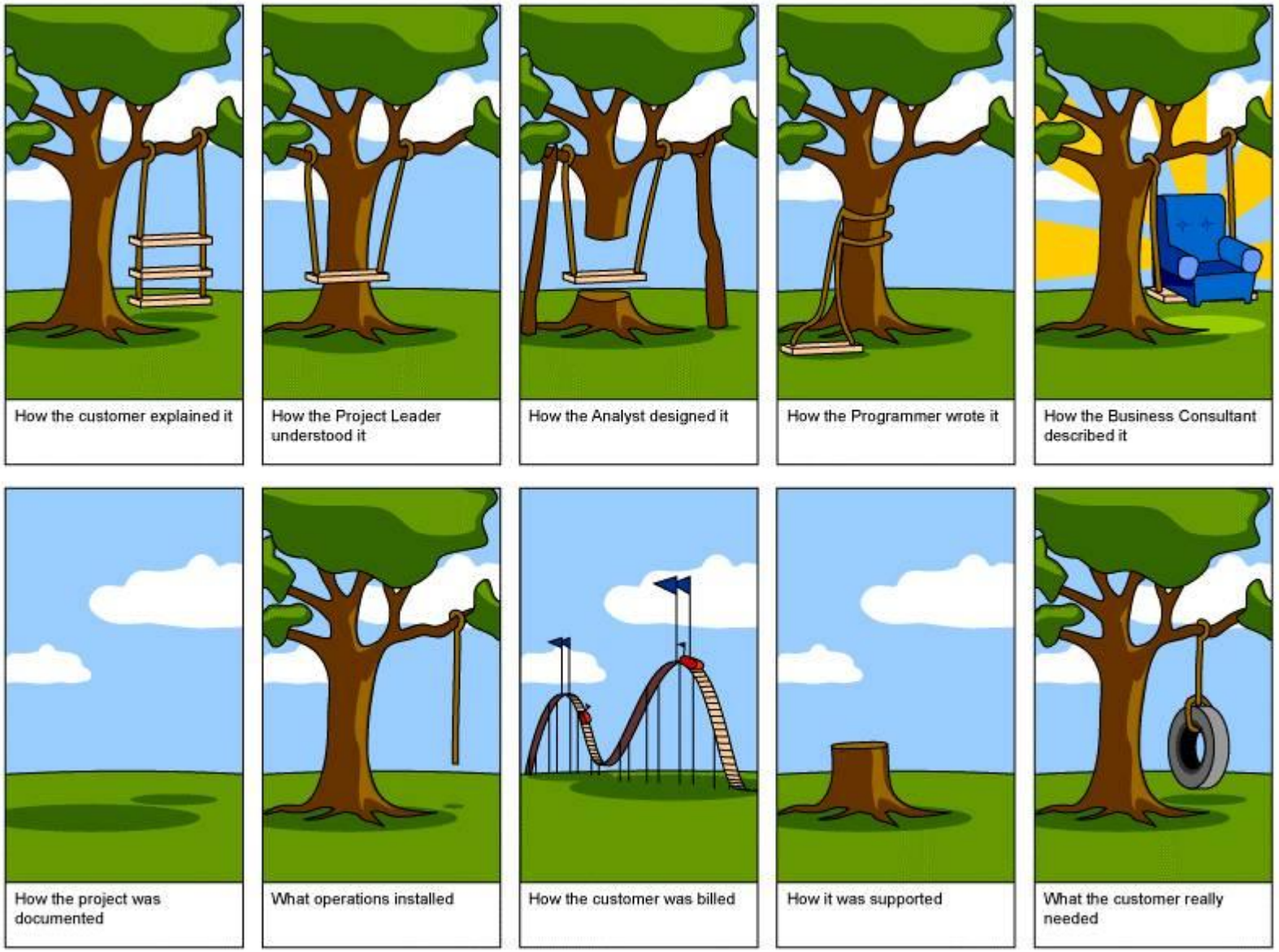
Politecnico
di Milano


# Software Design & Software Architecture

# Software design
## (how IT projects really work)



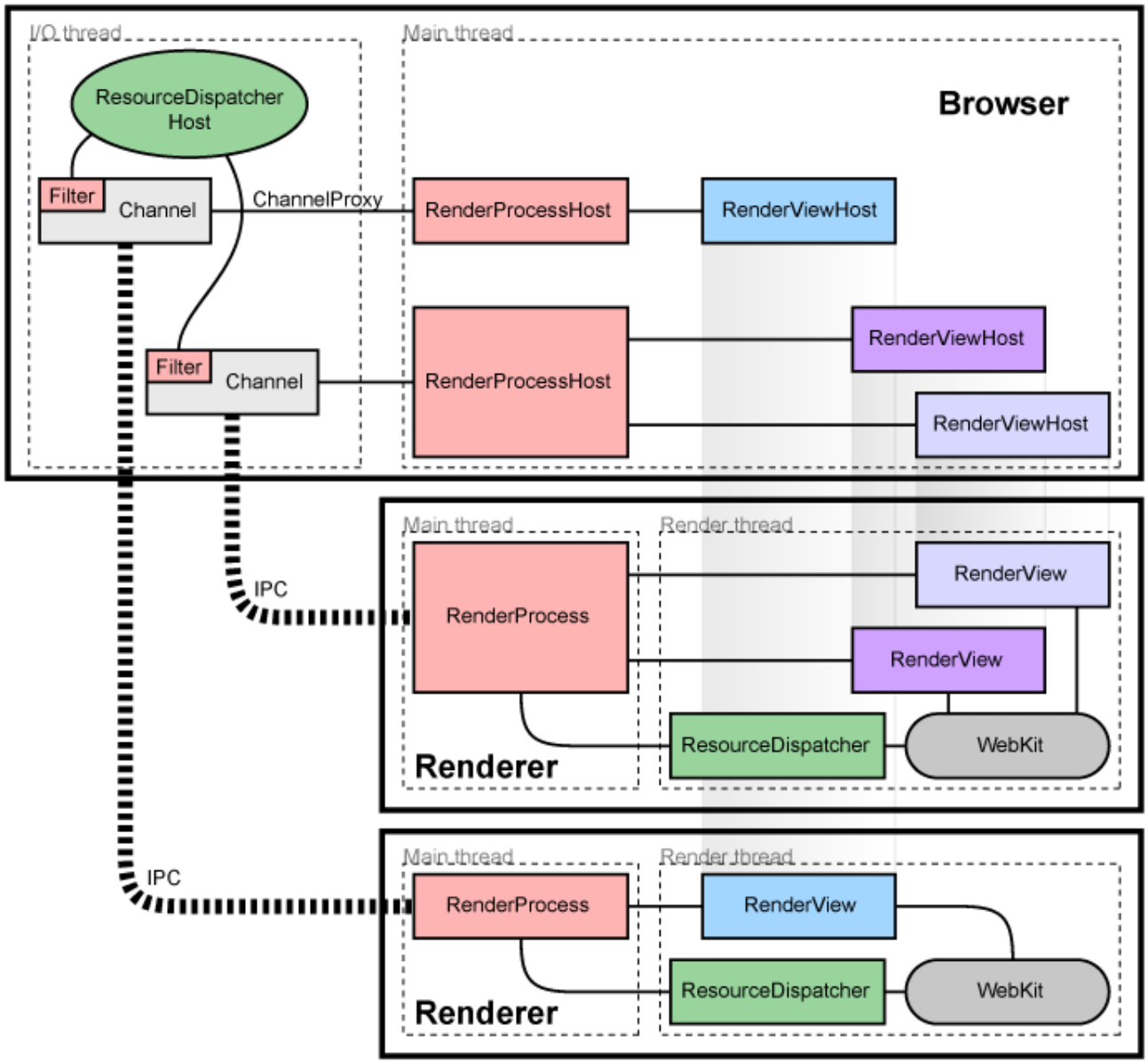
# Software architecture, definition (1)


The architecture of a software system defines that system in terms of computational components and interactions among those components.

(from Shaw and Garlan, *Software Architecture, Perspectives on an Emerging Discipline*, Prentice-Hall, 1996.)

# Example – Chromium Architecture

https://www.chromium.org/developers/design-documents/multi-process-architecture



# Software Architecture, definition (2)


The software architecture of a system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.

(from Bass, Clements, and Kazman, *Software Architecture in Practice*, SEI Series in Software Engineering. Addison-Wesley, 2003.)

