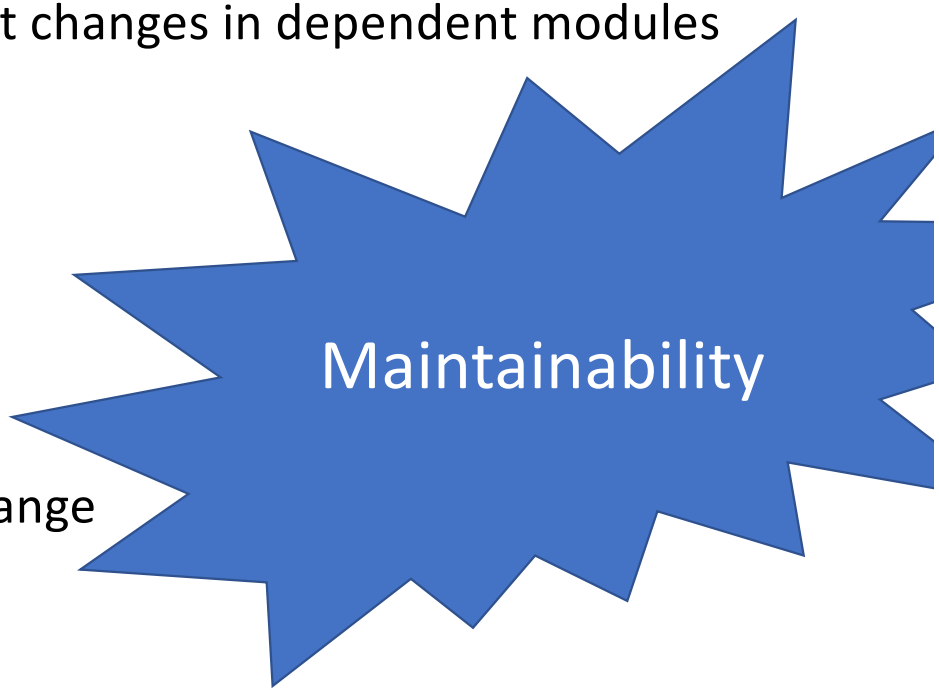
Introduction to Software Architecture

2022-2023

Problems

- Rigidity (difficulty to change)
  - One change causes a cascade of subsequent changes in dependent modules
- Fragility
  - When changed, breaks in many places
- Immobility
  - Inability to reuse software
- Viscosity
  - Resistance against making the adequate change



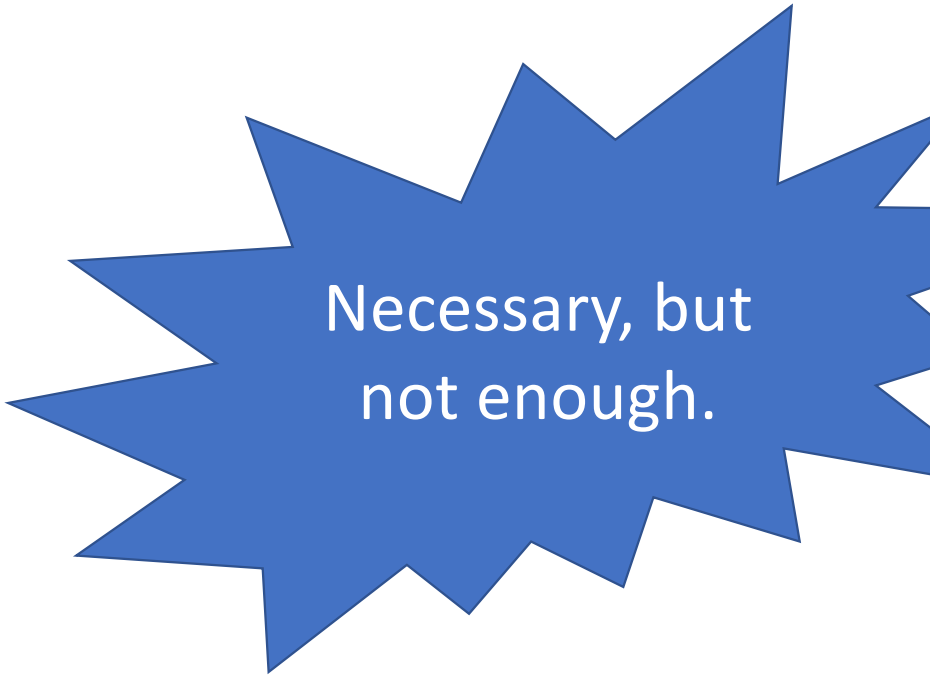
Key architectural principles

Maintainability →
Evolutionary Architectures

- Build to change instead of building to last
- Model to analyze and reduce risk
- Use models to communicate and collaborate
- Identify and describe architectural alternatives and decisions

Styles and Patterns

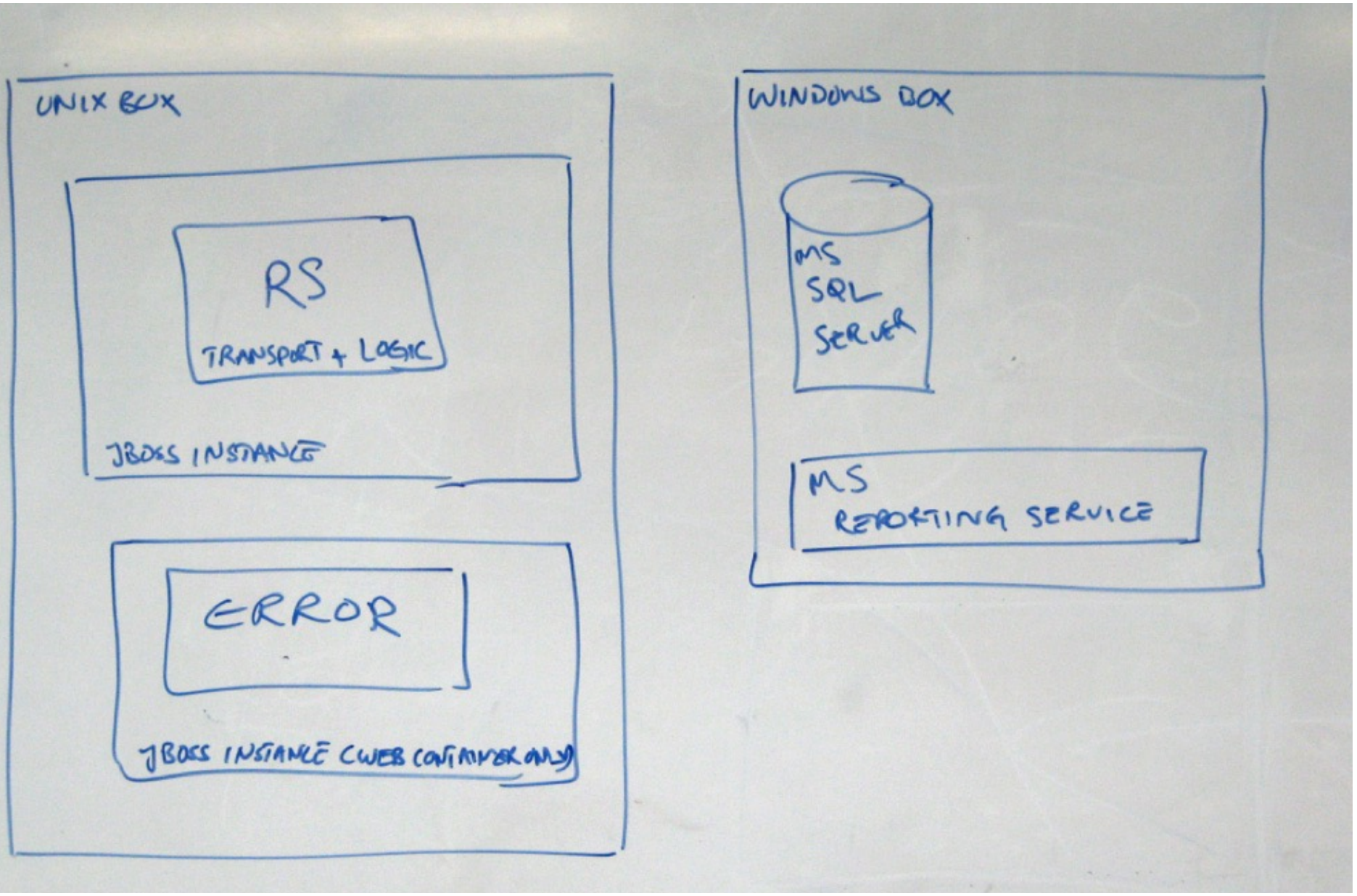
- GRASP
- SOLID
- GoF
- DDD
- MVC
- DTO
- Layered architecture
- Onion Architecture (or Clean architecture)
- SOA
- N-Tiers
- …

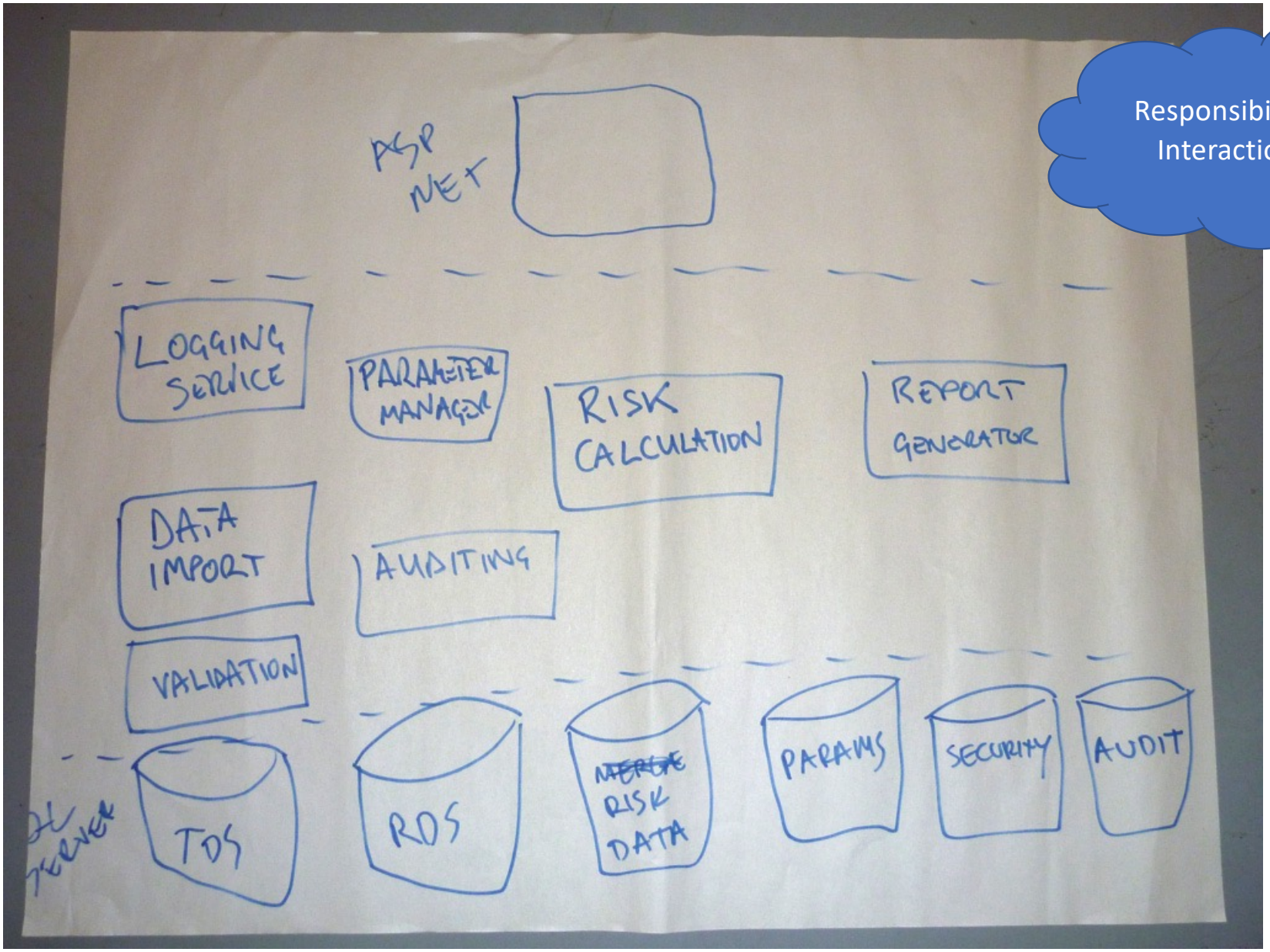


Summary

- Problem
  - Difficulty in abstracting
  - Focus on technology/programming instead of focus in engineering/design
  - Lack of rigor in representing/conveying ideas to stakeholders
- Goal
  - Increase engineering and design competencies
  - Adopt abstraction in representation
  - Adopt rigorous representation
- Approach
  - Adoption of best practices, patterns
  - Adoption of (semi-)formal notation in representation (e.g. UML)

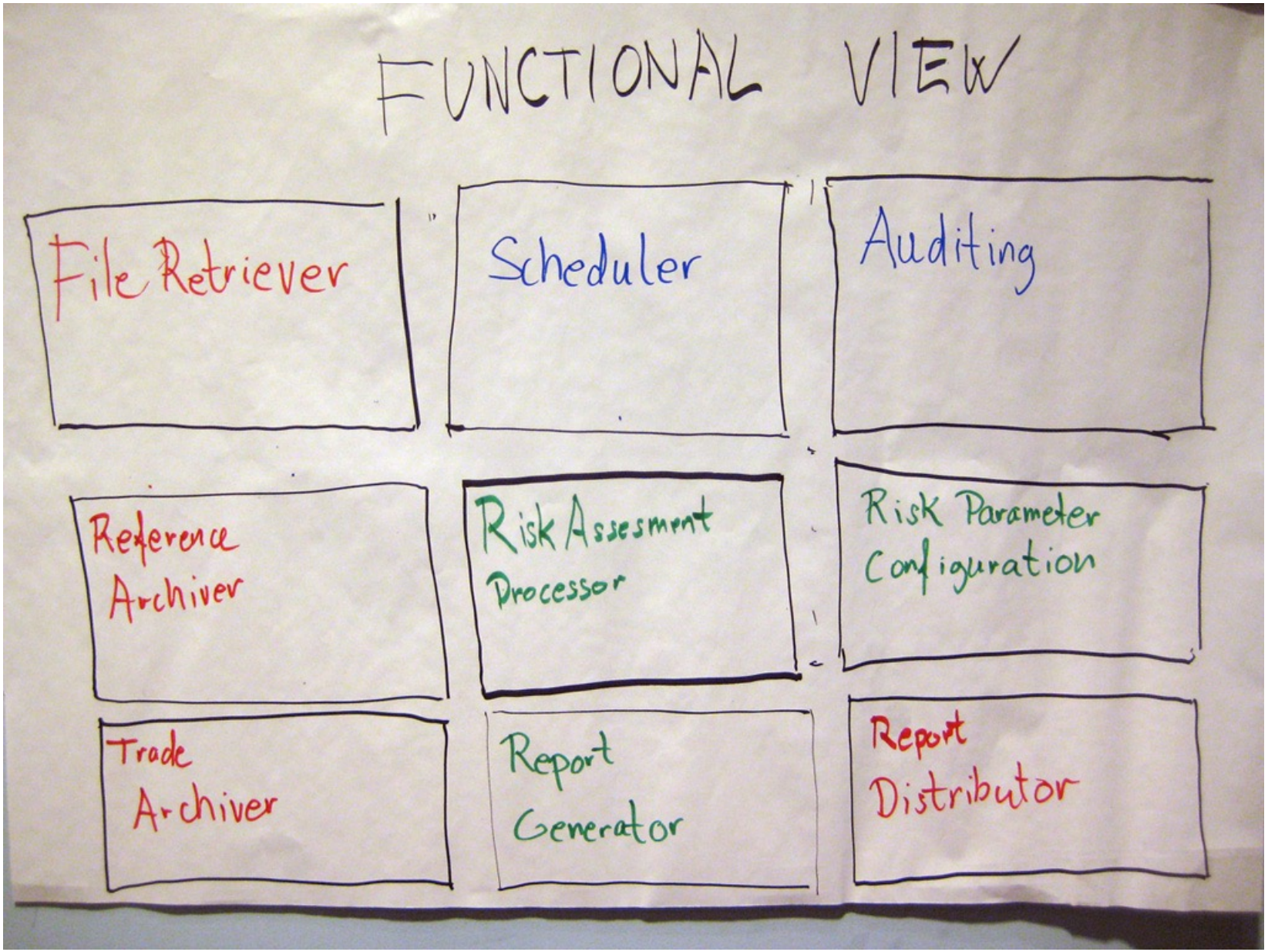
Representação de design (arquitetural) habituais

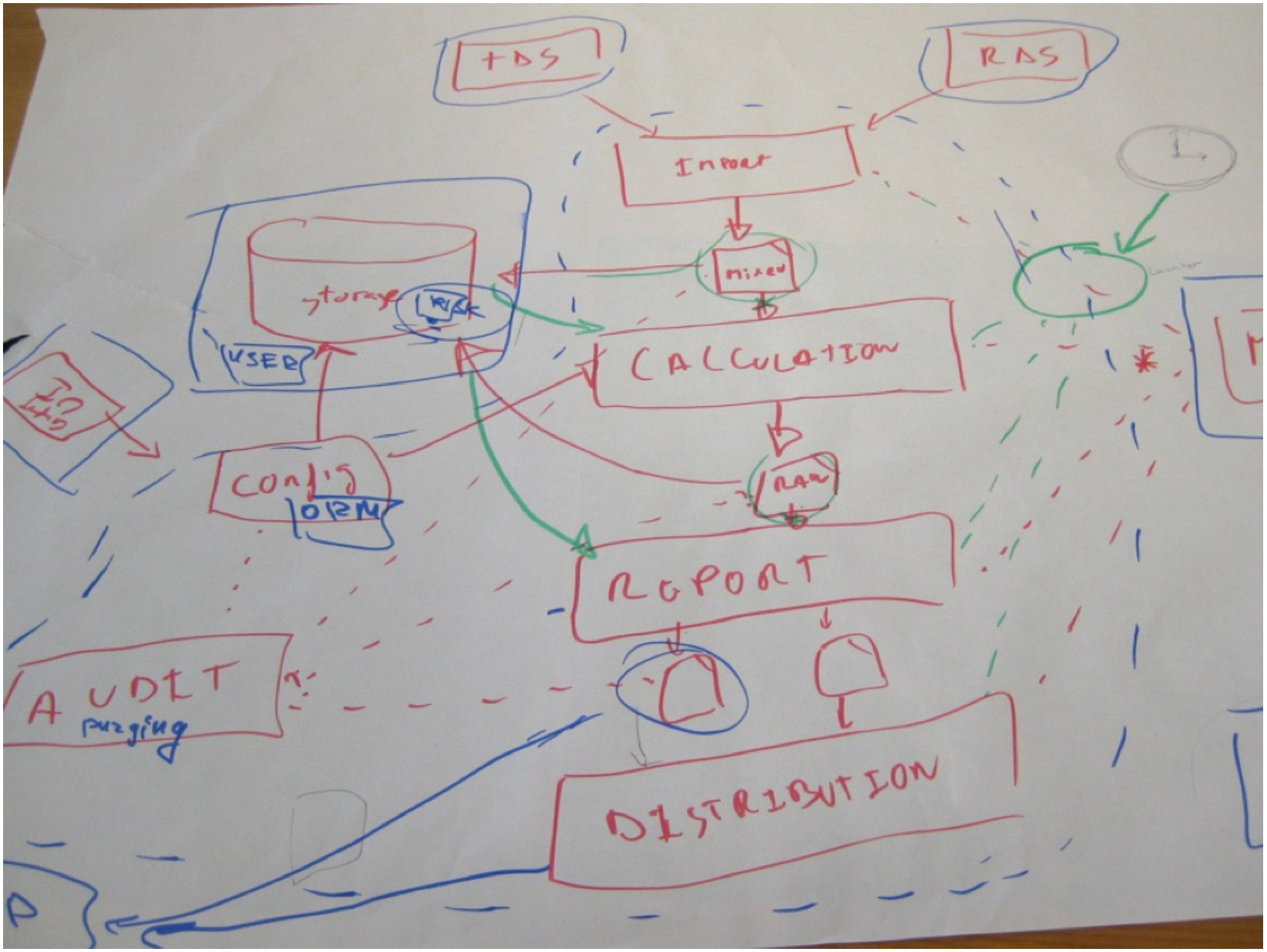

UNIX BOX
RS
TRANSPORT + LOGIC
JBOSS INSTANCE
ERROR
JBOSS INSTANCE (WEB CONTAINER ONLY)
WINDOWS BOX
MS
SQL
SERVER
MS
REPORTING SERVICE

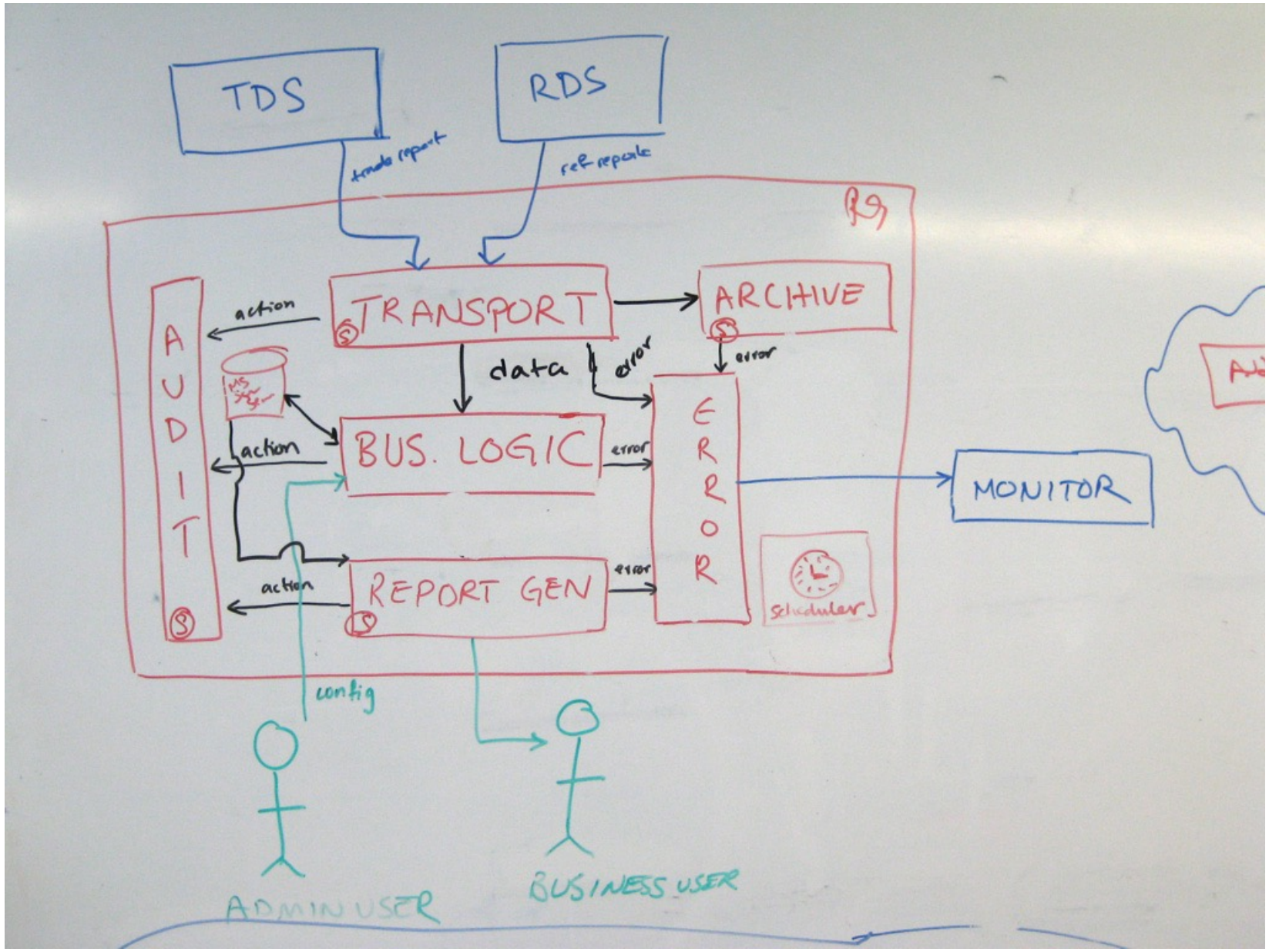


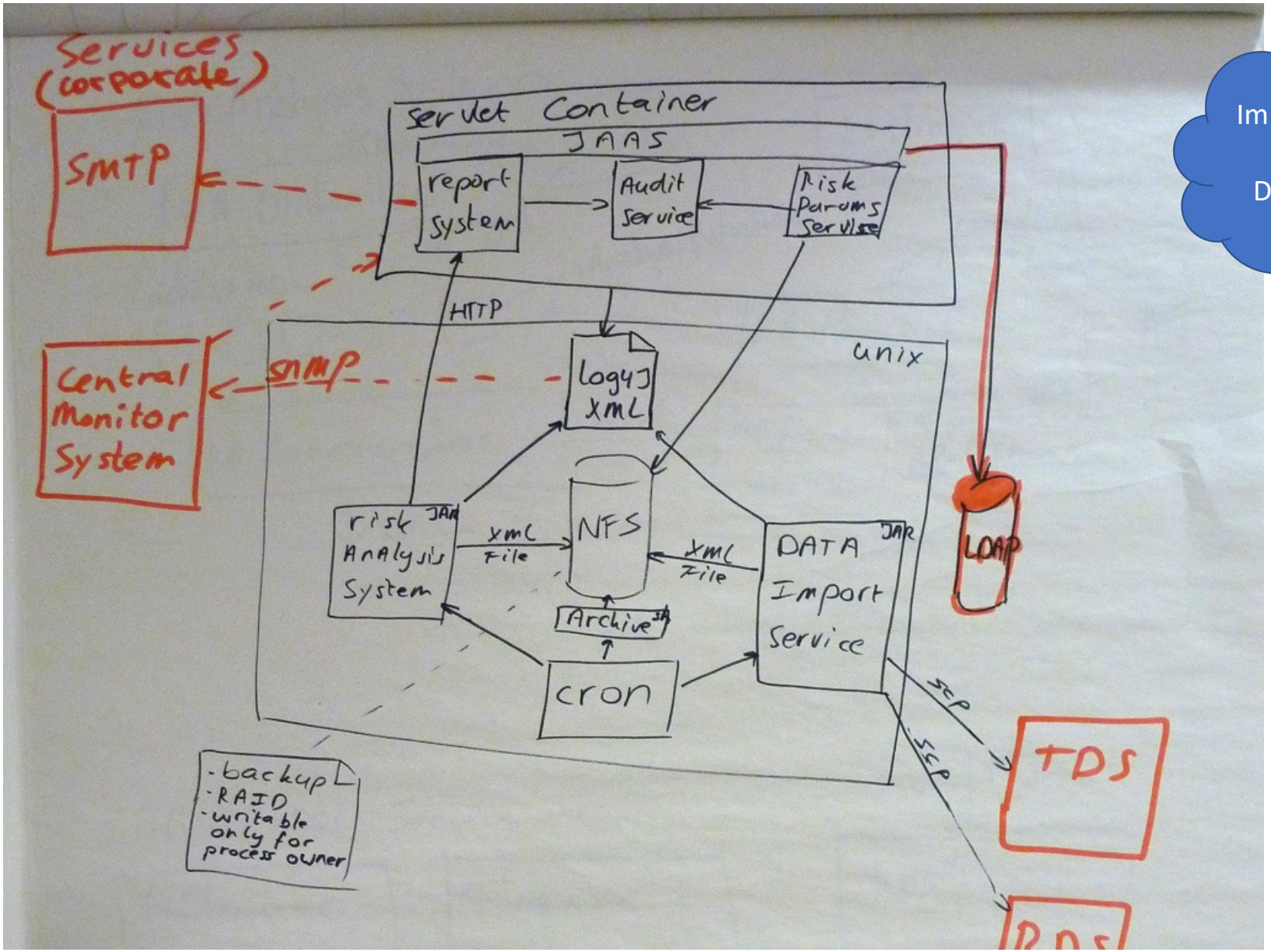
Responsibilities?
Interactions?

FUNCTIONAL VIEW

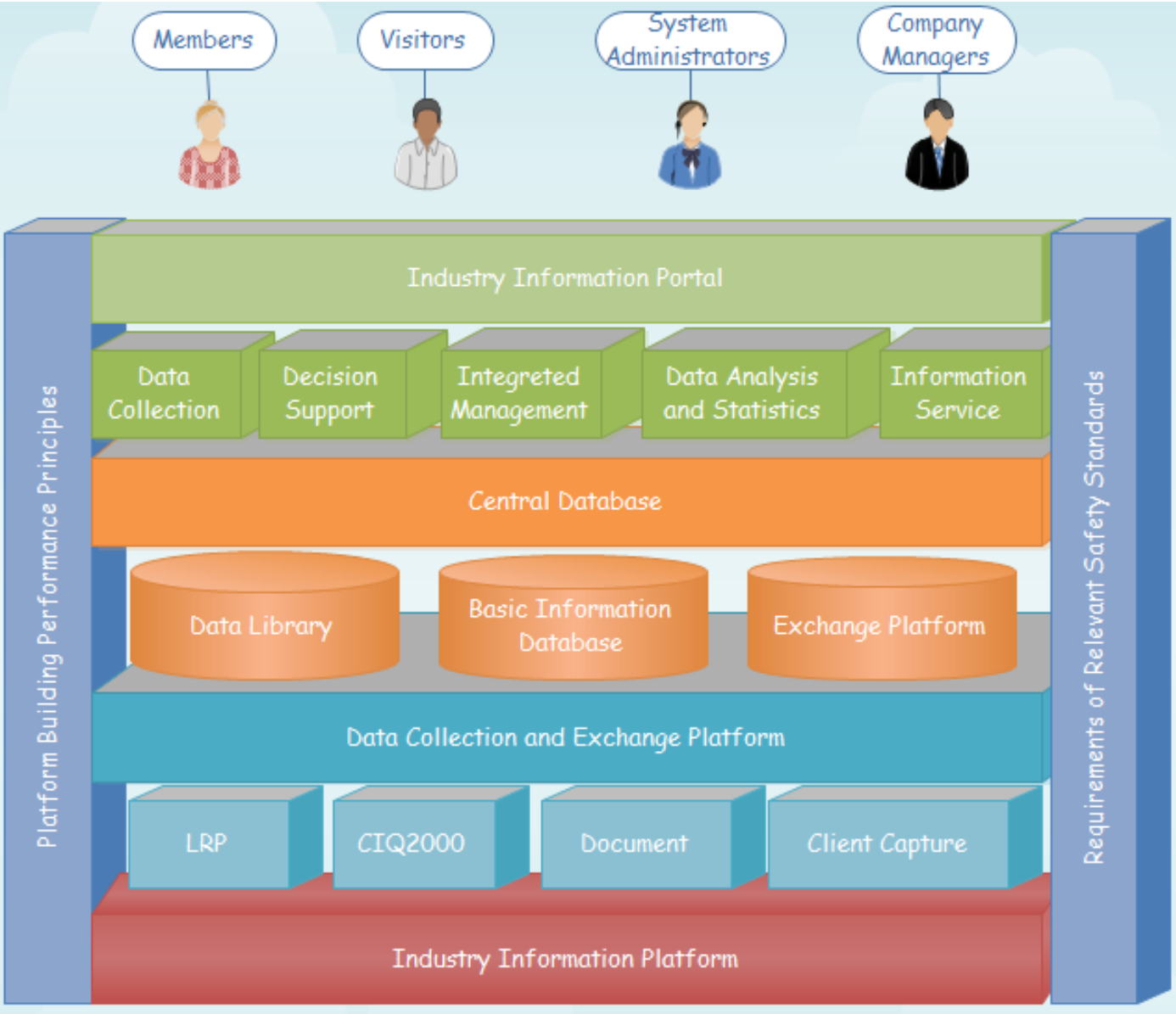


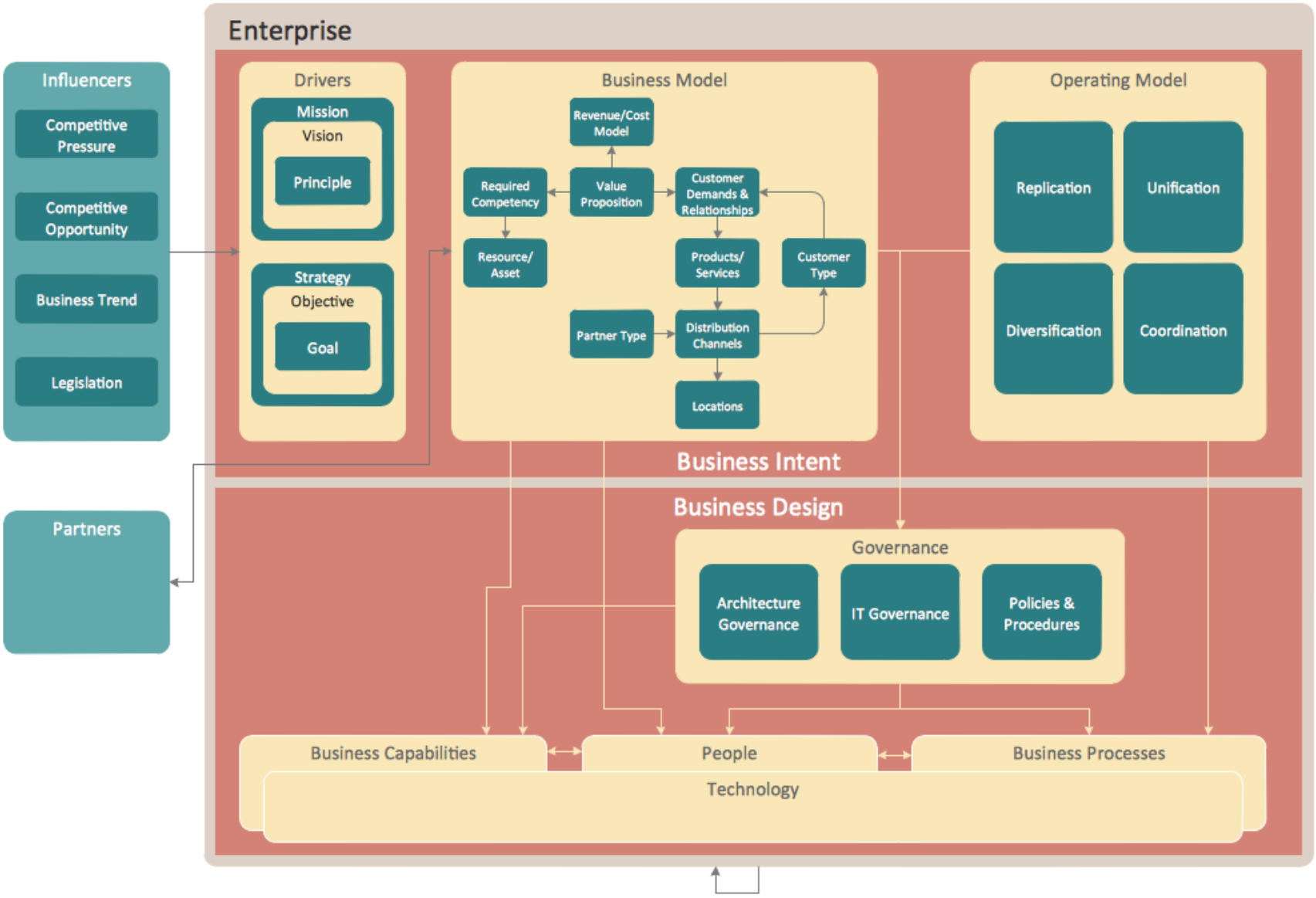

TDS
RDS
Import
mixed
storage
USER
CALCULATION
config
ORM
raw
REPORT
AUDIT
purging
DISTRIBUTION

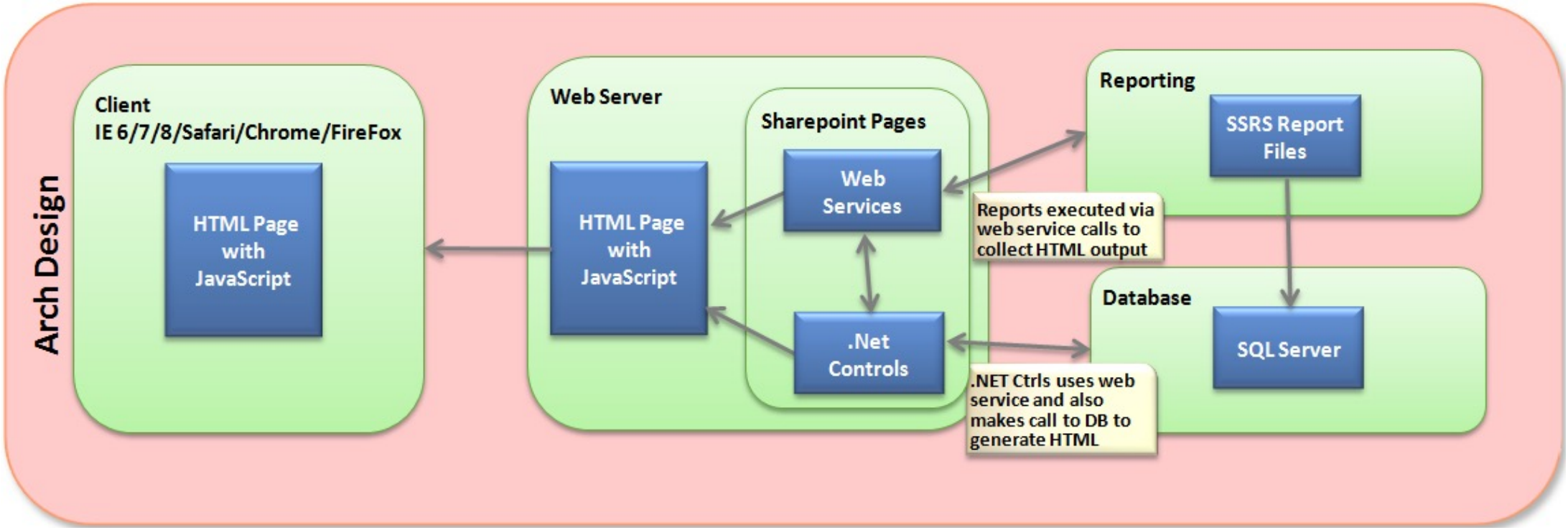

TDS
RDS
trade report
ref repo
TRANSPORT
ARCHIVE
AUDIT
action
data
error
error
BUS. LOGIC
action
error
ERROR
MONITOR
REPORT GEN
action
error
Scheduler
config
ADMIN USER
BUSINESS USER

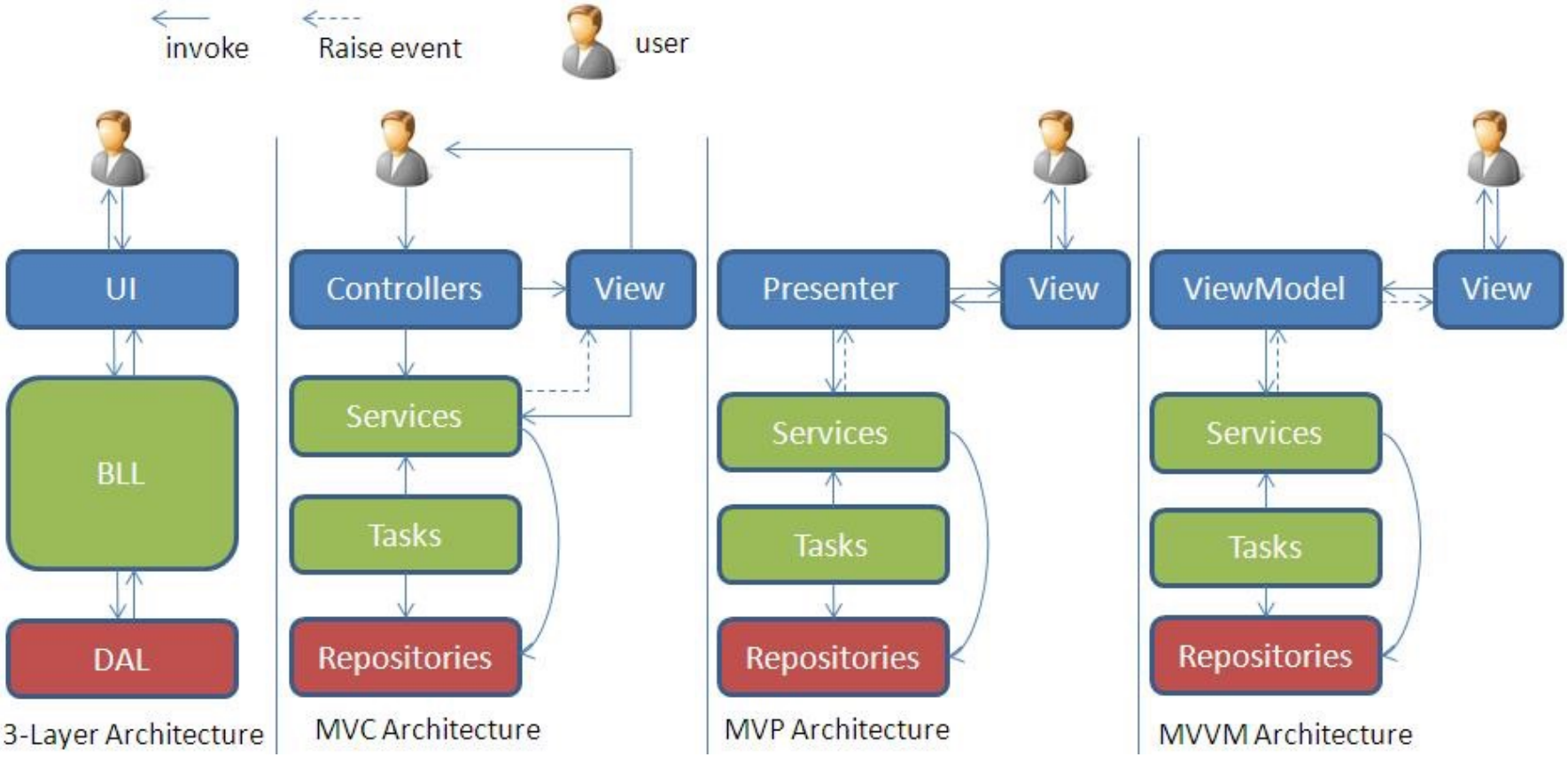


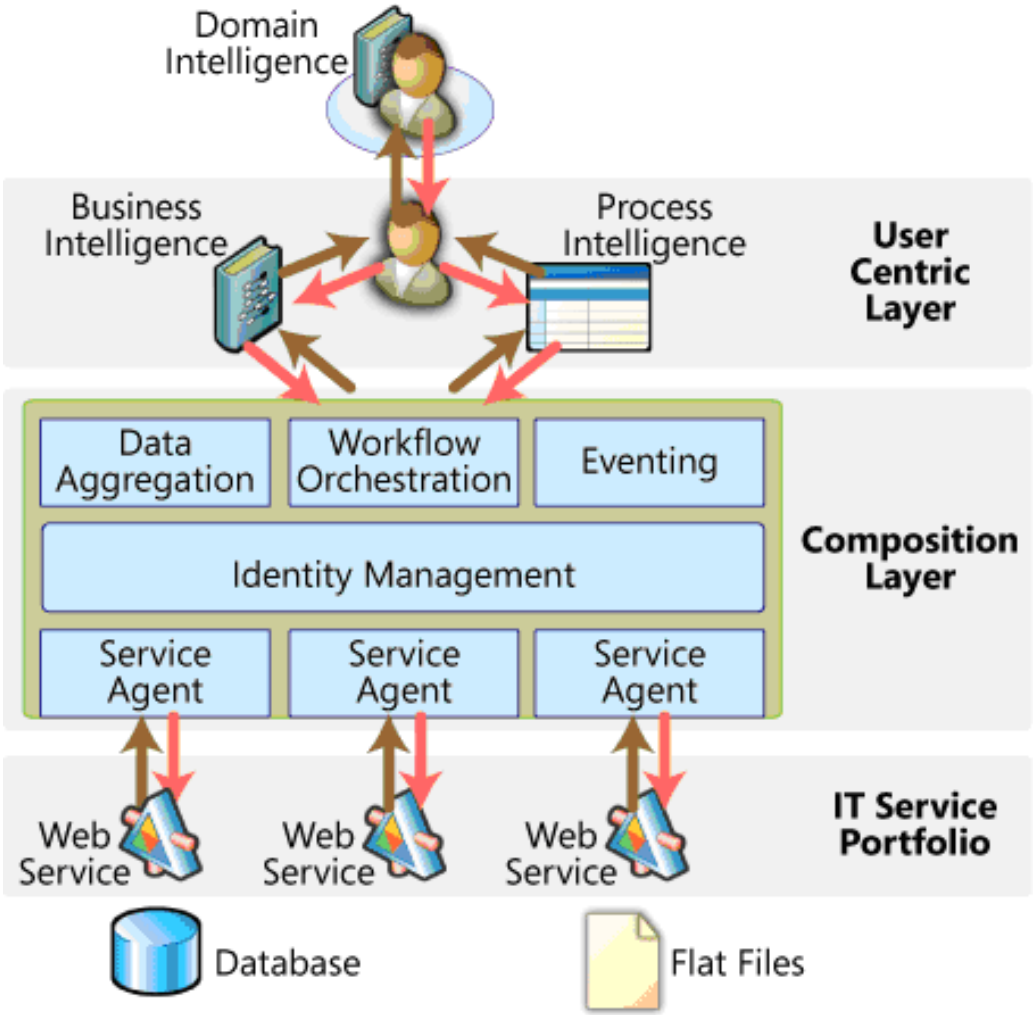
Implementation and Deployment?

