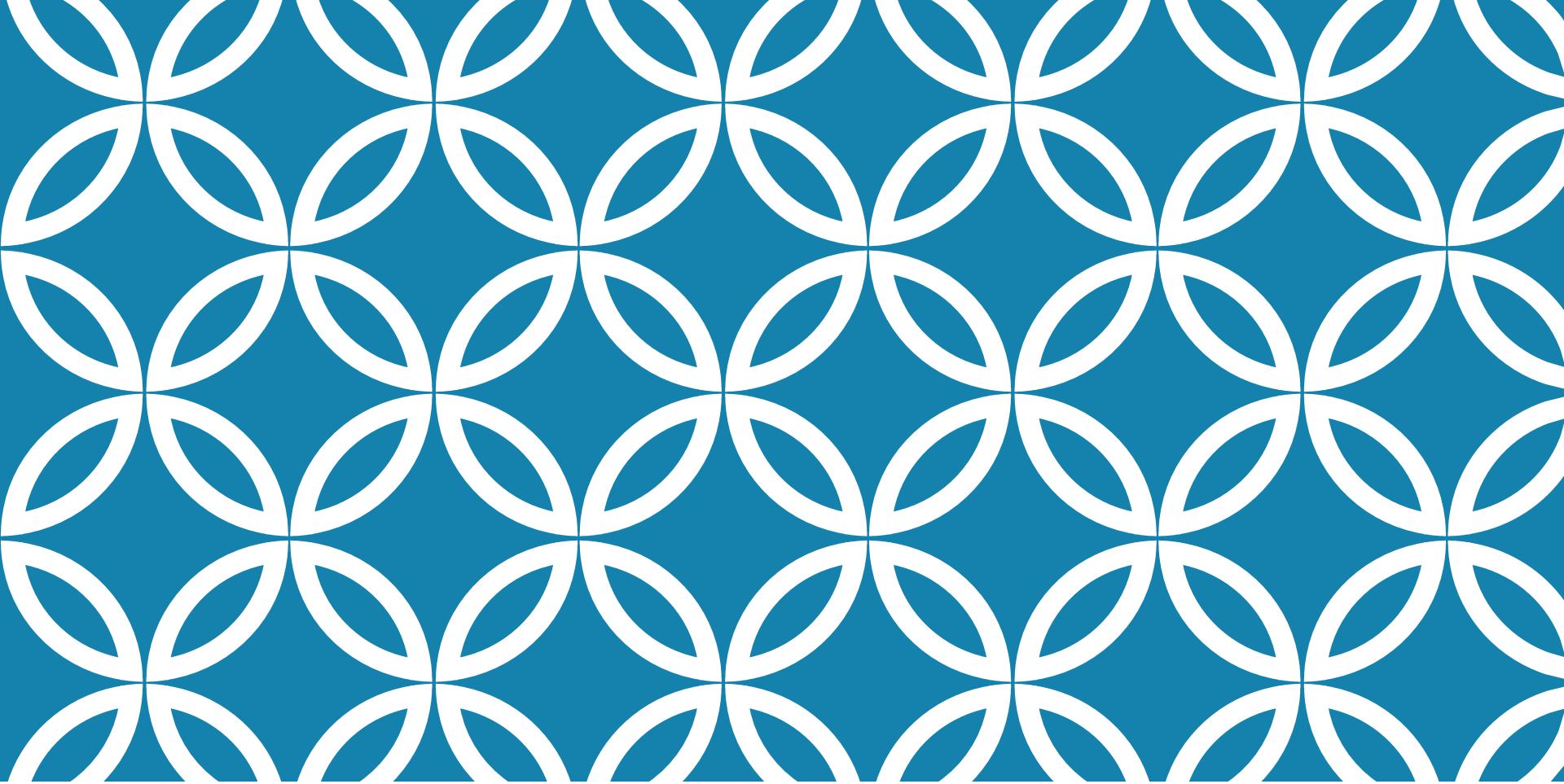


# BUSINESS LOGIC

Lecture 4

# CONTENT

- Organizing the Business Logic
  - Domain driven design
  - Service driven design

# REFERENCES


- Martin Fowler et. al, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2003 [Fowler]
- **Abel Avram, Floyd Marinescu, *Domain Driven Design Quickly*, InfoQ Enterprise Software Development, 2006 [DDDQ]**
- **Vaughn Vernon, *Domain Driven Design Distilled*, Addison Wesley, 2016. [DDDD]**
- **Eric Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software***
- Microsoft Application Architecture Guide, 2009 [MAAG]
- Armando Fox, David Patterson, and Koushik Sen, SaaS Course Stanford, Spring 2012 [Fox]
- Jacques Roy, SOA and Web Services, IBM
- Mark Bailey, Principles of Service Oriented Architecture, 2008
- Erl, Thomas. Service-Oriented Architecture: Analysis and Design for Services and Microservices. Pearson Education. 2016
- Erl, Thomas. SOA Design Patterns, Prentice Hall, 2009.

http://soapatterns.org

# PATTERNS FOR ENTERPRISE APPLICATIONS [FOWLER]

Enterprise Applications

- Persistent data
- Volume of data
- Concurrent access
- Complex user interface
- Integration with other applications
  - Conceptual dissonance
- Business logic

# ENTERPRISE APPLICATIONS

Example: B2C online retailer

- High volume of users: scalability

Example: processing of leasing agreements

- Complicated business logic
- Rich-client interface
- Complicated transaction behavior

Example: expense tracking for small company

# PRINCIPAL LAYERS

See pattern Layers in [POSA]

Here: applied to enterprise applications

**Presentation logic**

- Interaction with user
- Command-line or rich client or Web interface

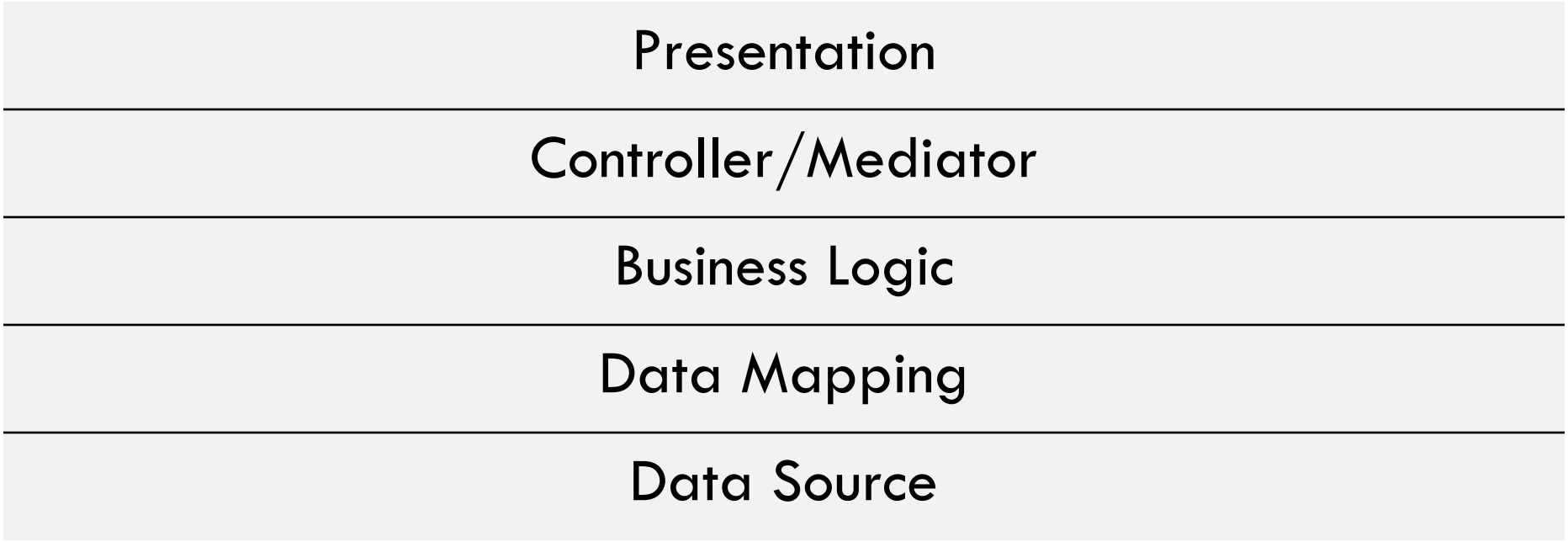
**Business logic**

- Validation of input and calculation of results

**Data access logic**

- Communication with databases and other applications

# MORE DETAILED LAYERING



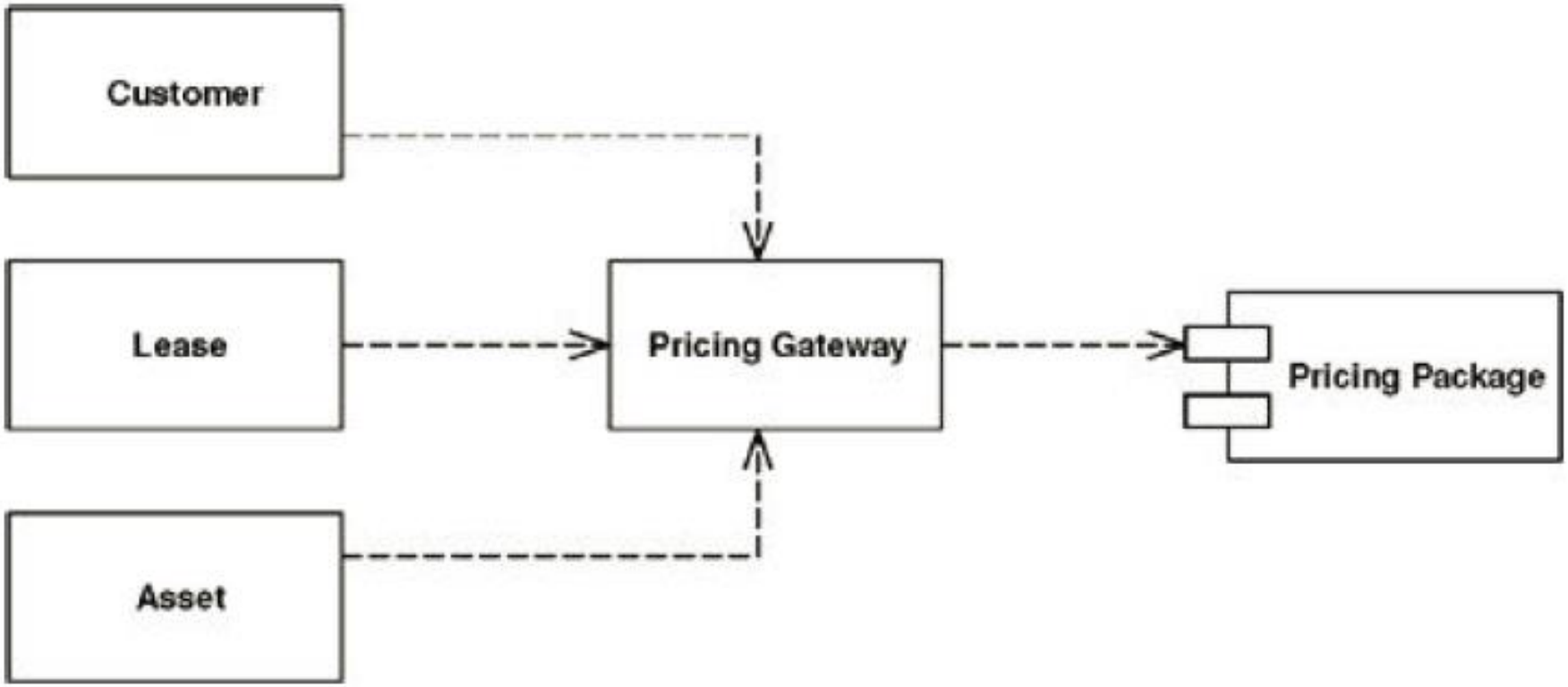
# BUT FIRST…

Some **basic** patterns

- Gateway
- Record Set

# GATEWAY

*An object that encapsulates access to an external system or resource*



# GATEWAY – HOW IT WORKS

- External resources - each with its own API
- Wraps all the API specific code into a Gateway whose interface is defined by the client
- Should be simple and minimal. Additional complex logic should be placed in the client
- Can be generated.

- Examples: Gateways to access Databases (DAOs)

# GATEWAY - BENEFITS

- Easier handling of awkward API's
- Easier to swap out one kind of resource for another
- Easier to test by giving you a clear point to deploy *Service Stubs (A stand-in implementation of an external service)*

# GATEWAY - EXAMPLE

Build a gateway to an interface that just sends a message using the message service

```
int send(String messageType, Object[] args);
```

Confirmation message

```
messageType = 'CONFIRM';
args[0] = id;
args[1] = amount;
args[2] = symbol;
```

# BETTER...

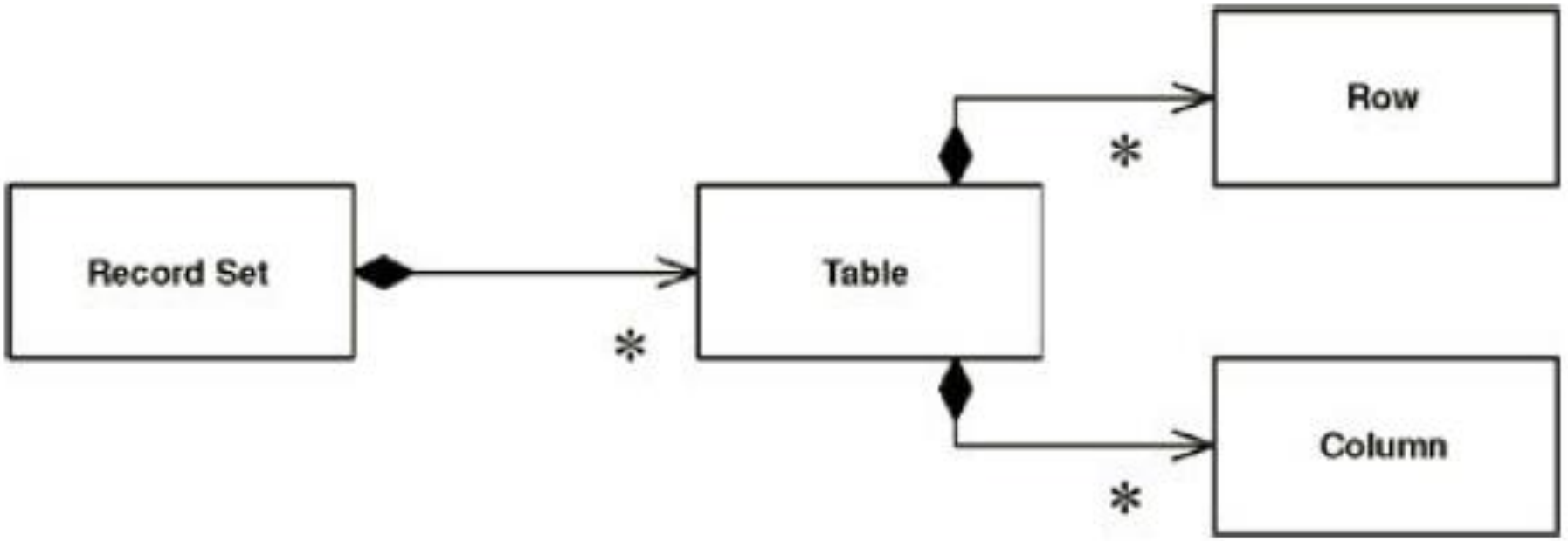
```
public void sendConfirmation(String orderID, int amount, String symbol);


class Order...

   public void confirm() {
      if (isValid()) Environment.getMessageGateway().sendConfirmation(id, amount,
symbol);
   }

 class MessageGateway...

    protected static final String CONFIRM = "CNFRM";
    private MessageSender sender;
    public void sendConfirmation(String orderID, int amount, String symbol) {
       Object[] args = new Object[]{orderID, new Integer(amount), symbol};
       send(CONFIRM, args);
    }
    private void send(String msg, Object[] args) {
       int returnCode = doSend(msg, args);
       if (returnCode == MessageSender.NULL_PARAMETER)
```

# GATEWAY VS. FAÇADE VS. ADAPTER

- The *façade* is usually done by the **writer of the service** for general use, while a *Gateway* is written by the **client** for their particular use.
- A *façade* always implies a **different interface** to what it's covering, while a *Gateway* may copy the wrapped interface entirely, being used for substitution or testing purposes
- *Adapter* alters an implementation's interface to match **another interface** which you need to work with.
- With *Gateway* there usually **isn't an existing interface**, although you might use an adapter to map an implementation to an existing *Gateway* interface. In this case the adapter is part of the implementation of the *Gateway*.