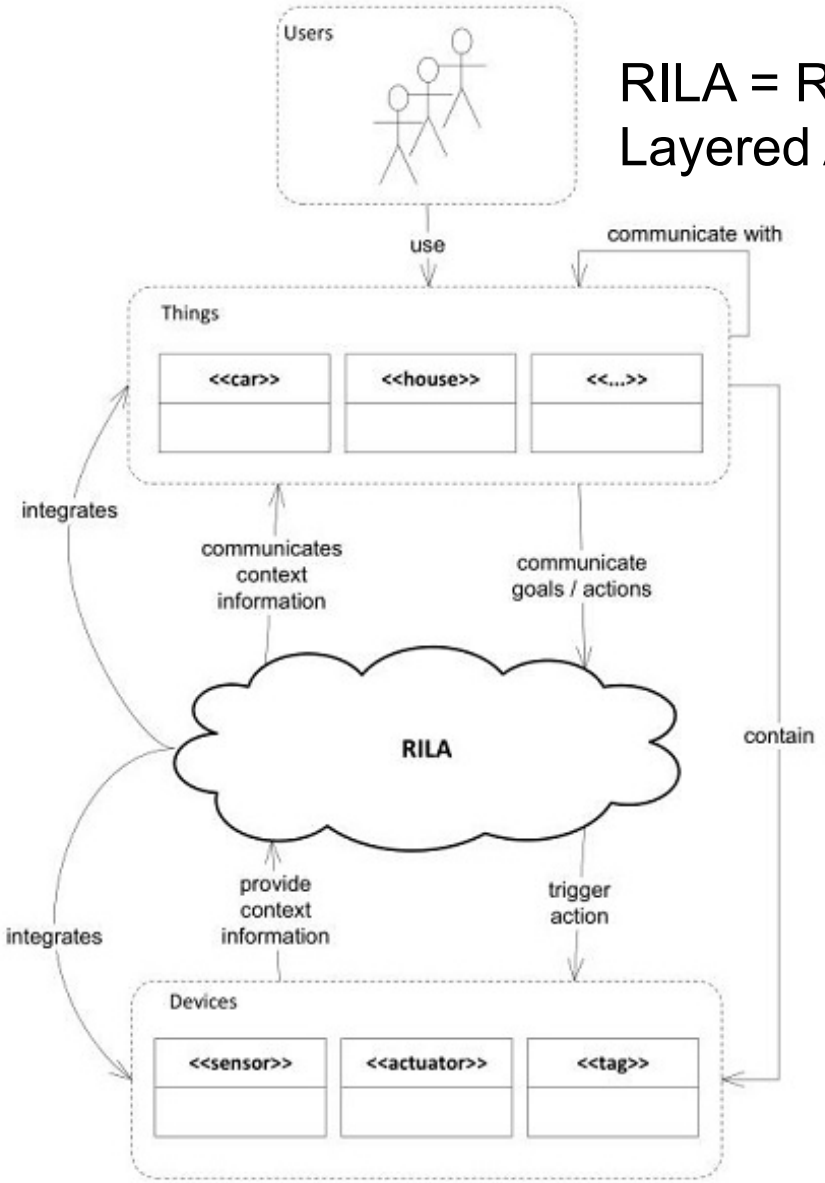
# Software Architecture


- Important issues raised in this definition:
  - multiple system structures;
  - externally visible (observable) properties of components.

- The definition does not include:
  - the process;
  - rules and guidelines;
  - architectural styles.

# Example – an Internet of Things reference architecture (1)

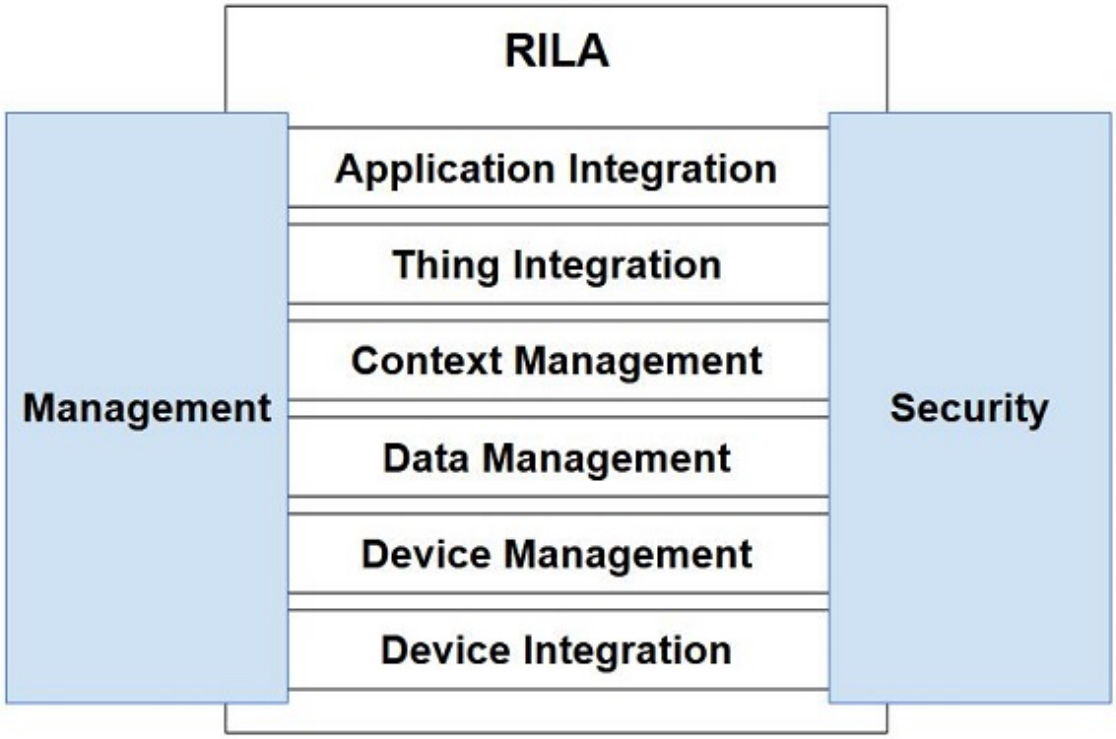
https://www.infoq.com/articles/internet-of-things-reference-architecture




