Berner Fachhochschule
Haute école spécialisée bernoise
Bern University of Applied Sciences

Software Architecture Part 1

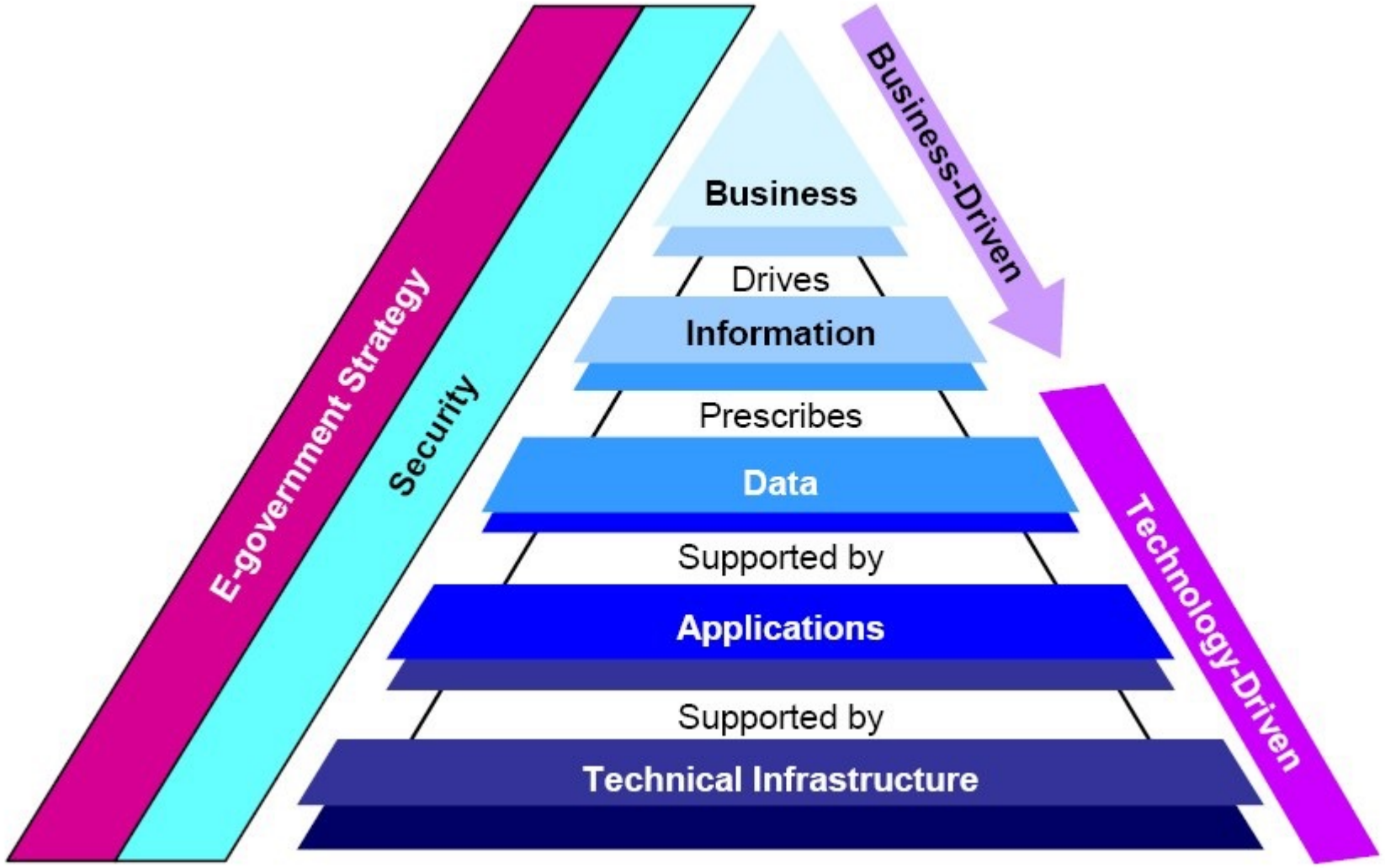
CAS Spring Application Development

Simon Martinelli, simon.martinelli@bfh.ch, v2025.10

TABLE OF CONTENTS

1. ARCHITECTURE

ARCHITECTURE TYPES



THE VALUE OF ARCHITECTURE

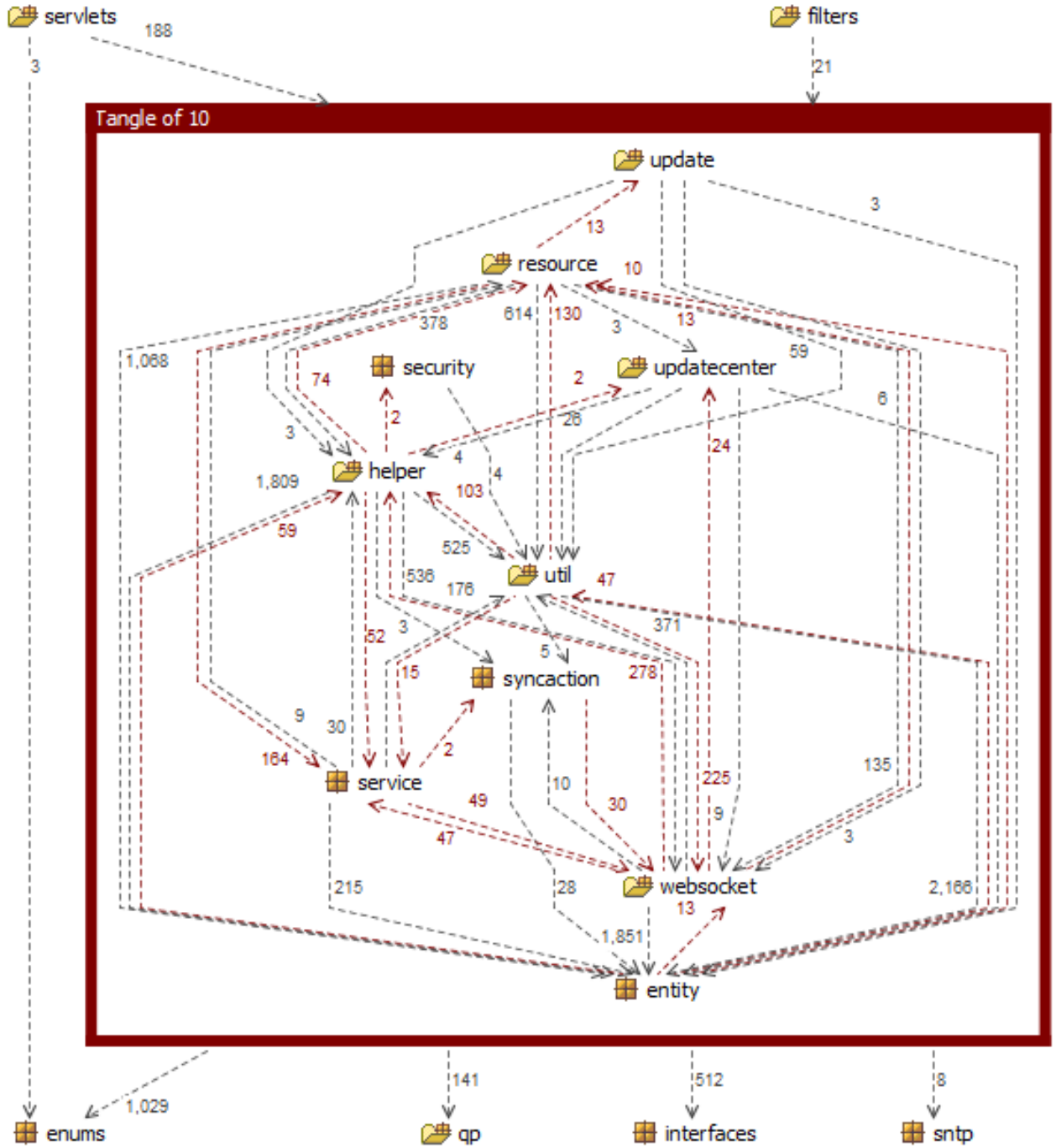
The architecture is the foundation of an application

Successful architecture

The Cheops Pyramid, approx. 2680 BC

Less successful architecture

The Leaning Tower of Pisa, 1173


servlets
188
3
filters
21
Tangle of 10
update
3
13
resource
10
378
614
130
3
13
59
1,068
security
updatecenter
74
2
6
2
28
3
4
24
helper
1,809
4
103
59
525
util
536
176
47
371
52
3
5
15
syncaction
278
9
30
164
2
service
10
225
135
49
30
9
47
3
215
28
websocket
2,166
13
1,851
entity
1,029
141
512
8
enums
qp
interfaces
sntp

DEFINITION

- **Software architecture** refers to the high-level structures of a software system and the discipline of **creating** such **structures** and **systems**.
  Each structure comprises software **elements**, **relations** among them, and properties of both elements and relations

Clements, Paul; Felix Bachmann; Len Bass; David Garlan; James Ivers; Reed Little; Paulo Merson; Robert Nord; Judith Stafford (2010). Documenting Software Architectures: Views and Beyond, Second Edition. Boston: Addison-Wesley. ISBN 978-0-321-55268-6.

SYSTEM REQUIREMENTS

Functional Requirements

- Use cases
- Scenarios
- Epics/Stories
- UI Mockups

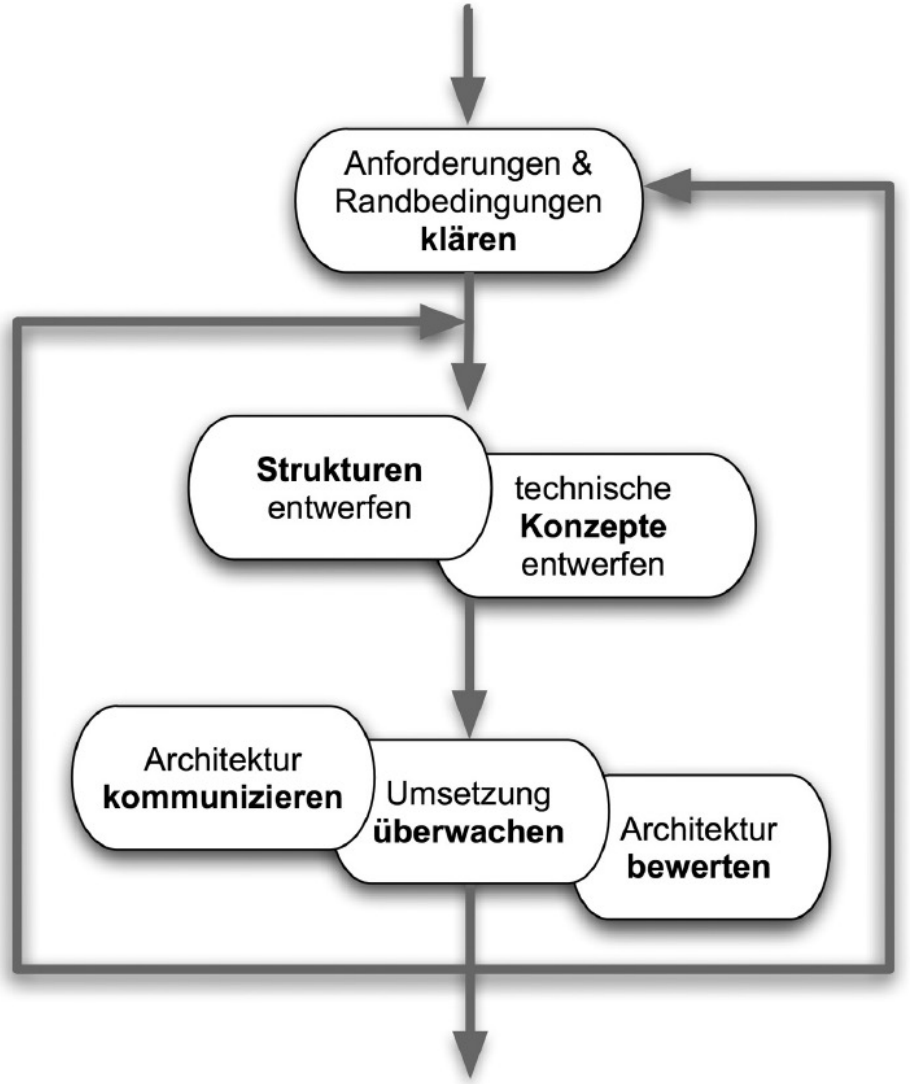
Non-functional requirements

- Security
- GPDR
- Compliance with standards
- Availability (99.9% per year)
- Reliability (Fail Over, Stand By)
- Performance (response time <= 0.5 s)
- Scalability (example: extension from 10 to 100 parallel users)

NON-FUNCTIONAL REQUIREMENTS ISO/IEC 9126-1

- **Functionality**
  - Suitability
  - Accuracy
  - Interoperability
  - Security
- **Reliability**
  - Maturity
  - Fault tolerance
  - Recoverability
- **Usability**
  - Understandability
  - Learnability
  - Operability
  - Attractiveness
- **Efficiency**
  - Time behavior
  - Resource utilization
- **Maintainability**
  - Analyzability
  - Changeability
  - Stability
  - Testability
- **Portability**
  - Adaptability
  - Installability
  - Replaceability

TASKS OF THE SOFTWARE ARCHITECT



COMPONENTS

- A **component** is **a self-contained** unit of software, consisting of a sequence of processing steps and data structures
- Components provide an **encapsulation** via the separation of **interface** and implementation
- A component can even call up other components - thus a **hierarchy** of program calls is possible
- Components are important for several reasons:
  - Large, complex programs can be organized and **structured** through the use of components
  - Functionalities can be **integrated** on the modular principle
  - Several development groups can **independently** edit and test individual components
  - (Program logic is **reusable**, without the code having to be redundantly created and maintained)

- Modular programming, David L. Parnas, 1971

SEPARATION OF INTERFACE AND IMPLEMENTATION

- **Interface**
  - Each component has an interface that clearly specifies the performance of the component, regardless of how this performance is realised
  - An interface represents the sum of the functions or services, which provide its users with a component

- **Implementation**
  - The code that is actually executed
  - This implementation is exchangeable, whereby, in order to maintain the consistency of the components, the interface remains constant

- Information hiding, David L. Parnas, 1972

SEPARATION OF CONCERNS

- **Separation of responsibilities**
  - Frequently used to separate functional from non-functional requirements
  - The Business Code is not to become bloated
  - Can be achieved by Aspect-Oriented Programming (**AOP**)

- On the role of scientific thought, Edsger W. Dijkstra, 1974

WHY DISTRIBUTION AT ALL?

