# From **Software Architecture** to **N-tiers Architectures**

**Philippe Collet**

with many slides from

Sébastien Mosser

POLYTECH NICE SOPHIA | UNIVERSITÉ CÔTE D'AZUR



pictures: sxc.hu

# Software Architecture **Definition**

---

> The **structure** of the system, which comprise **software elements**, externally **visible properties** of those elements, and the **relationships** among them.

[SAiP]

# Architecture versus Design?

---

## Architecture is a subset of design

## "External" design

# Software Architecture **Objectives**

- It has the **functionality** required by the customer
- It is safely buildable on the **required schedule**
- It **performs adequately**
- It is **reliable**
- It is **usable** and **safe to use**
- It is **affordable**

# Architectural

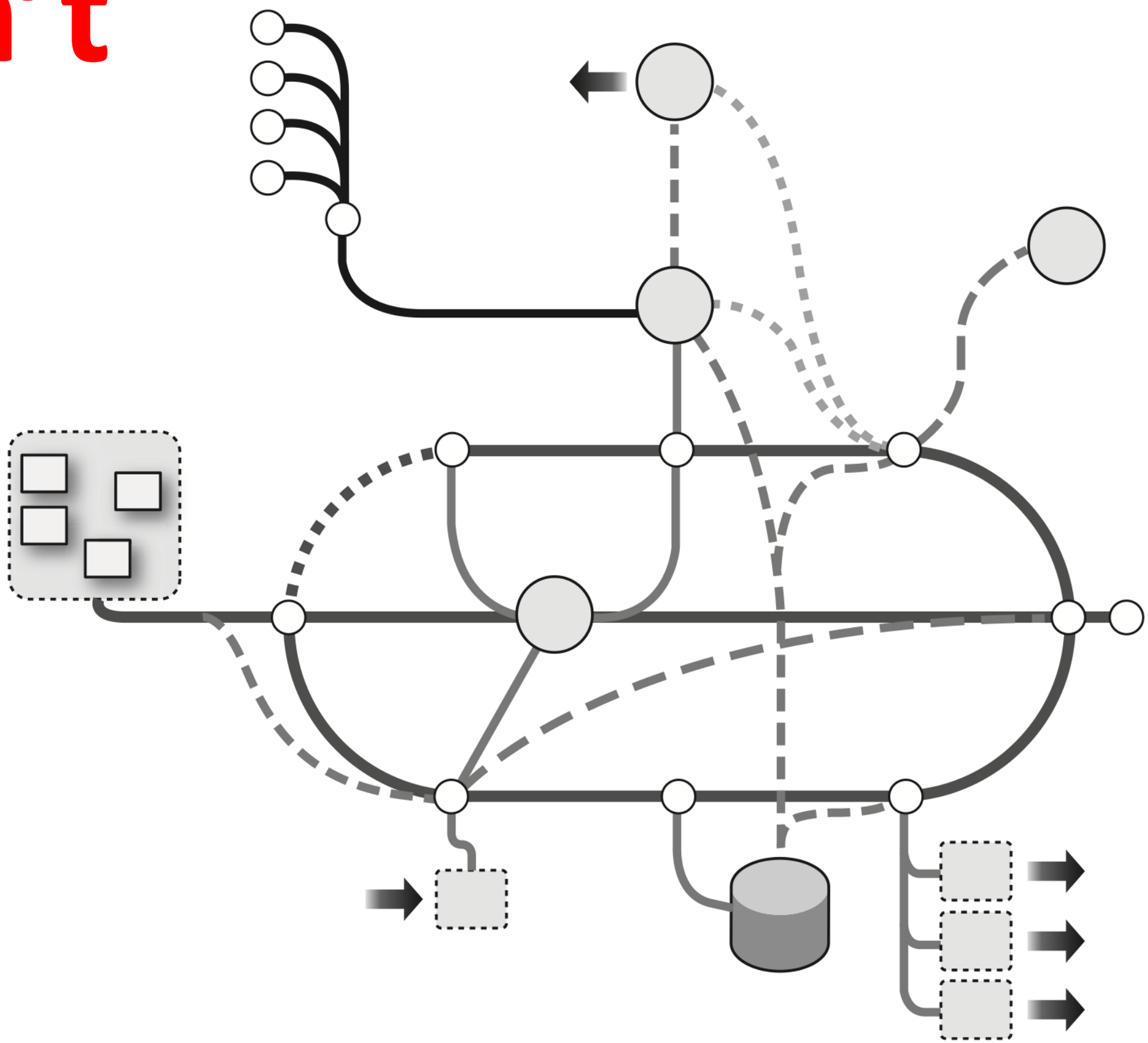
# rule of thumb

“

Software **architecture** is **not set in stone**.

**Change** if you need it.

# Don’t



“

A **fuzzy architecture**

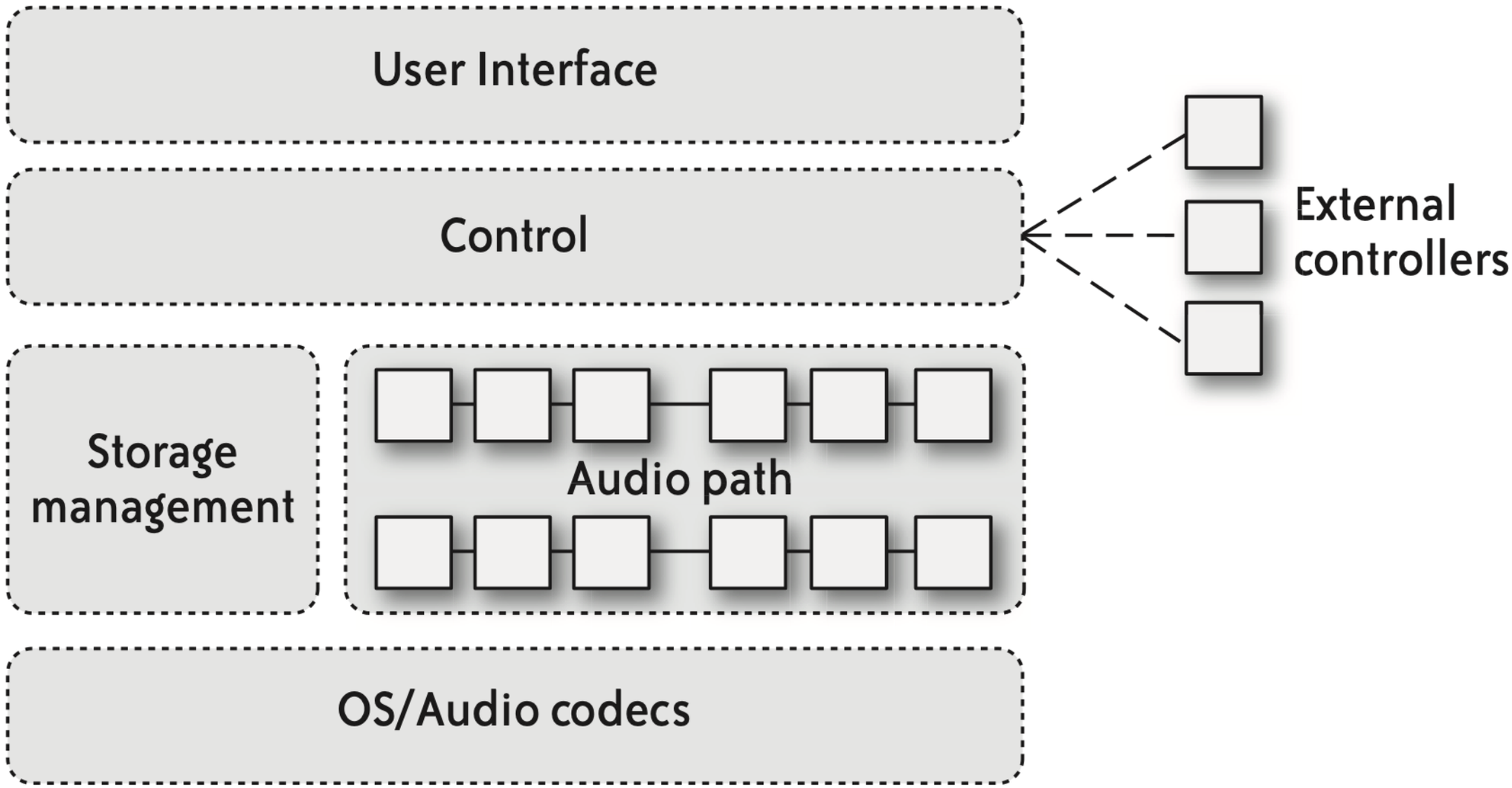
leads to **individual** code,

**duplication** of code and effort.

“

**Bad** architectural **design**
**leads to further**
**bad** architectural **design**.

# Do


User Interface
Control
Storage management
Audio path
OS/Audio codecs
External controllers

“

A **clear architectural** design leads to a **consistent system**.

# **Cohesion** versus **Coupling**

---

**Cohesion**:

"how related functionality is gathered together".

**Coupling**:

"Measurement of interdependency between modules".

?