# RECORD SET

*An in-memory representation of tabular data*



# RECORD SET – HOW IT WORKS

- Provides an in memory structure that looks exactly like the result of a SQL query, but can be generated and manipulated by other parts of the system.
- Usually provided by the framework/platform

Examples:

- ResultSet in Java
- Recordset in ADO.Net

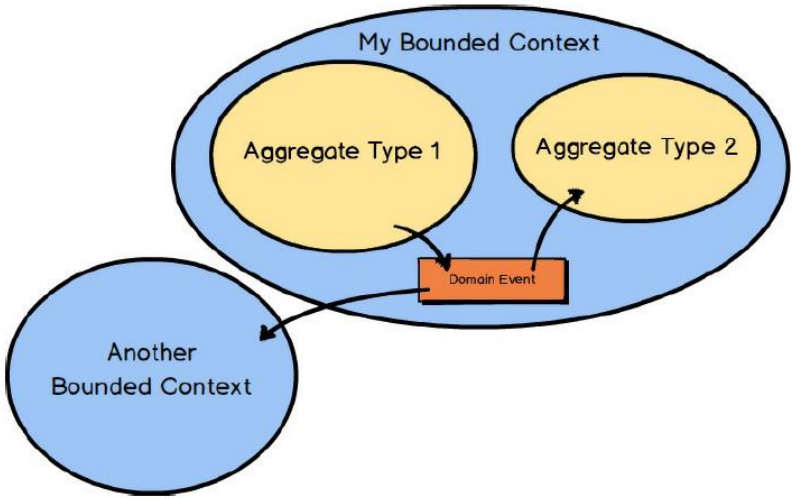
# BUSINESS LOGIC LAYER

" …It involves *calculations* based on inputs and stored data, *validation* of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch …" [Fowler]

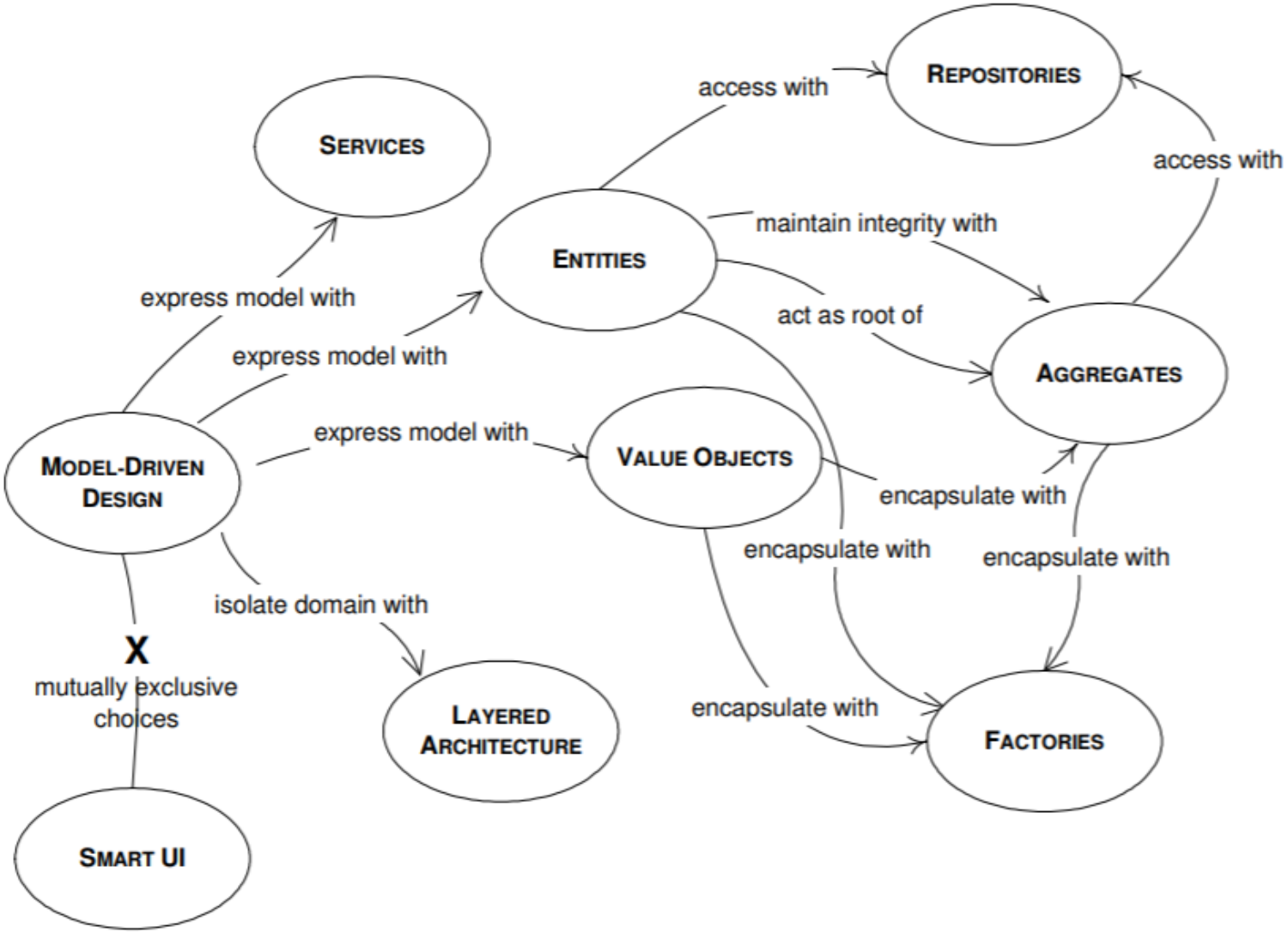
Also called Domain Layer

# DOMAIN DRIVEN DESIGN

- **Strategic** Design
  - Subdomains (**problem** perspective)
    - Core Domain
    - Support Domain
    - Generic Domain
  - Bounded Context (**solution** perspective)
  - Context Mapping
- **Tactical** Design
  - Aggregates
  - Domain Events



# MODEL DRIVEN DESIGN [DDDQ]

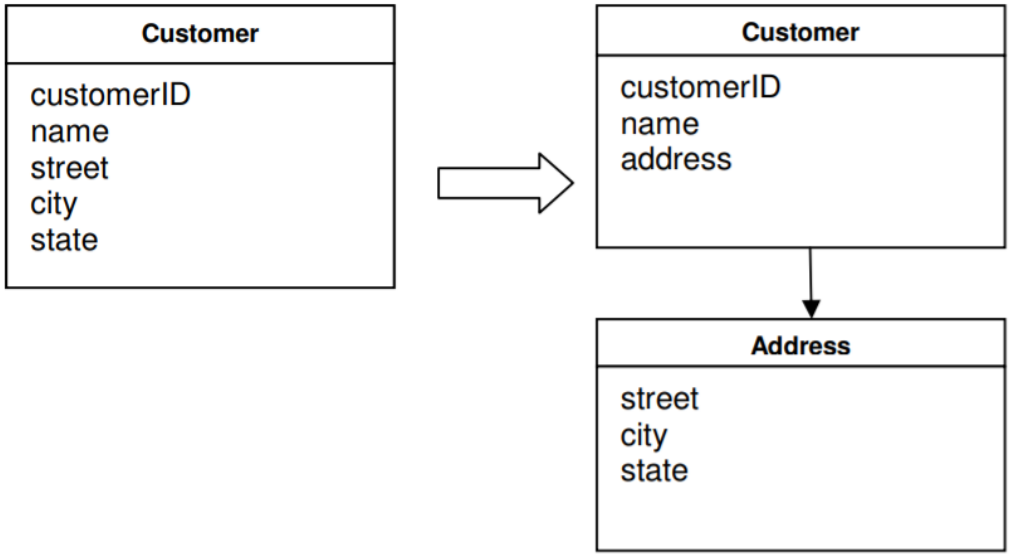


# ENTITIES

- Have an identity that remains unchanged through the states of the application
- Focus on identity and continuity, not on values of the attributes
- The identity can be
  - An attribute
  - A combination of attributes
  - An artificial attribute
- Examples:
  - Person
  - Account

# VALUE OBJECTS

- Do not have identities
- The values of the attributes are important
- Easily created and discarded (no tracking is needed)
- Highly recommended to make them immutable => can be shared!



# ENTITY OR VALUE OBJECT?

- **Entity**
  - Need to decide how to uniquely identify it
  - Need to track it
  - Need one instance for each object => affects performance

  Ex. Customer object

- **Value Object**
  - If the Value Object is shareable, it should be immutable
  - Can contain other Value Objects and even references to Entities

# SERVICES

- Contain operations that do not belong to any Entity/Value object
- Do not have internal states
- Operate on Entity/Value objects becoming a point of connection => loose coupling between objects

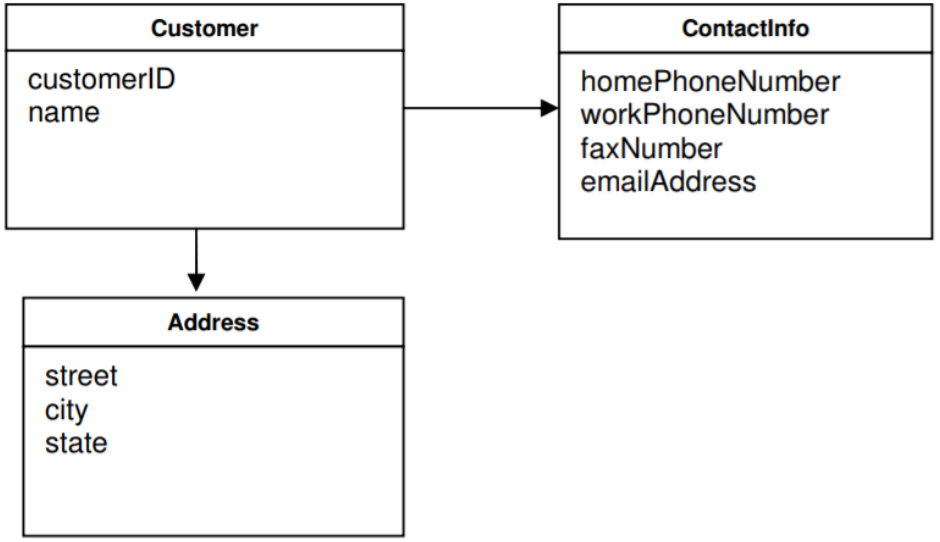
# SERVICES

Characteristics:

- The operation performed by the Service refers to a domain concept which does not naturally belong to an Entity or Value Object.
- The operation performed refers to other objects in the domain.
- The operation is stateless.

Ex. transferring money from one account to another.

# AGGREGATES

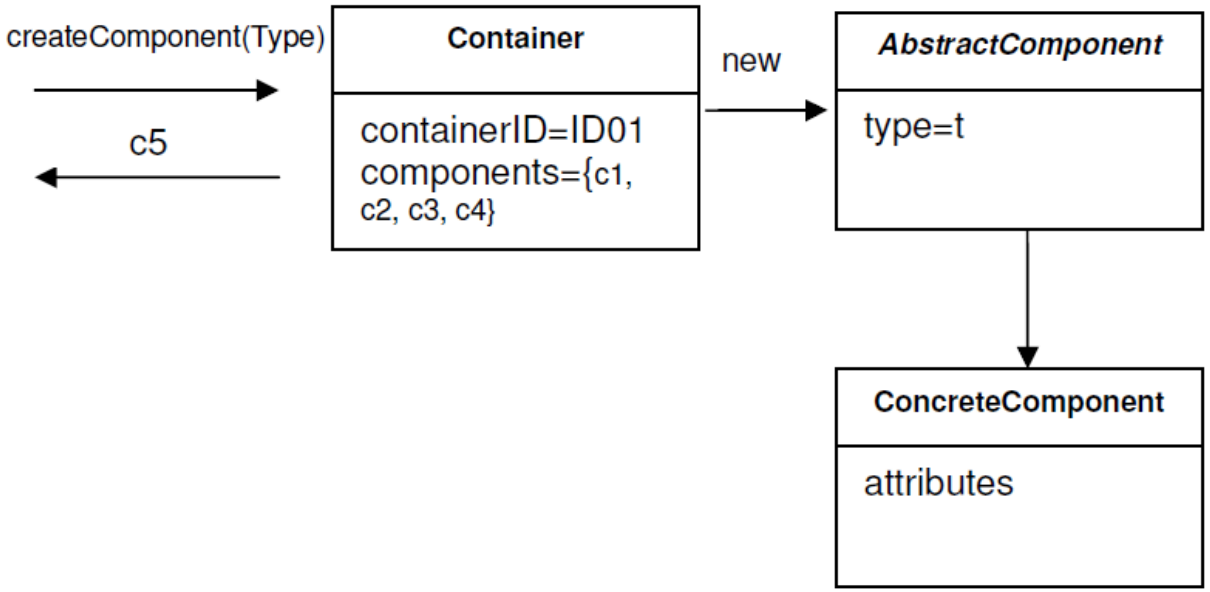



- **A domain pattern used to define object ownership and boundaries**
- Associations
  - One-to-one
  - One-to-many
  - Many-to-many
- Groups associated objects that represent one unit with regard to data change => defines a **transactional consistency boundary**
- Each aggregate has one root (an Entity)
- Only the root is accessible from outside