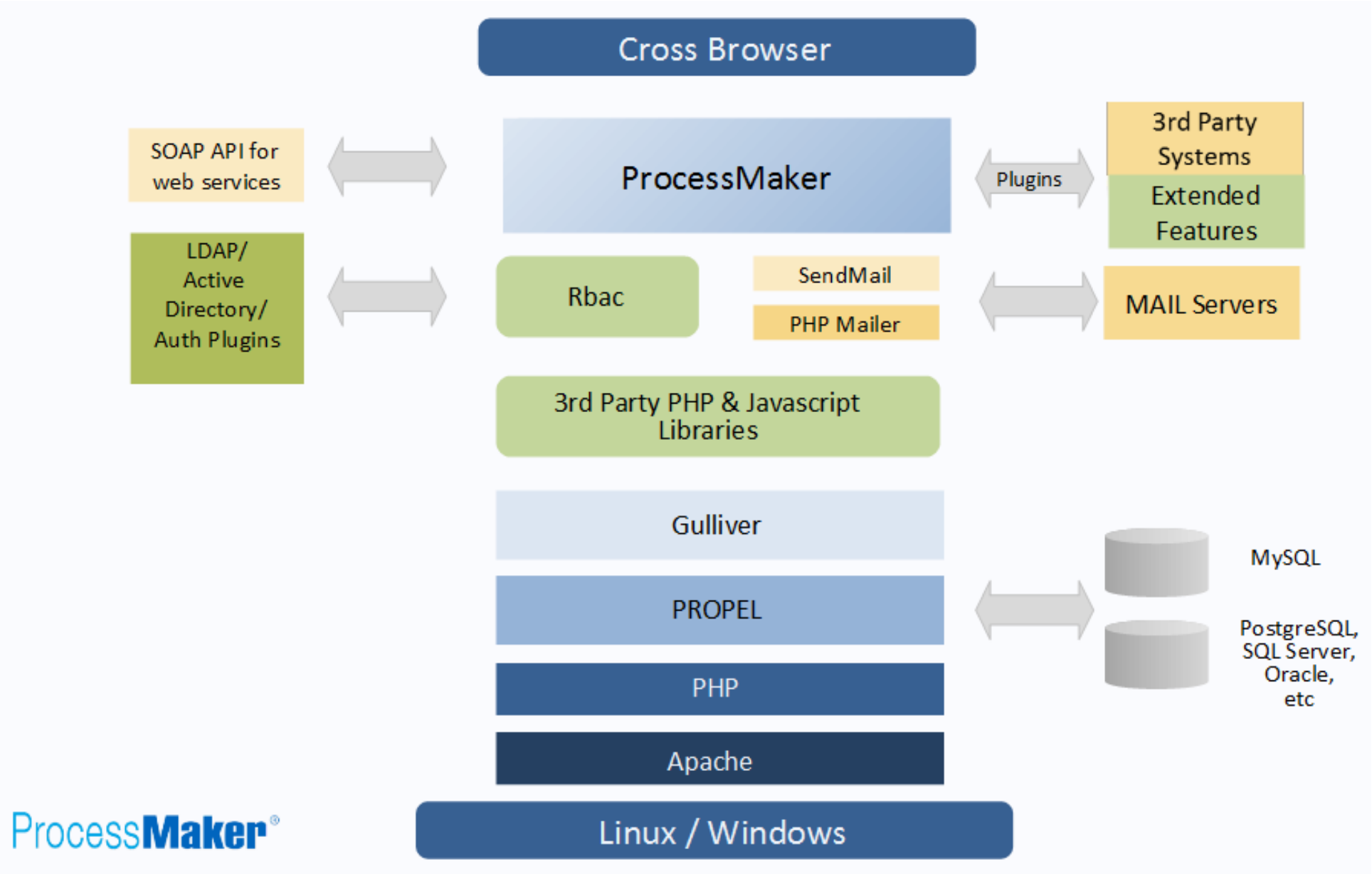

Members
Visitors
System Administrators
Company Managers
Industry Information Portal
Data Collection
Decision Support
Integreted Management
Data Analysis and Statistics
Information Service
Central Database
Data Library
Basic Information Database
Exchange Platform
Data Collection and Exchange Platform
LRP
CIQ2000
Document
Client Capture
Industry Information Platform
Platform Building Performance Principles
Requirements of Relevant Safety Standards

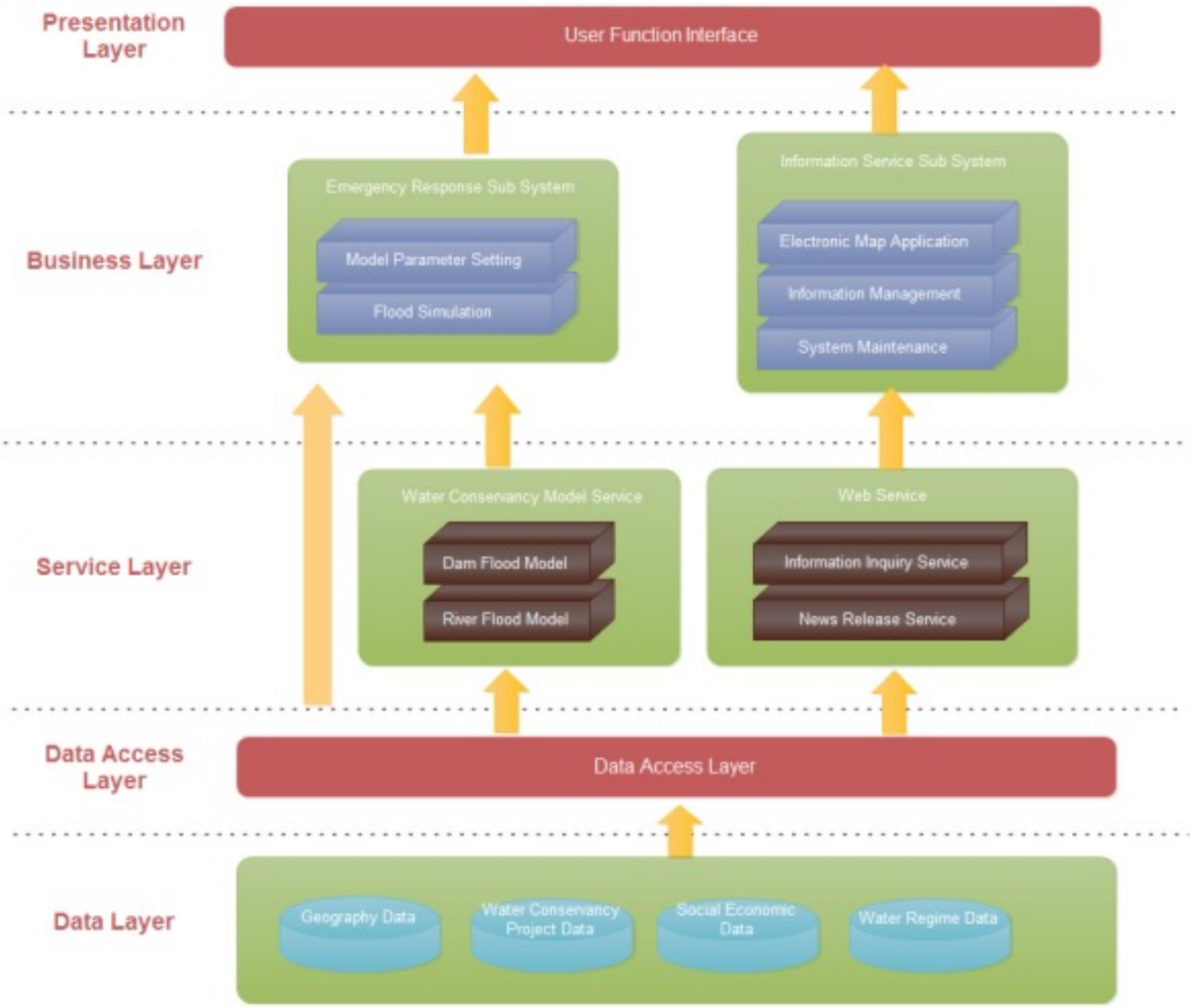

Enterprise
Influencers
Competitive Pressure
Competitive Opportunity
Business Trend
Legislation
Partners
Drivers
Mission
Vision
Principle
Strategy
Objective
Goal
Business Model
Revenue/Cost Model
Required Competency
Value Proposition
Customer Demands & Relationships
Resource/ Asset
Products/ Services
Customer Type
Partner Type
Distribution Channels
Locations
Operating Model
Replication
Unification
Diversification
Coordination
Business Intent
Business Design
Governance
Architecture Governance
IT Governance
Policies & Procedures
Business Capabilities
People
Business Processes
Technology


Arch Design
Client
IE 6/7/8/Safari/Chrome/FireFox
HTML Page with JavaScript
Web Server
HTML Page with JavaScript
Sharepoint Pages
Web Services
.Net Controls
Reporting
SSRS Report Files
Reports executed via web service calls to collect HTML output
Database
SQL Server
.NET Ctrls uses web service and also makes call to DB to generate HTML


invoke
Raise event
user
UI
BLL
DAL
3-Layer Architecture
Controllers
View
Services
Tasks
Repositories
MVC Architecture
Presenter
View
Services
Tasks
Repositories
MVP Architecture
ViewModel
View
Services
Tasks
Repositories
MVVM Architecture


Domain Intelligence
Business Intelligence
Process Intelligence
User Centric Layer
Data Aggregation
Workflow Orchestration
Eventing
Identity Management
Service Agent
Service Agent
Service Agent
Composition Layer
Web Service
Web Service
Web Service
IT Service Portfolio
Database
Flat Files


Cross Browser
SOAP API for web services
ProcessMaker
Plugins
3rd Party Systems
Extended Features
LDAP/ Active Directory/ Auth Plugins
Rbac
SendMail
PHP Mailer
MAIL Servers
3rd Party PHP & Javascript Libraries
Gulliver
PROPEL
PHP
Apache
MySQL
PostgreSQL, SQL Server, Oracle, etc
ProcessMaker®
Linux / Windows


Presentation Layer
User Function Interface
Business Layer
Emergency Response Sub System
Model Parameter Setting
Flood Simulation
Information Service Sub System
Electronic Map Application
Information Management
System Maintenance
Service Layer
Water Conservancy Model Service
Dam Flood Model
River Flood Model
Web Service
Information Inquiry Service
News Release Service
Data Access Layer
Data Access Layer
Data Layer
Geography Data
Water Conservancy Project Data
Social Economic Data
Water Regime Data

Resumo

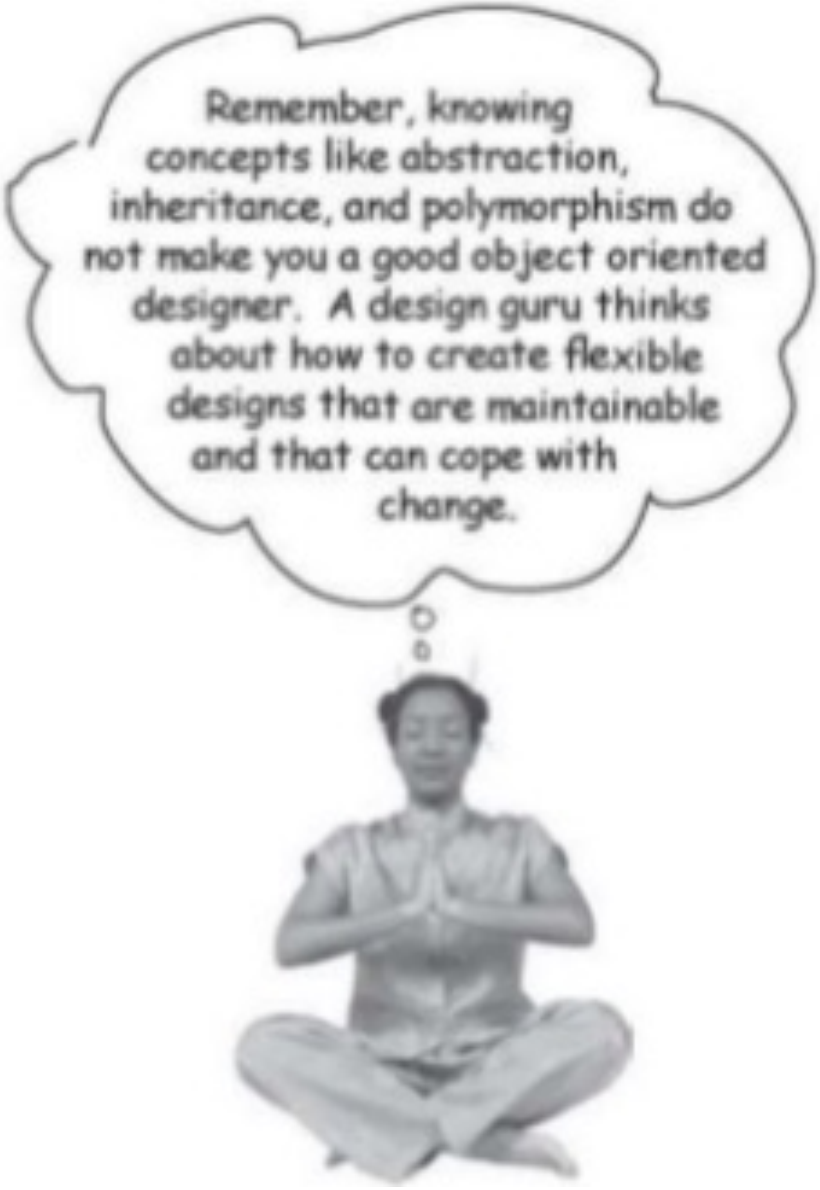
- Todos os diagramas anteriores são insuficientes, incapazes, ambíguos e/ou inconsequentes!

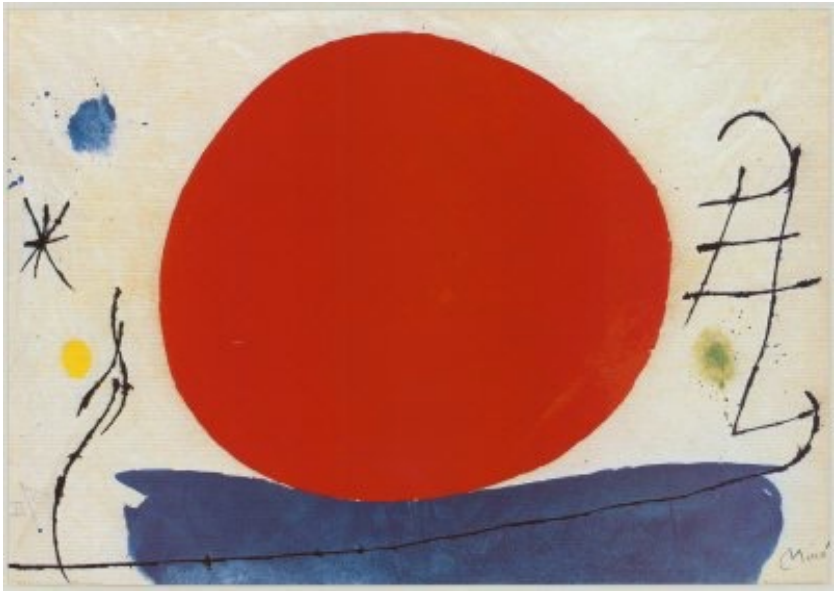
- Notação de desconhecida
- Assunção de ideias feitas (e.g. símbolo de base de dados, camadas)
- ...

Argumentos (desconcertantes)

- “eu entendo”/“nós entendemos”
- “foi feito à pressa…”
- “pode não fazer sentido, mas eu explico…”
- “é o que a (minha) empresa usa…”
- ”até a Microsoft usa destes diagramas!...”
- “certo, mas depois na implementação decide-se”
- “tem que se considerar as camadas/cores/forma/símbolos”
- “onde é que já vai ESOFT/ARQSIS/etc…”
- “mas funciona!...”

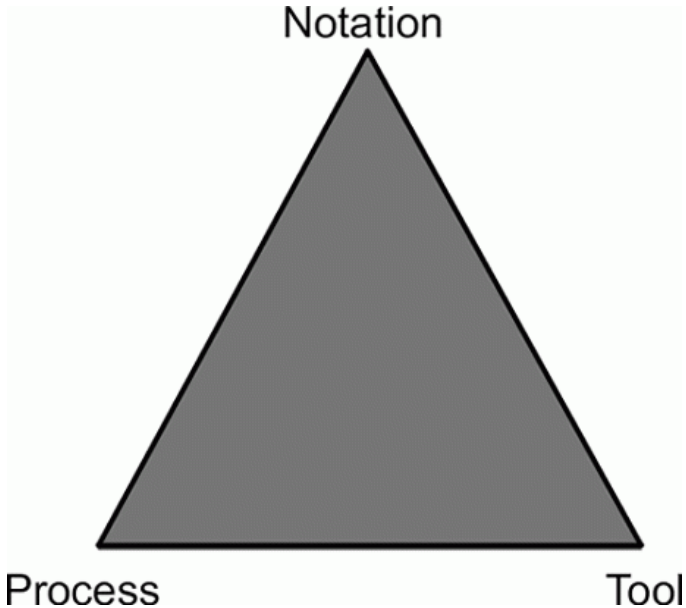
Resumo


Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.



I.e., a descrição arquitetural deve servir fundamentalmente para se raciocinar sobre o sistema, e não pela simples razão de descrever.
Depois de dominar a notação e o processo, pode-se relaxar ambos, mas talvez deixe de se querer tal.

Process

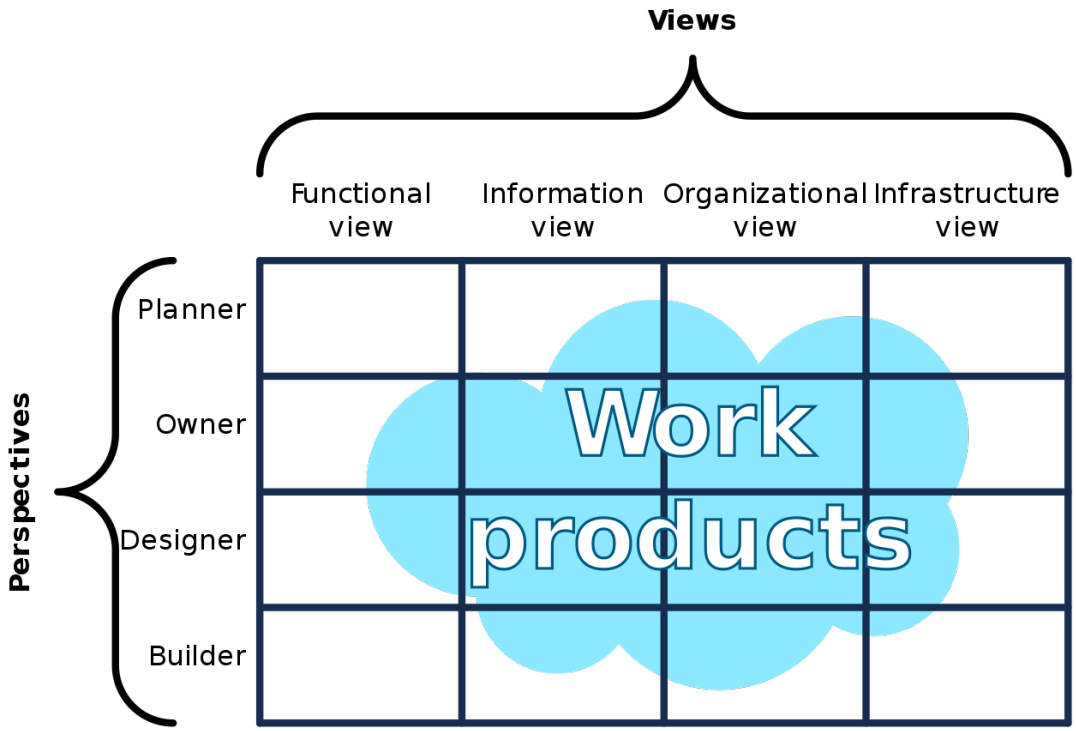


Notation Only – You can learn a **notation (e.g UML)**, but if you don't know how to use it (**process**), you will probably fail.
Process Only – You may have a great **process**, but if you can't communicate the process (**notation**), you will probably fail.
No Tool Support – if you cannot effectively document the artifacts of your work (**tool**), you will probably end up waste a lot of time and eventually fail.

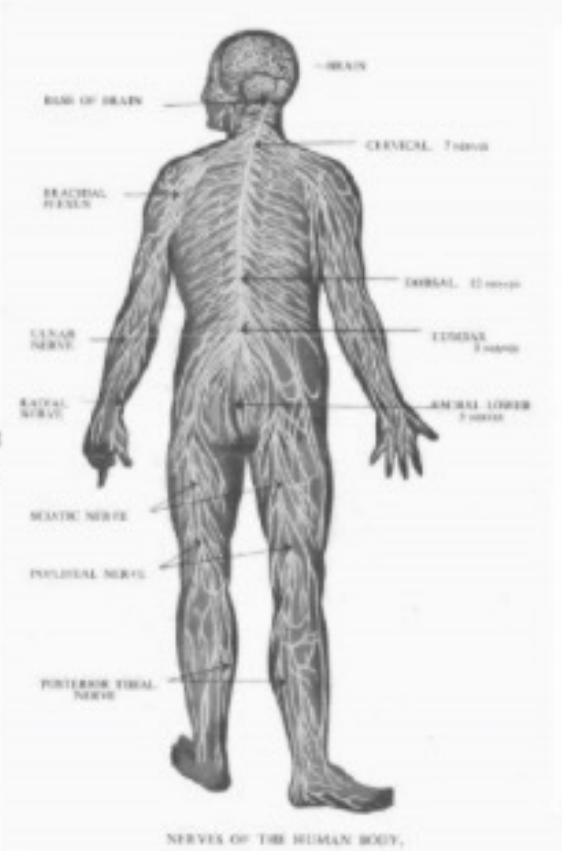
Views (from Perspetives)

Views vs. Perspetives

- "A *view* is a representation of a whole system from the perspective of a related set of concerns."

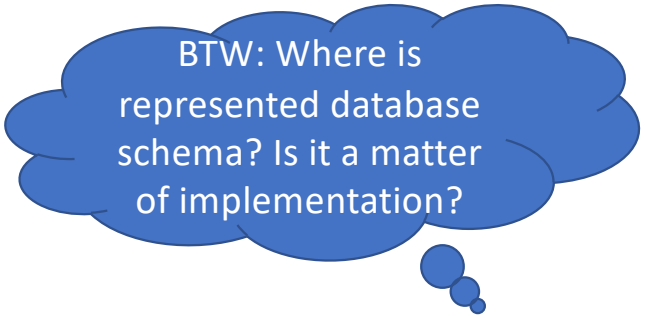


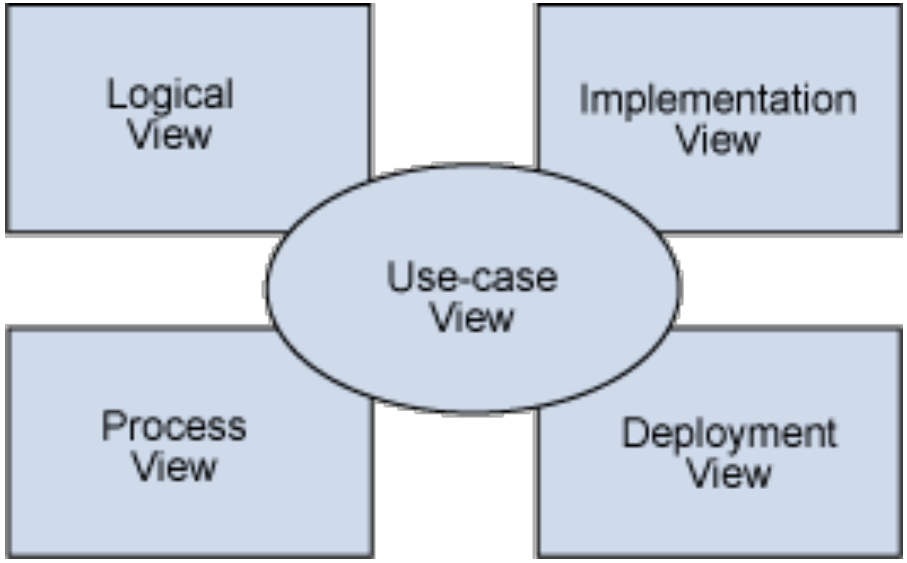
What is this?



dipartimentoinformatica

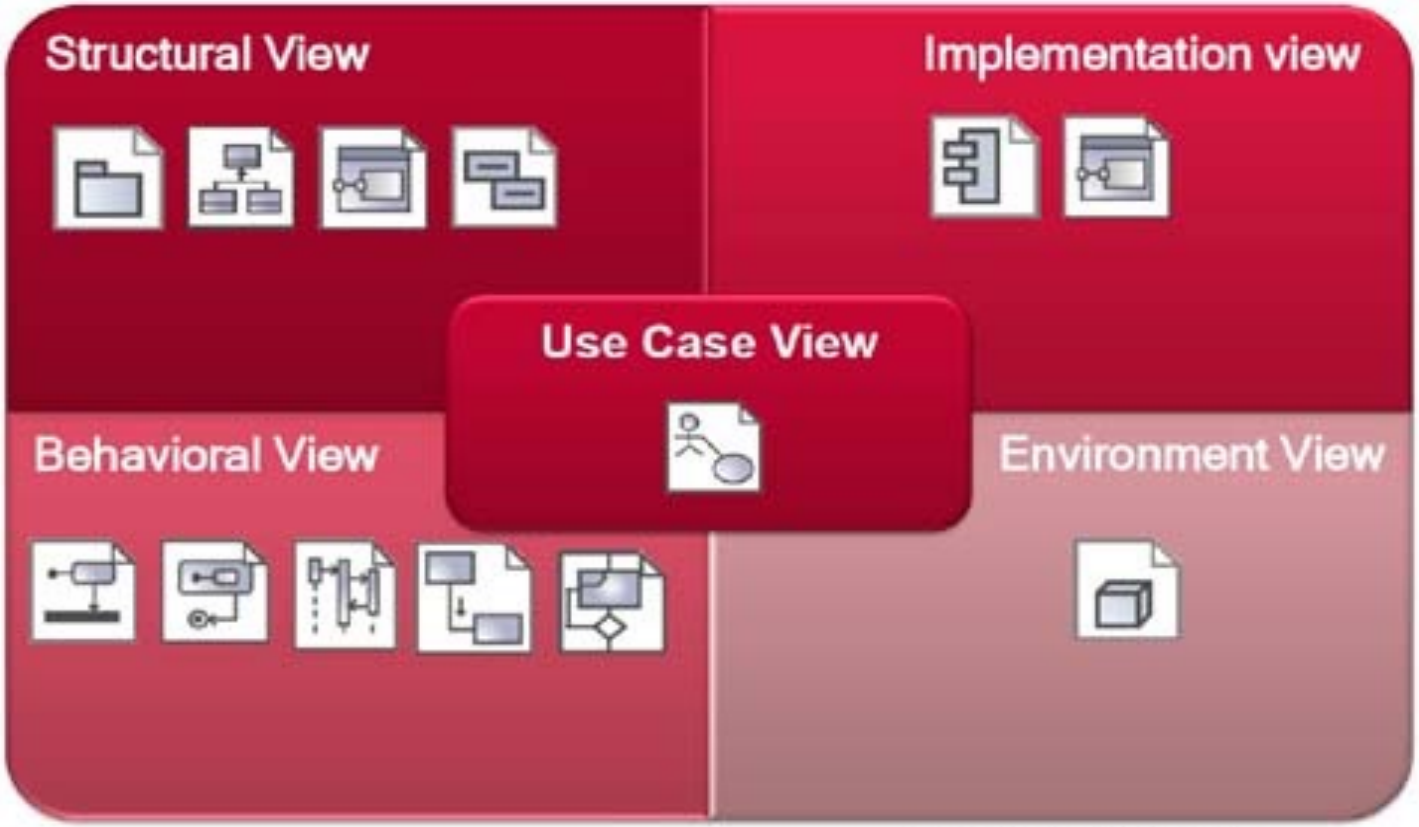
Modelo 4+1





1. **Logical View (or Structural View)** - *an object model of the design*
2. **Process View (or Behavioral View)** - *concurrency and synchronization aspects*
3. **Development View (or Implementation View)** - *static organization (subset) of the software*
4. **Physical View (or Deployment View)** - *mapping of the software to the hardware*

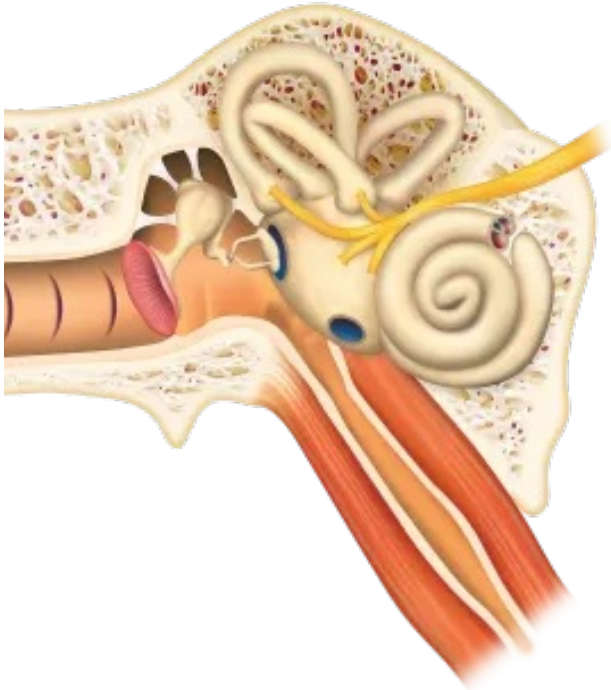
+1. Use-cases View (or Scenarios) - *various usage scenarios*



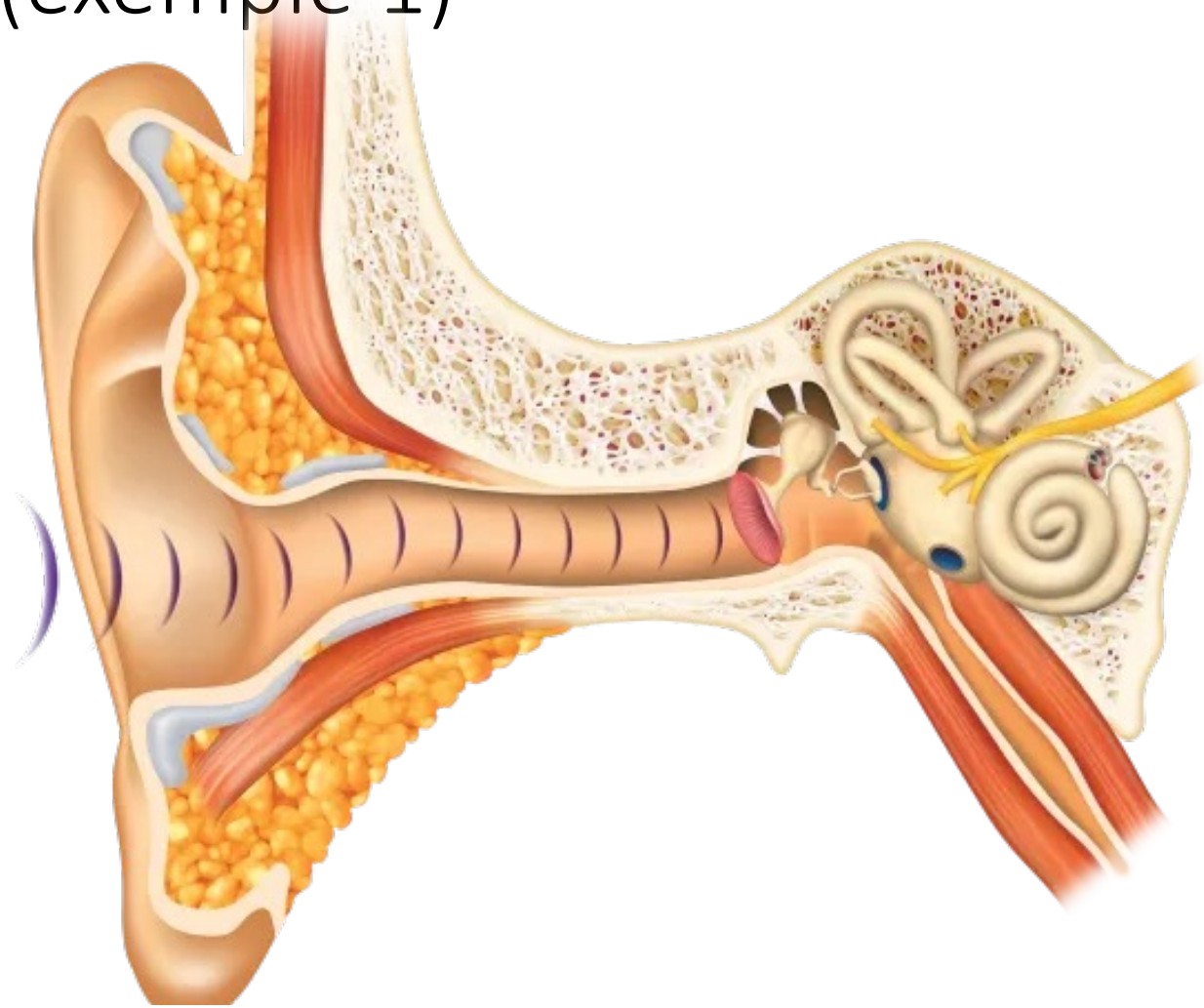
http://www.umlzone.com/articles/efficient-uml-from-anti-patterns-to-best-practices/

Abstraction

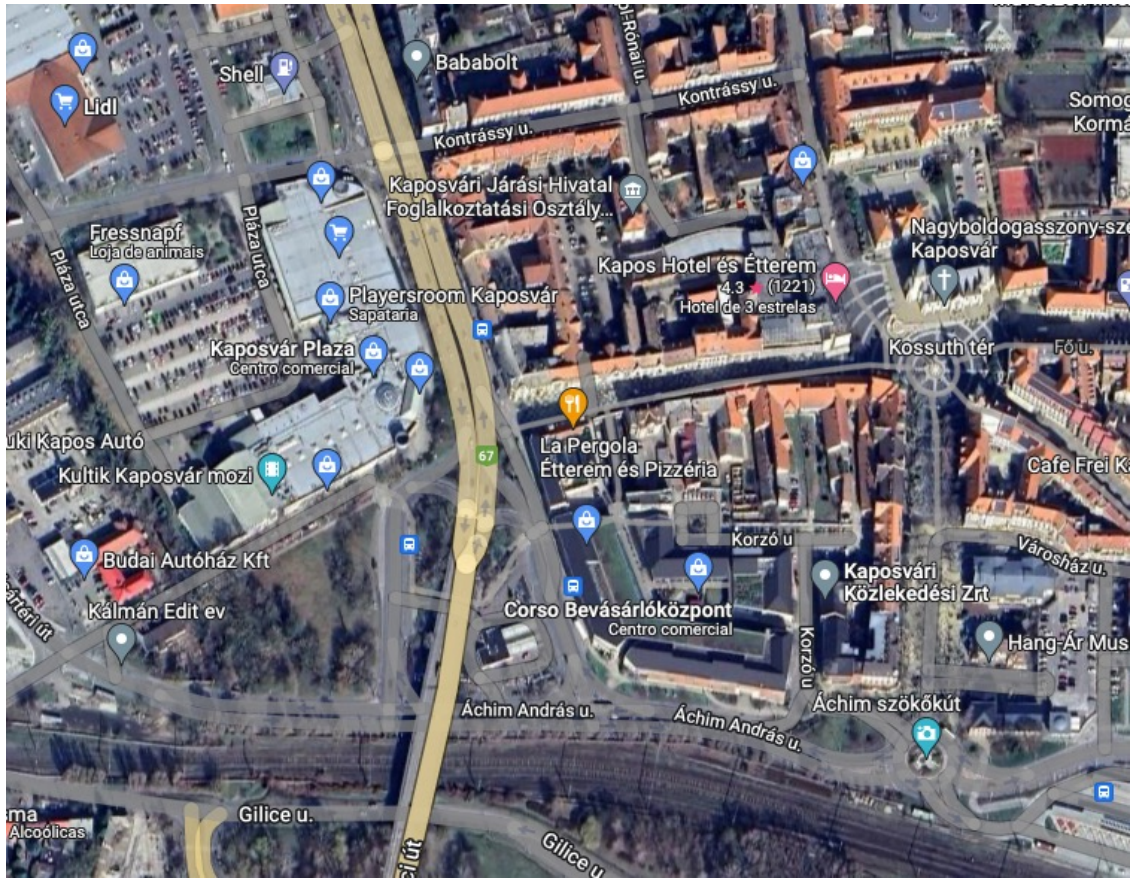
What is this? (exemple 1)



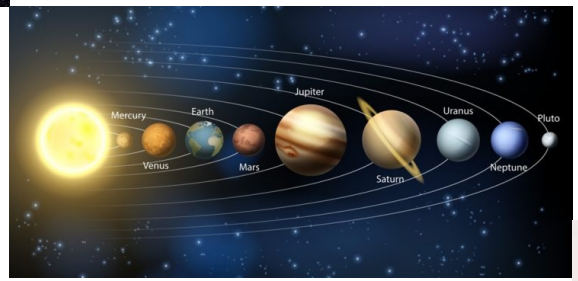
What is this? (exemple 1)



Where is this? (exemple 2)