Low cohesion, **Strong** cohesion,

**Str** coupling **Low** coupling

# Coupling dimensions

**Functional**

**Location**

**Temporal**

**Structural**

# Object

**Fine**-grained

**Encapsulate** logic ⊕ data

**Whitebox** reuse

**Stateful** behavior

**Local** (often)

# Component

**Blackbox**

reuse

Behavior

**abstraction**

**Structural** ↘

**Functional** ↘

**Coarse**-grained

(usually) hosted in a

**container**

**Remote** (often)

# Service

**Blackbox**

reuse

Behavior

**abstraction**

**Coarse**-grained

Hosted in a

**container**

**Remote** (often)

# Fail!

# ervice = Component

C.line (c

# Services

target: customer

are **business**-

oriented

# Compone

# nts

are (more)

**code**-

oriented

target:develope

Location

↑

Temporal

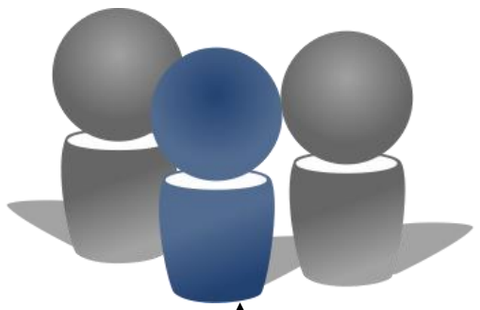
↘

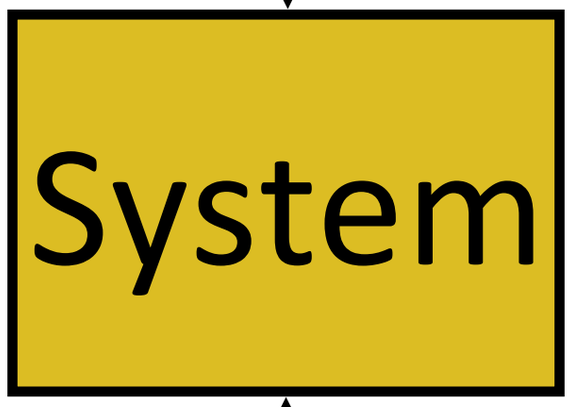
Structural ↘↘

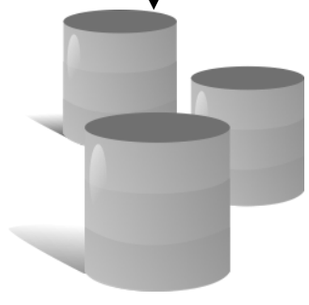
Functional ↘↘

# Layered Architectures

2



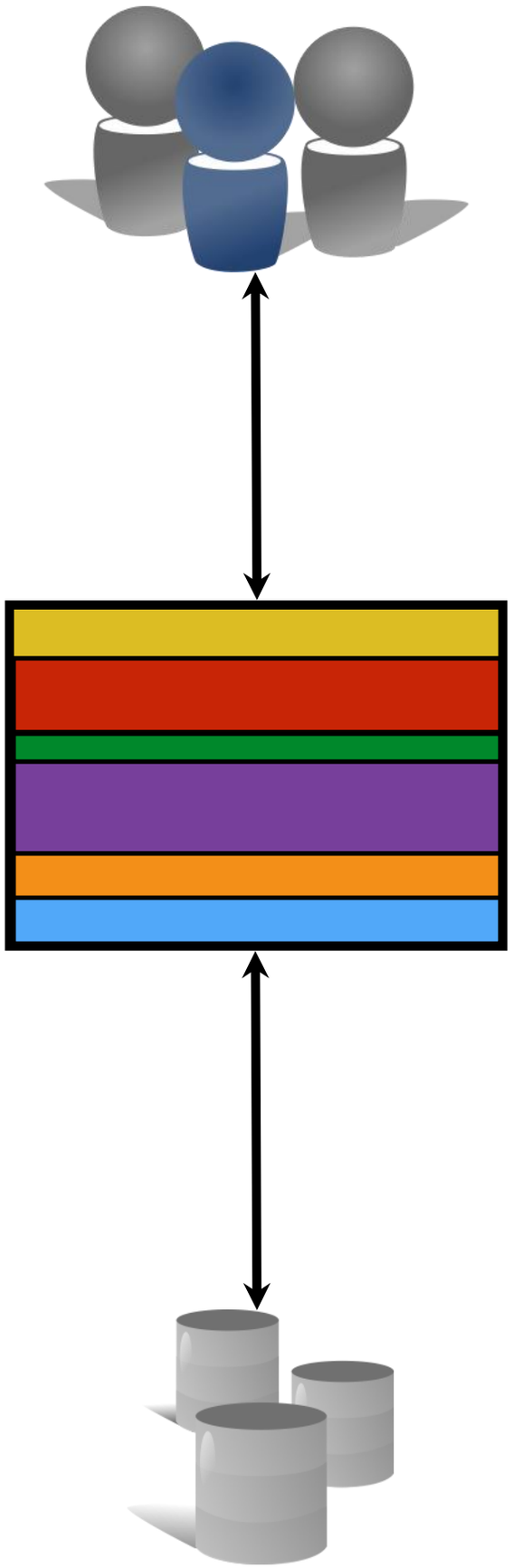

System



# Layers

support

# modularity

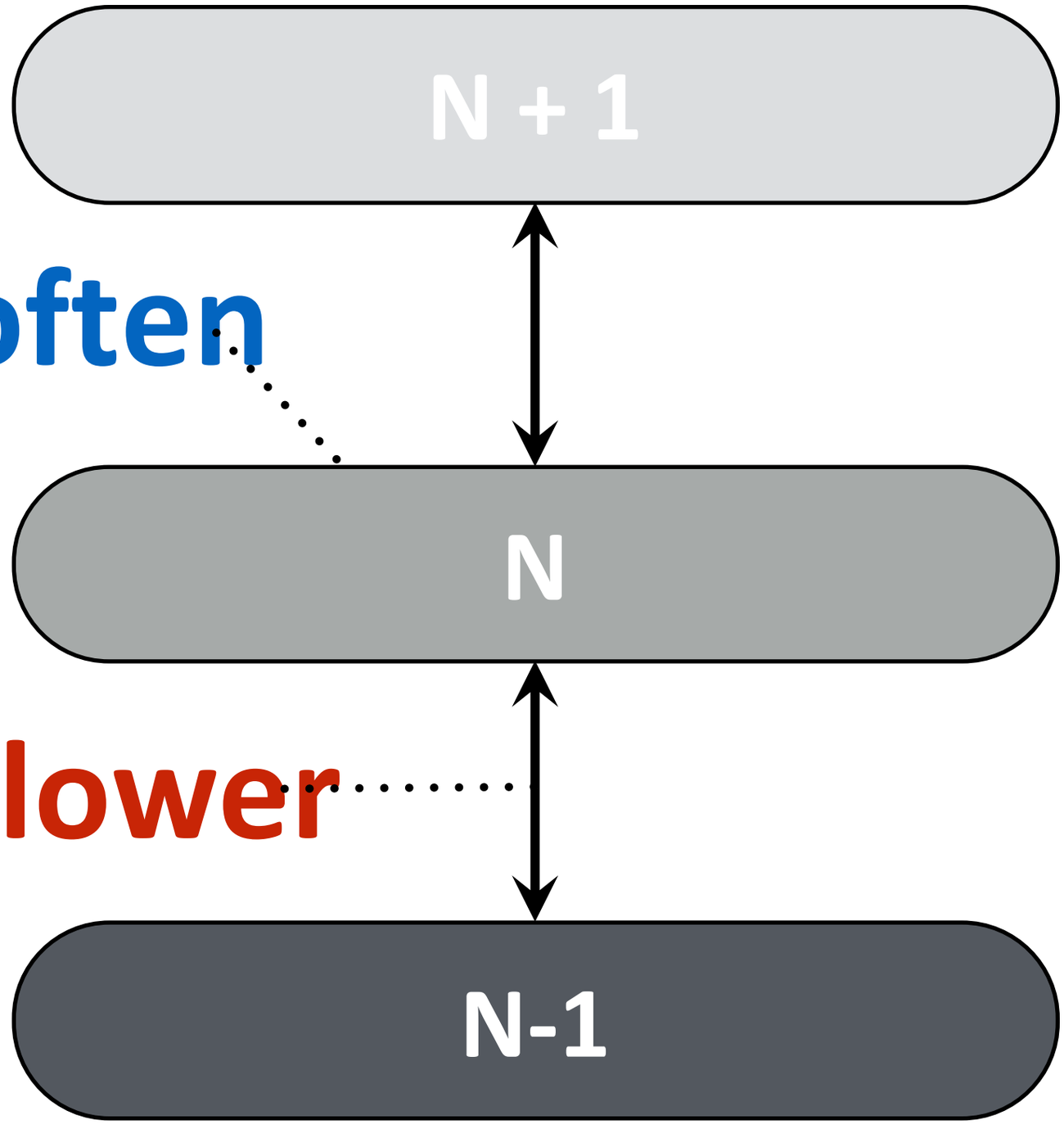


# “Ma-gni-fique”

thanks to Clèm for the reference

# The rule of 10

**10 times often**



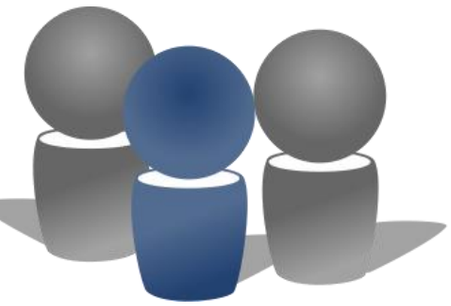

**10 times slower**

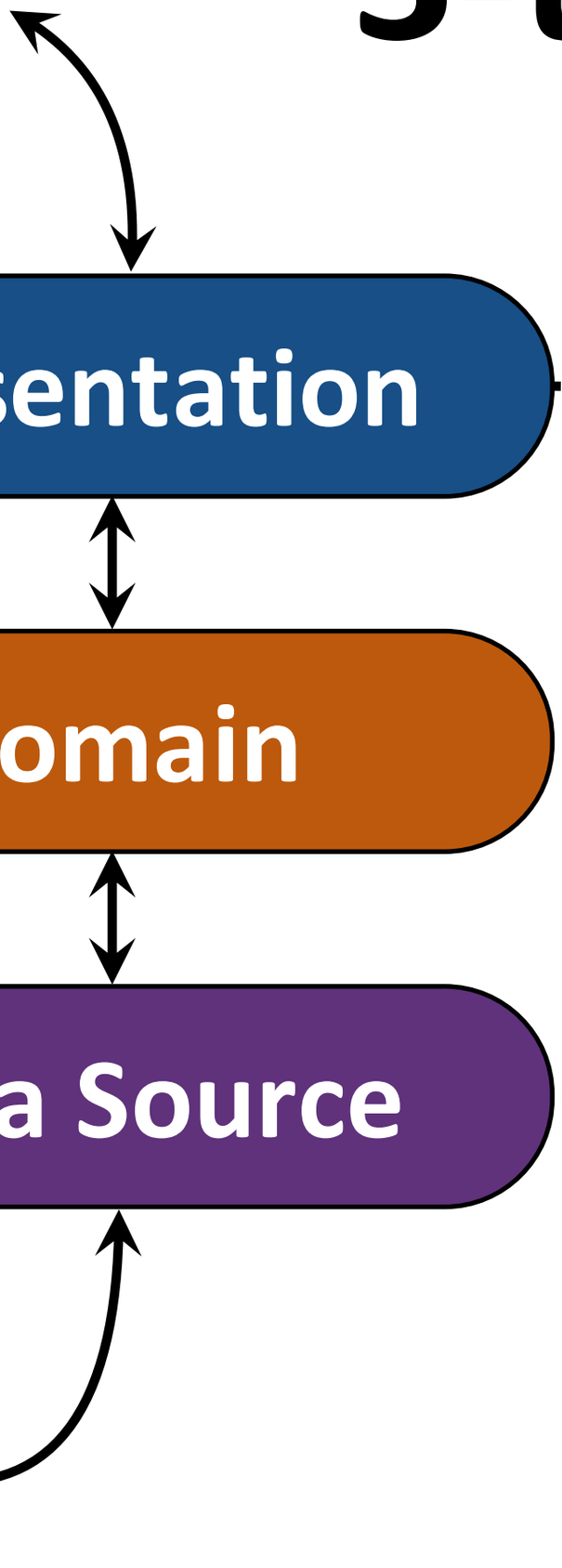
In theory, **N-tiers** architecture

**In practice, N = 3**

(or 5)

