is to understand and balance all of those important things
(whatever they are)

.....

and to define and document in such a way that all involved understand it and do so in the same way



The important stuff



accessibility accountability accuracy adaptability administrability
affordability agility auditability autonomy availability
compatibility composable configurability correctness credibility
customizability debugability degradability determinability
demonstrability
dependability deployability discoverability distributability durability
effectiveness efficiency usability extensibility failure transparency
fault tolerance fidelity flexibility inspectability installability
integrity interoperability learnability maintainability manageability
mobility modifiability modularity operability orthogonality
portability precision predictability process capabilities producibility
provability recoverability relevance reliability repeatability
reproducibility resilience responsiveness reusability robustness
safety scalability seamlessness self-sustainability serviceability
securability simplicity stability standards compliance survivability
sustainability tailorability testability timeliness traceability



Architecture and Software Design

“....Architecture is the bigger picture: the choice of frameworks, languages, scope, goals, and high-level methodologies (Rational, waterfall, agile, etc.).

Design is the smaller picture: the plan for how code will be organized; how the contracts between different parts of the system will look; the ongoing implementation of the project's methodologies and goals. Specification are written during this stage....”

Architecture => Structure

Design => Structure and Meaning



The central question

How do we **define** and
communicate
this **structure** and
meaning
of the system (to be) build?



So what is (a) Software Design

“...Software design is both a process and a model...”

- The design as a model is the “plan” of what to build
- It should define the structure of the code
- It should describe, make explicit, the functionality
- Using abstractions to arrange and reduce complexity

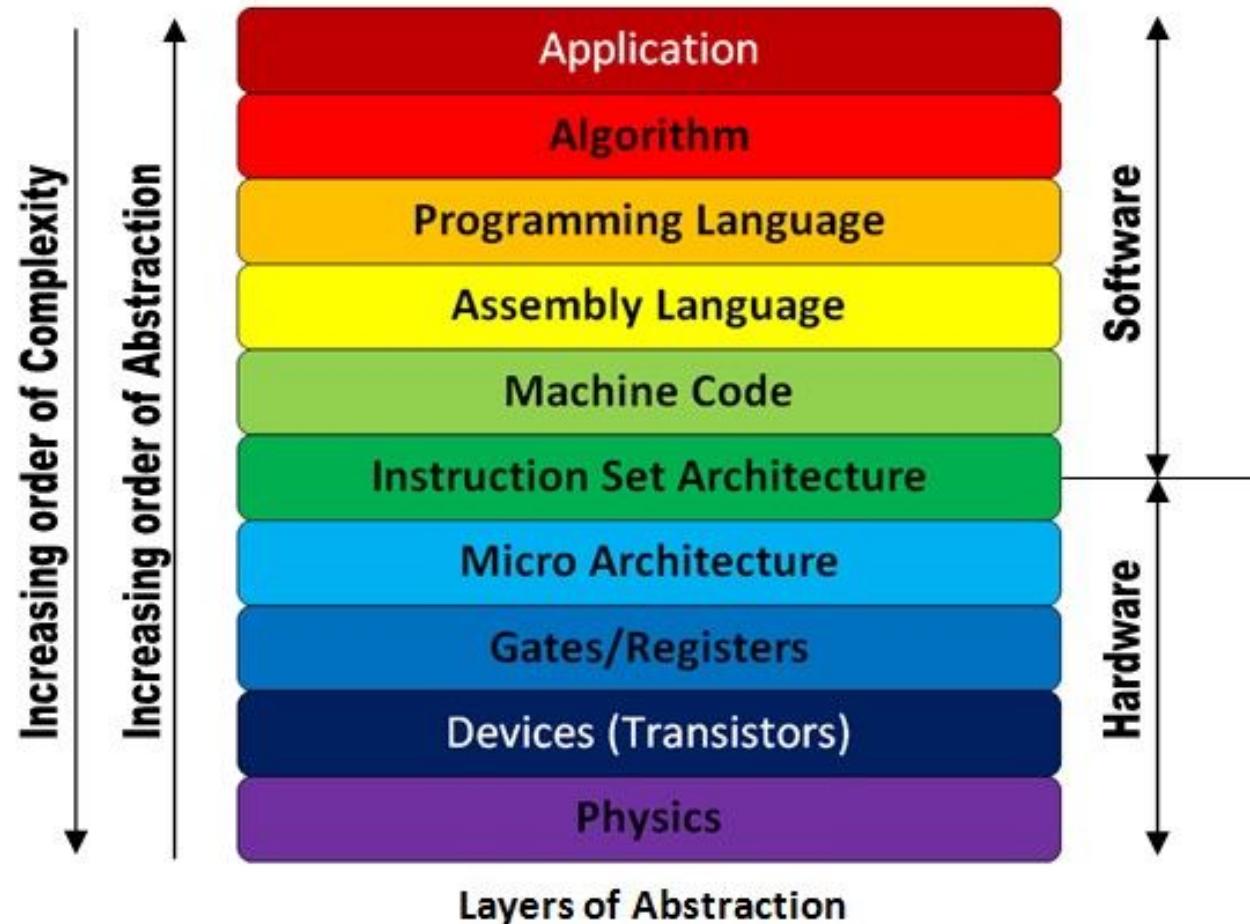


“...It's too abstract...”

- The imperative mind-set
- “...It's too vague...” or “.. too abstract..”
- Mistake low-level imperative operations for exactness
- Every programming language, even assembler, is composed of high-level abstractions



In computers, ALL is abstraction

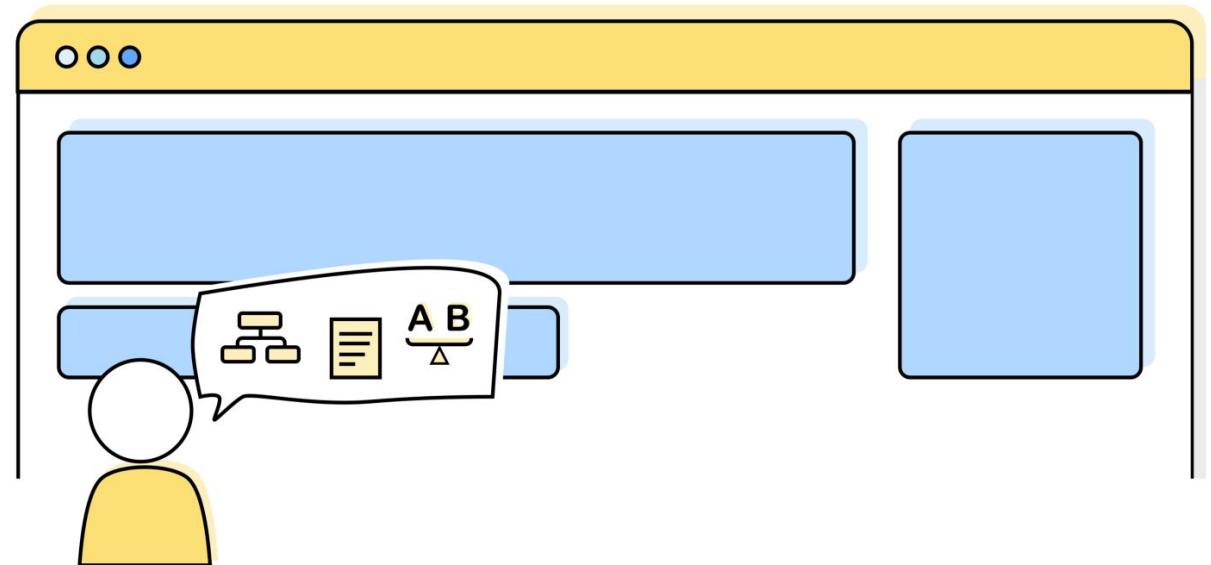




The design

A software design is based on
a composition of *abstractions*

(but what does that mean?)

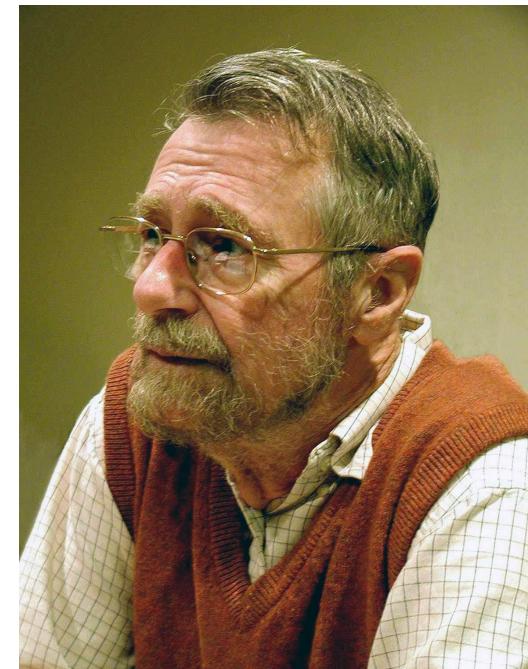




Abstraction

“...Being abstract is something profoundly different from being vague ... The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise....”

Edsger Dijkstra





How to express Abstractions

Depending the language

Types

Classes

Methods

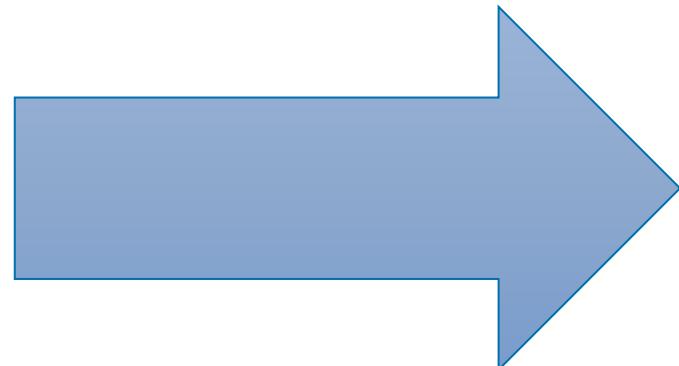
Functions

Decorators

Attributes

Modules

Macros

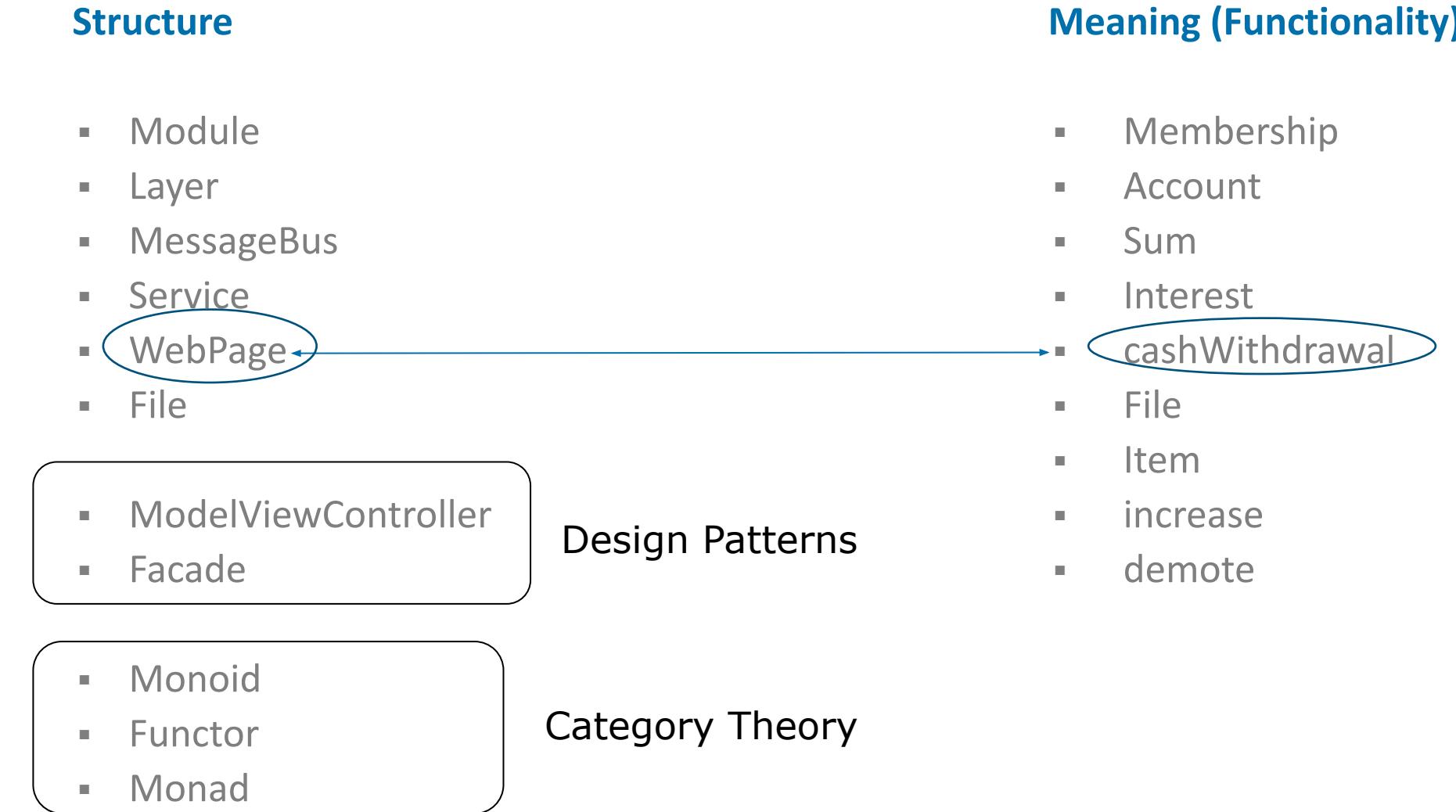


Nouns, verbs and adjectives

(and Spells, Magical incantations, through macros and embedded languages or DSL. But that is for another presentation)



Example abstractions





Don't overdo structure

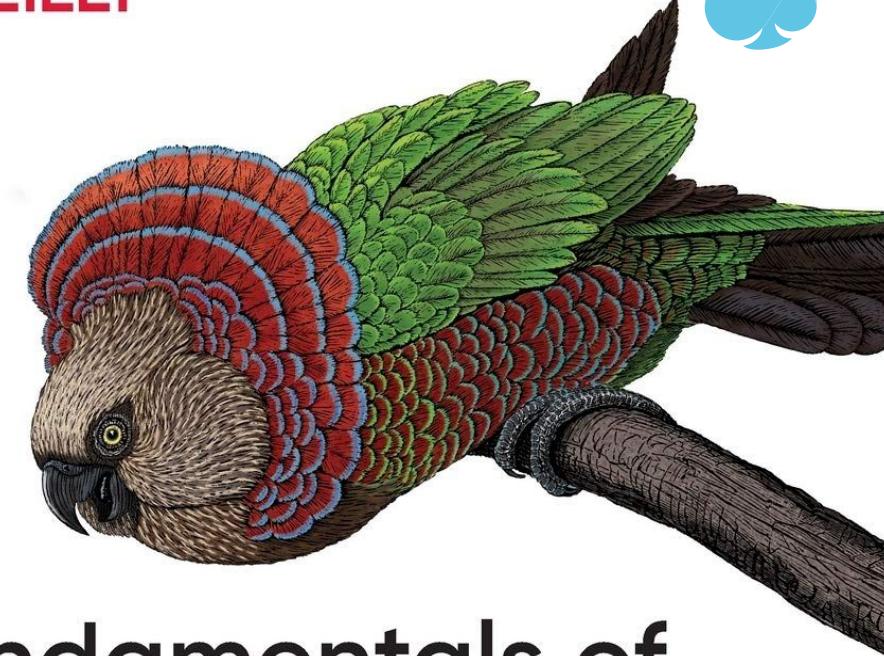
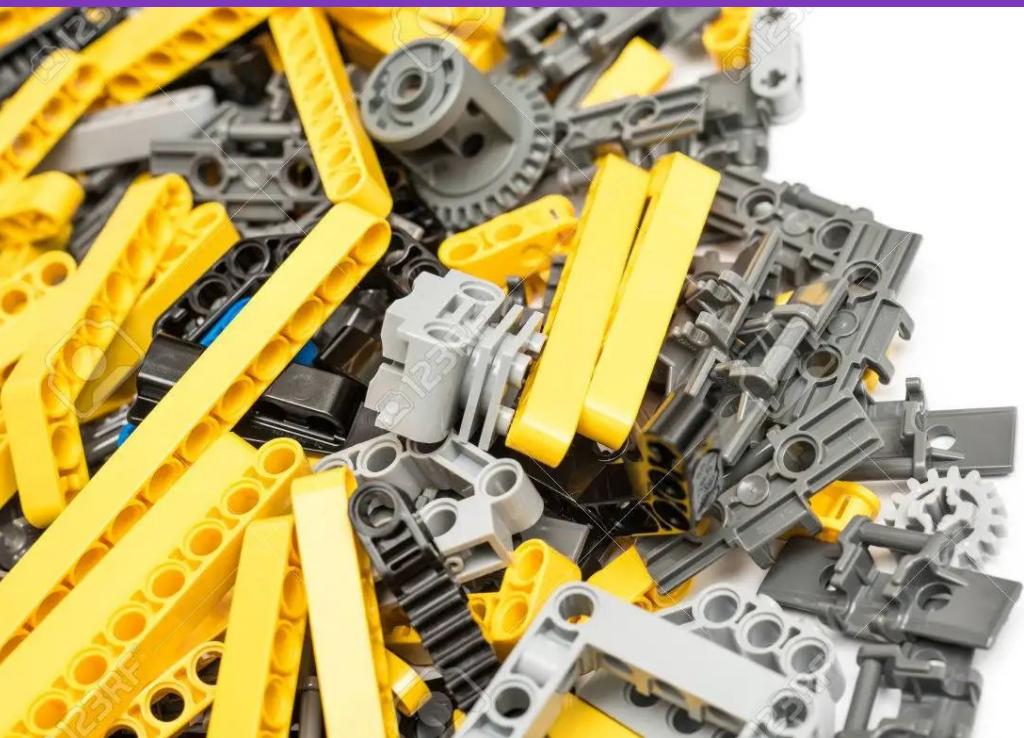
KEEP IT
SIMPLE





It's an engineering discipline...but...

You are not going to "build" physical objects. Focus on meaning and not on structure but do so using Engineering principles



Fundamentals of Software Architecture

An Engineering Approach

Mark Richards & Neal Ford



Mindset and Practices



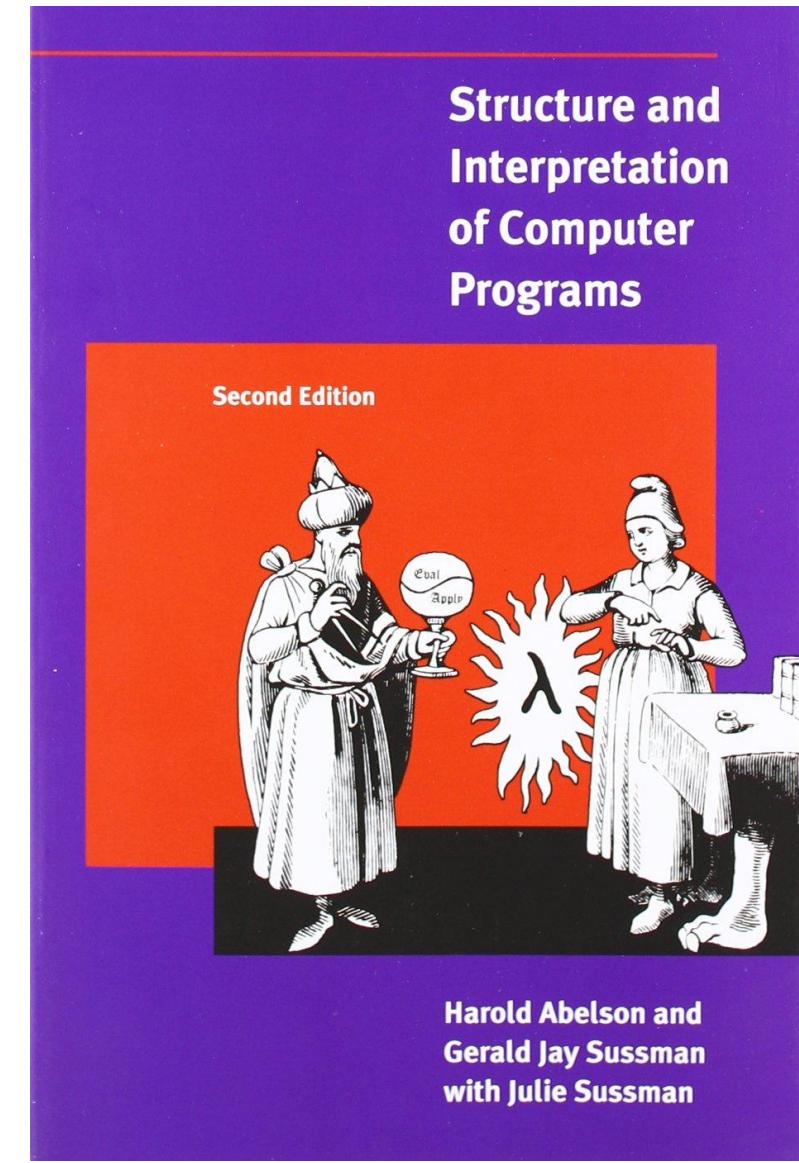
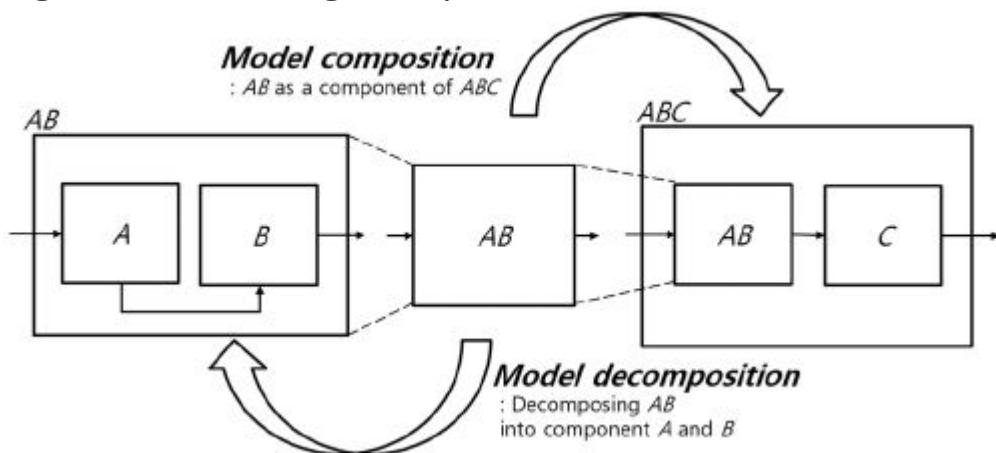
Software Design as Storytelling

An abstraction can be created and combined with others to create a larger abstraction (**composition**). Once an abstraction needs to be detailed and specific it needs to be broken up in smaller components (**decomposition**).

This is the essence of programming....

This can be seen as a technical process. However, due to the intangible nature of software and modelling, it can actually be a linguistic process. Abstractions can consist of metaphors, similes, stories.

Software Design becomes a linguistic process. It becomes akin to telling a story.