Non-distributed applications

- Are more secure
- Perform better
- Are simpler to implement

The main **reason for distribution** is sharing resources

- Hardware → cost savings
  - Printers, plotters, scanners
- Data and information → information exchange
  - File server, database, World Wide Web, search engines
- Functionality → error prevention and reuse
  - Centralising functionality

THE THREE-DIMENSIONAL DISTRIBUTED SYSTEMS

- **Distribution and communication**
  - Comfort of the programming
  - Middleware
  - Remote Procedure Call (RPC)
- **Concurrency**
  - Concurrently independent of each other
  - synchronisation
  - Concurrency
  - Synchronous
  - Asynchronous (callback, polling)
- **Persistence**
  - Permanent storage

BOOKSTORE: FIRST STEP

- Create the Use Cases Diagram

ARCHITECTURE TYPES

http://www.oreilly.com/programming/free/software-architecture-patterns.csp

ARCHITECTURE TYPES

1. Layered Architecture
2. Hexagonal/Onion/Clean Architecture
3. Event-Driven Architecture
4. Microkernel Architecture
5. Microservice Architecture
6. Serverless Architecture

Source: Software Architecture Pattern, Mark Richards,
http://www.oreilly.com/programming/free/software-architecture-patterns.csp

LAYER ARCHITECTURES

1. Physical tiers
2. Logical tiers

PHYSICAL TIERS

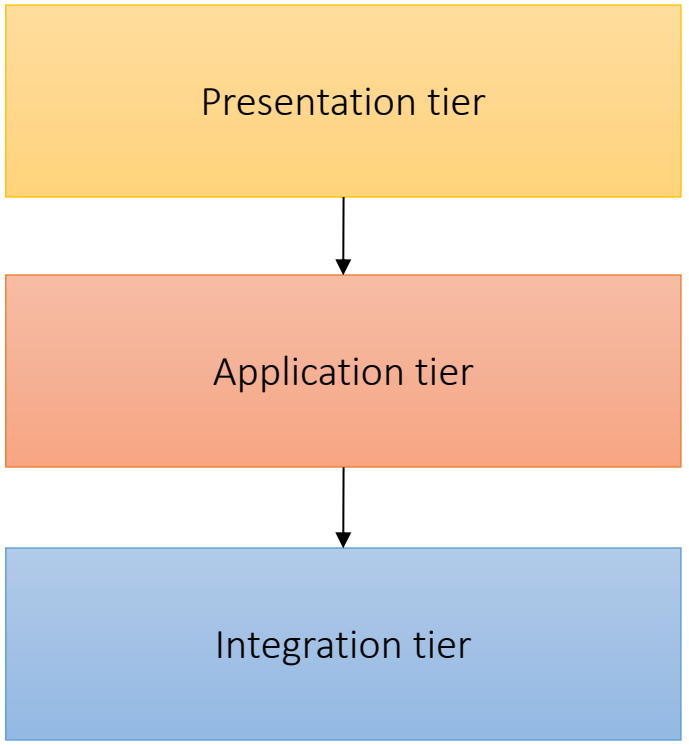
- A physical tier features an **independent process** area within a distributed application
- The physical tier is the basis of the n-tier architectures
- n-tier architectures are an extension/refinement of the client-server architecture model
- An n-tier architecture determines how many different client and server process areas exist within a distributed application
- The n specifies how many process areas are involved
- A process area may, but need not (!), correspond to a physical computer.

- The use of n-tier architectures makes sense especially for information systems
  - Distribution of software to reduce complexity
  - Interactive Human-Machine Interface
  - Data-centric approach

LOGICAL TIERS

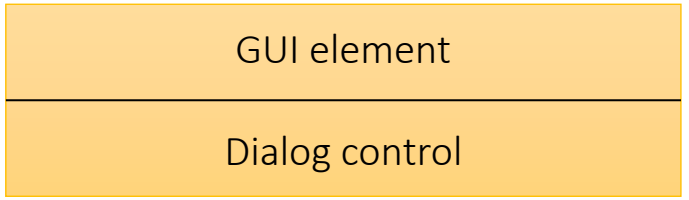
- Advantages of distributed systems
  - Central data storage
  - Scalability
  - Fault tolerance

- Classical logical 3 tiers
  - Presentation tier
  - Application tier
  - Persistency or Integration tier



PRESENTATION TIER

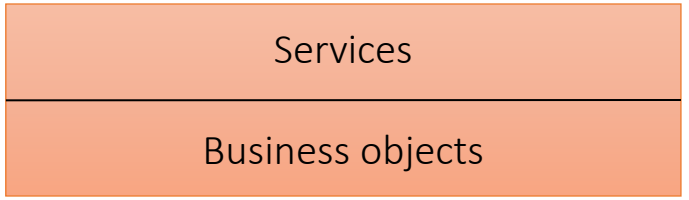
- Primary Task
  - User interface to present data and to respond to user input
- Design Pattern Model View Controller (MVC)
  - GUI Elements (View)
  - Dialog control (Controller)
  - Data (Model)
- In order to achieve a loose coupling the presentation layer knows as little as possible of the business logic
- Moreover, it should - in terms of a layered architecture - have no knowledge of the persistency tier



APPLICATION TIER

- Primary Task
  - Implement business processes

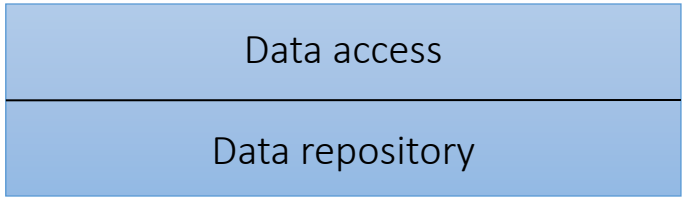
- Can be separated in
  - Services
    - Makes methods and services available in order to realise the business processes
    - Meets the requirements of the presentation tier
    - Mostly stateless
  - Business objects
    - Business Object Model
    - Usually persistent



PERSISTENCY OR INTEGRATION TIER

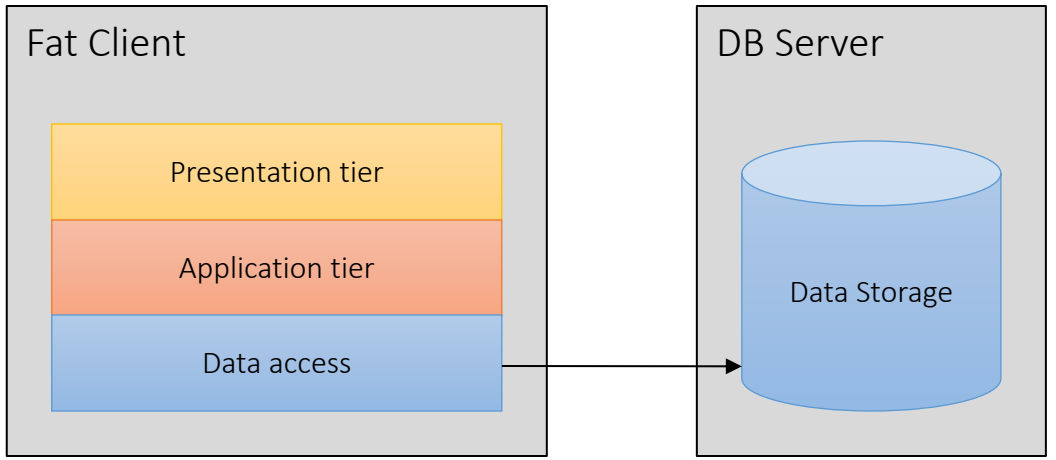
- Primary Task
  - to distribute services necessary for managing the data in a database system or acessing data in a 3rd party system like SAP

- Can be separated in
  - Data access
    - O/R Mapping (e.g. JPA)
    - JDBC/SQL
    - JCA
    - REST/SOAP clients
  - Data repository
    - Database system (usually relational)



2-TIER aka Client/Server

- The model supports 2 tiers: client and server
- Assignment of tasks to the tiers:
  - Presentation - Client-Tier
  - Application logic - Client tier and/or server tier
  - Data repository - server tier
- The distribution of the application logic to the client and server tier may vary

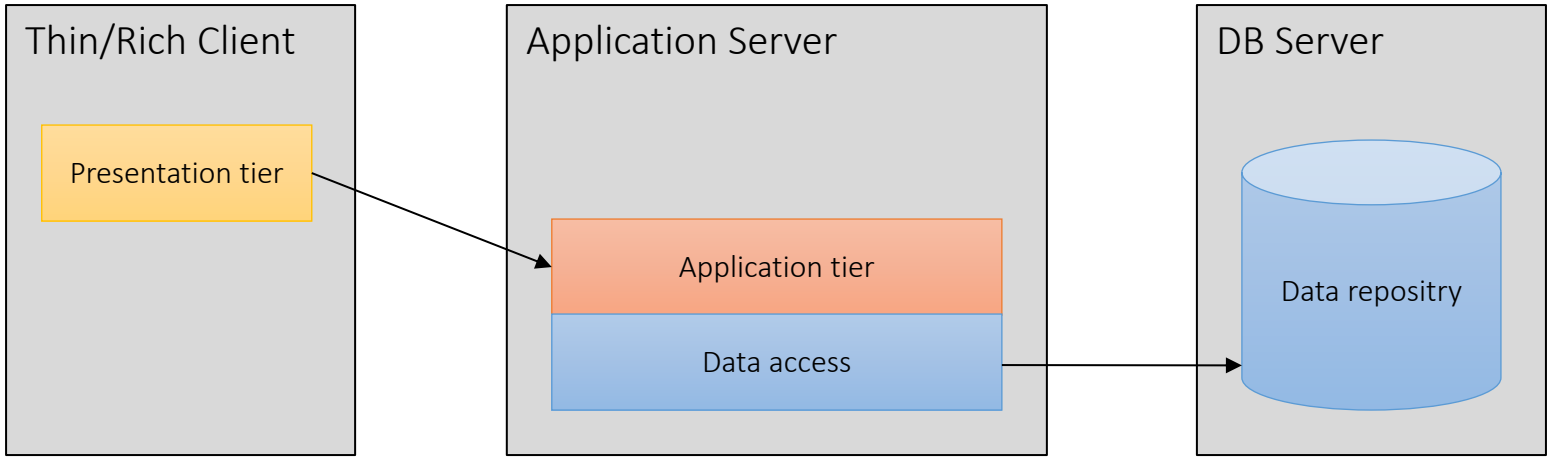


2-TIER

- First and oldest distribution model
- Often used in conjunction with stored procedures and 4GL:
  - 4GL (Fourth Generation Languages) directly support language elements for accessing a database
  - Stored procedures are pre-defined SQL statements that are stored in the database and invoked at runtime.
- Advantages:
  - simple and rapid implementation
  - performance
- Problems:
  - hard to maintain
  - poorly scalable
  - software update problem

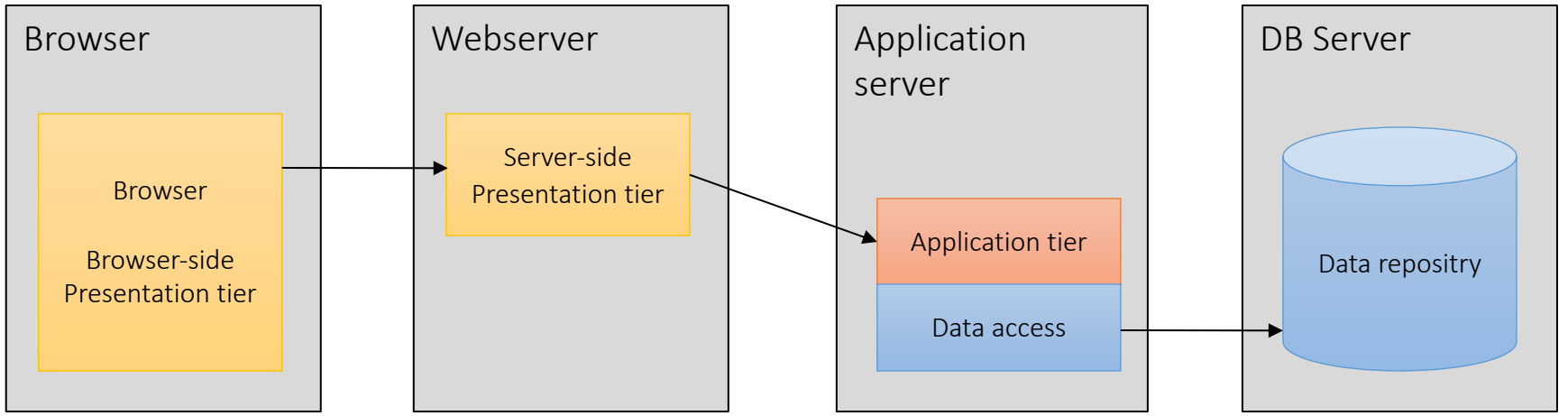
3-TIER

- Lightweight Client (Thin or Rich Client)
- Complete business logic on the application server
- Advantages:
  - Better scalable and monitorable
  - Easier software distribution
  - Also workable on low power devices (PDA, mobile phones, etc.)