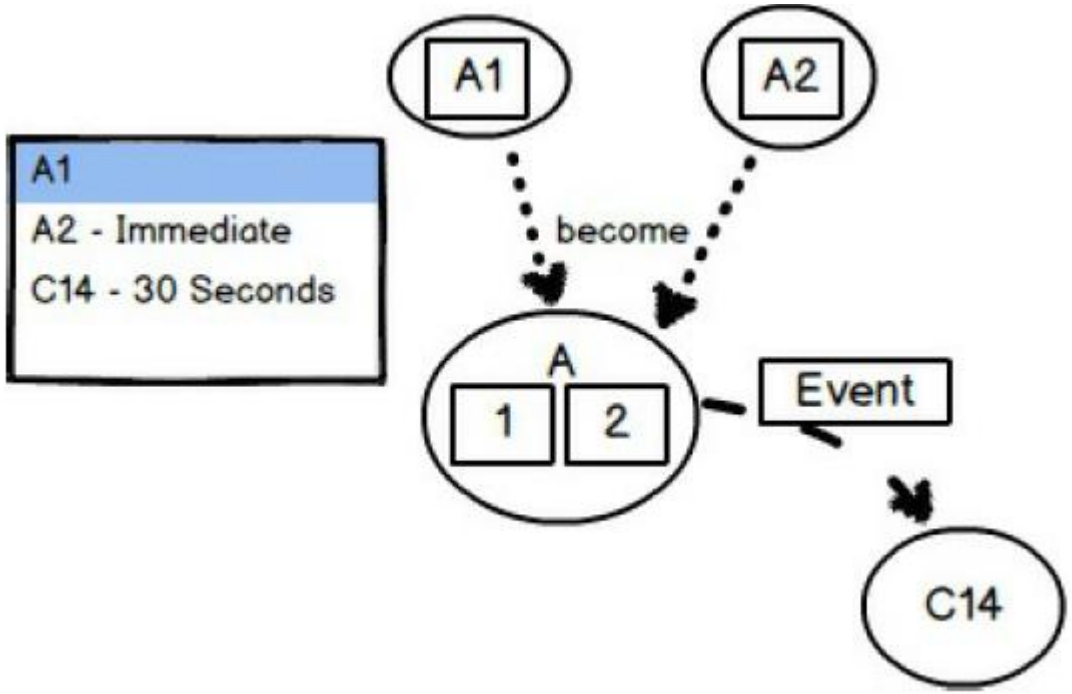
# DISCUSSION

- Root Entity has global identity, inner entities have local identity
- Root Entity enforces invariants
- If the root is deleted => all aggregated objects are deleted, too
- Creating aggregates => atomic process

⇒Use Factories



# AGGREGATES




**Rules of thumb**

- Protect business invariants inside *Aggregate* boundaries.

(transactional consistency within Aggregate)

- Design small *Aggregates*.
- Reference other *Aggregates* by identity only.
- Update other *Aggregates* using eventual consistency.

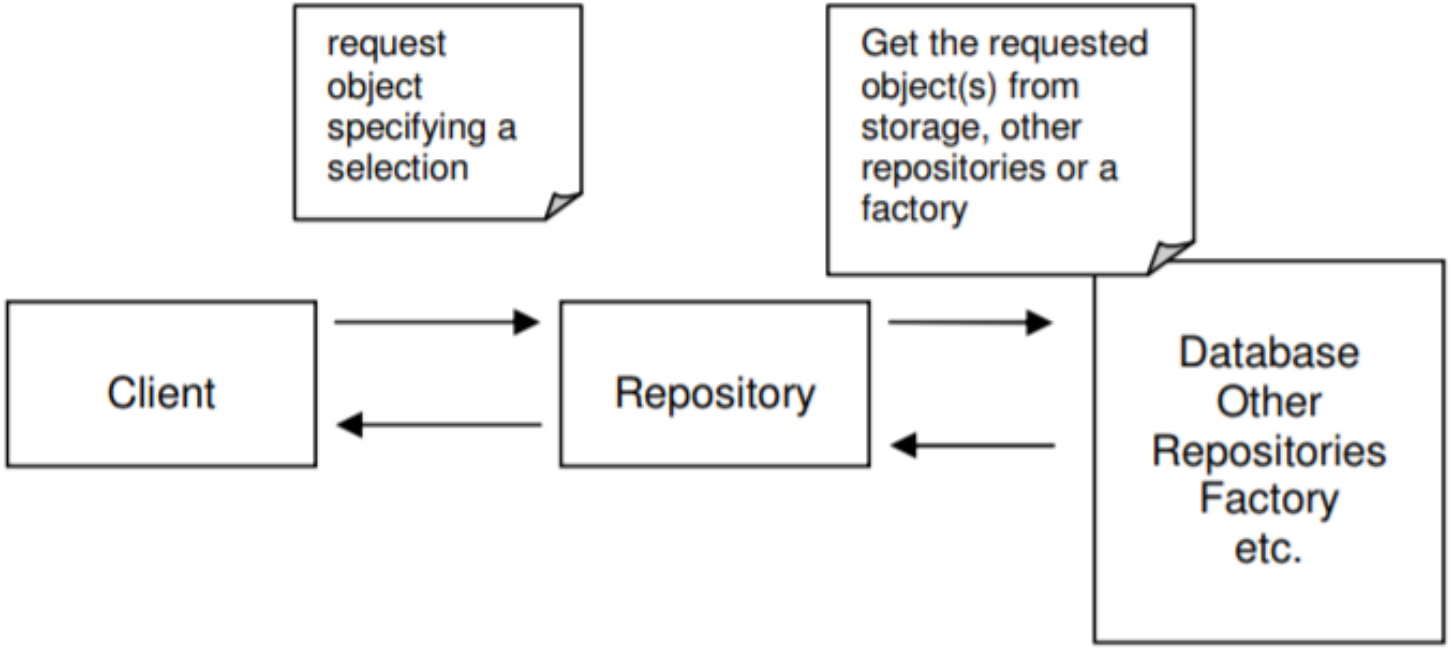
(transactional consistency across Aggregates)

# REPOSITORY

- How do we get the object (i.e. Entity, Value)?
  - Create it (Factories)
  - Obtain it
    - If it is a Value Object => need the root of the Aggregate
    - If it is an Entity => can be obtained directly from the database.
  - Problems:
    - Dependency on the database structure
    - Mixing database access into the domain logic

# REPOSITORY

Repository encapsulates all the logic needed to obtain object references (either by already storing them, or by getting them from the storage).

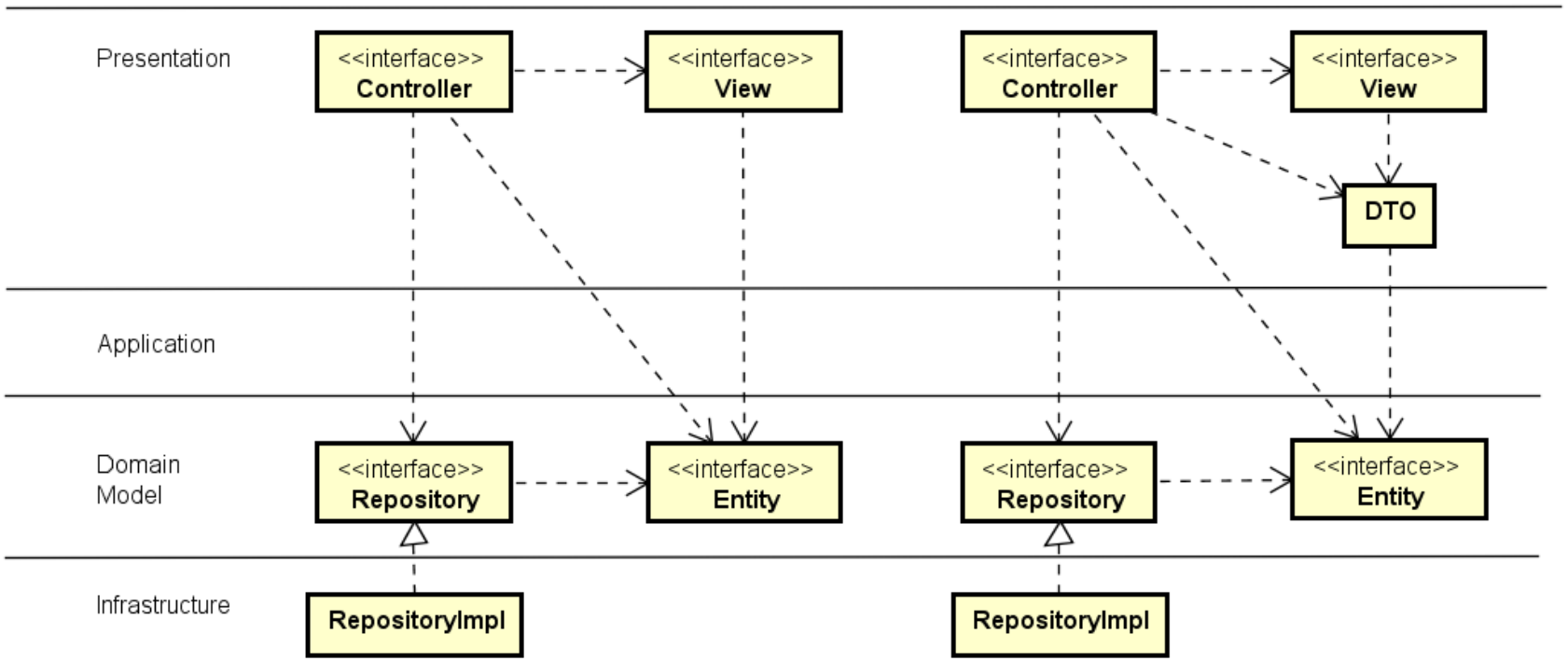


# DISCUSSION

How complex should the Domain Model be?

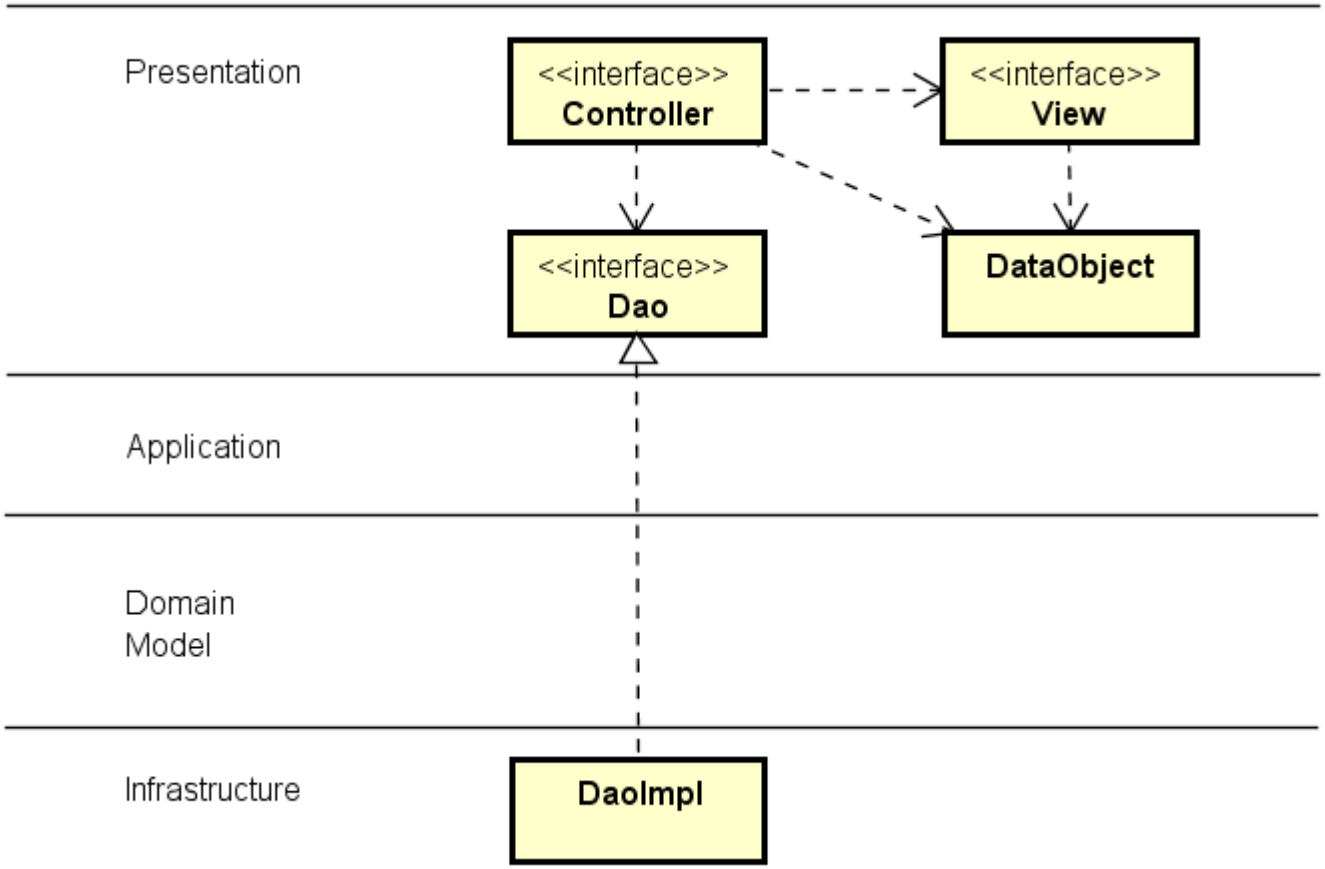
Anemic Domain Model (**Anti-pattern**)