So

How do we **define** and
communicate
this **structure** and
meaning
of the system (to be) build?

By telling a story;
Software Design is
Storytelling with Abstractions



The design is relevant NOT the medium

Defining and communicate the structure and meaning
of the system (to be) build

Telling WHAT to BUILD and HOW it WORKS

Create artifacts as diverse as:

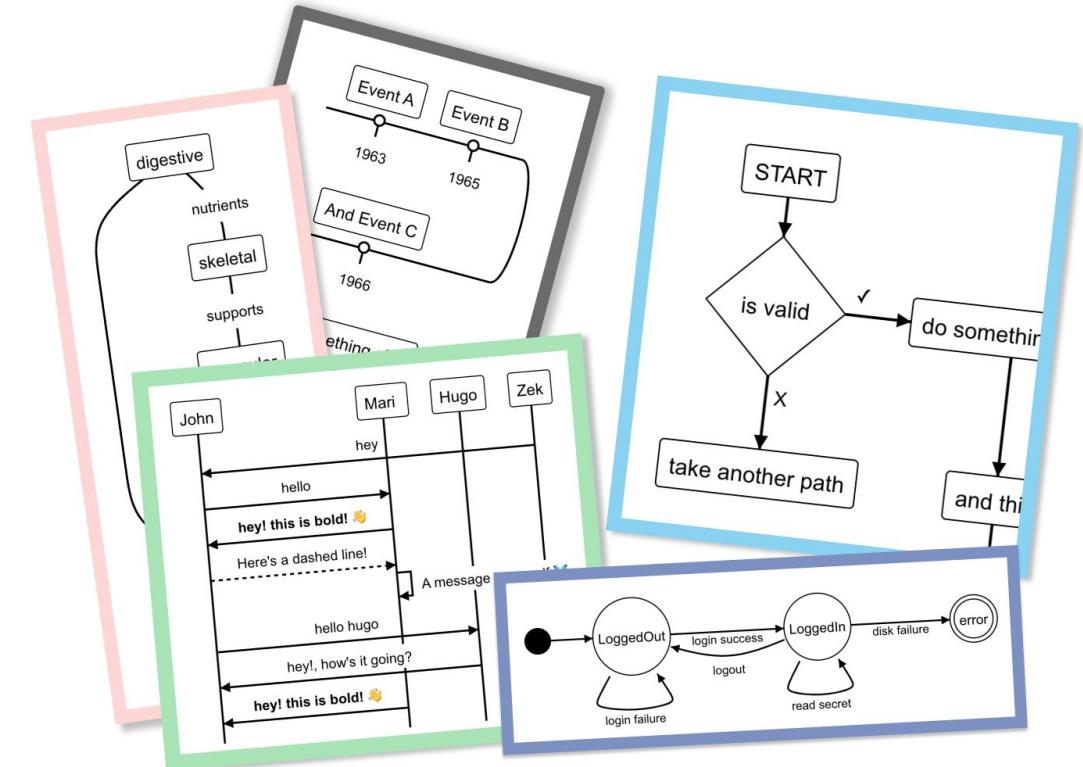
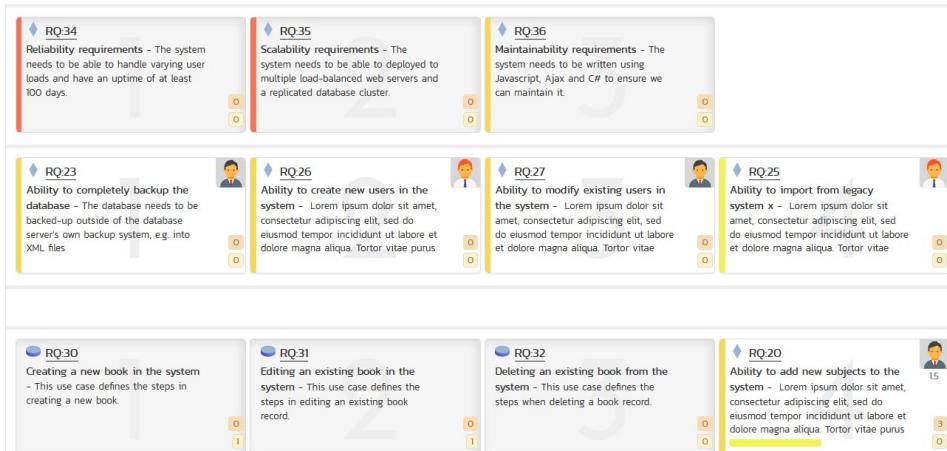
- Wireframing
- User Stories
- Requirement Specs
- Risk Logs
- Prototypes (code, paper, etc)
- Spikes (coding activity)
- Type definitions (code)
- Context Maps
- Flow Charts
- Sequence Diagrams
- etc....





Telling a story with Text, Code & Diagrams

```
1  from typing import List, Dict, Any
2
3  from agents import CoalitionAgent
4  from base import Coalition, Product
5  from communication import CoalitionAction
6
7
8  class MyCoalitionAgent(CoalitionAgent):
9      PROD_TYPE_R1 = "r1"
10     PROD_TYPE_R2 = "r2"
11
12    def __init__(self, name: str, resources: float, products: List[Product]):
13        super(MyCoalitionAgent, self).__init__(name, resources, products)
14
15    def create_single_coalition(self):
16        c = Coalition(self.products)
17        c.set_agent(self, share=self.PROD_TYPE_R1):
18        {
19            Coalition.PROD_CONTRIB: self.resources,
20            Coalition.PROD_VALUE: 0
21        }
22    }
23
24    return c
25
26    def state_announced(self, agents: List[CoalitionAgent], coalitions: List[Coalition]):
27        super().state_announced(agents, coalitions)
28
29    def do_actions(self, messages: List[CoalitionAction] = None):
30        return []
```



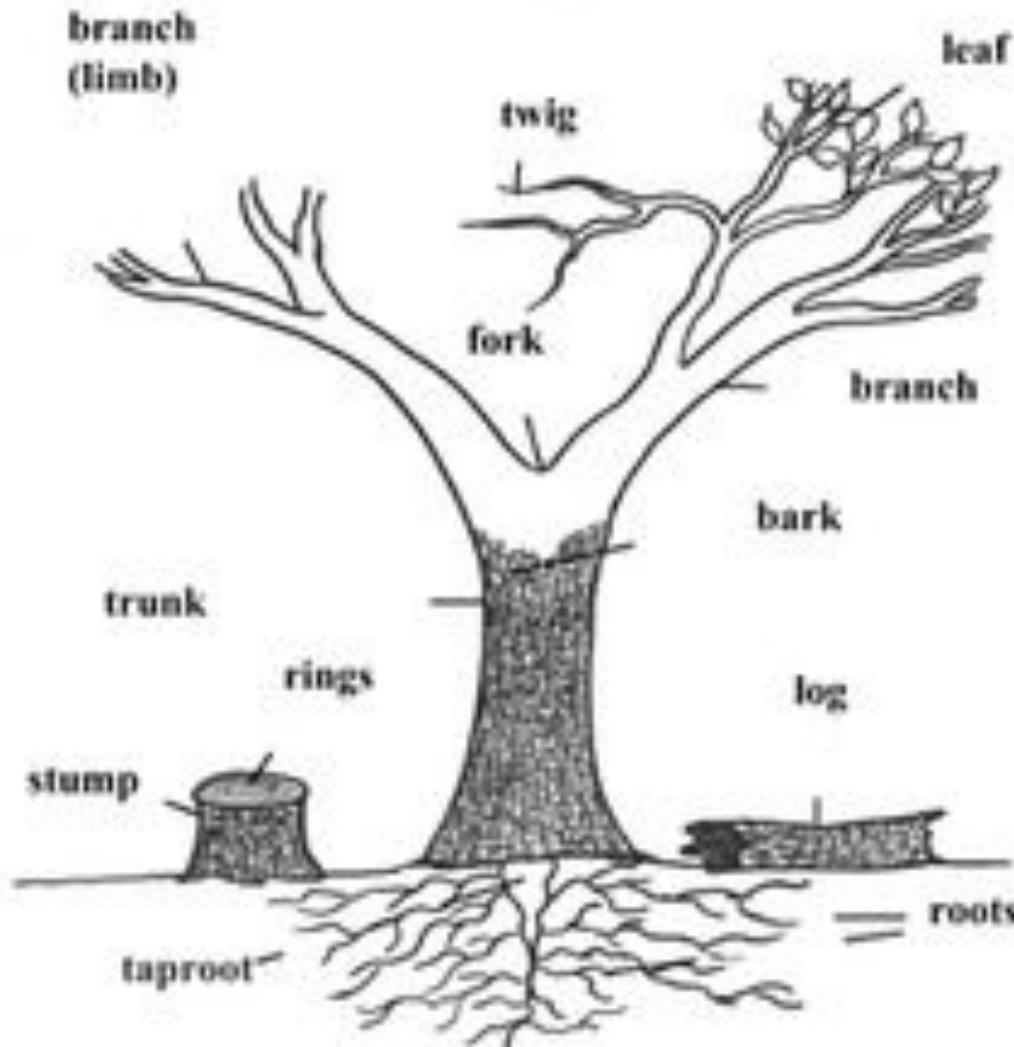


EVERYTHING HAS A NAME

Naming

“...There are only two hard things in Computer Science: cache invalidation and naming things...”

Phil Karlton (Netscape)

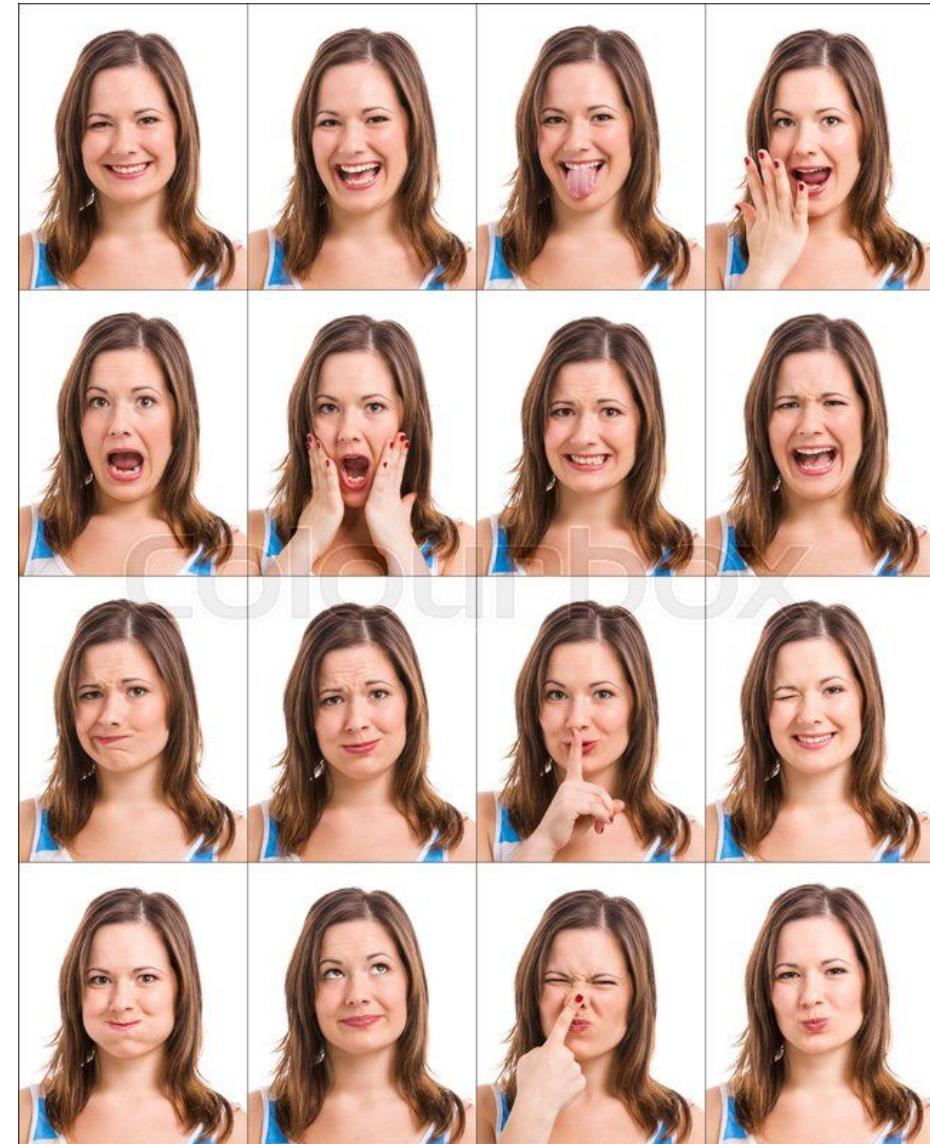


“..If you consider programming to be a subset of writing, and I certainly do

David Heinemeier Hansson (Ruby on Rails / Basecamp)



Simplifying for conciseness





Language and formulas

Gezelligheid (Dutch pronunciation: [\[yə'zɛləx̥it\]](#) ([listen](#))) is a [Dutch](#) word which, depending on context, can be translated as '[conviviality](#)', '[coziness](#)', '[fun](#)'. In Spanish it can be translate by combining terms like '[acogedor](#)', '[convivencia](#)', or even '[calidez humana](#)', which can communicate its essence in specific contexts.

It is often used to describe a social and relaxed situation. It can also indicate belonging, time spent with loved ones, catching up with an old friend or just the general togetherness that gives people a warm feeling.

A common trait to all descriptions of *gezelligheid* is a general and abstract sensation of individual well-being that one typically shares with others. All descriptions involve a positive atmosphere, *flow* or *vibe* that colours the individual personal experience in a favorable way and in one way or another corresponds to social contexts.

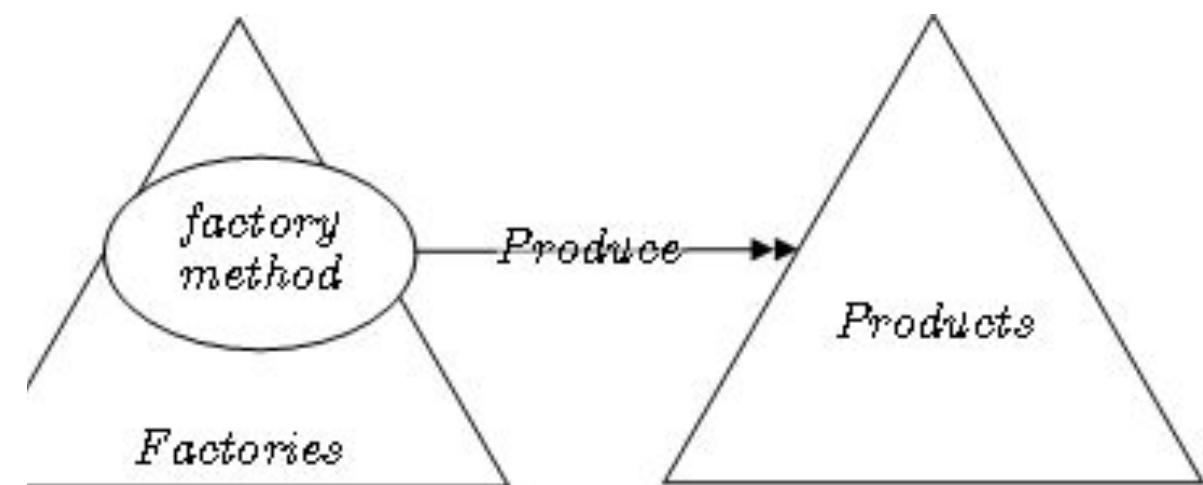
Being a vague, abstract notion, the word is considered by some to be an example of [untranslatability](#), and one of their hardest words to translate to [English](#) or [Spanish](#). Some consider the word to encompass the heart of [Dutch culture](#).^[1]

$$F_g = G \frac{m_1 m_2}{r^2}$$



Metaphor

Factory
Abstract Factory
Observer Facade
Strategy
Singleton
Adapter





Abstraction: representing *and* hiding State, Structure and Flow



state
structure
& flow

The *structure* of the code and in-memory representation.

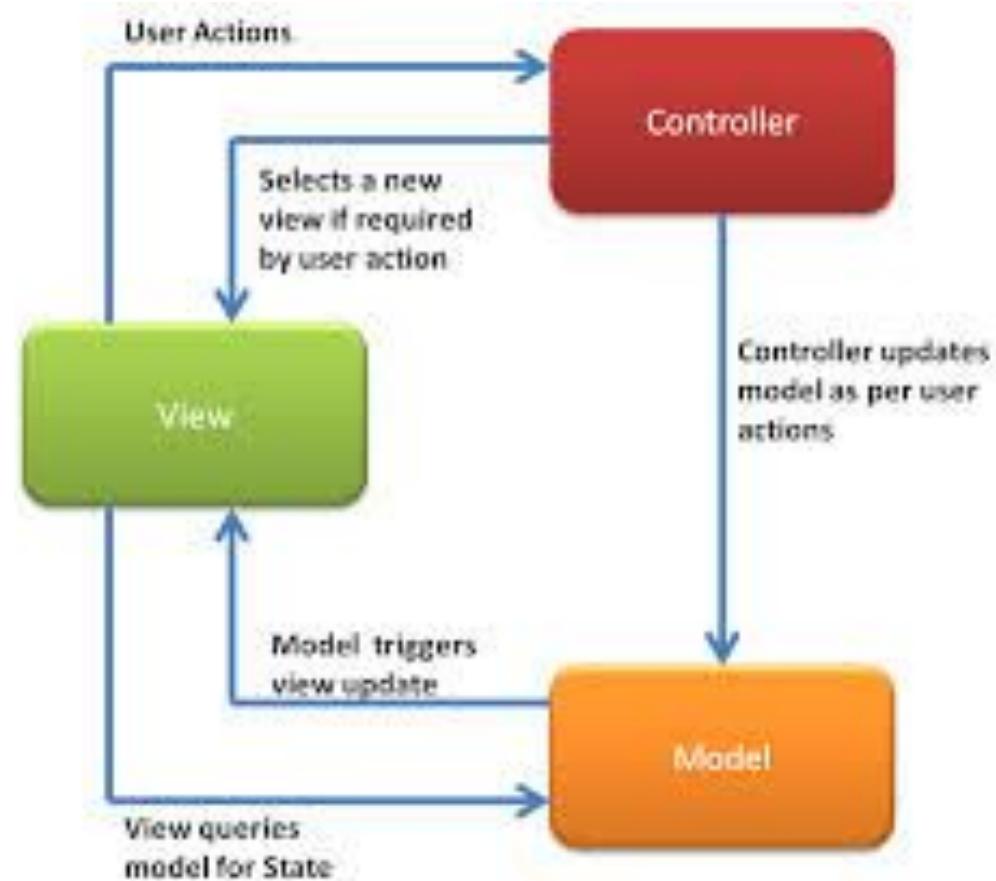
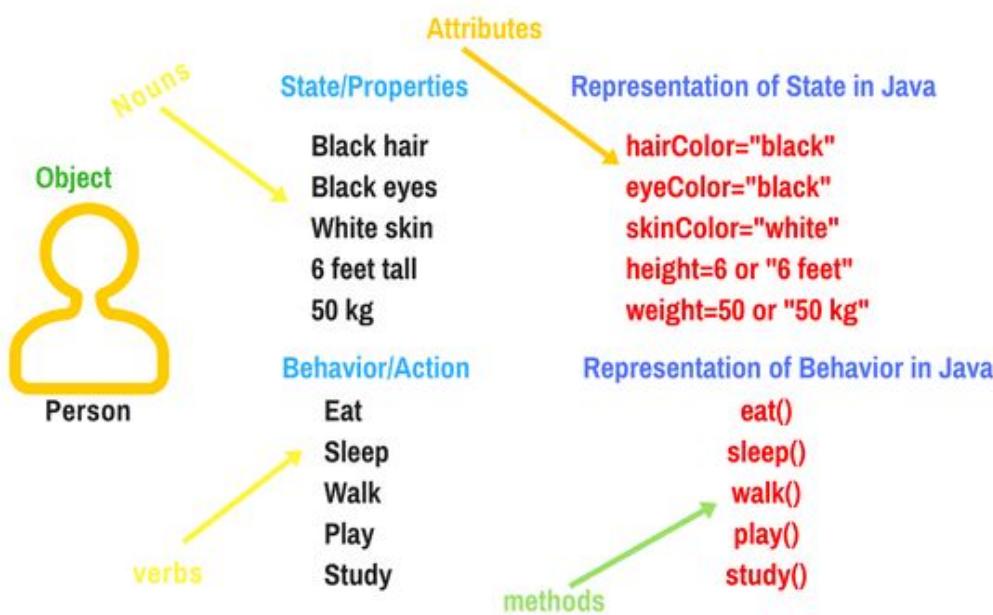
The *state* or particular condition that a program is in at a specific time. The total of all data in all variables in a particular system.

The flow or transition or transformation of one particular state to another.