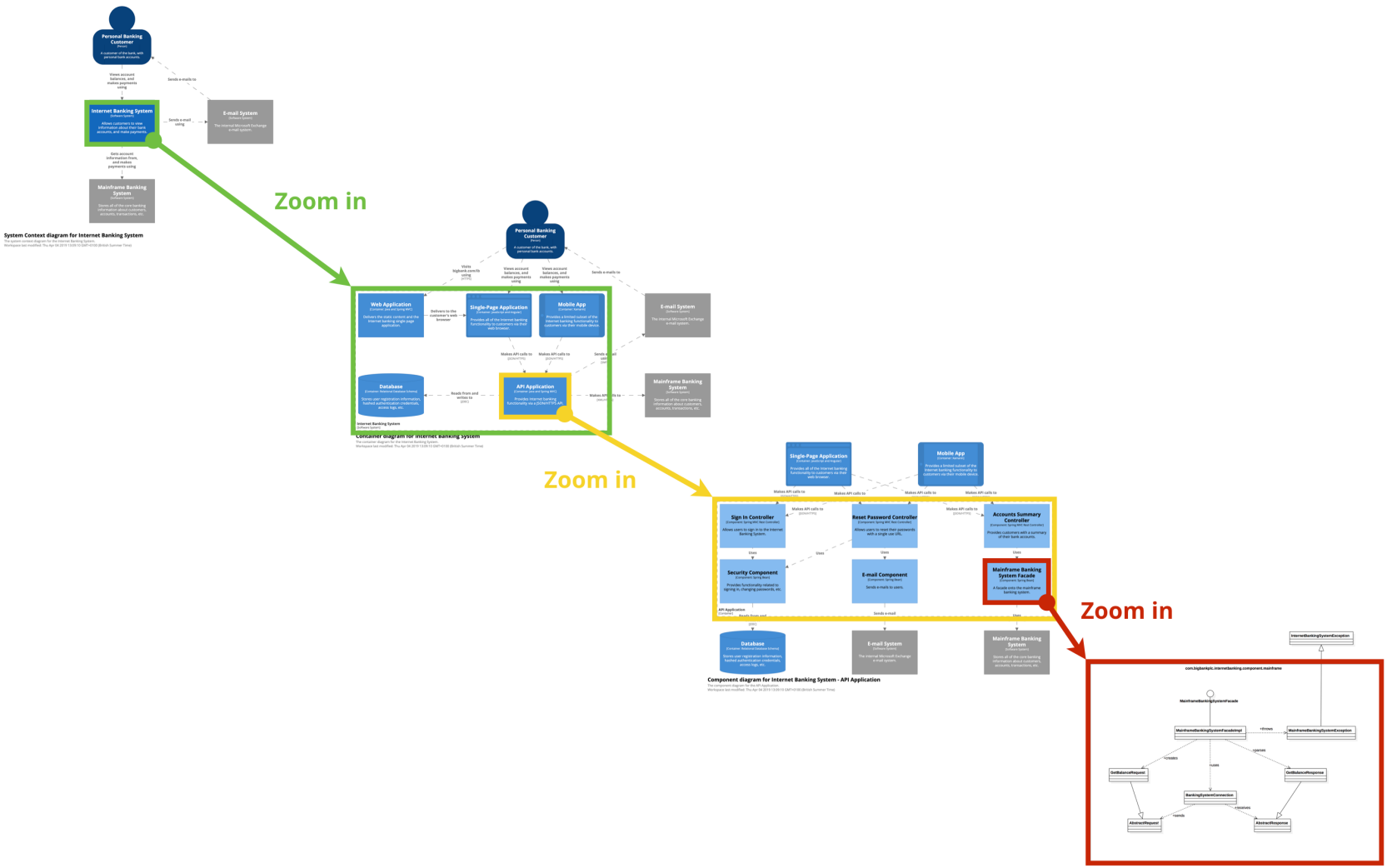
YOU ARE HERE


Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto

zoom

EUROPA
ÁFRICA
OCEANIA
ANTÁRTIDA

Mar do Norte
Mar Mediterrâneo



Level 1
Context

Level 2
Containers

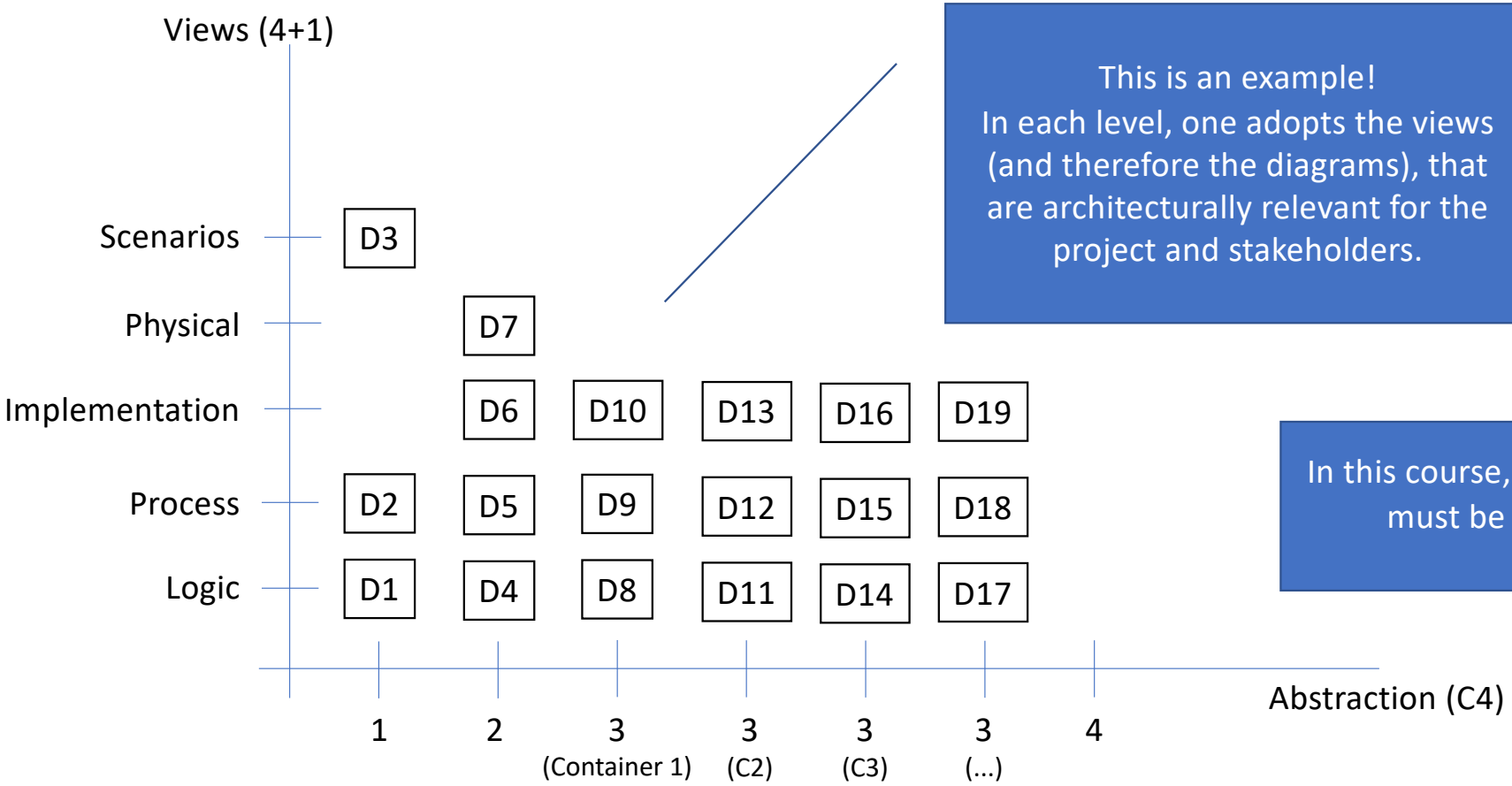
Level 3
Components

Level 4
Code

Combining 4+1 with C4

Combining 4+1 with C4

There are no mandatory diagrams, but relevant or irrelevant.



This is an example!
In each level, one adopts the views (and therefore the diagrams), that are architecturally relevant for the project and stakeholders.

In this course, the diagrams must be in UML!

Combining 4+1 with C4

There are no mandatory diagrams, but relevant or irrelevant.

View / Abstraction	Part(s)	Logic	Process	Implementation	Physical	Scenarios
Context (System)	-					
Containers	(C1, C2, .. Cn)					
Components	C1.1					
	C1.2					
	...					
	Cn.m					
Code	C1.1.1					
	C1.1.2					
	...					
	Cn.m.q					

In each cell, what are the relevant diagrams?

Example

Document the aimed system using 4+1 and C4

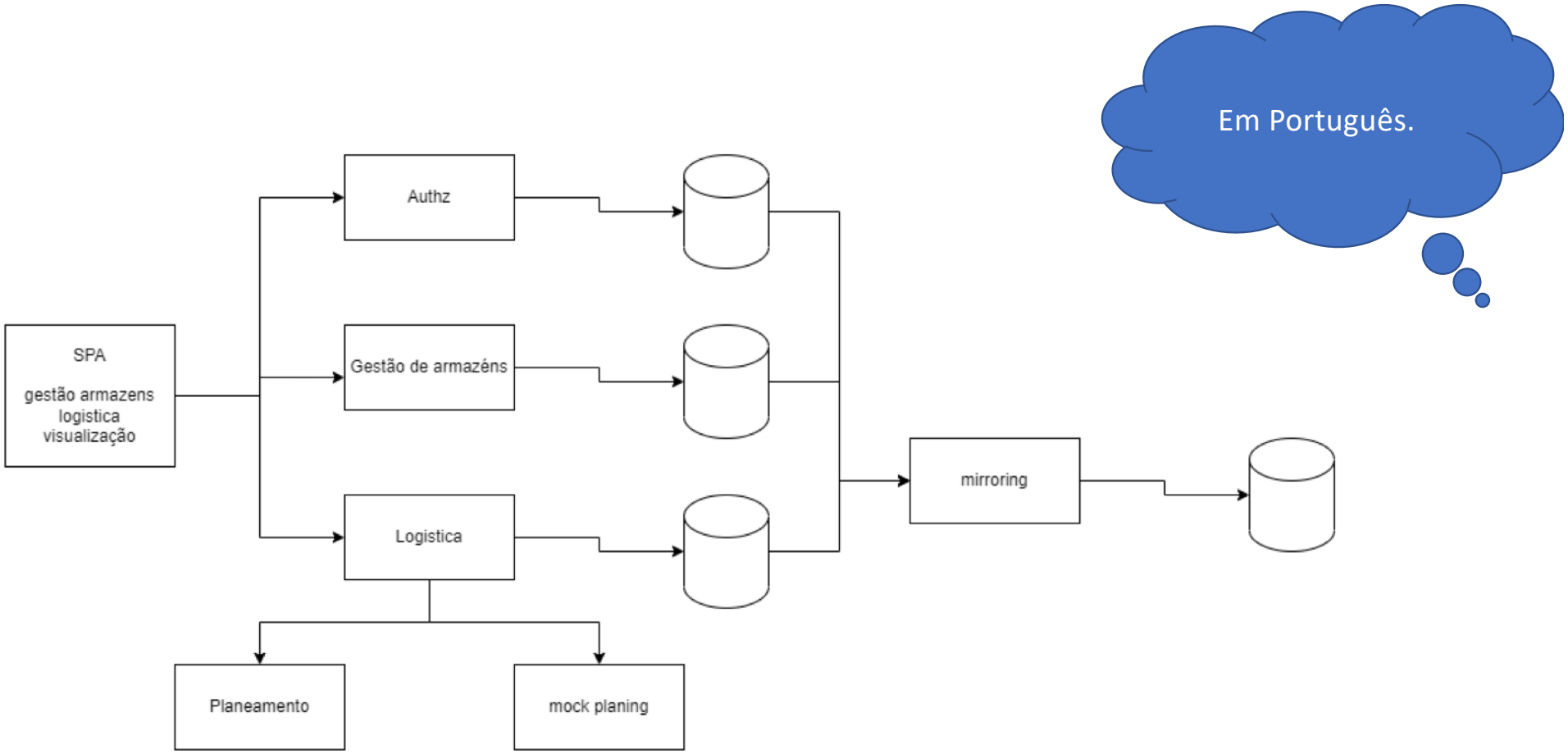


Figura 1. Visão geral do sistema

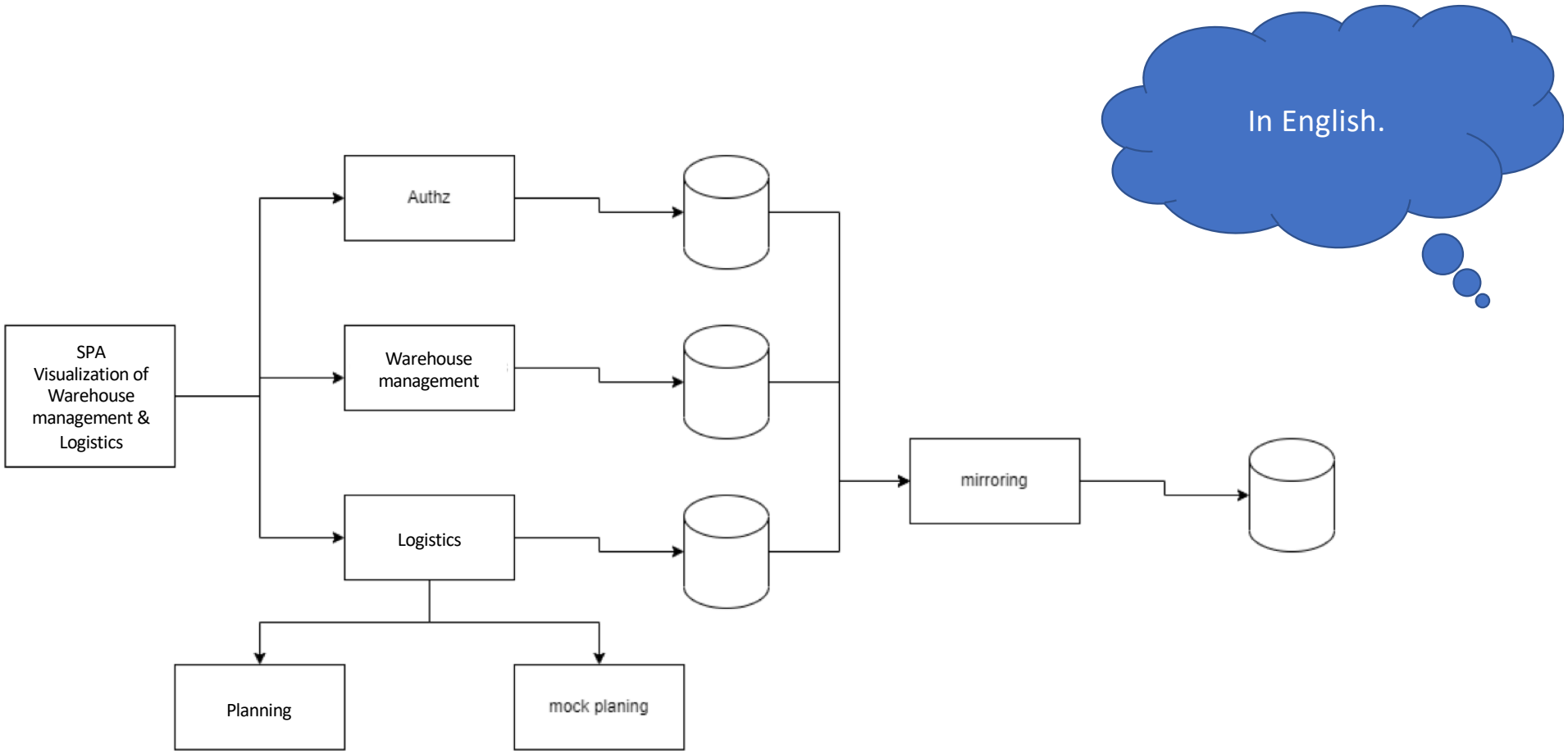


Figure 1: Outlook of the system

How to, with C4 and 4+1

- TBD in white board during lecture class

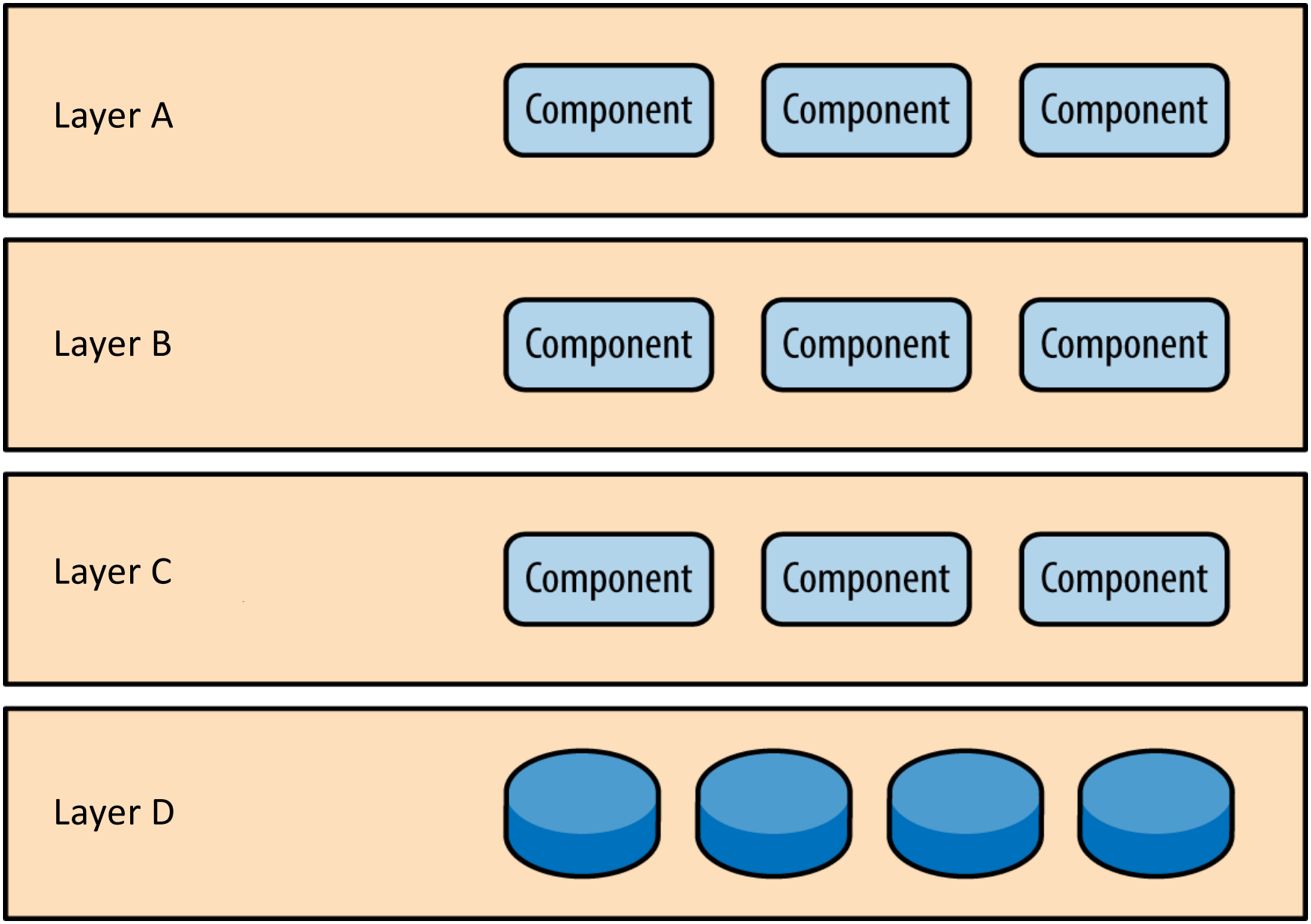
- Example
  - https://bitbucket.org/nunopsilva/bulletproof-nodejs-ddd/wiki/Home

Arquitetura por Camadas e Andares

Layered and Tier Architecture

Architectural factors

Architecture style	Description
Client/Server	Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.
Domain Driven Design	An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message Bus	An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.
N-Tier / 3-Tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.



https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

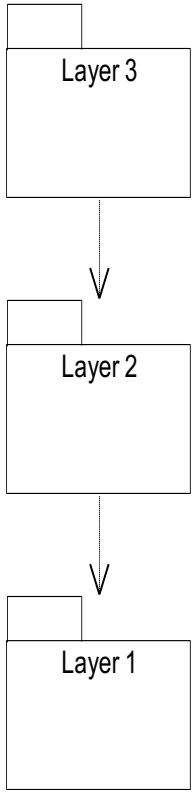
Arquitetura por Camadas

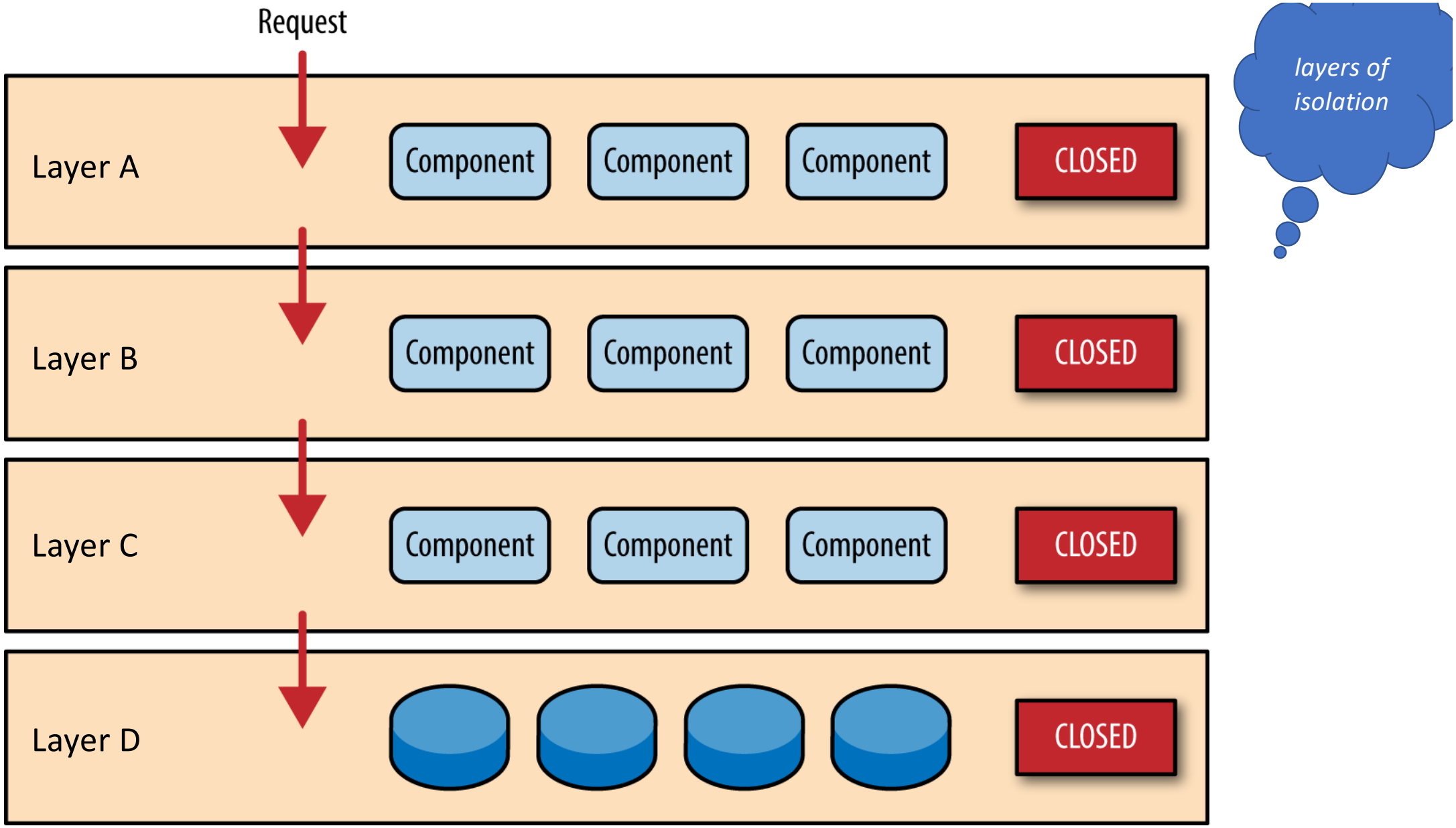
Sem violar estes princípios de acoplamento, como é que se resolve esta "limitação"?

- A camada de nível superior usa vários serviços definidos na camada de nível inferior, mas a camada mais a baixo desconhece (não depende) da camada superior

- Cada camada, normalmente, esconde os seus níveis inferiores das suas camadas de nível superior – camada 3 usa serviços da camada 2, que por sua vez usa serviços da camada 1, mas a camada 3 não conhece a camada 1

- nem sempre é tão rigoroso

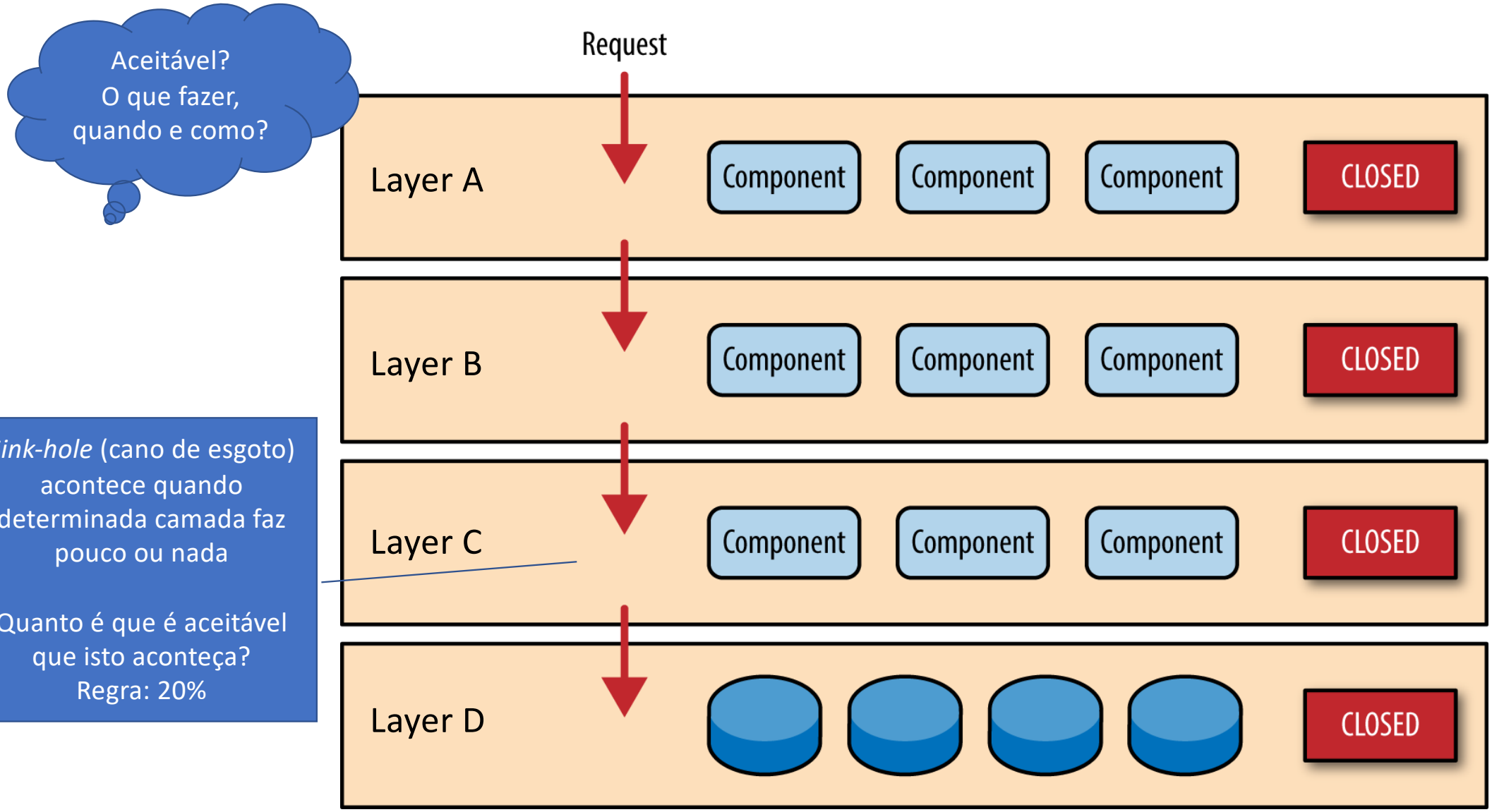


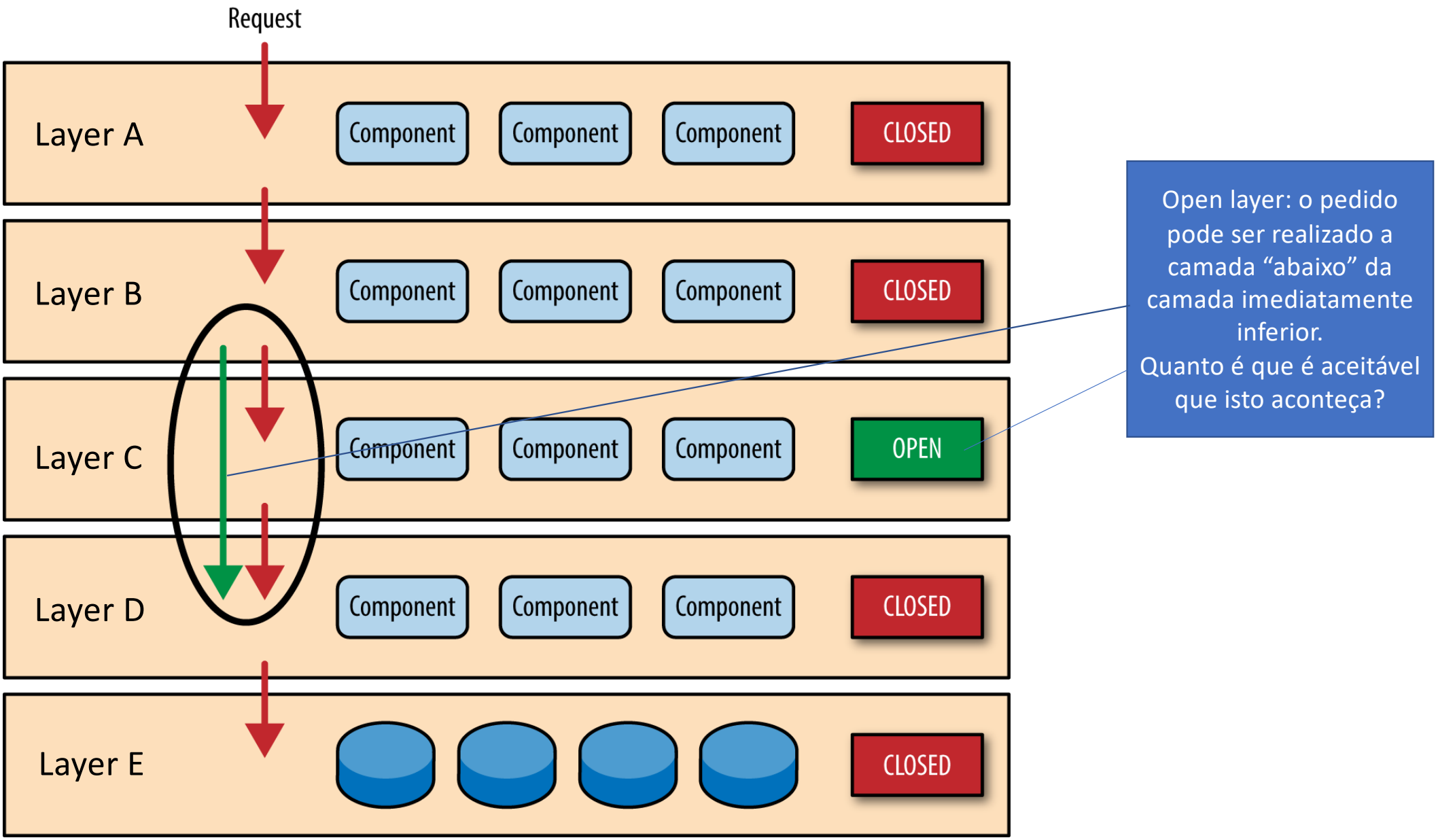

Request
Layer A
Component
Component
Component
CLOSED
Layer B
Component
Component
Component
CLOSED
Layer C
Component
Component
Component
CLOSED
Layer D
CLOSED
layers of isolation

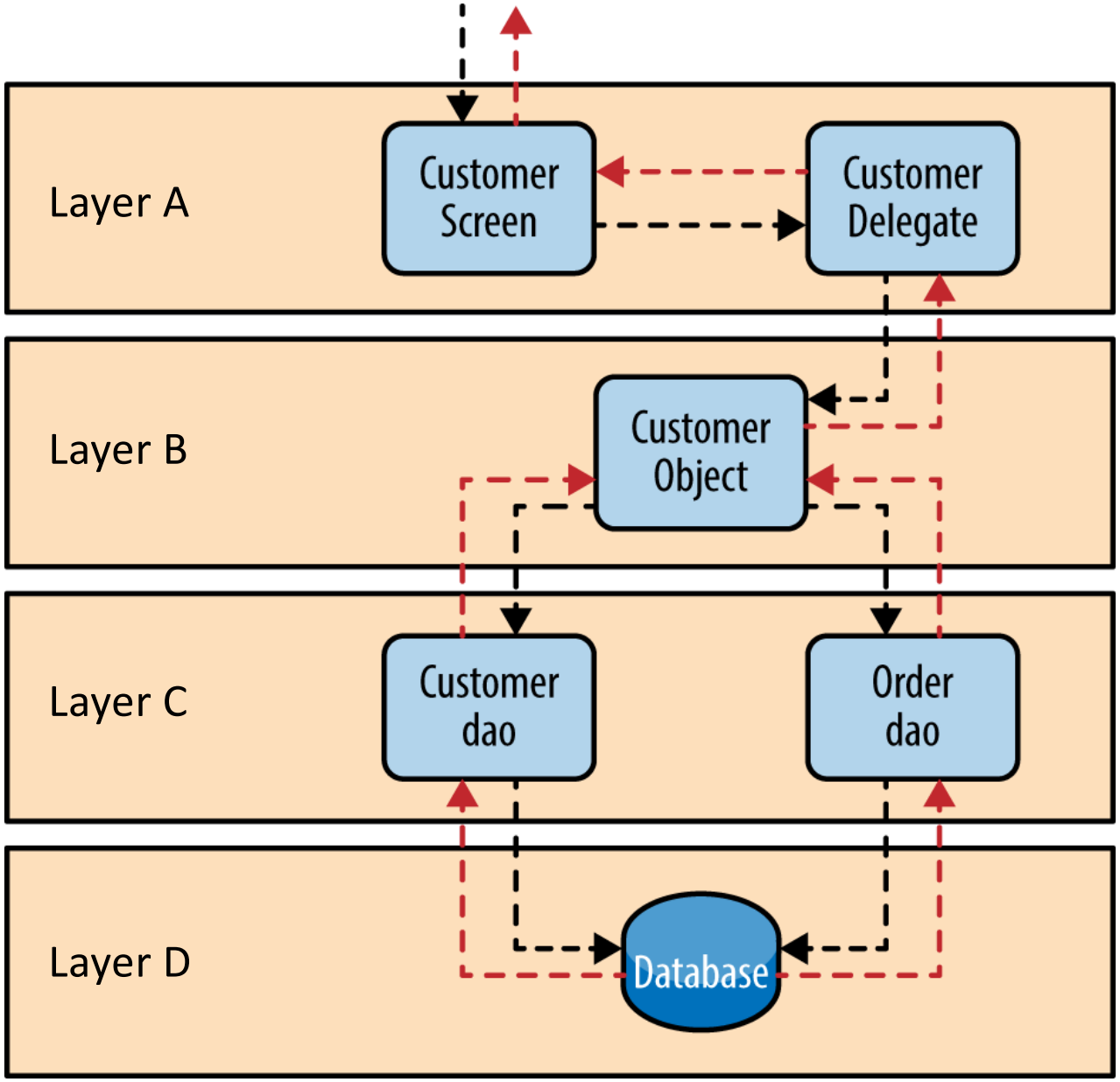
Layers of Isolation

- *The layers of isolation concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers: the change is isolated to the components within that layer, and possibly another associated layer (such as a persistence layer containing SQL).*
- *If you allow the presentation layer direct access to the persistence layer, then changes made to SQL within the persistence layer would impact both the business layer and the presentation layer, thereby producing a very tightly coupled application with lots of interdependencies between components.*
- *This type of architecture then becomes very hard and expensive to change.*

in https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html


Aceitável?
O que fazer,
quando e como?
Request
Layer A
Component
Component
Component
CLOSED
Layer B
Component
Component
Component
CLOSED
Layer C
Component
Component
Component
CLOSED
Layer D
CLOSED
Sink-hole (cano de esgoto) acontece quando determinada camada faz pouco ou nada
Quanto é que é aceitável que isto aconteça?
Regra: 20%


Request
Layer A
Component
Component
Component
CLOSED
Layer B
Component
Component
Component
CLOSED
Layer C
Component
Component
Component
OPEN
Layer D
Component
Component
Component
CLOSED
Layer E
CLOSED
Open layer: o pedido pode ser realizado a camada “abaixo” da camada imediatamente inferior.
Quanto é que é aceitável que isto aconteça?


Layer A
Customer Screen
Customer Delegate
Layer B
Customer Object
Layer C
Customer dao
Order dao
Layer D
Database

Arquitetura por Camadas - vantagens

- Localização das alterações numa parte da aplicação minimiza o impacto nas outras partes

- NB: o estilo de arquitetura por camadas não preconiza camadas específicas

- Reduz o trabalho de depuração e resolução de erros
- Facilita a manutenção da aplicação

- Equipas de desenvolvimento diferentes podem trabalhar em partes da solução sem estar fortemente dependentes de outras equipas

Arquitetura por Camadas - vantagens

- Os componentes da solução podem ser distribuídos separadamente e a sua manutenção e evolução podem ser realizados em tempos diferentes

- Facilita o desenvolvimento de aplicações multi-canal (e.g. Web, Windows, Mobile), reutilizando os componentes
  - E.g. a camada de negócios

- Facilita a realização de testes da aplicação. Como cada camada comunica com outra através de interfaces claramente definidas, é mais simples substituir a implementação de uma camada e testar camadas independentemente

Arquitetura por Camadas - problemas

- Atravessar várias camadas/componentes tem um efeito negativo no desempenho do sistema

- A organização em camadas pode provocar alterações em cascata
  - E.g. acrescentar um campo à base de dados que precisa de ser apresentado na UI provoca alterações em todos as camadas intermédias

- O desenvolvimento e o código ficam mais complexos

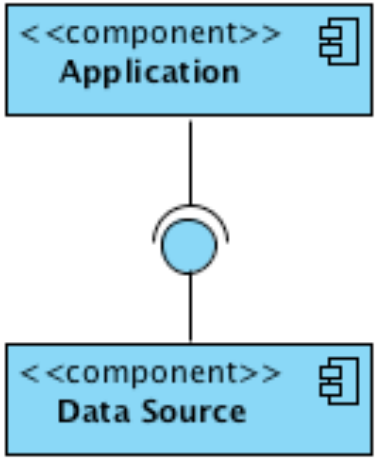
Deve ser utilizada quando a complexidade da aplicação o justifique

Quantas camadas?
Que camadas?