# 3-tiers architecture

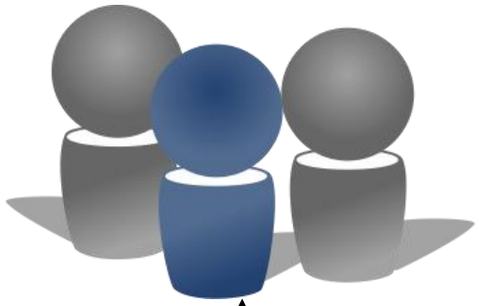


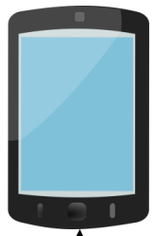



Handle users

Business functionality

Handle data storage





Presentation

Mob. App

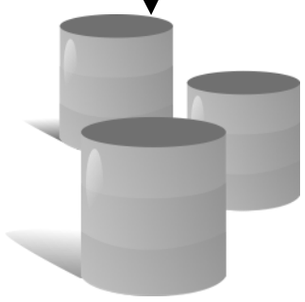
Facebook app

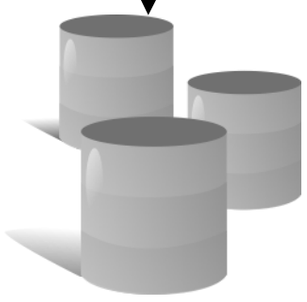
Domain

Social-Net

Data Source

NoSQL DB





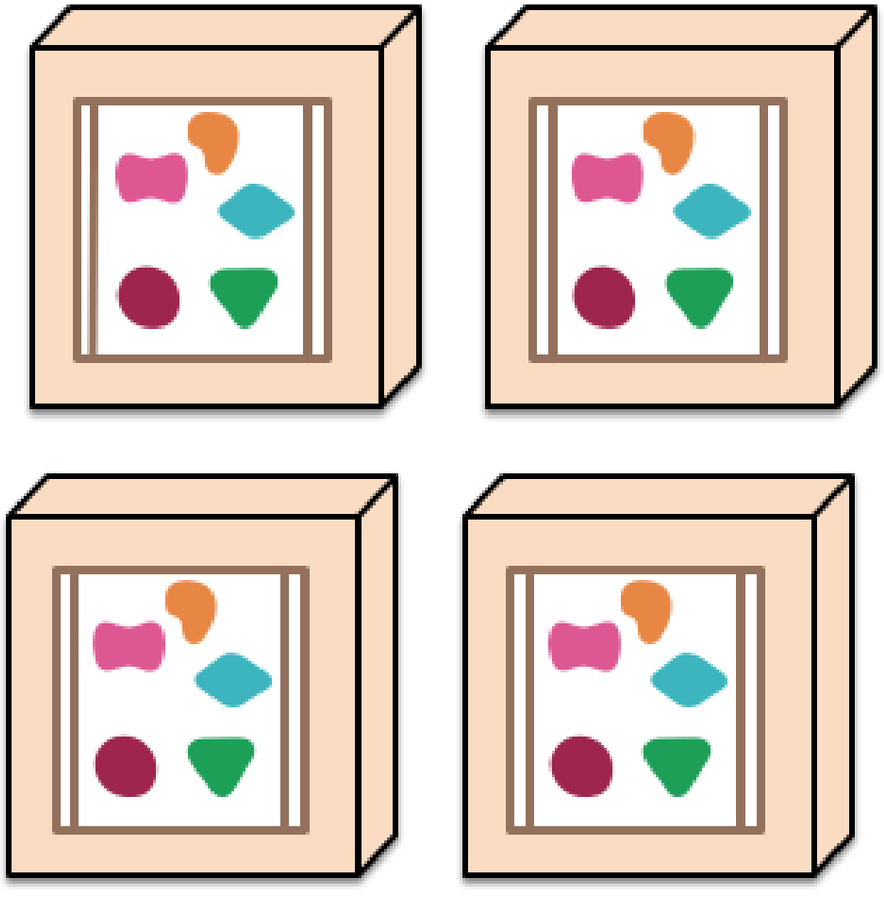
# Evolution


YOU WANT TO DO
LAYERED MODULAR MONOLITHS
WE WANT TO
DO MICROSERVICES
WE ARE
NOT THE SAME
imgflip.com

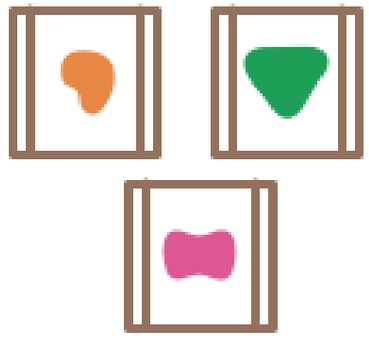
# Monolith vs microservices - simplified

*A monolithic application puts all its functionality into a single process...*

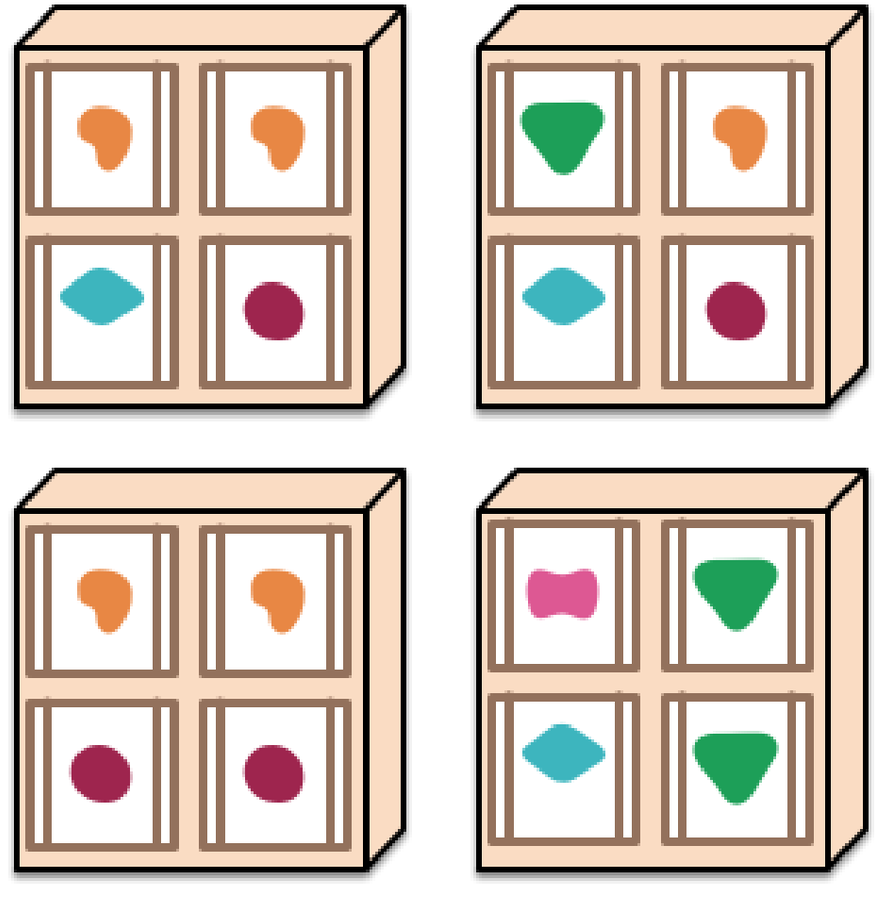
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*



*... and scales by distributing these services across servers, replicating as needed.*





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

# Microservices are complex because they cause **problems**

---

Distributed system problems

Design for failures is needed

Data consistency problems

Distributed transactions do not scale

Eventual Consistency

SAGA Pattern (compensation event)

# Microservices are complex because they cause **problems**

---

Communications become complex

Async APIs cause integration bugs

Operational Overhead

Debugging and testing are way more complex

Distributed monitoring is needed

Senior Fellow Google
Designer of MapReduce, Google File System, Bigtable et Spanner

Senior Director of Engineering at Google Cloud (former techinal director of Microsoft Azure)

Principal Software Engineer at Google (former lead of JVM implementation at Apple)

Fellow et Chief Technologist for AI Infrastructure at Google

# Towards Modern Development of Cloud Applications

Sanjay Ghemawat, Robert Grandl, Srdjan Petrovic, Michael Whittaker
Parveen Patel, Ivan Posva, Amin Vahdat
Google

Research paper 2023

**Abstract**

When writing a distributed application, conventional wisdom says to split your application into separate services that can be rolled out independently. This approach is well-intentioned, but a microservices-based architecture like this often backfires, introducing challenges that counteract the benefits the architecture tries to achieve. Fundamentally, this is because microservices conflate logical boundaries (how code is written) with physical boundaries (how code is deployed). In this paper, we propose a different programming methodology that decouples the two in order to solve these challenges. With our approach, developers write their applications as logical monoliths, offload the decisions of how to distribute and run applications to an automated runtime, and deploy applications atomically. Our prototype implementation reduces application latency by up to 15× and reduces cost by up to 9× compared to the status quo.

like [5, 6, 23, 30]. The prevailing wisdom when using these technologies is to manually split your application into separate microservices that can be rolled out independently.

Via an internal survey of various infrastructure teams, we have found that most developers split their applications into multiple binaries for one of the following reasons: (1) It improves *performance*. Separate binaries can be scaled independently, leading to better resource utilization. (2) It improves *fault tolerance*. A crash in one microservice doesn't bring down other microservices, limiting the blast radius of bugs. (3) It improves *abstraction boundaries*. Microservices require clear and explicit APIs, and the chance of code entanglement is severely minimized. (4) It allows for flexible *rollouts*. Different binaries can be released at different rates, leading to more agile code upgrades.