Examples: Queue, Stream, Workflow

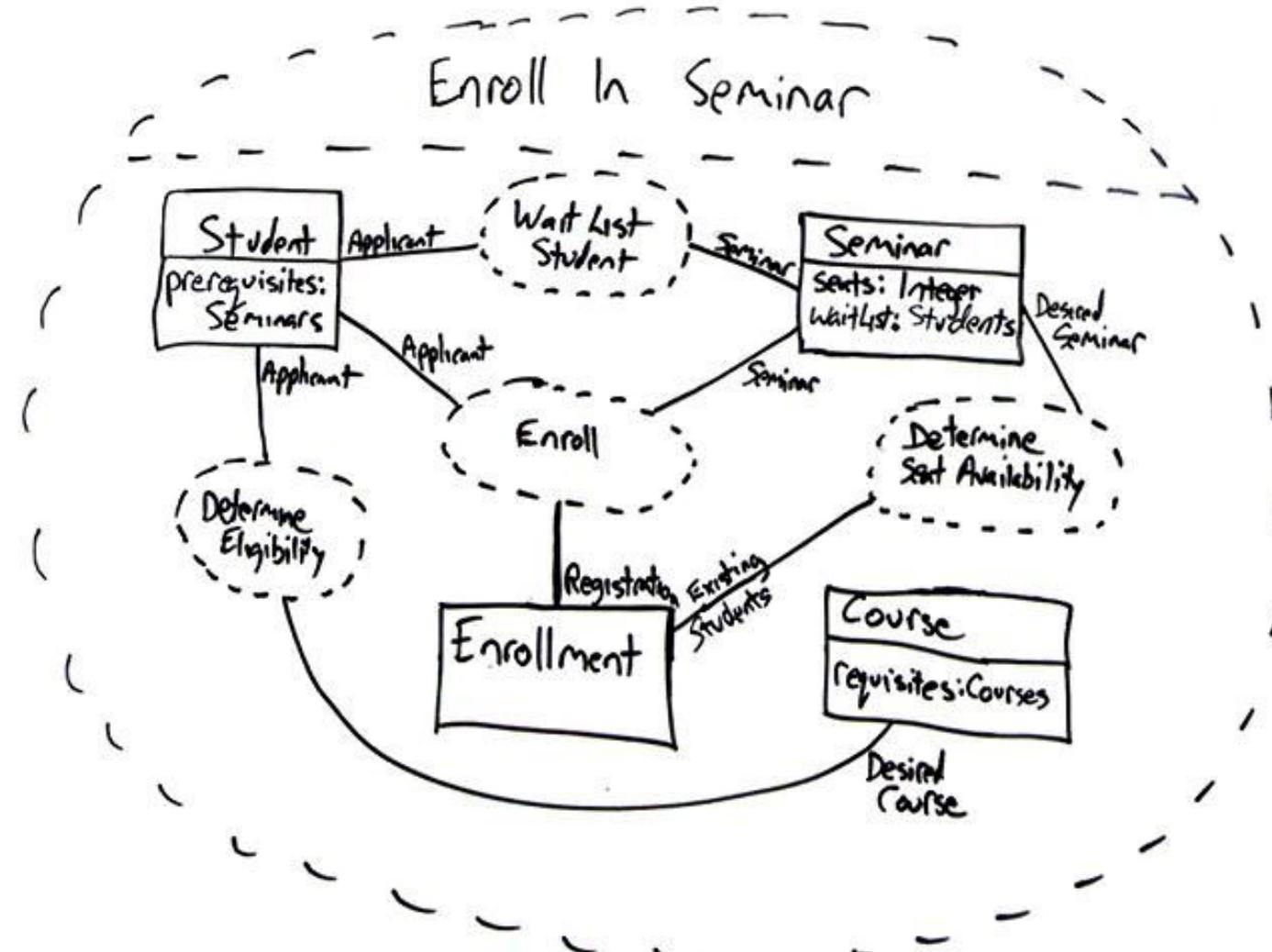


Representing concepts, things, structure, behaviour





Software design: combining abstractions into larger units or components





Naming, composition: try to read and write *stories*



```
3 let year = 2020
4 let bonus_ceiling = 5.5
5
6 try {
7     let employees = db.exec("FROM EMPLOYEES SELECT * WHERE ACTIVE=1").toArray()
8     for(let n=0; n < employees.count; n++){
9
10         let employee = employees[n]
11         let bonus = service.EmployeeAppraisal(employee.ID, year)
12         if(bonus <= bonus_ceiling) {
13             db.exec("UPDATE EMPLOYEES " +
14                 "SET bonus = %d" +
15                 "WHERE ID = %i", bonus, employee.ID)
16         }
17     }
18 } catch(error){
19     throw error
20 }
21
22
23
24
25 // or
26 service.updateActiveEmployeesWith(bonus({year: 2020, ceiling: 5.5}));
27
28 //
29 service.updateActiveEmployeesWith(changeGlobalAddress);
30
```





Separation of Concerns

Separation of concerns is a principle for breaking up your program into sections so each of them does a single thing.

Each of your components (modules, classes, etc) should be a grouping of functions (methods, etc) that are related to a single task. Each of the functions in that component should do one thing and do it well. For example, in a User component, have a function to add a user, another to delete, etc. By separating things logically you make them more dependable, portable, and flexible. You reduce dependencies across your components that otherwise would make debugging a nightmare.

Do one thing and do it well.





Compose existing abstractions rather than inventing your own

**Pick and choose existing abstractions,
libraries, frameworks**

Design your code when a design is lacking
but don't try to be an "Architect".

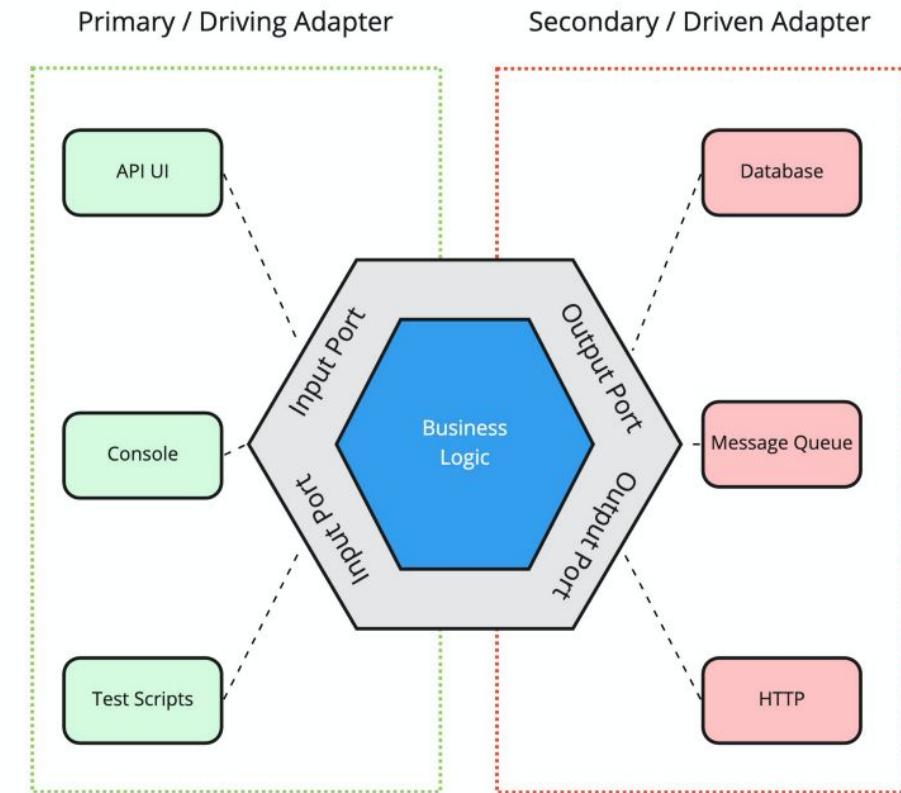
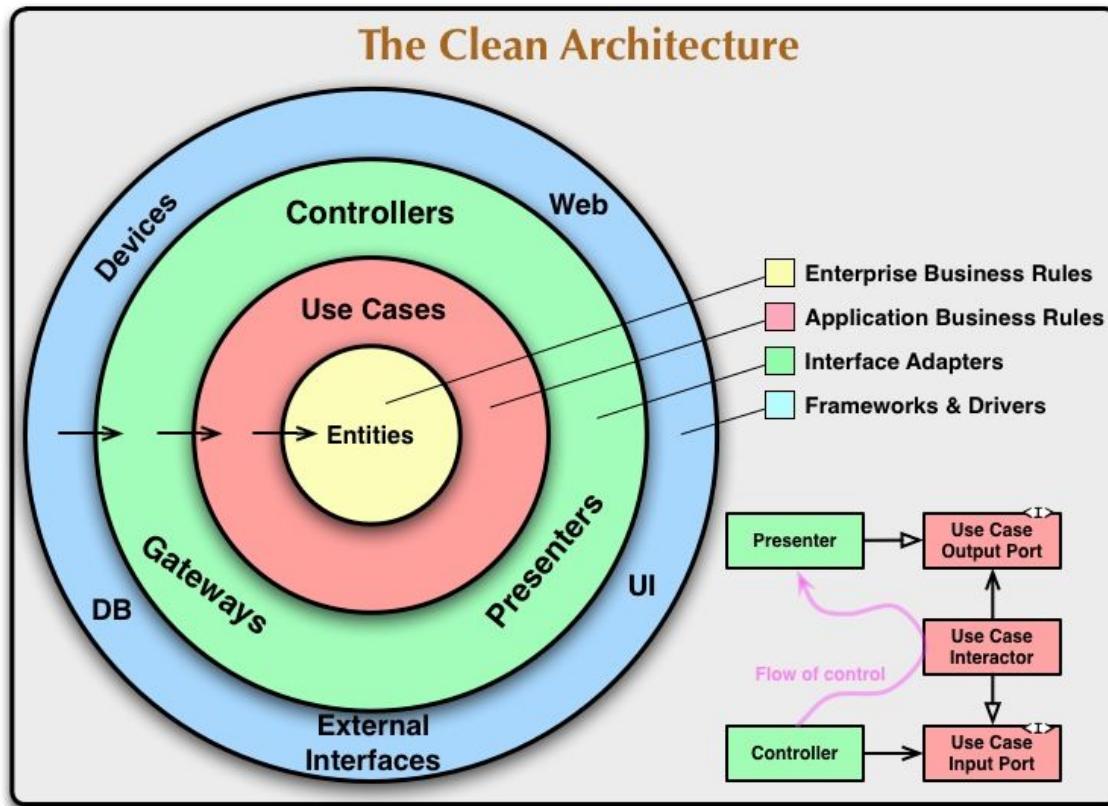
"Architecture" is a big upfront expense
and difficult to get right.

So use an existing architecture or
framework which defines how should be
structured.



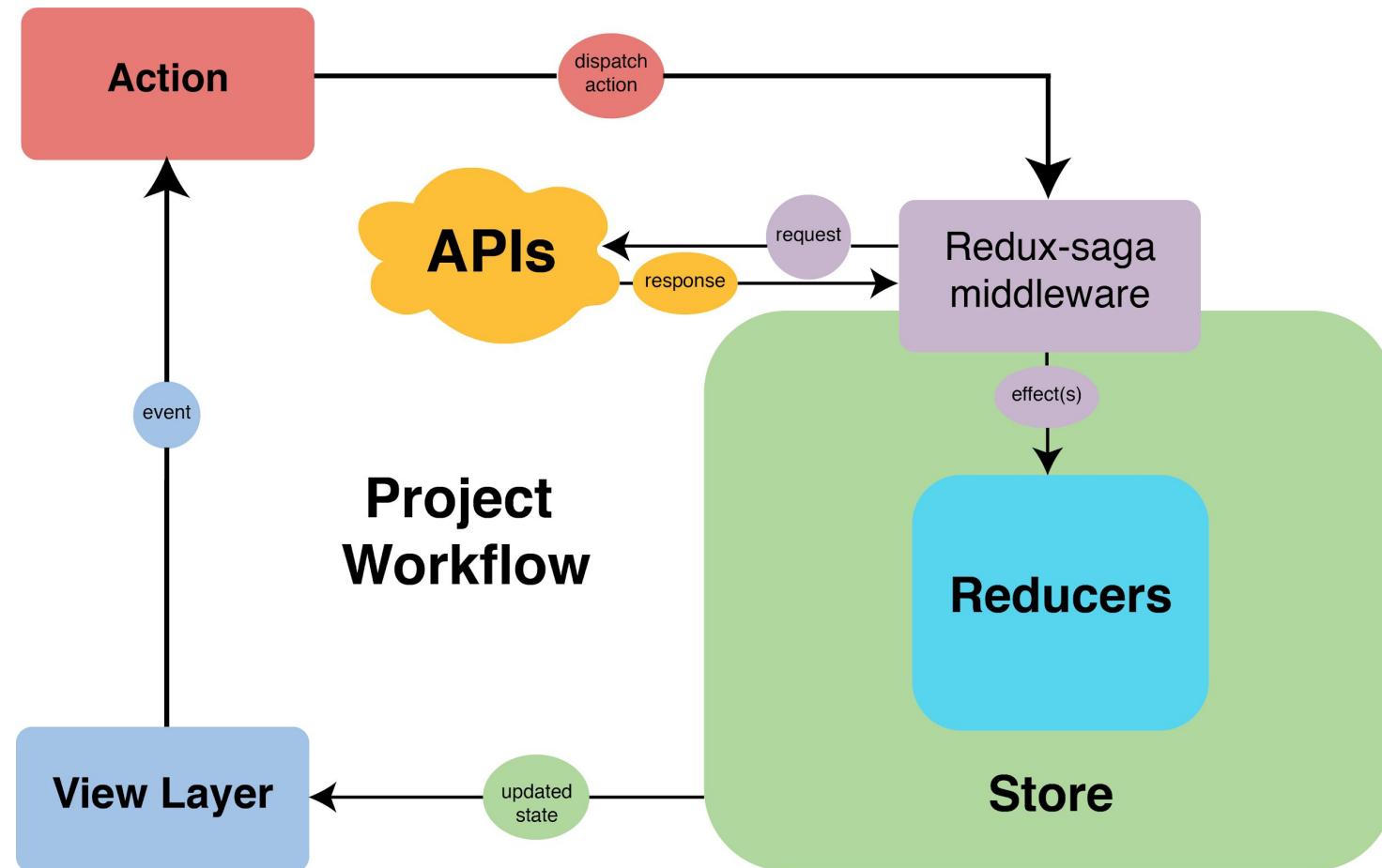


Use existing Architecture design models (for example Clean or Ports and Adapters)





Use existing component architectures (for example React / Redux)



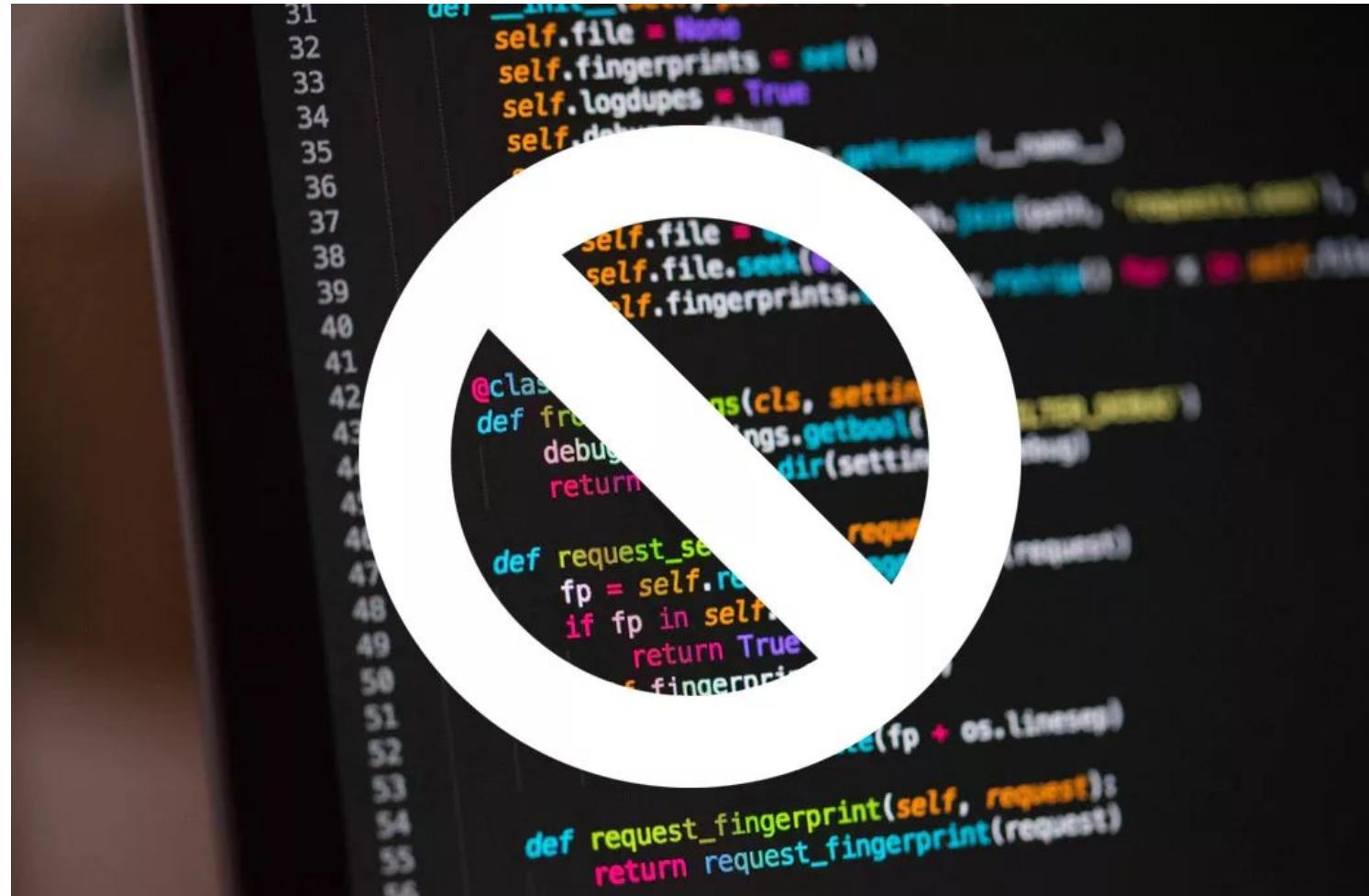
Don't "Program". Or at least postpone it as long as possible But do use "Code"



- Use "off the shelf" products
- Configure
- Design
- Implement Stories
- Use libraries

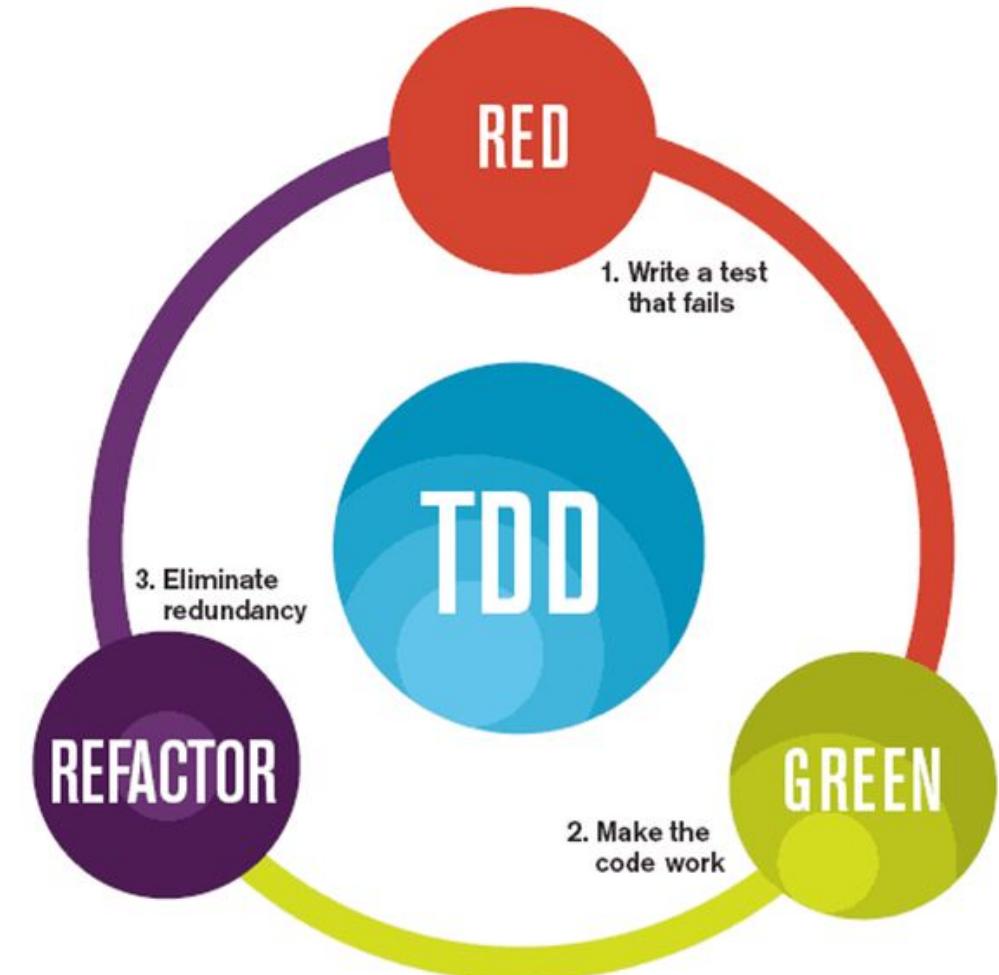
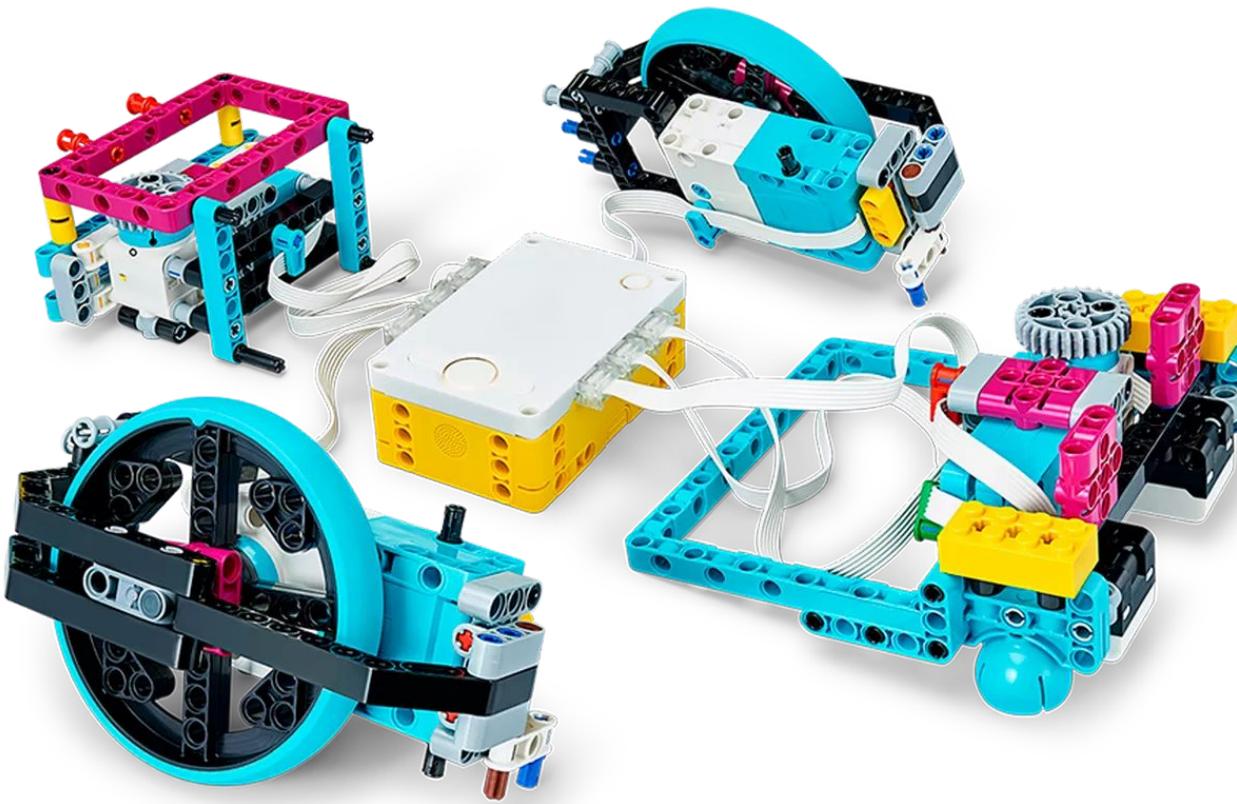
"programming" is expensive (time, effort) and error prone

Code to test, to investigate, to play around, to design





Code to experiment, test, prototype to gain knowledge and take away uncertainty



The mantra of Test-Driven Development (TDD) is “red, green, refactor.”