O estilo de arquitetura por camadas não preconiza camadas específicas

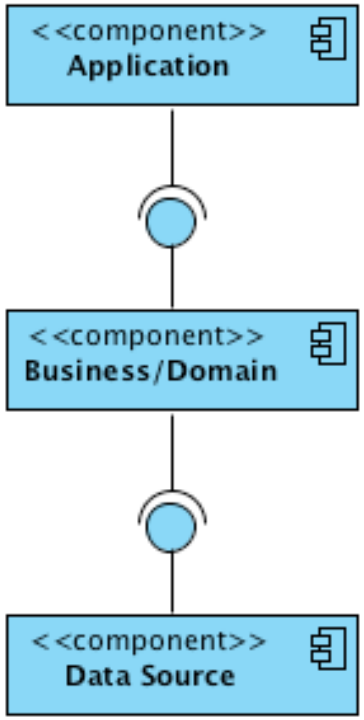
2 camadas

- Um só modelo lógico de dados
- Protótipos com ciclo de vida curto
- Alterações à API não são esperadas
- Pequenas aplicações com custo e tempo baixos



3 camadas

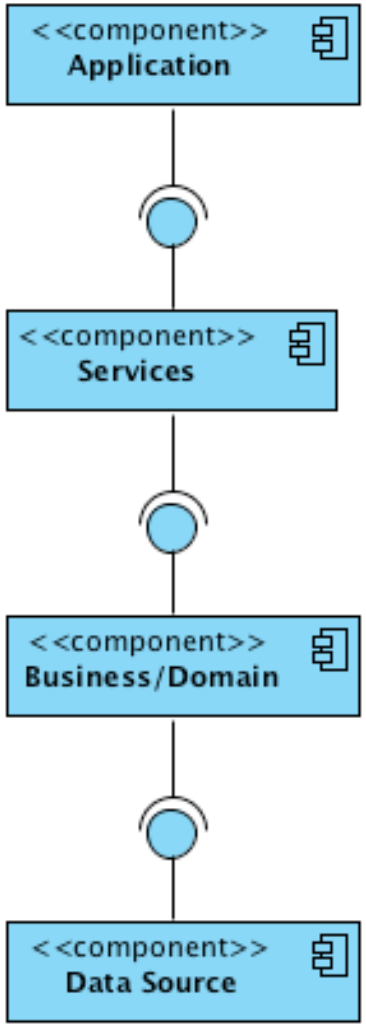
- Permite maior isolamento da aplicação das camadas inferiores
  - O desenvolvedor da aplicação necessita de saber menos dos dados
- Múltiplas/variadas vistas dos dados
- Quando é esperado que um dos componentes se altere
  - E.g. as fontes de dados
- Requisitos: concorrência no acesso aos dados



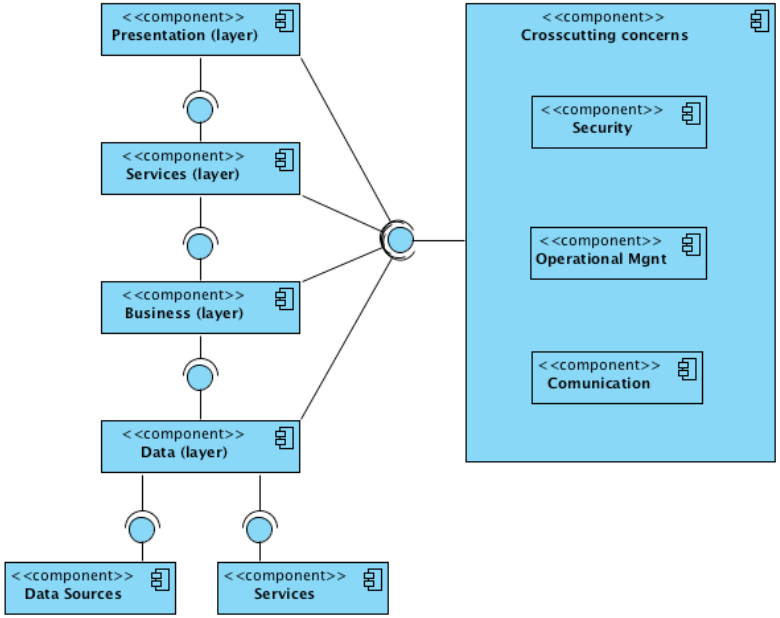
4 camadas

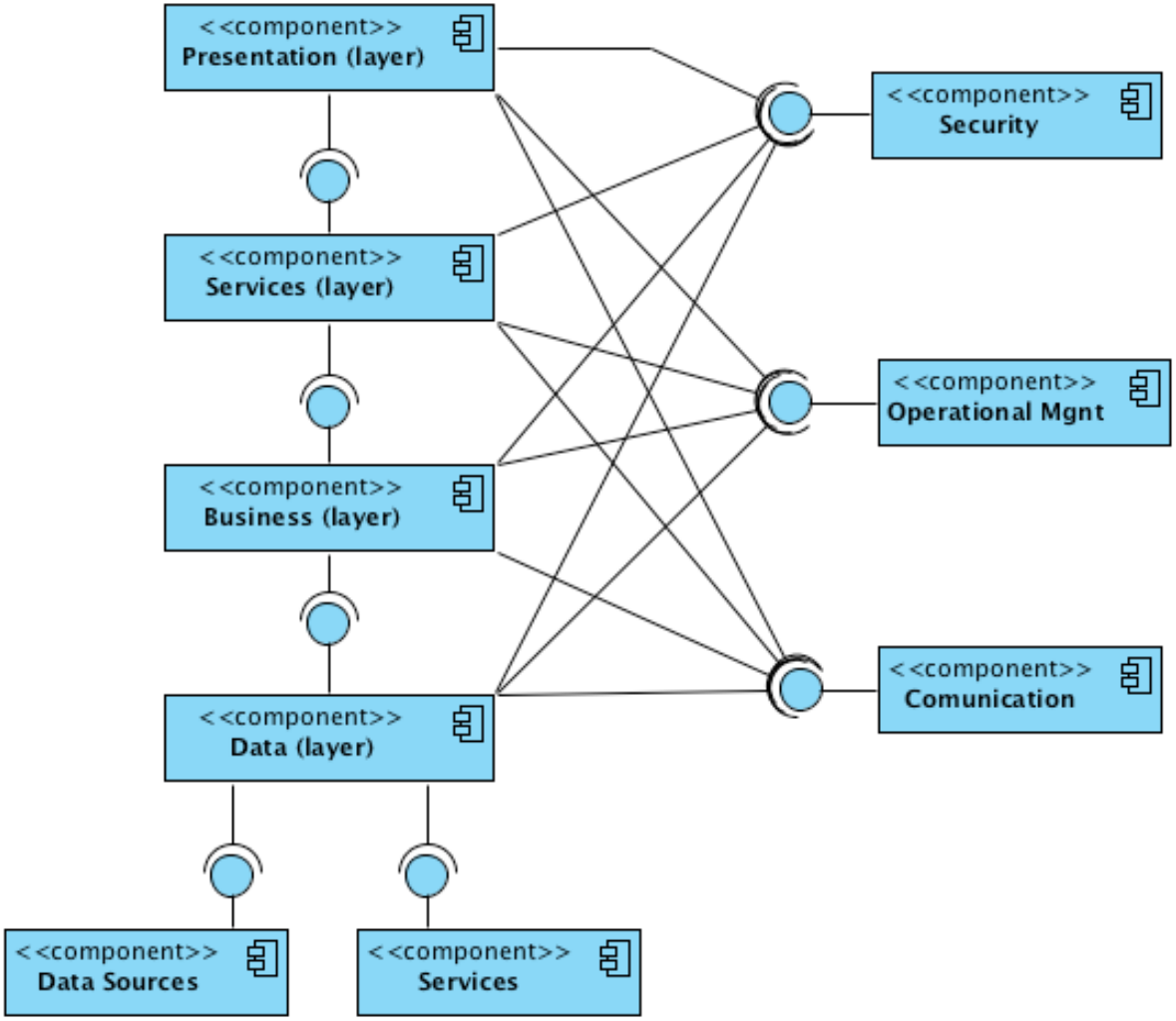
- Necessárias várias operações/interfaces de negócio
  - Protocolos de comunicação
  - Formatos dos dados
  - Requisitos de segurança

- "Services" expõe operações (métodos) de negócio como um serviço
  - É um ponto de entrada para os serviços da aplicação
  - O contracto do serviço deve especificar o protocolo da comunicação, o formato dos dados, requisitos de segurança e modo de localização do serviço
  - Para uma funcionalidade podem ser implementadas múltiplas interfaces do serviço, para clientes diferentes



Exemplos


<<component>>
Presentation (layer)
<<component>>
Services (layer)
<<component>>
Business (layer)
<<component>>
Data (layer)
<<component>>
Data Sources
<<component>>
Services
<<component>>
Crosscutting concerns
<<component>>
Security
<<component>>
Operational Mgnt
<<component>>
Comunication


<<component>>
Presentation (layer)
<<component>>
Services (layer)
<<component>>
Business (layer)
<<component>>
Data (layer)
<<component>>
Data Sources
<<component>>
Services
<<component>>
Security
<<component>>
Operational Mgnt
<<component>>
Comunication

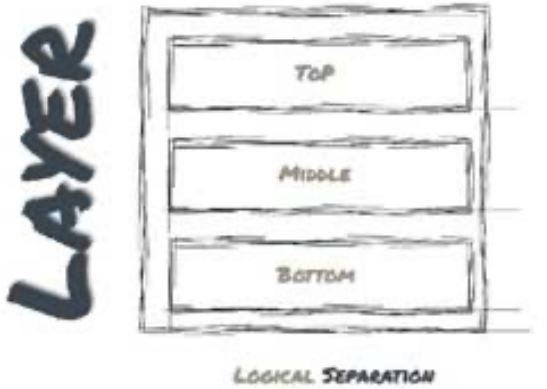
Camadas vs. Tiers

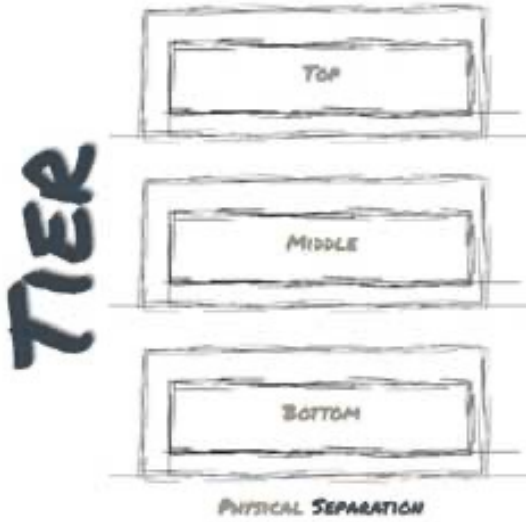
Camadas vs. Tiers

- ***Layer*** é uma separação lógica de componentes de software, ao nível do desenvolvimento, de modo a tornar mais simples a partição de responsabilidades e tarefas de um sistema. Refere-se à organização do código e dos dados.

- ***Tier (nível ou andar em português)***, é uma camada de hardware físico (máquina) ou virtual (máquina ou ambiente virtual), em que o software é executado. Refere-se à implantação (deploy) da aplicação.

- <u>n-layer não implica n-tier, nem n-tier implica n-layer (n+1)</u>

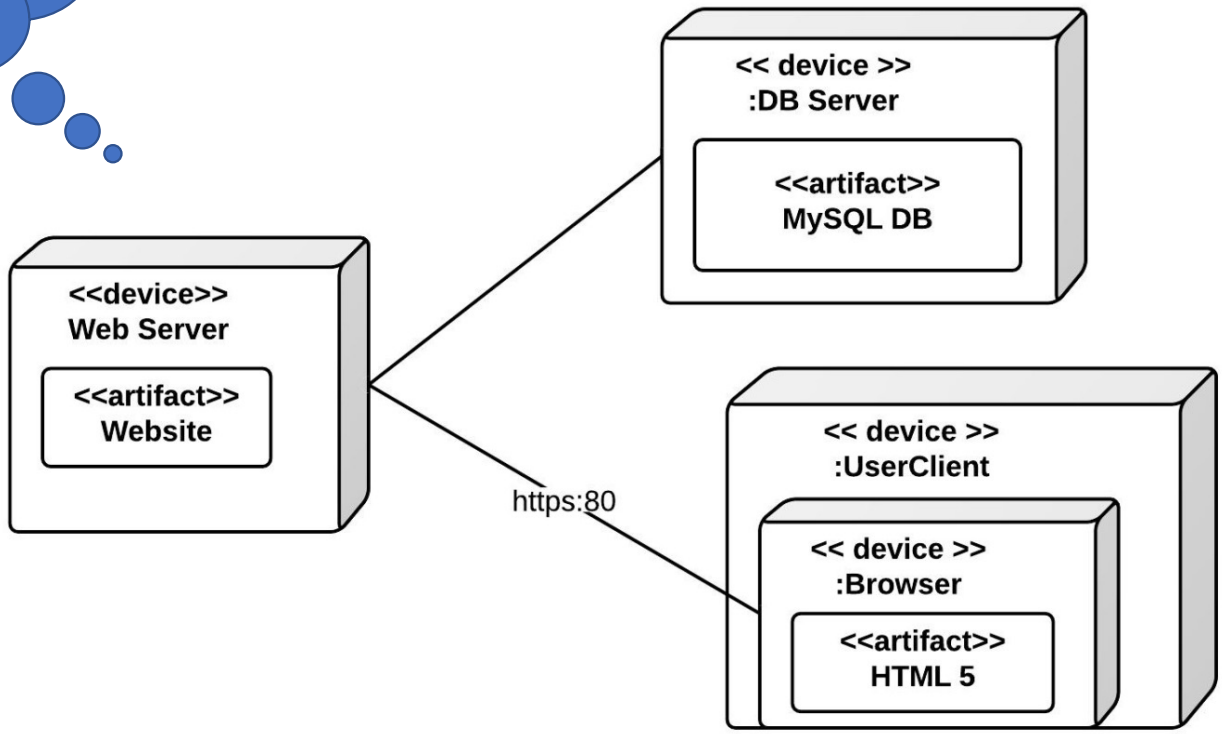


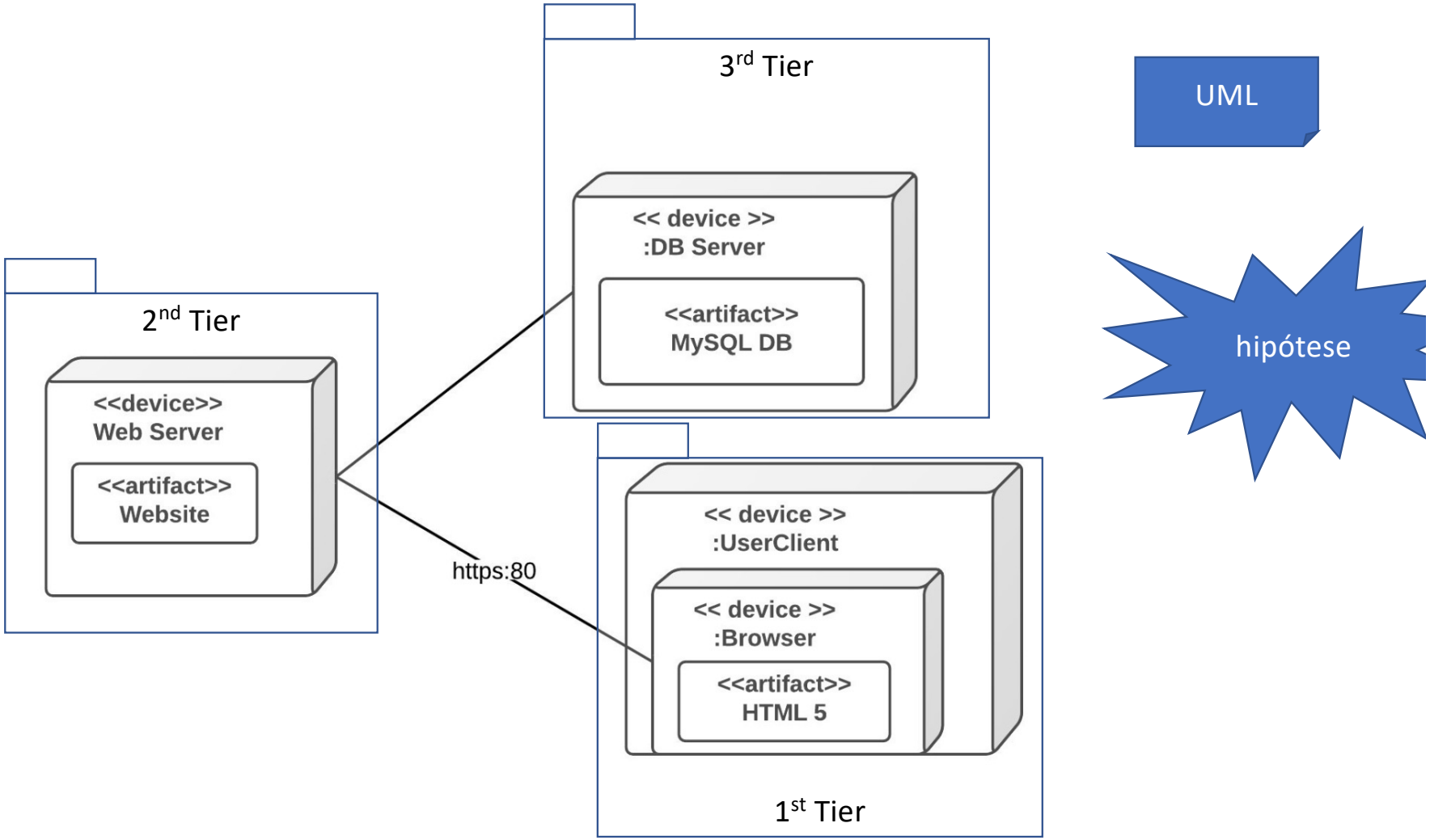


Exemplos

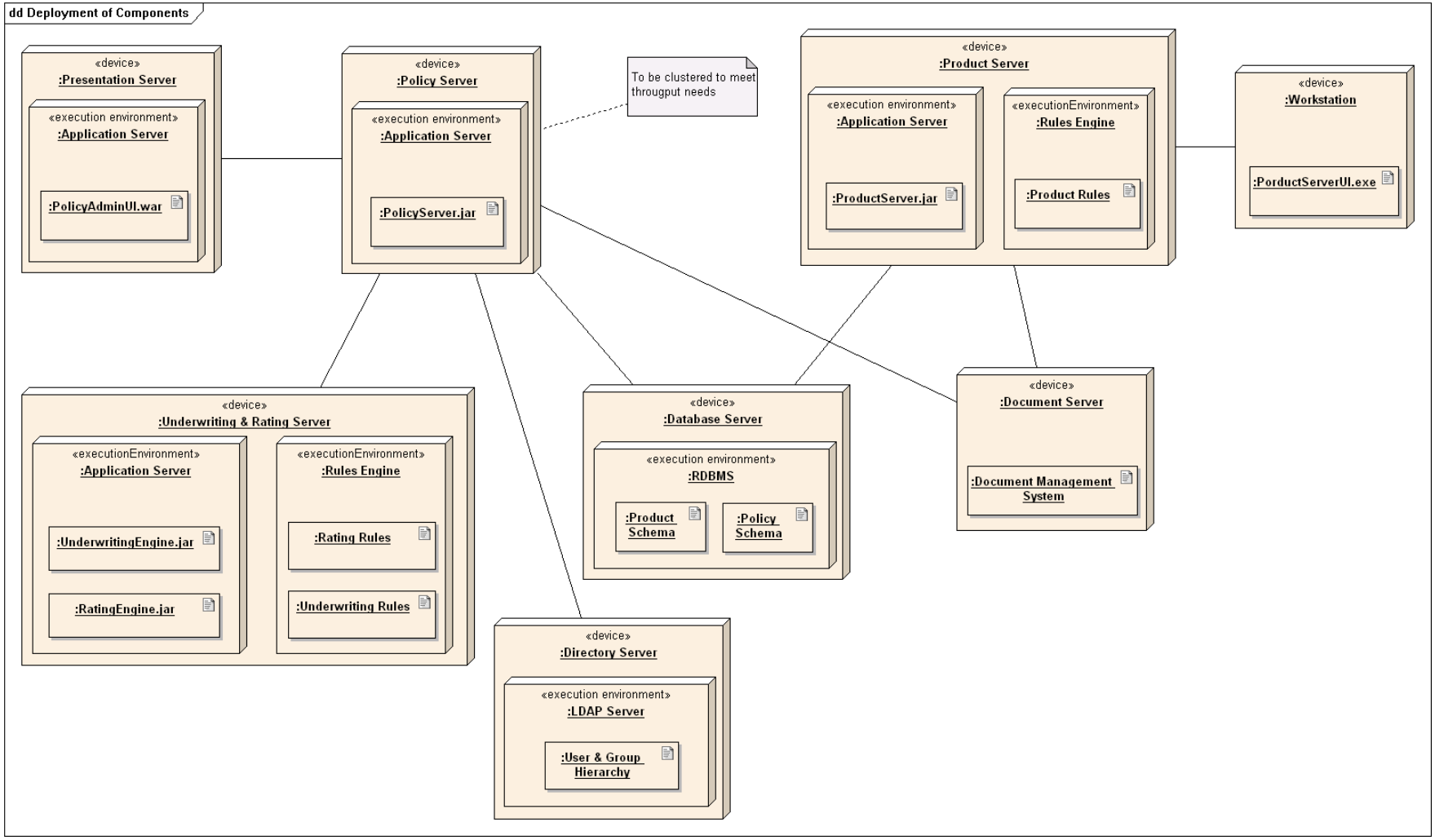
Quais os níveis/tiers?

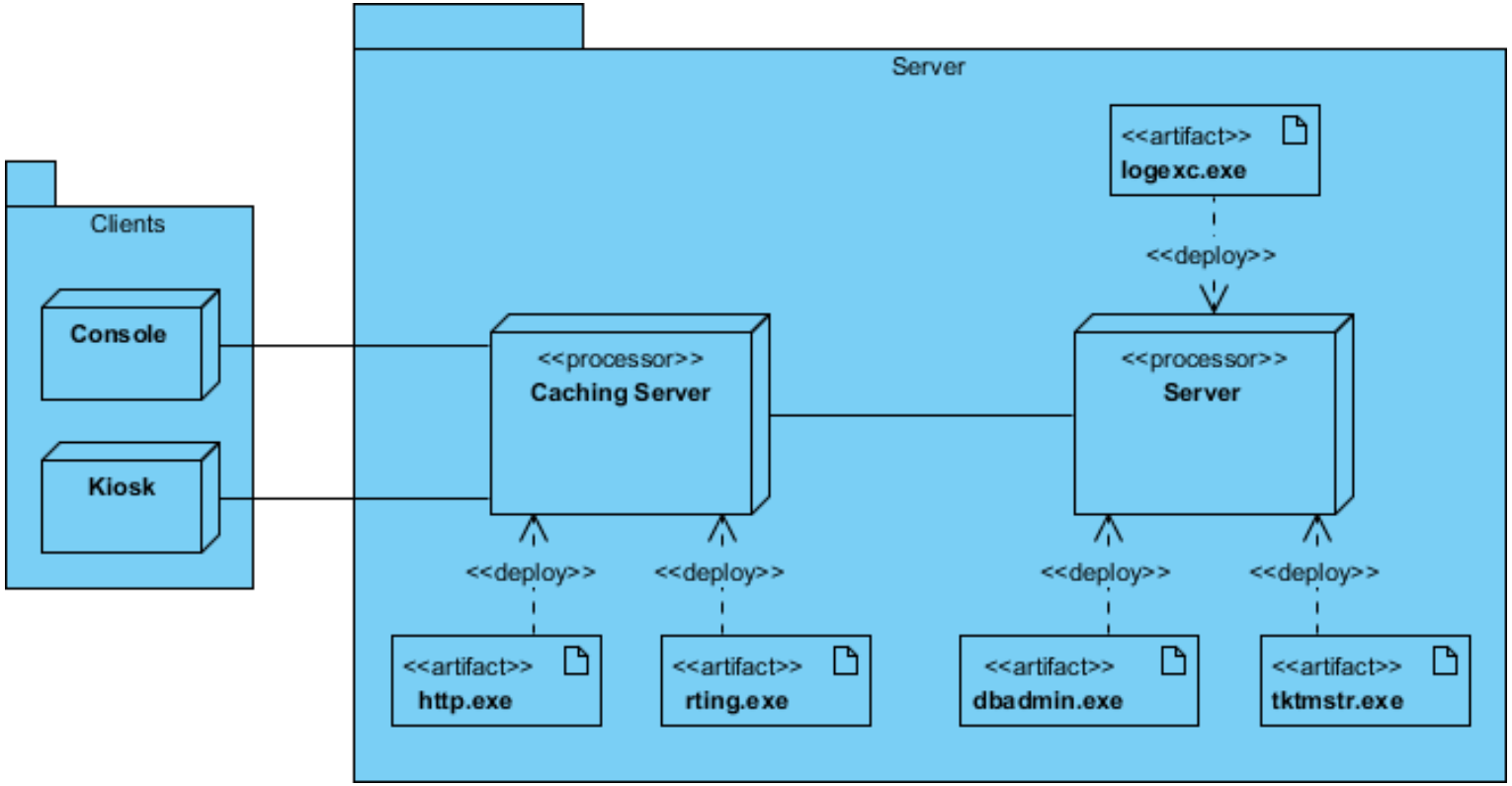
UML

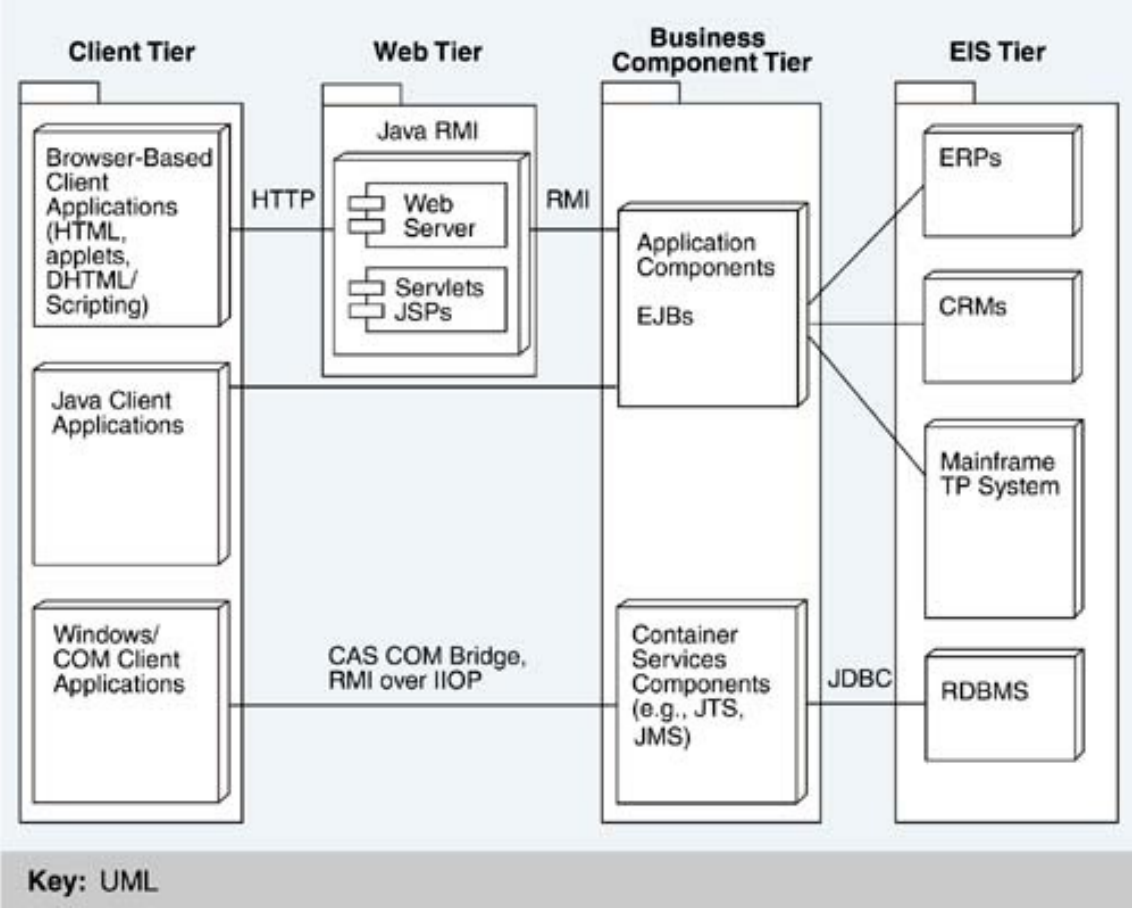


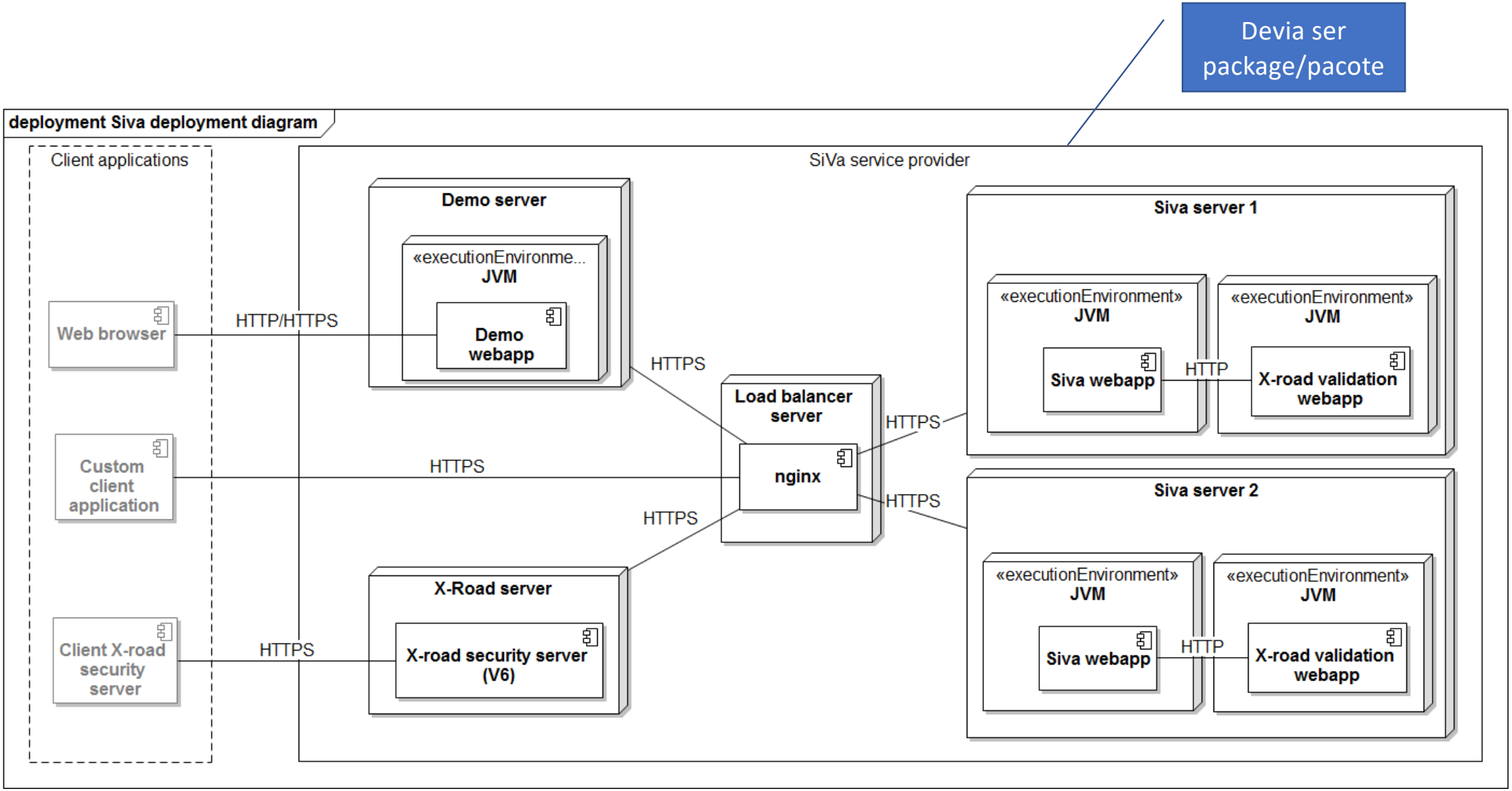

3rd Tier
<< device >>
:DB Server
<<artifact>>
MySQL DB
2nd Tier
<<device>>
Web Server
<<artifact>>
Website
https:80
<< device >>
:UserClient
<< device >>
:Browser
<<artifact>>
HTML 5
1st Tier
UML
hipótese

UML

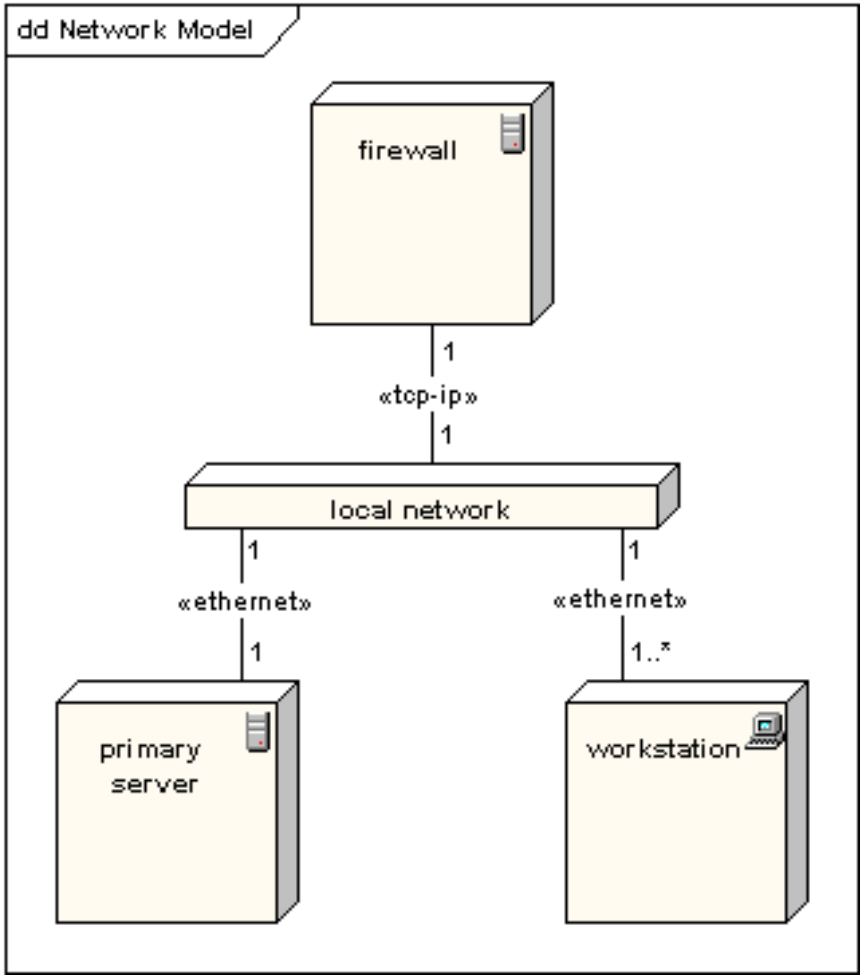

dd Deployment of Components
«device»
:Presentation Server
«execution environment»
:Application Server
:PolicyAdminUI.war
«device»
:Policy Server
«execution environment»
:Application Server
:PolicyServer.jar
To be clustered to meet
througput needs
«device»
:Product Server
«execution environment»
:Application Server
:ProductServer.jar
«executionEnvironment»
:Rules Engine
:Product Rules
«device»
:Workstation
:PorductServerUI.exe
«device»
:Underwriting & Rating Server
«executionEnvironment»
:Application Server
:UnderwritingEngine.jar
:RatingEngine.jar
«executionEnvironment»
:Rules Engine
:Rating Rules
:Underwriting Rules
«device»
:Database Server
«execution environment»
:RDBMS
:Product
Schema
:Policy
Schema
«device»
:Document Server
:Document Management
System
«device»
:Directory Server
«execution environment»
:LDAP Server
:User & Group
Hierarchy


Server
Clients
Console
Kiosk
<<artifact>>
logexc.exe
<<deploy>>
<<processor>>
Caching Server
<<processor>>
Server
<<deploy>>
<<deploy>>
<<deploy>>
<<deploy>>
<<artifact>>
http.exe
<<artifact>>
rting.exe
<<artifact>>
dbadmin.exe
<<artifact>>
tktmstr.exe


Client Tier
Web Tier
Business Component Tier
EIS Tier
Browser-Based Client Applications (HTML, applets, DHTML/ Scripting)
HTTP
Java RMI
Web Server
Servlets JSPs
RMI
Application Components
EJBs
ERPs
CRMs
Mainframe TP System
Java Client Applications
Windows/ COM Client Applications
CAS COM Bridge, RMI over IIOP
Container Services Components (e.g., JTS, JMS)
JDBC
RDBMS
Key: UML


Devia ser package/pacote
deployment Siva deployment diagram
Client applications
Web browser
Custom client application
Client X-road security server
HTTP/HTTPS
HTTPS
HTTPS
SiVa service provider
Demo server
«executionEnvironme...
JVM
Demo webapp
HTTPS
Load balancer server
nginx
HTTPS
HTTPS
X-Road server
X-road security server (V6)
Siva server 1
«executionEnvironment»
JVM
Siva webapp
HTTP
«executionEnvironment»
JVM
X-road validation webapp
Siva server 2
«executionEnvironment»
JVM
Siva webapp
HTTP
«executionEnvironment»
JVM
X-road validation webapp

UML


dd Network Model
firewall
1
«tcp-ip»
1
local network
1
«ethernet»
1
1
«ethernet»
1..*
primary server
workstation

Bibliografia

- Software Architecture Patterns by Mark Richards (Chapter 1: Layered Architecture); https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html
- https://www.codeproject.com/Articles/8058/Applying-Robustness-Analysis-on-the-Model-View-Con
- https://www.methodsandtools.com/archive/onionsoftwarearchitecture.php
- UML 2 Tutorial; https://www.sparxsystems.com.au/resources/tutorials/uml2/index.html
- UML basics: The component diagram; https://www.ibm.com/developerworks/rational/library/dec04/bell/index.html
- Why Do We Need Architectural Diagrams? https://ionutbalosin.com/2019/02/why-do-we-need-architectural-diagrams/

Something about Layers and Onion Architecture Patterns

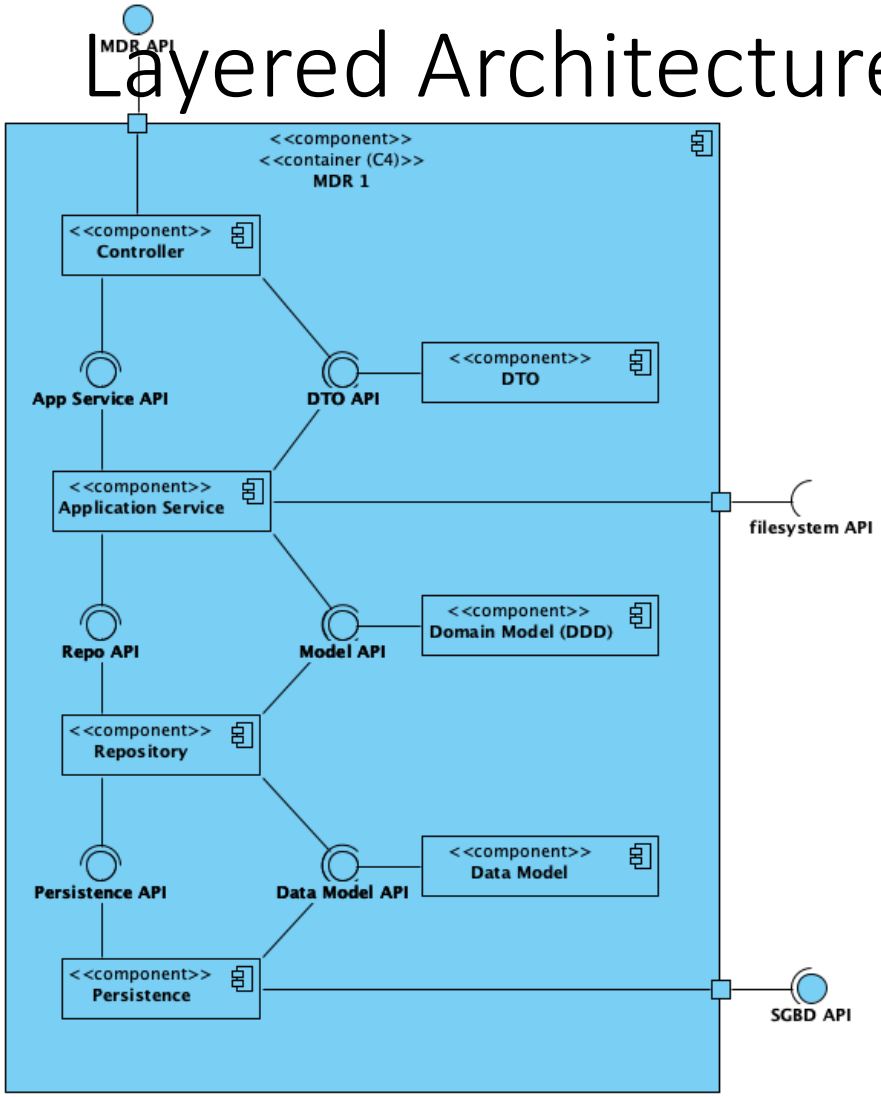
SOLID pattern

- Single Responsibility Principle
  - “A class should only have a single responsibility, that is, only changes to one part of the software's specification should be able to affect the specification of the class.”
- Open/Closed Principle
  - "Software entities ... should be open for extension, but closed for modification."
- Liskov Substitution Principle
  - "Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program."
- Interface Segregation Principle
  - "Many client-specific interfaces are better than one general-purpose interface."
- Dependency Inversion Principle
  - “depend upon abstractions, [not] concretions”

In Wikipedia: Martin, Robert C. (2000). "Design Principles and Design Patterns" (PDF). Archived from the original (PDF) on 2015-09-06.

Layered Architecture vs. SOLID

This layered architecture does not conform to onion pattern/style.



Single Responsibility Principle

→ Adopted: each layer (e.g. controller) has a single responsibility. Other components (e.g. DTO) has a single responsibility too.

Open/Closed Principle

→ Not explicitly adopted.

Liskov Substitution Principle

→ Not explicitly adopted.

Interface Segregation Principle

→ Not explicitly adopted.