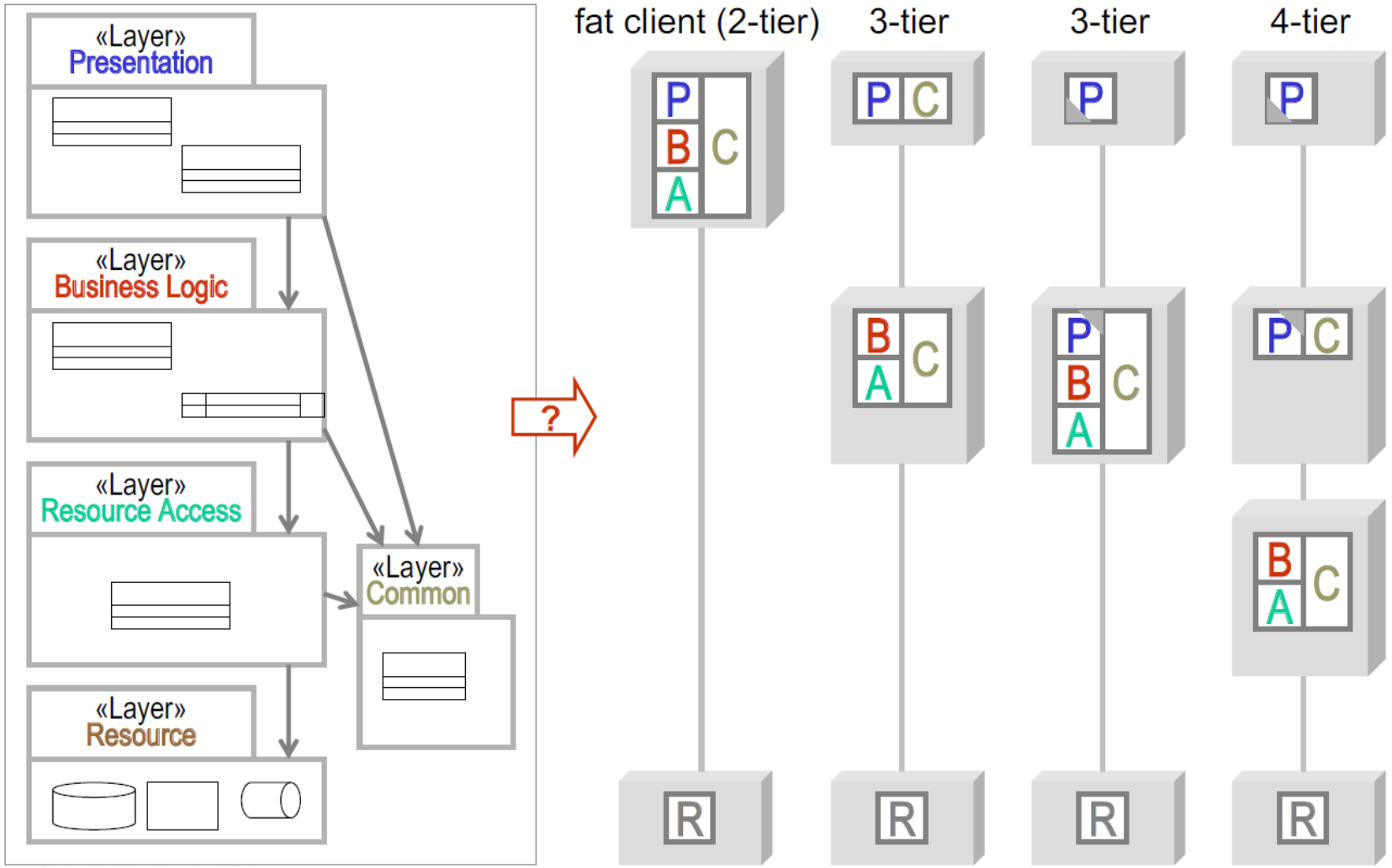


N-TIER

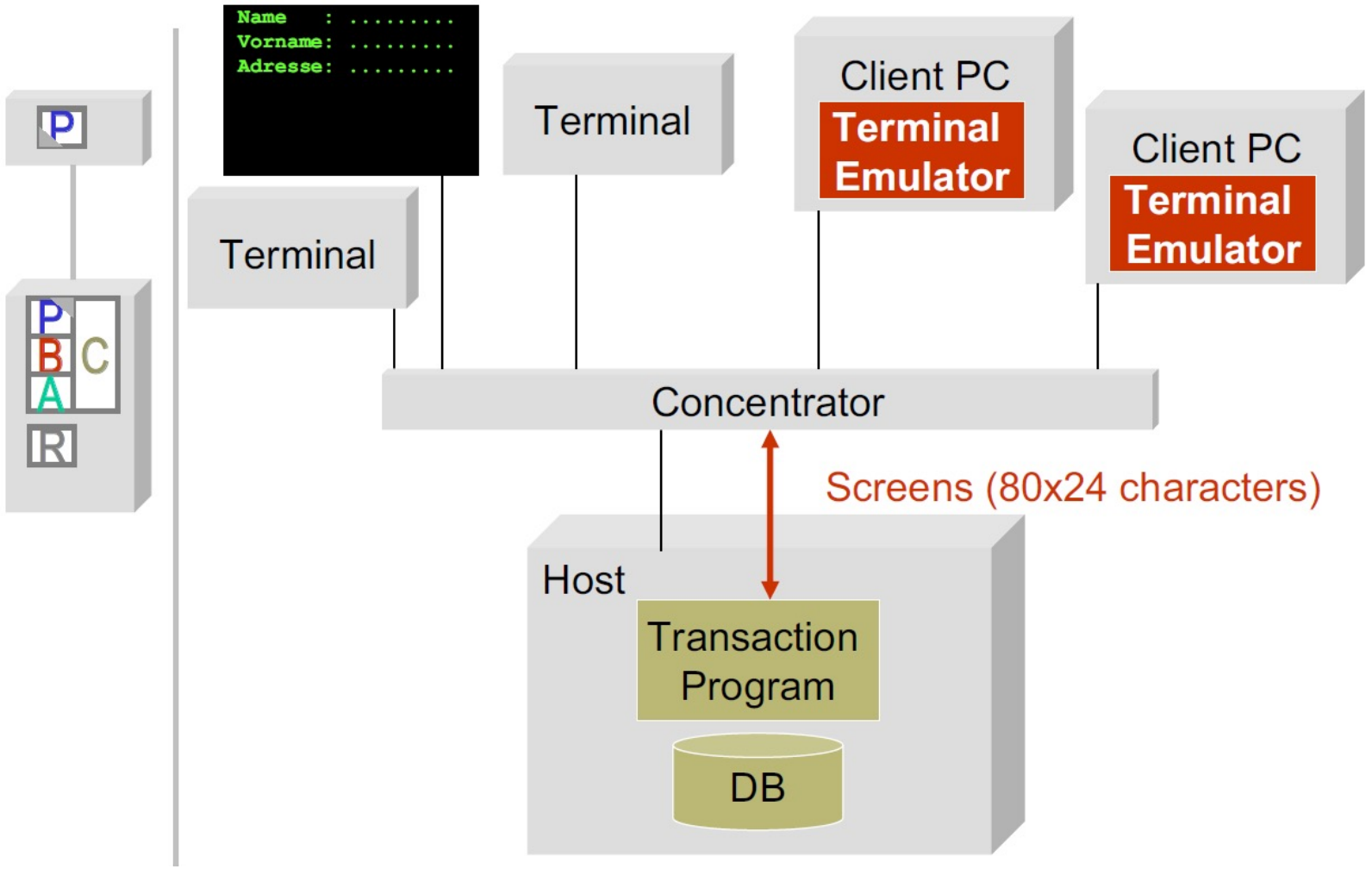
- Clients very lightweight
- Web server (usually integrated in the application server) generates dynamic content (JSP, JSF, ASP.NET, etc.)
- Client hardware reducible to a minimum
- No software distribution required