Code prototyping / Type Driven Development example

Flash sales, left-over sales, launch day sales, holiday sales and limited inventory sales are all high volume, high velocity events. Here we call them "Markets". The following User Stories help to define the general scope:

As a client of a market I want to receive a notification so I can bid for items temporarily available on that market

As a client I want to receive a notification whether a bid for an item has been overbid so I can make a counter bid if so required

As a Client I want to receive the final status of a bid so that I can arrange for the post-sale process (etc)

```
type AvailableItemsHandler = fn(_:[&Item]);
type ProcessedBidHandler = fn(_:&Bid, _:BidStatus);

enum BidStatus {
    OverBid = 0,
    Canceled,
    Succeeded
}

trait Market {
    fn register_bid(_:Item, _:Bid);
    fn notify_available_items(_:AvailableItemsHandler);
    fn notify_bid_action(_:ProcessedBidHandler);
}

struct FleaMarket {

    bids: Vec<Item>
    //... more data
}

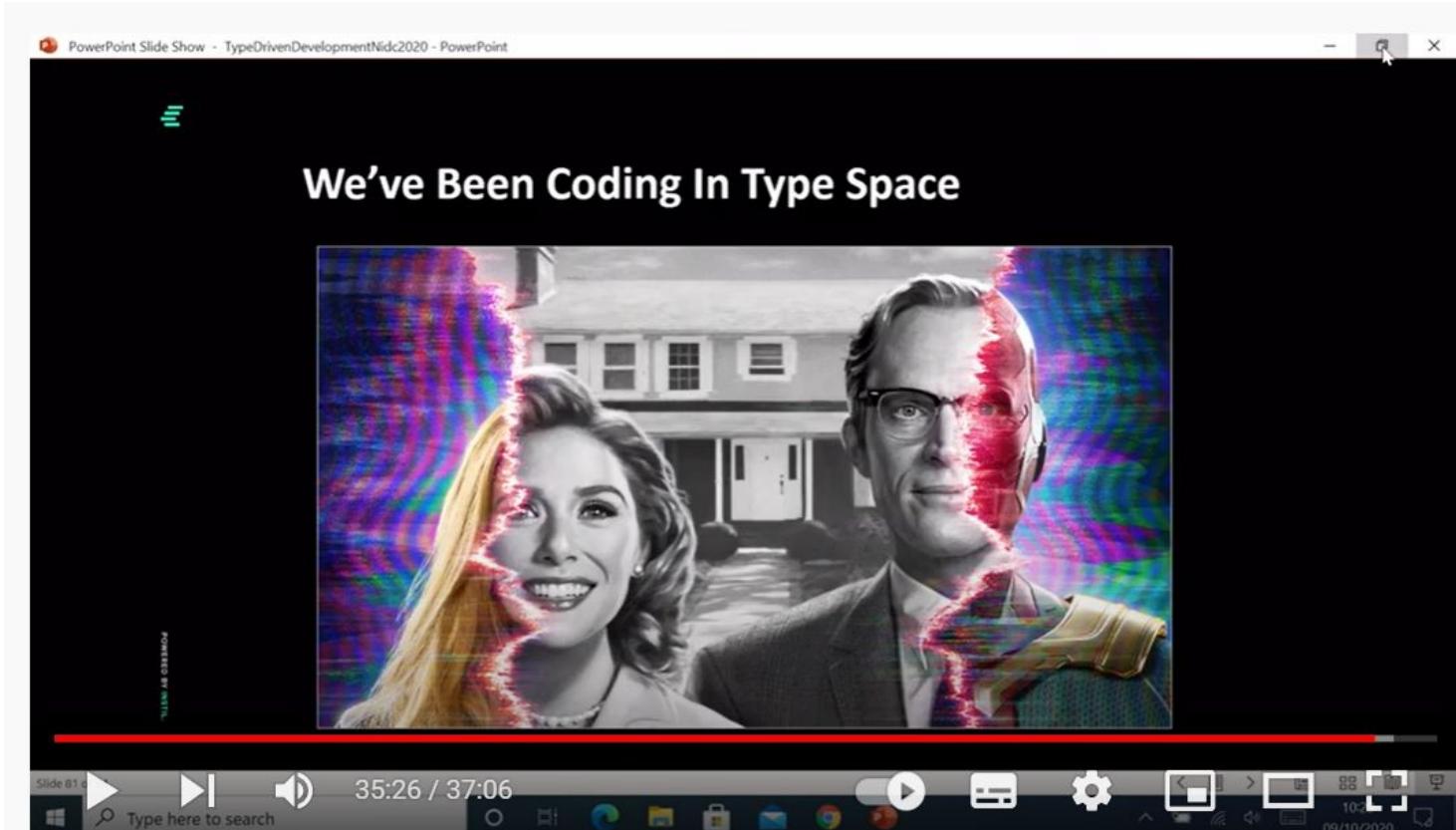
impl Market for FleaMarket {

    fn register_bid(item: Item, bid: Bid) { /* code here */ }
    fn notify_available_items(handler: AvailableItemsHandler){ /* code here */ }
    fn notify_bid_action(handler: ProcessedBidHandler){/* code here */ }
}
```





Use code to define, "Type", to design data models, interfaces, functions etc



Not Your Mother's TDD: Type Driven Development in TypeScript - G Gilmour & R Gibson -
NIDC2020

<https://youtu.be/YidUm-CO9kk>

<https://www.slideshare.net/ggilmour/type-driven-development-with-typescript>



Type Driven Design



"Type-Driven API Design in Rust" by Will Crichton
<https://youtu.be/bnnaclegg6k>



Type Driven API Design in Rust
by Will Crichton @wcrichton
(sit close to the screen!)

```
ls -l [strangeloop-talk]intro.txt
```

All Text Projectiles strangeloop-talk/rust-cargo

Wrote /Users/will/Code/strangeloop-talk/intro.txt

Domain Modeling Made Functional

```
type Contact = {  
    FirstName: string  
    MiddleInitial: string  
    LastName: string  
  
    EmailAddress: string  
    IsEmailVerified: bool  
} // true if ownership of  
// email address is confirmed
```

Find out soon!

How many things are wrong with this design?



Domain Modeling Made Functional - Scott Wlaschin
<https://youtu.be/Up7LcbGZFu0>

Get, grasp, grok, understand the problem and the solution



It is critical that you fully understand the underlying architecture, concepts, patterns in order to be able to use them competently. **At the level of the abstractions, not the “internals”.** In order to be able to drive a car you need to know the mechanics of driving and the traffic rules, not the internals of the car engine....



And when you Code, it should tell the story, reflect the Design

src > system.clj

```
1
2 (system social-security-benefit-upload
3   (component
4     (security :config "/config/security.conf" :cert-store "/config/cert.cer"))
5   (component
6     (view upload-service)
7     (logic upload)
8     (data item-store-dao log-data)))
9 (component
10  (view report-service)
11  (logic reporting)
12  (data jasper-reports item-store-dao log-data)))
```





But....

- Not all people find textual representation readable for all aspects of the design (especially relationships)
- Most programming languages cannot adequately express the whole software design satisfactorily - their syntax is not sufficiently expressive, flexible or extensible enough to viably communicate the wider architecture concerns
- We therefore reach for models and diagrams, representing the code

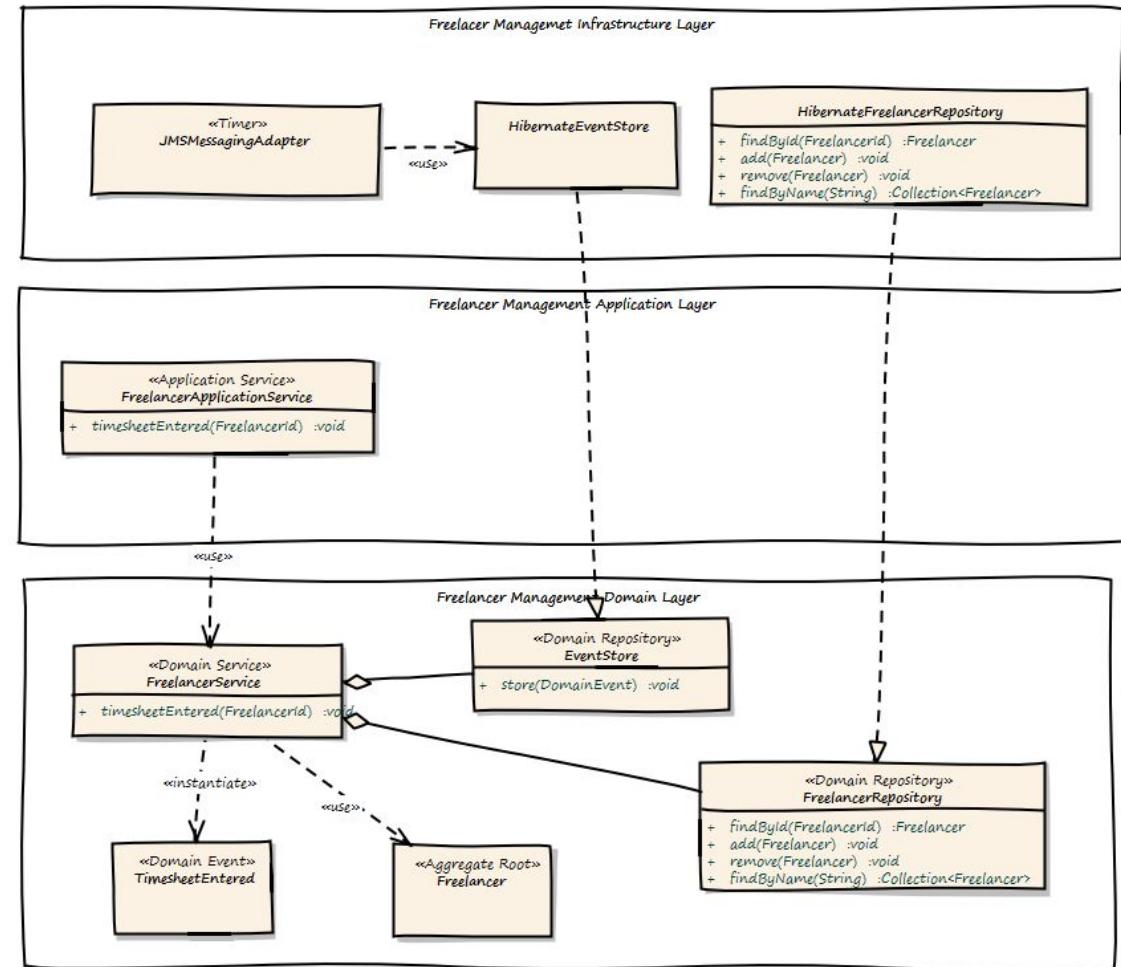


Draw it; model it. But trace it back to the code

If you cannot draw a system or component you probably have not been able to proper model it; or did not think enough about it.

You can use any modeling methodology but the more important thing is conveying meaning in a concise and clear way.

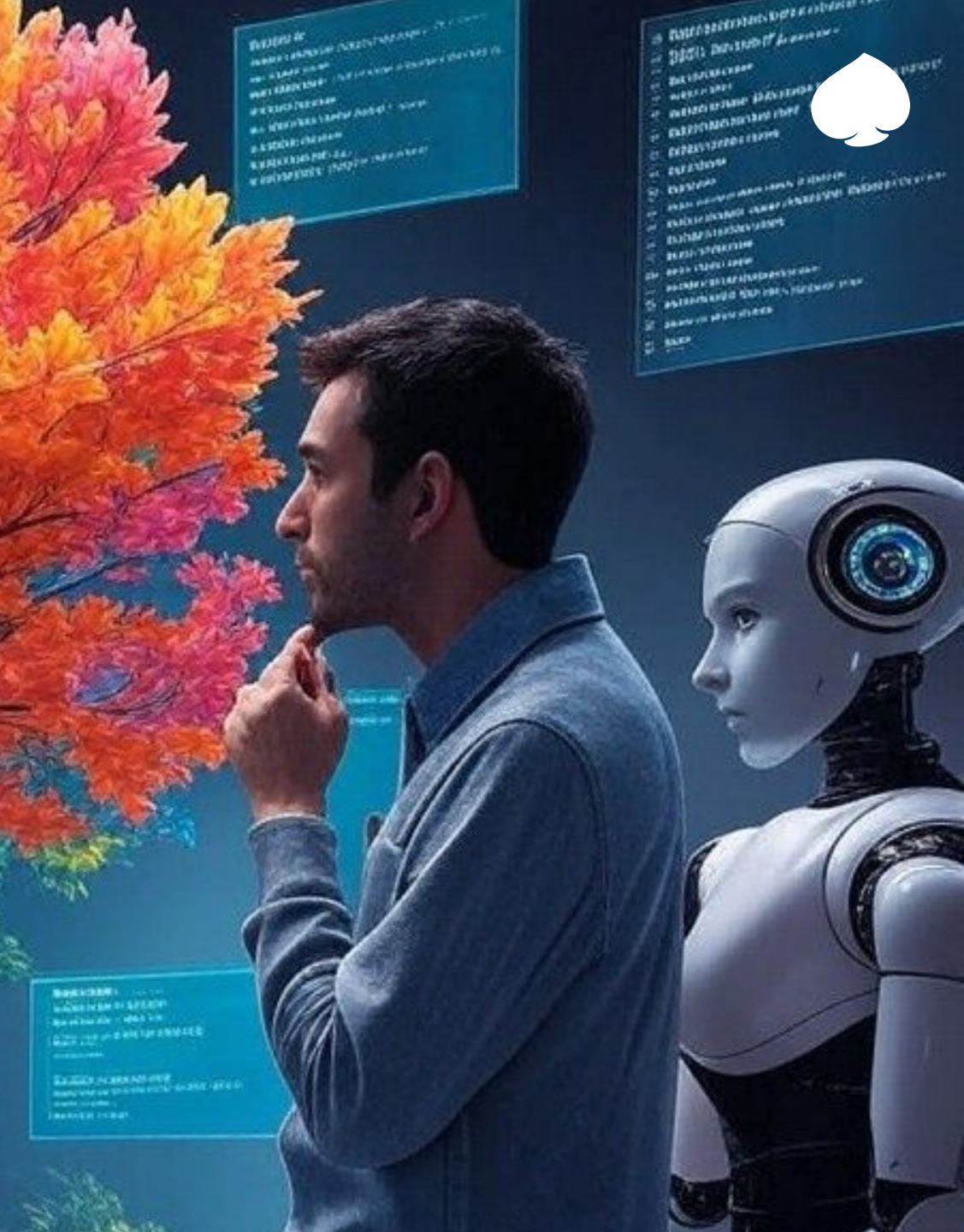
The structure of the architecture and your code should be directly mapped to the model and vice versa.





The advent of AI Augmented Software Engineering does not change the essence of programming as a linguistic process of abstraction, composition, and decomposition, enhanced by storytelling.

This shift makes it feasible for software engineering to evolve into a process where design maintains its primacy as a human-led activity, while AI efficiently handles the more repetitive task of writing code, allowing humans to focus on creativity and strategic thinking.





A Practical Guide to Collaborative Software Design



Summary session 1

Overview

Session 1 introduces software design as a collaborative, narrative-driven process rooted in abstraction and storytelling. It emphasizes that software engineering should be treated as both a craft and a culture, focusing on human-centric design principles. Key elements include:

Philosophy:

- Software design is a process of creating and communicating a system's structure and meaning.
- It integrates user experience (UX) as an inseparable part of the design process.
- Design is defined as both a process and a model, balancing abstract and concrete elements.

Key Concepts:

- Architecture vs. Design: Architecture addresses high-level decisions (frameworks, methodologies) while design focuses on organizing code and defining relationships.
- Abstraction: Abstract representations reduce complexity, forming the backbone of programming and design.
- Storytelling: Software design parallels storytelling, where abstractions convey meaning and guide implementation.

Practices:

- Prioritize existing architectures and frameworks to minimize complexity.
- Use abstraction effectively, ensuring precision rather than vagueness.
- Represent concepts visually through diagrams, ensuring traceability to code.

Recommendations:

- Treat programming as a linguistic process, combining abstraction and composition.
- Postpone programming specifics by focusing on high-level design and prototypes.
- Adopt modular and testable structures with clear separation of concerns.



1 Architecture and Software Design

2 Mindset and practices

3 Models

4 How to Design

5 Next Steps &
Further reading and viewing...

Next Session



A Practical Guide to Collaborative Software Design

2025 edition - Session 2



1 Architecture and Software Design

2 Mindset and practices

3 Models

4 How to Design

5 Next Steps &
Further reading and viewing...

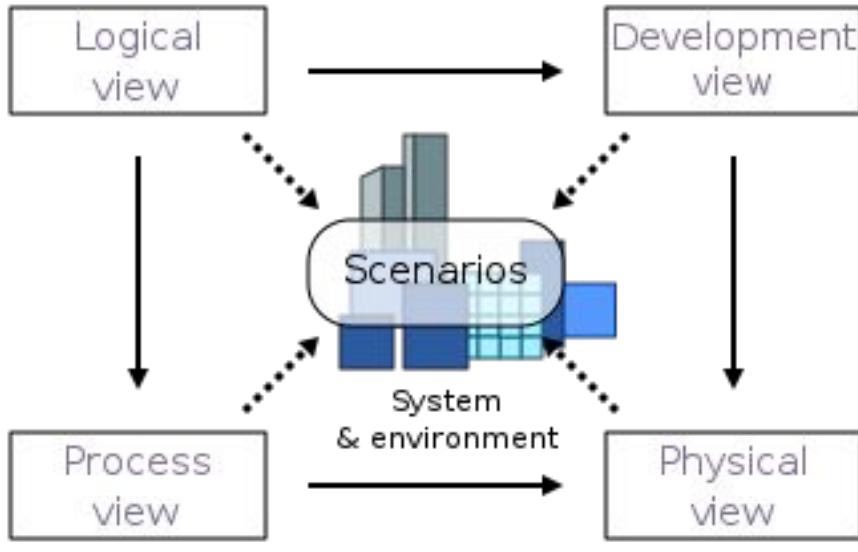
Today



Models



We can create models using existing paradigms like the 4+1 architectural view model



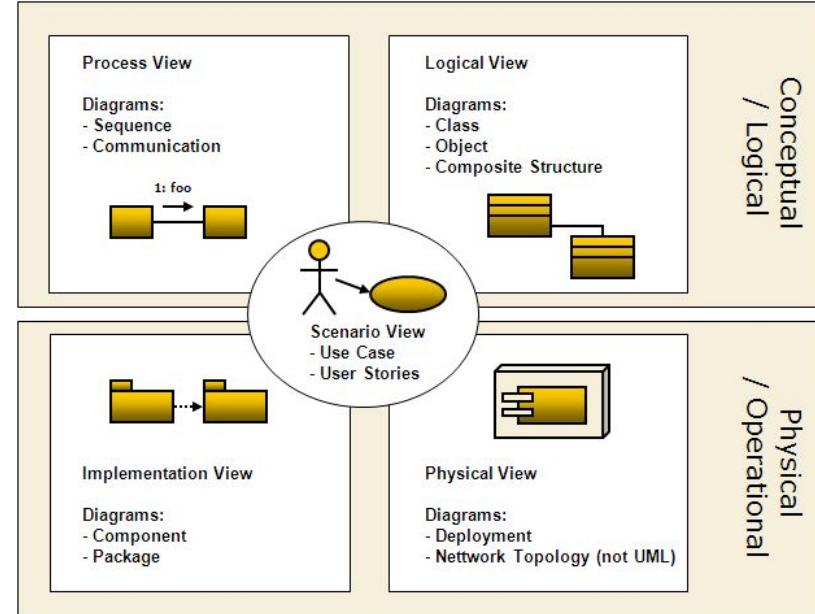
CHALMERS | UNIVERSITY OF GOTHENBURG |

The 4+1 view model

In order to eventually address large and challenging architectures, the model we propose is made up of five main views (cf. fig. 1):

1. The logical view, which is the object model of the design (when an object-oriented design method is used),
2. the process view, which captures the concurrency and synchronization aspects of the design,
3. the physical view, which describes the mapping(s) of the software onto the hardware and reflects its distributed aspect,
4. the development view, which describes the static organization of the software in its development environment.
5. The description of an architecture—the decisions made—can be organized around selected use cases, or scenarios which become a fifth view.

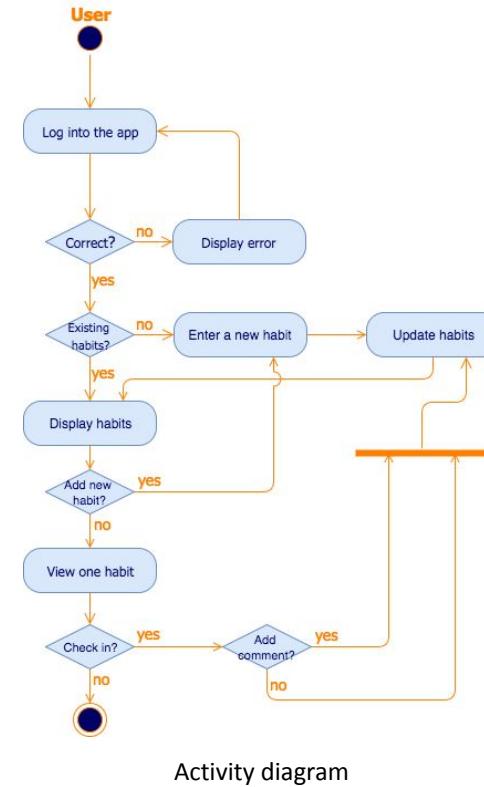
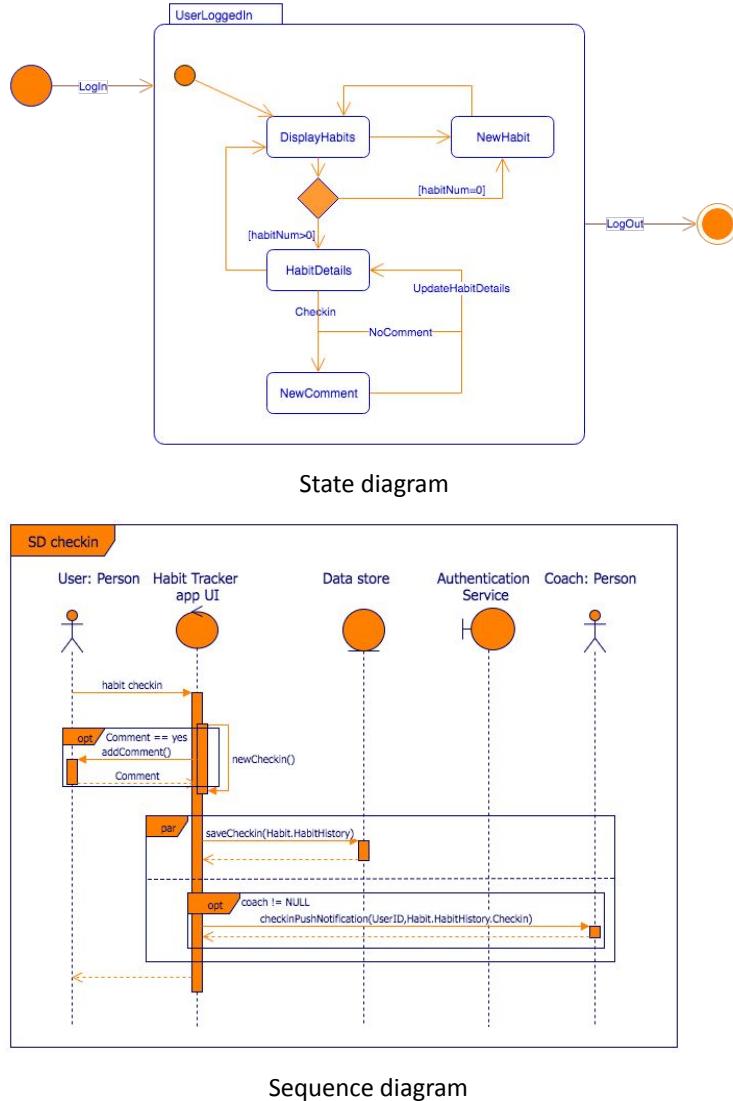
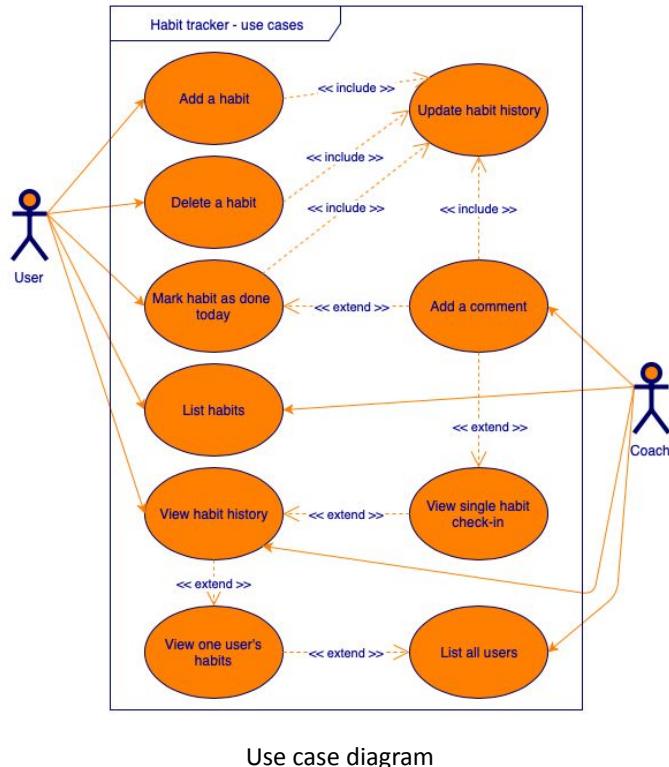
+1 architecture view model & behavior model: state chart diagrams