However, splitting applications into independently deployable microservices is not without its challenges, some of which directly contradict the benefits.

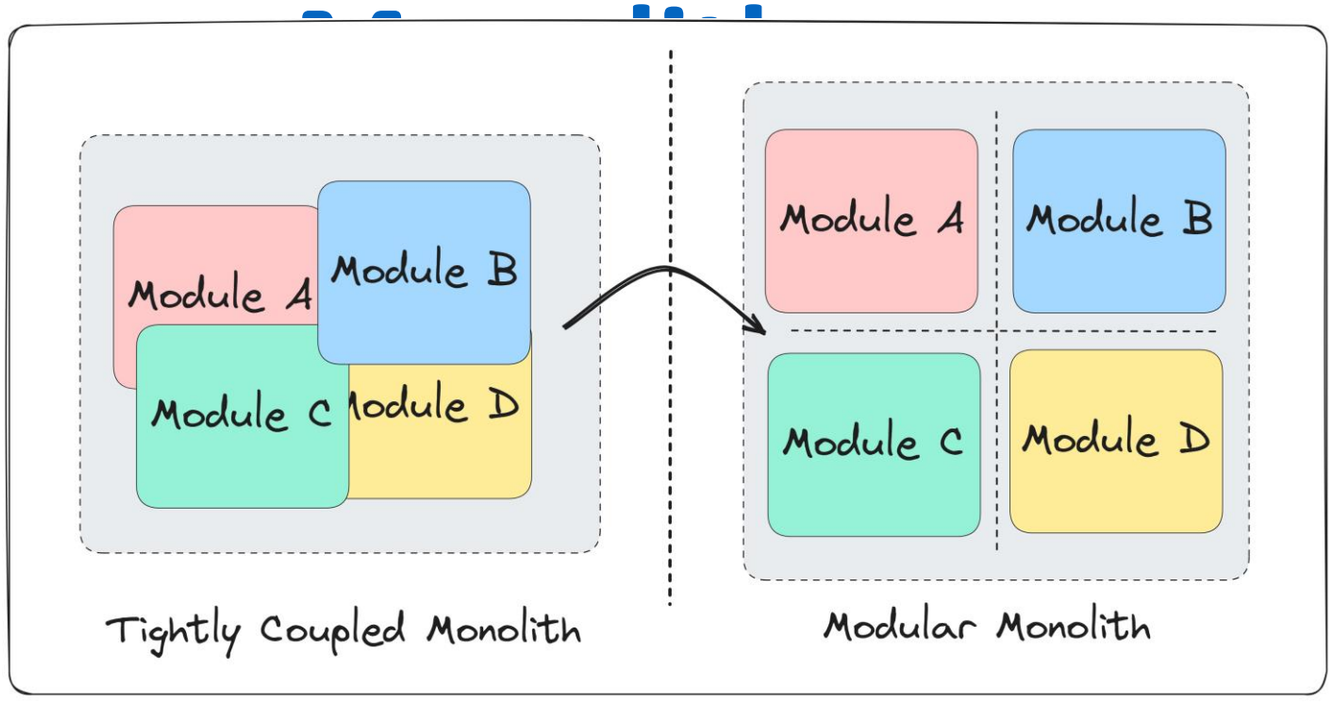
O'REILLY®

# Monolith to Microservices

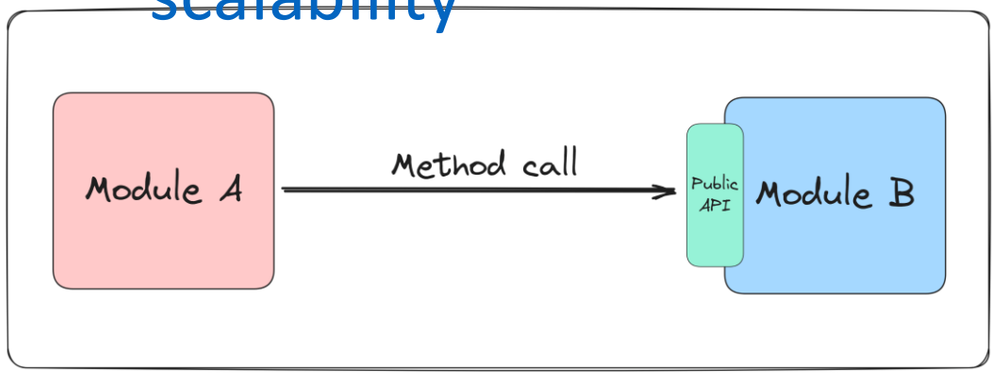
Evolutionary Patterns to Transform Your Monolith