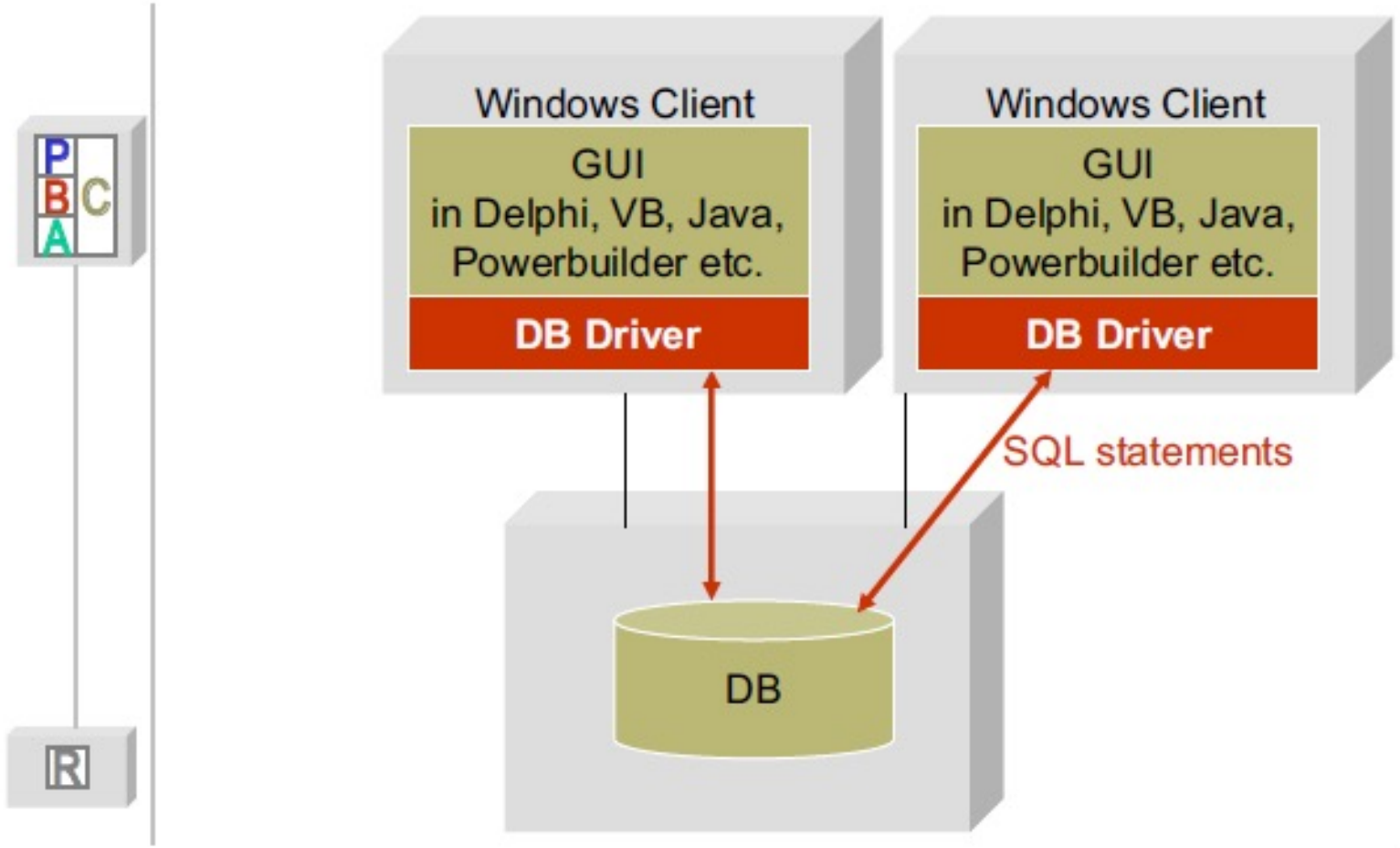
LAYERS AND DISTRIBUTION



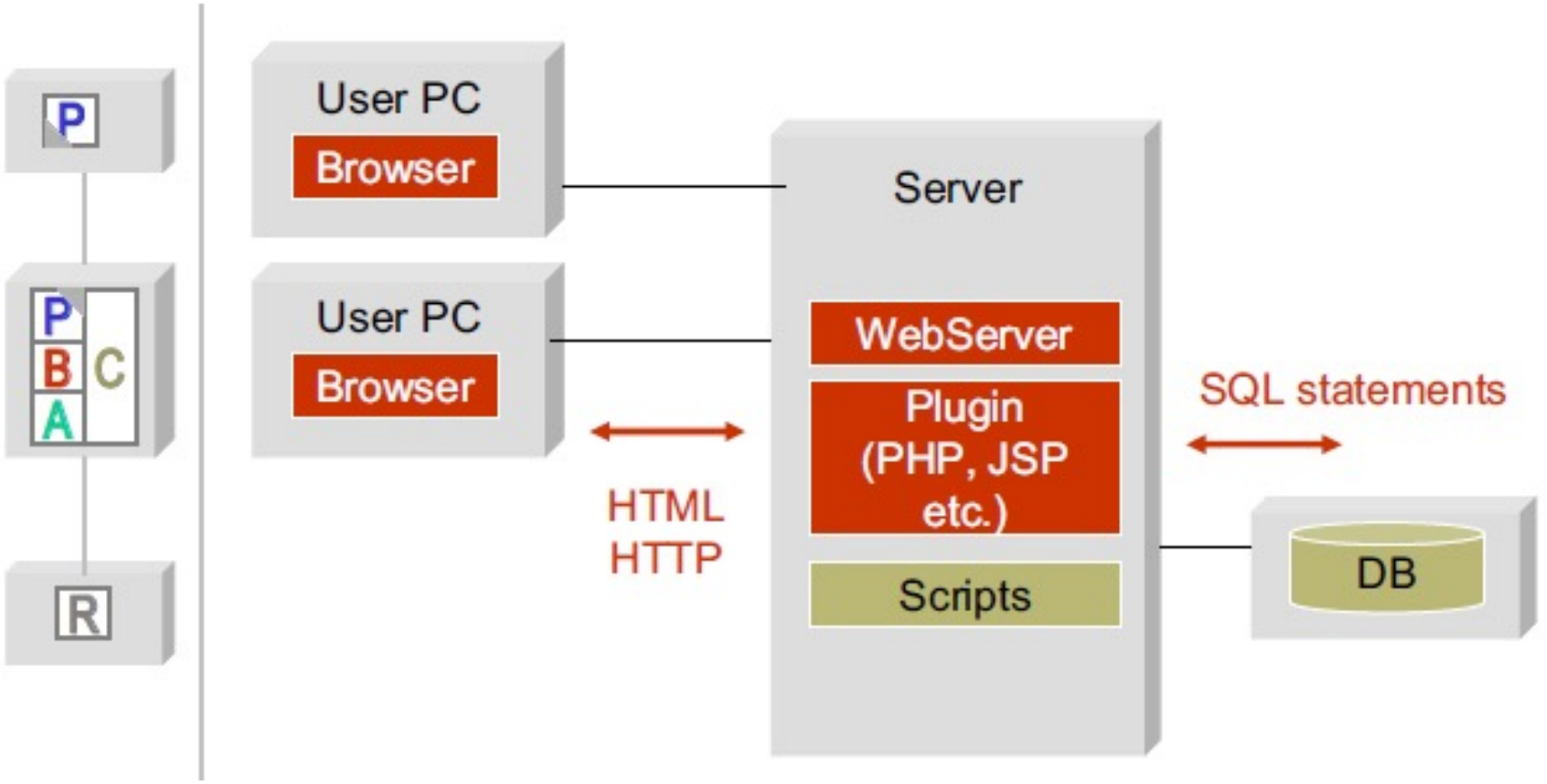
HOST


Name : .....
Vorname: .....
Adresse: .....
P
P
B
A
C
R
Terminal
Terminal
Client PC
Terminal Emulator
Client PC
Terminal Emulator
Concentrator
Screens (80x24 characters)
Host
Transaction Program
DB

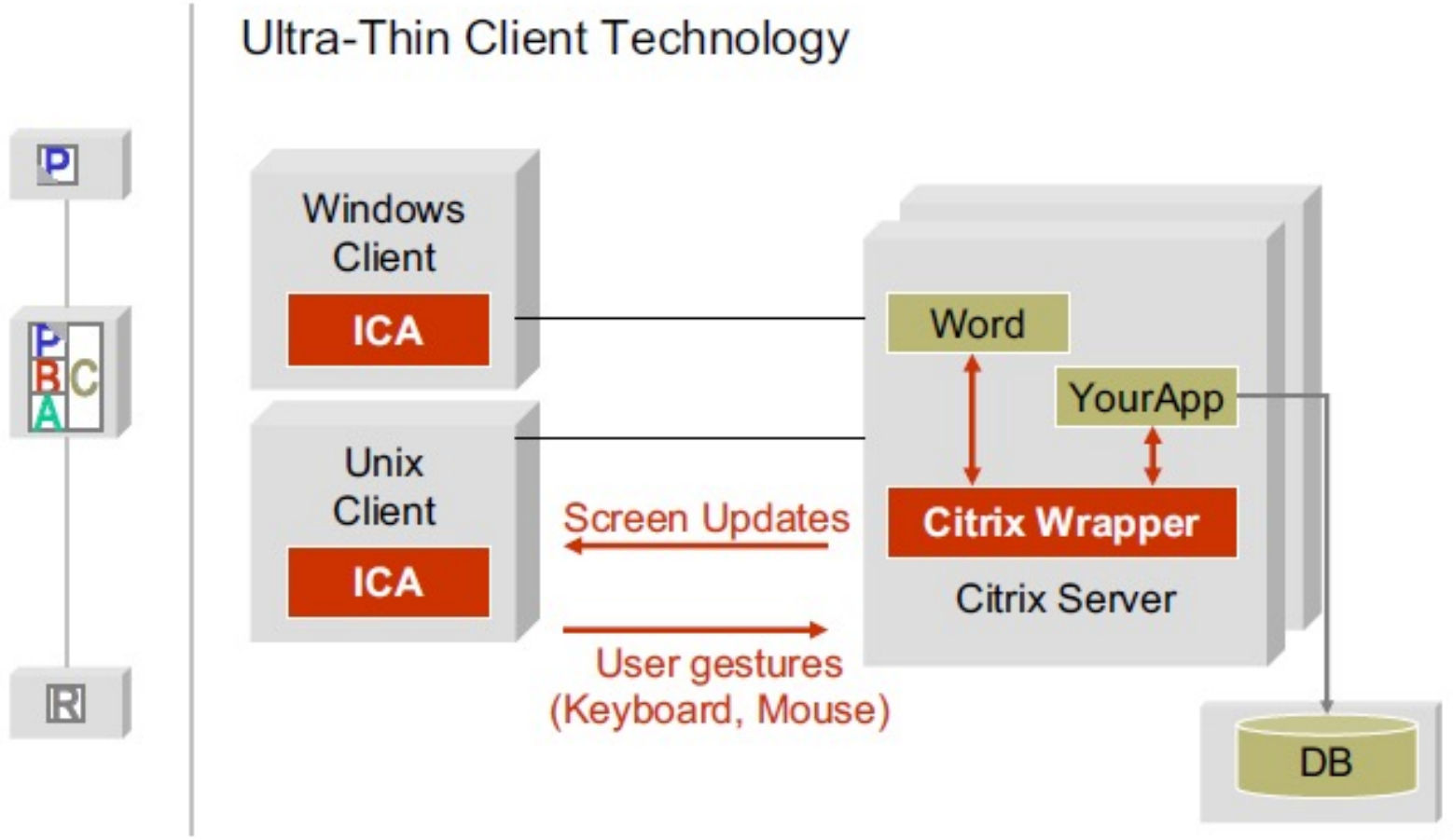
FAT CLIENT


P
B
C
A
R
Windows Client
GUI
in Delphi, VB, Java,
Powerbuilder etc.
DB Driver
Windows Client
GUI
in Delphi, VB, Java,
Powerbuilder etc.
DB Driver
SQL statements
DB

WEB


P
P
B
A
C
R
User PC
Browser
User PC
Browser
HTML
HTTP
Server
WebServer
Plugin
(PHP, JSP
etc.)
Scripts
SQL statements
DB

CITRIX METAFRAME

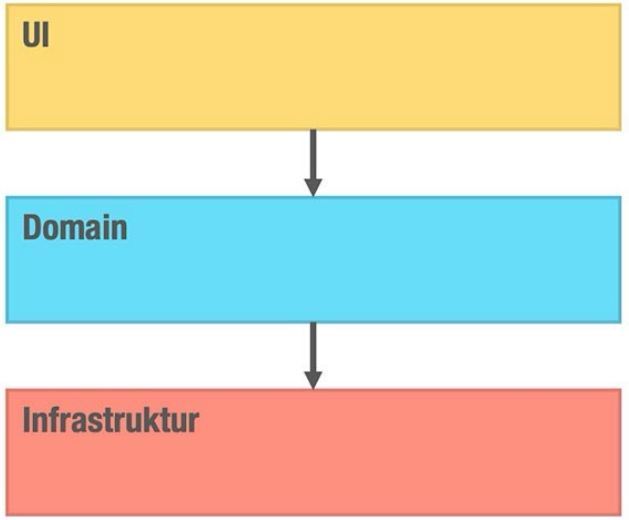


HEXAGONAL/ONION/CLEAN ARCHITECTURE

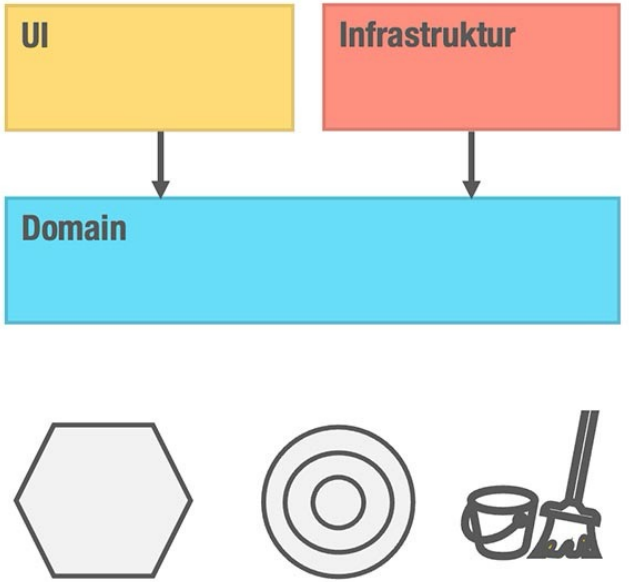
https://entwickler.de/software-architektur/onion-clean-hexagonale-architektur-ddd