<https://arxiv.org/abs/2006.04975>

<https://youtu.be/r8ucofiI8vY>

Unified Modeling Language (UML)





Unified Modeling Language (UML)

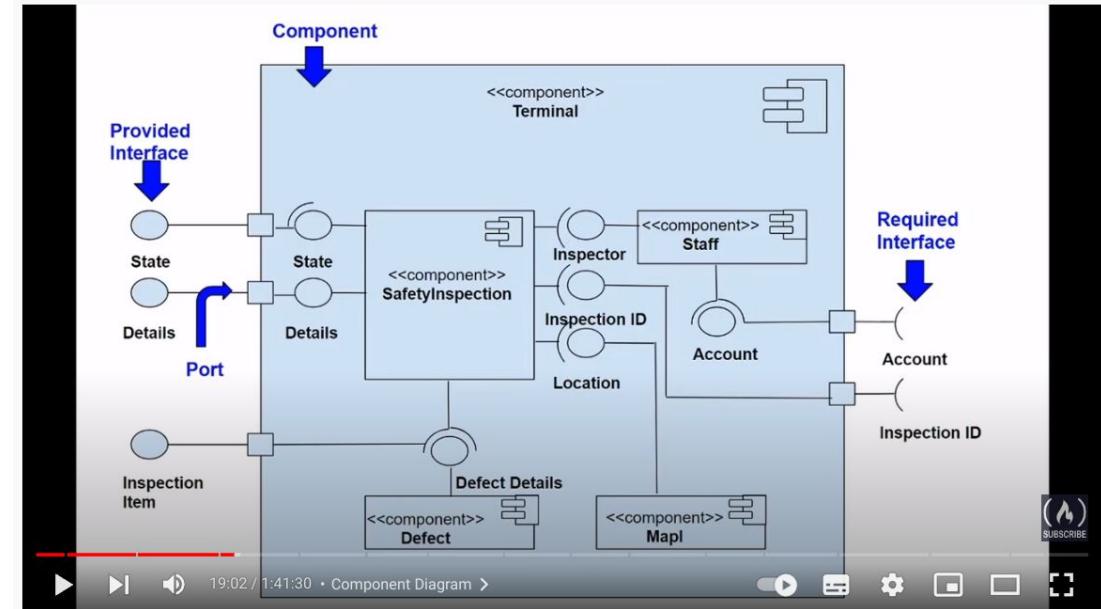
- Standard notation
- Good variety in diagrams
- Great tool support (see PlantUML for usage in VSCode)

But...

- Low level
- Limited expressiveness (needs OCL)
- Too complex
- Not very suitable for programming paradigms which are not “OOP”
- “Nobody uses UML”

Associated with “Big Design up-front”, “Waterfall”

Not being taught at University anymore



<https://youtu.be/WnMQ8HlmeXc>



As many diagram types as there are designs



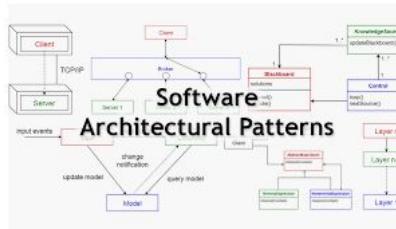
Google

software architecture diagram

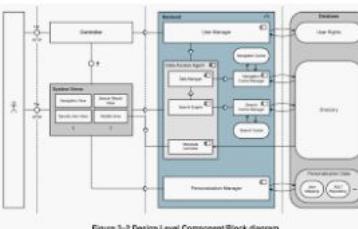


All Images Videos Shopping News More Settings Tools Collections SafeSearch ▾

software architecture design example deployment uml azure visio draw application level visualising powerpoint rational software component templates static architecture design >



Software Architectural Patterns ...
towardsdatascience.com



Software Architecture Modelling ...
softwareengineering.stackexchange.com

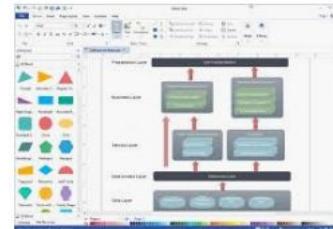
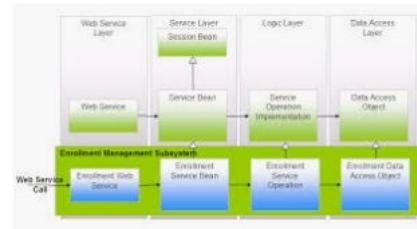


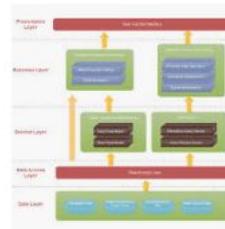
diagram software architecture ...
quora.com



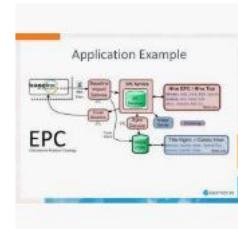
A Software Architect's View On Diagrams ...
slideshare.net



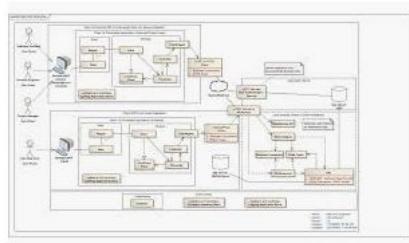
Examples of well designed software ...
graphicdesign.stackexchange.com



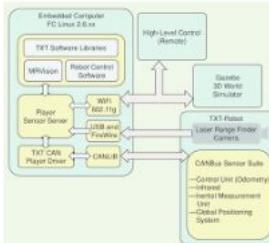
Software Architecture Exam...
edrawsoft.com



A Software Architect's Vie...
slideshare.net



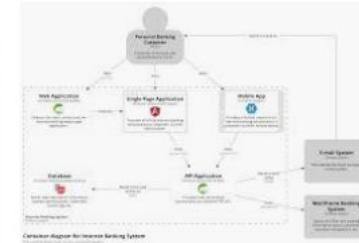
What is Software Architecure
predic8.com



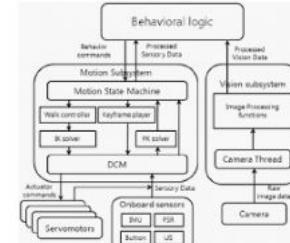
block diagram shows how ...
researchgate.net



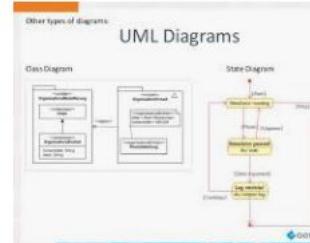
Easy Architecture Diagram Software
edrawsoft.com



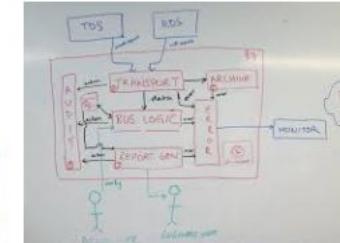
Structurizr



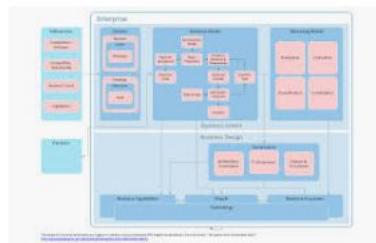
Block Diagram of the Software ...
researchgate.net



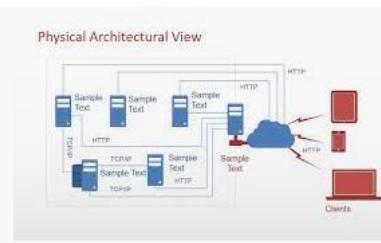
A Software Architect's View On Diagrams ...
slideshare.net



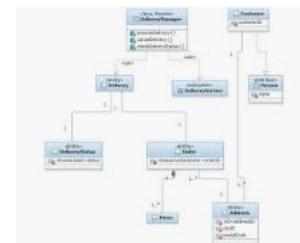
model for visualising software architect...
c4model.com



Computer Software Microsoft Visio ...
kisspng.com



Software Diagrams for PowerPoint ...
slidemodel.com



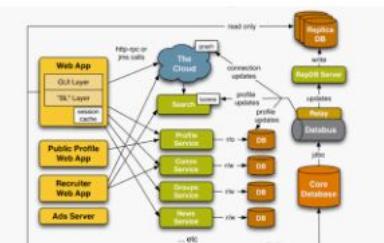
Rational Software Architect ...
ibm.com



Web Portal Static Software Architecture ...
slidemodel.com



Software architecture diagram - software ...
ricksnursery.com



technical architecture diagram ...
softwareengineering.stackexchange.com

Model-code gap

“...Your architecture models and your source code will not show the same things. The difference between them is the *model-code gap*. Your architecture models include some abstract concepts, like components, that your programming language does not, but could. Beyond that, architecture models include intentional elements, like design decisions and constraints, that cannot be expressed in procedural source code at all.

Consequently, the relationship between the architecture model and source code is complicated

JUST ENOUGH SOFTWARE ARCHITECTURE

A RISK-DRIVEN APPROACH

GEORGE FAIRBANKS

FOREWORD BY DAVID GARLAN





Visualise, document and explore your software architecture - Simon Brown



Premium ES

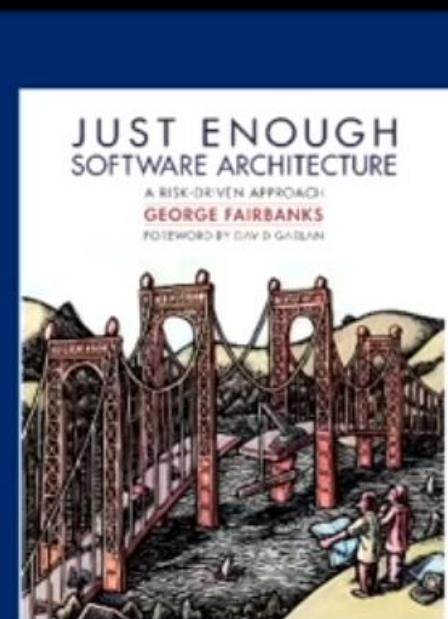
Search



NDC { London }

16-20 January 2017

Inspiring Software Developers
since 2008



The image shows the front cover of the book "JUST ENOUGH SOFTWARE ARCHITECTURE: A RISK-DRIVEN APPROACH" by George Fairbanks, with a foreword by David Guban. The cover features a colorful illustration of a suspension bridge over a river, with two figures standing on the bank in the foreground.

Model-code gap. Your architecture models and your source code will not show the same things. The difference between them is the *model-code gap*. Your architecture models include some abstract concepts, like components, that your programming language does not, but could. Beyond that, architecture models include intensional elements, like design decisions and constraints, that cannot be expressed in procedural source code at all.

Consequently, the relationship between the architecture model and source code is complicated. It is mostly a refinement relationship, where the extensional elements in the architecture model are refined into extensional elements in source code. This is shown in Figure 10.3. However, intensional elements are not refined into corresponding elements in source code.

Upon learning about the model-code gap, your first instinct may be to avoid it. But reflecting on the origins of the gap gives little hope of a general solution in the short term: architecture models help you reason about complexity and scale because they are abstract and intensional; source code executes on machines because it is concrete and extensional.

"model-code gap"



The C4 model for visualising software architecture

Context, Containers, Components and Code

Abstractions Core diagrams Supplementary diagrams Notation
Examples FAQ Diagramming vs modelling Training Tooling

⌚ In a hurry? Read the 5 minute introduction to the C4 model at InfoQ:

The C4 model for software architecture

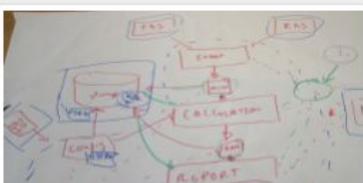
O modelo C4 de documentação para Arquitetura de Software

用于软件架构的C4模型

ソフトウェアアーキテクチャのためのC4モデル

Introduction

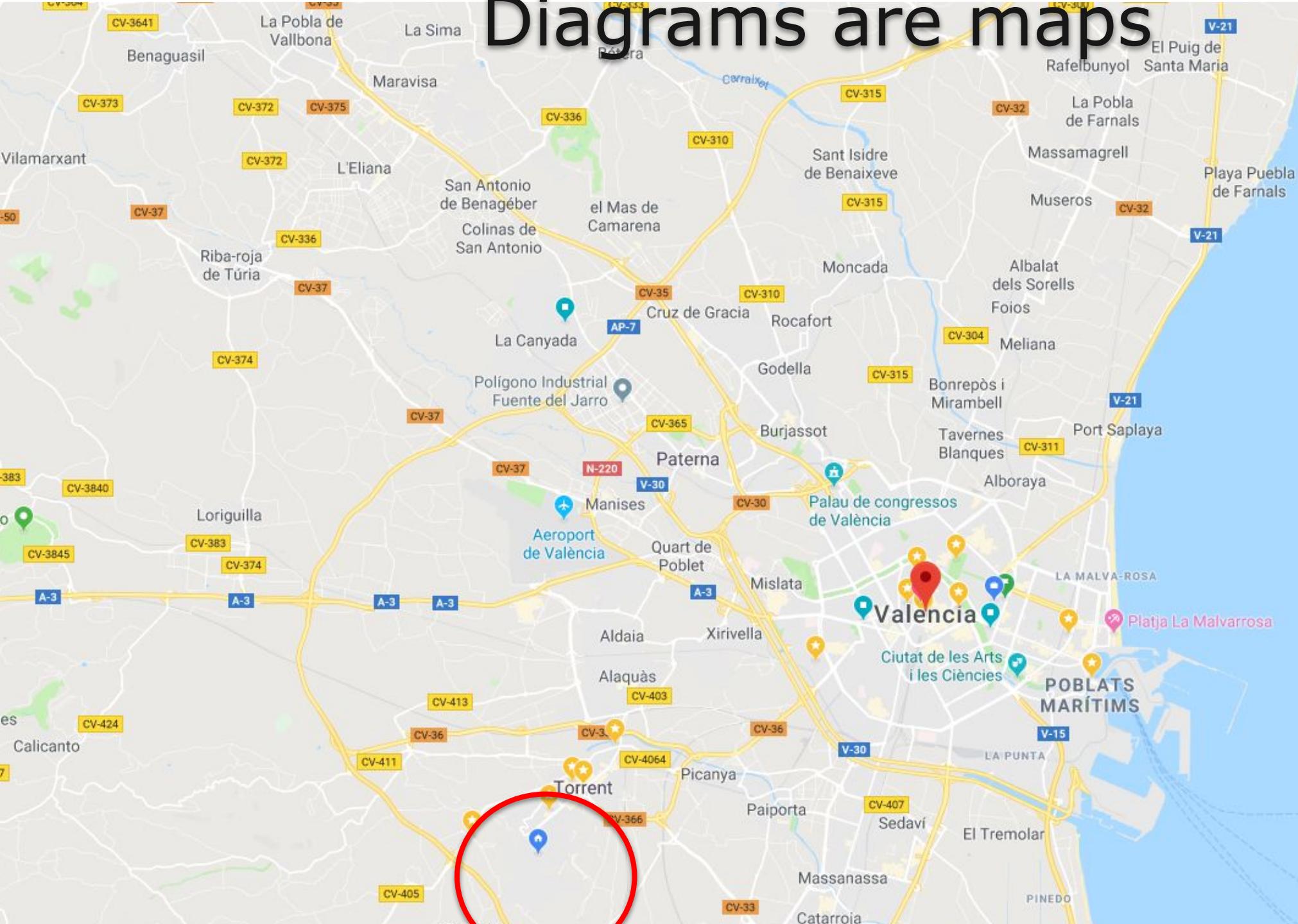
Ask somebody in the building industry to visually communicate the architecture of a building and you'll be presented with site plans, floor plans, elevation views, cross-section views and detail drawings. In contrast, ask a software developer to communicate the software architecture of a software system using diagrams and you'll likely get a confused mess of boxes and lines ... inconsistent notation (colour coding, shapes, line styles, etc), ambiguous naming, unlabelled relationships, generic terminology, missing technology choices, mixed abstractions, etc.



Diagrams are maps



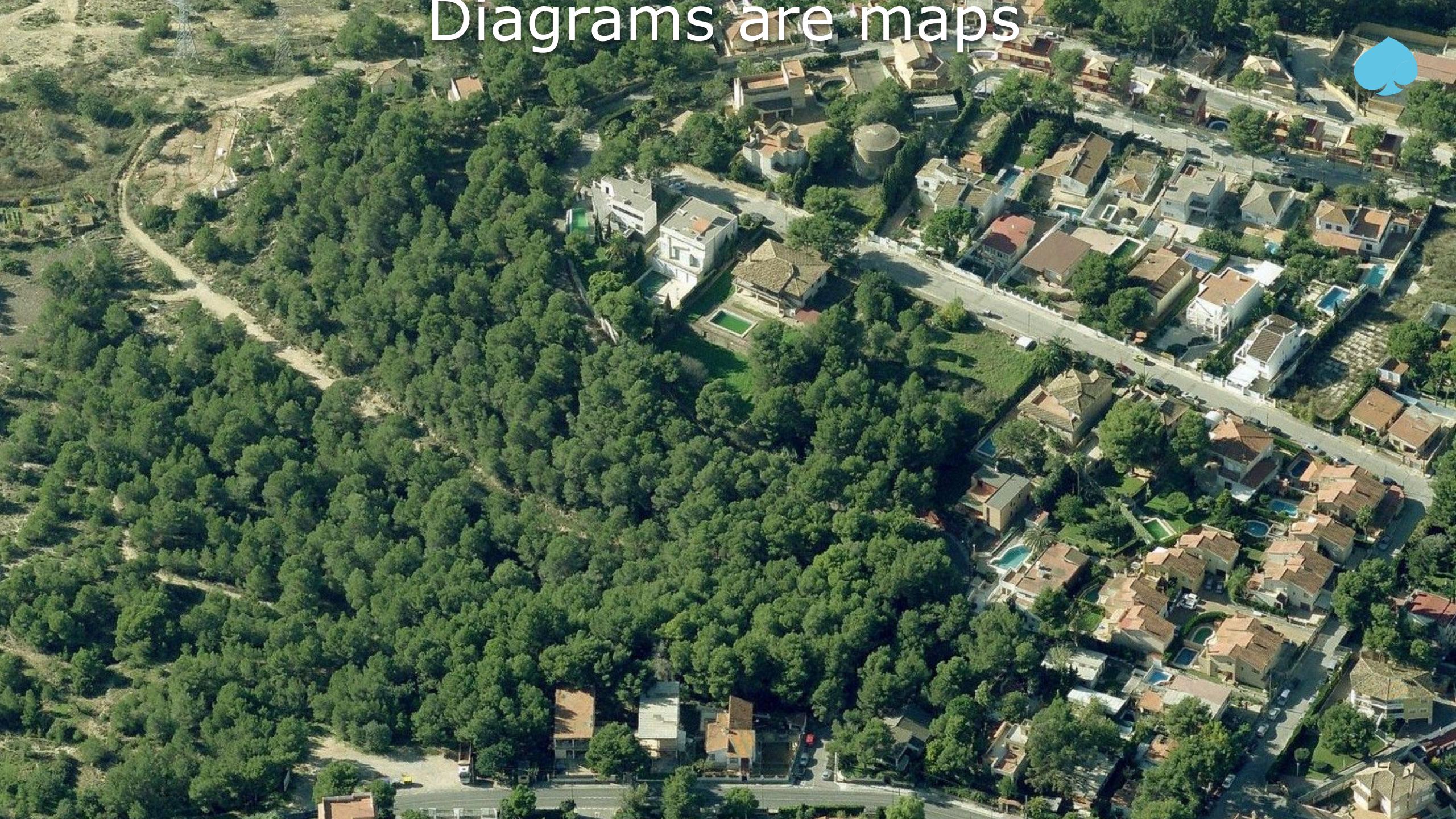
Diagrams are maps



Diagrams are maps

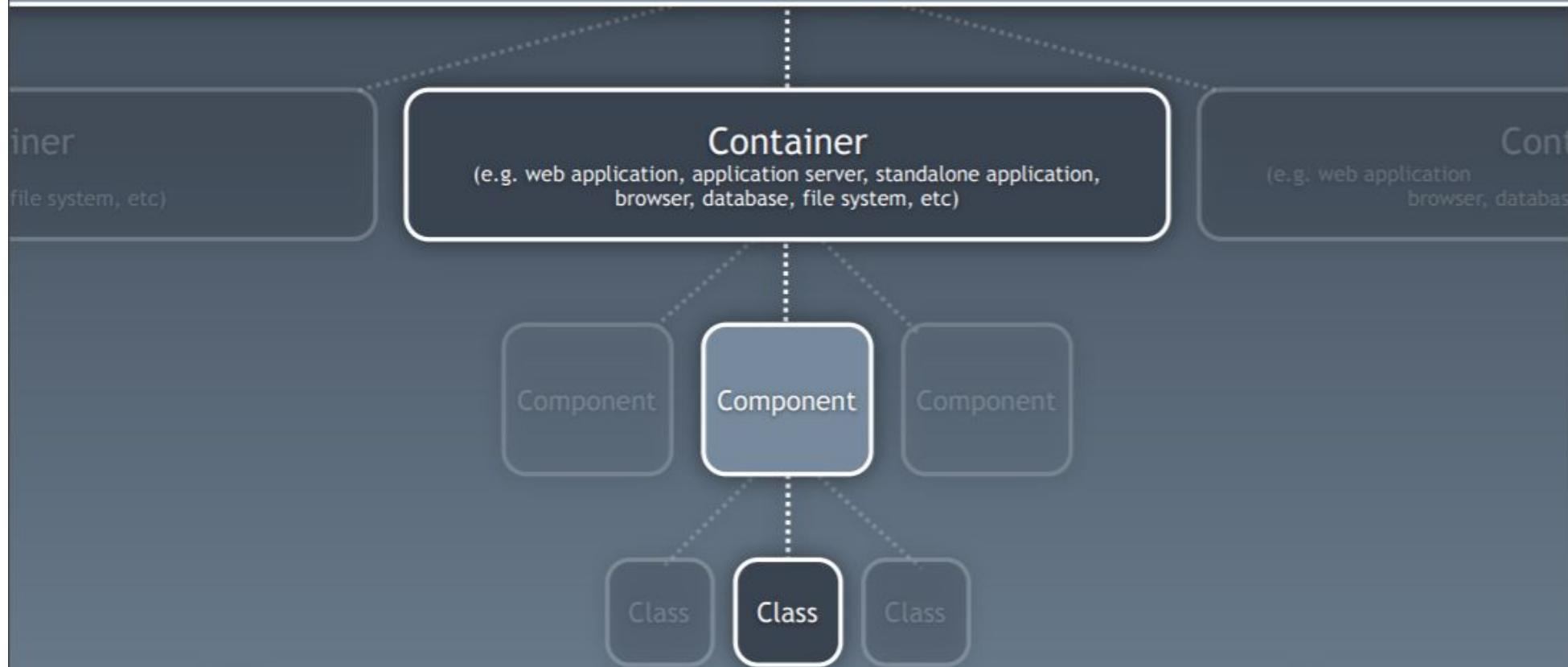


Diagrams are maps





Software System



A **software system** is made up of one or more **containers**,
each of which contains one or more **components**,
which in turn are implemented by one or more **classes**.