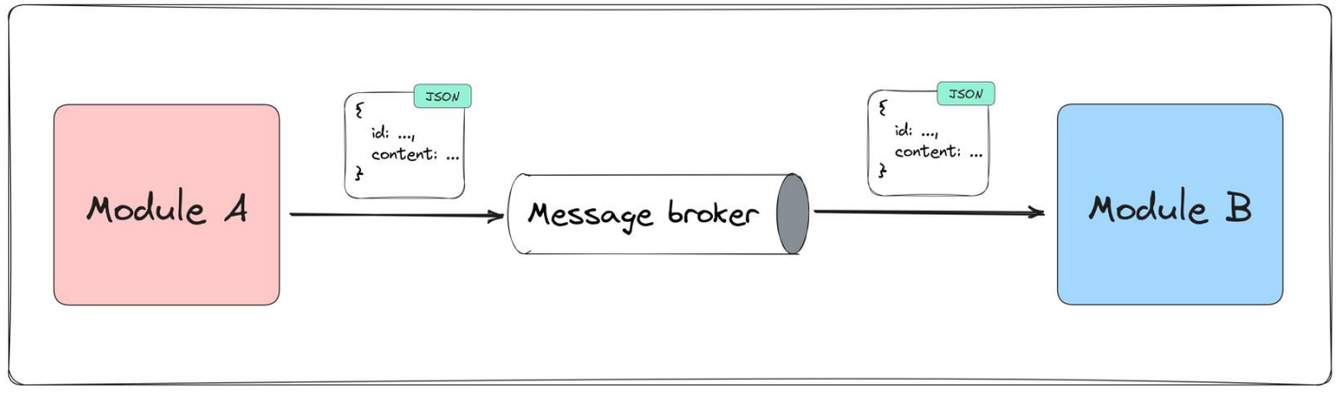

Sam Newman

# Modular




## 💡can be applied progressively and on parts that REALLY needs scalability




https://www.milanjovanovic.tech/blog/modular-monolith-communication-patterns

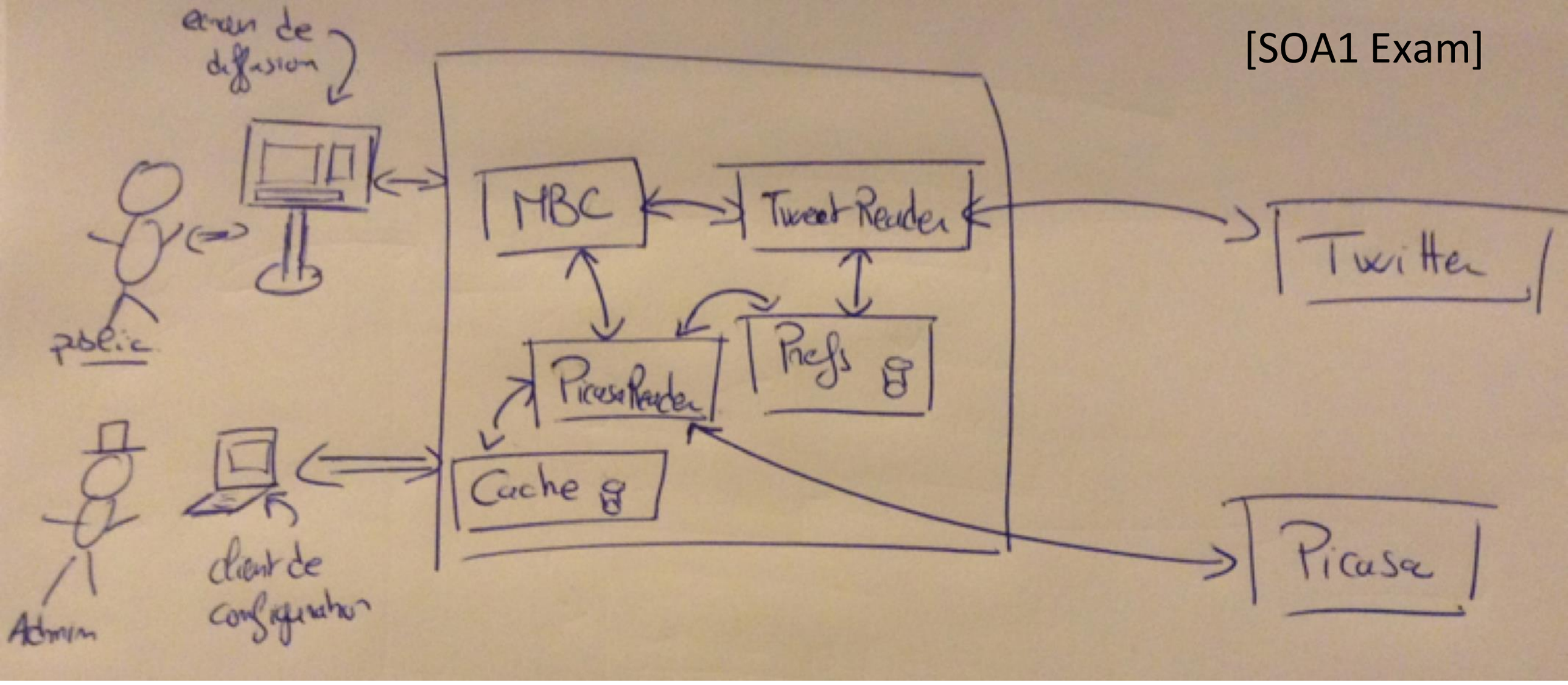
# Support from the


The **UML** is just a

**standard syntax**

for modeling

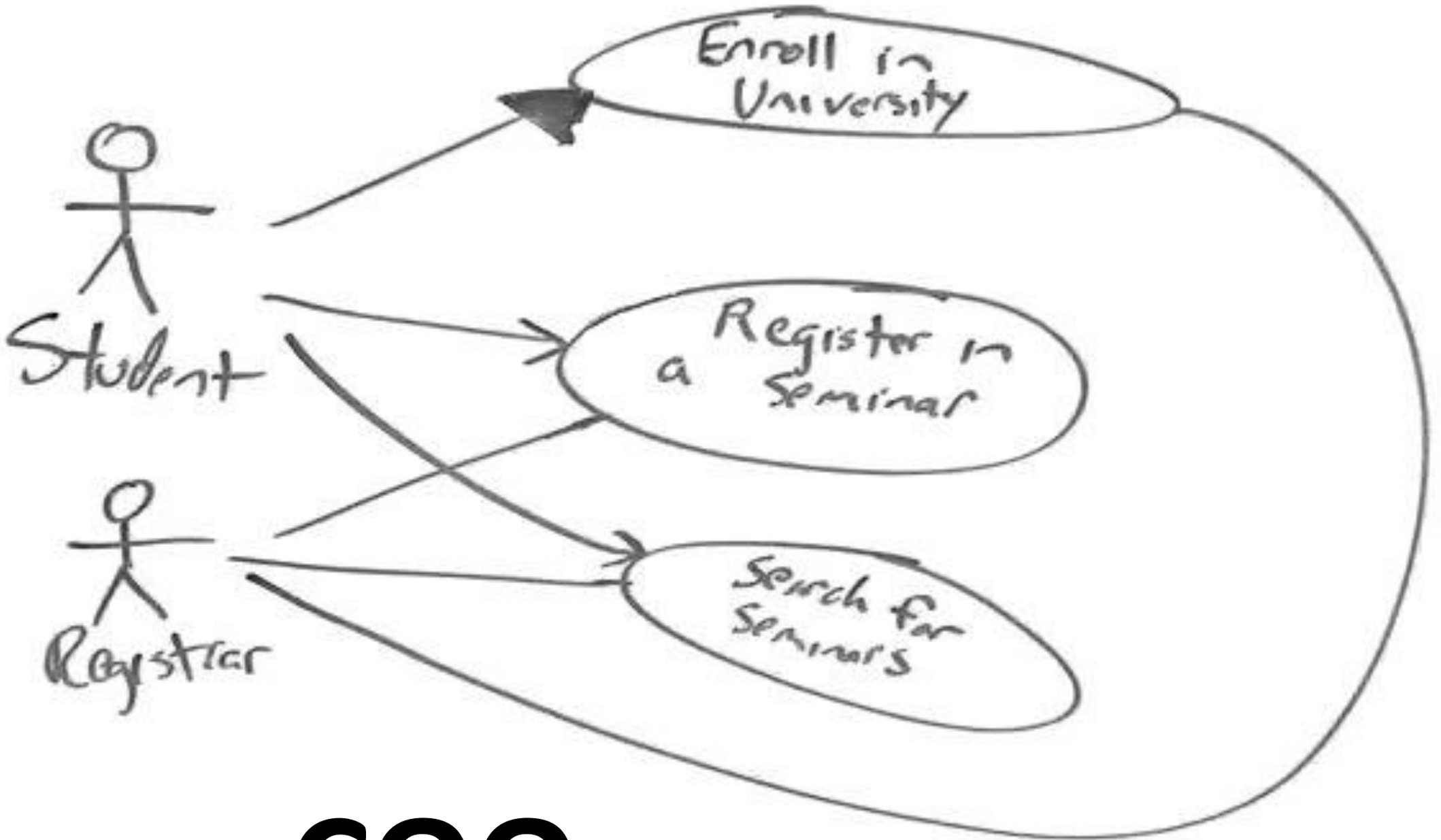
[SOA1 Exam]