RILA = Reference IoT Layered Architecture

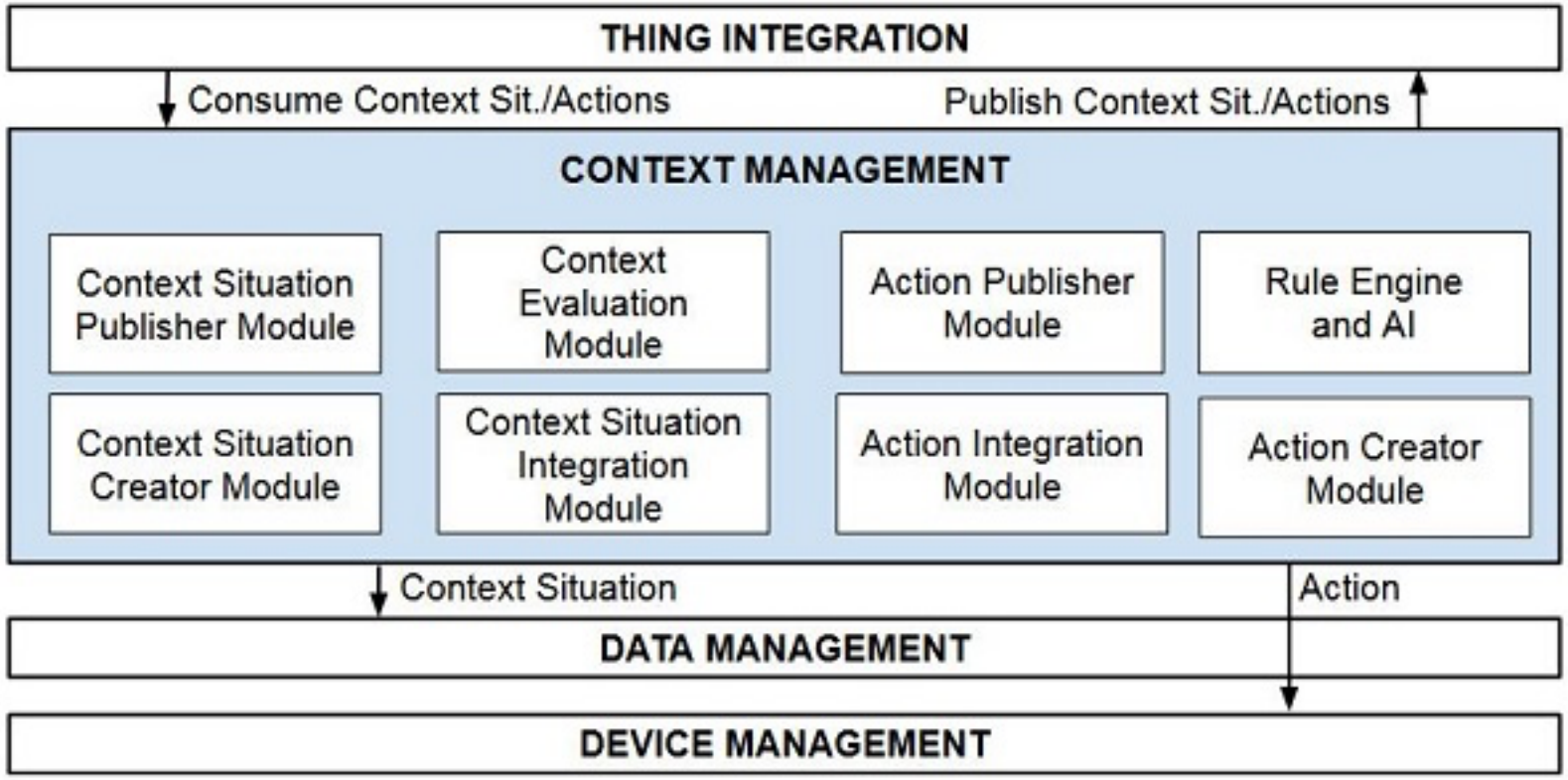
# Example – an Internet of Things reference architecture (2)