- Just forms over data (CRUD operation + some validation



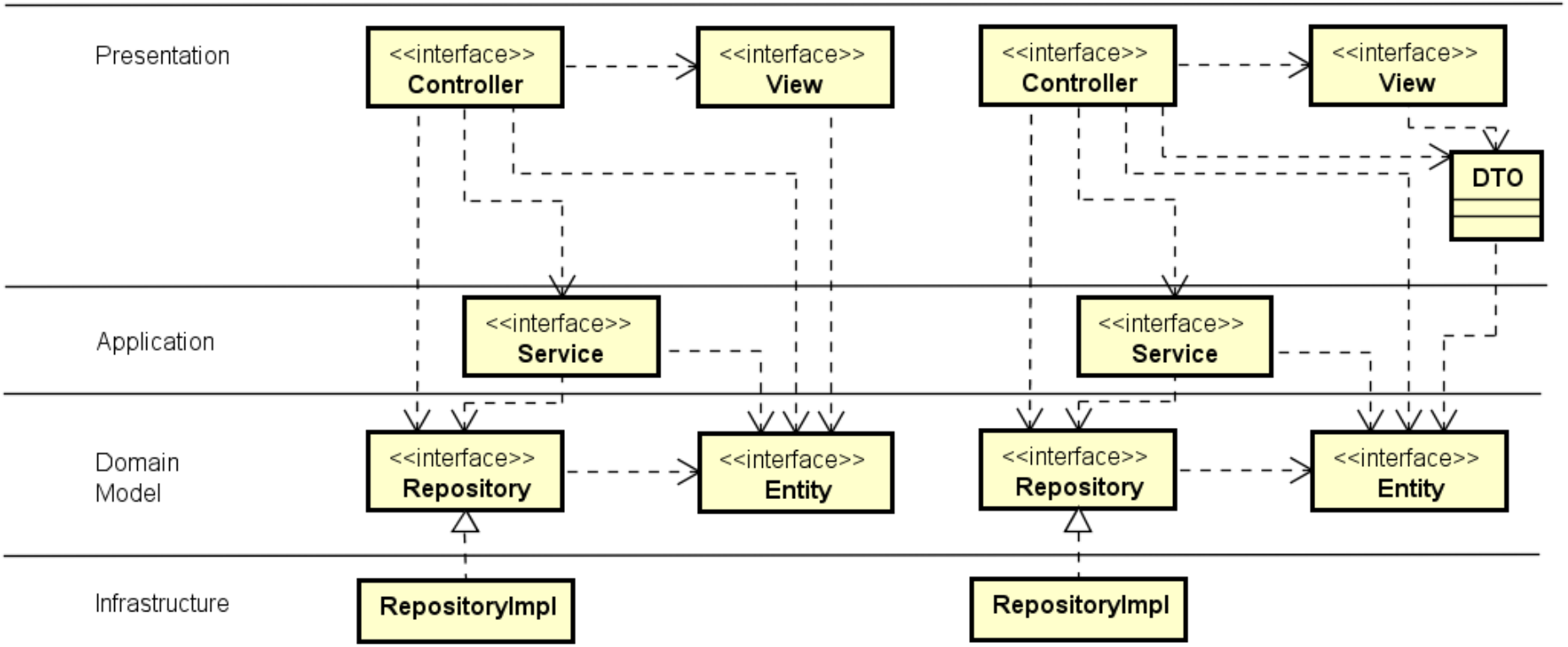
# ALTERNATIVES

Even less Domain Model



# WHEN TO USE SERVICES?

- Need for driving the workflow, coordination transaction management
- E.g. The controller needs to update more than one entity



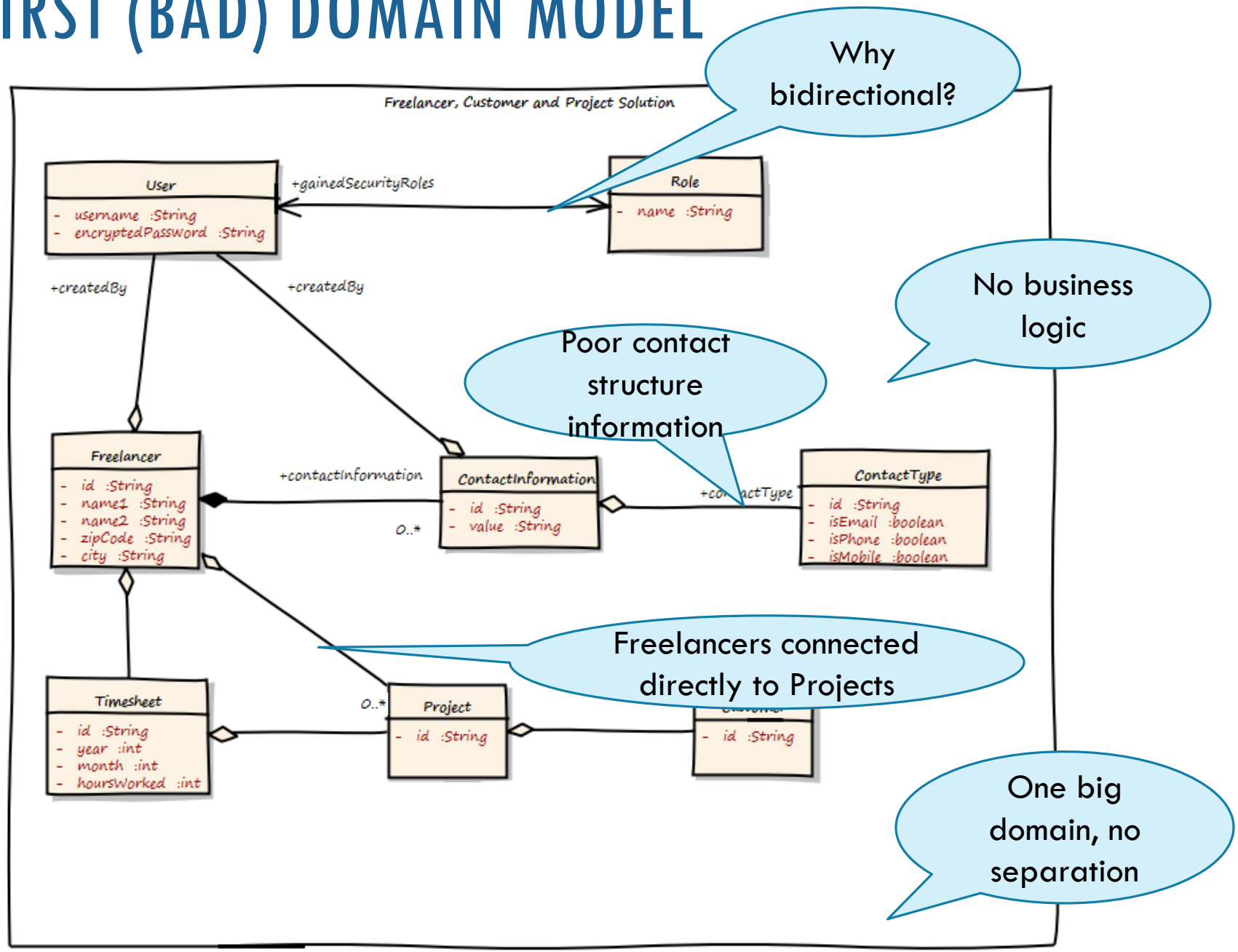
# THE IT BODY LEASING EXAMPLE

A company provides IT Body Leasing. They have some Employees, and also a lot of Freelancers as Subcontractors. Requirements:

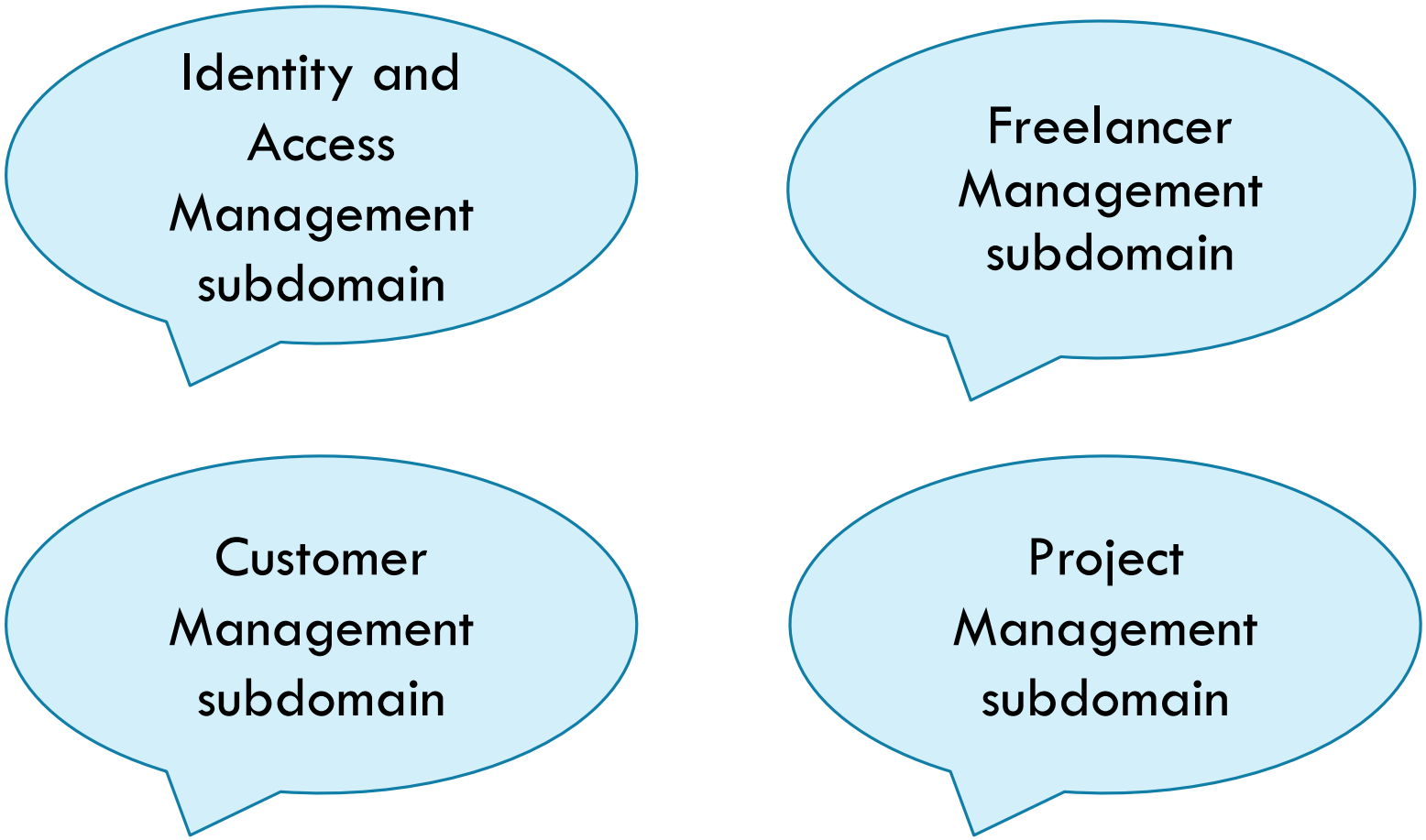
- A searchable catalog of Freelancers must be provided
- Allows to store the different Communication Channels available to contact a Freelancer
- A searchable catalog of Projects must be provided
- A searchable catalog of Customers must be provided
- The Timesheets for the Freelancers under contract must be maintained

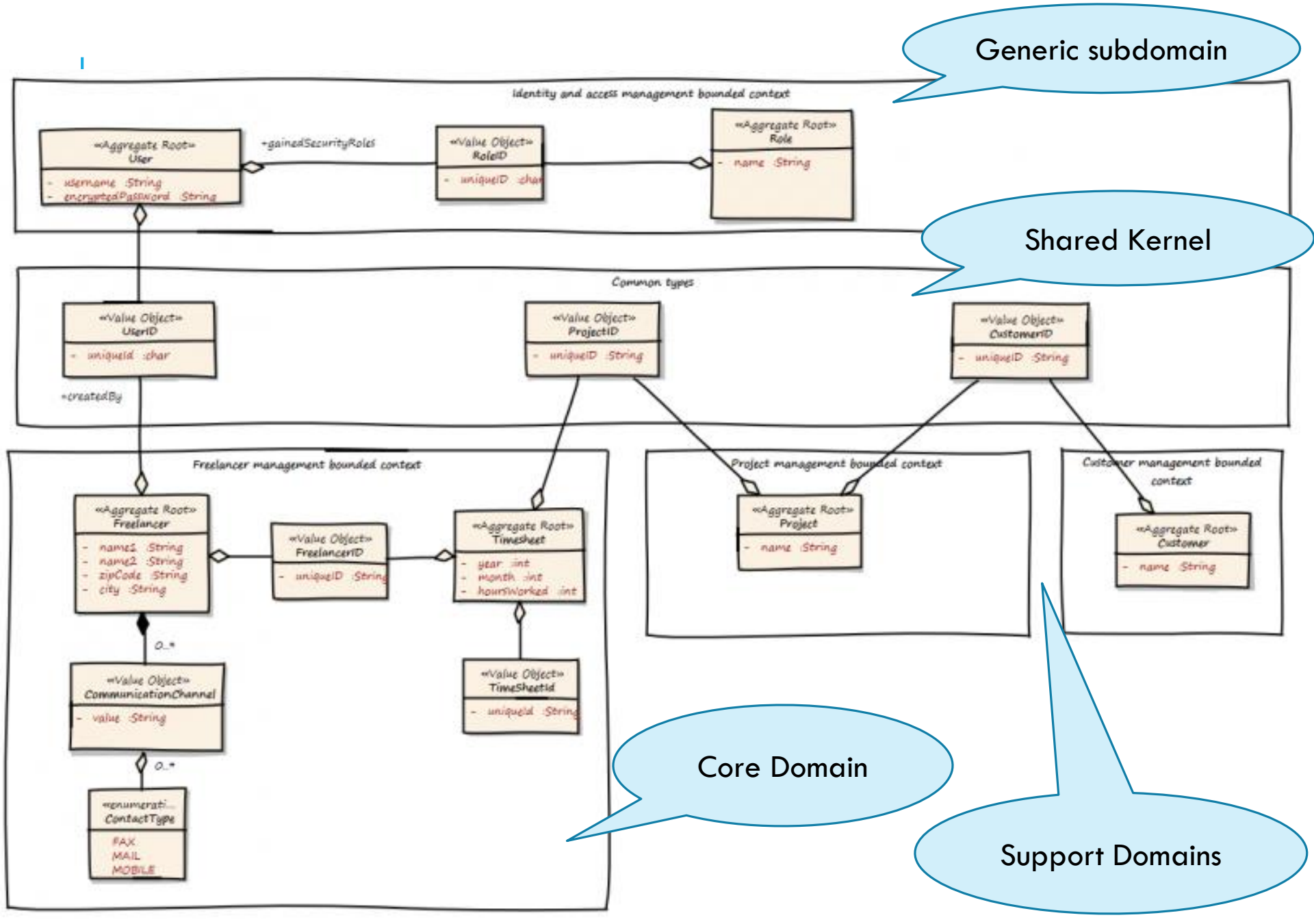
https://www.mirkosertic.de/blog/2013/04/domain-driven-design-example/