The C4 model



System Context

The system plus users and system dependencies



Containers

The overall shape of the architecture and technology choices



Components

Components and their interactions within a container



Classes (or Code)

Component implementation details



Compose

Tell the story

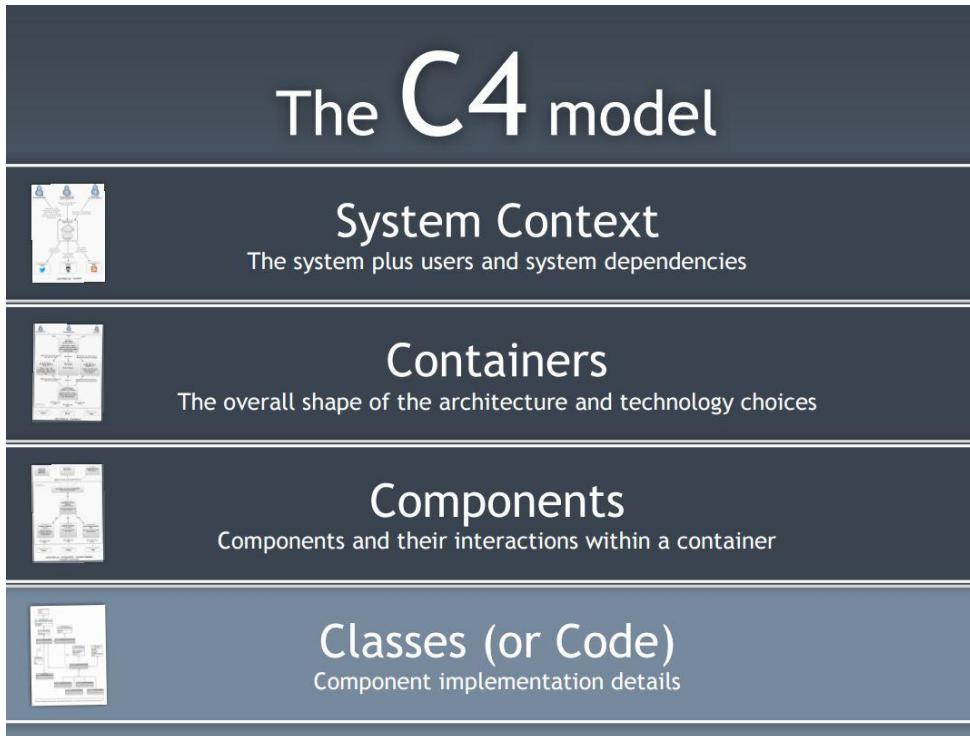
Decompose



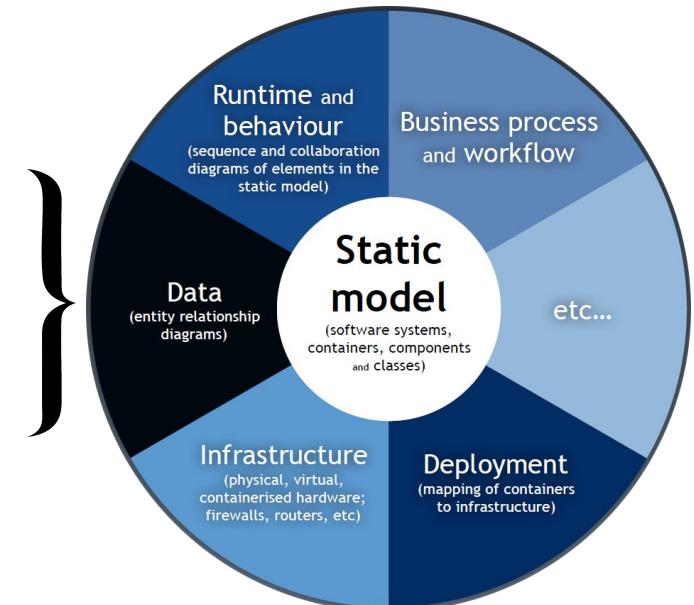


A Hierarchy of Models

The C4 diagrams



- UX/UI prototype
- UML
(Sequence, State, Class etc)
- Flow diagrams
- Sketches and drawings
- Interfaces
- Code fragments
- etc...





How to Design



C4 and UML in Visual Studio Code, Jetbrains editors etc using PlantUML

The screenshot shows a code editor with a C4 System Landscape diagram for Big Bank plc. The diagram is titled "System Landscape diagram for Big Bank plc". It features a central enterprise boundary labeled "Big Bank plc [Enterprise]" connected to various internal systems and external stakeholders. Stakeholders include "Personal Banking Customer", "Customer Service Staff", and "Back Office Staff". Internal systems shown are "ATM", "Internet Banking System", "Mainframe Banking System", and "E-mail system". Relationships are indicated by arrows: "Asks questions to" from customer to enterprise; "Withdraws cash using" from customer to ATM; "Allows customers to withdraw cash" from ATM to enterprise; "Uses" from customer to Internet Banking System, Mainframe Banking System, and E-mail system; "Sends e-mails to" from customer to E-mail system; and "Sends e-mail using" from Internet Banking System and Mainframe Banking System to E-mail system.

```
graph TD; subgraph "System Landscape diagram for Big Bank plc"; direction TB; subgraph "Big Bank plc [Enterprise]"; end; subgraph "Personal Banking Customer"; end; subgraph "Customer Service Staff"; end; subgraph "Back Office Staff"; end; subgraph "ATM"; end; subgraph "Internet Banking System"; end; subgraph "Mainframe Banking System"; end; subgraph "E-mail system"; end; "Asks questions to" --> "Big Bank plc [Enterprise]"; "Withdraws cash using" --> "ATM"; "Allows customers to withdraw cash" --> "Big Bank plc [Enterprise]"; "Uses" --> "Internet Banking System"; "Uses" --> "Mainframe Banking System"; "Uses" --> "E-mail system"; "Sends e-mails to" --> "E-mail system"; "Sends e-mail using" --> "E-mail system"; "Sends e-mail using" --> "Mainframe Banking System"; "Sends e-mail using" --> "Internet Banking System";
```

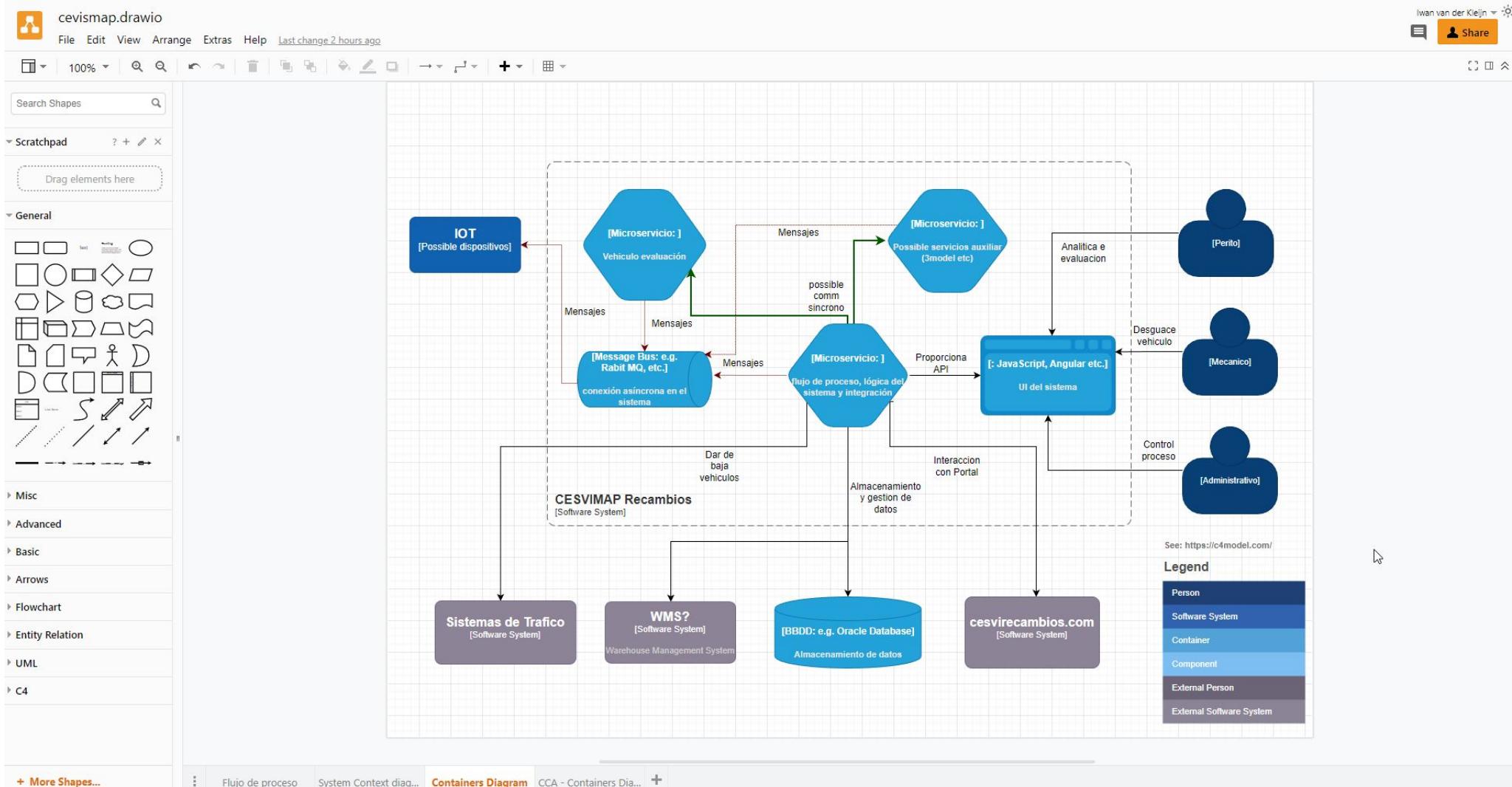
The screenshot shows a code editor with a C4 SocialSecurity Component Diagram Sample.puml. The diagram is titled "Component diagram for Social Security Use Case - Processor". It illustrates a flow starting from a "Kafka Queue" which "Listens for and responds to events". These events are processed by a "Processor (Kafka consumer) [Container]". The processor then interacts with several services: "Event Controller (Kafka Consumer)", "File Upload Service (Spring Bean)", "Report Service (Spring Bean)", and "File Storage Service (Spring Bean)". The "Event Controller" receives events from the Kafka queue and dispatches them to the appropriate service. The "File Upload Service" processes uploaded files. The "Report Service" generates reports. The "File Storage Service" stores files. The "File Storage Service" also interacts with a "File System (File Server Cluster (NTFS))" which stores files for batch processing. A "Backoffice" component connects to the "File System" and the "File Storage Service". The "Report Service" sends reports via "Email Service (Spring Bean)" to an "E-mail" recipient. A "JasperReports based Report Generator (Spring Bean)" is also shown, along with a "MongoDB" database and a "Redis" database.

```
graph TD; subgraph "Component diagram for Social Security Use Case - Processor"; direction TB; subgraph "Processor (Kafka consumer) [Container]"; end; subgraph "Event Controller (Kafka Consumer)"; end; subgraph "File Upload Service (Spring Bean)"; end; subgraph "Report Service (Spring Bean)"; end; subgraph "File Storage Service (Spring Bean)"; end; subgraph "File System (File Server Cluster (NTFS))"; end; subgraph "Backoffice"; end; subgraph "MongoDB"; end; subgraph "Redis"; end; subgraph "JasperReports based Report Generator (Spring Bean)"; end; subgraph "E-mail"; end; "Kafka Queue (Apache Kafka)" --> "Processor (Kafka consumer) [Container]"; "Processor (Kafka consumer) [Container]" --> "Event Controller (Kafka Consumer)"; "Event Controller (Kafka Consumer)" --> "File Upload Service (Spring Bean)"; "Event Controller (Kafka Consumer)" --> "Report Service (Spring Bean)"; "Event Controller (Kafka Consumer)" --> "File Storage Service (Spring Bean)"; "File Upload Service (Spring Bean)" --> "Report Service (Spring Bean)"; "Report Service (Spring Bean)" --> "Email Service (Spring Bean)"; "Email Service (Spring Bean)" --> "E-mail"; "File Storage Service (Spring Bean)" --> "File System (File Server Cluster (NTFS))"; "File System (File Server Cluster (NTFS))" --> "Backoffice"; "File Storage Service (Spring Bean)" --> "MongoDB"; "File Storage Service (Spring Bean)" --> "Redis"; "MongoDB" --> "Report Service (Spring Bean)"; "Redis" --> "Report Service (Spring Bean); "JasperReports based Report Generator (Spring Bean)" --> "Report Service (Spring Bean);
```

<https://github.com/plantuml-stdlib/C4-PlantUML>

C4 and UML in Diagrams.net (draw.io)

Single use and collaborative





Collaborative Design (using Mural or equivalent)

Wayat - initial design

Facilitator All changes saved!

Share

1 Context Diagram

2 System Diagram

3 Data Model

4 Sequence Diagrams

5 State Diagram

6 Sequence Diagram - Opening Map

7 Sequence Diagram - Closing Map

8 Logic Flow Diagram

9 Sequence Diagram - Implementation - Update Location and Contacts State

10 Sequence Diagram - Implementation - Client connection to altered state on startup

11 Process: RegenerateMapUser:input

Representation of the issue

OUTDATED

Reason: forcing all Contacts to send the location at least once when a map opens solves

Navigation Settings



Social Security Example

The application is used to give benefit recipients the option of uploading 'supporting documents' (documents such as pay slips) that prove a working relationship.

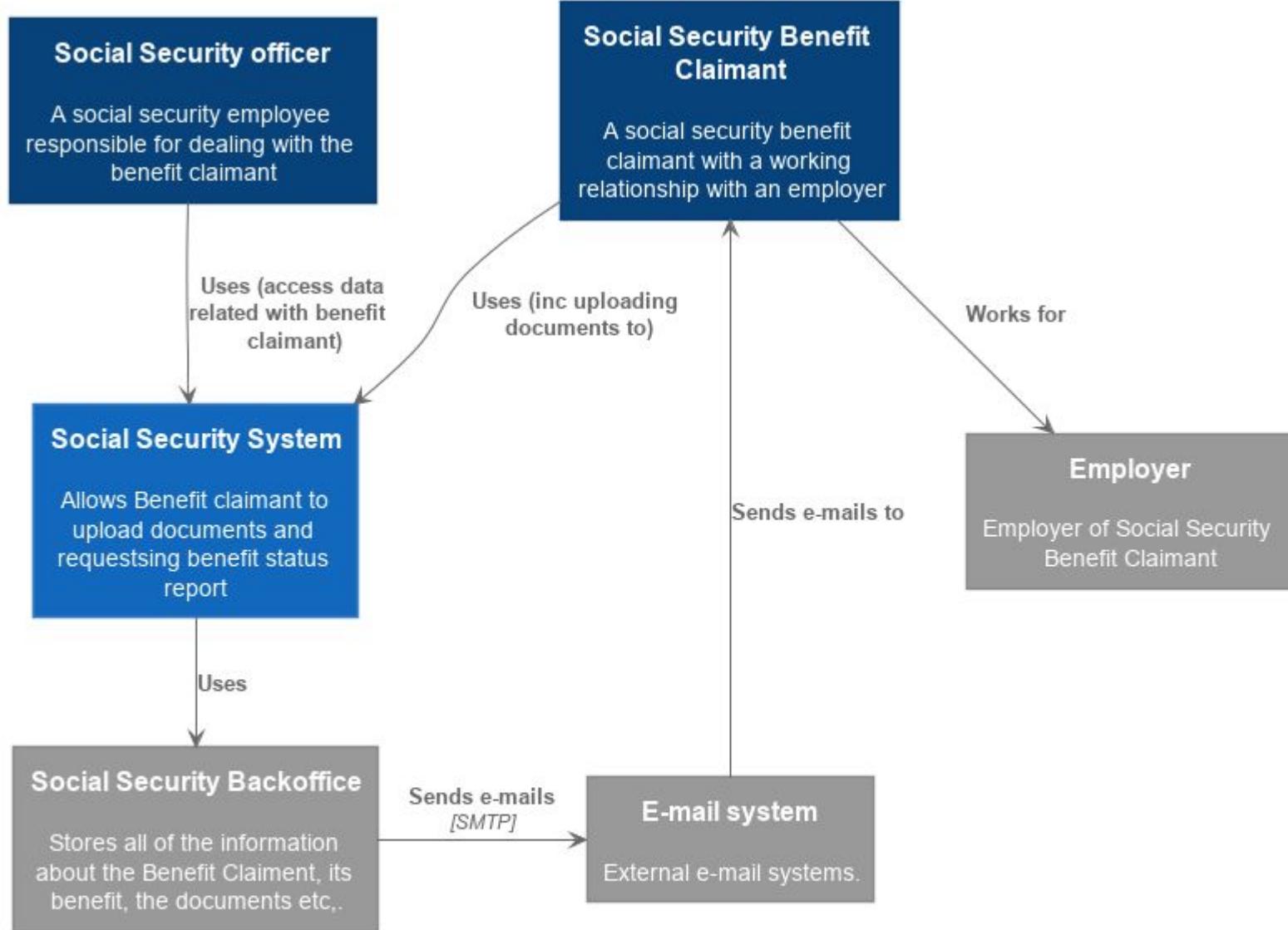
If a benefit claimant has a working relationship with an employer, this has an impact on the benefit. These supporting documents are checked and it is checked whether the data is in accordance with the data made available to the Social Security from the employers.

The application can also be used to obtain an official, signed, report about any change of the benefit requested by or granted to the benefit claimant.

The Social Security Officer responsible for the case of the claimant should be able to access the information through an internal application.