DEPENDENCY INVERSION

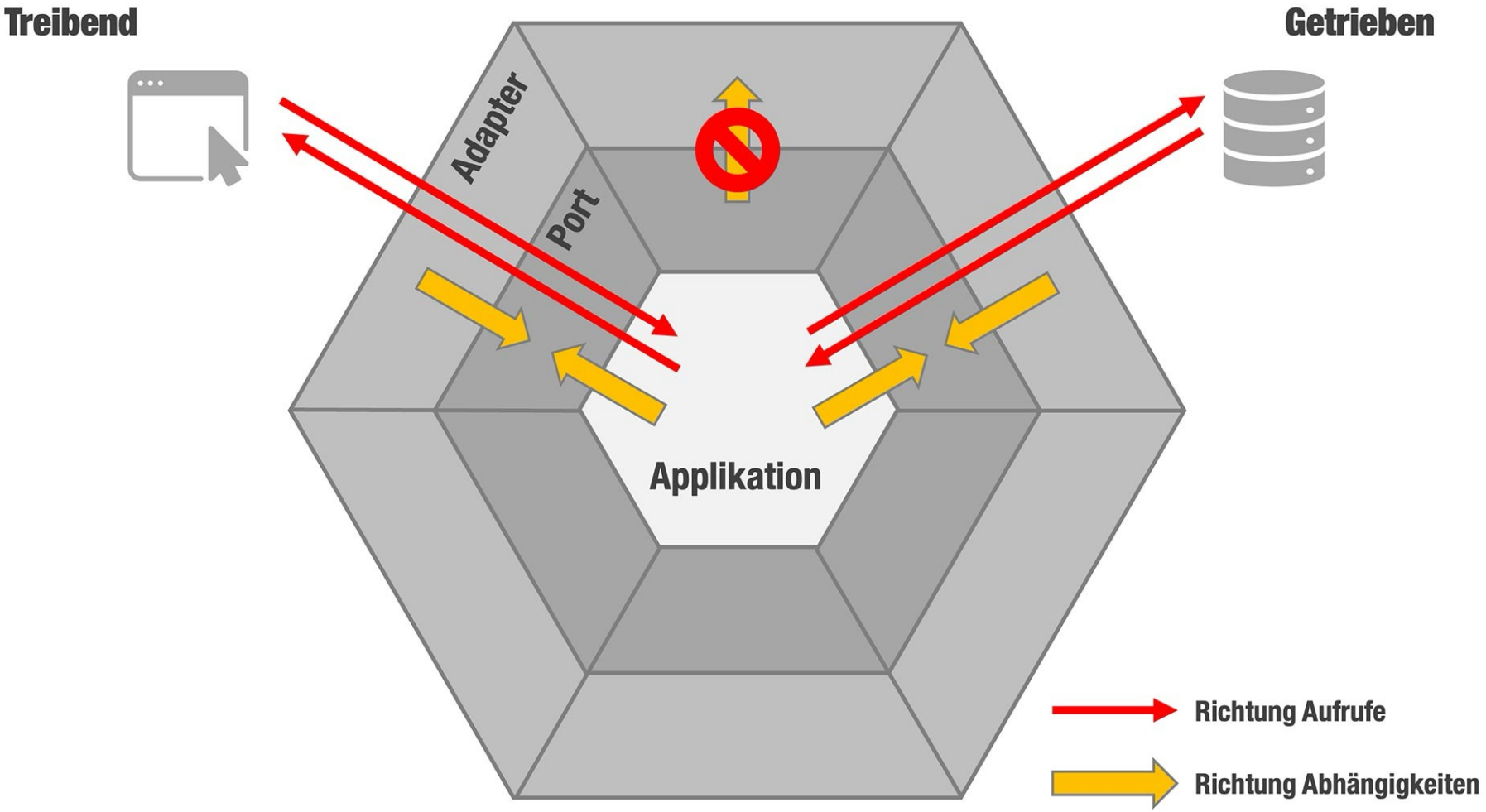
Schichten



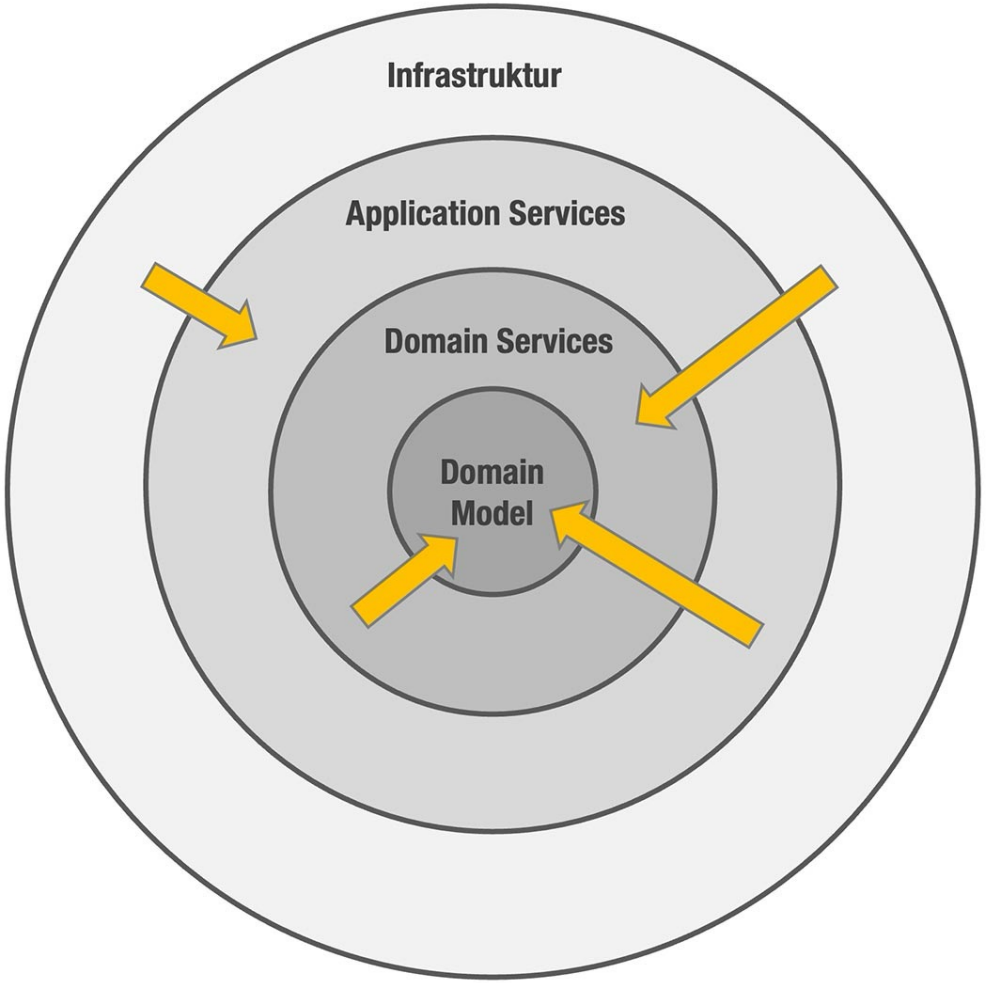
Dependency Inversion Architectures



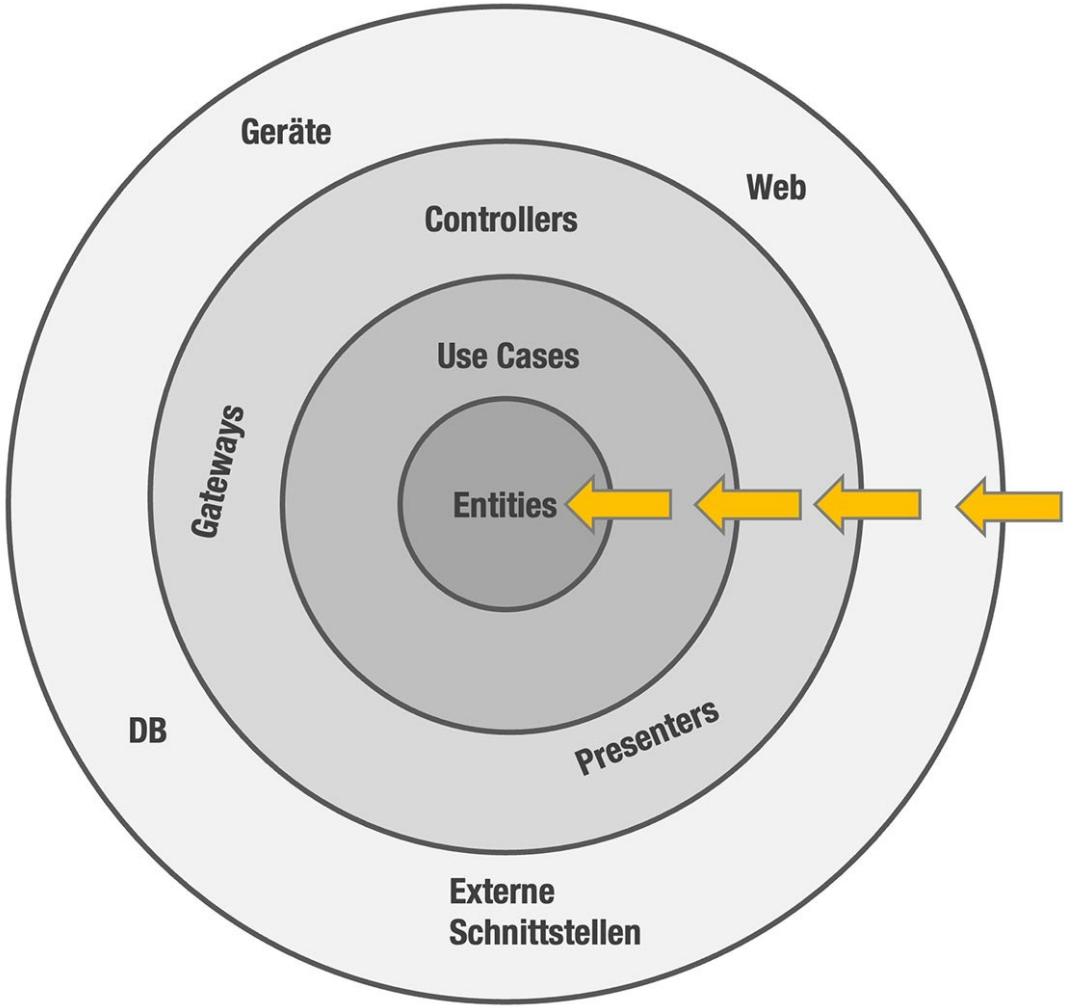
HEXAGONAL ARCHITECTURE



ONION ARCHITECTURE

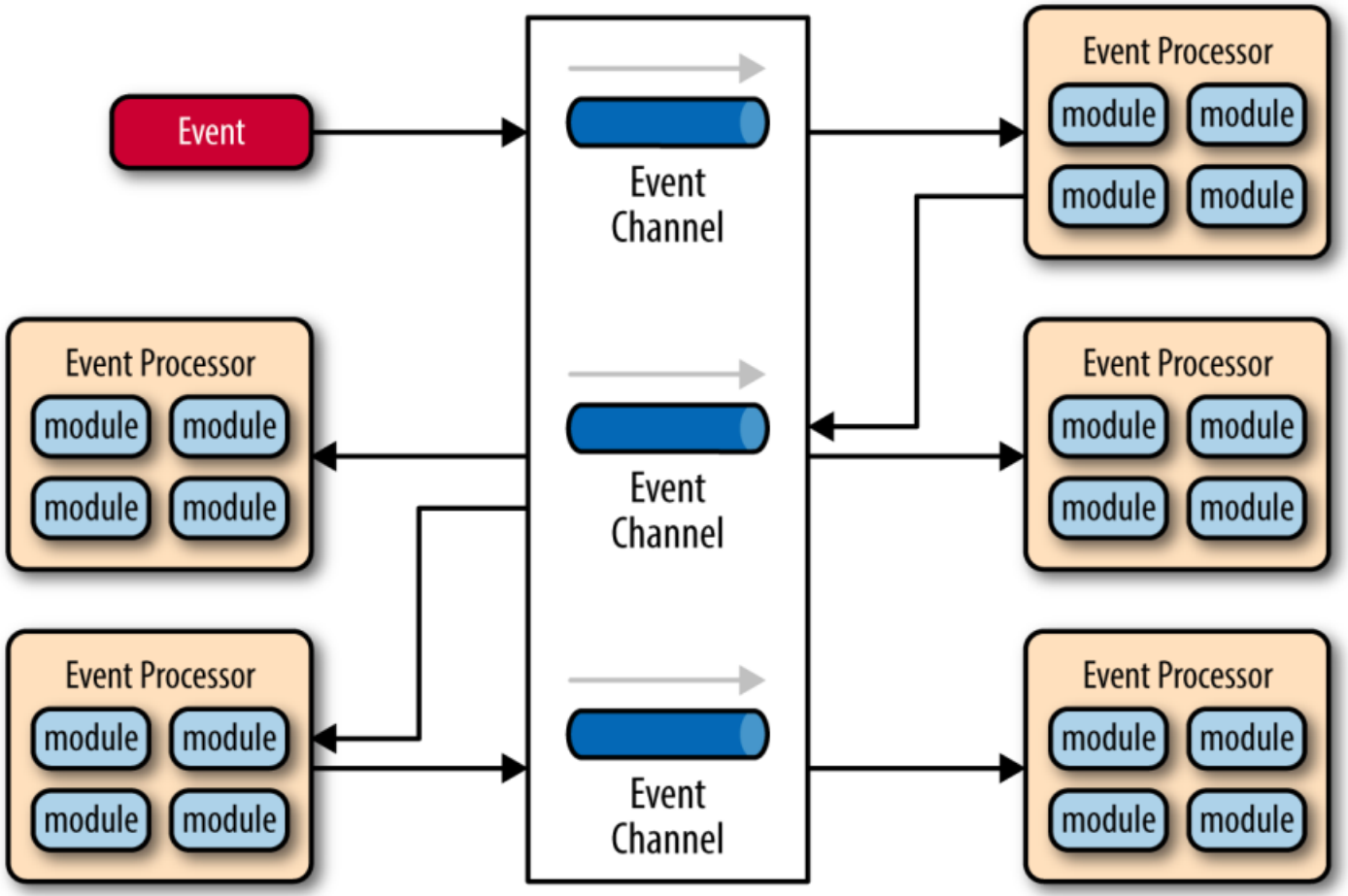


CLEAN ARCHITECURE



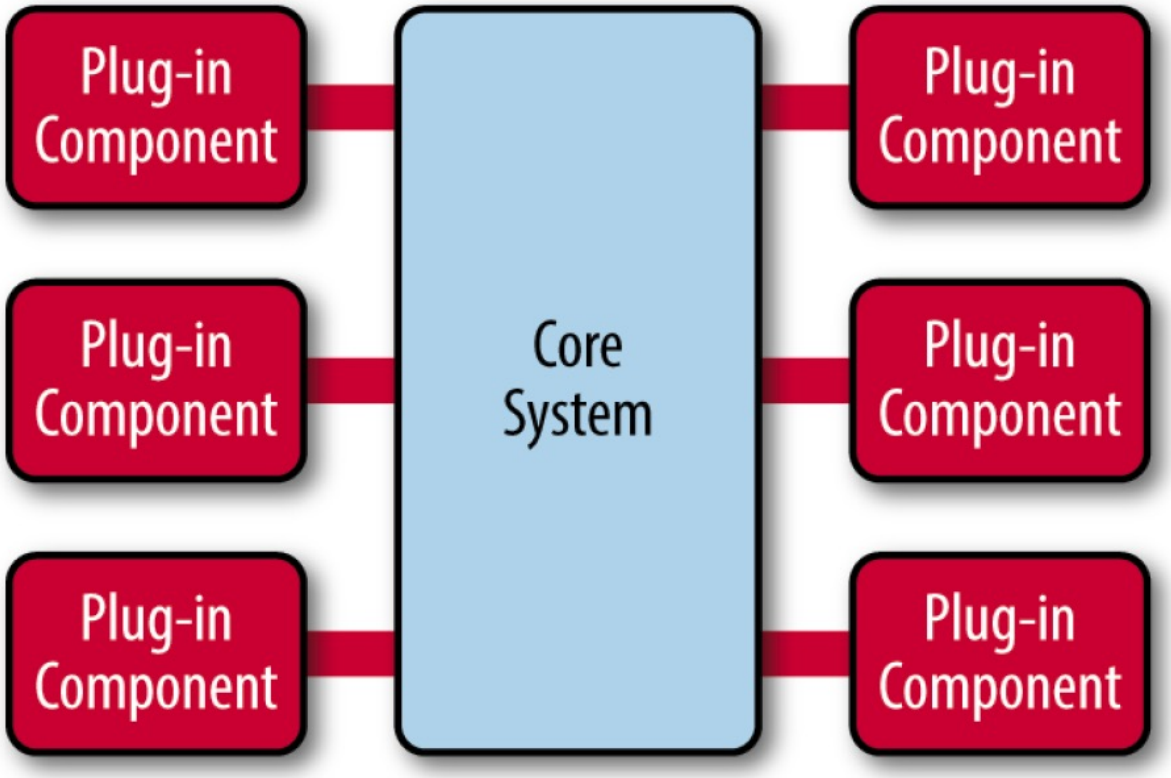
EVENT-DRIVEN ARCHITECTURE

EVENT-DRIVEN ARCHITECTURE



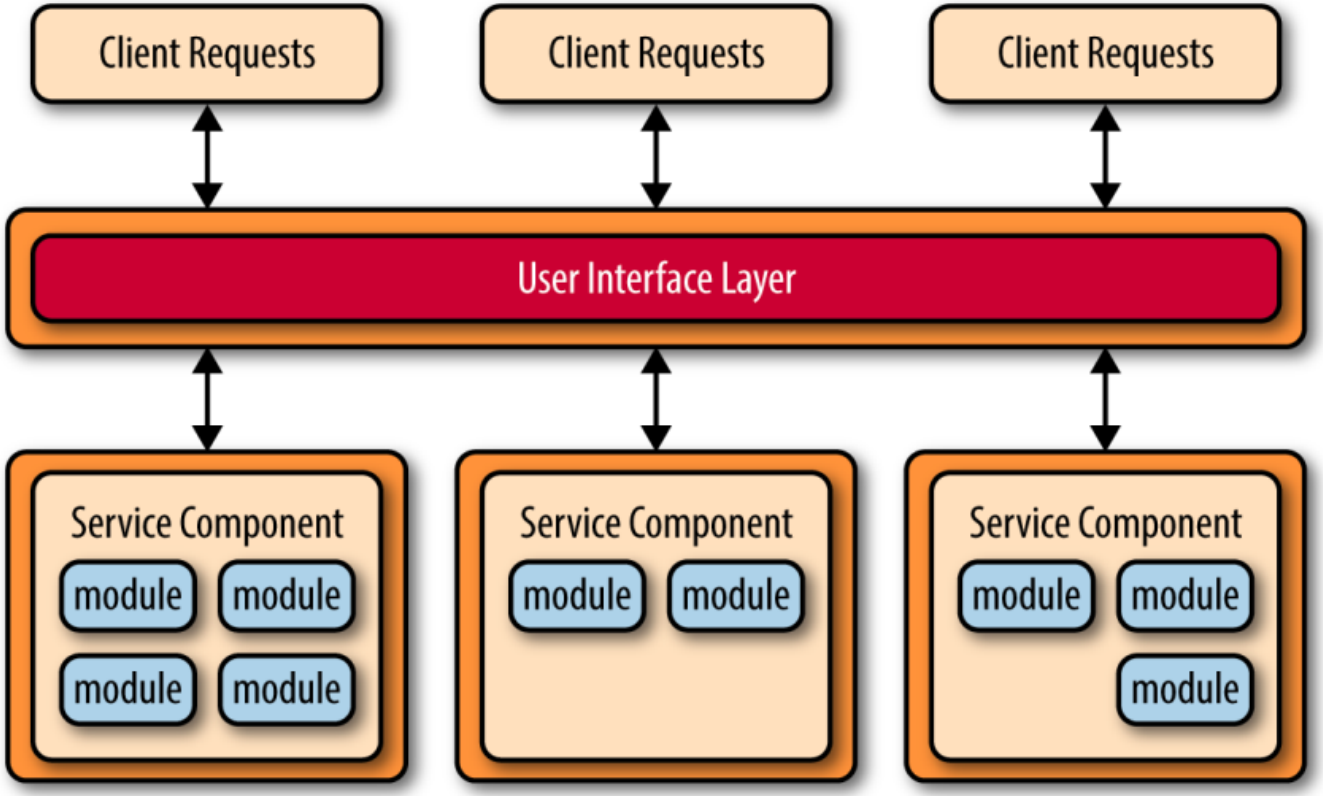
MICROKERNAL ARCHITECTURE

MICROKERNEL ARCHITECTURE



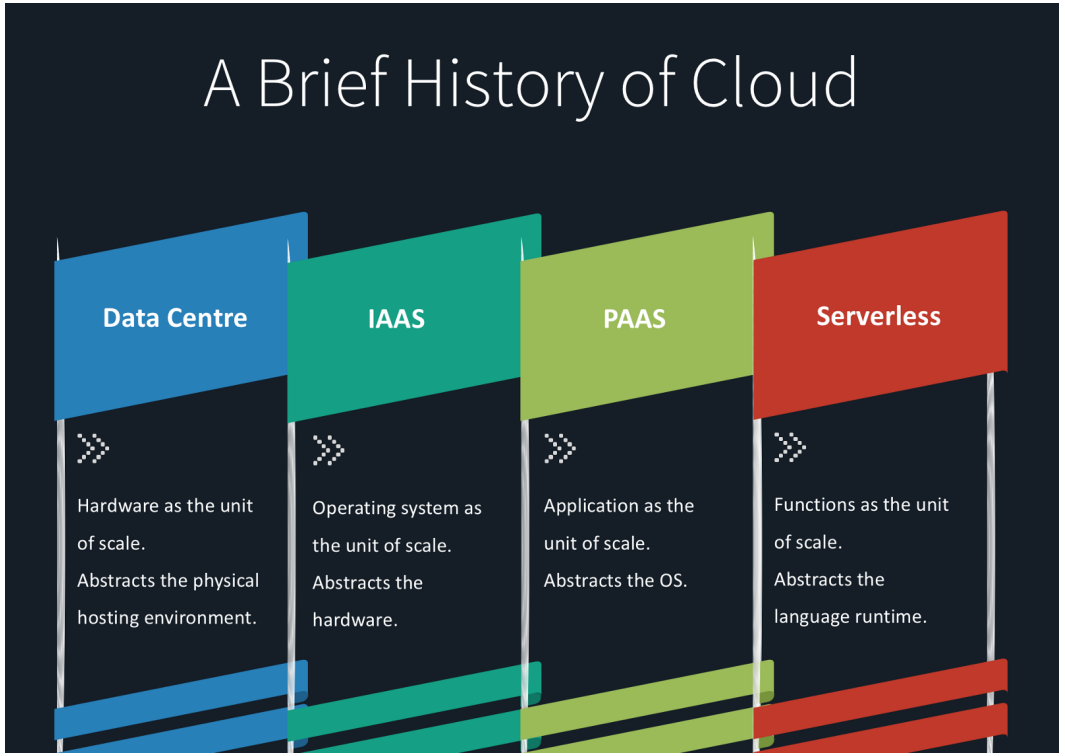
MICROSERVICE ARCHITECTURE