# FIRST (BAD) DOMAIN MODEL

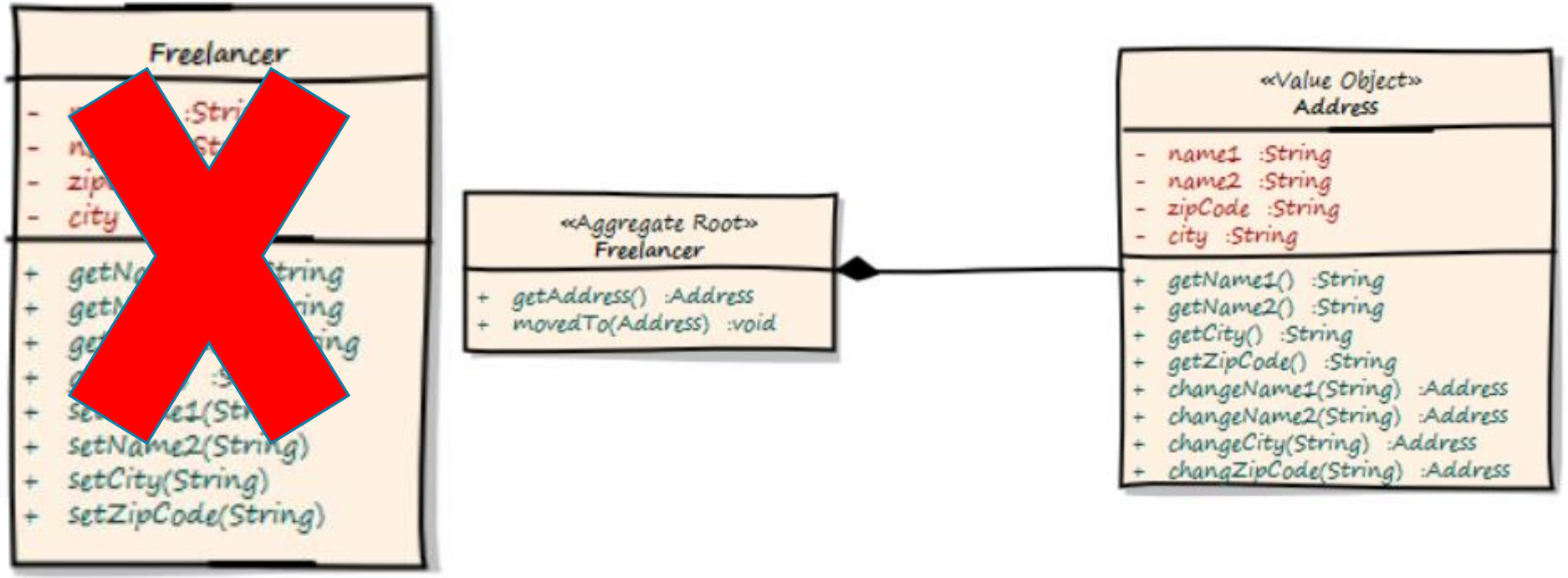


# SPLITTING THE PROBLEM INTO SUBDOMAINS



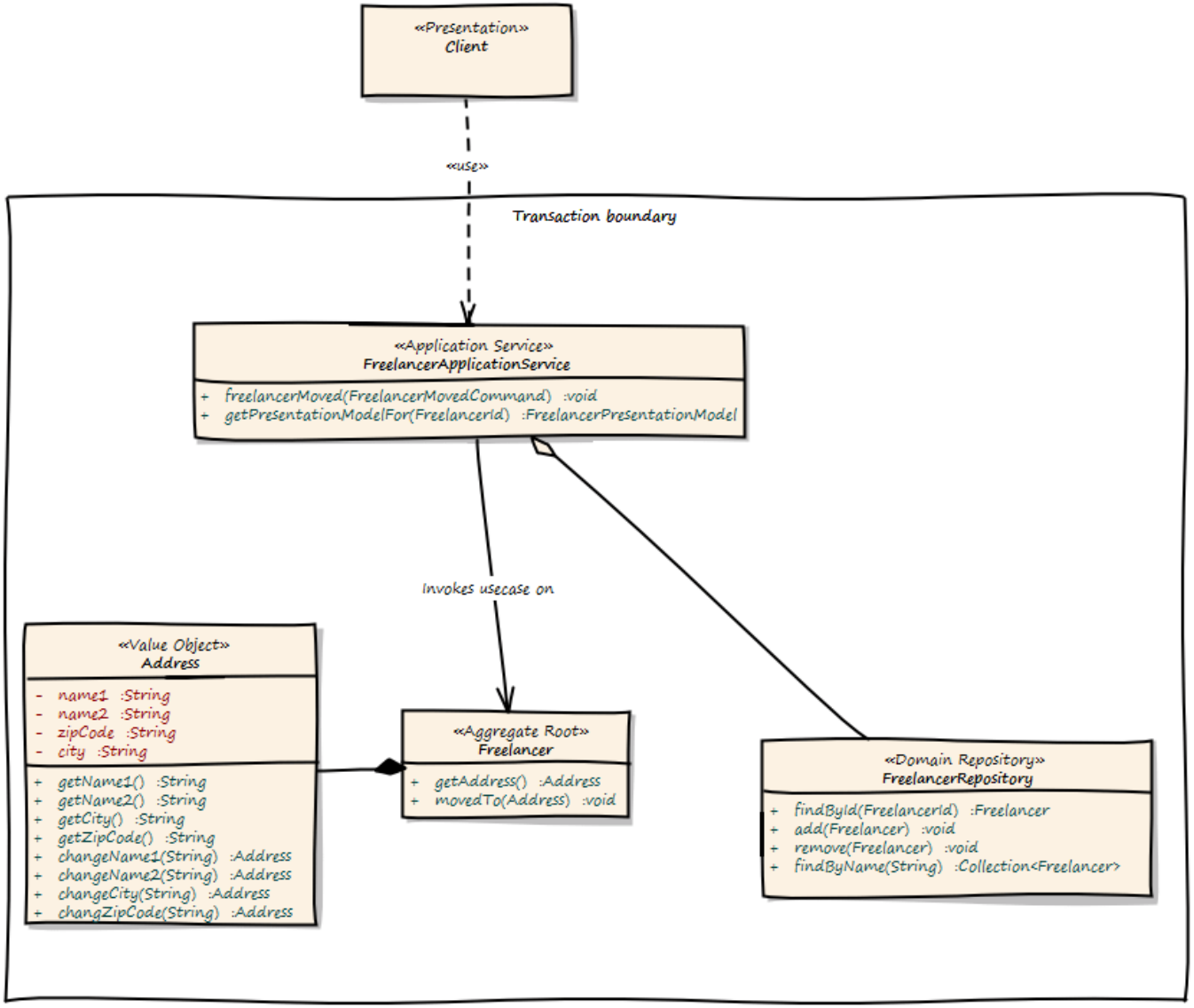

Generic subdomain
Identity and access management bounded context
«Aggregate Root»
User
- username :String
- encryptedPassword :String
+gainedSecurityRoles
«Value Object»
RoleID
- uniqueID :char
«Aggregate Root»
Role
- name :String
Shared Kernel
Common types
«Value Object»
UserID
- uniqueId :char
+createdBy
«Value Object»
ProjectID
- uniqueID :String
«Value Object»
CustomerID
- uniqueID :String
Freelancer management bounded context
«Aggregate Root»
Freelancer
- name1 :String
- name2 :String
- zipCode :String
- city :String
«Value Object»
FreelancerID
- uniqueID :String
«Aggregate Root»
Timesheet
- year :int
- month :int
- hoursWorked :int
0..*
«Value Object»
CommunicationChannel
- value :String
«Value Object»
TimeSheetId
- uniqueId :String
0..*
«enumerati...
ContactType
FAX
MAIL
MOBILE
Project management bounded context
«Aggregate Root»
Project
- name :String
Customer management bounded context
«Aggregate Root»
Customer
- name :String
Core Domain
Support Domains