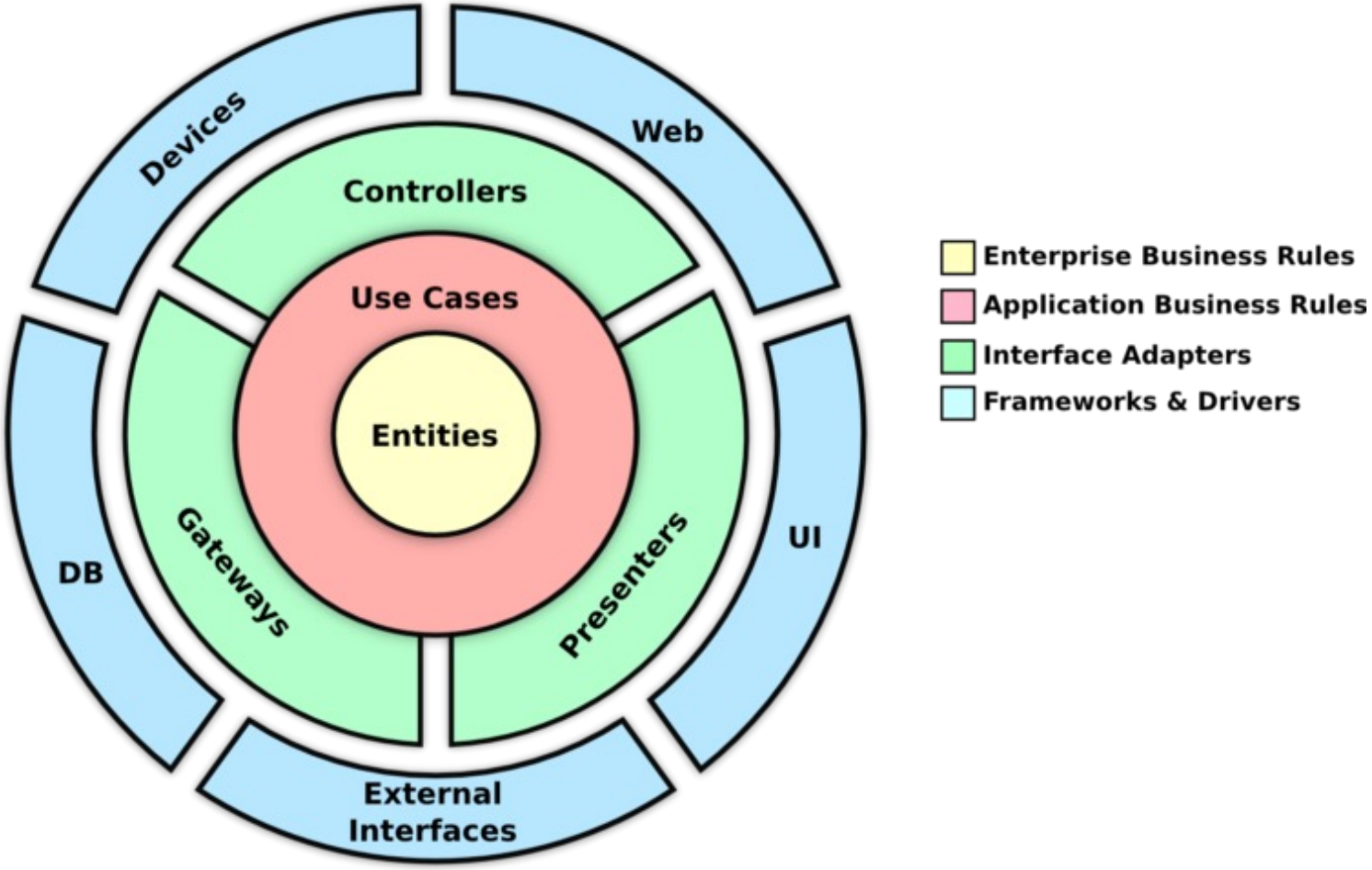
→ Should the AppX API be segregated? How?

Dependency Inversion Principle

→ Not explicitly adopted.

Onion Architecture

This is a methaphore!

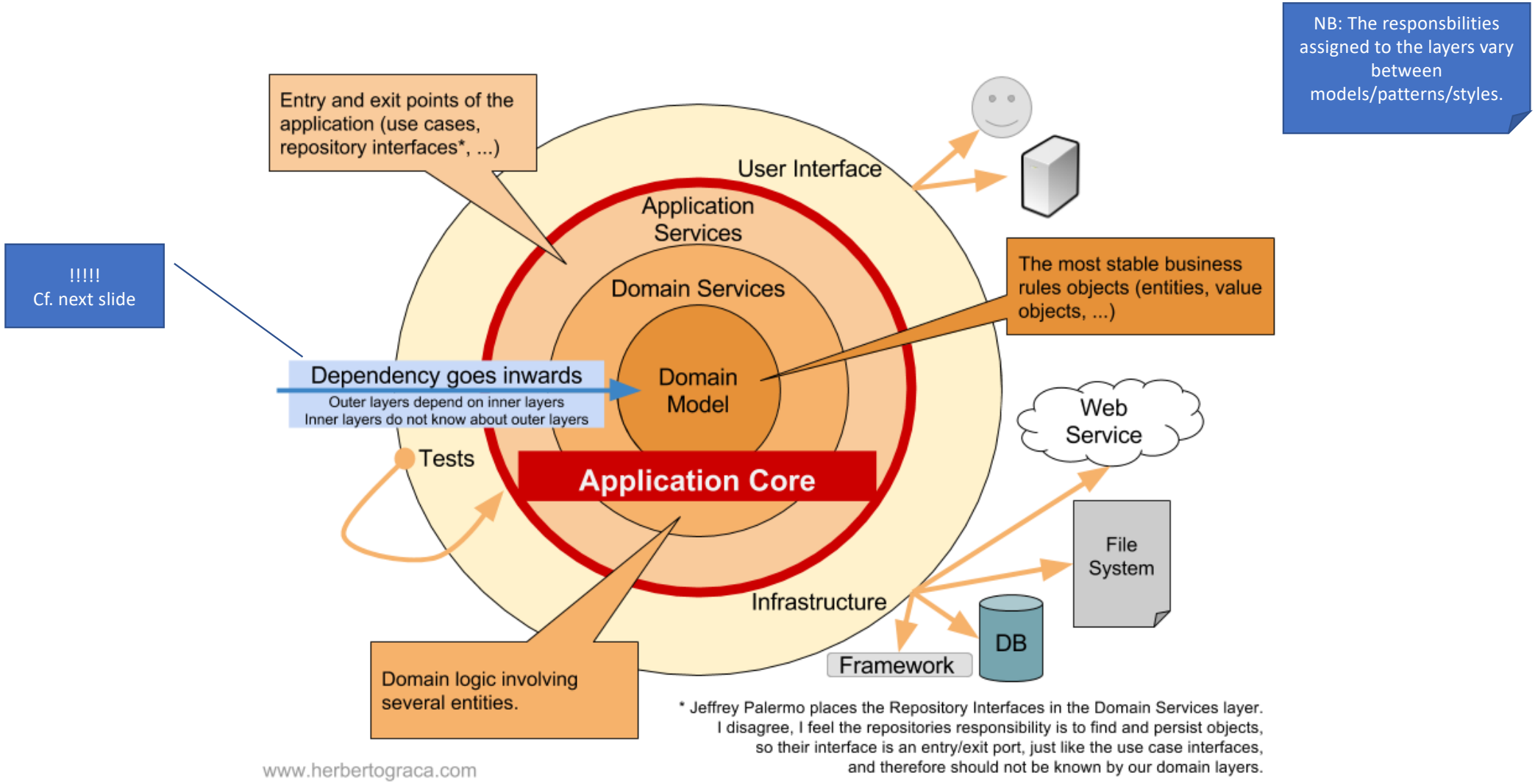


Onion vs. Hexagonal vs. Clean vs. ...

Architecture styles/patterns with the same objectives:

- **Independent of Frameworks**. The architecture does not depend on the existence of some library of feature laden software. This allows you to use such frameworks as tools, rather than having to cram your system into their limited constraints.
- **Testable**. The business rules can be tested without the UI, Database, Web Server, or any other external element. Independent of UI. The UI can change easily, without changing the rest of the system. A Web UI could be replaced with a console UI, for example, without changing the business rules.
- **Independent of Database**. You can swap out Oracle or SQL Server, for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.
- **Independent of any external agency**. In fact your business rules simply don't know anything at all about the outside world.

https://stackoverflow.com/questions/23479879/clean-architecture-vs-onion-architecture

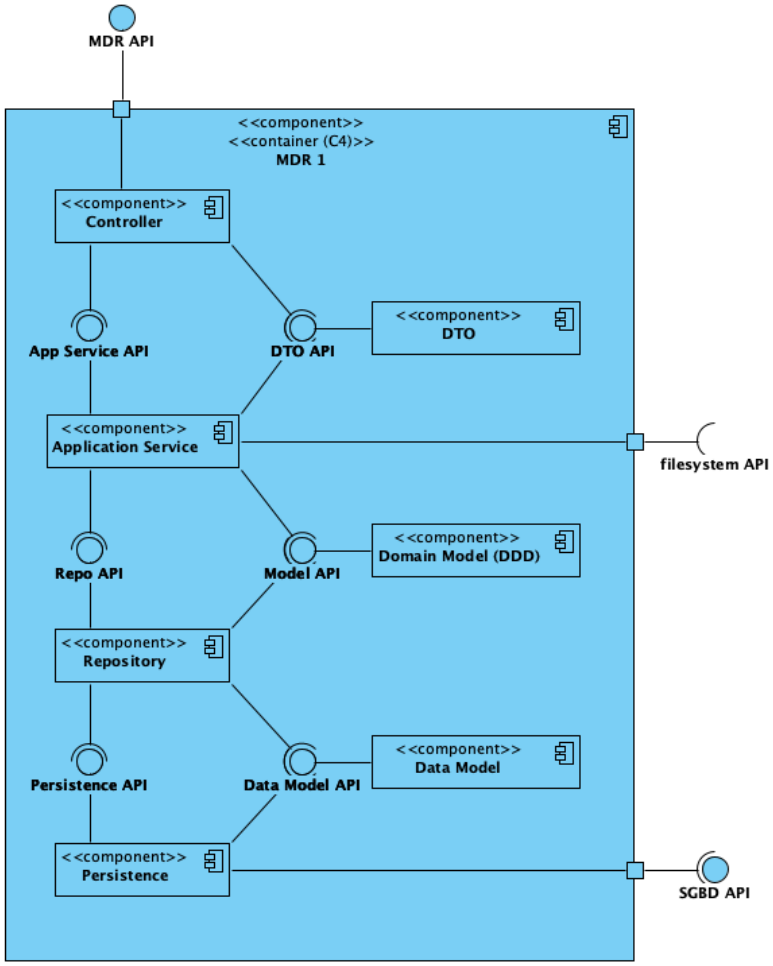


https://herbertograca.com/2017/09/21/onion-architecture/

From Layered Architecture to Onion

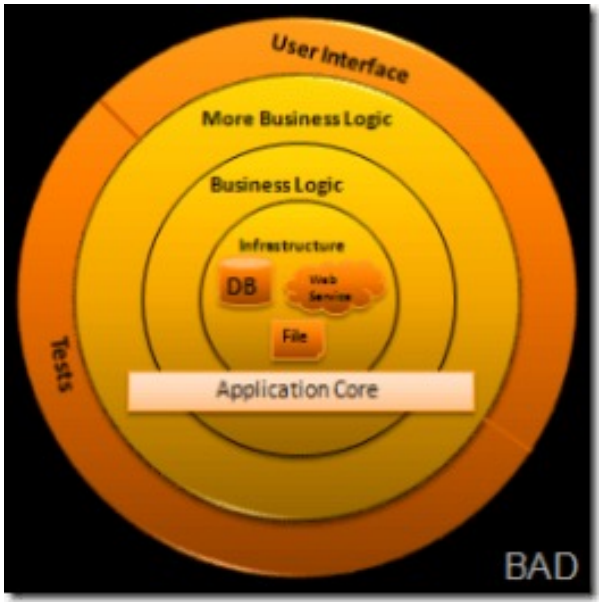
Classify layers/components according to Onion style/pattern

Overlapped vs. Concentric Layered Architecture

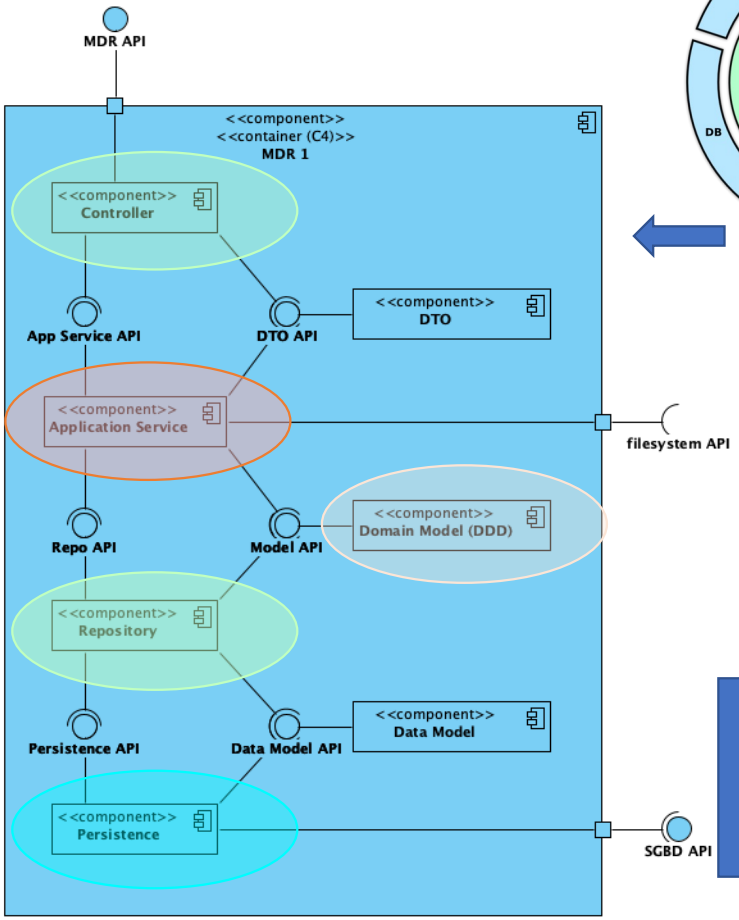


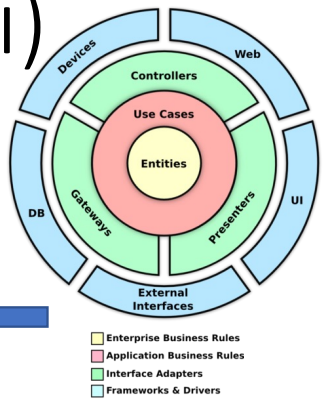
This is not what Onion pattern/style is about.

This layered architecture does not conform to onion pattern/style.

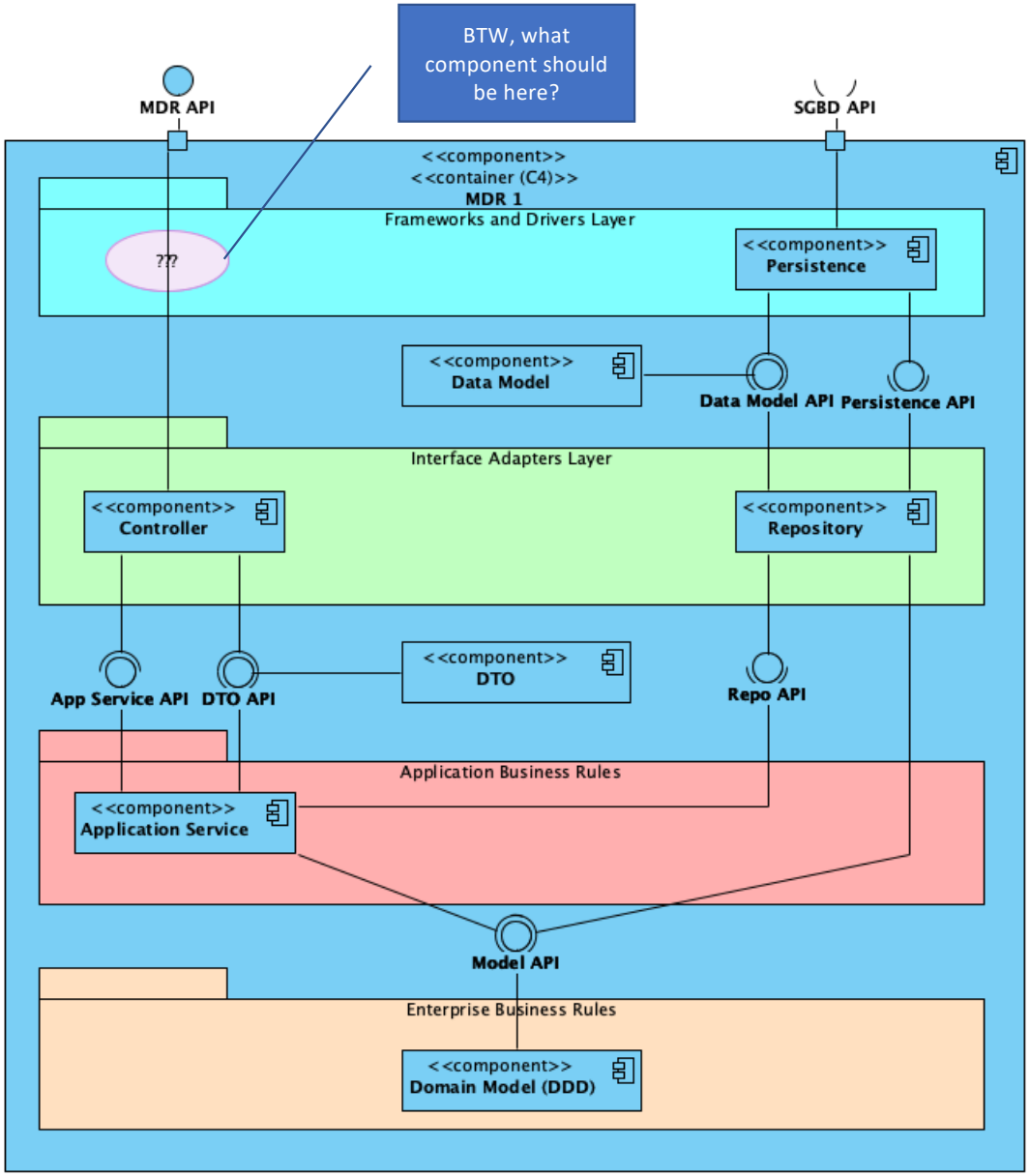


Classify and layout layers (i)

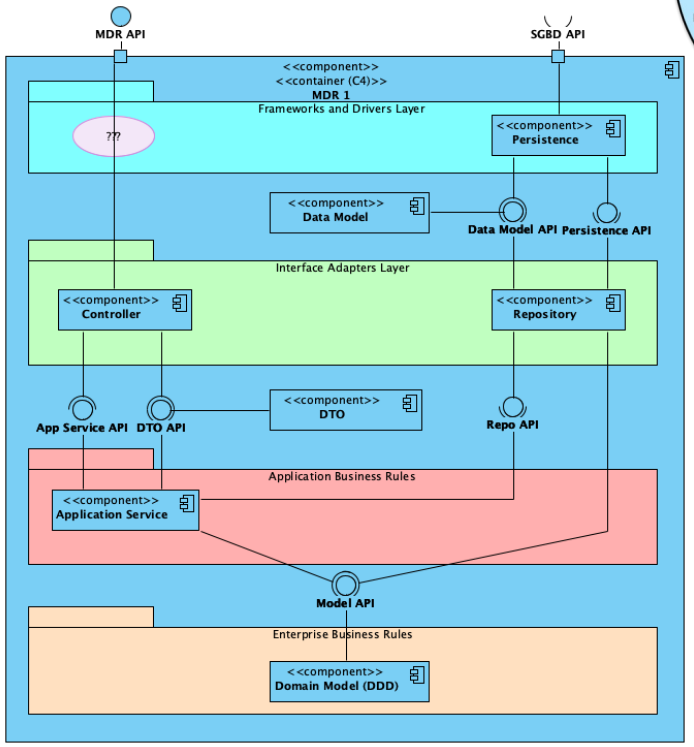


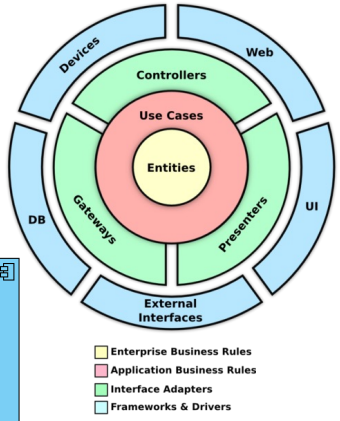


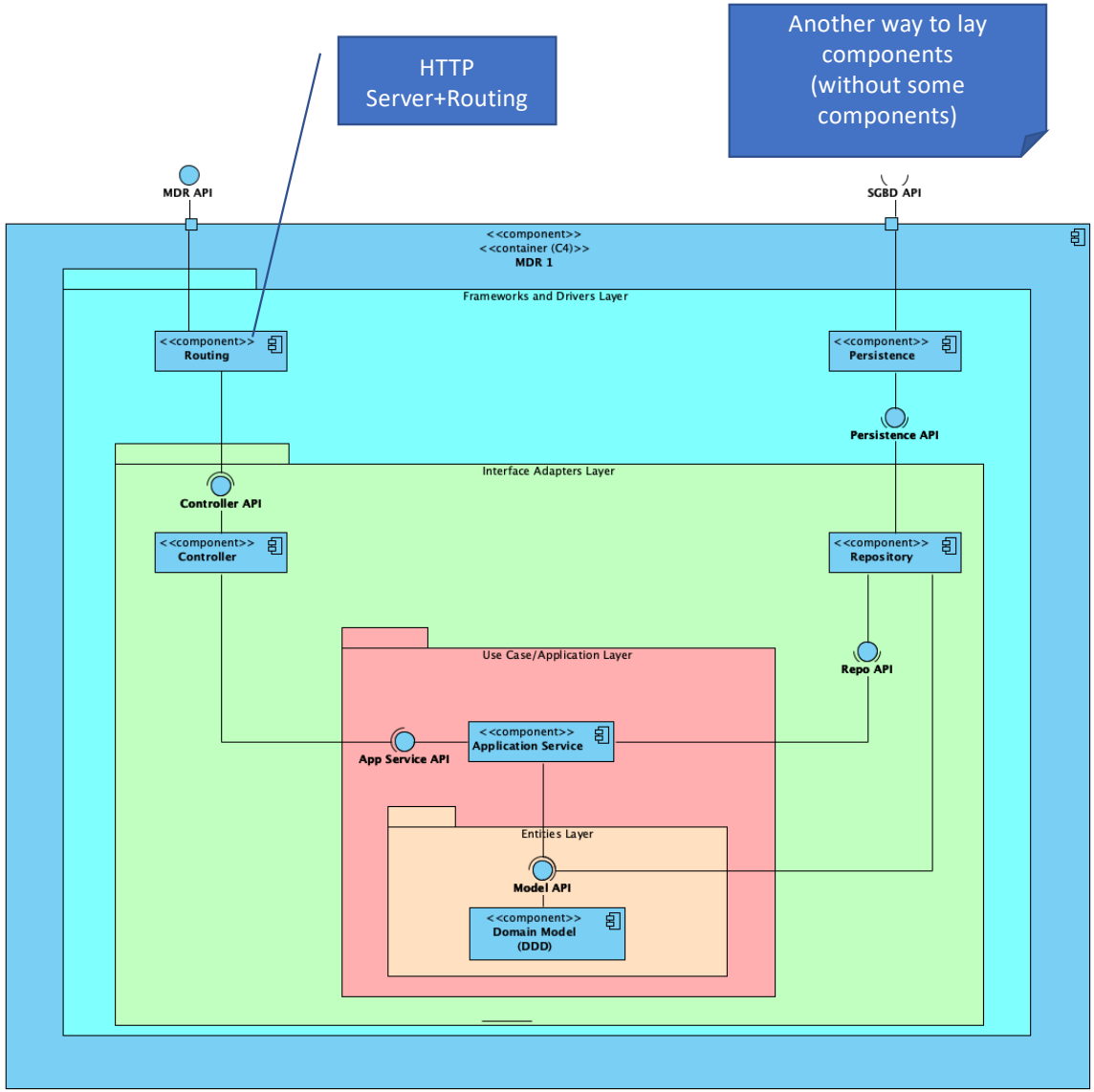
Colors are semantically meaningless but may help following from the onion metaphor diagram to the UML representation.



Classify and layout layers (ii)

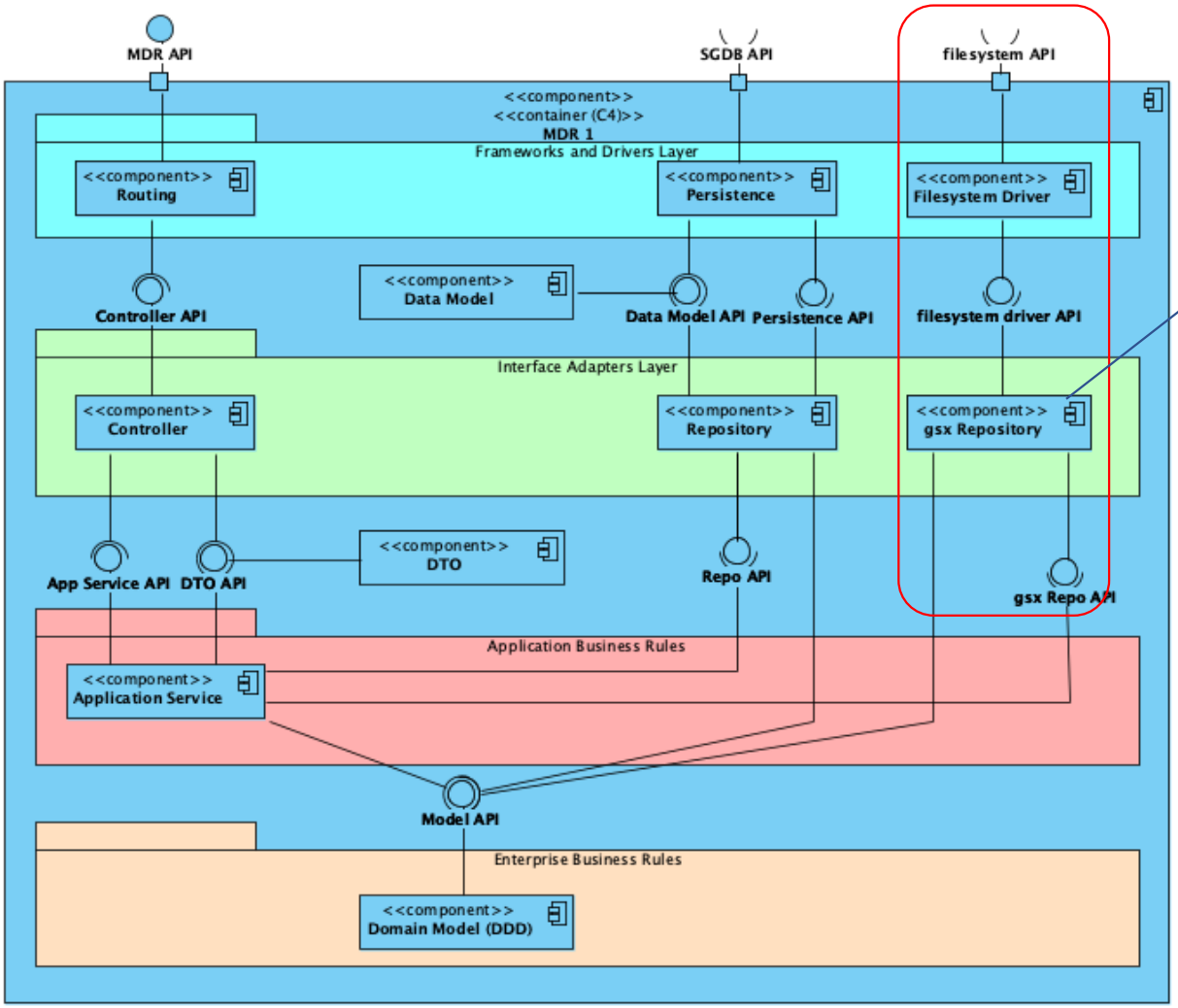






NB: still, without filesystem supported/included.

Including the filesystem API



Is it necessary?
If its interface is different from that of the “Repository”, in order to apply the ISP, yes. Otherwise, no.

But this is not enough

Dependency Inversion Principle is not explicitly adopted

DIP: Dependency Inversion Principle

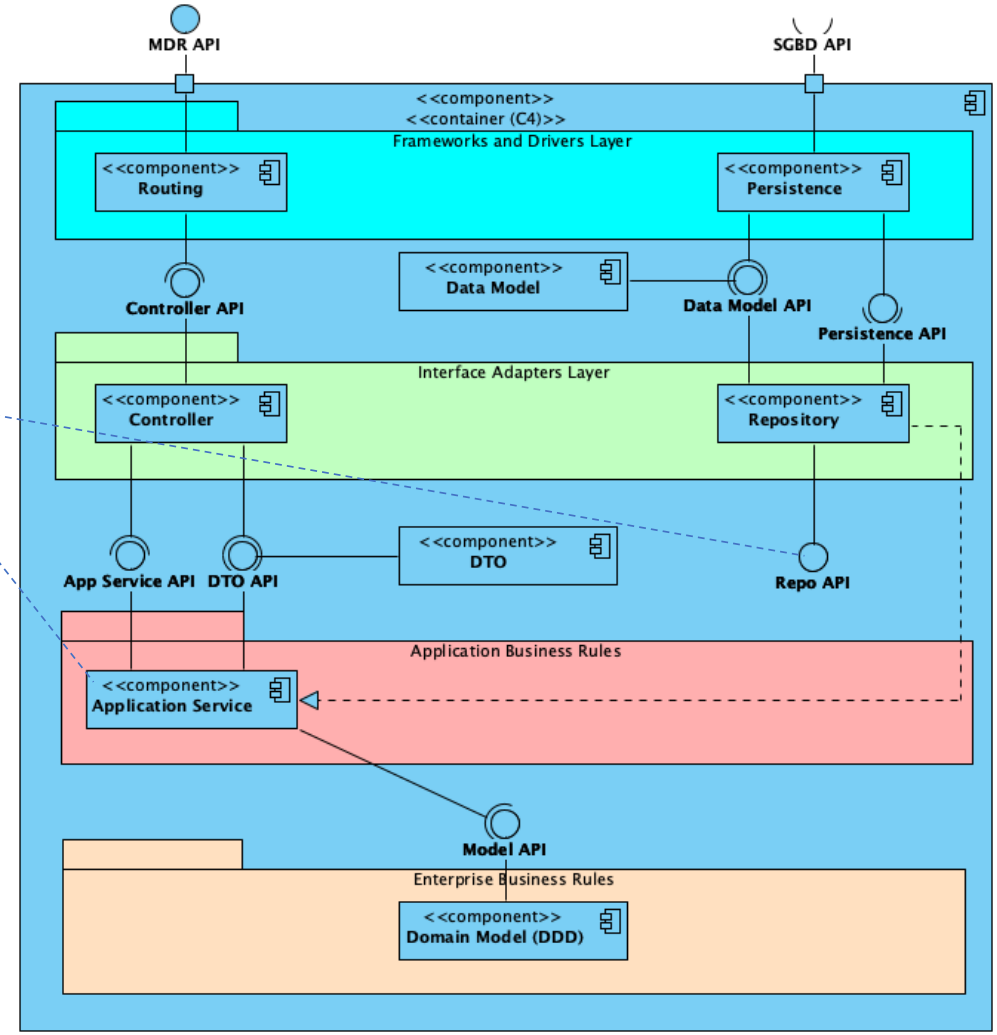
The Application Services classes define the contract that the Repository classes must implement.

The Repository implements interface (contract).

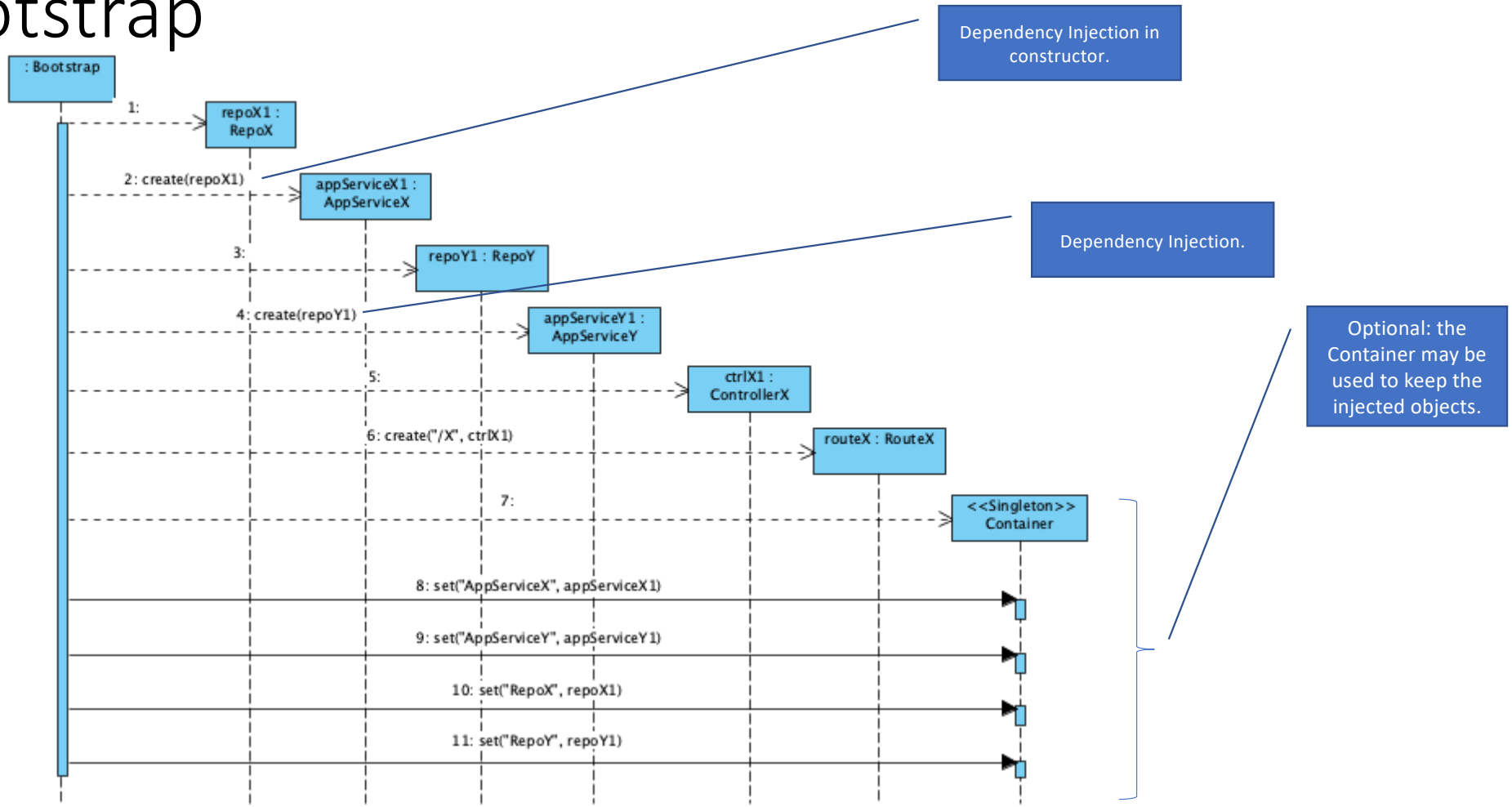
The Application Service does not need to know in development time the Repository class that implements the interface. Instead, the Repository class/object can be provided during execution (dependency injection).

Hence, the Application Service behavior (i.e. which object is called) is controlled by the upper layers (Inversion of Control/Dependency Inversion Principle).

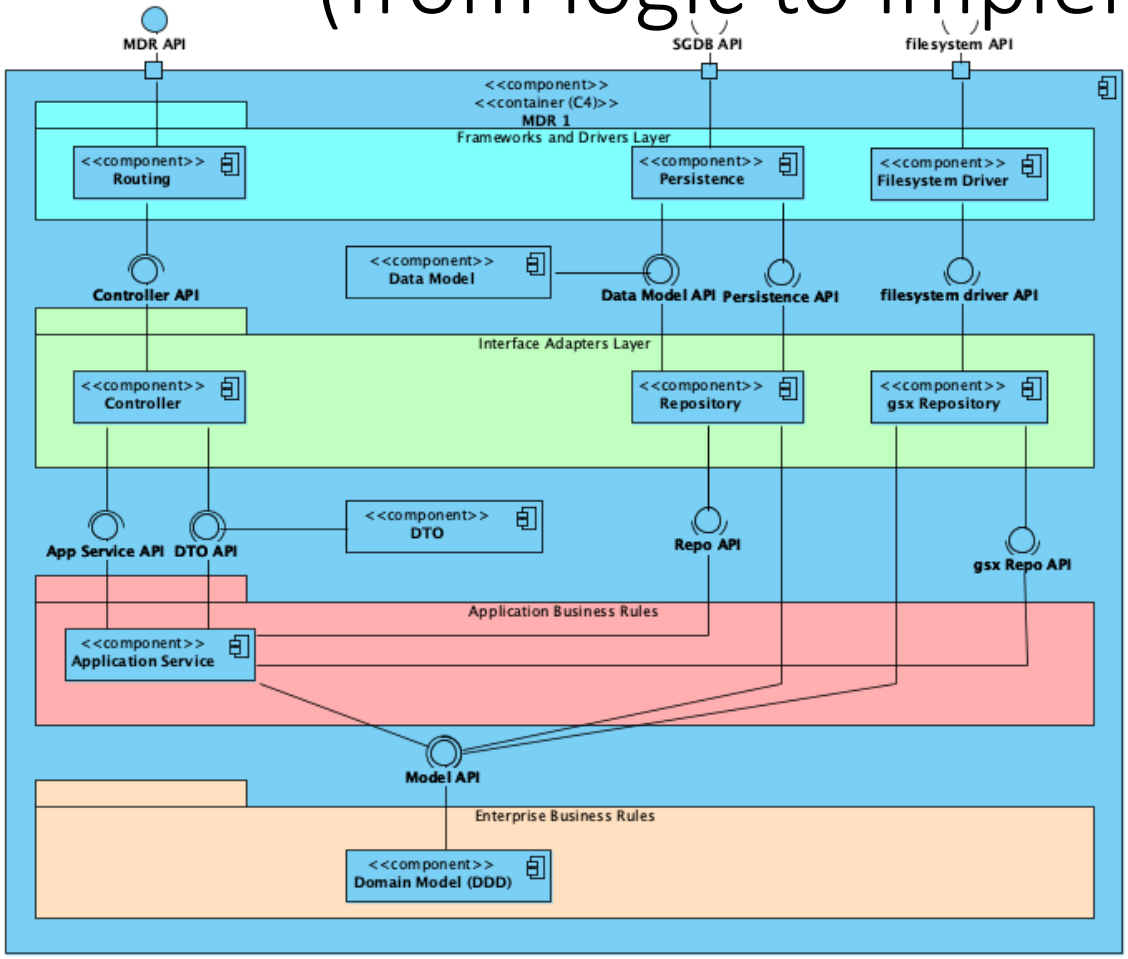
Cf. next slides

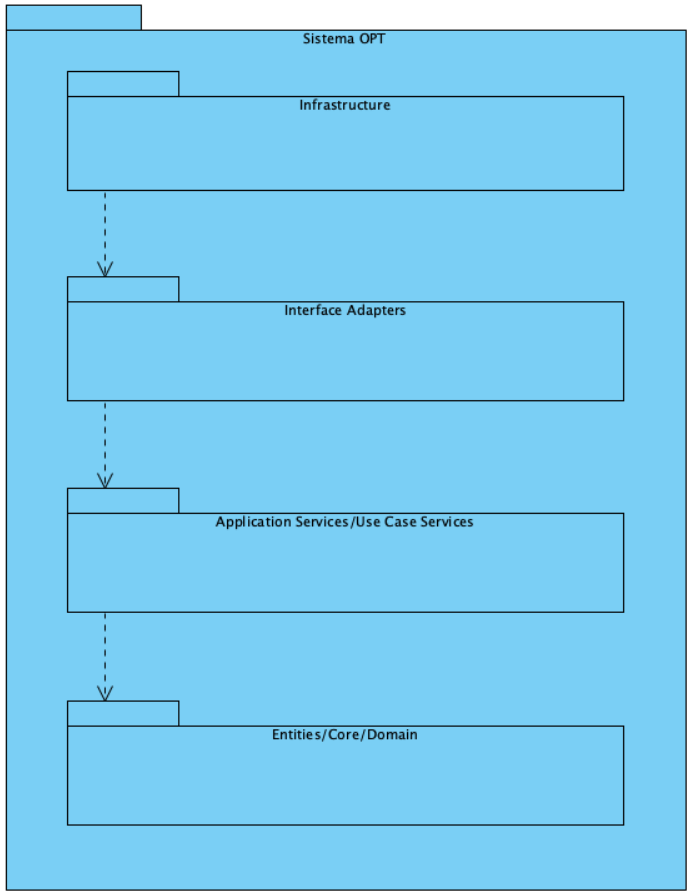


Bootstrap

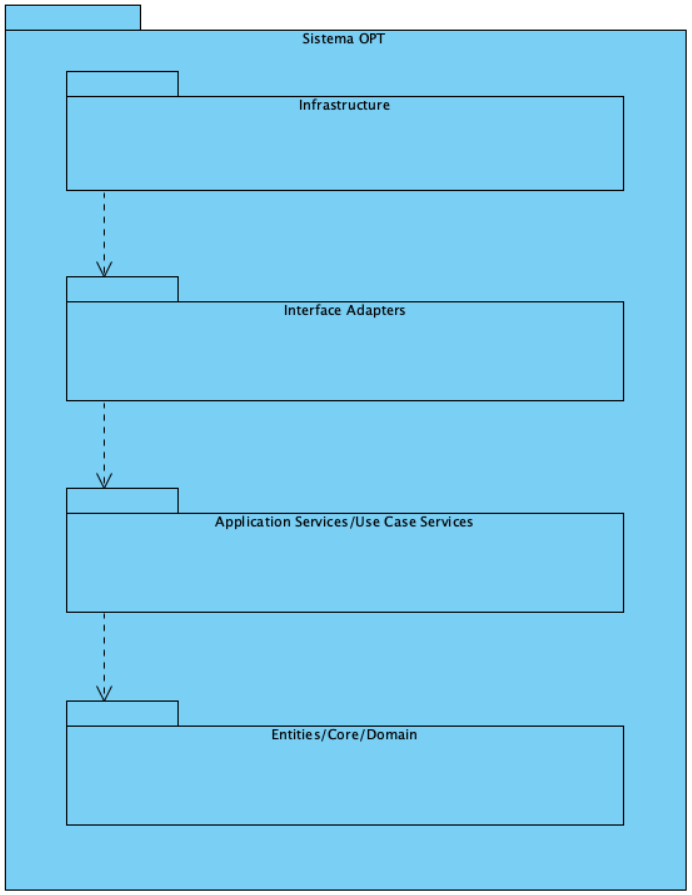

: Bootstrap
1:
repoX1 : RepoX
2: create(repoX1)
appServiceX1 : AppServiceX
3:
repoY1 : RepoY
4: create(repoY1)
appServiceY1 : AppServiceY
5:
ctrlX1 : ControllerX
6: create("/X", ctrlX1)
routeX : RouteX
7:
<<Singleton>> Container
8: set("AppServiceX", appServiceX1)
9: set("AppServiceY", appServiceY1)
10: set("RepoX", repoX1)
11: set("RepoY", repoY1)
Dependency Injection in constructor.
Dependency Injection.
Optional: the Container may be used to keep the injected objects.