https://www.infoq.com/articles/internet-of-things-reference-architecture



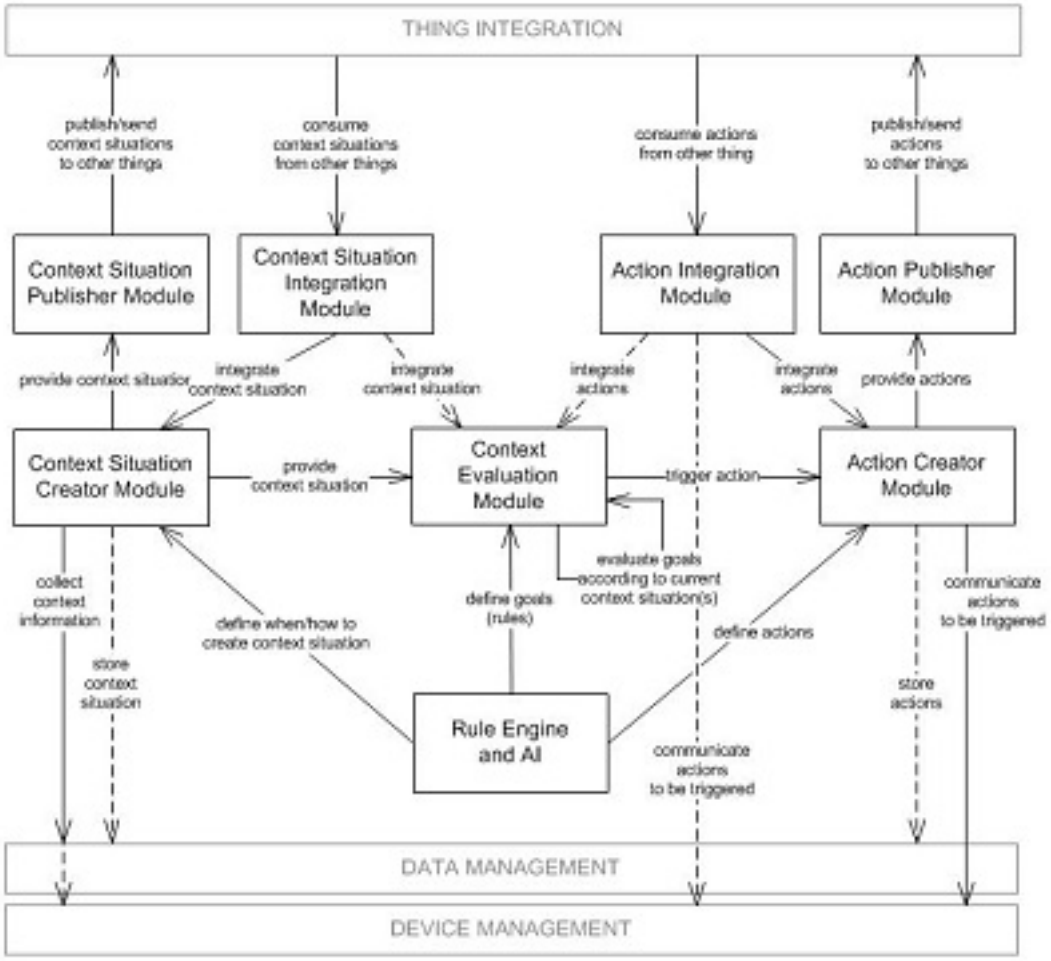
# Example – an Internet of Things reference architecture (3)

https://www.infoq.com/articles/internet-of-things-reference-architecture



# Example – an Internet of Things reference architecture (4)

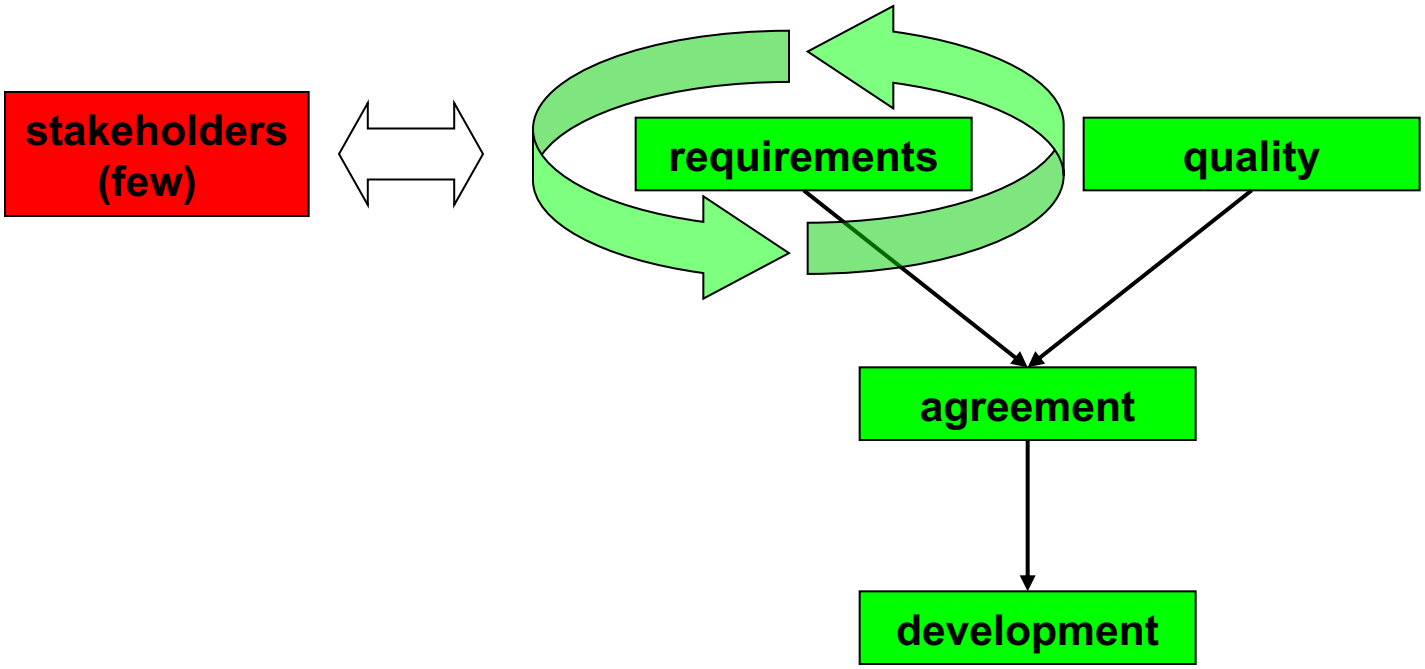
https://www.infoq.com/articles/internet-of-things-reference-architecture