System Context

System Context diagram for Social Security Use Case



Type
person
external person
system
external system

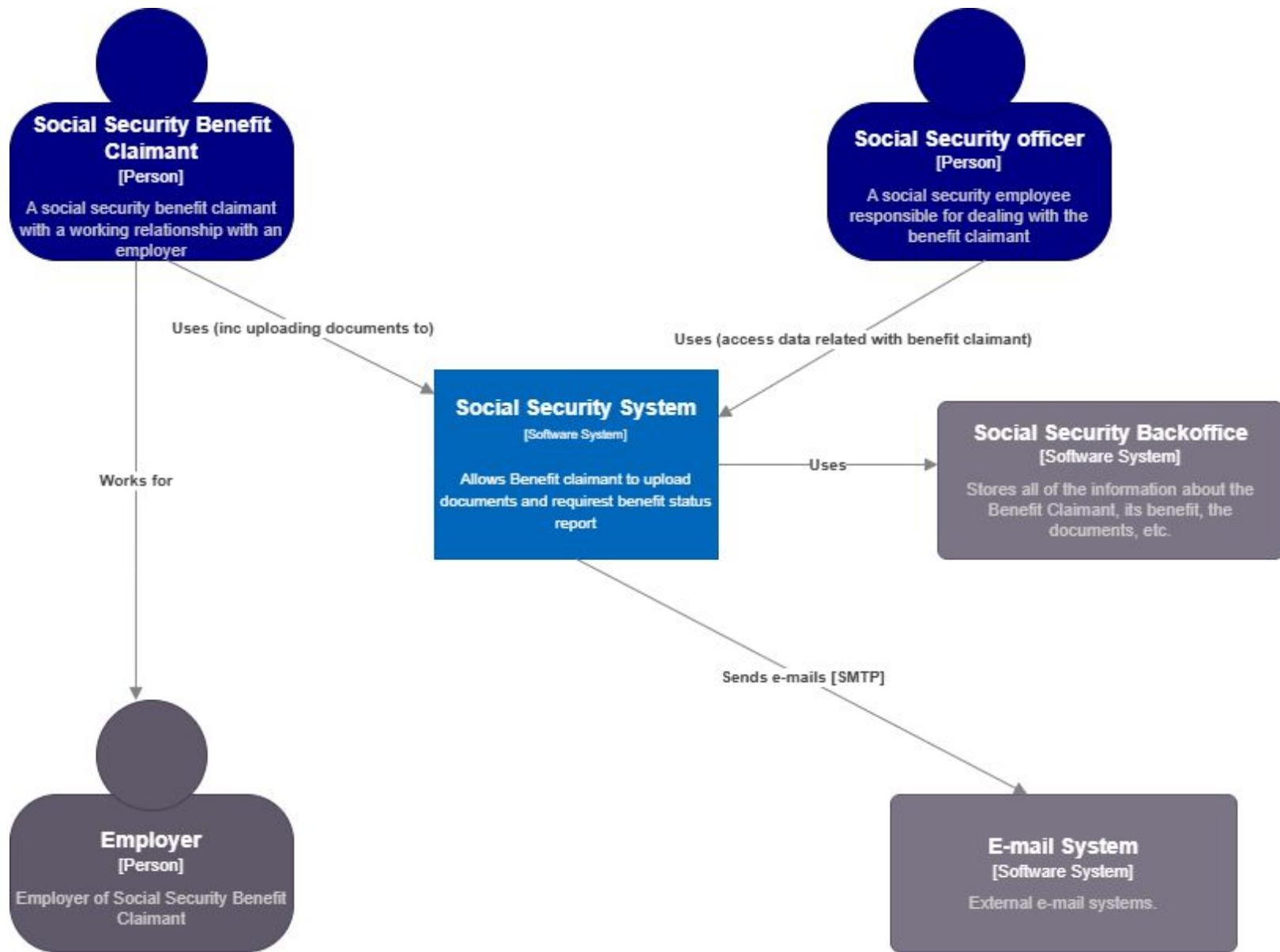


PlantUML

System Context

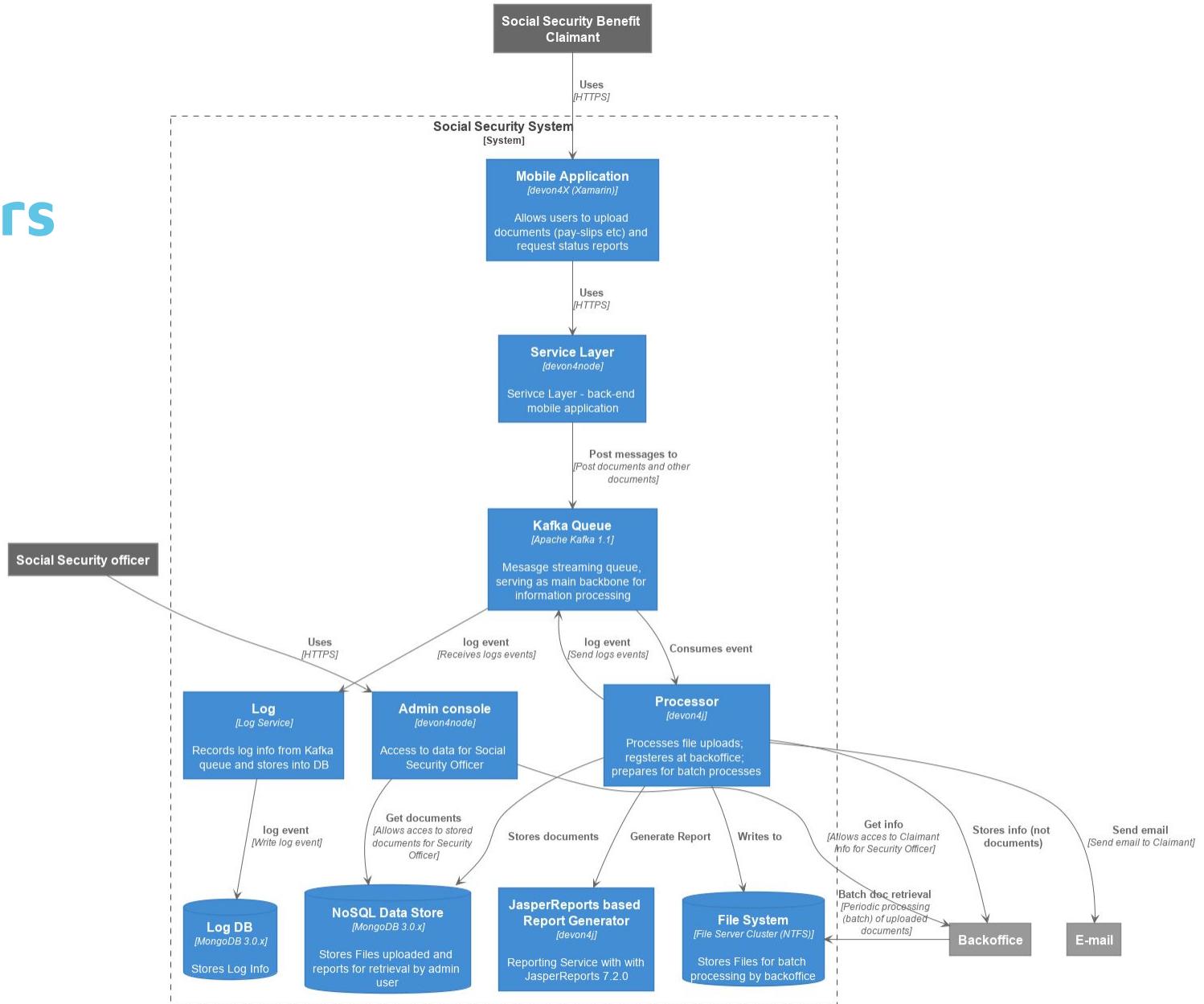


draw.io



Containers

Container Diagram for Social Security Use Case



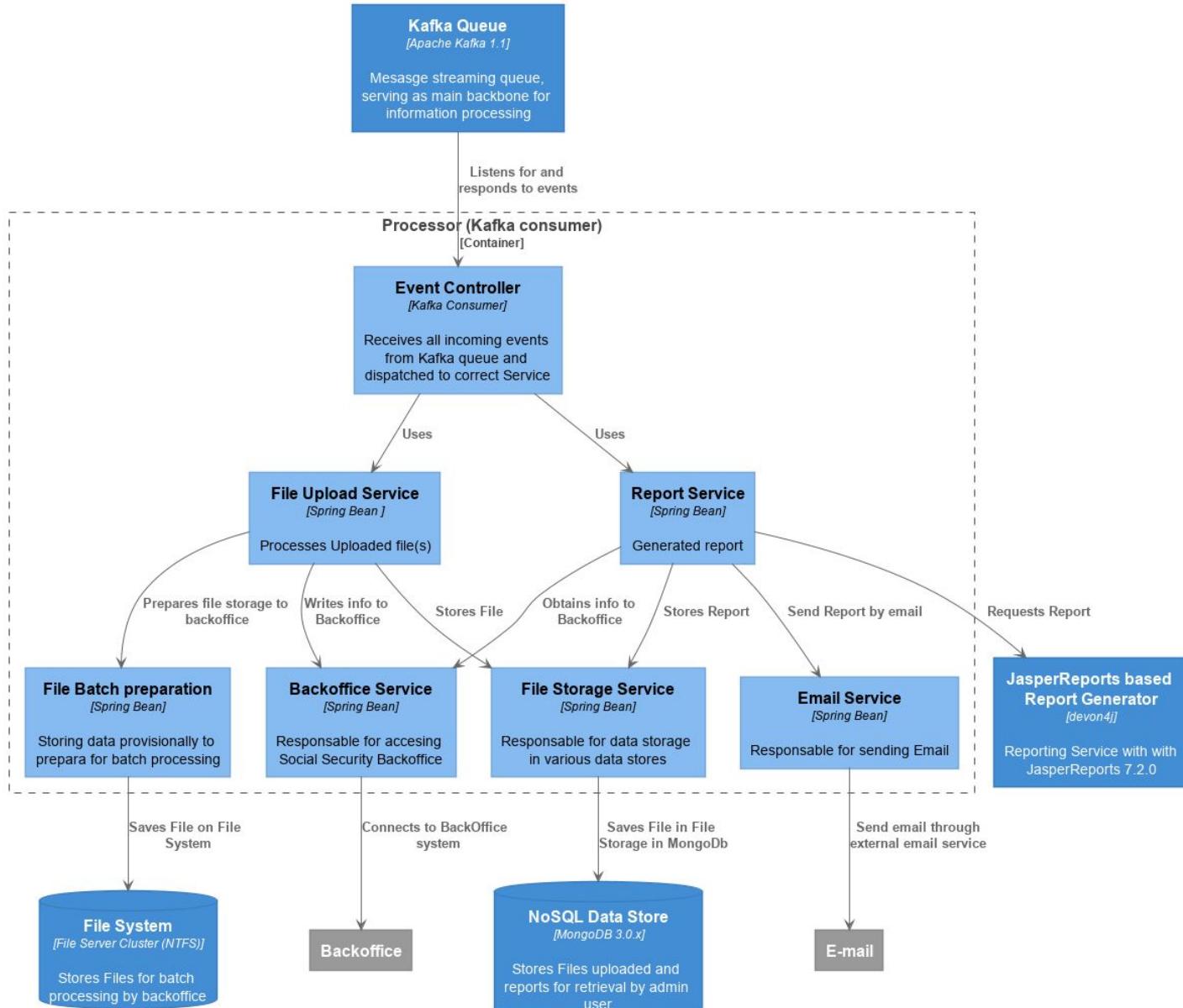
Type
person
external person
system
external system
container



Components



Component diagram for Social Security Use Case - Processor



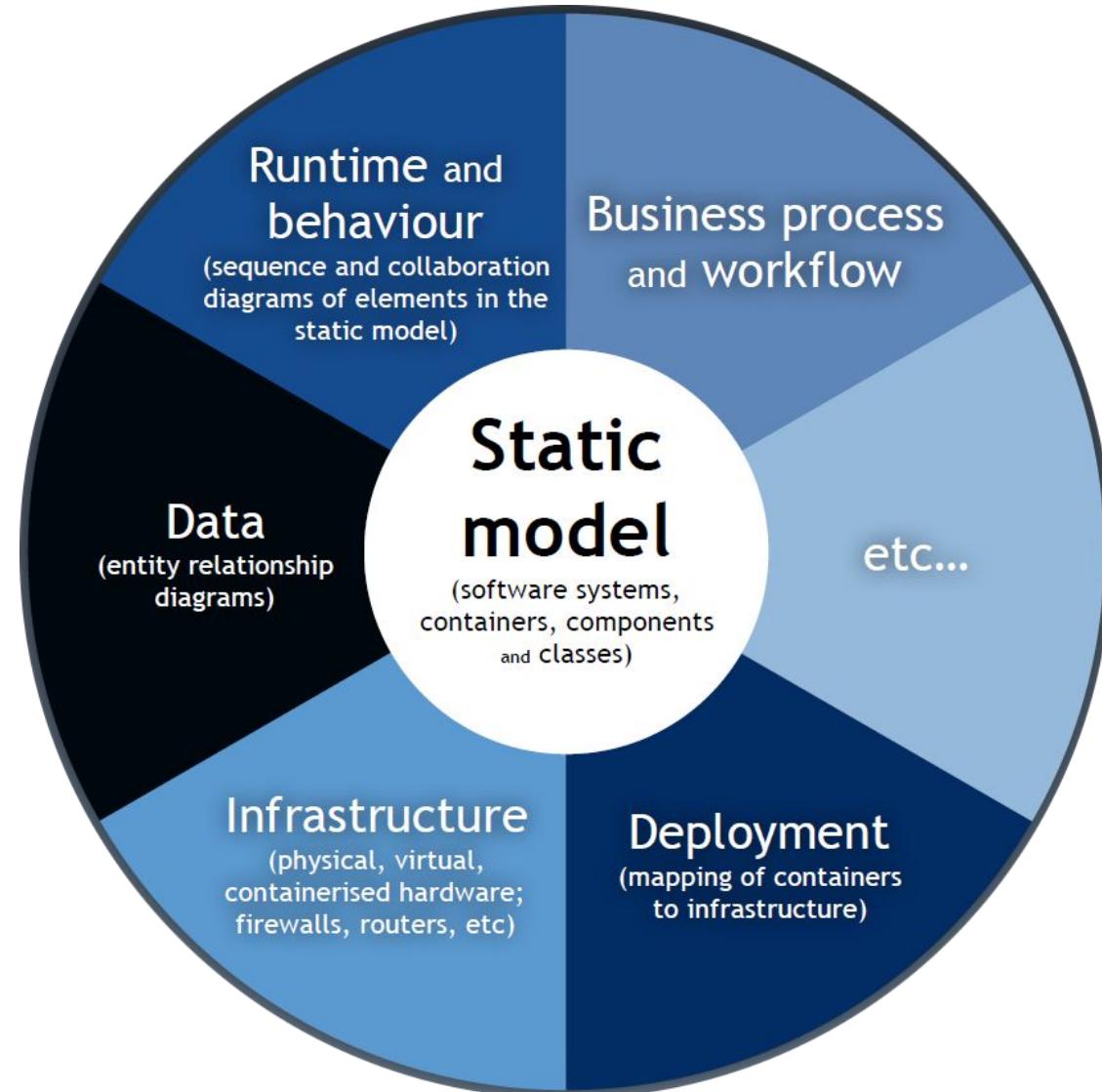
Type
person
external person
system
external system
container
component



Extend the model

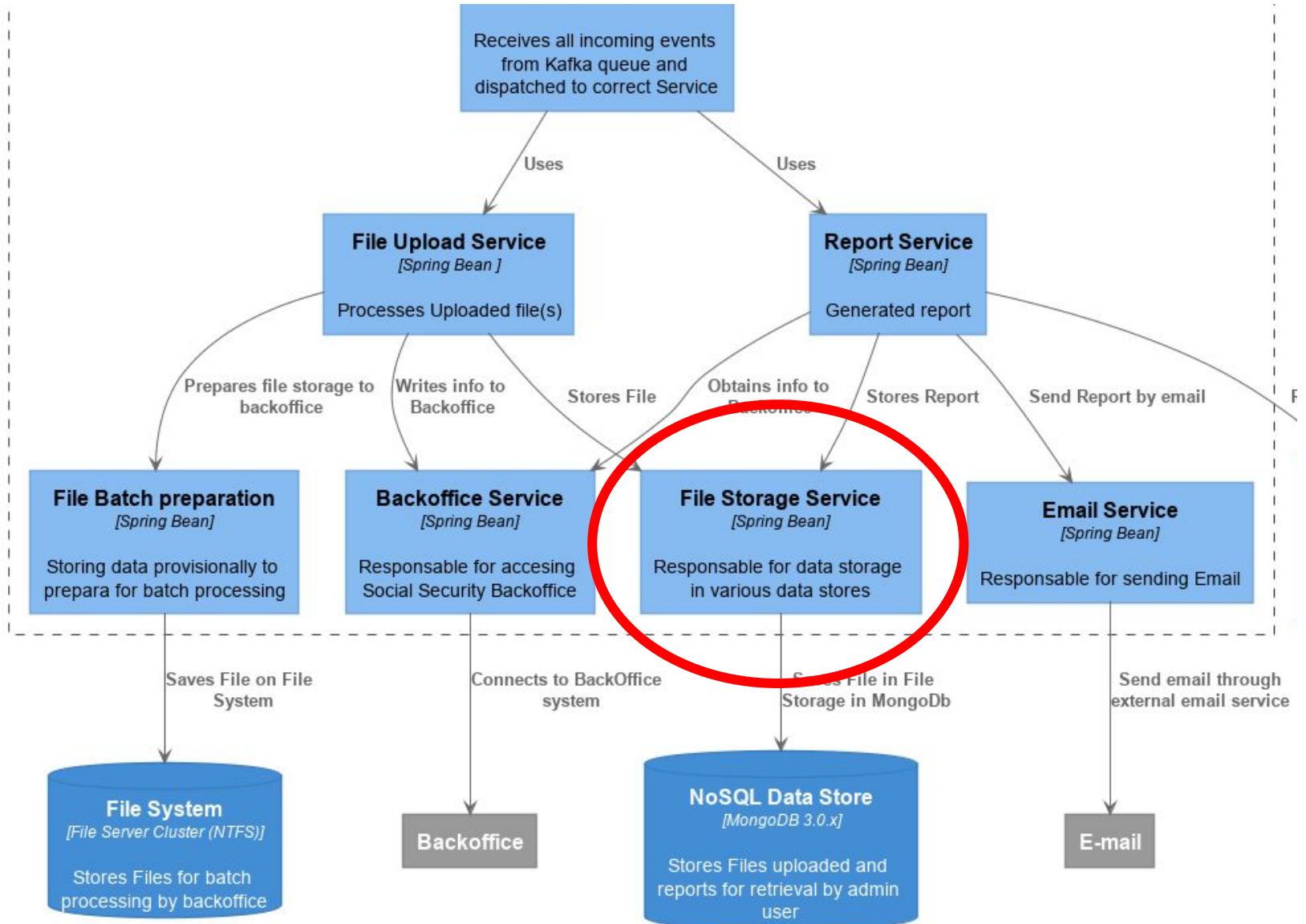
Based on the static model, the Design can be enriched with other diagram types.

UML and other (in)formal standards (ERD) can play a very useful role





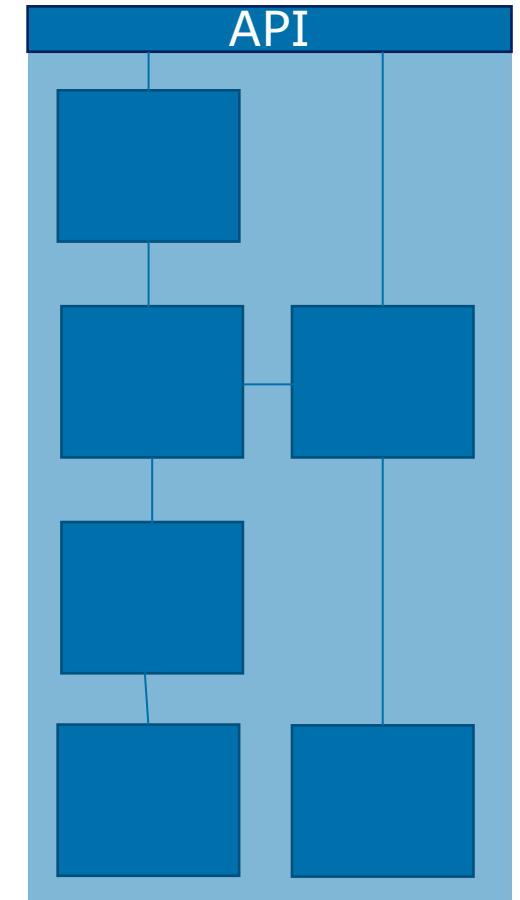
How to design a Component





Component

- Definition of a component in the C4 model
- Should be the smallest building block of the “Architecture”
- It should contain a clearly defined API / **Interface**
- Will typically consist of multiple classes and functions
- A component should restrict access to its subcomponents (i.e.. Java packages, .NET Assemblies, ECMAScript/TypeScript modules etc)
- “Visible as” Angular Service, Spring Bean, Node module
- It should typically be a global singleton and not Support instantiation (multiple instances)
- Should have a Functional or Service Oriented Interface rather than OOP (no maintaining of state between invocations!)





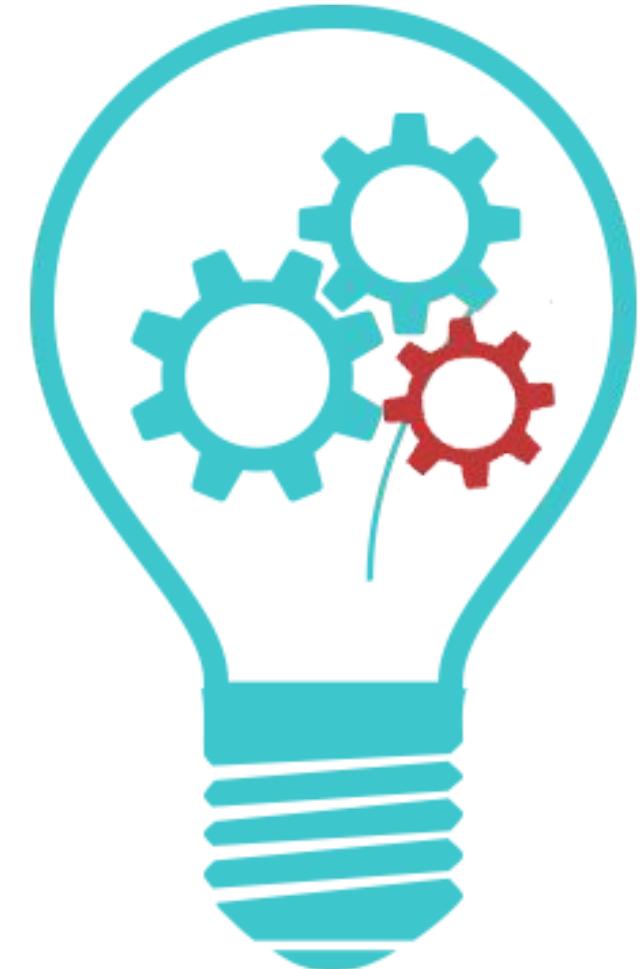
Create abstractions to define meaning (functionality)

Apart from the structure of the system, the design also needs to describe the functionality of the system.

The basis for this form the Functional Requirements for the applications as contained in the Specifications like Use Cases or User Stories.

As with the “Structure” of the Architecture, there needs to be a clear link to defined functionality and Code.

And telling the story works here as well....





System Design as a story

As a user generating a **document**,
I need to be able to **store** the document,
so that it can be **retrieved** at any later moment by
myself or other parties

“I can have a **storage unit** when I can **store** documents.
I or other parties can **retrieve** documents from the store
For this the stored document needs to be represented by a **unique ID**”

Use abstractions!
Postpone the “definitions” of these concepts

```
interface Storage {  
    save(arg0: Store, arg1: Document) : ID;  
    get(arg0: Store, arg1: ID): Document;  
}
```



System Design as a story

As a user generating a **document**,
I need to be able to **store** the document,
so that it can be **retrieved** at any later moment by
myself or other parties

“I can have a **storage unit** when I can **store** documents.
I or other parties can **retrieve** documents from the store
For this the stored document needs to be represented by a **unique ID**”

Use abstractions!
Postpone the “definitions” of these concepts

```
@runtime_checkable
class Storage1(Protocol):
    def save(self, arg0: Store, arg1: Document) -> ID:
        pass

    def get(self, arg0: Store, arg1: ID) -> Document:
        pass
```





Elaborating the Story

“I can have **storage unit** when I can **store and retrieve documents of an undeterminable length and/or varying size**. Basically I should consider them to be a **stream of data**.”

“This stream of data can be either a **stream of bytes** or a **stream of utf-8 encoded text**. As the stream does not represent a “Document” I can accompany it with a corresponding set of **Document attributes**”

```
type Data = Text | Binary
interface Storage {

    save(arg0: Store, arg1: Stream<Data>, arg2: DocumentAttrs) : ID;
    get(arg0: Store, arg1: ID): Stream<Data>;
    getAttr(arg0: Store, arg1: ID): DocumentAttrs;
}
```



Elaborating the Story

“I can have **storage unit** when I can **store and retrieve documents of an undeterminable length and/or varying size**. Basically I should consider them to be a **stream of data**. “

“This stream of data can be either a **stream of bytes** or a **stream of utf-8 encoded text**. As the stream does not represent a “Document” I can accompany it with a corresponding set of **Document attributes**”

```
Data = Union[Text, Binary]
Stream = TypeVar('Stream')

@runtime_checkable
class Storage2(Protocol):
    def save(self, arg0: Store, arg1: Stream[Data], arg2: DocumentAttrs) -> ID:
        pass

    def get(self, arg0: Store, arg1: ID) -> Stream[Data]:
        pass

    def getAttr(self, arg0: Store, arg1: ID) -> DocumentAttrs:
        pass
```





Refining the Story

“I can have **storage unit** when I can **store** and **retrieve documents of an undeterminable length and/or varying size without having to wait for the completion of the operation”**

“I need to be able to **initiate an asynchronous operation**. In case of both an asynchronous as well as a synchronous operation, I need to be **advised of the success of the operation.”**

```
interface Storage {  
  
    save(arg0: Store, arg1: Stream<Data>, arg2: DocumentAttrs) : Promise<ID>;  
    get(arg0: Store, arg1: ID): Promise<Stream<Data>>;  
    getAttr(arg0: Store, arg1: ID): Result<DocumentAttrs>;  
  
    remove(arg0: ID): Result<void>  
}
```

“Finally, I should be able to **remove the stored document.”**



Refining the Story

“I can have **storage unit** when I can **store** and **retrieve documents of an undeterminable length and/or varying size without having to wait for the completion of the operation”**

“I need to be able to **initiate an asynchronous operation**. In case of both an asynchronous as well as a synchronous operation, I need to be **advised of the success** of the operation.”