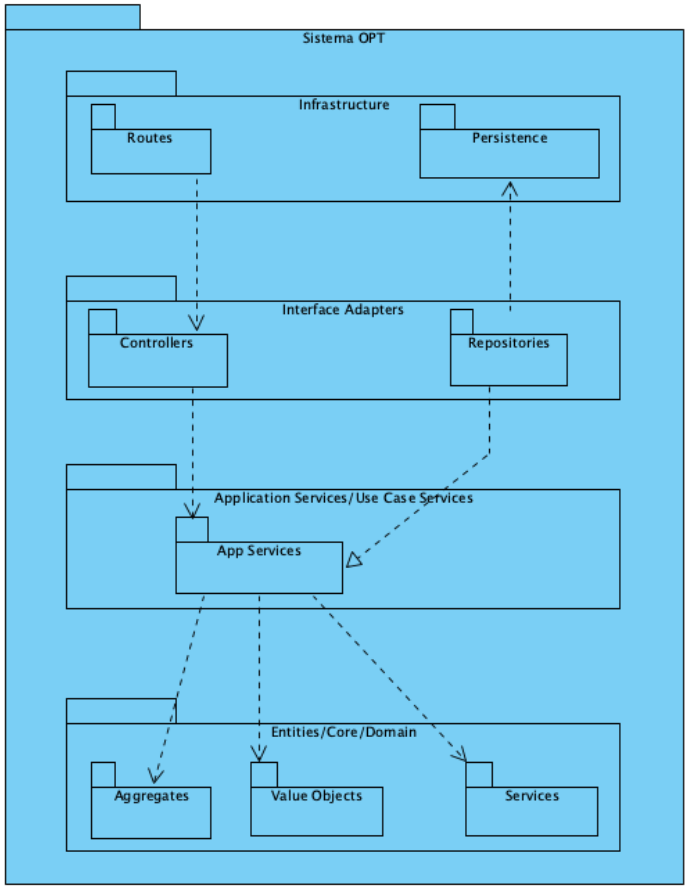
Implementation View
(from logic to implementation)

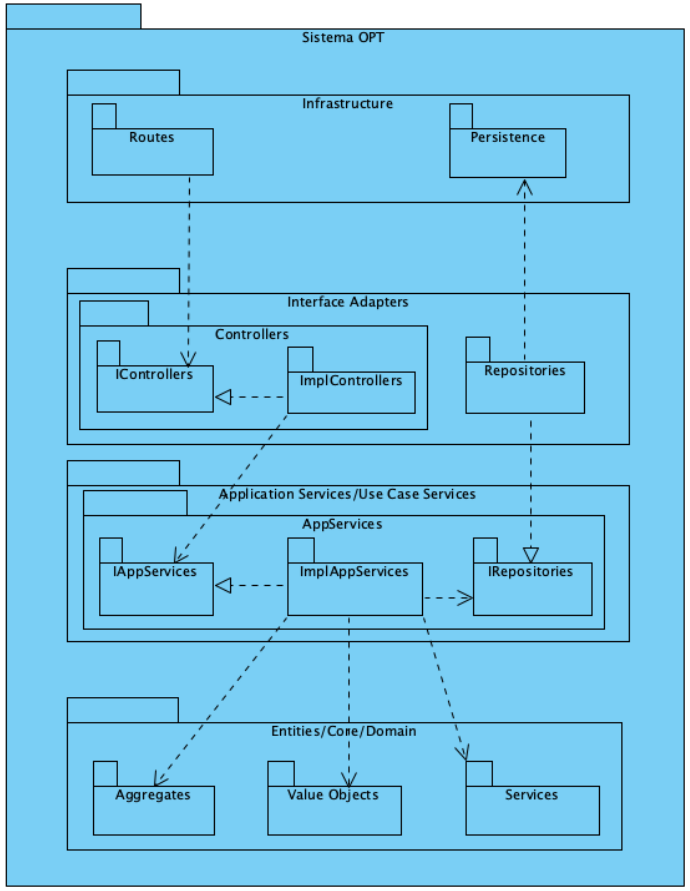




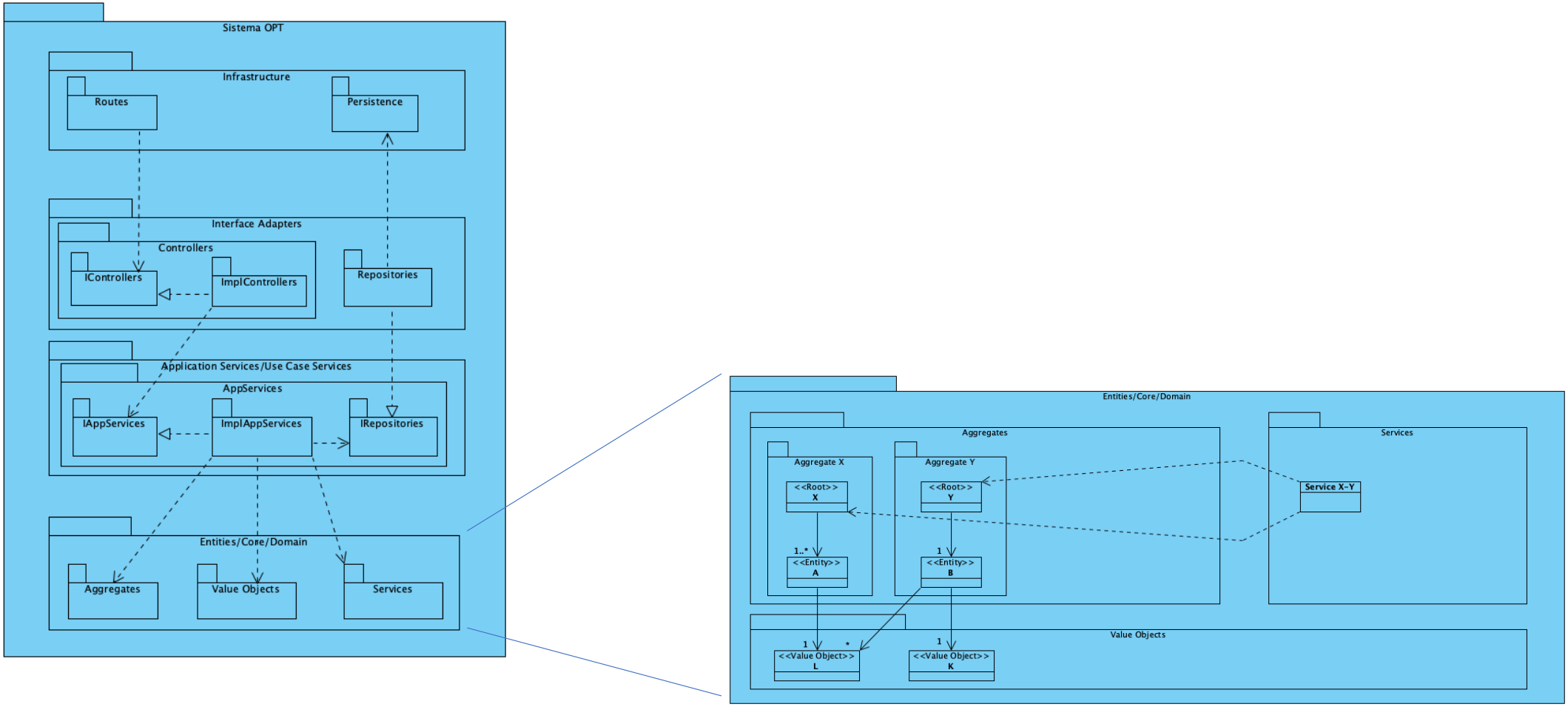
Implementation Views (increased finer granularity)



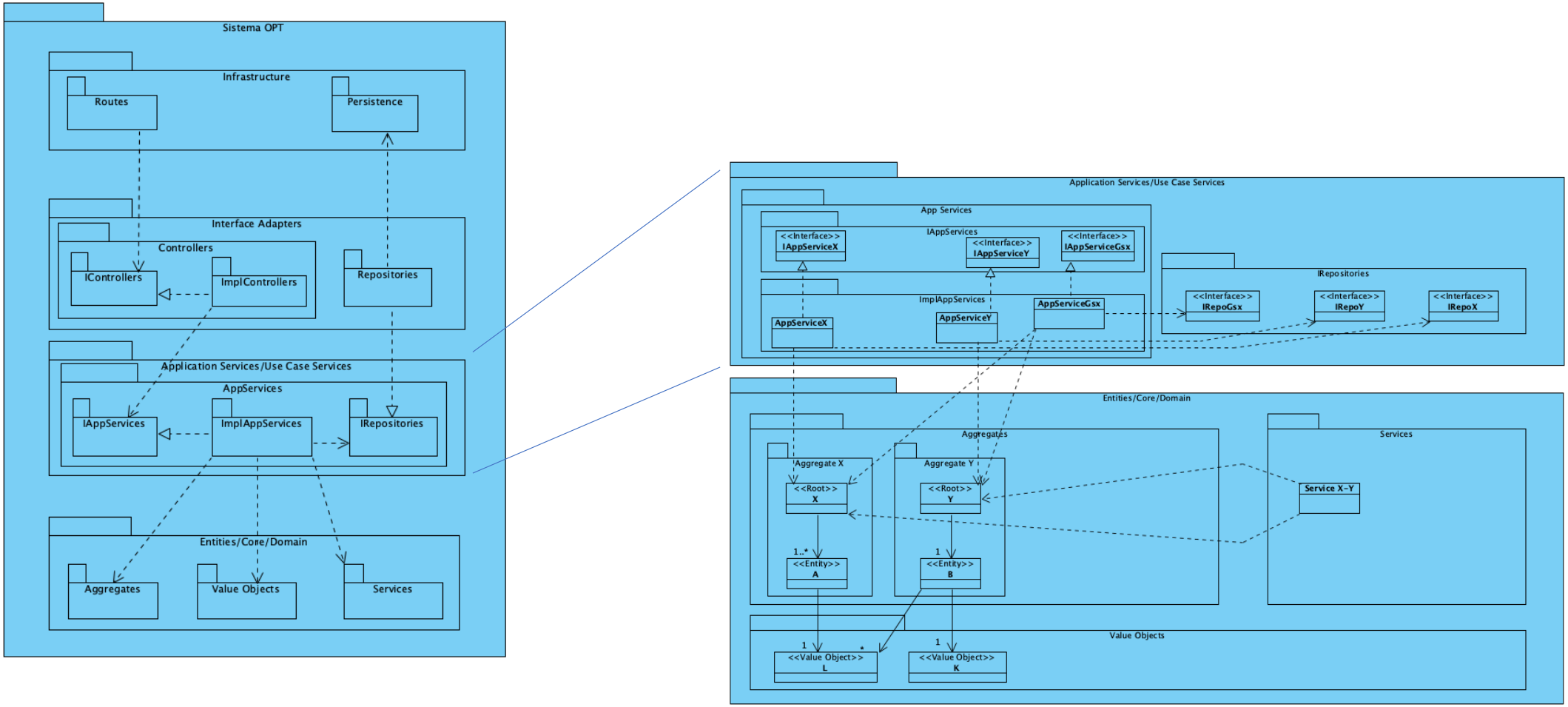




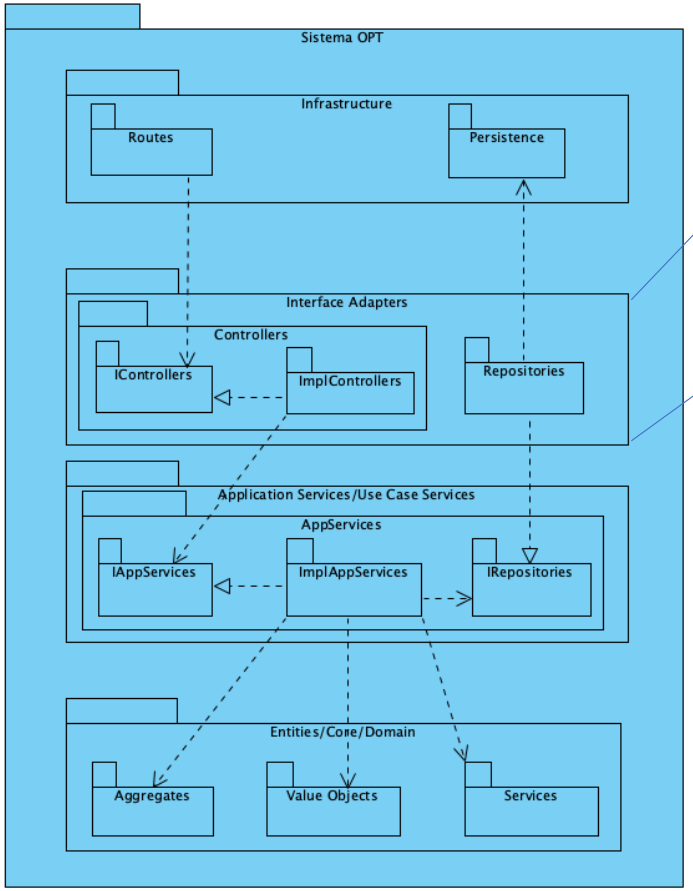
Class diagram (i)

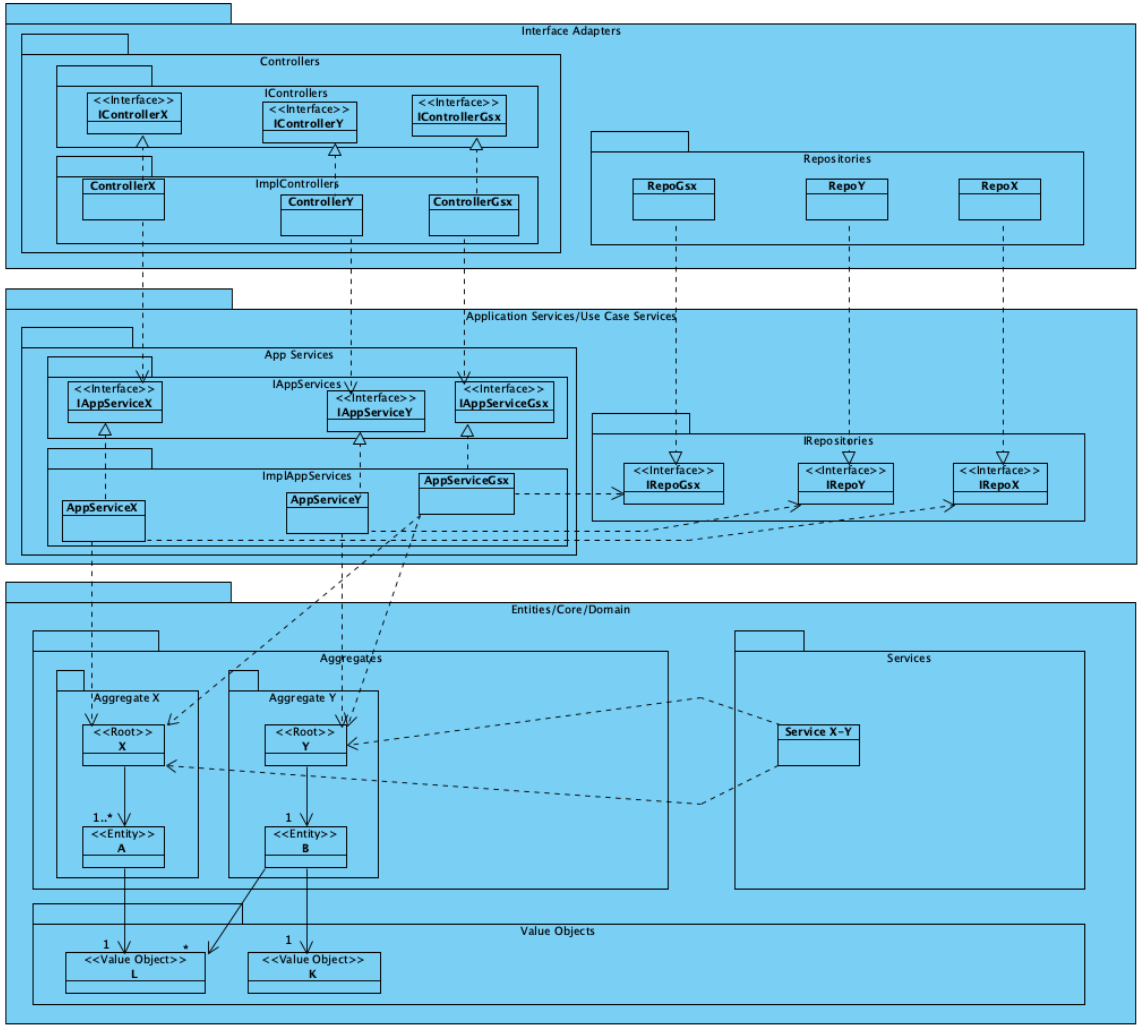


Class diagram (ii)

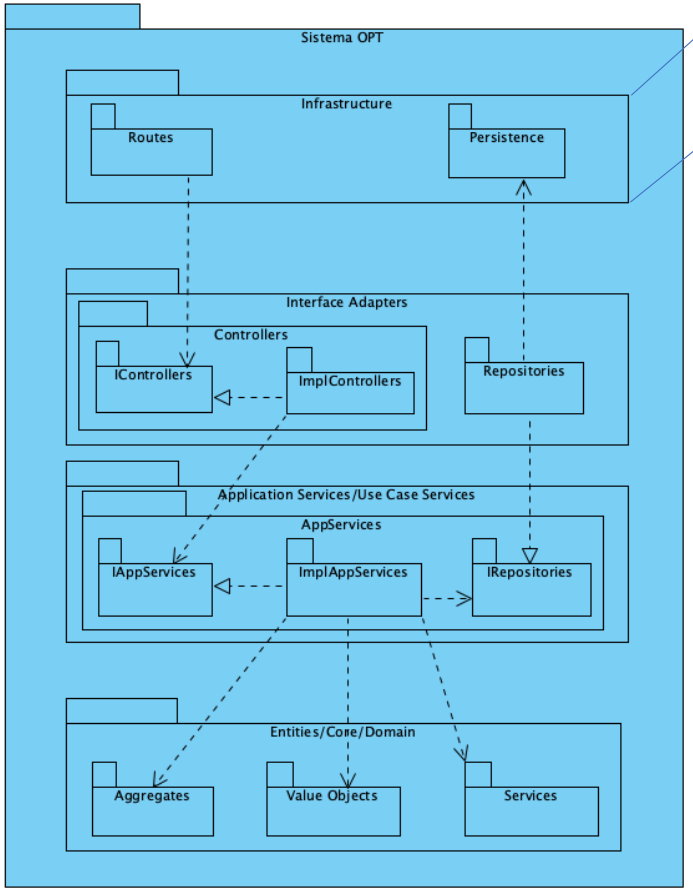


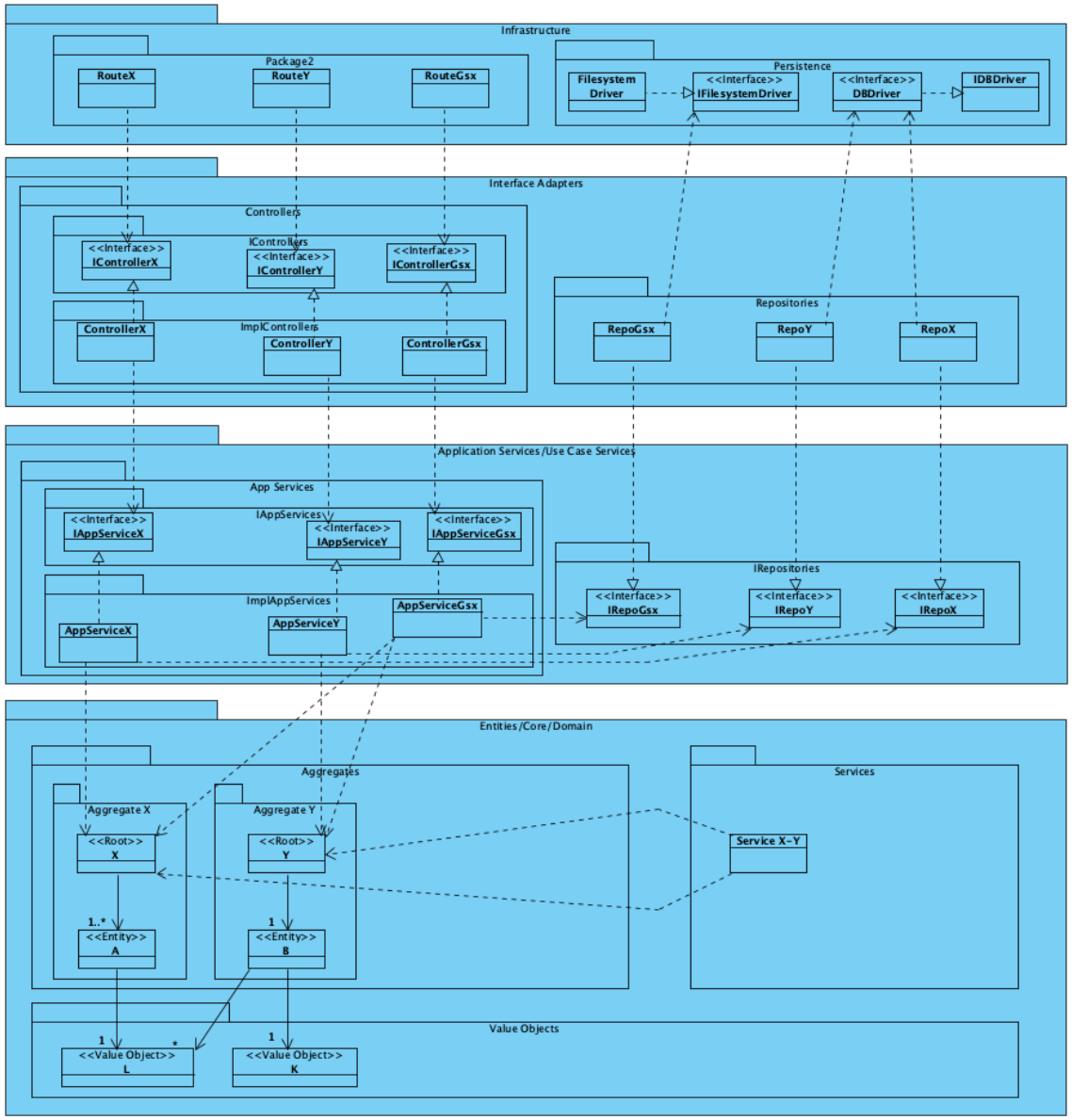
Class diagram (iii)





Class diagram (iv)

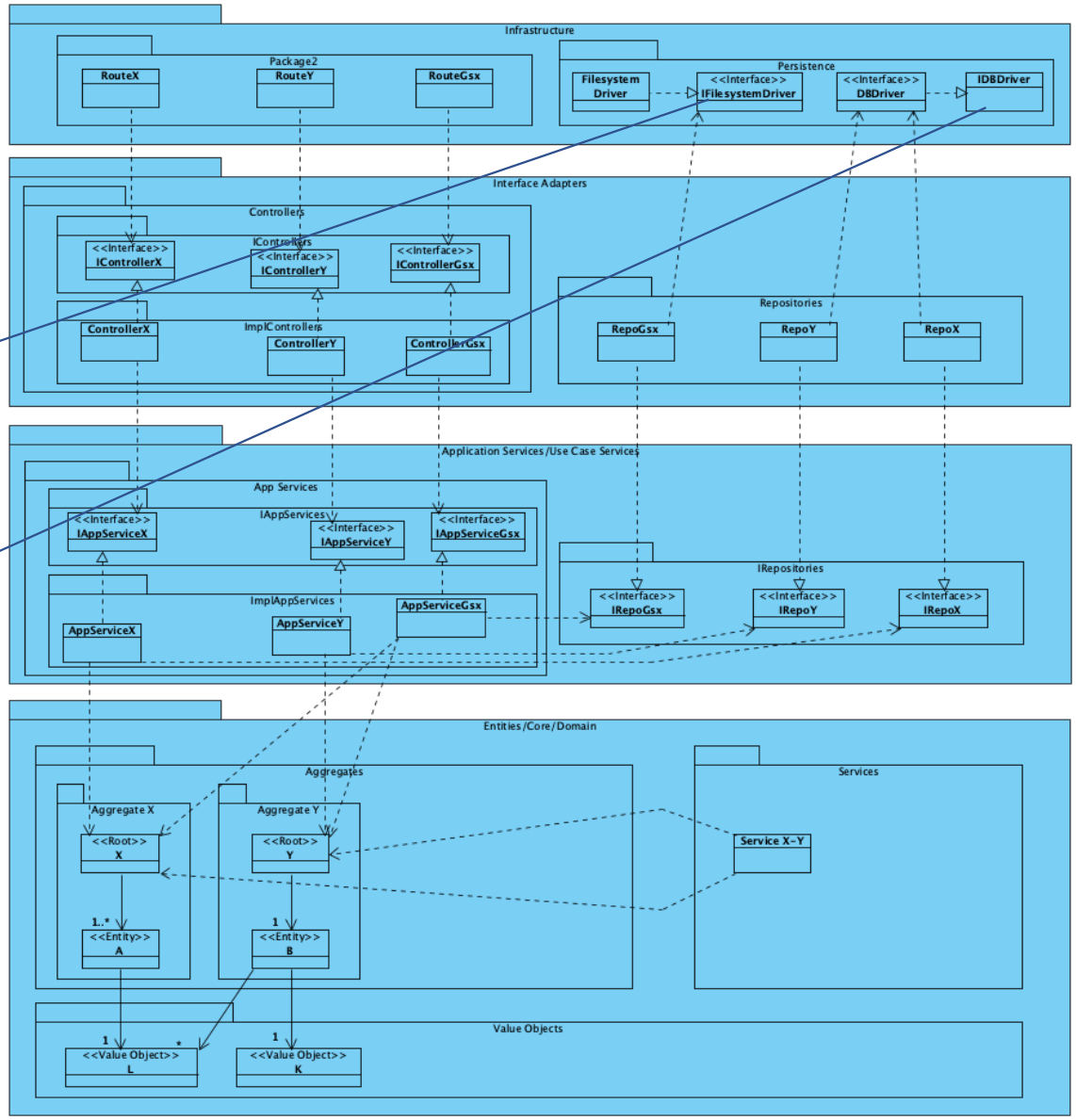


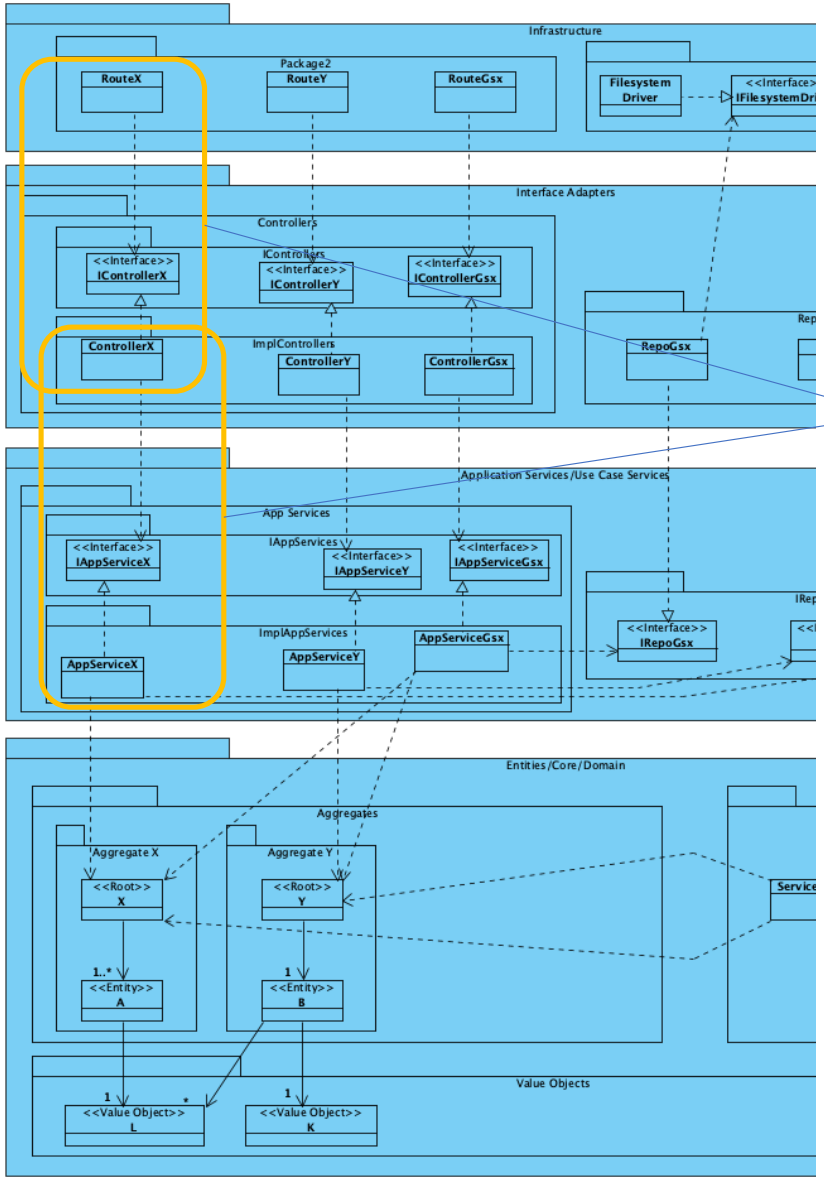


Class diagram (v)

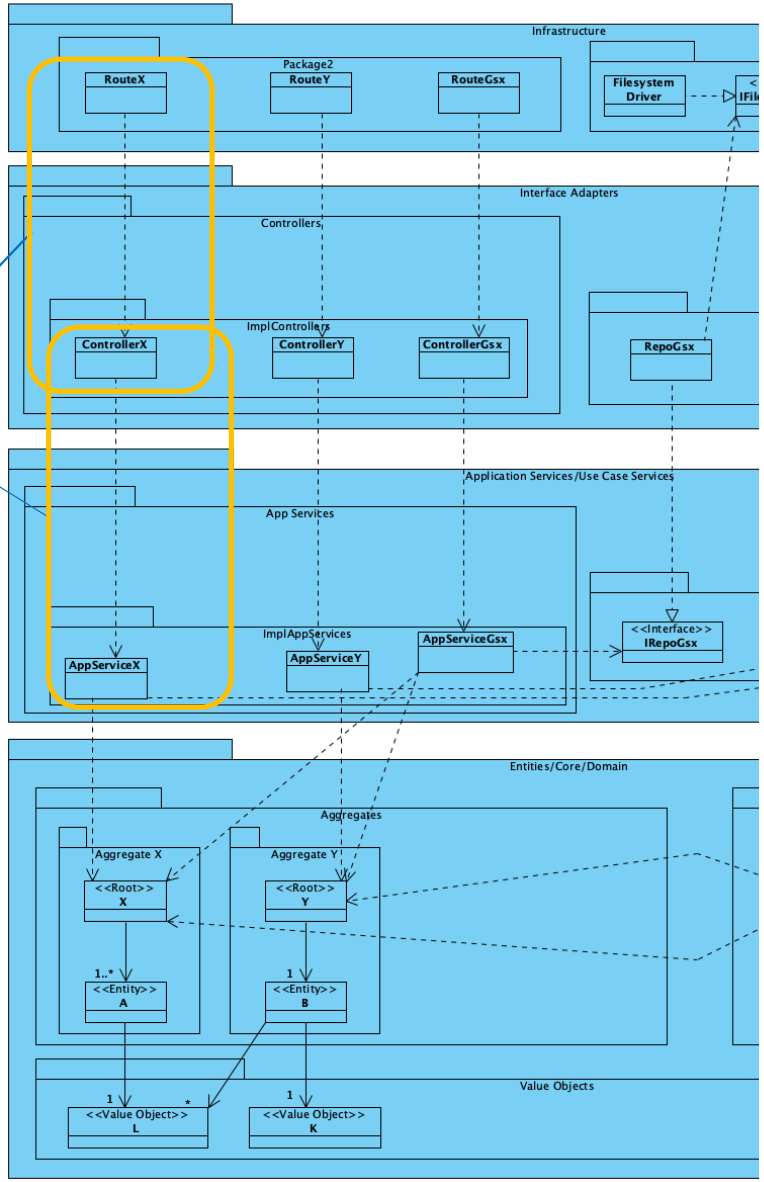
It would be uncommon that the FilesystemDriver implements an interface defined in/by Repositories.

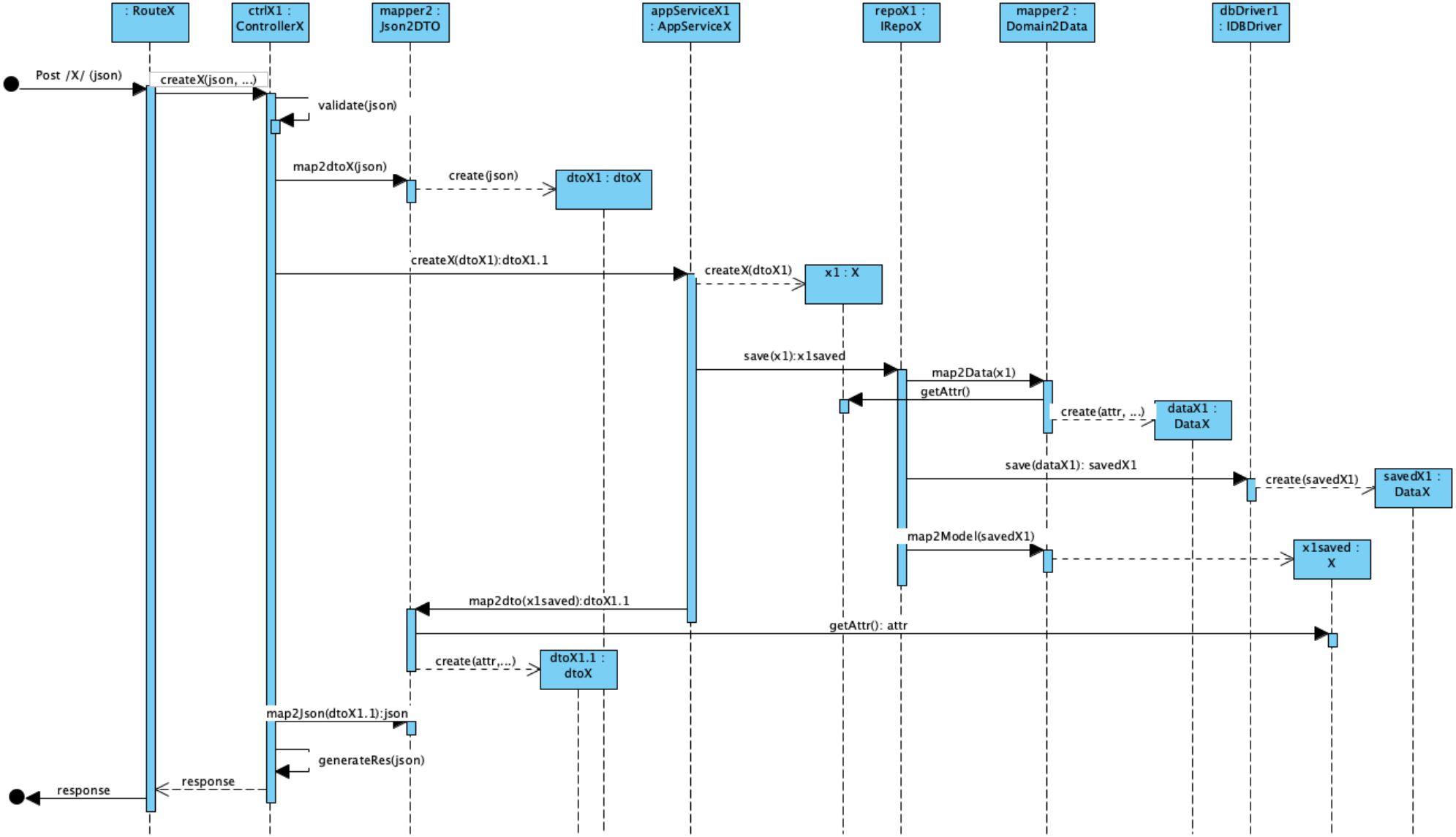
It would be uncommon that the DBDriver implements an interface defined in/by Repositories.





NB: because the dependency is from exterior to interior, it is not mandatory to adopt Interface nor Dependency Injection.




: RouteX
ctrlX1 : ControllerX
mapper2 : Json2DTO
appServiceX1 : AppServiceX
repoX1 : IRepoX
mapper2 : Domain2Data
dbDriver1 : IDBDriver
Post /X/ (json)
createX(json, ...)
validate(json)
map2dtoX(json)
create(json)
dtoX1 : dtoX
createX(dtoX1):dtoX1.1
createX(dtoX1)
x1 : X
save(x1):x1saved
map2Data(x1)
getAttr()
create(attr, ...)
dataX1 : DataX
save(dataX1): savedX1
create(savedX1)
savedX1 : DataX
map2Model(savedX1)
x1saved : X
map2dto(x1saved):dtoX1.1
getAttr(): attr
create(attr,...)
dtoX1.1 : dtoX
map2Json(dtoX1.1):json
generateRes(json)
response
response

References and Bibliography

- https://blog.ploeh.dk/2013/12/03/layers-onions-ports-adapters-its-all-the-same/
- https://jeffreypalermo.com/tag/onion-architecture/
- https://stackoverflow.com/questions/23479879/clean-architecture-vs-onion-architecture
- https://herbertograca.com/2017/09/21/onion-architecture/

Arquitetura de Software

Métodos/Processos

Design

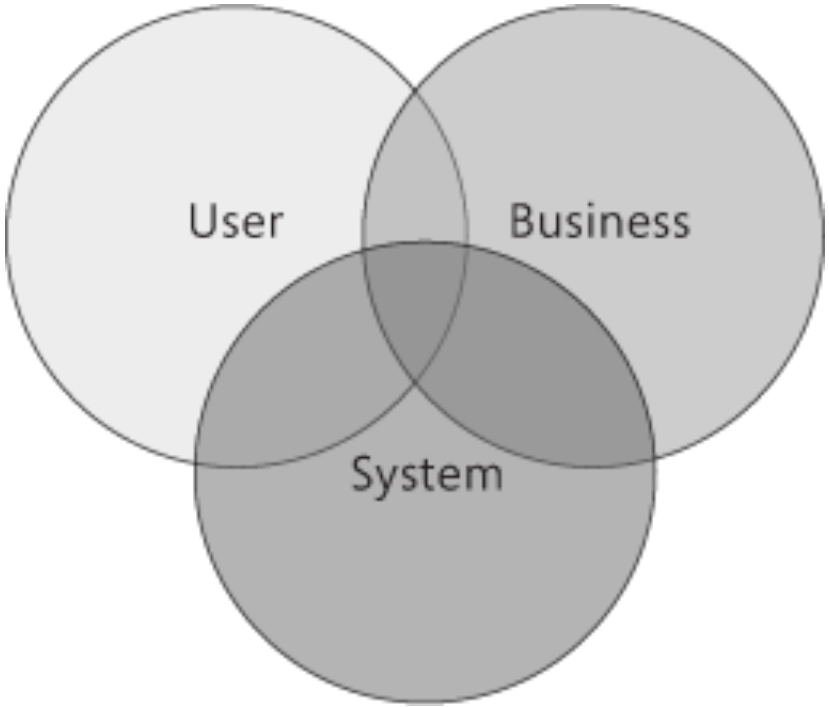
- Transformar requisitos no design do sistema a desenvolver
  - “Software architecture has to do with the large scale, the organization, patterns, collaborations, connections of a system and major subsystems.”
- Desenvolver uma arquitectura robusta do sistema
  - Modular (Modularidade)
  - Manutenível (Manutenibilidade)
- Adaptar o design ao ambiente de implementação e implantação

Análise Arquitectural: sumário

- Aborda o sistema numa escala grande
  - 1KM de largura, 1CM de profundidade

- Grandes ideias acerca do sistema (ou sistema de sistemas), no que respeita a:
  - Motivação
  - Restrições
  - Organização
  - Padrões
  - Responsabilidades
  - Interfaces

Análise e Design Arquitetural:
na interceção de 3 forças



Análise Arquitectural: definição

- O conjunto de decisões significativas acerca da organização do sistema de software
- Selecção dos elementos estruturais e suas interfaces que compõem o sistema
- Comportamento desses elementos nas suas colaborações
- Composição dos elementos estruturais e comportamentais em sistemas progressivamente maiores
- O estilo de arquitectura no contexto da organização: elementos, interfaces, colaborações e composição

Análise Arquitectural: resumo

- Especialmente relacionado com:
  - Atributos de qualidade (Requisitos Não Funcionais)
  - Utilizadores
  - Sistema
  - Negócio

  dando resposta aos requisitos funcionais
- Abordagem de larga escala: de sistema e sub-sistemas
- Procura e proposta de <u>alternativas</u> de solução, discutidas, justificadas e documentadas
- Dependência e “trade-offs” entre factores e decisões
  - Raramente (nunca?) existe a solução perfeita

Análise Arquitectural: 2 atividades

- Investigação/Pesquisa arquitetural
  - identificar requisitos funcionais e (especialmente) não funcionais que têm impacto significativo no desenho do sistema
  - semelhante à análise de requisitos, mas com foco nos requisitos que interessam para a “escala grande”

- Desenho arquitetural
  - envolve resolver requisitos no desenho da arquitectura de software
  - as decisões são tomadas considerando o software, hardware, redes de comunicação, e operações de negócio, entre outras

Investigação Arquitectural

Questões importantes

- Requisitos não funcionais
  - Definidos na Especificação Suplementar, e.g.:
    - A utilização do sistema pelo administrador, organizadores, autores (excepto o seu registo) e revisores requer autenticação;
    - O software deve ter uma taxa de disponibilidade >98%
    - Adoção do processo de desenvolvimento de software iterativo e incremental.
    - Adoção de boas práticas de design.