# Why Is Architecture Important?


- Architecture is the vehicle for stakeholder communication
- Architecture manifests the earliest set of design decisions
  - Introduces constraints on implementation
  - Dictates organizational structure
  - Inhibits or enables quality attributes
- Architecture is a transferable abstraction of a system
  - Product lines share a common architecture
  - Allows for template-based development
  - Basis for training

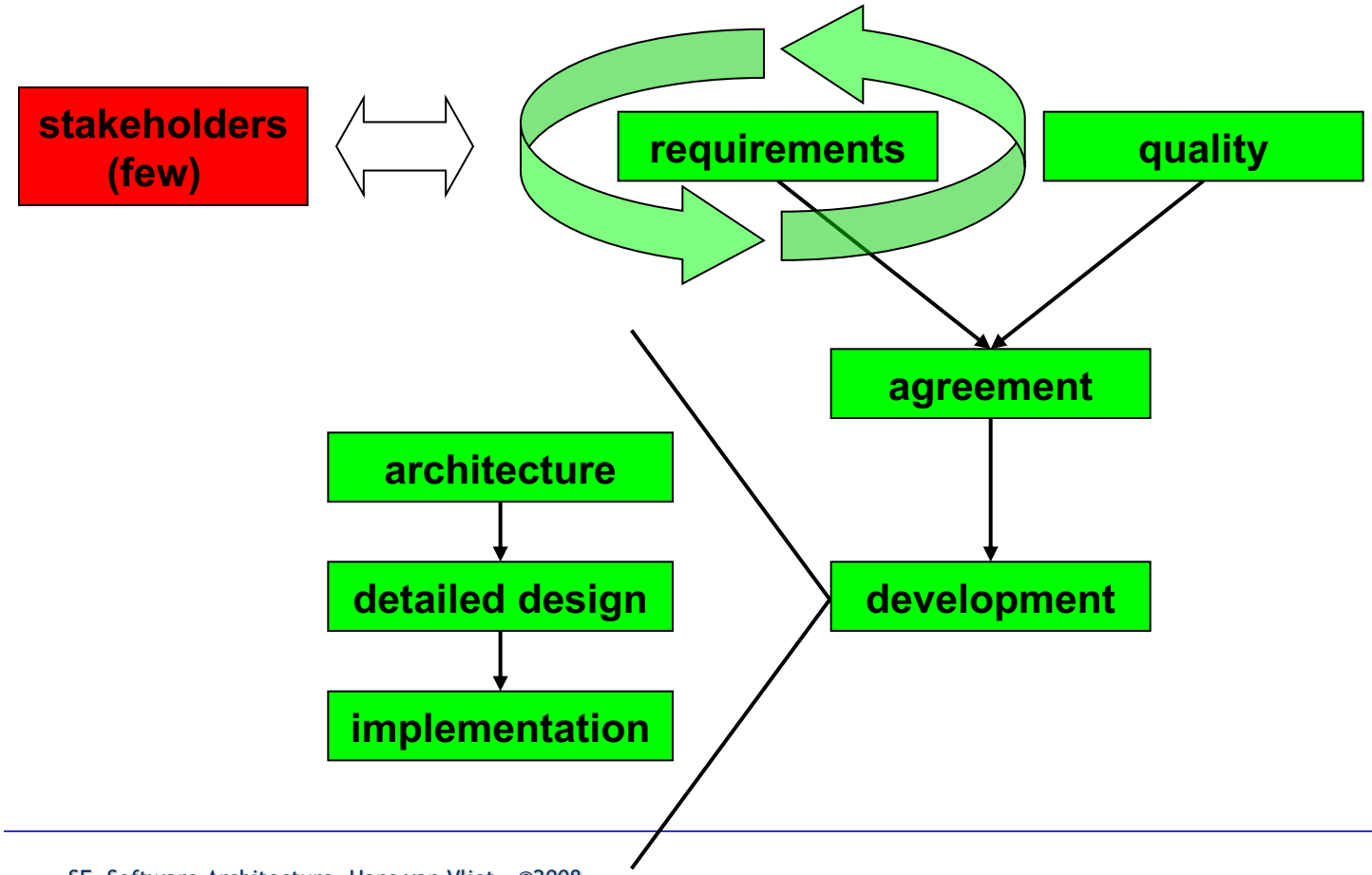
# Pre-architecture life cycle



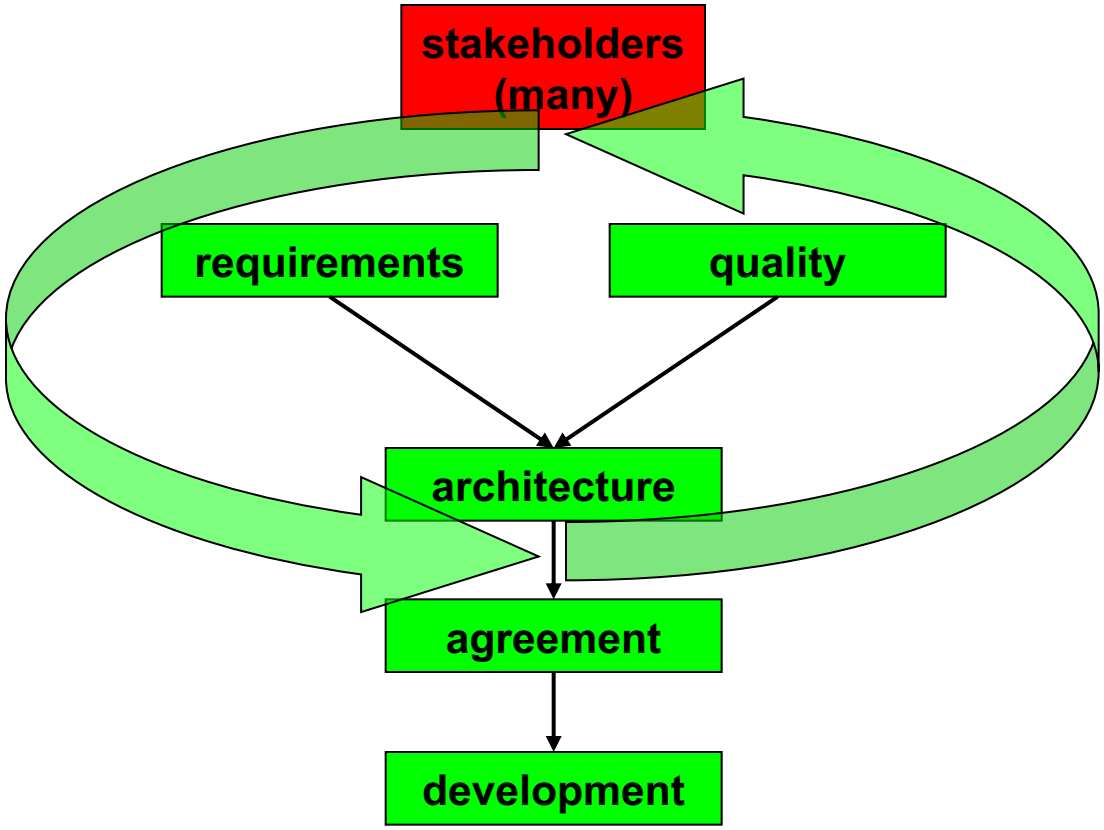
# Characteristics


- Iteration mainly on functional requirements
- Few stakeholders involved
- No balancing of functional and quality requirements

# Adding architecture, the easy way



# Giving a more crucial role to the architecture



# Characteristics


- Iteration on both functional and quality requirements
- Many stakeholders involved
- Balancing of functional and quality requirements


# The architecture is intrinsic to our software!

# A simple example: a trivial prototype of a “banking application” - Account


```
import java.rmi.*;
public interface Account extends Remote {
    public float balance() throws RemoteException;
}

import [...]
public class AccountImpl extends UnicastRemoteObject
                         implements Account {
    private float _balance;
    public AccountImpl(float balance) throws RemoteException {
        _balance = balance;
    }
    public float balance() throws RemoteException {
        return _balance;
    }
}
```

# A simple example: a trivial prototype of a “banking application” - AccountFactory


```
import java.rmi.*;
public interface AccountFactory extends Remote {
    public Account createAccount(String userName, int balance)
                    throws RemoteException;
    public int getLoad() throws RemoteException;
}

import [...]
public class AccountFactoryImpl extends UnicastRemoteObject
                                implements AccountFactory {