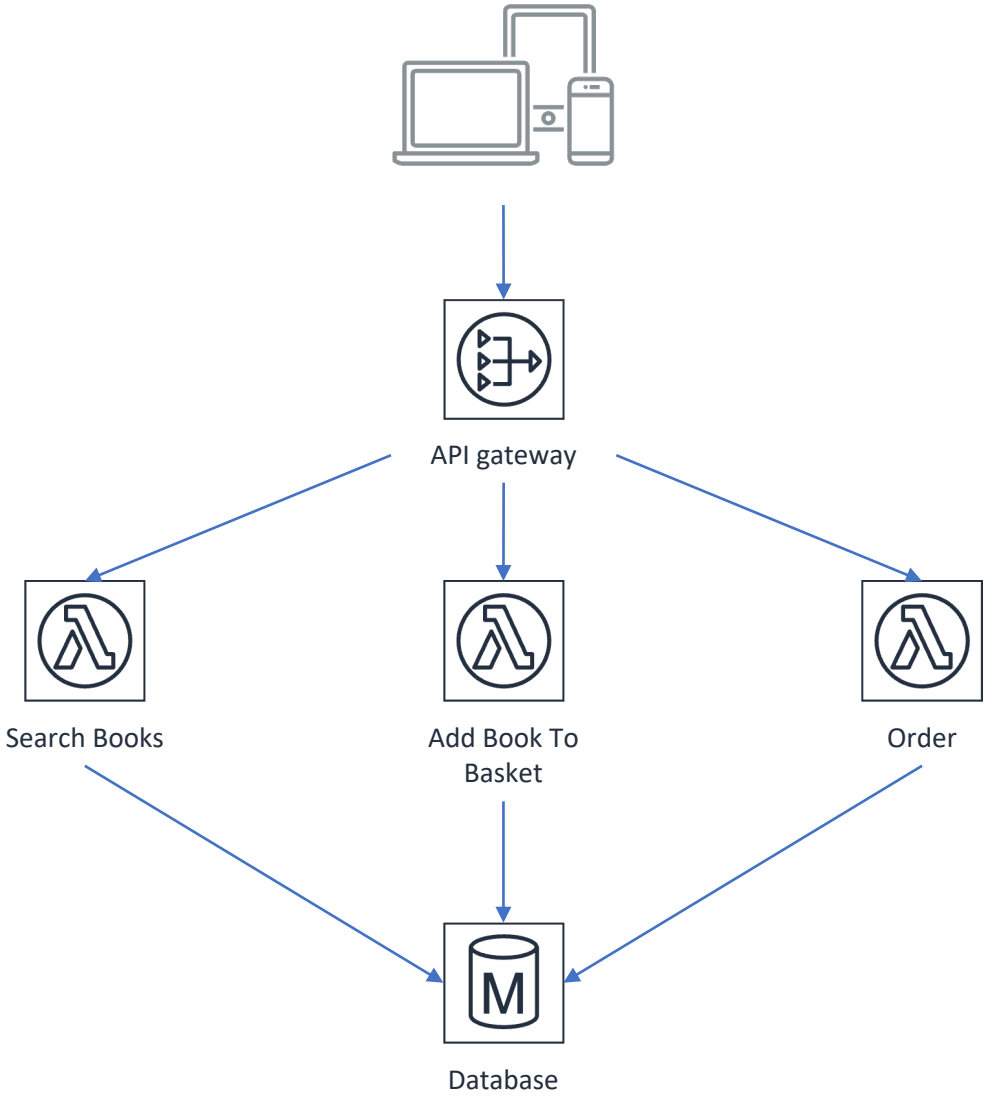
MICROSERVICE ARCHITECTURE



http://martinfowler.com/articles/microservices.html

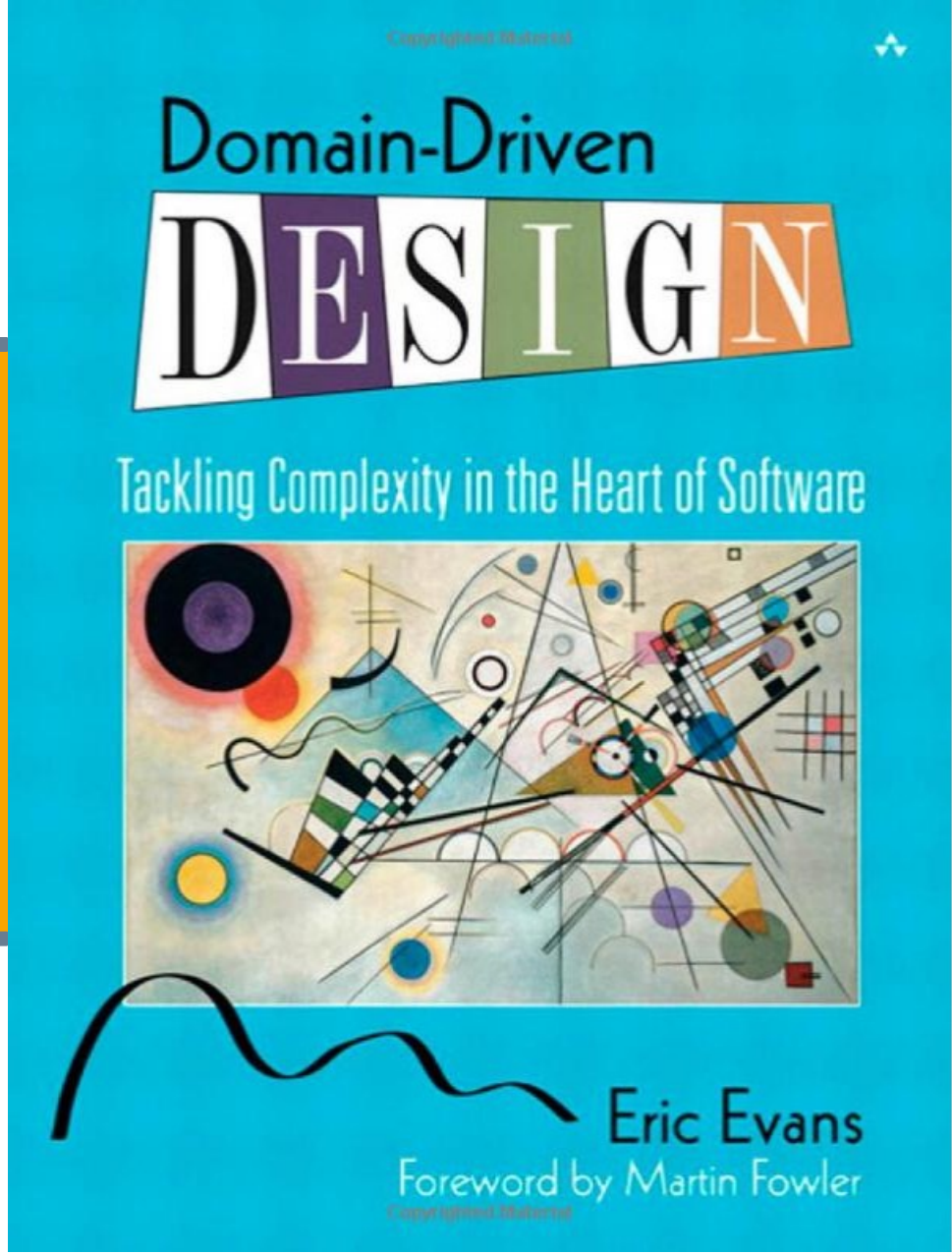
SERVERLESS ARCHITECTURE



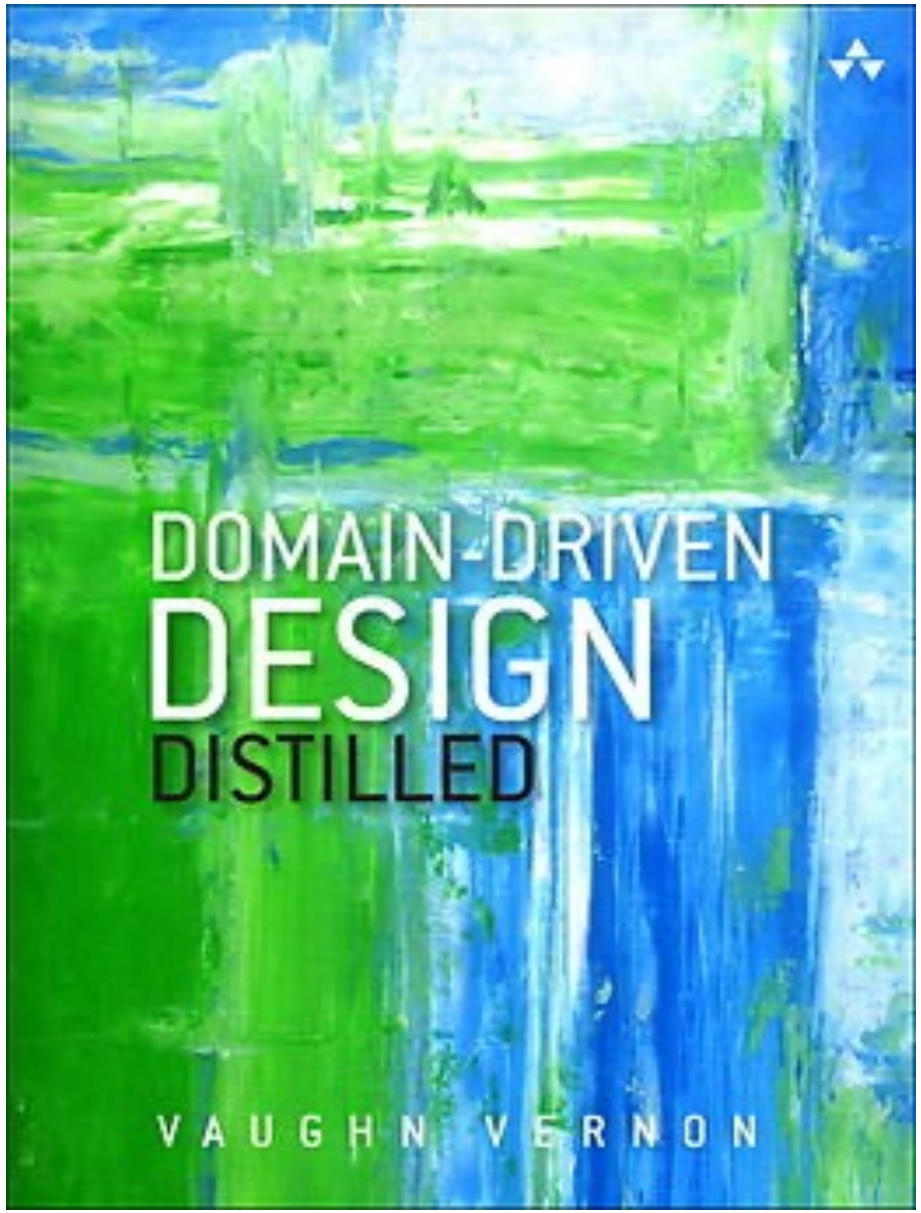


2. DOMAIN DRIVEN DESIGN

Eric Evans, 2003



Source: https://domainlanguage.com/wp-content/uploads/2016/05/DDD_Reference_2015-03.pdf


DOMAIN-DRIVEN
DESIGN
DISTILLED
VAUGHN VERNON


Vaughn Vernon
Domain-Driven
Design
kompakt
→ Aus dem Englischen übersetzt
von Carola Lilienthal und Henning Schwentner
dpunkt.verlag