# APPLYING DOMAIN DRIVEN DESIGN

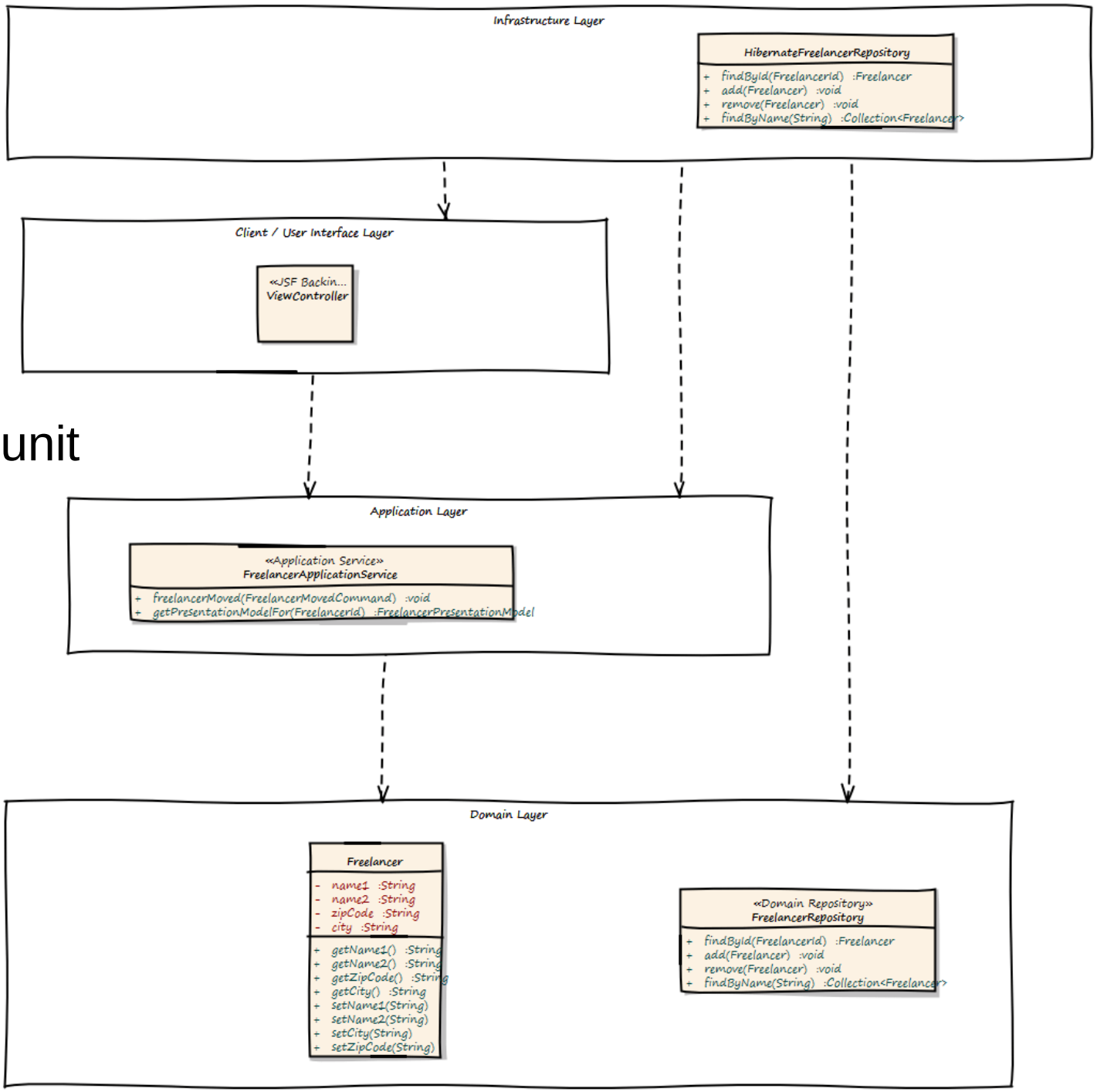


# ADDING BEHAVIOR

Freelancer moved to new location



# The Deployment unit

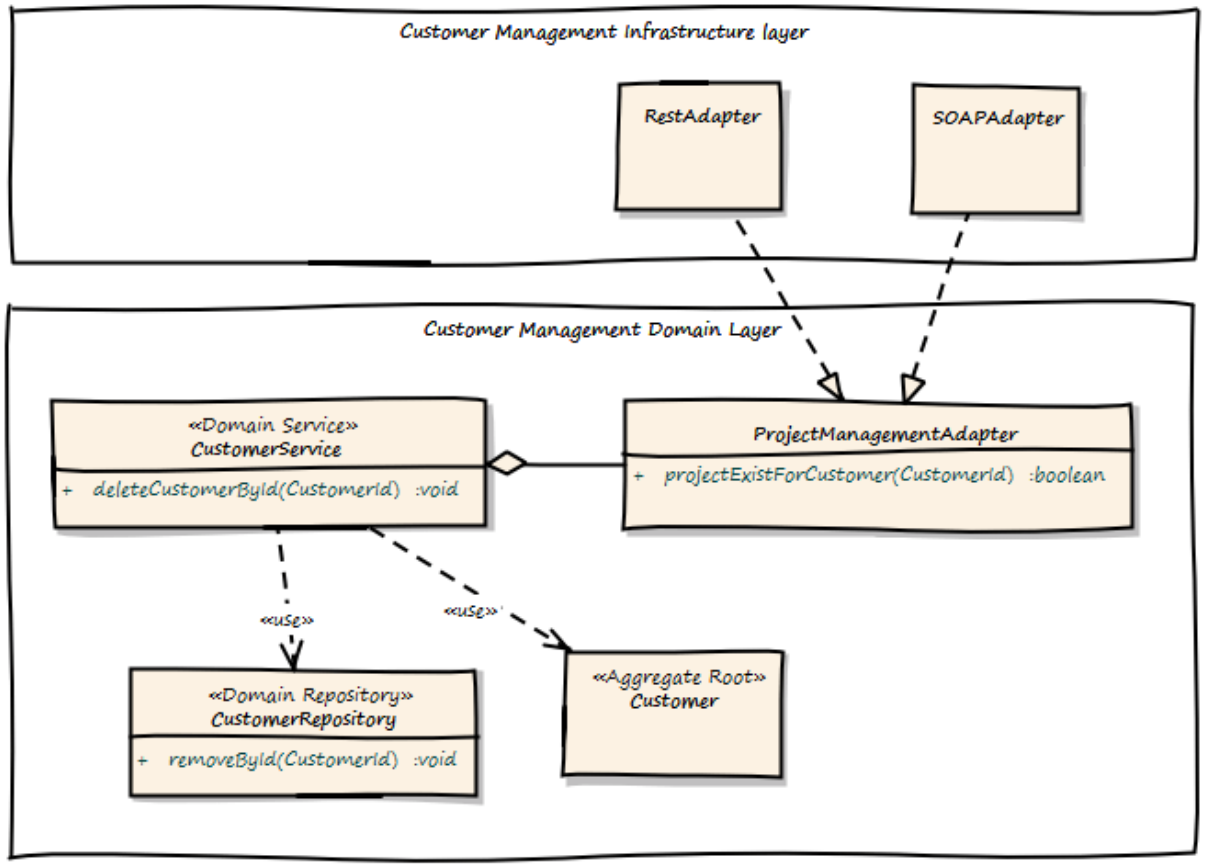


# CONTEXT INTEGRATION

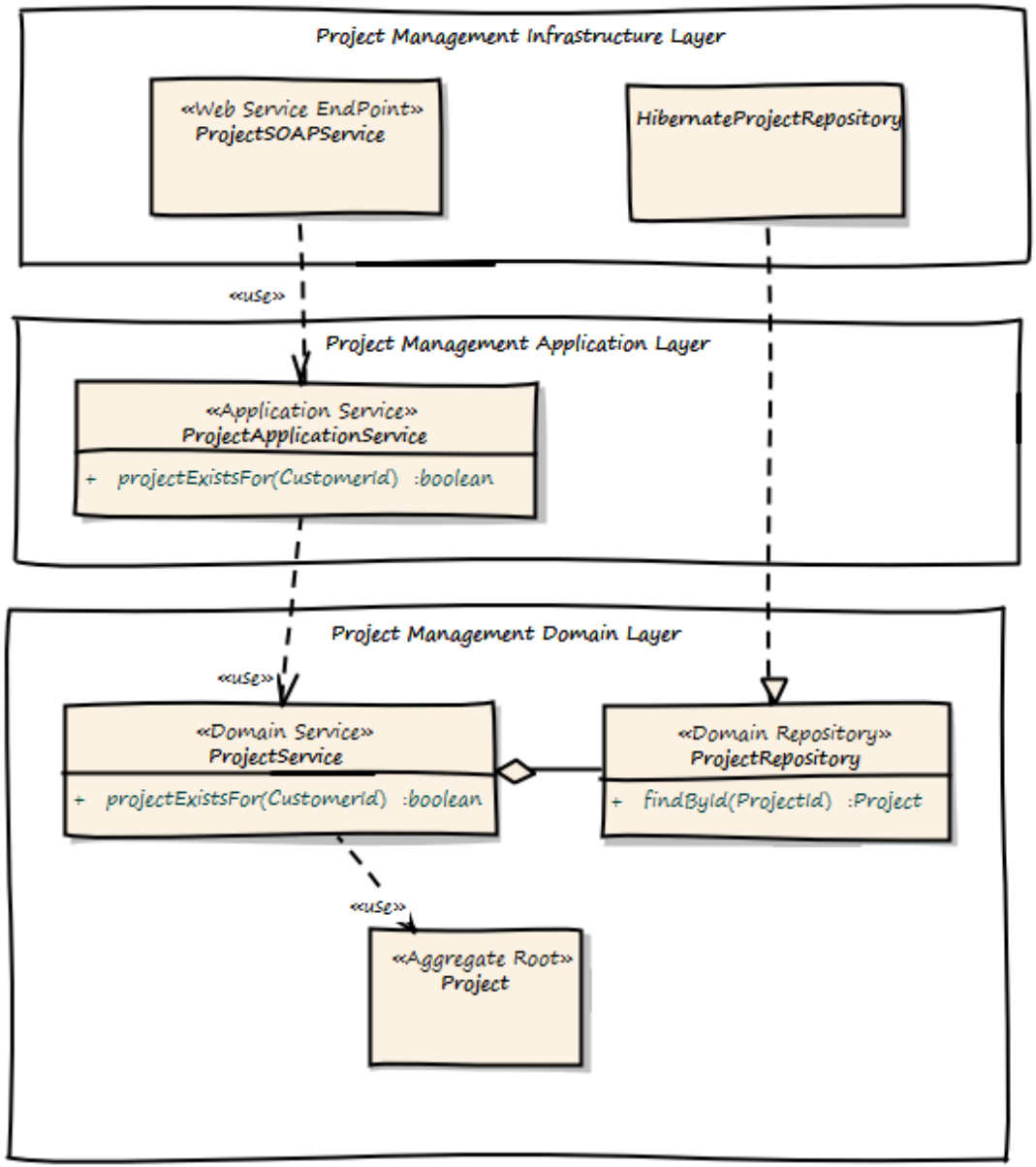
- A Customer can only be deleted if there is no Project assigned

⇒Domain Service

⇒Synchronous Integration



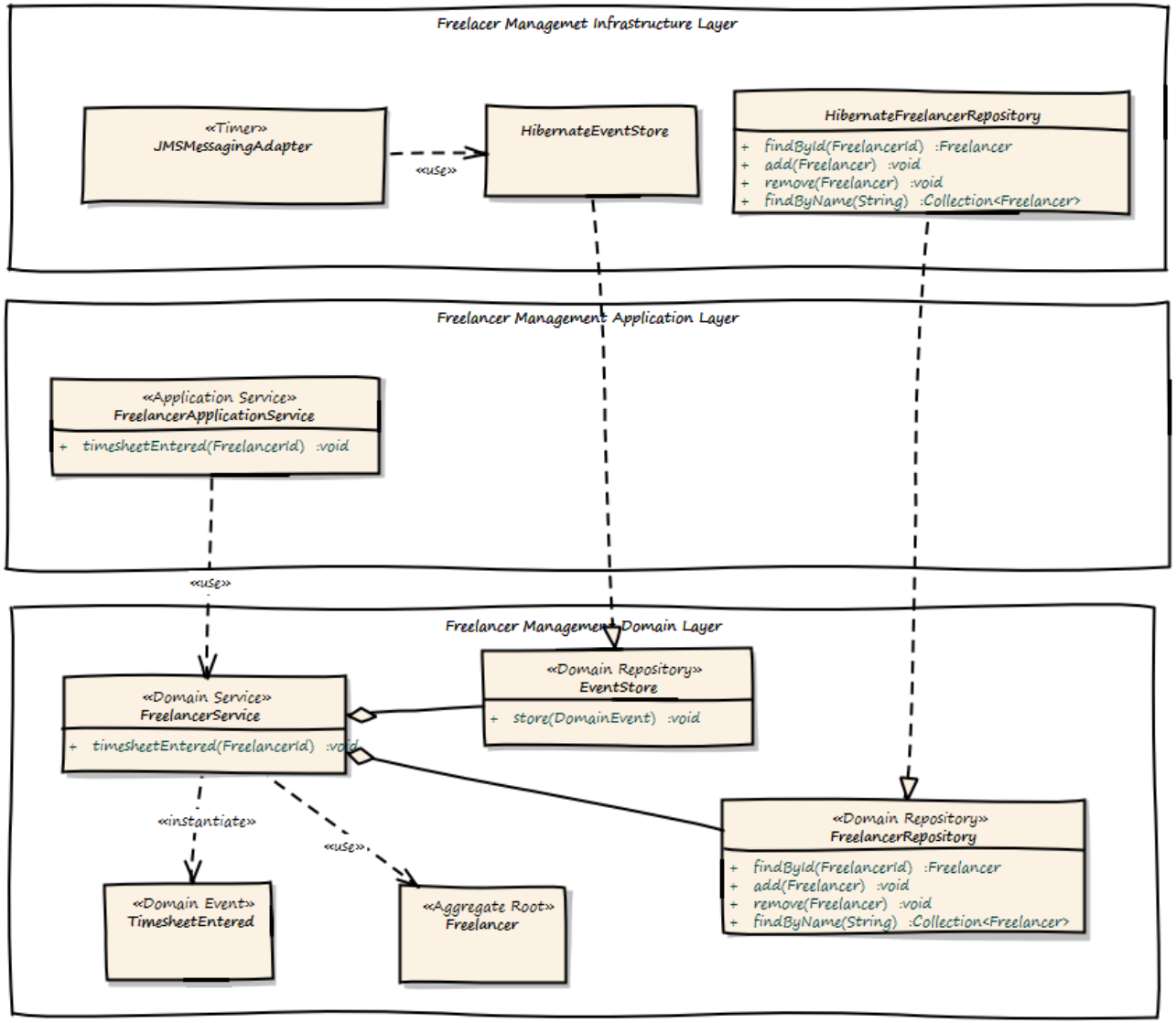
# ON THE PROJECT MANAGEMENT SIDE



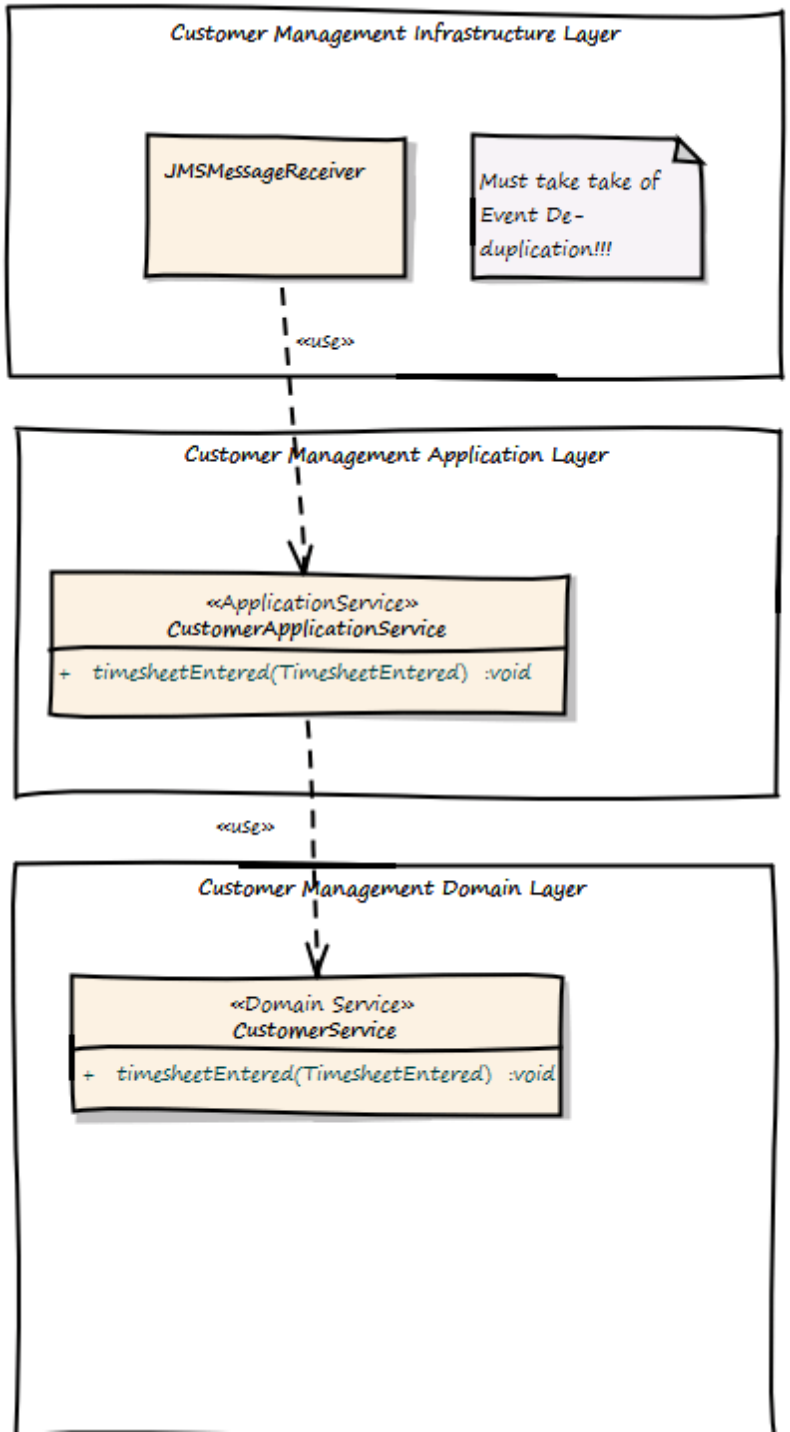
# ASYNCHRONOUS EXAMPLE

Once a Timesheet is entered, the Customer needs to be billed

Use Domain Events



# ON THE CUSTOMER SIDE



# SERVICE BASED ARCHITECTURES

"1.All teams will henceforth expose their data and functionality through **service interfaces**

2.Teams must **communicate** with each other **through** these **interfaces**

3.There will be **no other form of interprocess communication** allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever. The only communication allowed is via service interface calls over the network.

4. It **doesn't matter** what [API protocol] **technology** you use.

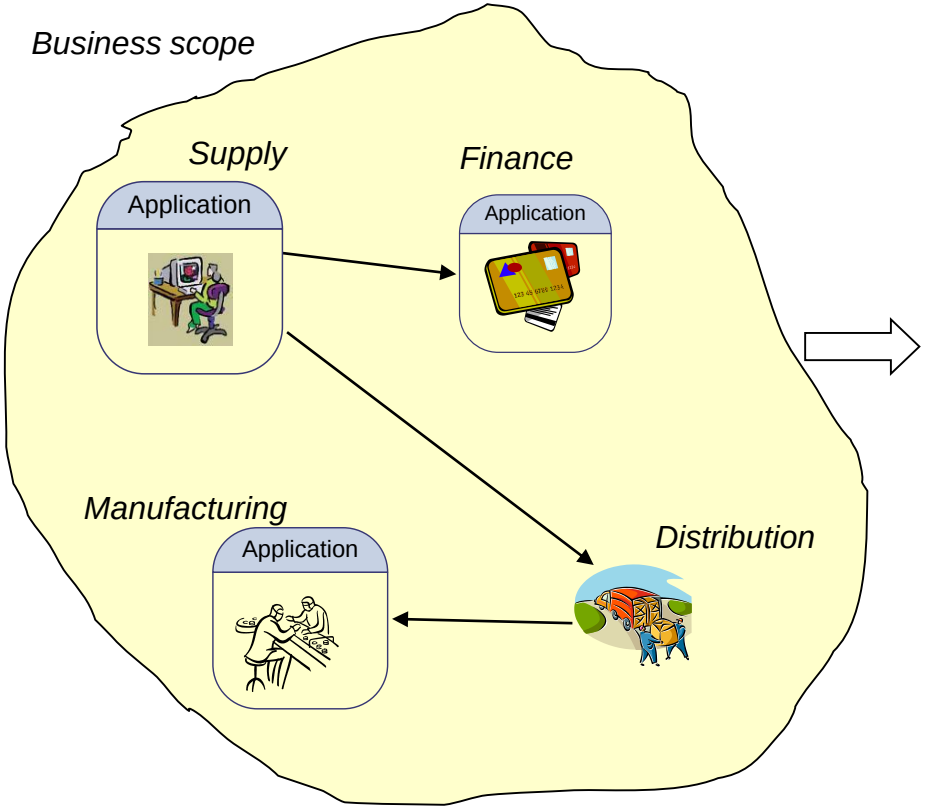
5. **Service interfaces**, without exception, must be designed from the ground up to be **externalizable**. That is to say, the team must plan and design to be able to expose the interface to developers in the outside world. No exceptions.

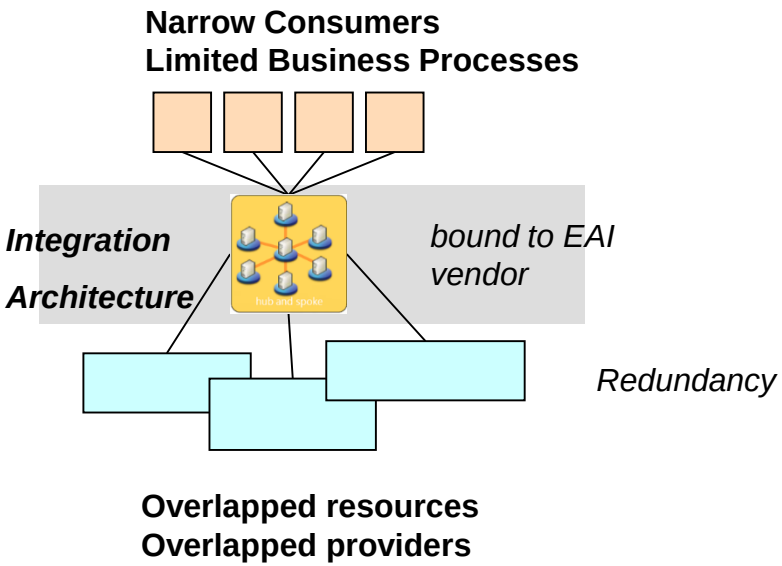
6. Anyone who doesn't do this **will be fired**.