    private String hostName;
    private int load;
    public AccountFactoryImpl(String _hostName)
            throws RemoteException {
        hostName = _hostName;
        load = 0;
    }
```

# A simple example: a trivial prototype of a “banking application” - AccountFactory


```
public Account createAccount(String userName, int balance)
                throws RemoteException {
    Account account = new AccountImpl(balance);
    try {
        Naming.rebind("//"+hostName+"/Account"+userName, account);
        System.out.println("Created " + userName +
                           "'s account: " + account);
    } catch (Exception e) {
      e.printStackTrace();
    }
    load++;
    return account;
}

public int getLoad() throws RemoteException { return load; }
```

# A simple example: a trivial prototype of a “banking application” - AccountFactory


```
public static void main(String[] args) {
    String _hostName;
    try {
        System.setSecurityManager(new RMISecurityManager());
        _hostName = InetAddress.getLocalHost().getHostName();
        AccountFactoryImpl server =
                                new AccountFactoryImpl(_hostName)
        Naming.rebind("//"+_hostName+"/AccountFactory", server);
        System.out.println("Factory bound on "+_hostName);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
}
```

# A simple example: a trivial prototype of a "banking application" - AccountManager


```
import java.rmi.*;
public interface AccountManager extends Remote {
        public Account open(String name) throws RemoteException;
}

import [...]
public class AccountManagerImpl extends UnicastRemoteObject
                                implements AccountManager {
    private Dictionary _accounts = new Hashtable();
    private String hostName;
    private Vector factoryList;
    public AccountManagerImpl(String _hostName)
            throws RemoteException {
        hostName = _hostName;
        factoryList = new Vector();
    }
```