One can design a Software Architecture

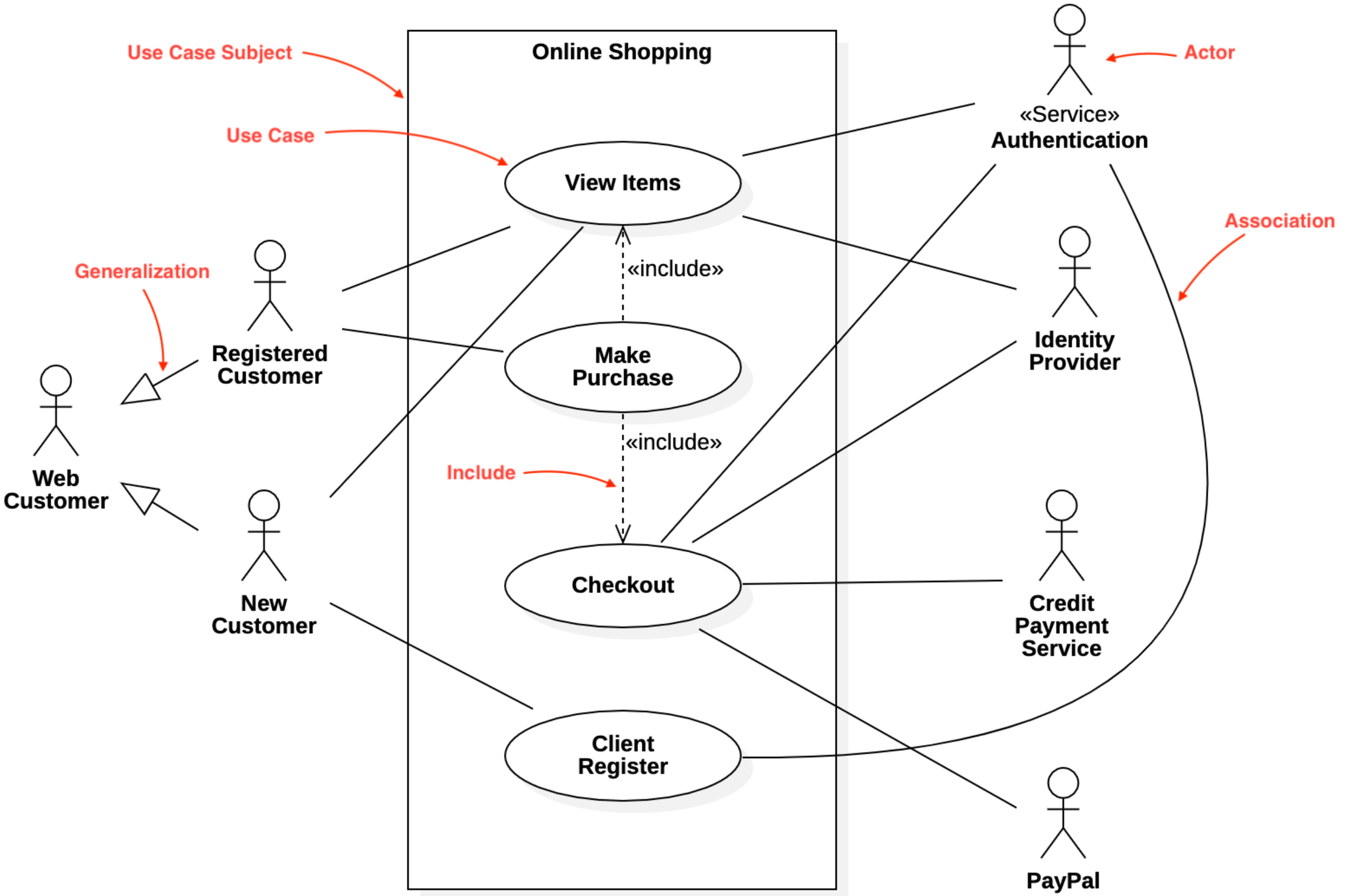
# **without the UML**

# Use case diagram




COO

# Use case diagram

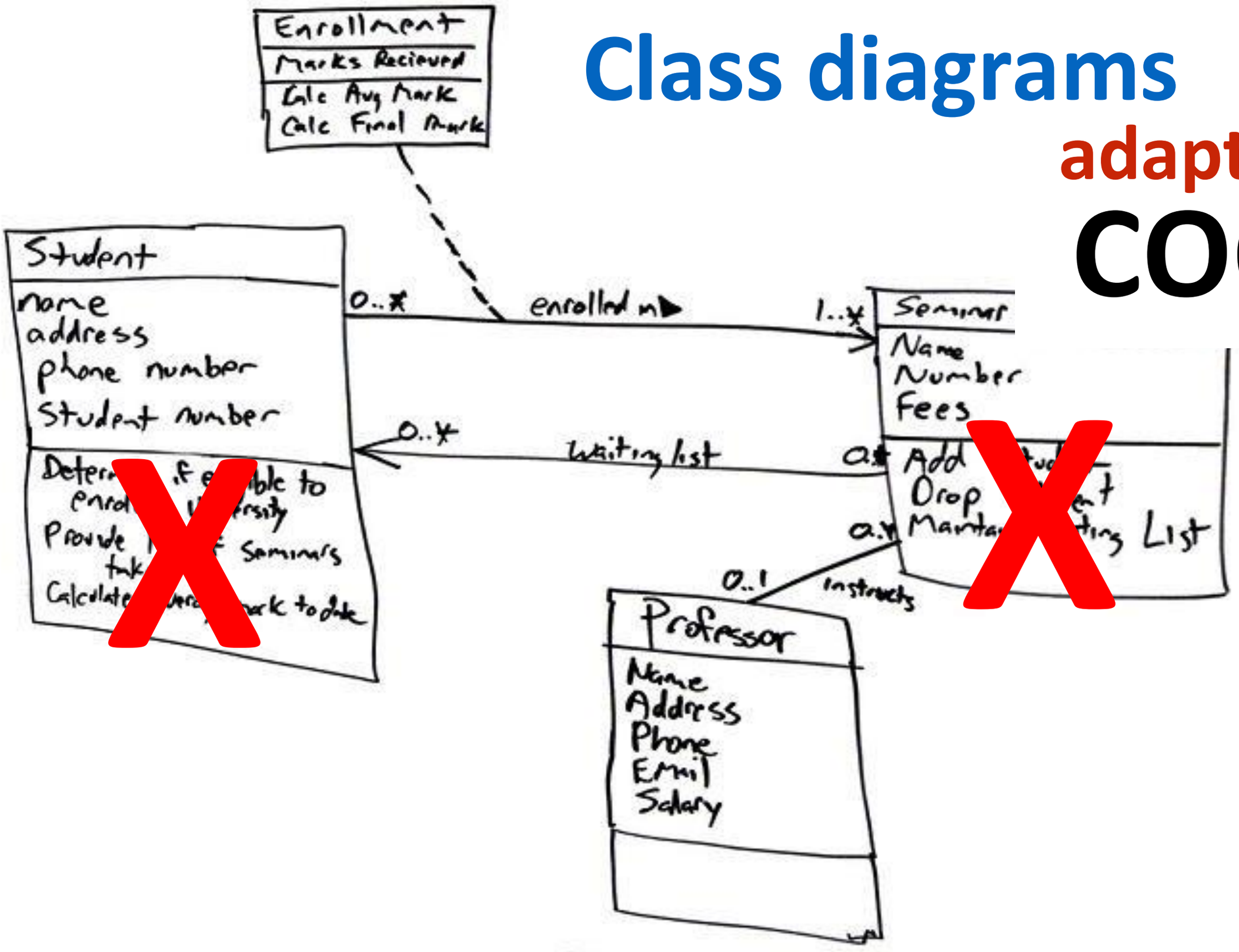



https://docs.staruml.io/working-with-uml-diagrams/use-case-diagram

# Class diagrams

## adapted

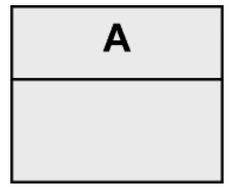
# COO




**Business** objects => **no business methods**

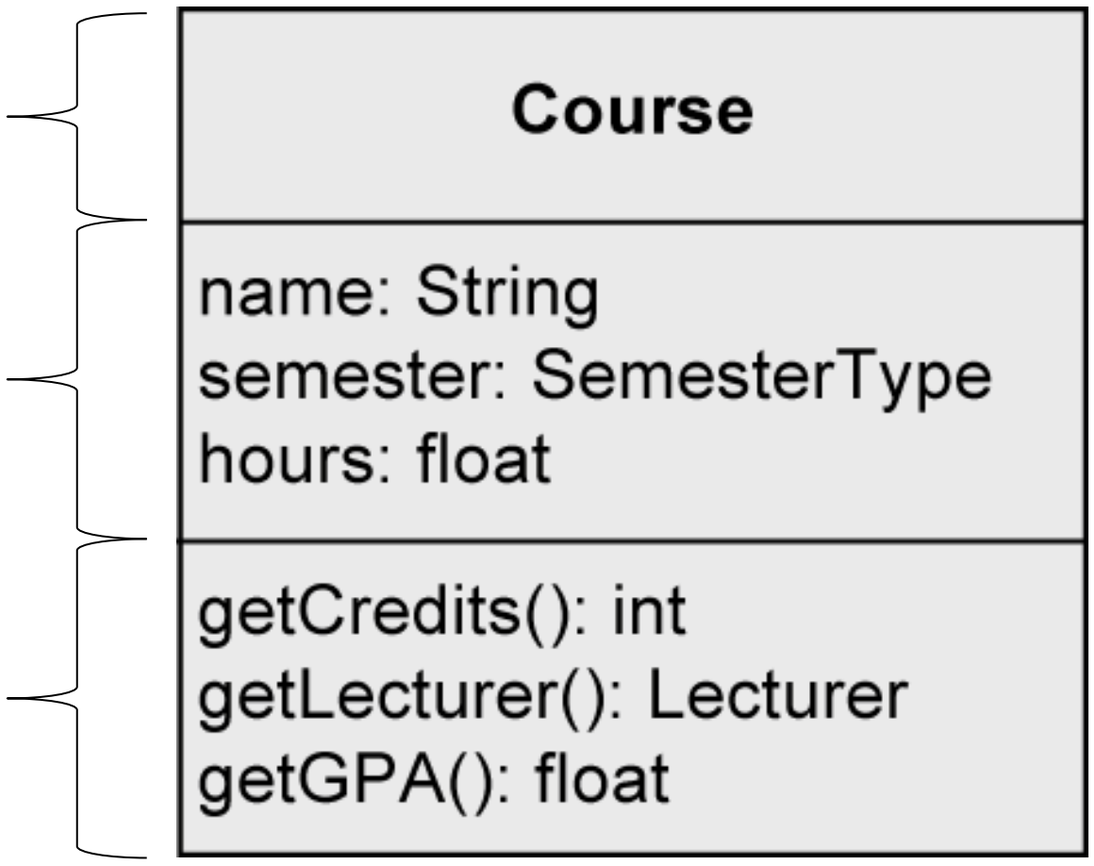
/www.agilemodeling.com/artifacts/classDiagram.htm