“Finally, I should be able to **remove the stored document.**”

```
@runtime_checkable
class Storage(Protocol):
    def save(self, arg0: Store, arg1: Stream[Data], arg2: DocumentAttrs) -> Awaitable[ID]:
        pass

    def get(self, arg0: Store, arg1: ID) -> Awaitable[Stream[Data]]:
        pass

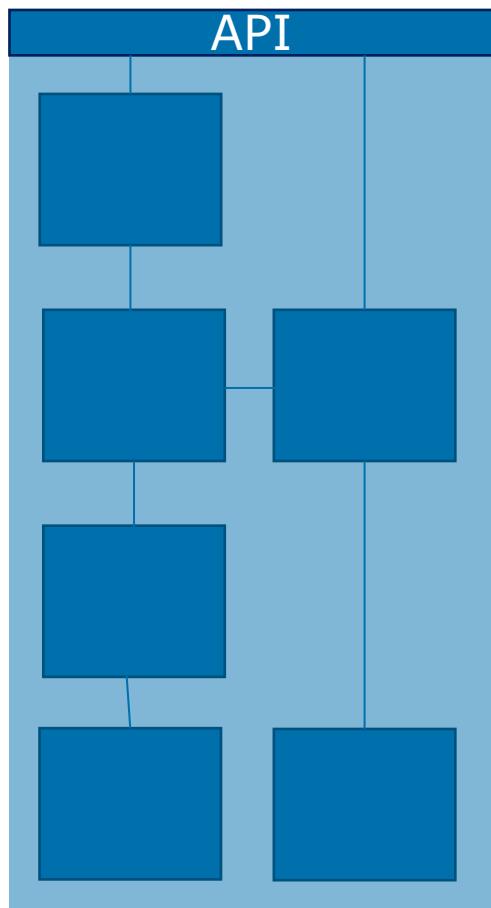
    def getAttr(self, arg0: Store, arg1: ID) -> Result[DocumentAttrs]:
        pass

    def remove(self, arg0: ID) -> Result[None]:
        pass
```





The Storage Interface



```
interface Storage {  
  
    save(arg0: Store, arg1: Stream<Data>, arg2: DocumentAttrs) : Promise<ID>;  
    get(arg0: Store, arg1: ID): Promise<Stream<Data>>;  
    getAttr(arg0: Store, arg1: ID): Result<DocumentAttrs>;  
  
    remove(arg0: ID): Result<void>  
}
```

By having begun as an abstraction, the component is assured to be fully decoupled of – i.e. have no dependencies on – any underlying storage mechanism. In that way:

- It is generic enough to be used by different storage mechanism (File system, Sharepoint, Mongo, etc etc)
- It is testable
- And “pluggable”



Further reading and Viewing...

The lost art of software design by Simon Brown



The lost art of software design



Simon Brown
Twitter: @simonbrown

<https://youtu.be/36OTe7LNd6M>

Large-Scale Architecture: The Unreasonable Effectiveness of Simplicity

by Randy Shoup



The image shows a screenshot of a video player interface. The main content area displays a slide with a dark blue background and white text. The title 'Immutable Log' is centered in large, bold, serif font. Below the title is a bulleted list:

- Store state as immutable log of events
 - Event Sourcing
- Often matches domain
 - E.g., Stitch Fix order processing / delivery state

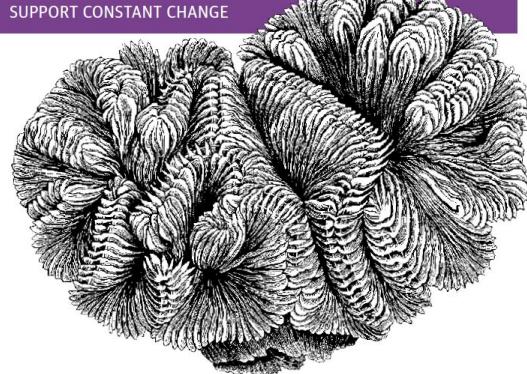
To the left of the list, there are two icons: a blue rounded square icon and a green icon consisting of several vertical bars of decreasing height from left to right. In the bottom left corner of the slide, there is a small watermark-like text '@randyshoup'. In the bottom right corner of the slide, there is a small text 'SUBSCRIBE'. The top right corner of the slide contains the text 'YOW!' in large white letters and '#YOW22' below it. The overall background of the video player has a dark blue gradient with a faint image of a landscape at the top. The video player interface includes standard controls like play/pause, volume, and progress bar at the bottom.

<https://youtu.be/oejXFgvAwTA>

Software Architecture in Practice

THIRD EDITION

Building Evolutionary Architectures



Neal Ford, Rebecca Parsons & Patrick Kua

O'REILLY®



Fundamentals of Software Architecture

An Engineering Approach

Mark Richards & Neal Ford

Further reading

PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE

MARTIN FOWLER
WITH CONTRIBUTIONS BY
DAVID RAES,
MATTHEW FOONAMEL,
EDWARD HIBEST,
ROBERT MEIR, AND
RANDY STAFFORD



Robert C. Martin Series

Clean Architecture

A Craftsman's Guide to Software Structure and Design

Robert C. Martin

Foreword by Kevin Henney
Afterword by Jason Gorman

PRENTICE HALL

O'REILLY®

Building Microservices

DESIGNING FINE-GRAINED SYSTEMS



Sam Newman

Domain-Driven DESIGN

Tackling Complexity in the Heart of Software



Eric Evans

Foreword by Martin Fowler



Domain Driven Design

A established, relatively formal, methodology for such language driven design is **Domain Driven Design** as introduced by Eric Evans in the 2000's

See: Domain Driven Design by Jimmy Bogard

<https://www.youtube.com/watch?v=U6CeaA-Phqo>

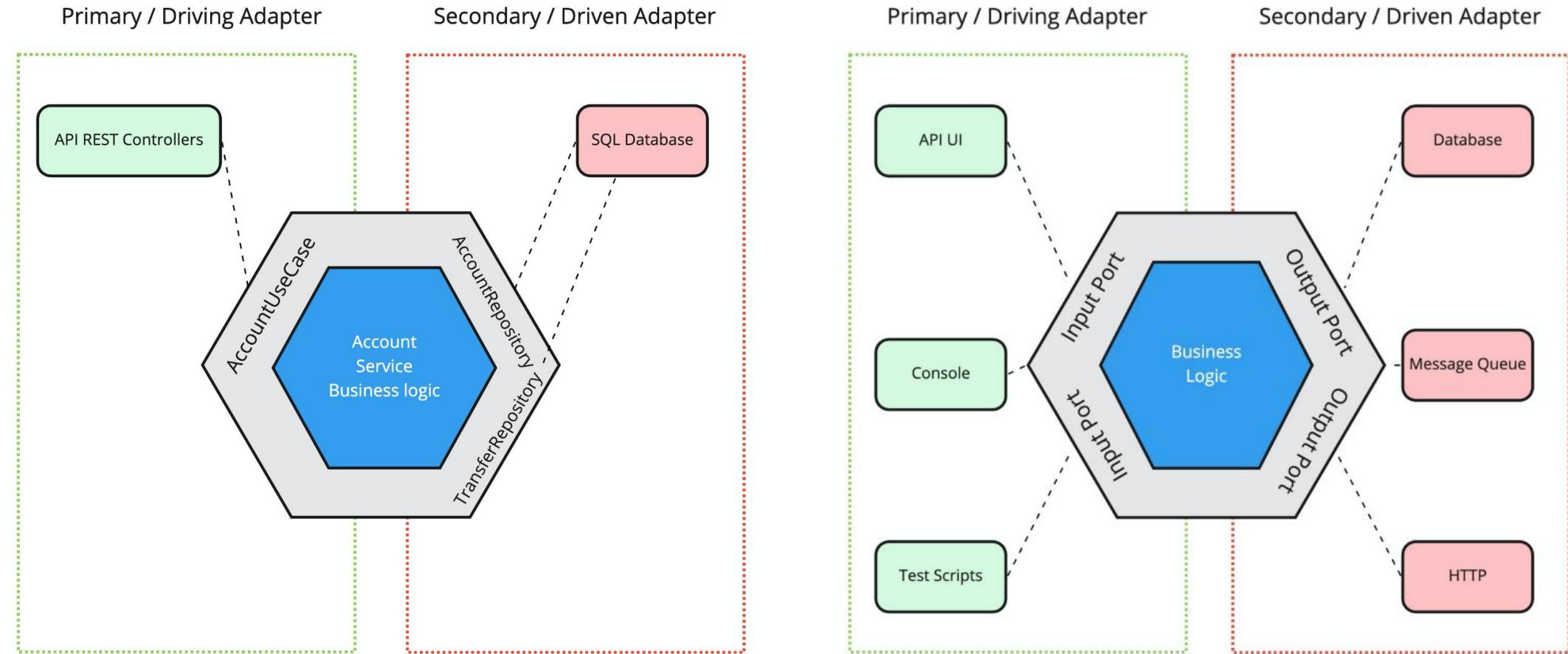
The image shows a YouTube video player interface. At the top, it says "Premium ES". To the right is a search bar with a magnifying glass icon. Below the video frame, there are navigation controls: a play button, a double arrow, a single arrow, and a volume icon. The video progress bar shows "29:18 / 58:38". In the bottom right corner of the video frame, there are several small icons: a square with a dot, a red square, a white square, a black square, and a double square.

The video content itself features a speaker at a podium during a presentation titled "NDC [Sydney]". The slide has the number "6" and the text "NDC { Sydney }". To the right of the video frame is a large, futuristic cityscape with tall, multi-layered skyscrapers and glowing lights, labeled "Reality".

Domain Driven Design: The Good Parts - Jimmy Bogard



Ports and Adapters



<https://medium.com/idealo-tech-blog/hexagonal-ports-adapters-architecture-e3617bcf00a0>



Next Steps for those in a hurry

Facilitating The Spread Of Knowledge And Innovation In Professional Software Development | More



En | 中文 | 日本 | Fr | Br
1,274,004 May unique visitors

Development

Architecture & Design

AI, ML and Data Engineering

Culture & Methods

FEATURED: Streaming Machine Learning Reactive Microservices Containers Observability Docker

InfoQ Homepage > Articles > Architecture As Language: A Story

Architecture as Language: A story



LIKE



10



BOOKMARKS



FEB 27, 2008 • 32 MIN READ

by

Markus Völter

[FOLLOW](#)

Abstract

Architecture is typically either a very non-tangible, conceptual aspect of a software system that can primarily be found in Word documents, or it is entirely driven by technology ("we use an XML architecture"). Both are bad: the former makes it hard to work with, and the latter hides architectural concepts behind technology hype.

What can be done? As you develop the architecture, evolve a language that allows you to describe systems based on this architecture. Based on my experience in a number of real-world projects, this makes the architecture tangible and provides an unambiguous description of the architectural building blocks as well as the concrete system while still staying away from technology decisions (which then can be made consciously in a separate step).

The first part of this paper illustrates the idea using a real-world story. The second part summarizes the key points of the approach.

<https://www.infoq.com/articles/architecture-as-language-a-story/>



Summary – Next Steps



Watch the videos

Read up on the **C4 Model**

Install and play around with **PlantUML** and/or **Draw.IO**

Try to model an “*imaginary real/real imaginary*” system with the **C4 Model**

Try to define the interfaces for the internal components

Rinse – Repeat

Read up on **UML** and try to play with it

Read up on **DDD** and try to play with it

Read about **Ports and Adapters**

Try to do the exercise (Appendix)



Deeper...

The Language of the System – Rich Hickey https://www.youtube.com/watch?v=ROor6_NGIWU

The image is a composite of three parts. On the left, a man with glasses and a white shirt stands behind a wooden podium with a Sheraton Raleigh logo, speaking. In the center, a presentation slide has a dark background. It features a large blue and green circular logo at the top left. Below it, the word "relevance" is written next to a small logo. To the right of the logo, the year "2012" is displayed. At the bottom of the slide, there is a photograph of a complex industrial machine with various components, pipes, and a control panel labeled with Chinese characters. On the right side of the slide, the text "Welcome to the Machine" is written in a large, white, sans-serif font. At the bottom of the slide, there is a bulleted list in white text.

- Machines apply force to accomplish work
- That's what systems do!

Summary session 2



Overview:

Session 2 shifts focus to modeling and visualization techniques, emphasizing their role in bridging the gap between design concepts and implementation. It provides practical guidance on tools and methodologies for collaborative and iterative software design.

Modeling Approaches:

- Utilize models like the 4+1 architectural view model to capture different perspectives of a system.
- Employ tools such as Unified Modeling Language (UML) for standard notations (e.g., use case, sequence, and state diagrams) while acknowledging its limitations for non-OOP paradigms.
- Highlight the importance of visualizing software architecture through C4 models and other diagramming approaches.

Challenges and Solutions:

- Model-Code Gap: Addressing differences between architecture models and source code by ensuring abstraction and traceability.
- Promote iterative refinement of abstractions, starting with high-level stories and decomposing into concrete implementations.

Component Design:

- Define components as the smallest architectural building blocks, with clearly defined APIs and functionality.
- Prioritize generic, pluggable, and testable designs, focusing on functional interfaces rather than maintaining state.

Best Practices:

- Use diagrams to visualize and document systems, ensuring alignment between models and code.
- Leverage domain-driven design and type-driven development for clarity in functionality and data modeling.

Next Steps:

- Experiment with modeling tools like PlantUML and other diagramming platforms.
- Study methodologies like Domain-Driven Design and Ports and Adapters for deeper insights.
- Practice defining interfaces and creating prototypes to enhance design understanding.



Appendix

An exercise...



Cognitive Canvas

Collaborative Design and Documentation Platform

Overview

Cognitive Canvas is a collaborative tool designed for software architects and developers, extending the capabilities of Excalidraw (an open-source whiteboard) with added functionality for project management, real-time collaboration, AI-assisted design, and Markdown-based documentation. It aims to streamline workflows by integrating drawing, chat, documents, and AI-generated content into a single, cohesive interface.

Key Features

- **Project Management:** Organize drawings, documents, and team chats under unified projects.
- **Collaborative Drawing:** Real-time editing powered by Excalidraw.
- **AI-Enhanced Design:** Analyze and modify drawings, assist documentation, and summarize project details using AI.
- **Integrated Documentation:** Combine drawings, documents, and AI content into cohesive reports using a Markdown editor (CodeMirror).
- **Extensible Backend:** Modular services for projects, chats, documents, and AI, designed to scale easily.
- **Code Generation:** Graphical elements and metadata are interpreted as software structures, enabling AI to generate starter code.

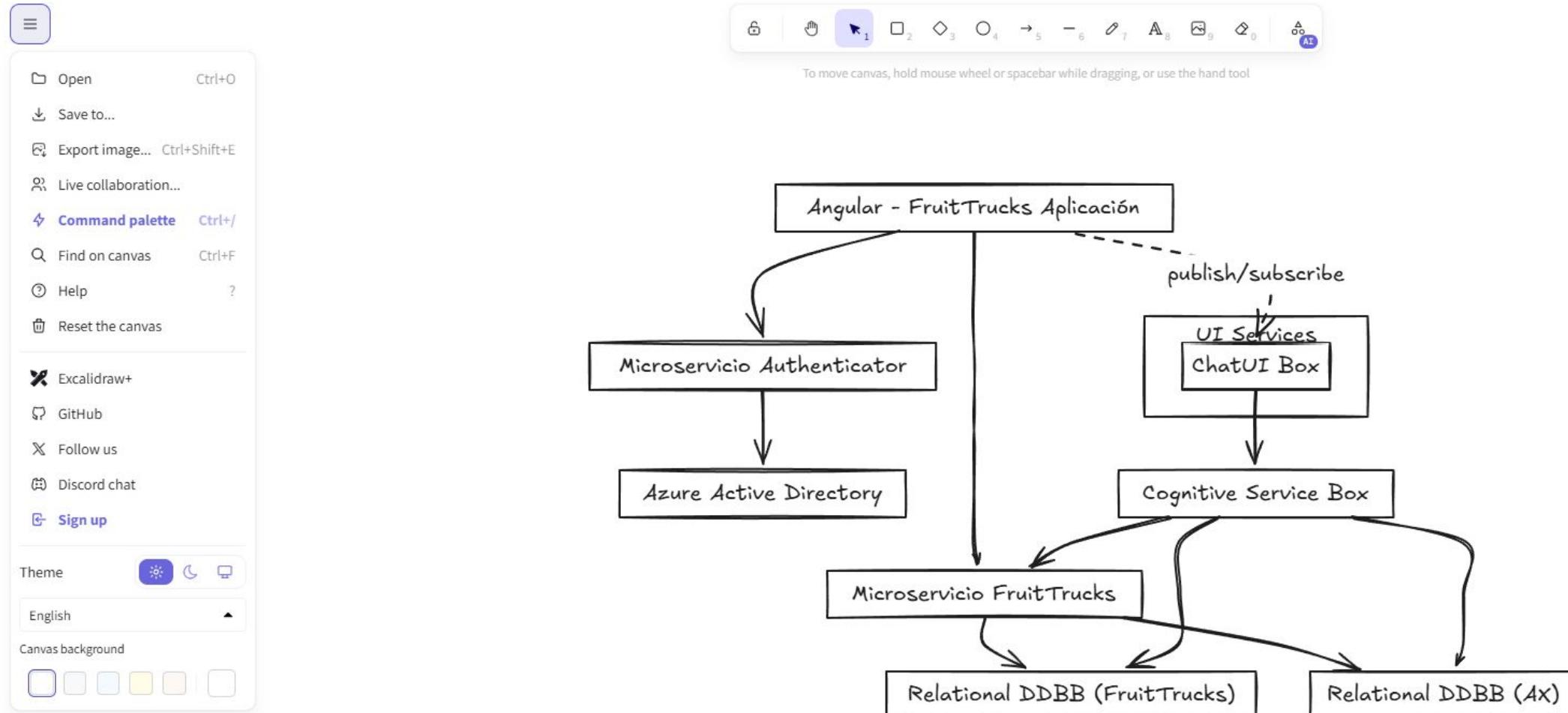
System Design Principles

- **Developer-Oriented:** Prioritizes modularity and extensibility for seamless integration with existing systems.
- **Scalability:** Built to grow with organizational needs, leveraging modern cloud-native solutions.
- **Open-Source Friendly:** Integrates existing components like Excalidraw, CodeMirror, and real-time libraries (e.g., Socket.IO).



Cognitive Canvas

Collaborative Design and Documentation Platform



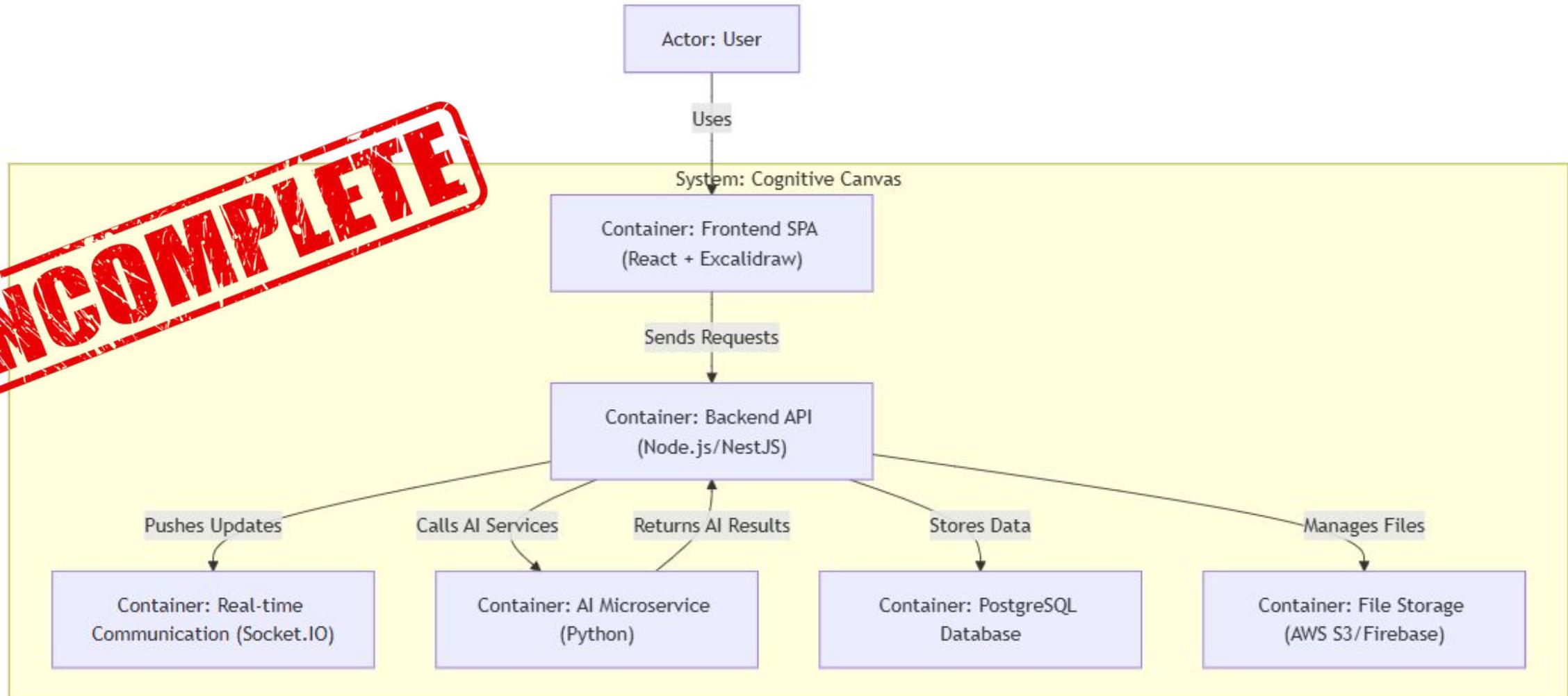
Screenshot Excalidraw

Cognitive Canvas

Collaborative Design and Documentation Platform



INCOMPLETE



Draft solution sketch (incomplete)



Capgemini



Capgemini is a global leader in partnering with companies to transform and manage their business by harnessing the power of technology. The Group is guided everyday by its purpose of unleashing human energy through technology for an inclusive and sustainable future. It is a responsible and diverse organization of over 300,000 team members in nearly 50 countries. With its strong 50-year heritage and deep industry expertise, Capgemini is trusted by its clients to address the entire breadth of their business needs, from strategy and design to operations, fuelled by the fast evolving and innovative world of cloud, data, AI, connectivity, software, digital engineering and platforms. The Group reported in 2020 global revenues of €16 billion.



Get The Future You Want | www.capgemini.com

This presentation contains information that may be privileged or confidential and is the property of the Capgemini Group.

Copyright © 2022 Capgemini. All rights reserved.