7. Thank you; **have a nice day**!"

Amazon CEO (supposedly)

# APPLICATION CENTRIC

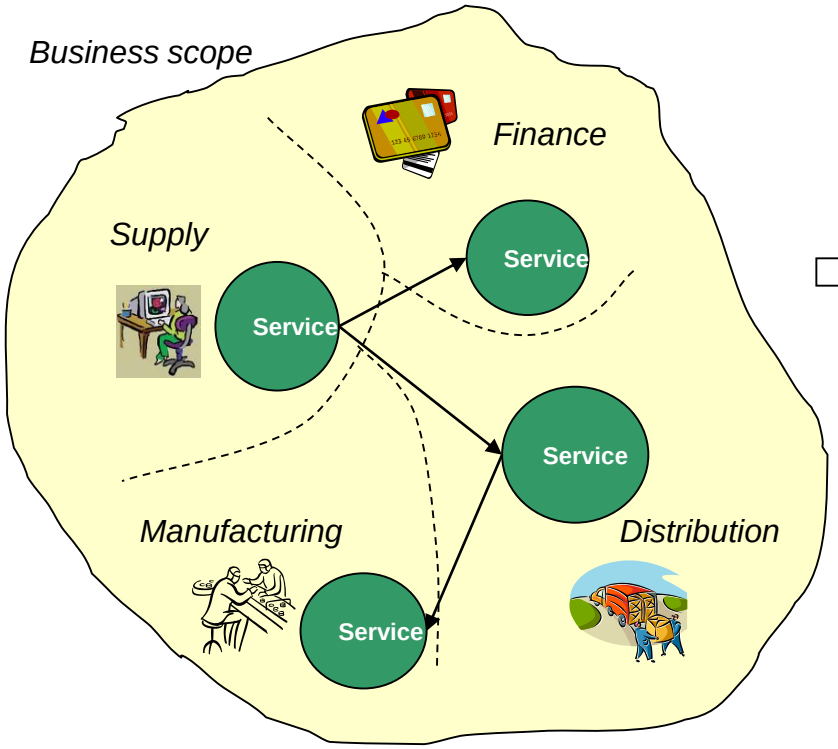


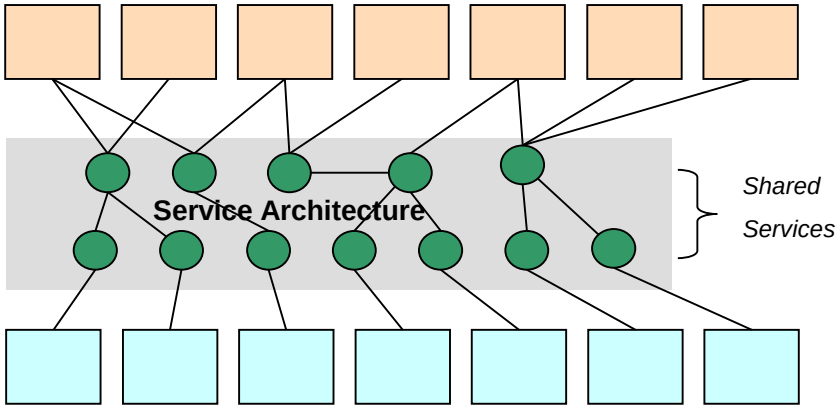

EAI ‘leverage’ application silos with the drawback of data and function redundancy.

**Business functionality is duplicated in each application that requires it.**

# SERVICE CENTRIC




**Multiple Service Consumers**
**Multiple Business Processes**



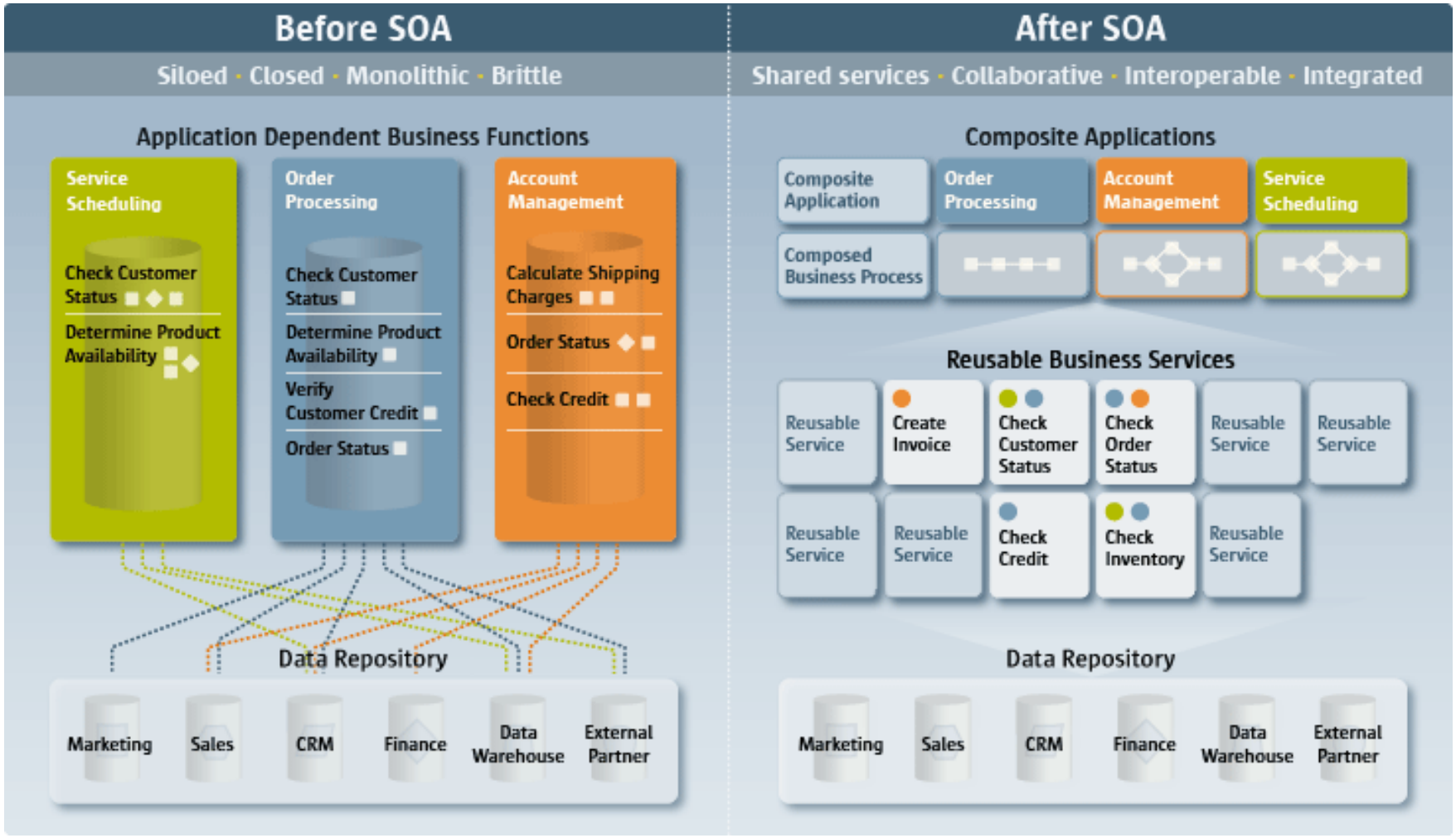

**Multiple Discrete Resources**
**Multiple Service Providers**

**SOA structures the business and its systems as a set of capabilities that are offered as Services, organized into a Service Architecture**

Service virtualizes how that capability is performed, and where and by whom the resources are provided, enabling multiple providers and consumers to participate together in shared business activities.

*source:TietoEnator AB, Kurts Bilder*

# BEFORE SOA – AFTER SOA




source:IBM

# DESIGN PRINCIPLES 1

**Services are reusable**

- Business functionalities exposed as services are designed with the intention of reuse whenever and where they are required

**Services share a formal contract**

- Services interact with each through a formal contract which is shared to exchange information and terms of usage

**Services are loosely coupled**

- Services are designed as loosely coupled entities able to interact while maintaining their state of loose coupling.

**Services abstract underlying logic**

- The business logic underpinning a service is kept hidden from the outside world. Only the service description and formal contract are visible for the potential consumers of a service

# DESIGN PRINCIPLES 2

**Services are composable**

- Services may composed of other services. Hence, a service's logic should be represented at different levels of granularity and promotes reusability and the creation of abstraction layers.

**Services are autonomous**

- A service should be independent of any other service

**Services are stateless**

- A service shouldn't required to maintain state information rather it should be designed to maximize statelessness

**Services are discoverable**

- A service should be discoverable through its description, which can be understood by humans and service users.

# DESIGN PRINCIPLES 3

**Services have a network-addressable interface**

- A service should be invoked from the same computer or remotely – through a local interface or Internet

**Services are location transparent**

- A service should be discoverable without the knowledge of its real location. A requestor can dynamically discover the location of a service looking up a registry.

The core principles are **autonomy, loose coupling, abstraction, formal contract**

# WHAT IS A WEB SERVICE? (W3C DEFINITION)

A Web service:

- is a software system designed to support **interoperable machine-to-machine interaction** over a network.
- has an **interface** described in a machine-processable format (specifically WSDL).

Other systems interact with the Web service in a manner prescribed by its description using SOAP-messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

# WHAT IS A WEB SERVICE: A SIMPLER DEFINITION

A Web Service is a standards-based way for an application to call a function over a network and to do it without having to know:

- the **location** where the function will be executed,
- the **platform** where the function will be run,
- the **programming language** it is written in, or even
- **who** built it.

# WEB SERVICES - SOAP BASED

Discovery

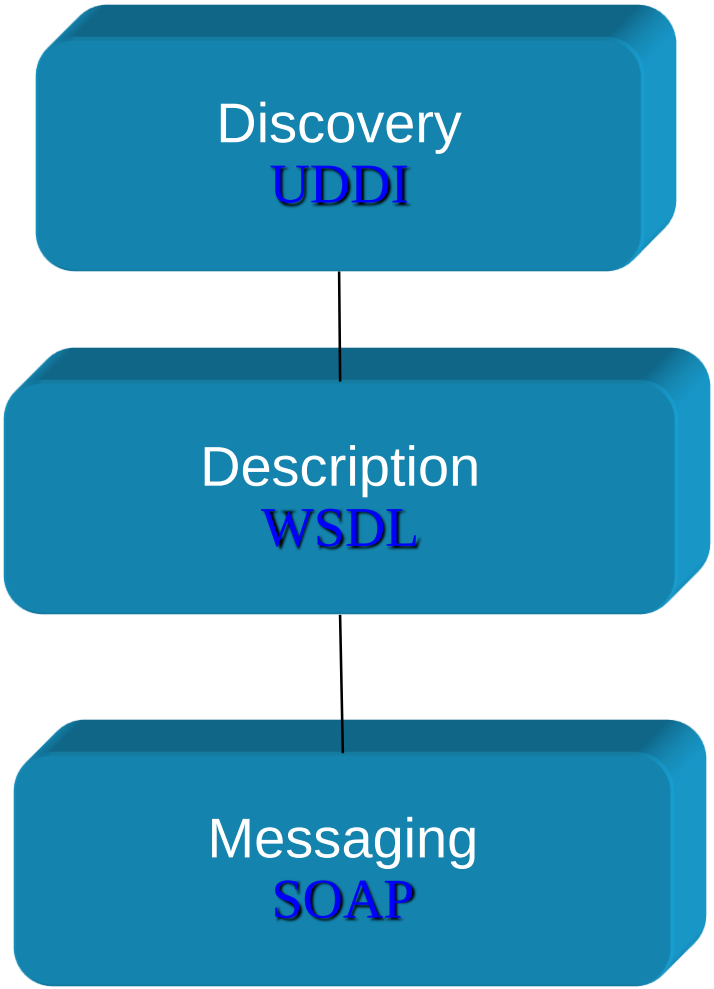
- Where is the service?

Description

- What service does it offer?
- How do I use it?

Messaging

- Let's communicate!



# STANDARD IS KEY

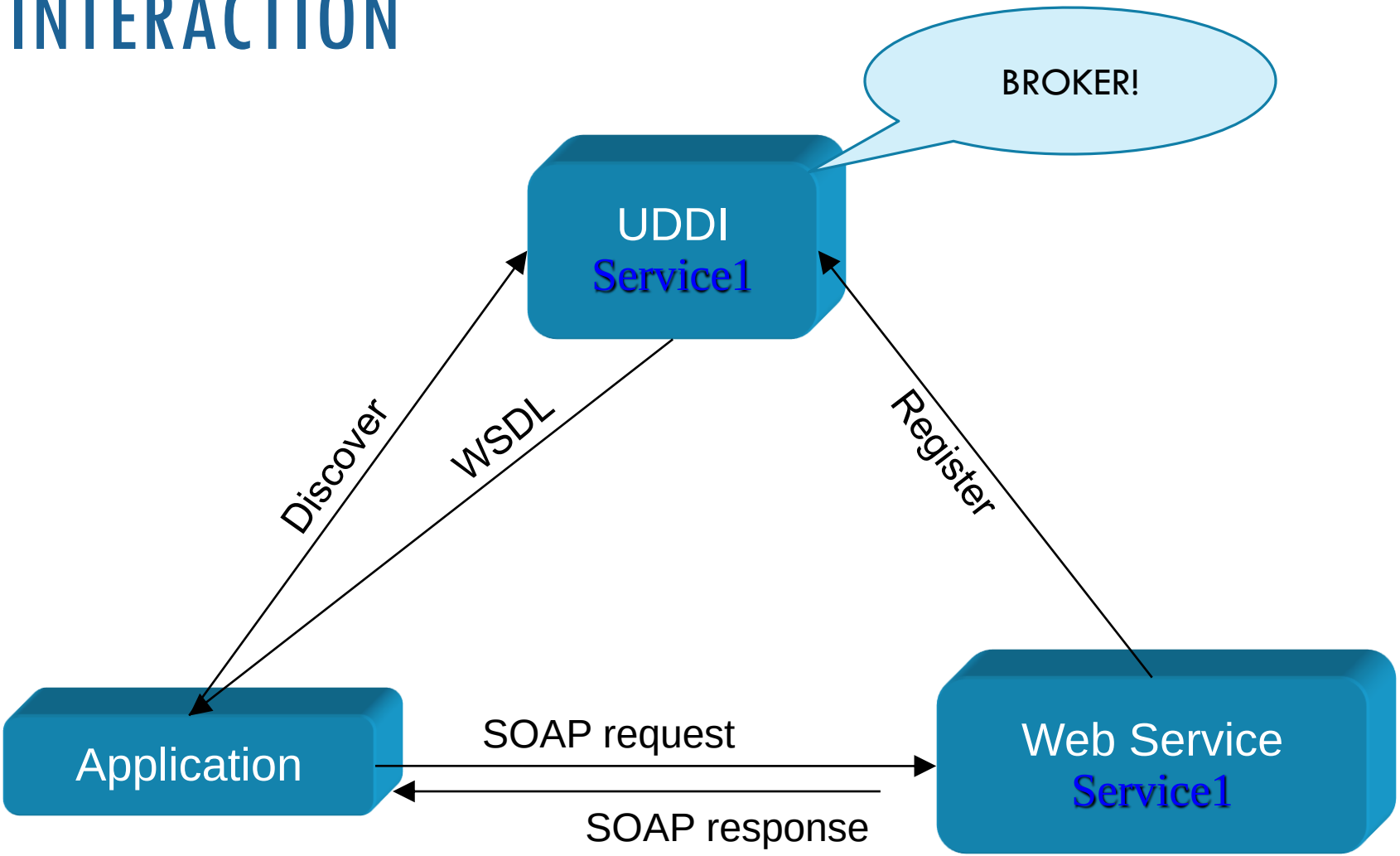
WSDL is used to describe the **function(s)** that an application will be calling documenting in a standard way its entry points, parameters and output