- Pontos de variação
  - Pontos do sistema que têm funcionamento diferente/variável, e.g.:
    - Vários e distintos mecanismos de autenticação
    - Vários e distintos serviços terceiros
    - Vários e distintos algoritmos/estratégias de distribuição de revisão por revisores

- Pontos de evolução
  - Especulação sobre a necessidade do sistema evoluir no futuro, de tal forma que pode condicionar o seu desenvolvimento actual ou adaptação posterior, e.g.
    - Vários e distintos mecanismos de persistência
    - Vários e distintos web services de fontes dos mesmos dados
  - Cuidado com a sobre-engenharia (demasiada engenharia) para prever evoluções que podem não ocorrer.
    - Perspetiva XP: ”fazer o que de mais simples funciona em cada momento e *refactoring* e re-engenharia sempre que necessário”

Factores Arquitecturais e Cenários de Qualidade

- Factores Arquitecturais
  - São requisitos não funcionais que conduzem decisões arquitecturais
- Cenários de Qualidade
  - UP/RUP sugere desenvolver cenários de qualidade na forma <estímulo>,<resposta mensurável>
- Por exemplo:
  - <Quando uma venda completa é enviada para o sistema de cálculo de impostos>,
  - <o resultado deve ser recebido até 2 segundos em 90% dos casos, em condições médias de carga (i.e. <100 pedidos por minuto)
- Tabelas de fatores arquiteturais (incluídas na SS)

Tabela de fatores arquiteturais (exemplo)

Factor	Cenário de Qualidade	Variabilidade	Impacto nas partes interessadas	Prioridade/ Probabilidade	Risco/ Severidade	Decisão
Confiabilidade – Recuperação						
Recuperar de falha de conexão com serviço	Quando o serviço falha, deve-se reestabelecer a conexão 1 min. após a disponibilização	De momento: serviços mínimos locais são aceitáveis Dentro de 2 anos: replicação total local do serviço	Grande impacto do desenho de larga escala. O cliente não gosta que os serviços falhem pois impede-os de vender	Alta	Médio	3
…	…	…	…	…	…	
Suporte - Adaptabilidade						
A UI/SPA deve permitir acesso a todos os módulos do sistema	O utilizador deve poder aceder a todas as funcionalidades na mesma aplicação	Podem ser consideradas aplicações nativas e aplicações não nativas	Impacto alto no gestor e administrador e médio no cliente	Alta	Baixo	2

		Prioridade		
		Baixa - Raramente	Média - Às vezes	Alta - Sempre
Severidade/Risco	Alta - Não se consegue contornar	3	3	4
	Média - Existem alternativas	2	2	3
	Baixa - Consegue realizar tarefas	1	2	2

exemplo

Resolução de Factores Arquitecturais

- Tomar decisões competentes respeitando as interdependências, prioridades e balanceamento
- Deve-se debater e manter registo de:
  - Soluções alternativas
  - Decisões
  - Factores de influência
  - Motivações
- Memorandos Técnicos (“technical memos”):
  - Registam estas questões
  - São úteis para tomar decisões aquando de alterações ao sistema

Memorando Técnico (Exemplo)

Questão: Legal – Adequação do cálculo de impostos

Problema

Como manter adequado o cálculo de impostos da aplicação?

Sumário da Solução

Compra dum componente de cálculo de impostos.

Factores

Taxas e regras de impostos devem ser corretamente aplicadas.

Solução

Compra dum componente com contracto de manutenção e actualização de regras e taxas. Note-se que componentes distintos podem ser aplicados em diferentes clientes.

Motivação

"Time-to-market", correção & completude, diminuição dos custos de manutenção, foco nas competências dos desenvolvedores (ver alternativas).

Os componentes a comprar são caros, o que afecta a contenção de custos e objectivos de negócio da aplicação, mas as alternativas não são aceitáveis.

Alternativas consideradas

Desenvolver um componente pela equipa? Estima-se que o seu desenvolvimento seja (i) demasiado demorado, (ii) caro de desenvolver e (iii) de manter e (iv) gere uma responsabilidade desinteressante para os desenvolvedores (não têm esta competência).

Assuntos pendentes

Quais os melhores produtos e as suas qualidades?

Design de Arquitetura

Vistas Arquiteturais no SAD

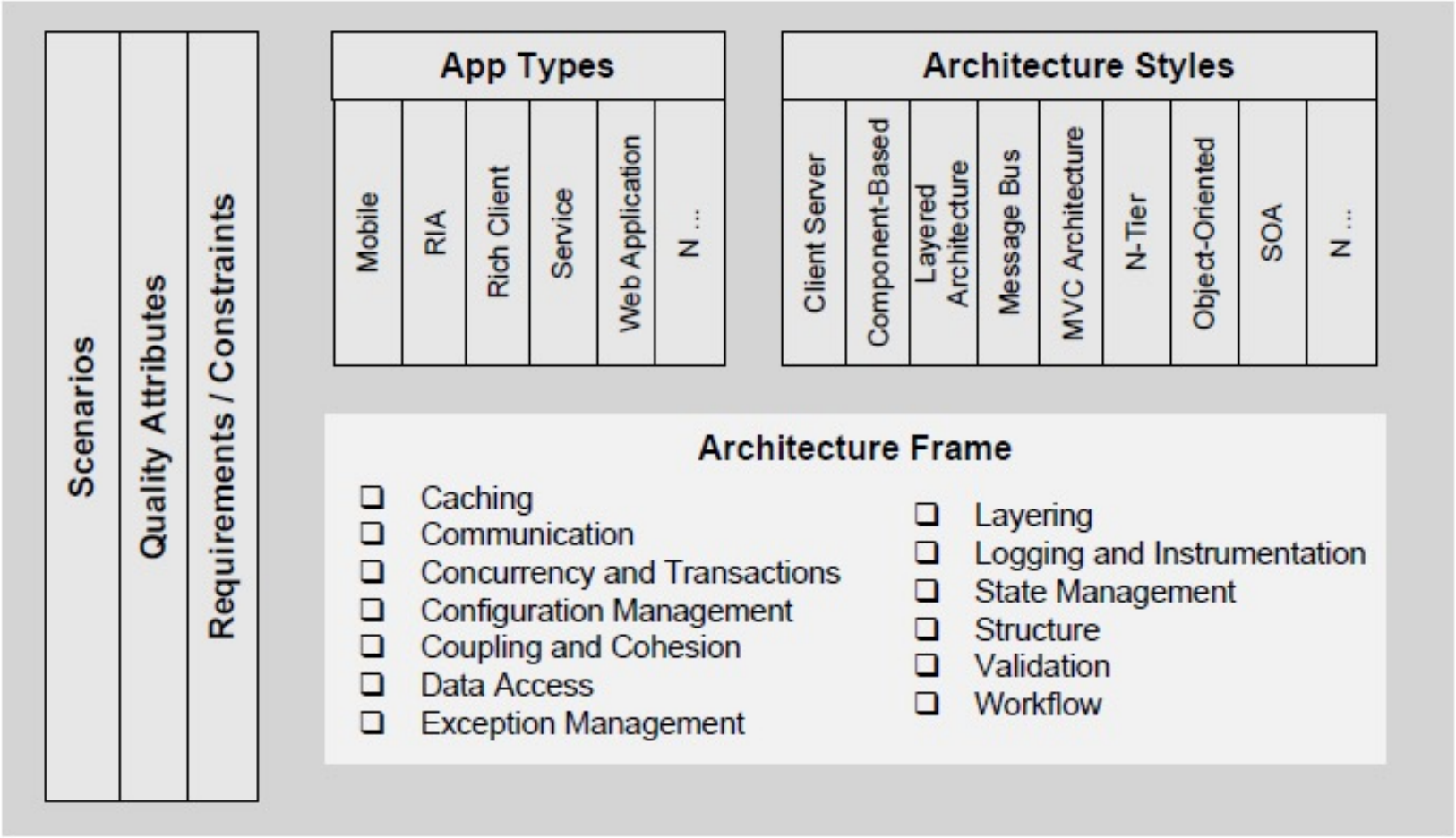
- SAD: Software Architecture Document
  - Semelhante ao documento em construção durante a UC
- Uma arquitetura de sistema pode ser analisada/descrita de várias:
  - perspectivas/vistas: cada vista é uma ferramenta de comunicação e de pensamento sobre o sistema/aplicação/módulo/...
    - Casos de Uso (aka Cenários)
    - Lógica
    - Processo
    - Implementação
    - Implantação (aka Física)
    - Dados
  - granularidades: nível de abstração com que o sistema/aplicação/módulo/... é analisado/descrito

Vistas Arquiteturais

- Casos de uso (aka cenários)
  - casos de uso mais significativos
  - seus requisitos não funcionais
- Lógica:
  - organização em larga escala das classes (e interfaces) de software
  - “lógica” porque não há uma relação com a implantação (deployment) por sistema operativo ou nós físicos (computadores), nem necessariamente com a implementação que se fará
- Processo:
  - processos e threads (que executam as classes)
  - alocação dos elementos lógicos
- Implementação
  - Entregáveis (“Deliverables”) mais significativos
  - Código que lhes dá origem
- Implantação (Deployment) (aka Física)
  - distribuição física dos artefatos de software pelos nós físicos
  - configuração de rede entre os nós
- Dados
  - esquema de dados para persistência
  - mapeamento do esquema de dados para esquema de classes

Exemplo de Documento de Arquitectura de Sistema

(exemplo de documento de arquitectura de sistema)


Scenarios
Quality Attributes
Requirements / Constraints
App Types
Mobile
RIA
Rich Client
Service
Web Application
N ...
Architecture Styles
Client Server
Component-Based
Layered Architecture
Message Bus
MVC Architecture
N-Tier
Object-Oriented
SOA
N ...
Architecture Frame
Caching
Communication
Concurrency and Transactions
Configuration Management
Coupling and Cohesion
Data Access
Exception Management
Layering
Logging and Instrumentation
State Management
Structure
Validation
Workflow

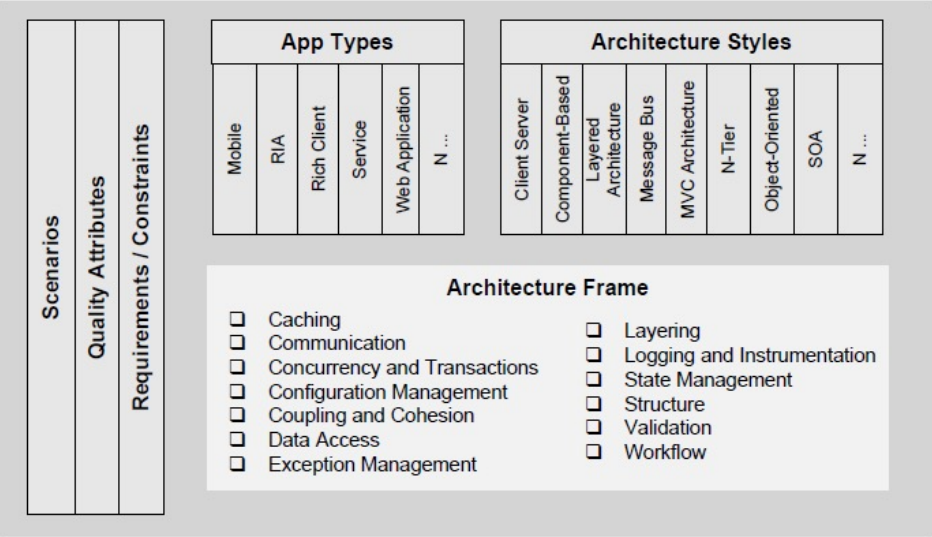
Como aplicar isto?

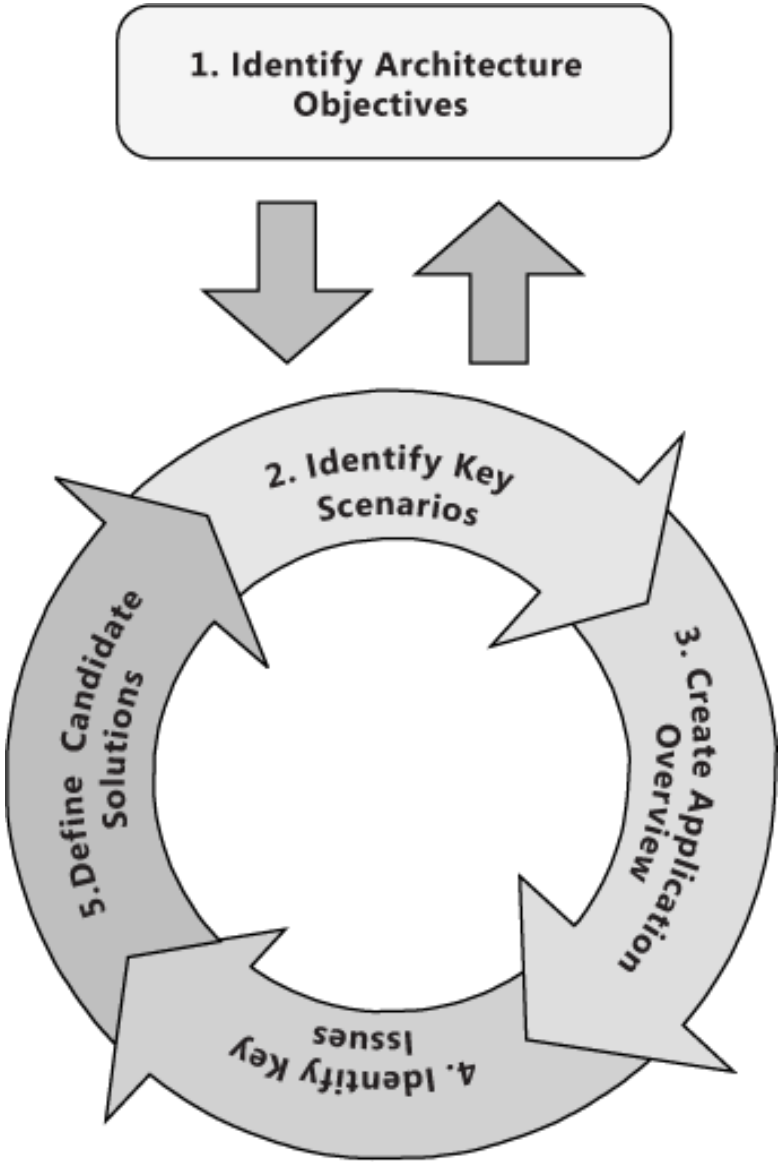
Método/Processo

segundo Microsoft

Method/Process

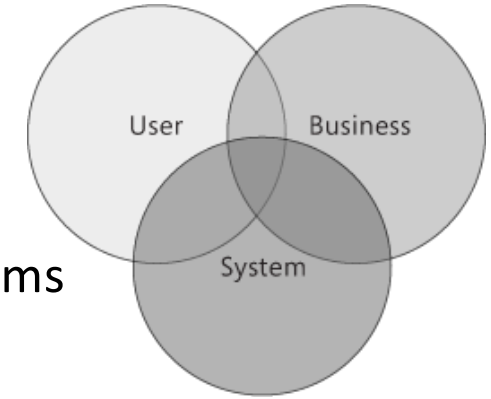
- Iterative and Incremental
- Vertical Iterations (end-to-end)





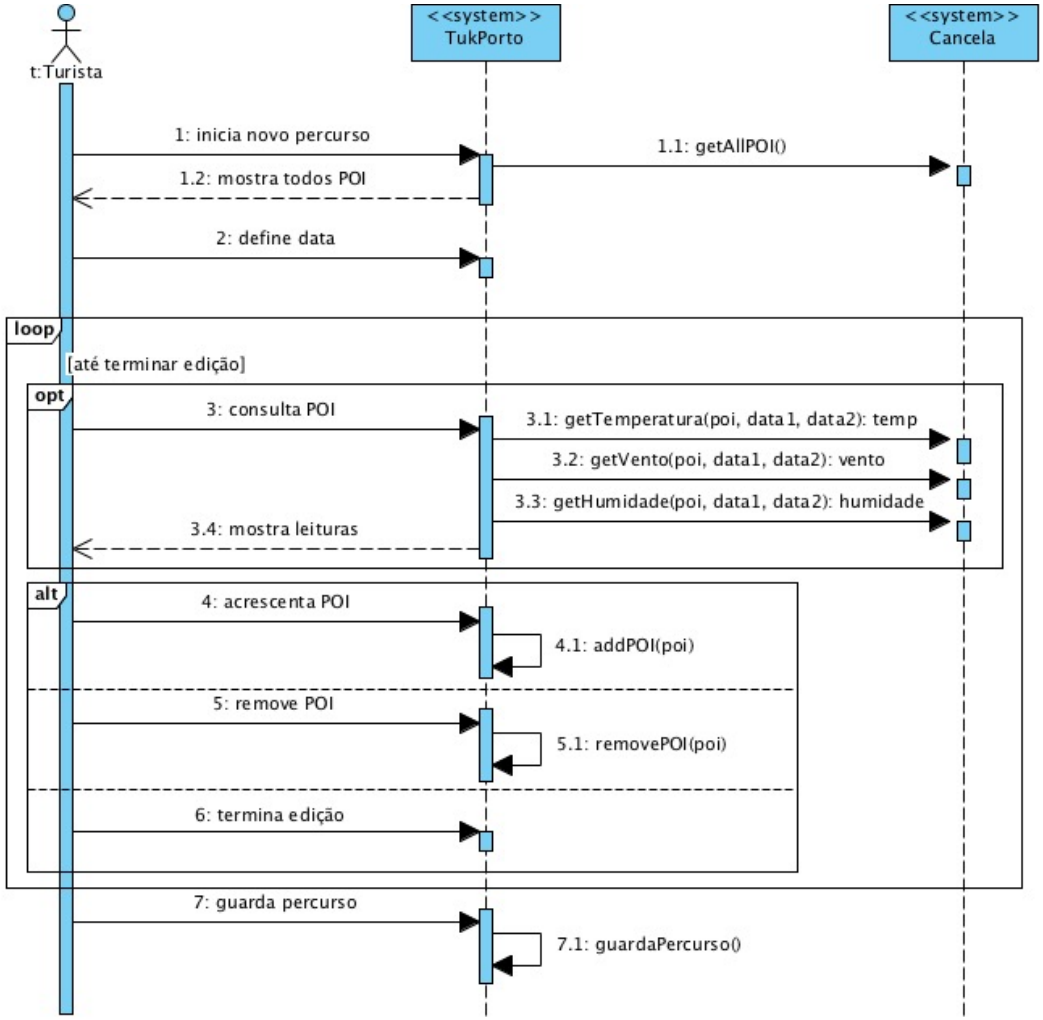
Identify Architecture Objectives

- Identify Objectives, e.g.:
  - Design a complete system architecture
  - Explore approaches/solutions
  - Identify technical risks
  - …
- Identify the consumers/users, e.g.:
  - Other software architects
  - Design/implementation/tests/deployment/management teams
  - …
- Identify constraints, e.g.:
  - User, business, systems



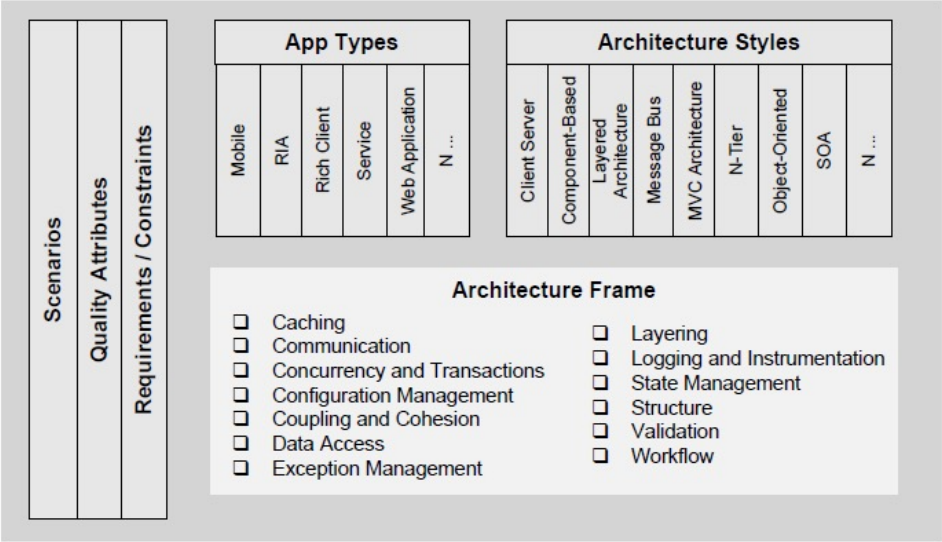
Key Scenarios

- Use cases architecturally significant
  - Broader scope than use cases
  - Business critical
  - Strong impact
  - User/business/system intersection
  - end-to-end
- Identify early
- Use for V&V (testes)
  - Verificação: de forma correta
  - Validação: o correto/desejado



Application Overview

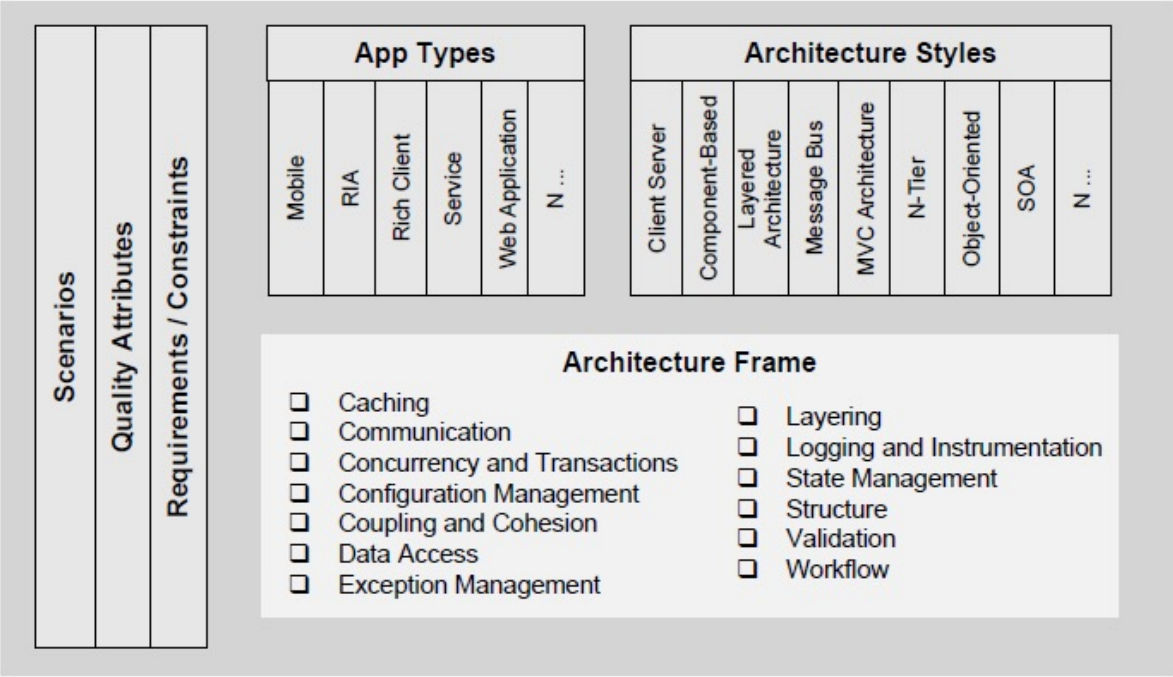
- Determine the type of application
  - Stand alone, RIA, SPA, Web Application, Services
- Determine deployment constraints
  - 1-Tier, 2-Tier, … n-Tier
  - in permisses, cloud (private, public, hybrid)
- Determine the architectural style
  - Client/Server, DDD, CBA, SOA, OO, …
- Identify relevant technologies
- Model/capture the architecture
  - E.g. 4+1 model, C4 model



Architecture style	Description
Client/Server	Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.
Domain Driven Design	An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message Bus	An architecture style that prescribes use of a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other.
N-Tier / 3-Tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.

Key Issues

- "Quality attributes"
  - Cf. non-functional requirements
  - Qualities of the system
- Cross-cutting concerns e.g.:
  - A&A
  - protocols
  - …
- Mitigation of risks
  - Evolution
  - Maintenance
  - Configuration



Candidate Solutions

- Baseline architecture
  - The existing architecture
- Candidate architecture (alternatives)
  - Is it better, without introducing new risks
  - Mitigates more risks?
  - Supports more constraints/requirements?
  - Supports (more) architectural scenarios?
  - Supports (more) quality attributes?
  - Supports (more) cross-cutting concerns?
- Prototypes and Spikes

http://www.scaledagileframework.com/spikes/

Key architectural principles

Manutenibiliadade → Evolutionary Architectures

- Build to change instead of building to last
- Model to analyze and reduce risk
- Using models to communicate and collaborate
- Identify and describe architectural alternatives and decisions

Bibliography/Source

- Software Architecture and Design (https://msdn.microsoft.com/en-us/library/ee658093.aspx)
- Chapter 4: A Technique for Architecture and Design (https://msdn.microsoft.com/en-us/library/ee658084.aspx)
- Building Evolutionary Architectures; by Patrick Kua, Rebecca Parsons, Neal Ford; O'Reilly Media, Inc.; Outubro *2017.*
- https://www.infoq.com/articles/why-architectural-diagrams?utm_medium=email&utm_source=topic+optin&utm_campaign=awareness&utm_content=20190126+prog+nl&mkt_tok=eyJpIjoiT1Rsa04yVmxNR1V4TWpZeiIsInQiOiJiVFwvMFlKNnhxdmZNZXlVTENjRUNXRks2QndmU1RUbUk3Z3MwclwvU1Raekc0bk5uWk9cLzE0VjdsSm1EVk9vaGRYUjJ1ODNDYmhlMjk2ZHRqTzVjNEZpVTRSWVJDVHlXV2FEYXIyRWdYeWJ6WWxrbnEzRVNYVWJRNTlJV2R3VHk5YyJ9
- https://www.infoq.com/articles/crafting-architectural-diagrams
- Rational Unified Process: http://www.ts.mah.se/RUP/RationalUnifiedProcess/index.htm
- http://www.dcs.bbk.ac.uk/~niki/SoftwareEngineering.htm
- http://www.cse.lehigh.edu/~glennb/oose/oose.htm
- Appling UML and Patterns: An Introduction To Object-Oriented Analysis And Design And Iterative Development; Craig Larman; Pearson, 2004.

Something about Authentication and Authorization

Identity and access management (IAM or IdAM)

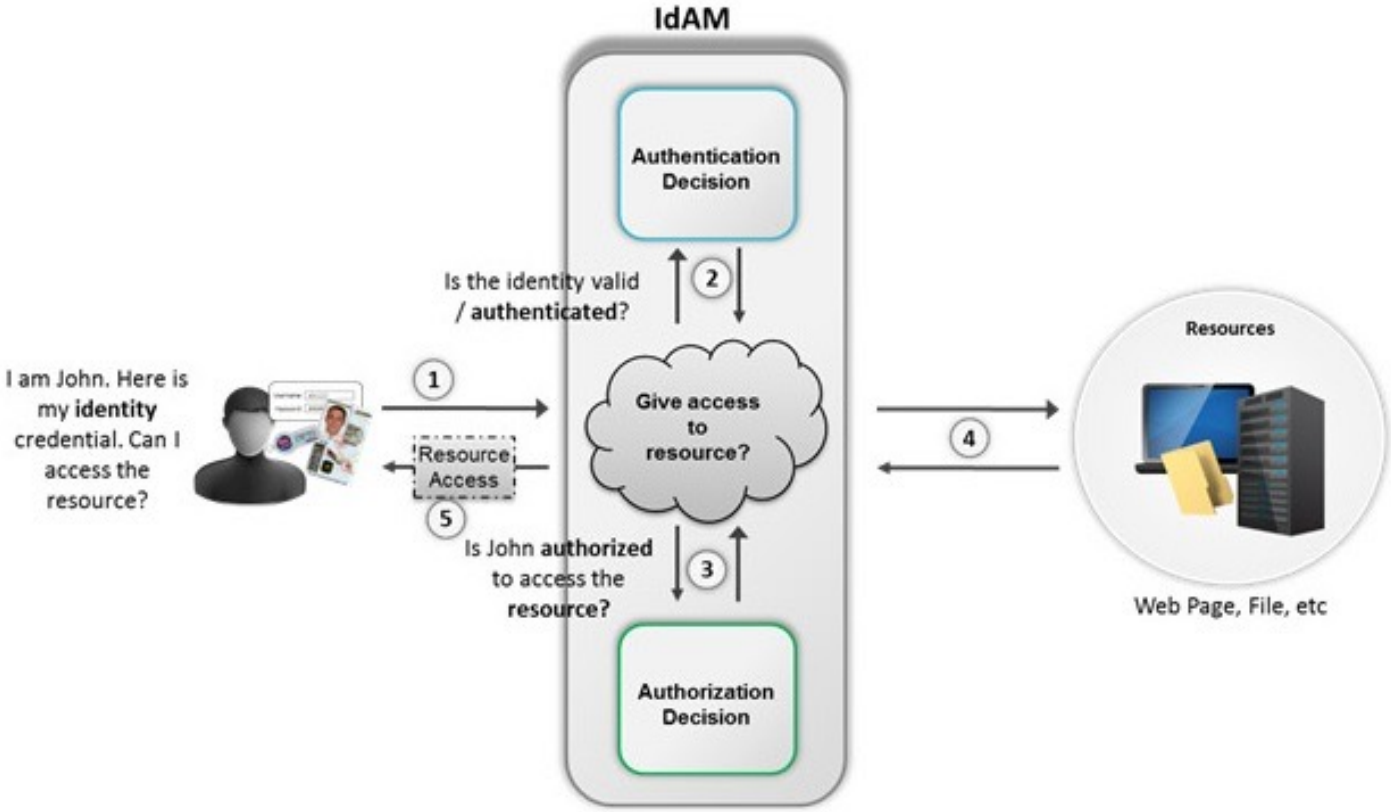
- Identity and Access Management (IdAM) is the combination of technical systems, policies and processes that create, define, and govern the utilization, and safeguarding of identity information, as well as managing the relationship between an entity, and the resources to which access is needed.
- IAM may be a single product, or it may be a combination of processes, software products, cloud services, and hardware that give administrators visibility and control over the organizational data that individual users can access.

https://public.cyber.mil/idam/idam-in-a-nutshell/
https://www.cloudflare.com/en-gb/learning/access-management/what-is-identity-and-access-management/

IdAM capabilities

- Manage Digital Identities
  - Managing digital identities provides the security and accountability for creating, defining, and trusting digital identity data. This includes the processes for uniquely identifying a user, securely binding a digital identity to a user, and managing digital identity data for use across the enterprise. Digital identity data includes user contact and organizational information and user attributes for access control. Accurate and consistent digital identity data provides the foundation for securing access to IT resources.
- Authenticate Users
  - Authenticate users provides the verification of the the digital identity of a user. It is the digital equivalent of proving who you are. This is accomplished by submitting and validating credentials as proof of identity. Credentials typically include something you have, something you know, and/or something you are. Different types of credentials used in the DoD include a username and password, PKI, and biometrics. Authentication provides resources a level of assurance that the users are who they claim to be.
- Authorize Access to Resources
  - Authorize access to resources enables an authority to restrict access to resources locally or in the enterprise based on the evaluation of applicable policies. Controlling resource access is paramount to protecting private and confidential information from unauthorized users. Access can be determined by the establishment of user accounts and user roles with access permissions or through user attributes and access control policies. The authorization process typically involves a request to access a resource and an access control decision based on the security context of the transaction. The access control decision returned from this evaluation determines whether the user attempting to interact with the resource complies with the policies and governance requirements tied to its access.

https://public.cyber.mil/idam/idam-in-a-nutshell/



https://public.cyber.mil/idam/idam-in-a-nutshell/

Digital Identitiy

What is identity in the context of computing?

- A person's entire identity cannot be uploaded and stored in a computer, so "identity" in a computing context means a certain set of properties that can be conveniently measured and recorded digitally.
  - Think of an ID card or a passport: not every fact about a person is recorded in an ID card, but it contains enough personal characteristics that a person's identity can quickly be matched to the ID card.
- To verify identity, a computer system will assess a user for characteristics that are specific to them. If they match, the user's identity is confirmed. These characteristics are also known as "authentication factors," because they help authenticate that a user is who they say they are.

https://www.cloudflare.com/en-gb/learning/access-management/what-is-identity-and-access-management/

Digital Identity, or Authentication factors

- Digital user identity is associated with quantifiable factors that can be verified by a computer system.
- Something the user knows
  - This factor is a piece of knowledge that only one user should have, like a username and password combination. E.g. password
- Something the user has
  - This factor refers to possession of a physical token that is issued to authorized users. E.g. house key, USB device, phone (to receive a token)
- Something the user is (biometrics)
  - This refers to a physical property of one's body. E.g. face, fingerprint, retina
- Using multiple factors to verify user identify is called multi-factor authentication (MFA).

https://www.cloudflare.com/en-gb/learning/access-management/what-is-an-identity-provider/

Authentication

Authn

Authentication

- Authentication is the process of verifying identity.
- Authentication means making sure that a person or device is who (or what) they claim to be.
- Authentication ensures that data is not exposed to the wrong person.

Common authentication methods?

- Username and password combination
  - One of the most common methods for authentication is prompting a user to enter their username and password. The service is able to check those credentials, authenticate, and log the user in to her/his account.
  - While most people are familiar with this type of authentication, usernames and passwords can be used for more than just authenticating users. API endpoints can be authenticated in this fashion, for example.
- Multi-factor authentication (MFA)
  - The problem with username-password authentication is that passwords can often be guessed or stolen by malicious parties. Requiring additional factors of authentication increases security for users; this concept is called multi-factor authentication (MFA). When MFA is used, an attacker needs more than a password to falsely authenticate as a legitimate user.
  - MFA is most often implemented as two-factor authentication (2FA). Today many services implement 2FA by asking users to prove they have a token they were issued. There are two types of tokens: "soft" tokens, like a code sent to a user via SMS or through a mobile app, and "hard" tokens, like USB keys. 2FA and MFA can also use biometric authentication factors (described below).

Common authentication methods? (cont.)

- Public key certificate
  - Public key authentication is slightly more complex than these other forms of authentication, but when implemented properly, it can be more secure. It uses public key encryption to verify whether or not the authenticated party has the right private key.
  - The most common usage of public key authentication is in Transport Layer Security (TLS), in which it is used to authenticate a web server. User devices perform this type of authentication every time they load a website that uses HTTPS.
  - Public key authentication is also used for mutual authentication, which is when both sides of a communication authenticate each other, instead of just a client authenticating a server or a web service authenticating a user.
- Biometric authentication
  - Only usable for authenticating humans, biometric authentication involves verifying someone's identity by checking one of their physical characteristics against a database of their known physical characteristics. Face scanning or a retina scan are examples of this type of authentication.

Authorization

Authz

Access Management

- "Access" refers to what data a user can see and what actions they can perform once they log in.
  - Once John logs into his email application, he can see all the emails he has sent and received. However, he should not be able to see the emails sent and received by Tracy, his coworker.
  - In other words, just because a user's identity is verified, that doesn't mean they should be able to access whatever they want within a system or a network. For instance, a low-level employee within a company should be able to access their corporate email account, but they should not be able to access payroll records or confidential HR information.
- Access management is the process of controlling and tracking access. Each user within a system will have different privileges within that system based on their individual needs. An accountant does indeed need to access and edit payroll records, so once they verify their identity, they should be able to view and update those records as well as access their email account.

https://www.cloudflare.com/en-gb/learning/access-management/what-is-identity-and-access-management/

Authorization (Authz)

- Authorization is the process of giving the user permission to access a specific resource or function.
- Authorization determines what an authenticated user can see and do.
  - Think of what happens when a bank customer logs in to their account online. Because their identity has been authenticated, they can see their own account balance and transaction history — but they are not authorized to see anyone else's. A manager at the bank, conversely, could be authorized to see any customer's financial data.
  - Similarly, a person may be a legitimate employee of a business, and they may have verified their identity, but that does not mean they should have access to all of that business's files and data. An employee from outside the HR or accounting departments should not be able to see everyone's compensation, for instance.
- A user's authorization level determines what they have permission to do; therefore, a common term for authorized actions is "permissions." Another term for this concept is "privileges."

OAuth

https://learn.microsoft.com/en-us/azure/active-directory/develop/authentication-vs-authorization

OAuth

- OAuth is a technical standard for passing authorization from one service to another [without exposing your password].
- Often used for cloud services and web applications, OAuth enables users to authenticate on one service and then have their authorization passed to another service. Their authorization level is usually determined by an identity provider (IdP), which is a separate service.
- OAuth makes possible the use of single sign-on (SSO) services, with which a user can sign in once in order to access all their cloud applications.
  - Without the use of OAuth (or similar system), a user's permissions would have to be set up separately in each application.