# Class



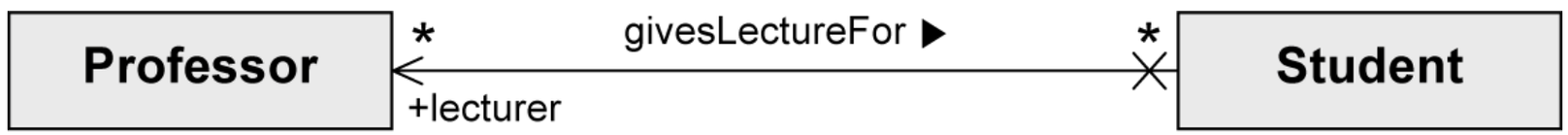

Attributes

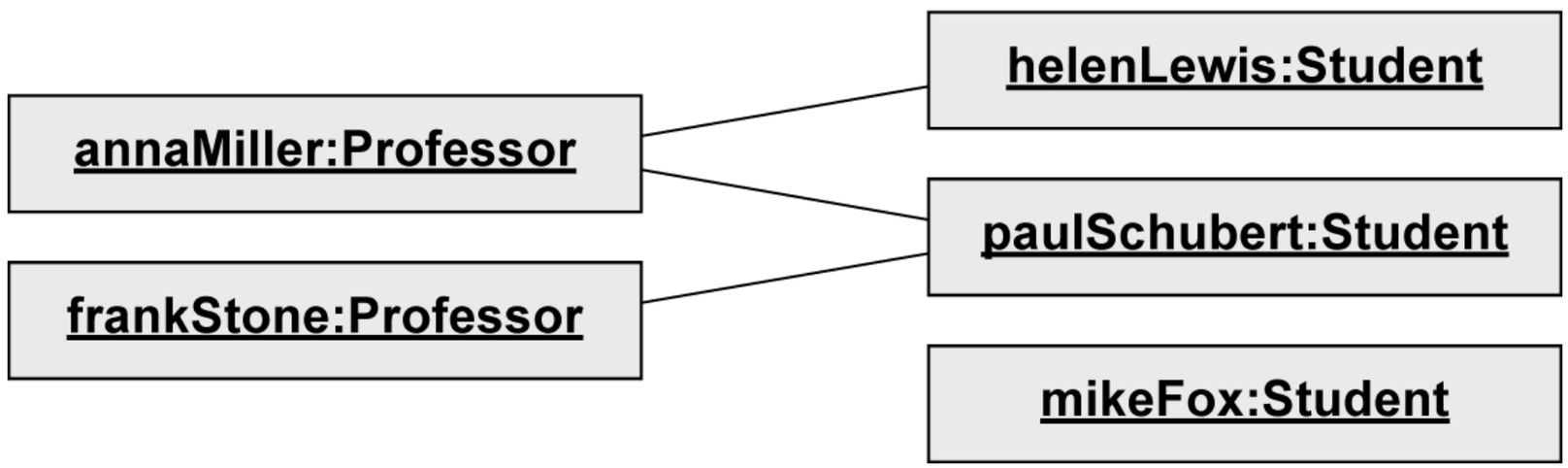
Operations



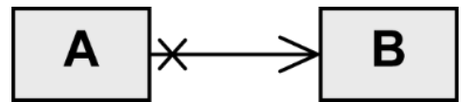
# Association

- Models possible relationships between instances of classes

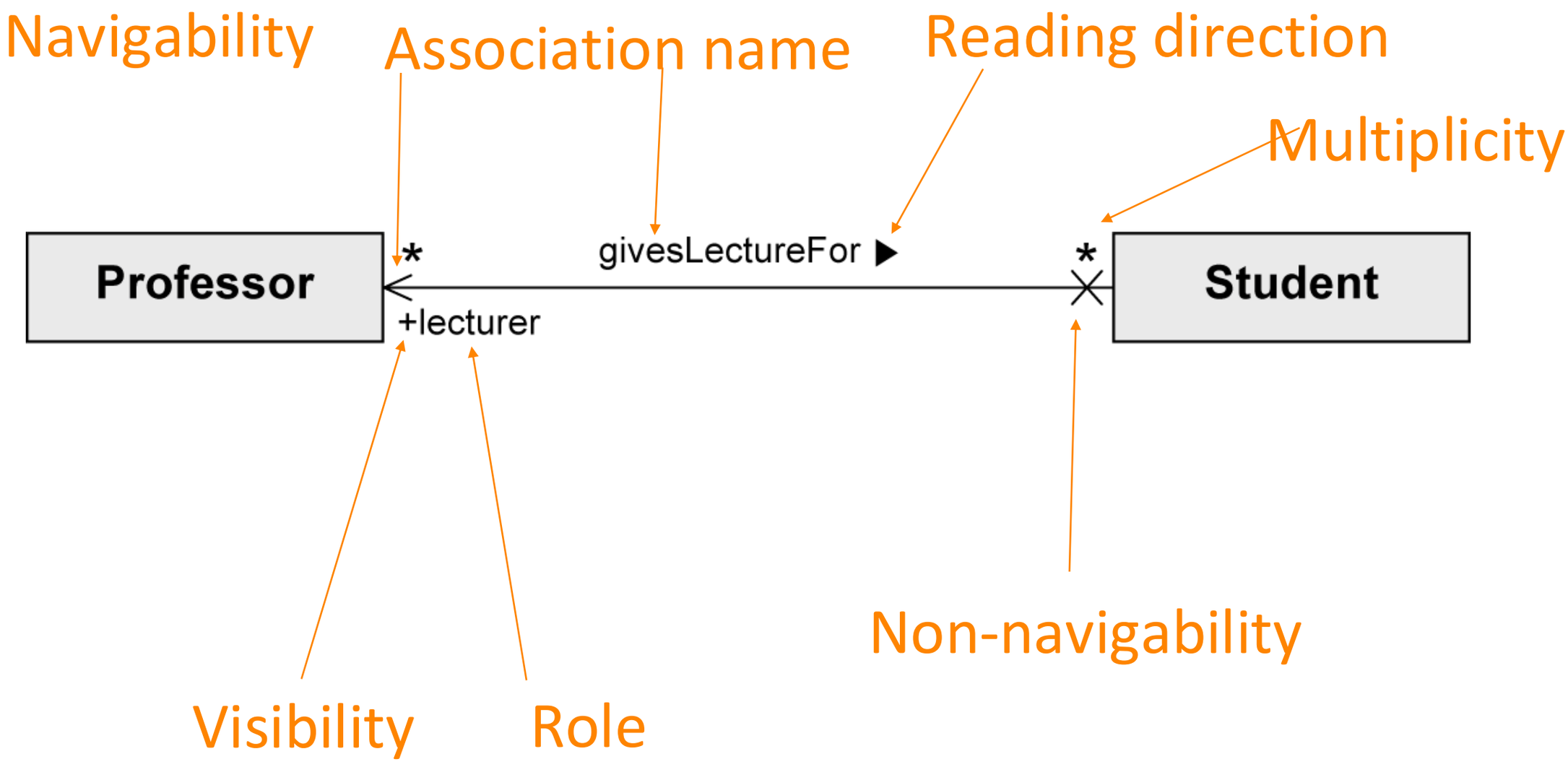


# Binary Association

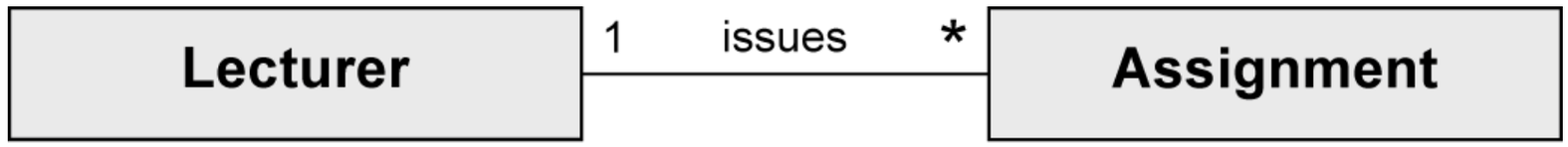


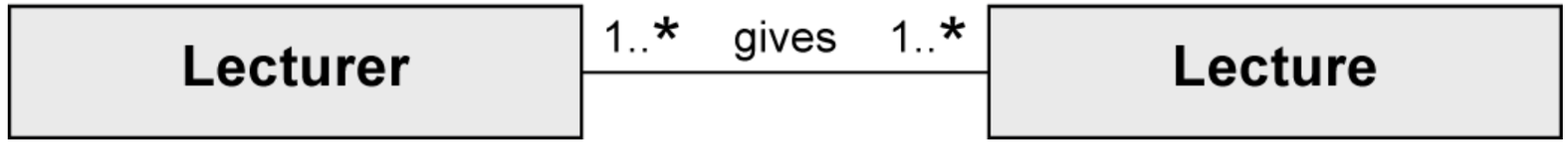

- Connects instances of two classes with one another

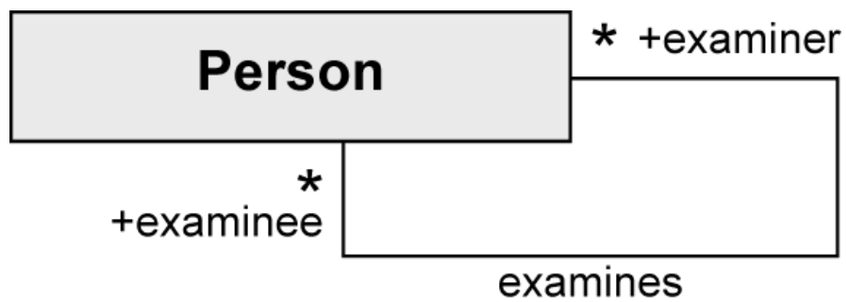


# Binary Association – Multiplicity and Role

- Multiplicity: Number of objects that may be associated with exactly one object of the opposite side

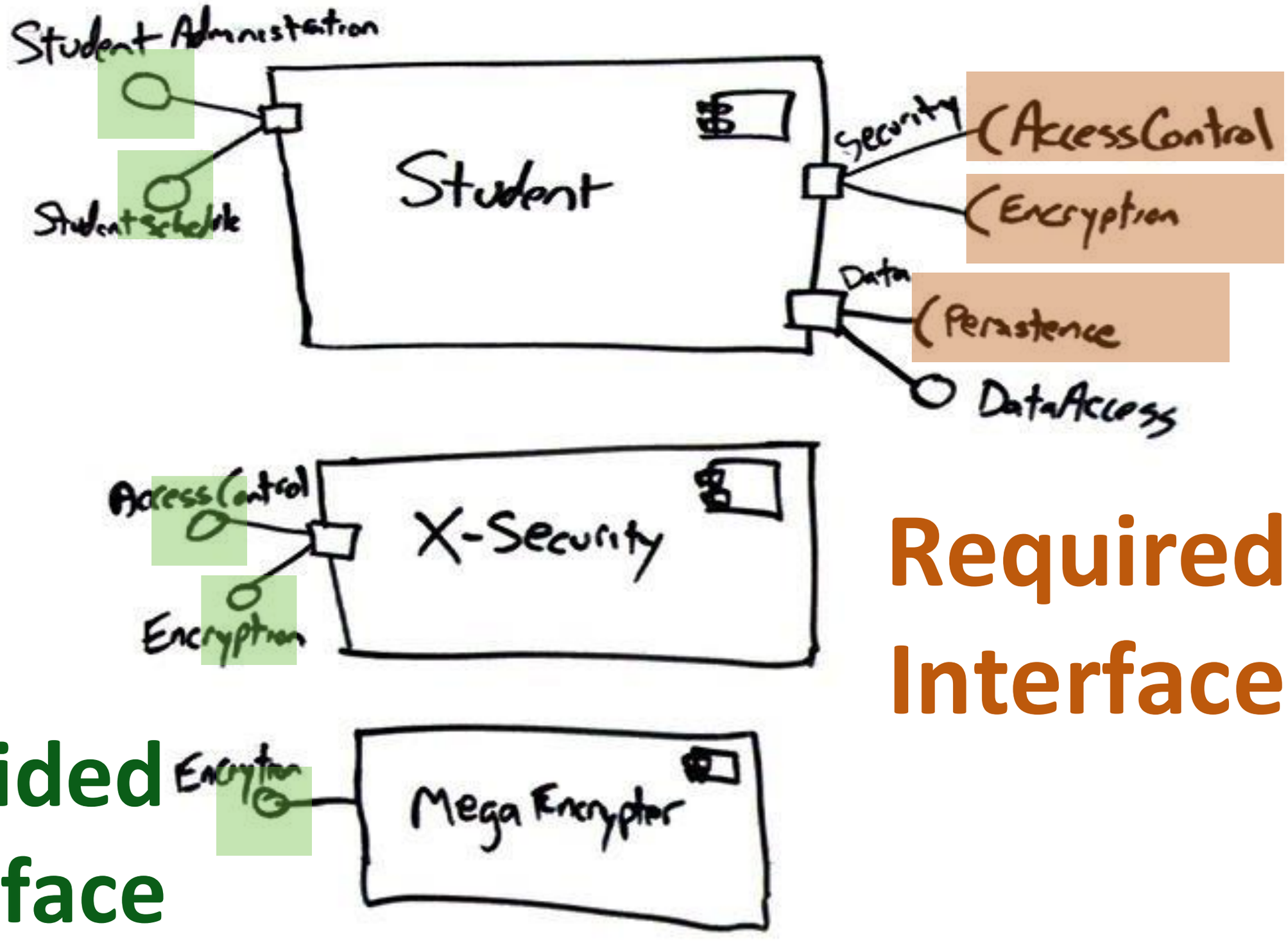



- Role: describes the way in which an object is involved in an association relationship