XML is used to carry the values of **parameters** and the **outputs** of the function

SOAP is used as the **messaging protocol** that carries content (XML) over a network transport (typically HTTP)

HTTP is used as the **network transport layer**

# INTERACTION



# WSDL: WEB SERVICE DESCRIPTION LANGUAGE

- WSDL (Web Services Description Language) is a public description of the interfaces offered by a web service.
- Expressed in XML
- Describes services as a set of endpoints
  - Document-oriented
  - Procedure-oriented

XML grammar

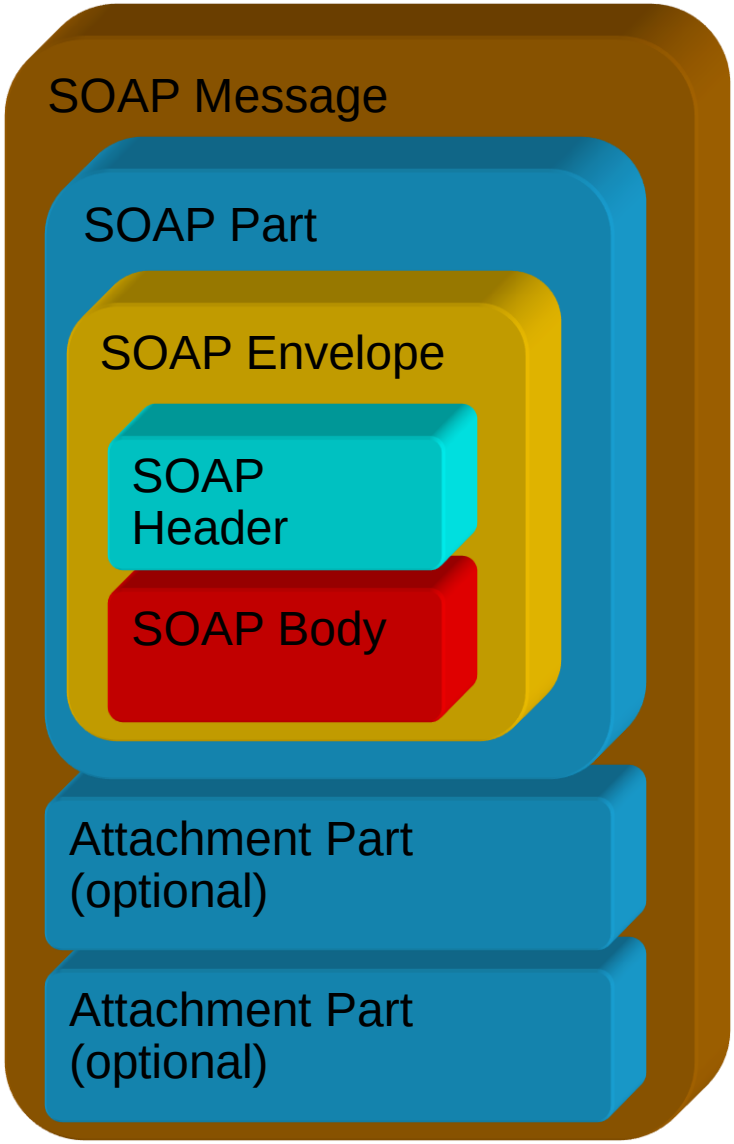
- <definitions>: root WSDL element
- <types>: data types transmitted (starts with XML Schema specifications)
- <message>: messages transmitted
- <portType>: functions supported
- <binding>: specifics of transmissions
- <service>: how to access it

# SOAP: A DESCRIPTION

- Industry standard message format for sending and receiving data between a web services consumer and a web service provider

- SOAP messages are XML documents which have an envelope and:
  - Header (optional): contains information about the message such as date/time it was sent or security information
  - Body: contains the message itself

- SOAP used to stand for Simple Object Access Protocol



# GIVE IT A REST

REST: **RE**presentation **S**tate **T**ransfer

Rest-ful: Follows the REST principles

Not strictly for web services

Term used loosely as a method of sending information over HTTP without using a messaging envelope

# REPRESENTATIONAL STATE TRANSFER (REST)

- Idea: *Self-contained* requests specify what *resource* to operate on and what to do to it [Roy Fielding's PhD thesis, 2000]

- A service (in the SOA sense) whose follows the REST principles (next slide) is a RESTful service

- Ideally, RESTful URIs name the operations

# REST PRINCIPLES

[RP1] The key abstraction of information is a **resource,** named by an URL. Any information that can be named can be a resource.

[RP2] The **representation** of a resource is a sequence of bytes, plus representation metadata to describe those bytes.

[RP3] All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it **(Stateless).**

# REST PRINCIPLES (CONT'D)

[RP4] Components perform only a small set of well-defined methods on a resource producing a representation to capture the current or intended state of that resource and transfer that representation between components. These methods are global to the specific architectural instantiation of REST; for instance, all resources exposed via HTTP are expected to support each operation identically **(Uniform interface)**.

[RP5] Idempotent operations and representation metadata are encouraged in support of **caching** and representation reuse.

[RP6] The presence of intermediaries is promoted. Filtering or redirection intermediaries may also use both the metadata and the representations within requests or responses to augment, restrict, or modify requests and responses in a manner that is transparent to both the user agent and the origin server **(Links between resources)**.

# RESOURCES

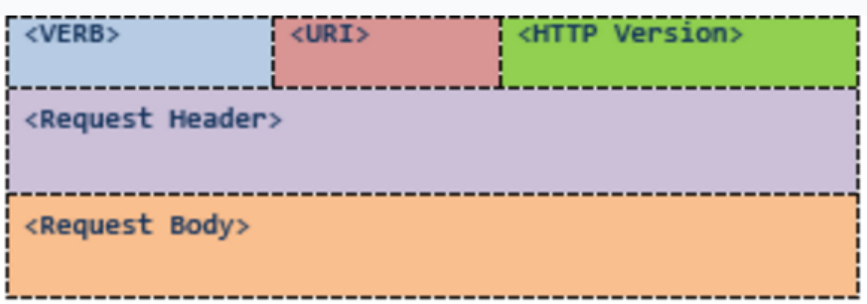
XML representation

```
<Person>
 <ID>1</ID>
 <Name>M Dinso</Name>

<Email>m.dinso@gmail.com</E
mail>
 <Country>Romania</Country>
</Person>
```

JSON representation

```
{
    "ID": "1",
    "Name": "M Dinso",
    "Email":
"m.dinso@gmail.com",
    "Country": "Romania"
}
```

# MESSAGES – HTTP REQUEST




<VERB> is one of the HTTP methods like GET, PUT, POST, DELETE, OPTIONS, etc

<URI> is the URI of the resource on which the operation is going to be performed

<HTTP Version> is the version of HTTP

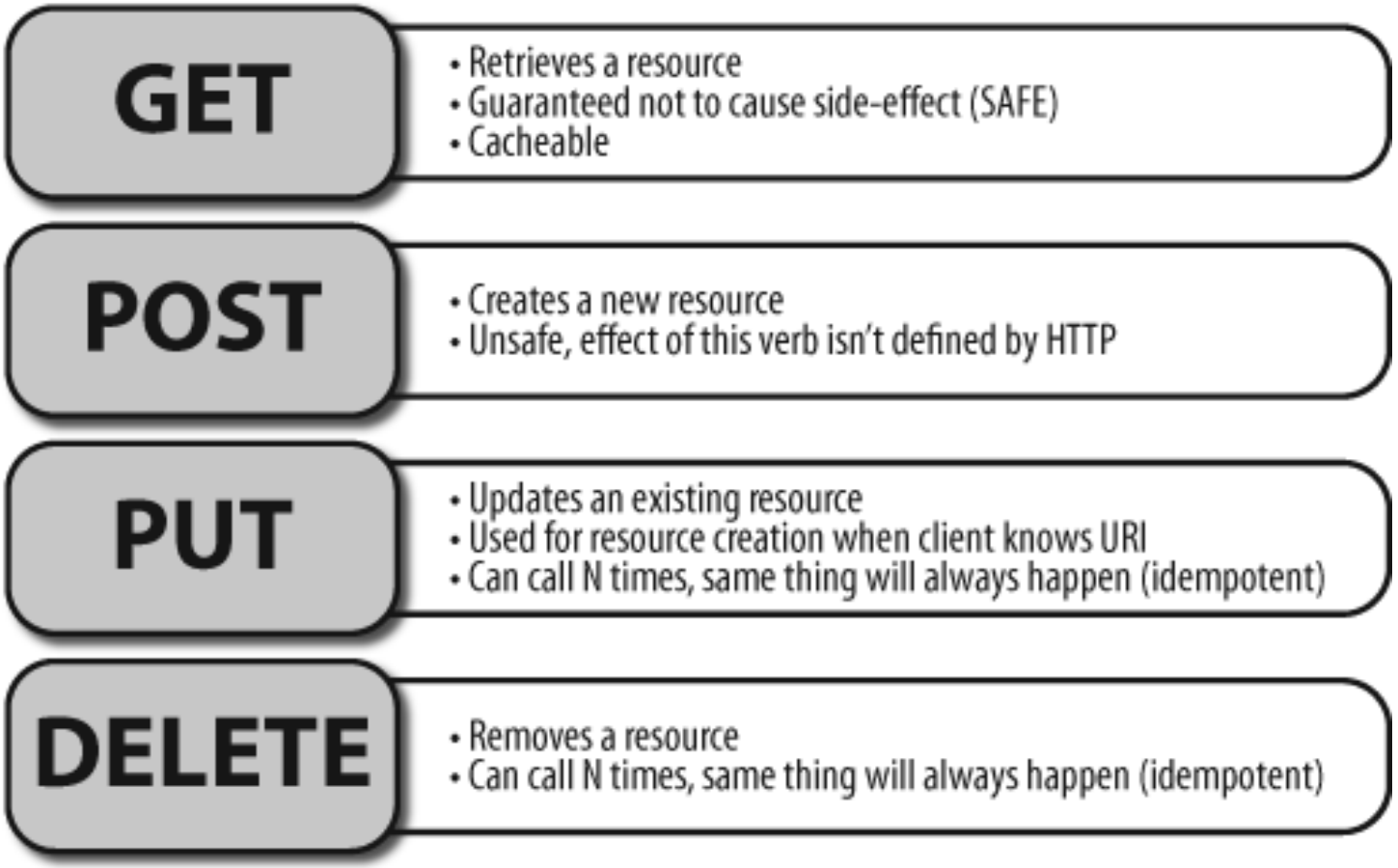
<Request Header> contains the metadata as a collection of key-value pairs of headers and their values. Ex. client type, the formats client supports, format type of the message body, cache settings for the response, etc.

<Request Body> is the actual message content. In a RESTful service, that's where the representations of resources sit in a message.

# POST REQUEST EXAMPLE

```
POST http://MyService/Person/
Host: MyService
Content-Type: text/xml; charset=utf-8
Content-Length: 123
<?xml version="1.0" encoding="utf-8"?>
<Person>
  <ID>1</ID>
  <Name>M Dinso</Name>
  <Email>m.dinso@gmail.com</Email>
  <Country>Romania</Country>
</Person>
```

# HTTP VERBS – UNIFORM INTERFACE




Taken from https://www.oreilly.com/library/view/restful-net/9780596155025/ch04.html

# STATELESSNESS

Does not maintain the application state for any client.

A request cannot be dependent on a past request and a service treats each request independently.

Example:

Request1: GET http://MyService/Persons/1 HTTP/1.1

Request2: GET http://MyService/Persons/2 HTTP/1.1

# SOAP VS. REST

| Difference | SOAP | REST |
|---|---|---|
| Style | Protocol | Architectural Style |
| Function | Function-driven: transfer structured information | Data-driven: access a resource for data |
| Data format | Only uses XML | Permits several data formats: HTML, XML, plain text, JSON |
| Security | Supports WS-Security and SSL | Supports SSL and HTTPS |
| Bandwidth | Requires more resources and bandwidth | Requires fewer resources and is lightweight |
| Data cache | Cannot be cached | Can be cached |
| Payload handling | Has a strict communication contract and needs knowledge of everything before any interaction | Needs no knowledge of the API |
| Transactional support | Has built-in ACID compliance | Lacks ACID compliance |

# WHEN TO USE REST?

- **Limited resources and bandwidth**
- **Statelessness** – If there is no need to maintain a state of information from one request to another
- **Caching** – If there is a need to cache a lot of requests
- **Ease of coding**
- **Challenges**
  - **Lack of Security** – REST does not impose any sort of security like SOAP. This is why REST is very appropriate for public available URL's, but when it comes down to confidential data being passed between the client and the server, REST is the worst mechanism to be used for web services.
  - **Lack of state** – Most web applications require a stateful mechanism. For example, if you had a purchasing site which had the mechanism of having a shopping cart, it is required to know the number of items in the shopping cart before the actual purchase is made

# WHEN TO USE SOAP?

- **Asynchronous processing and subsequent invocation**
- **A Formal means of communication**
- **Stateful operations**
- **Challenges**
  - **WSDL file:** the tight contract between the client and the server and one change could cause a large impact, on the whole, client applications
  - **Document size** – Large messages can be a problem for limited bandwidth

**SOAP Services are like envelopes while REST services are like postcards**

**SOAP Services cannot use REST while REST can use SOAP.**

# WRAP-UP

- The business/domain logic layer contains the independent logic
- Several approaches to model the business logic
- Domain driven design helps
  - Dividing a complex domain into subdomains
  - Identify boundary contexts
  - Identify entities and value objects linked into aggregates
  - Better dependency management
  - Better transaction management
- Service orientation