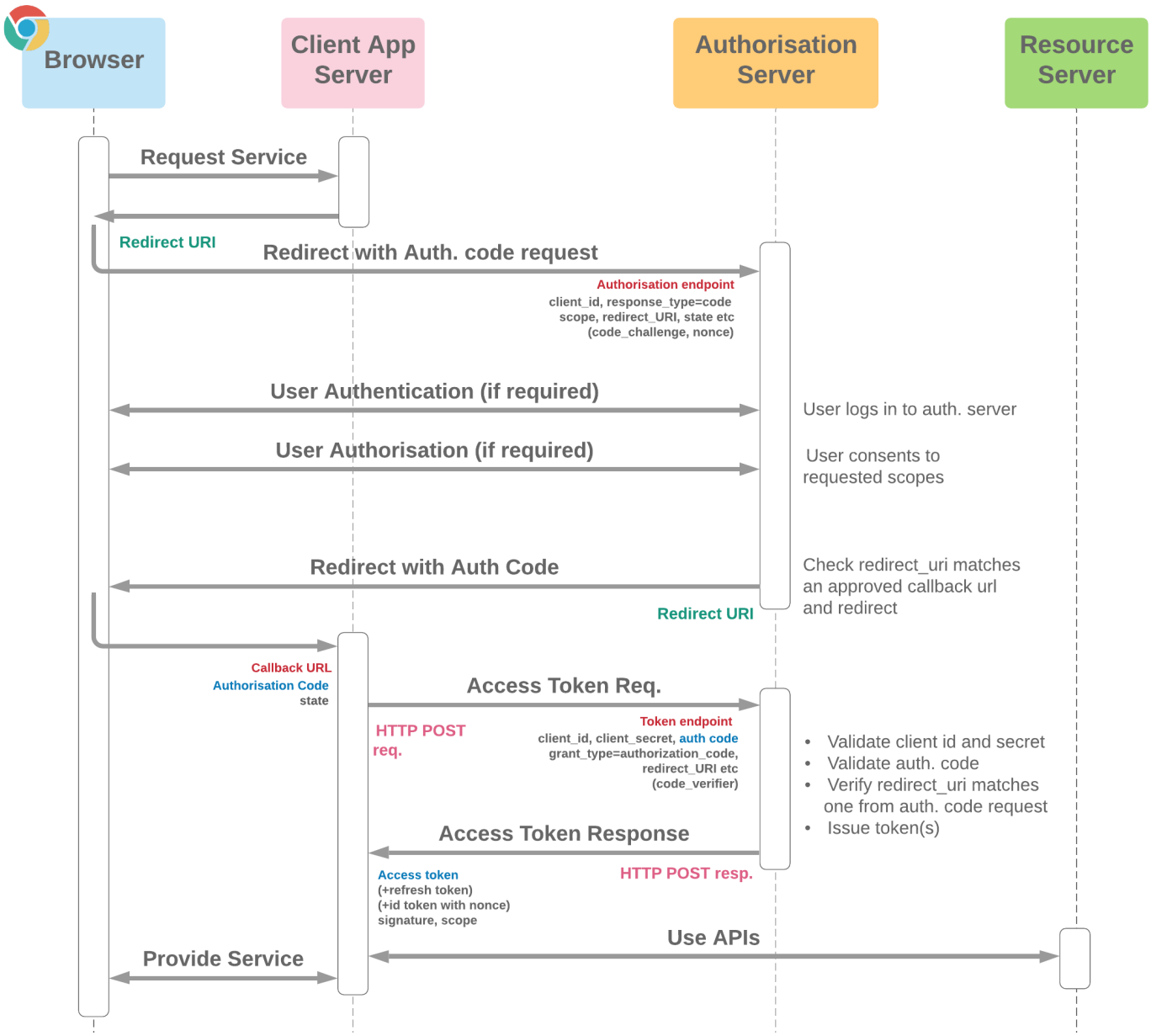
https://www.cloudflare.com/en-gb/learning/access-management/authn-vs-authz/

The Simplest Guide To OAuth 2.0

- https://darutk.medium.com/the-simplest-guide-to-oauth-2-0-8c71bd9a15bb
- Step-by-step
  - Context
  - Challenge
  - Problem
  - Solution
- Illustrated
- 10 minutes reading

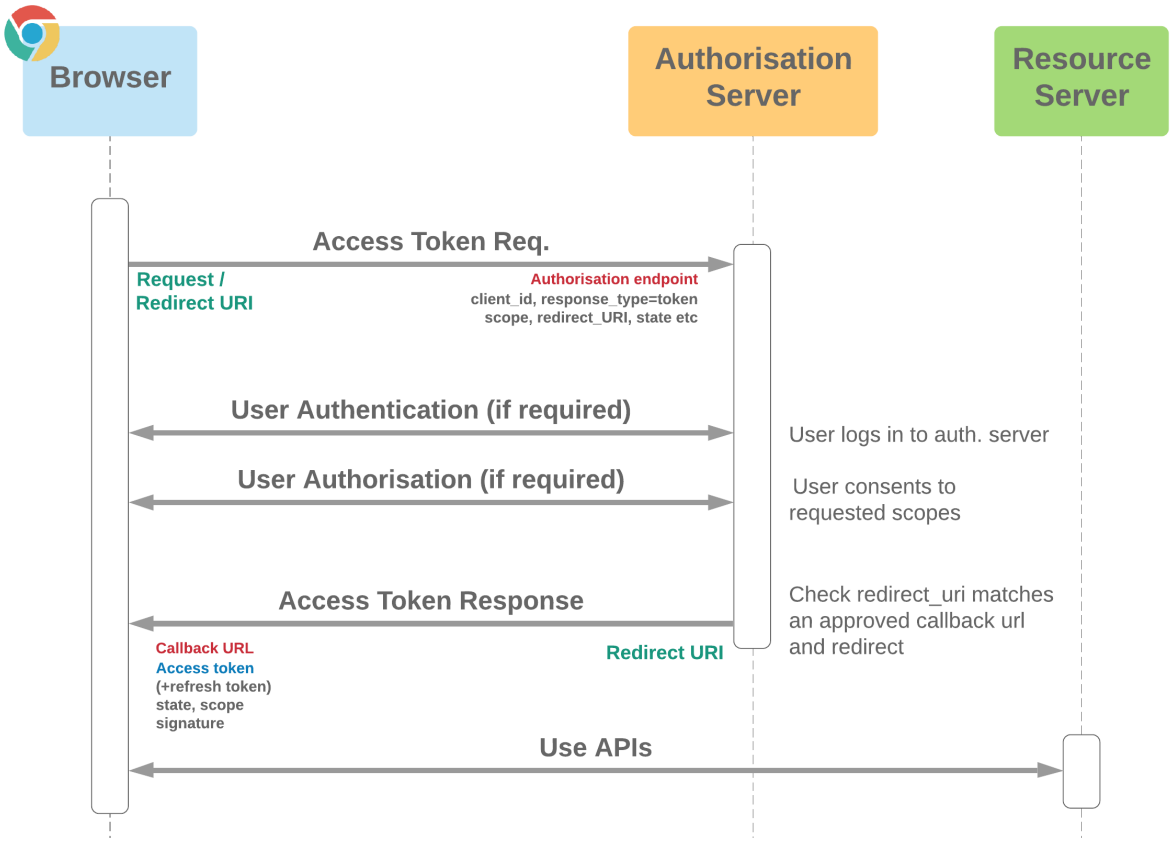
OAuth flows

- OAuth flows used for either user or server authorisation and authentication
- Authorization code with secret (Web Server)
- Implicit grant (User Agent)
- Authorization code with PKCE
- JWT Bearer Flow



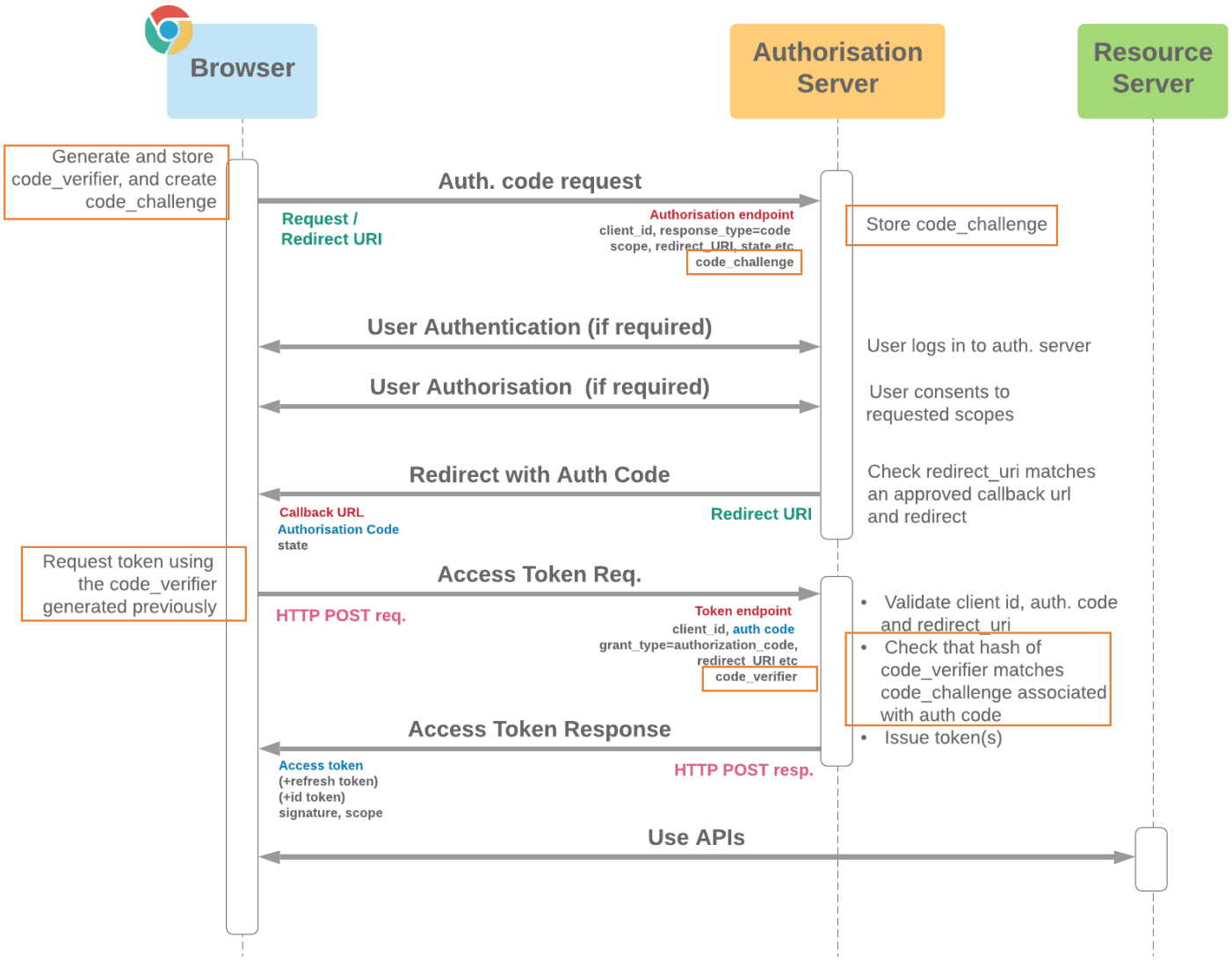
AUTHORISATION CODE WITH SECRET (WEB SERVER)

- This flow provides the highest level of security possible through standard OAuth flows, but is suitable only if the client secret can be protected by the client application - generally this is the case when the application has a backend server component which is operating in a trusted environment (e.g. an internal enterprise server application)
- The current IETF OAuth working group recommendation is for public and confidential clients participating in any auth. code flow to implement PKCE using code_challenge and code_verifier (more info on how these are used in the Authorisation Code with PKCE flow below)
- As an alternative to client secret, a JWT with issuer & subject = client_id signed with the client app's key can be provided in the access token request. For this variation, client_assertion and client_assertion_type should be set instead of client_secret
- Cf. https://cloudsundial.com/salesforce-oauth-flows



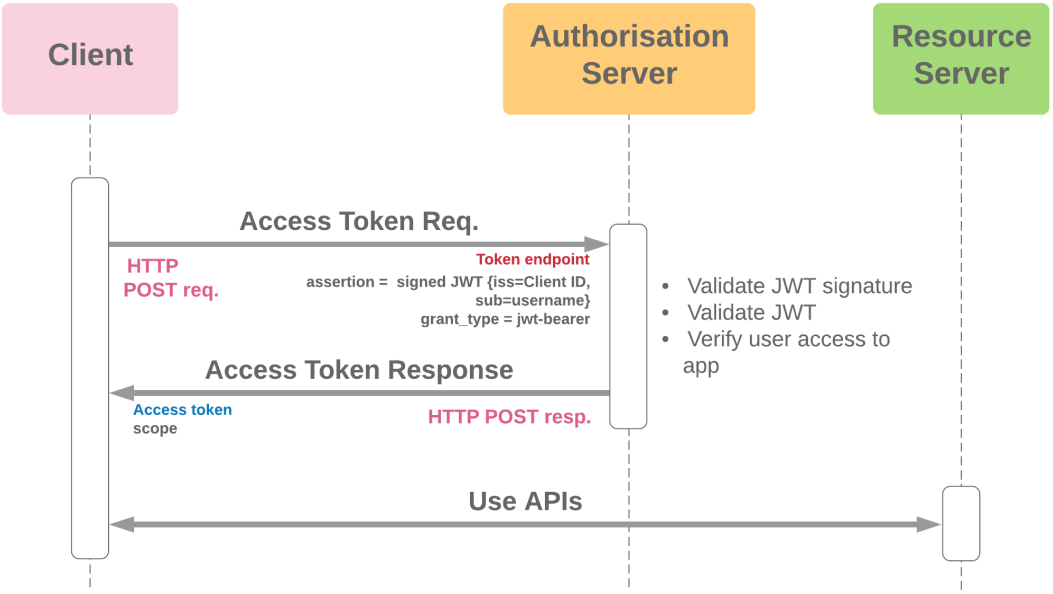
IMPLICIT GRANT (USER-AGENT)

- The flow can be used in circumstances where a client secret can't be protected, for example in a mobile or desktop application, or client app running in a browser, such as a SPA.
- Identity is based on the “same-origin policy” (redirect_uri must match one of the Callback URL values in connected app)
- The flow is sub-optimal from a security perspective in many scenarios, as tokens are transmitted in URL. This means they are accessible to the browser and stored in browser history, which is frequently synced across devices and susceptible to attack. It's advisable to ensure that access token validity is short and refresh tokens aren't used to reduce the risk of such attacks
- Prior to 2017, this flow was recommended as best practice for mobile and native applications, however the current recommendation is to implement the more secure Authorisation Code with PKCE flow described below in contexts where an out of band POST can be made
- Cf. https://cloudsundial.com/salesforce-oauth-flows



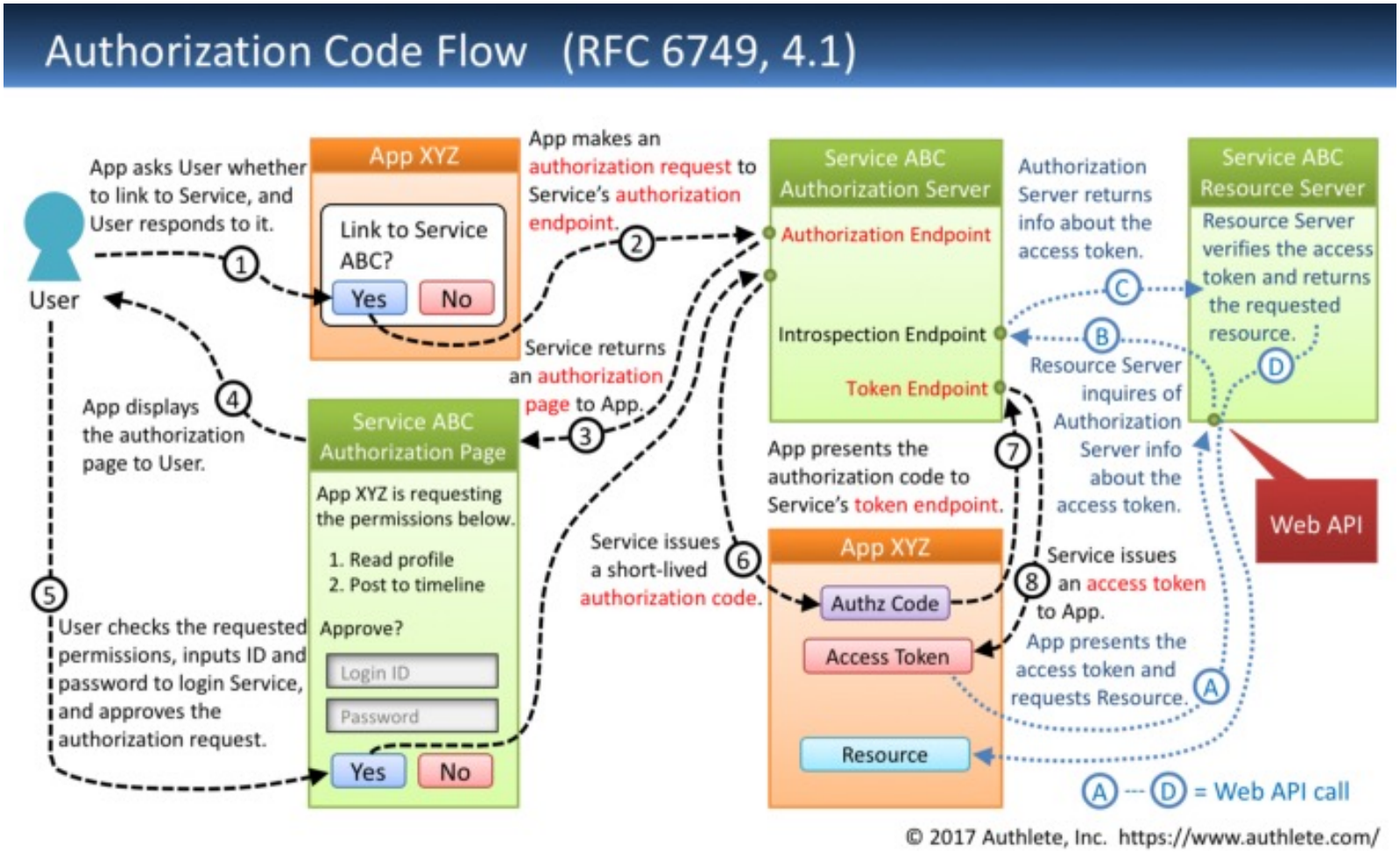
AUTHORISATION CODE WITH PKCE

- Since 2017, Authorisation Code with PKCE (Proof Key for Code Exchange) has been recommended as the most secure flow for mobile and native apps by the Internet Engineering Task Force (IETF).
- PKCE refers to the use of "code verifier" and "code challenge", which provide an additional layer of security. Code verifier is a 128 byte base64url encoded random string generated by the app, and code challenge is the base64url encoded SHA256 hash of this string.
- The authorisation server checks the application is making an authorised token request by comparing code challenge to the hash of code verifier. When these match, the authorisation server can be confident the entity making the token request is the same one which requested the authorisation code - the one-way nature of the hash function makes it difficult for an attacker to find a matching code verifier and so impersonate the authorised client even if code challenge and auth code are intercepted.
- Cf. https://cloudsundial.com/salesforce-oauth-flows

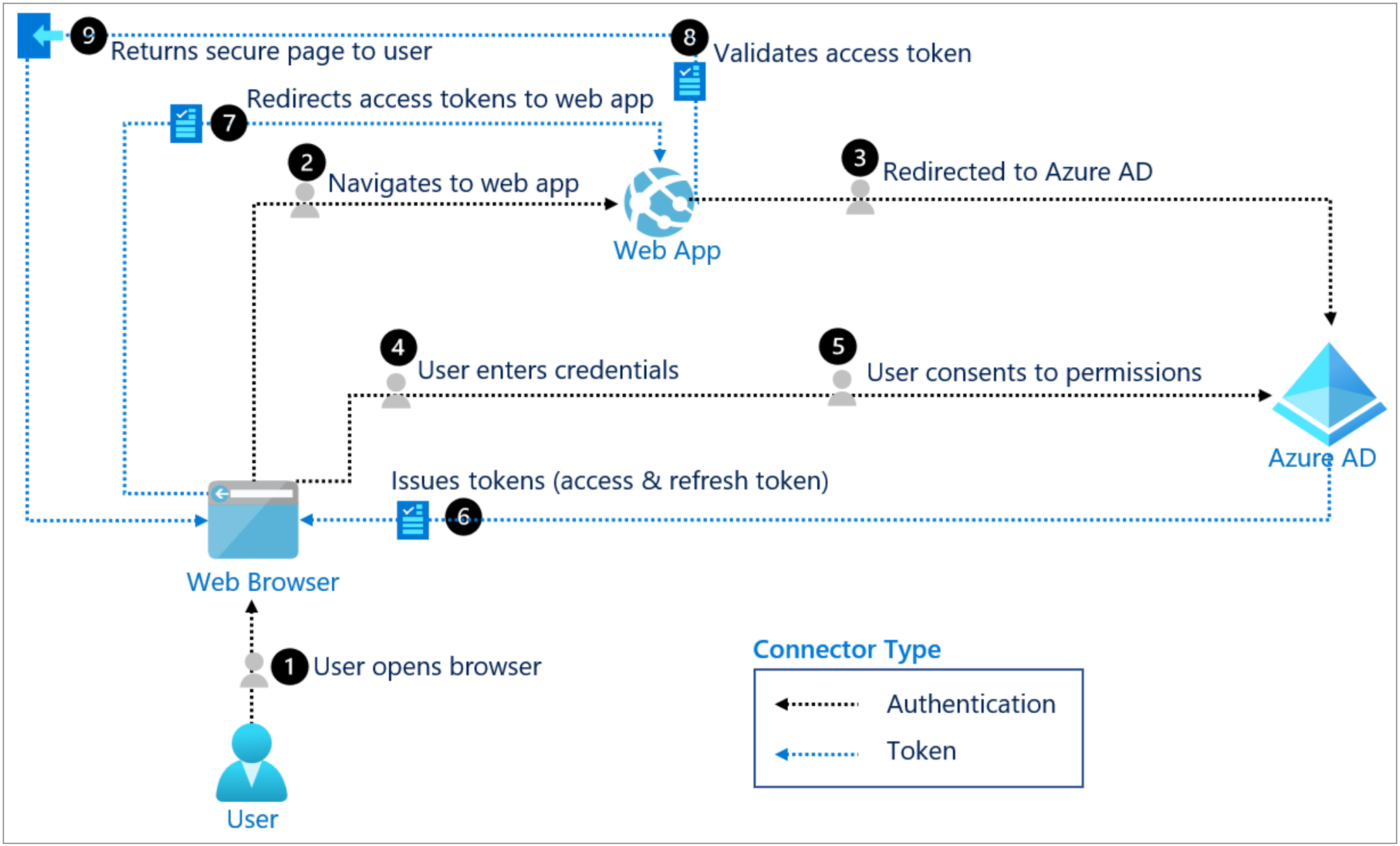


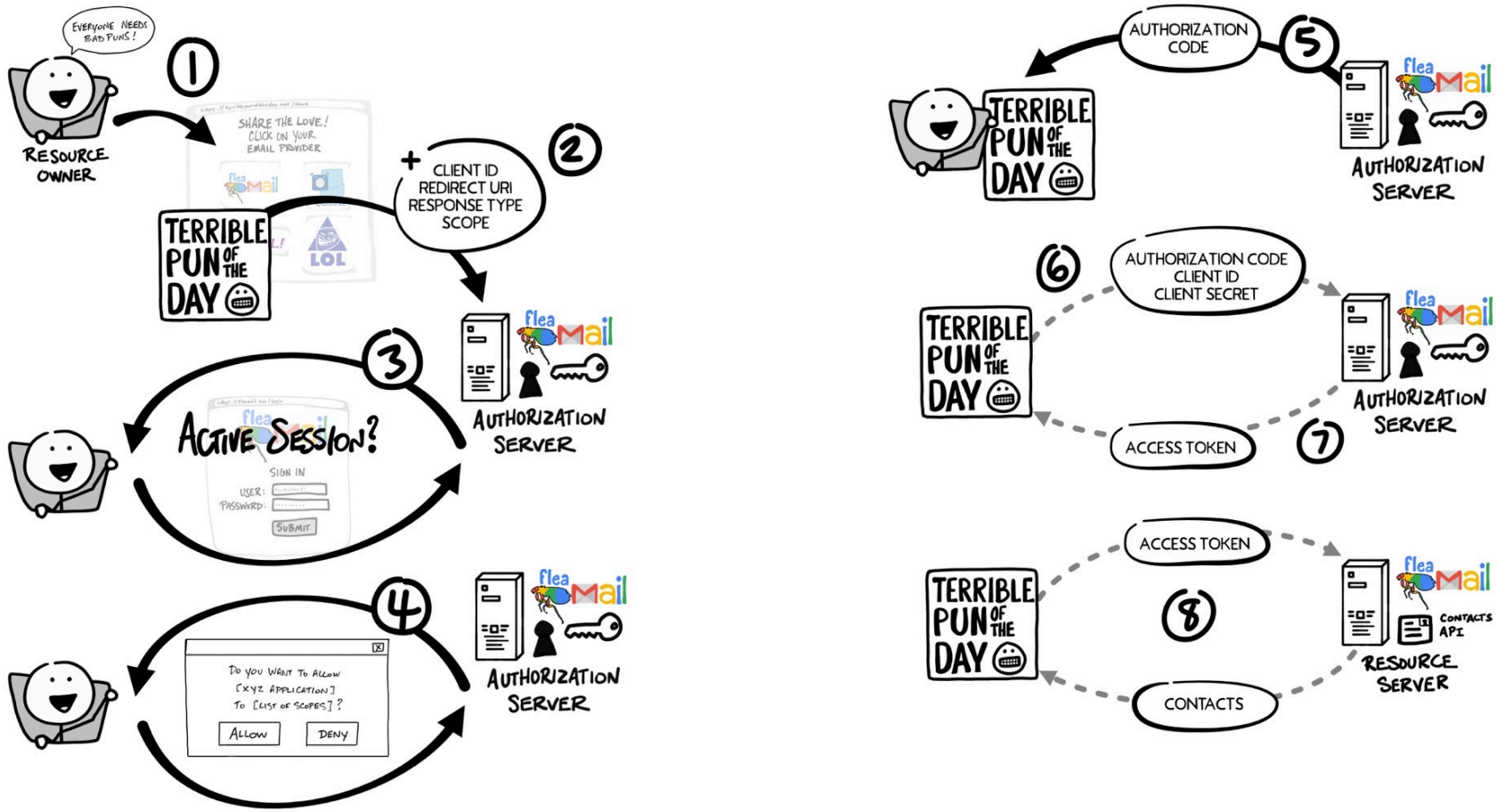
JWT BEARER FLOW

- JSON Web Token (JWT) is created and signed with private key of an X509 certificate by client app.
- Authorisation server checks signature against public key of certificate to verify authenticity of request.
- Allowed scopes are based on approval previously provided by the user through another OAuth flow.
- Additional claims can be included in the JWT to communicate extra user or context info to the authorization server.
- Cf. https://cloudsundial.com/salesforce-oauth-flows



https://darutk.medium.com/diagrams-and-movies-of-all-the-oauth-2-0-flows-194f3c3ade85


9 Returns secure page to user
8 Validates access token
Redirects access tokens to web app
7
2 Navigates to web app
3 Redirected to Azure AD
Web App
4 User enters credentials
5 User consents to permissions
Azure AD
Issues tokens (access & refresh token)
6
Web Browser
1 User opens browser
User
Connector Type
Authentication
Token

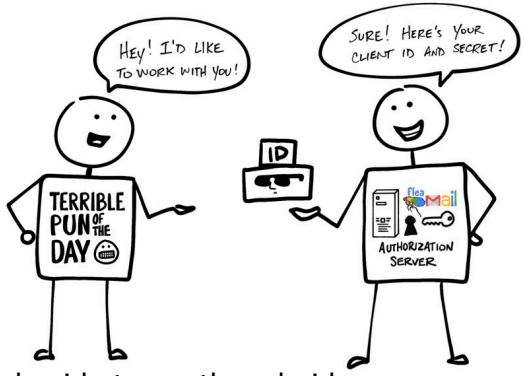


As far as the Client is concerned, the Access Token is just a string of gibberish to pass with any request to the Resource Server, and the Resource Server knows if the token is valid.

https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc

Client ID and Secret

- The Client and the Authorization Server must agree on Client ID and Client Secret (aka App ID and App Secret).
- The Client uses the Client ID and Client Secret for all future OAuth exchanges.
- The Client Secret must be kept secret so that only the Client and Authorization Server know what it is. This is how the Authorization Server can verify the Client.



https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc

OpenID Connect vs. OAuth

- OAuth is used for authorization
  - OAuth was created to remove the need for users to share their passwords with third-party applications.
- OpenID Connect (OIDC) is used for authentication
  - OIDC was created for federated authentication, meaning that it lets a third-party application authenticate users for you, using accounts that you already have.
  - OIDC enables scenarios where one login can be used across multiple applications, also known as single sign-on (SSO). For example, an application could support SSO with social networking services such as Facebook or Twitter so that users can choose to leverage a login they already have and are comfortable using.

https://www.securew2.com/blog/oauth-vs-openid-which-is-better

OpenID Connect vs. OAuth (cont.)

- A misinterpretation arises as both protocols can be used to accomplish similar tasks, but that doesn't mean they should be used interchangeably.
- OpenID provides an identity assertion while OAuth is more generic.
- When a client uses OAuth, a server issues an access token to a third party, the token is used to access a protected resource, and the source validates the token.
  - Notice, that at no point is the identity of the owner of the token verified.
- You can think of a token issued by a resource server like it's a ticket to a movie. Nowhere on the ticket does it say any identifying information about an individual, it simply is used as a way of saying I have permission to enter. This means that the issued token may be in the hands of a bad actor or a machine.

https://www.securew2.com/blog/oauth-vs-openid-which-is-better

OIDC with OAuth

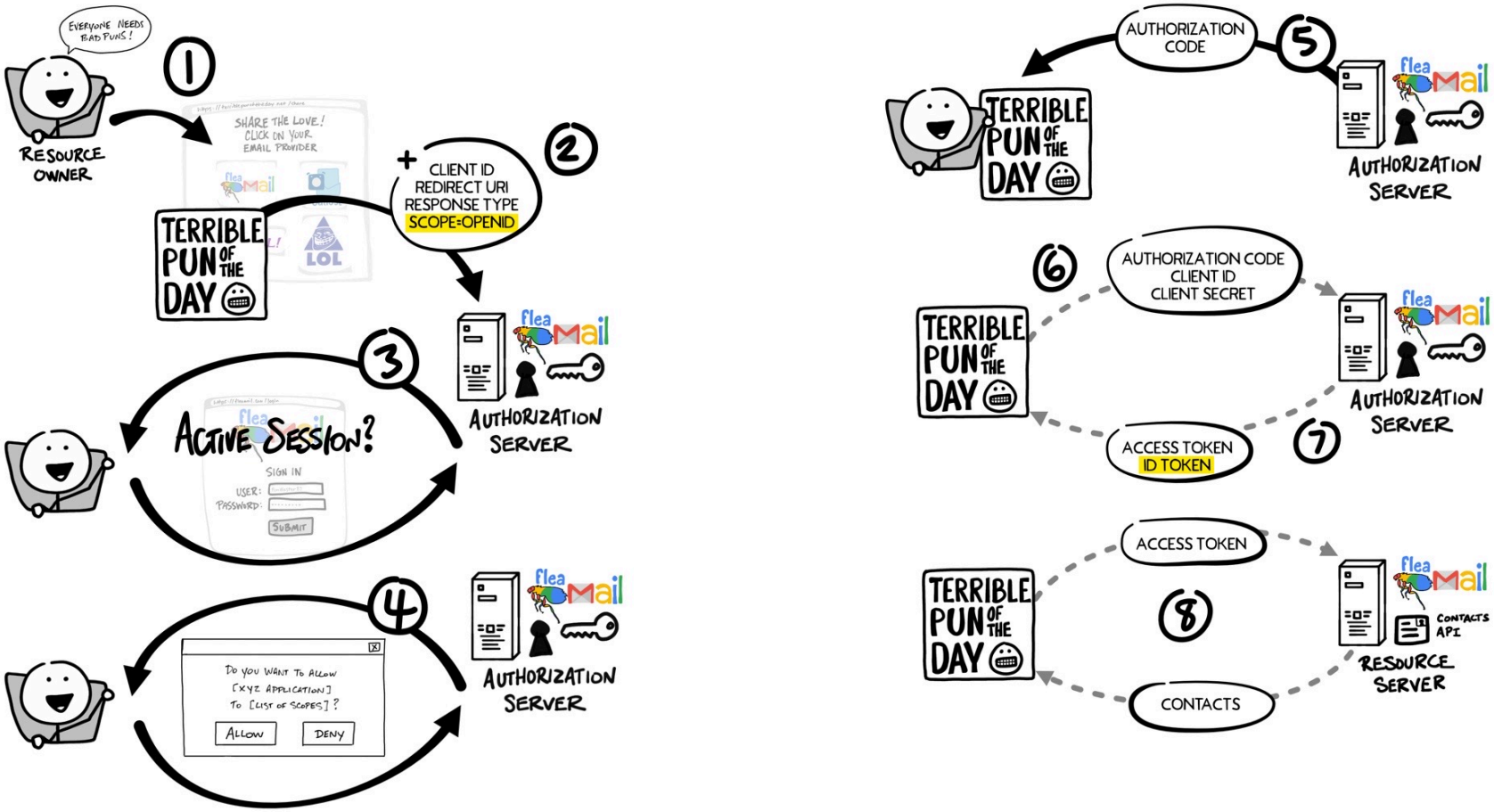
- OAuth 2.0 is designed only for authorization, for granting access to data and features from one application to another.
- OpenID Connect (OIDC) is a thin layer that sits on top of OAuth 2.0 that adds login and profile information about the person who is logged in. Establishing a login session is often referred to as authentication, and information about the person logged in (i.e. the Resource Owner) is called identity.
- OpenID Connect enables scenarios where one login can be used across multiple applications, also known as single sign-on (SSO).
  - For example, an application could support SSO with social networking services such as Facebook or Twitter so that users can choose to leverage a login they already have and are comfortable using.

https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc

Access tokens as proof of authentication

- In OAuth, the token is designed to be opaque to the client, but in the context of a user authentication, the client needs to be able to derive some information from the token.
- Since an authentication usually occurs ahead of the issuance of an access token, it is tempting to consider reception of an access token of any type proof that such an authentication has occurred. However, mere possession of an access token doesn't tell the client anything on its own.
  - This problem stems from the fact that the client is not the intended audience of the OAuth access token. Instead, it is the authorized presenter of that token, and the audience is in fact the protected resource. The protected resource is not generally going to be in a position to tell if the user is still present by the token alone, since by the very nature and design of the OAuth protocol the user will not be available on the connection between the client and protected resource.
- To counter this, there needs to be an artifact that is directed at the client itself.
  - Protocols like OpenID Connect's ID Token and Facebook Connect's Signed Response provide a secondary token along side the access token that communicates the authentication information directly to the client.
  - This allows the primary access token to remain opaque to the client, just like in regular OAuth.

https://oauth.net/articles/authentication/

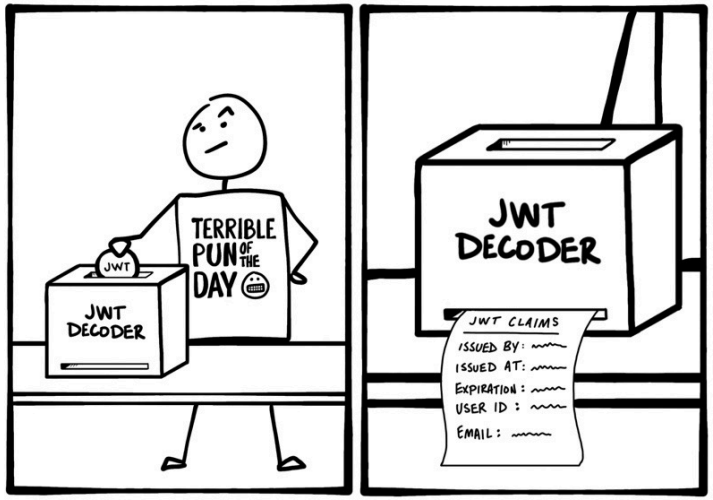


As with the OAuth flow, the OpenID Connect Access Token is a value the Client doesn't understand. The ID Token, however, is very different.

https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc

JWT as ID TOKEN

- An ID Token is a specifically formatted string of characters known as a JSON Web Token, or JWT. JWTs are sometimes pronounced “jots.”
- A JWT may look like gibberish, but the Client can extract information embedded in the JWT such as your ID, name, when you logged in, the ID Token expiration, and if anything has tried to tamper with the JWT. The data inside the ID Token are called claims.
- With OIDC, there’s also a standard way the Client can request additional identity information from the Authorization Server, such as their email address, using the Access Token.



https://developer.okta.com/blog/2019/10/21/illustrated-guide-to-oauth-and-oidc

Identity Provider (IdP, IDP)

- An identity provider (IdP) is a product or service that helps manage identity, i.e. stores and manages users' digital identities.
- An IdP often handles the actual login process.
- Single sign-on (SSO) providers fit into this category.
- IdPs can be part of an IAM framework, but typically they don't help with managing user access.
- When an Authorization Server supports OIDC, it is sometimes called an Identity Provider (IdP), since it provides information about the Resource Owner back to the Client.

Identity-as-a-Service (IDaaS)

- Identity-as-a-Service (IDaaS) is a cloud service that verifies identity.
- It is a SaaS offering from a cloud vendor, a way of partially outsourcing identity management.
- In some cases, IDaaS and IdP are essentially interchangeable – but in other cases, the IDaaS vendor offers additional capabilities on top of identity verification and management.
- Depending on the capabilities offered by the IDaaS vendor, IDaaS can be a part of an IAM framework, or it can be the whole IAM system.

IdP and IDaaS services/servers

- Auth0
- Okta
- KeyCloak
- OneLogin
- Asgardeo
- Azure Active Directory (AAD)
- Google Sign-in
- Facebook login

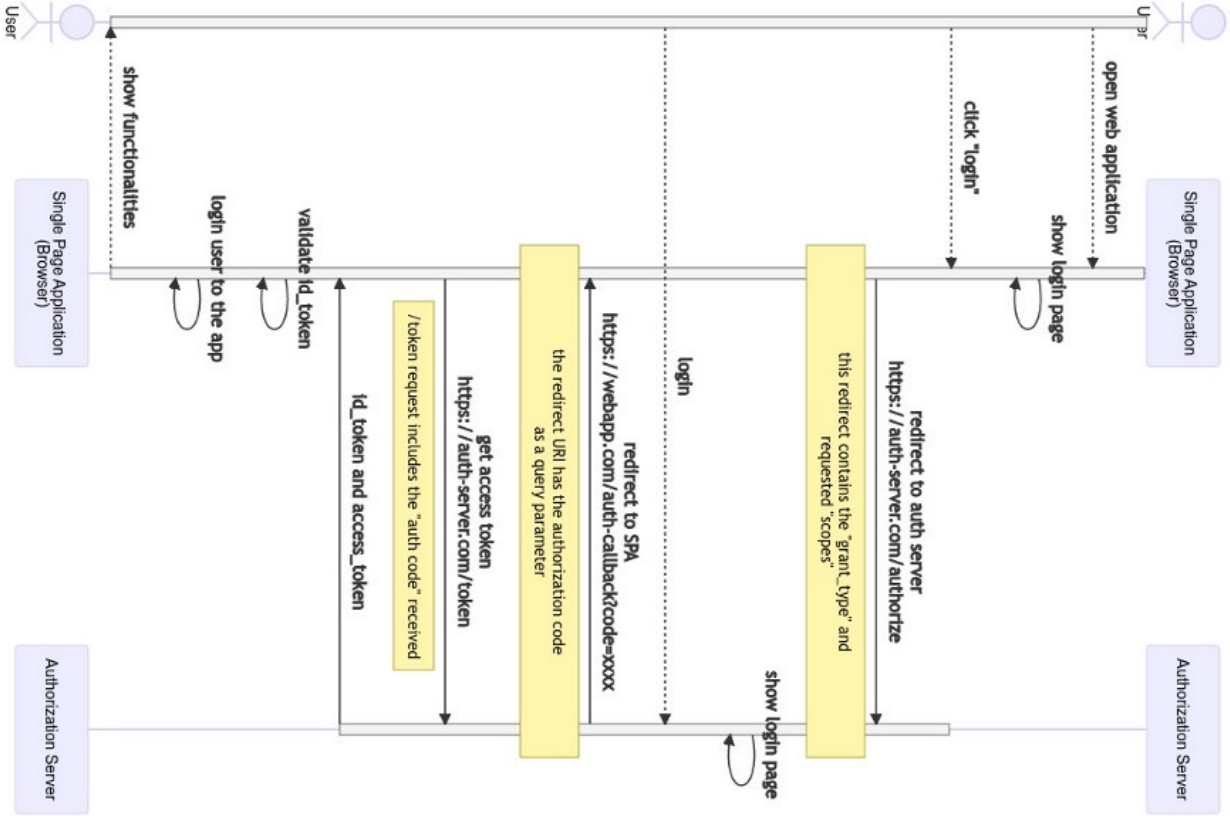
OIDC in SPA

Concepts

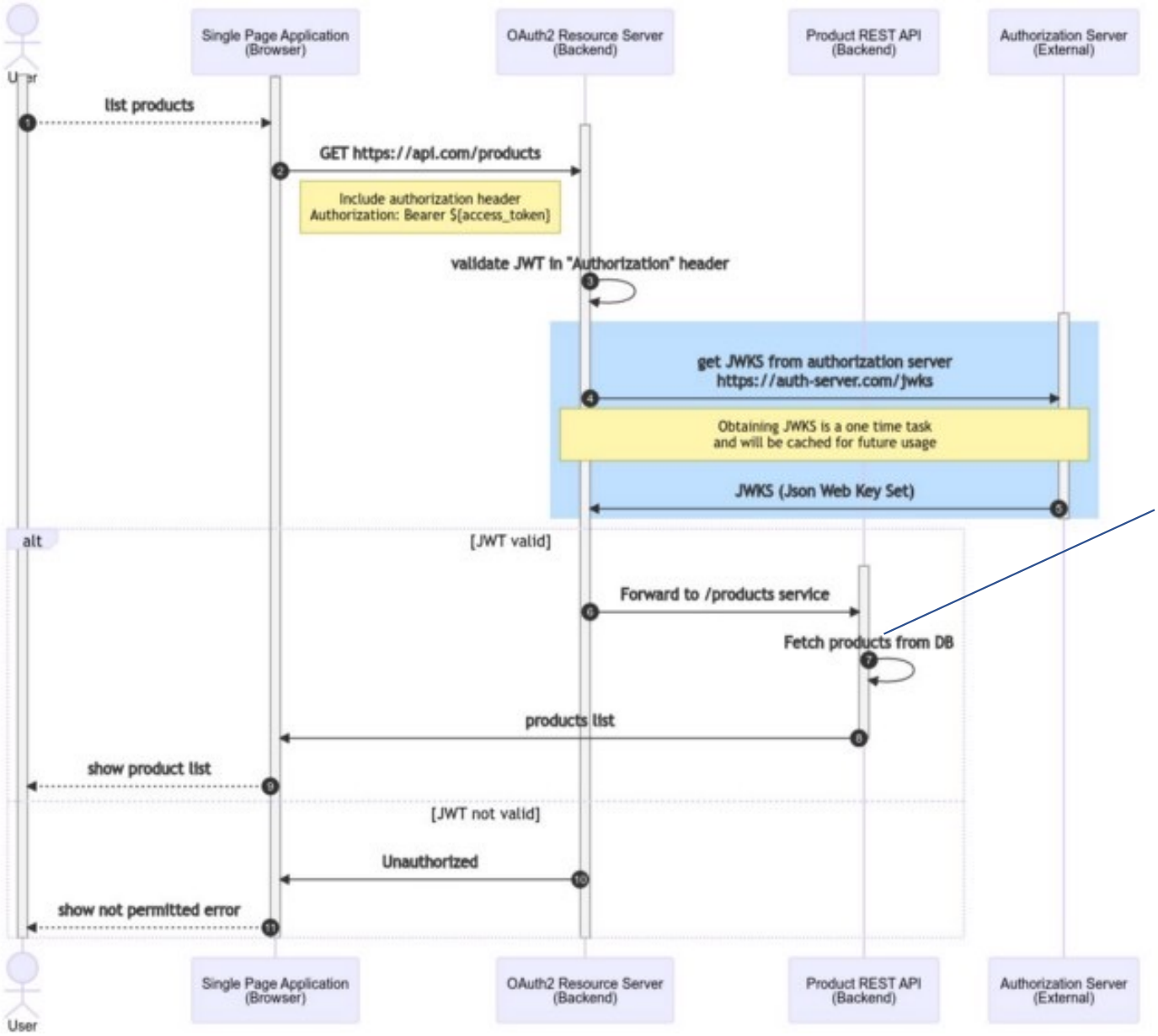
OIDC in SPA

- OIDC is used to authenticate users when implementing SSO.
- The authorization server returns an access_token and an id_token.
- id_token is used by the single page app to know if a user is authenticated.
- The id_token additionally contains some information about the user like username, email and display name depending on the scopes we requested.
- The access_token is used to authenticate API calls made to backend APIs.
- Once the frontend has obtained the access_token it can pass that JWT as the Bearer token (in the Authorization header when invoking the backend API.
- The backend can then validate the signature and the content of the JWT to authenticate the API call against the user.

https://medium.com/geekculture/jwt-authentication-with-oauth2-resource-server-and-an-external-authorization-server-2b8fd1524fc8



https://medium.com/geekculture/jwt-authentication-with-oauth2-resource-server-and-an-external-authorization-server-2b8fd1524fc8



Despite the user is authenticated, it does not mean he/she has authorization to request and access the answer.

Therefore, further authorization checks may be necessary before providing the answer.

https://medium.com/geekculture/jwt-authentication-with-oauth2-resource-server-and-an-external-authorization-server-2b8fd1524fc8

Other Bibliography and References

- https://medium.com/@robert.broeckelmann/saml2-vs-jwt-a-comparison-254bafd98e6