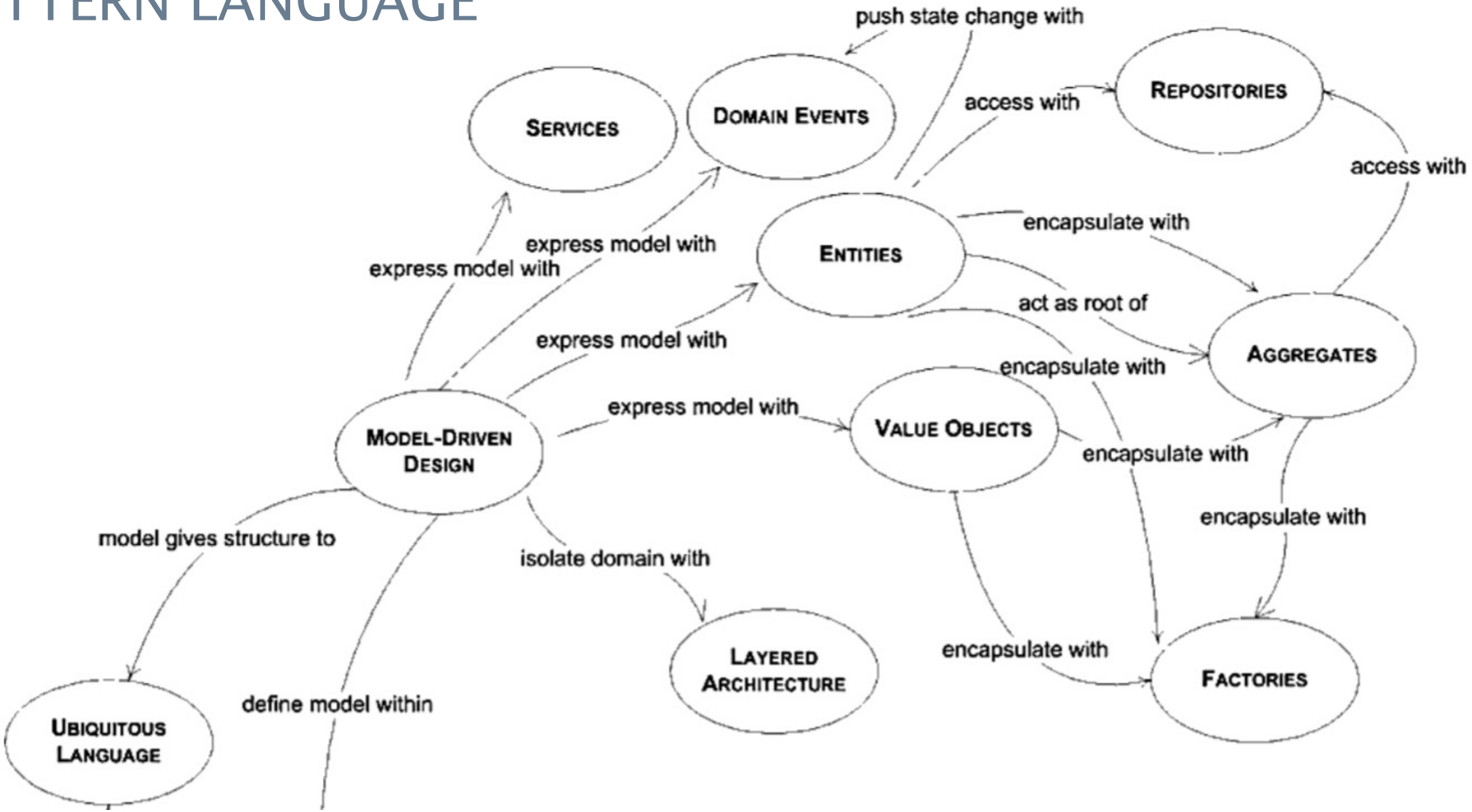
CORE CONCEPTS OF DOMAIN DRIVEN DESIGN

- **Focus on business functionality**
- Joint creation of models by **domain experts and developers**
- Usage of **ubiquitous language**
- Separation of concerns in **bounded contexts**
- **Encapsulation of business and infrastructure code**

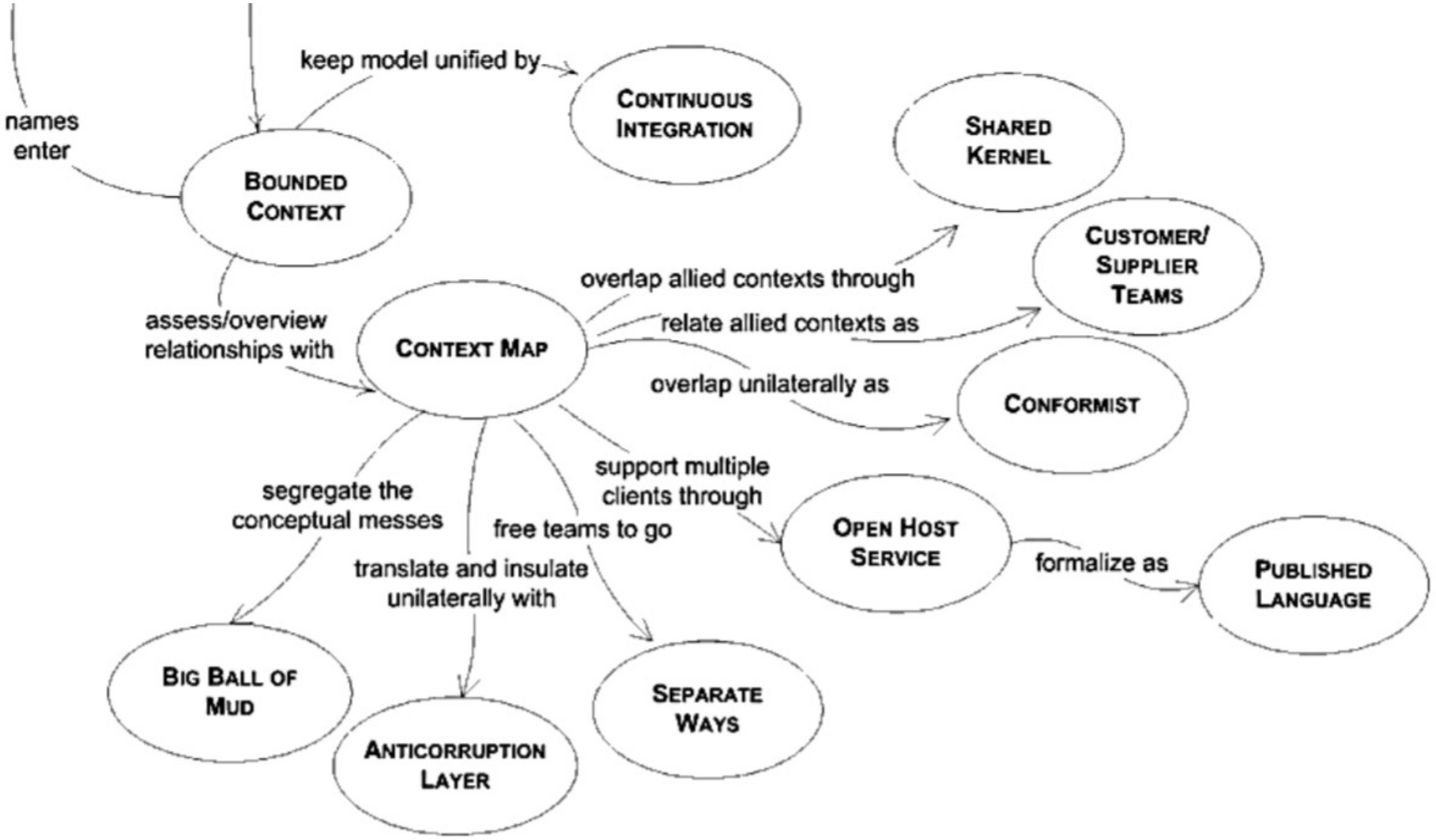
CONSEQUENCES

- **Close corporation between business and development**
- **Late definition with implementation details**
- **It's all about the business domain → New role of developer**