# Component diagrams

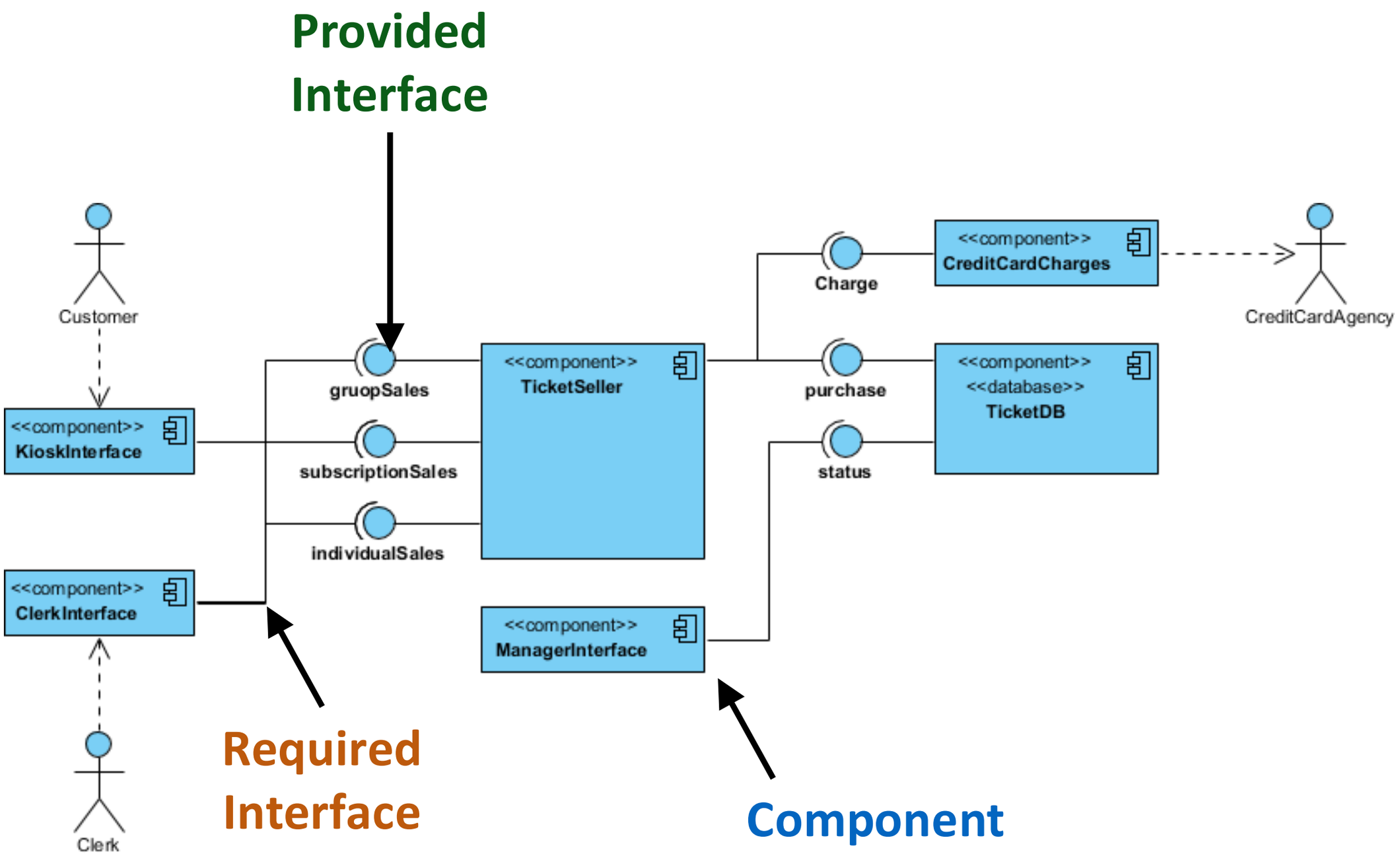



**Required Interface**

**Provided Interface**

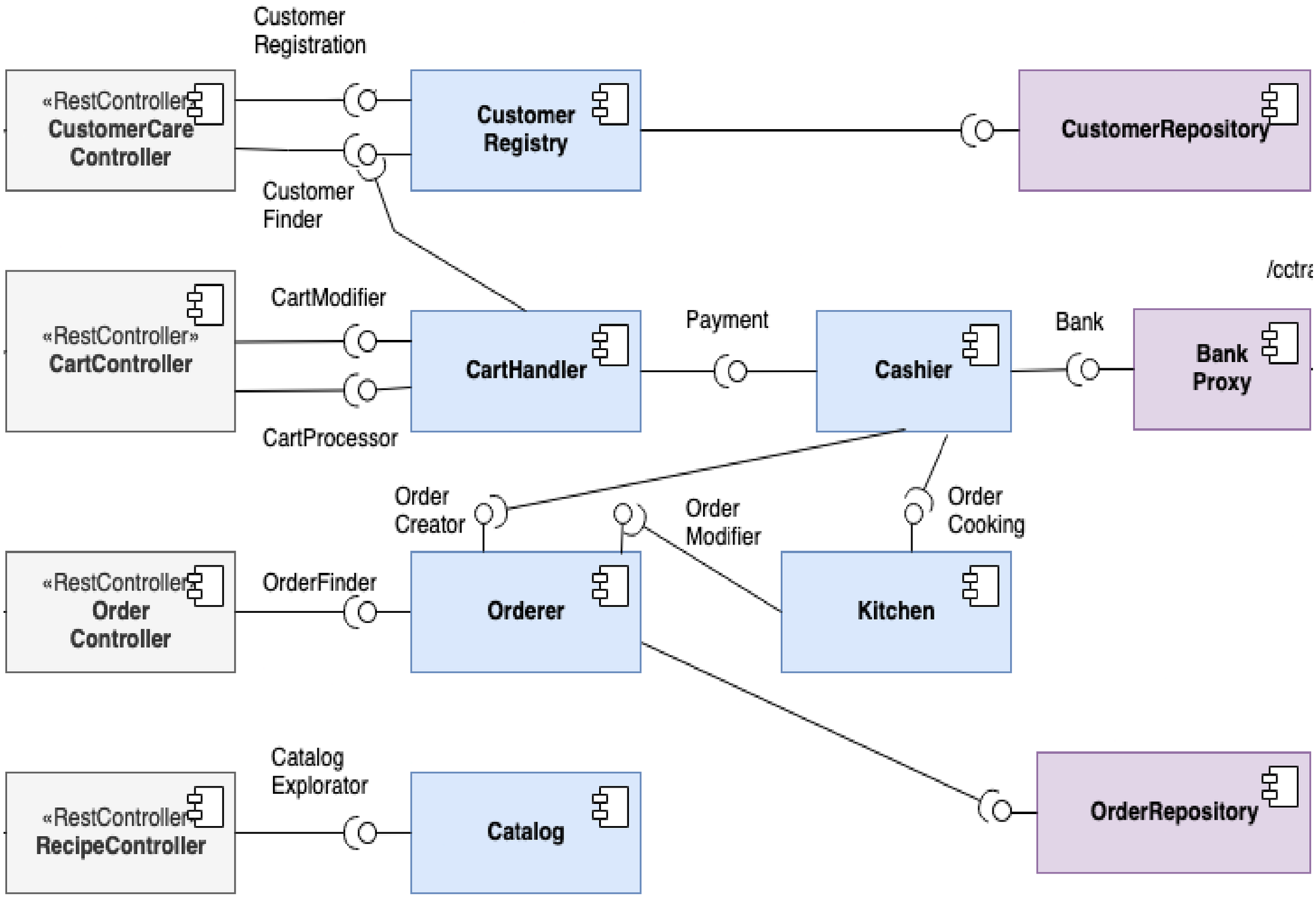
www.agilemodeling.com/artifacts/componentDiagram.htm

# Component diagrams




http://uml-questions.blogspot.com/p/component-diagram.html

# Component diagrams



# **I & D** of the **SOLID** principles

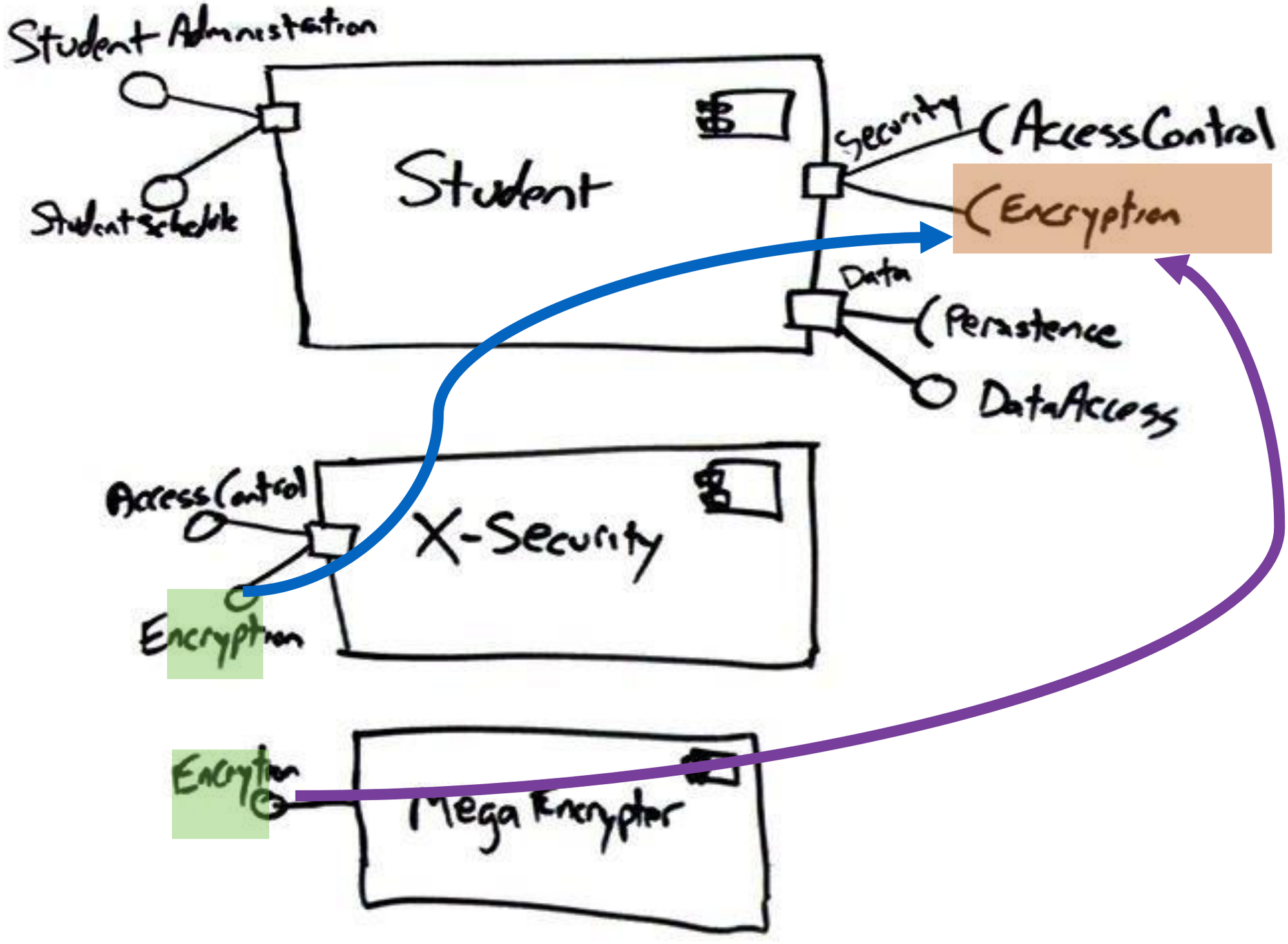
**3A**

| Initial | Stands for | Concept |
|---|---|---|
| S | SRP[4] | **Single responsibility principle**<br>a class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class) |
| O | OCP[5] | **Open/closed principle**<br>"software entities … should be open for extension, but closed for modification." |
| L | LSP[6] | **Liskov substitution principle**<br>"objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program." See also design by contract. |
| I | ISP[7] | **Interface segregation principle**<br>"many client-specific interfaces are better than one general-purpose interface."[8] |
| D | DIP[9] | **Dependency inversion principle**<br>one should "depend upon abstractions, [not] concretions."[8] |

**4A**

https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)

# Binding Components




www.agilemodeling.com/artifacts/componentDiagram.htm

# Example: Annotation-based injection in Java

```
class Student implements
    StudentAdministration, StudentSchedule {

  @Inject
  private Encryption e;

}
```

**Provided Interface**

**Required Interface**

# All-in-One (in Spring)

**Component**

**Provided Interfaces**

```
@Service
public class CartHandler implements CartModifier, CartProcessor {

    private static final Logger LOG = LoggerFactory.getLogger(CartHandler.class);

    private final Payment payment;

    private final CustomerFinder customerFinder;

    @Autowired
    public CartHandler(Payment payment, CustomerFinder customerFinder) {
        this.payment = payment;
        this.customerFinder = customerFinder;
    }
```

**Required Interfaces**