PATTERN LANGUAGE



PATTERN LANGUAGE



DOMAIN

- A sphere of knowledge, influence or activity

- The subject area to which a user applies a program is the domain of the software

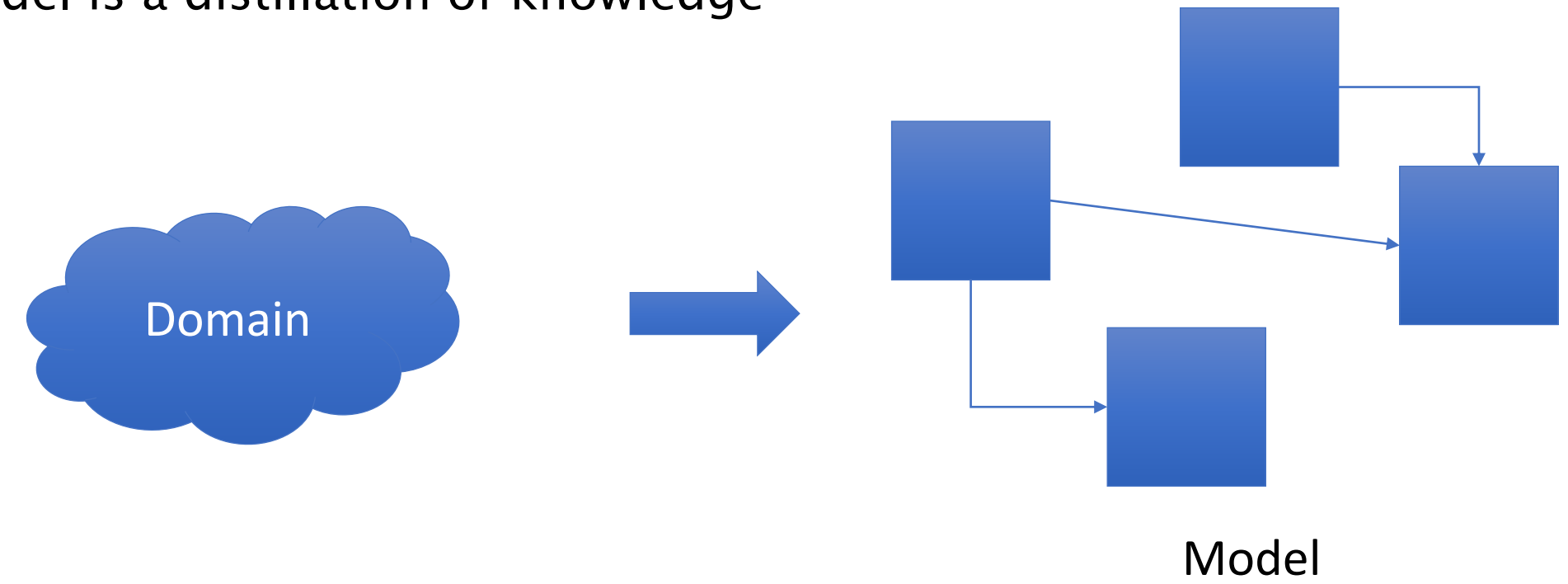
TACTICAL DESIGN

- The Tactical Design, is a set of technical resources used in the construction of your Domain Model
- These resources must be applied to work in a single Bounded Context

DOMAIN MODEL

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

- A model is a distillation of knowledge



STRATEGIC DESIGN

- Lays out techniques for recognizing, communicating, and choosing the limits of a model and its relationship to others

- Defines Bounded contexts, the Ubiquitous Language and the Context Maps

UBIQUITOUS LANGUAGE

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

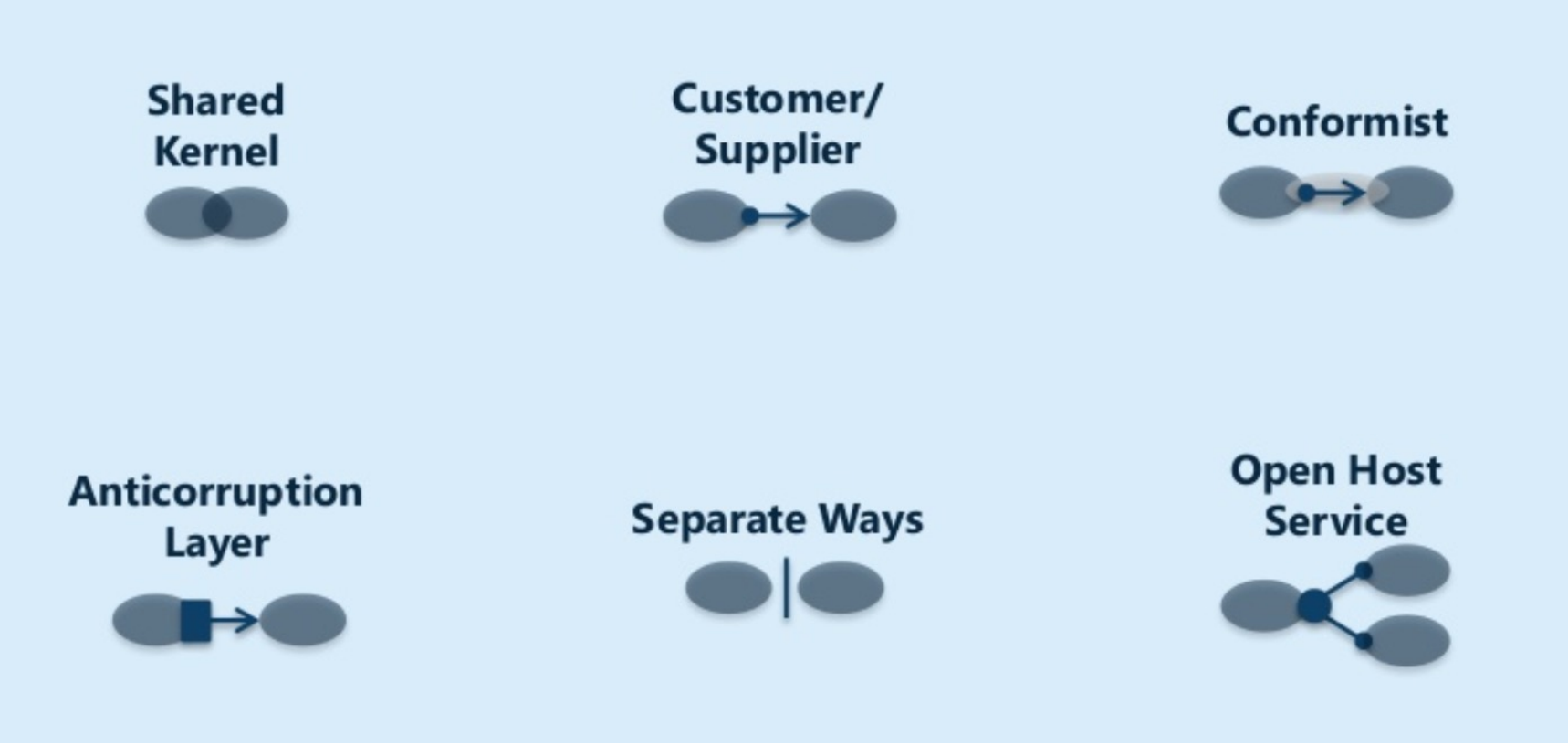
BOUNDED CONTEXT

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

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

CONTEXT MAP (1)

- Describe the points of contact between the models, outlining the explicit translation for any communication and highlighting any sharing



CONTEXT MAP (2)

- **Shared Kernel**
  - Designate with an explicit boundary some subset of the domain model that the teams agree to share. Keep this kernel small
- **Customer/Supplier Development**
  - Establish a clear customer/supplier relationship between the two teams, meaning downstream priorities factor into upstream planning
- **Conformist**
  - Eliminate the complexity of translation between bounded contexts by slavishly adhering to the model of the upstream team
- **Anticorruption Layer**
  - As a downstream client, create an isolating layer to provide your system with functionality of the upstream system in terms of your own domain model

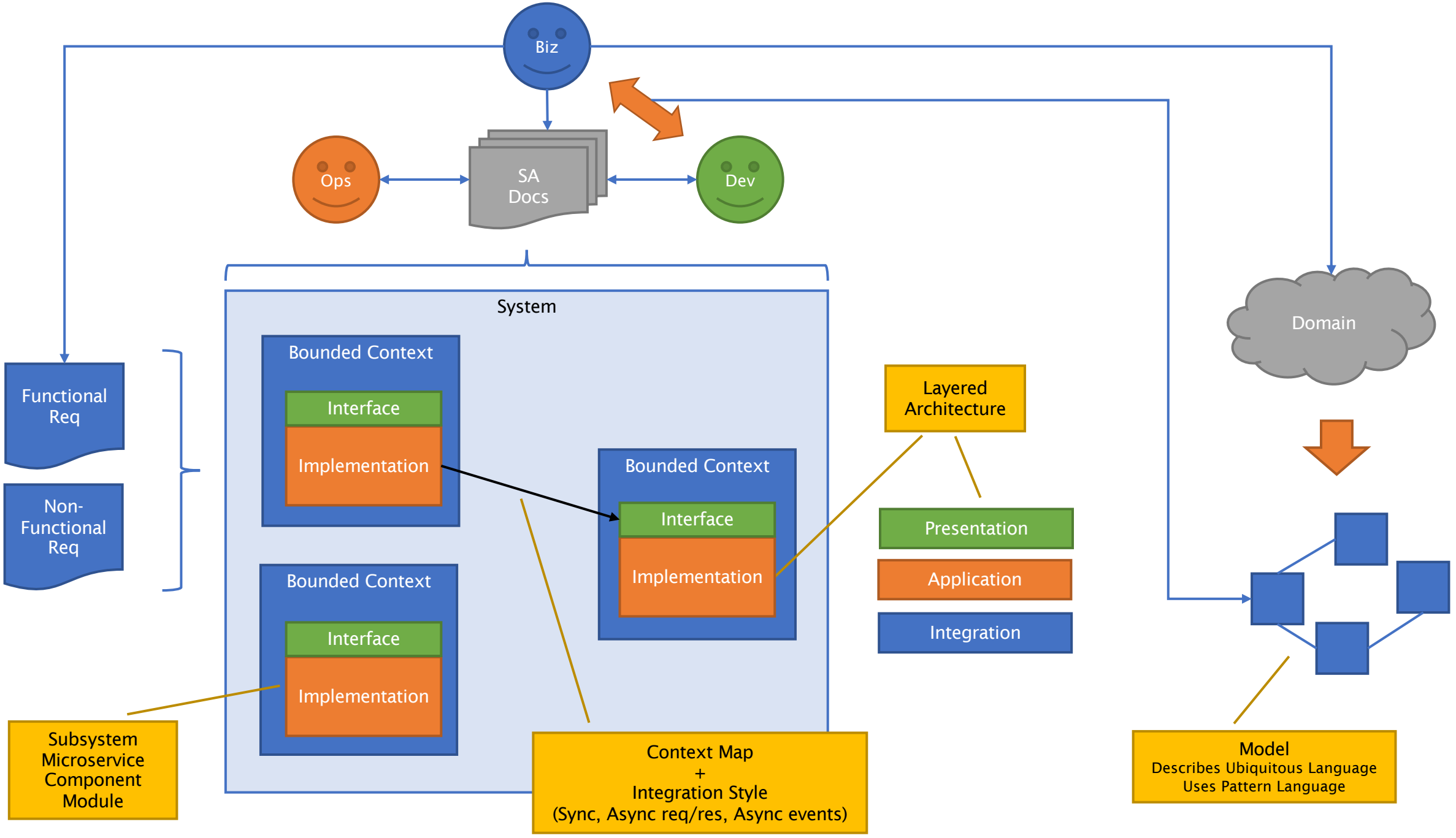
CONTEXT MAP (3)

- **Open-host Service**
  - Define a protocol that gives access to your subsystem as a set of services. Open the protocol so that all who need to integrate with you can use it
- **Published Language**
  - Use a well-documented shared language that can express the necessary domain information as a common medium of communication, translating as necessary into and out of that language
- **Separate Ways**
  - Declare a bounded context to have no connection to the others at all, allowing developers to find simple, specialized solutions within this small scope

CONCLUSION

- DDD acts as a stabilizing factor in complex software projects
- The ubiquitous language ensures a common domain understanding
- The domain model is the basis for development and is strongly coupled with the code
- **Strategic design with bounded contexts helps to define strongly encapsulated subsystems (aka Microservices) and their dependencies**

3. Software Architecture Distilled


Biz
Ops
SA
Docs
Dev
System
Bounded Context
Interface
Implementation
Bounded Context
Interface
Implementation
Bounded Context
Interface
Implementation
Functional
Req
Non-
Functional
Req
Layered
Architecture
Presentation
Application
Integration
Domain
Subsystem
Microservice
Component
Module
Context Map
+
Integration Style
(Sync, Async req/res, Async events)
Model
Describes Ubiquitous Language
Uses Pattern Language