# A simple example: a trivial prototype of a “banking application” - AccountManager


```
public void addFactory(String hostName) {
    try {
        AccountFactory fct =
            (AccountFactory) Naming.lookup("//" + hostname +
                                           "/AccountFactory");
      factoryList.addElement(fct);
    } catch (Exception e) {
        System.out.println("Error: Factory unreachable");
      e.printStackTrace();
    }
}
```

# A simple example: a trivial prototype of a “banking application” - AccountManager


```
public AccountFactory selectFactory() {
    int minLoad, load, index = 0;
    AccountFactory f;
    try {
        Enumeration e = factoryList.elements();
        f = (AccountFactory) e.nextElement();
        minLoad = f.getLoad();
        index = factoryList.indexOf(f);
        while (e.hasMoreElements()) {
            f = (AccountFactory) e.nextElement();
            load = f.getLoad();
            if (minLoad > load) {
                minLoad = load; index = factoryList.indexOf(f);
    }    }    } catch (Exception e) {  e.printStackTrace(); }
    return (AccountFactory) factoryList.elementAt(index);
}
```

# A simple example: a trivial prototype of a “banking application” - AccountManager


```
public Account open(String name) throws RemoteException {
    AccountFactory factory;
    /* ...Security checks... */
    Account account = (Account) _accounts.get(name);
    if (account == null) {
        factory = selectFactory();
        try {
            account = factory.createAccount(name, 0);
            _accounts.put(name, account);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    return account;
}
```

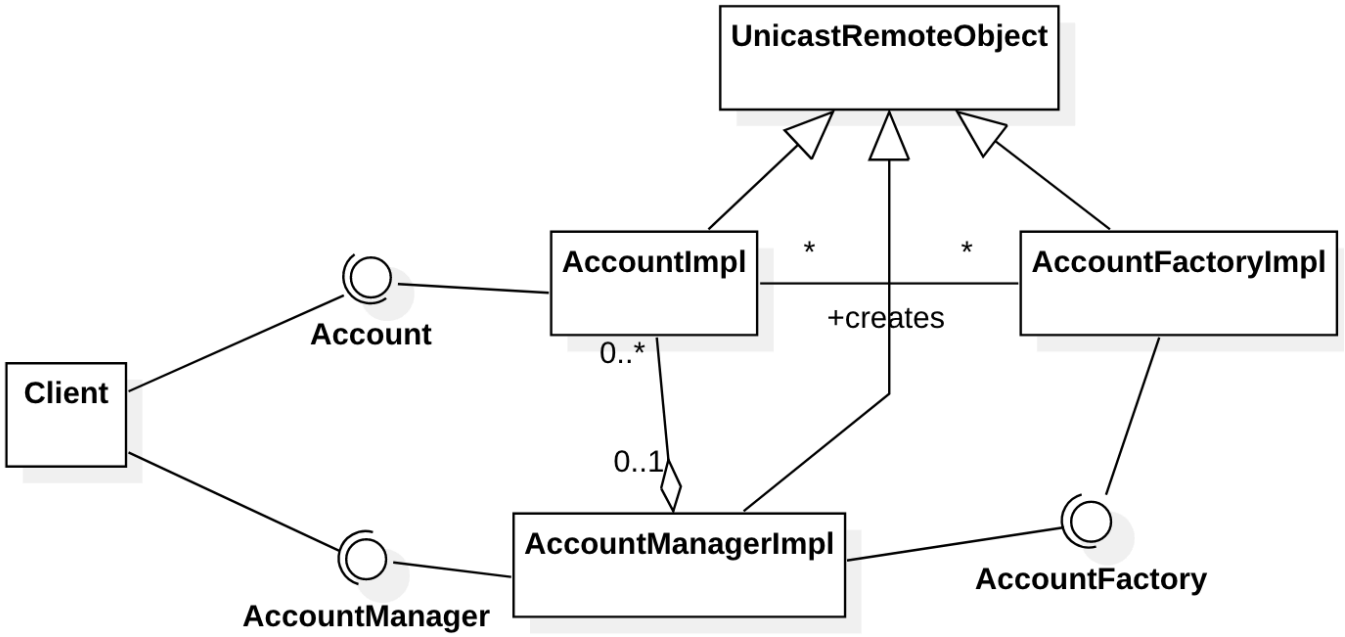
# A simple example: a trivial prototype of a “banking application” - AccountManager


```
public static void main(String[] args) {
    String _hostName;
    try {
        System.setSecurityManager(new RMISecurityManager());
        _hostName = InetAddress.getLocalHost().getHostName();
        AccountManagerImpl server =
            new AccountManagerImpl(_hostName);
        server.addFactory(args[0]);
        Naming.rebind("//"+_hostName+"/AccountManager", server);
        System.out.println("Server bound on "+_hostName);
    } catch (Exception e) {
        e.printStackTrace();
    }
  }
}
```

# A simple example: a trivial prototype of a “banking application” - Client

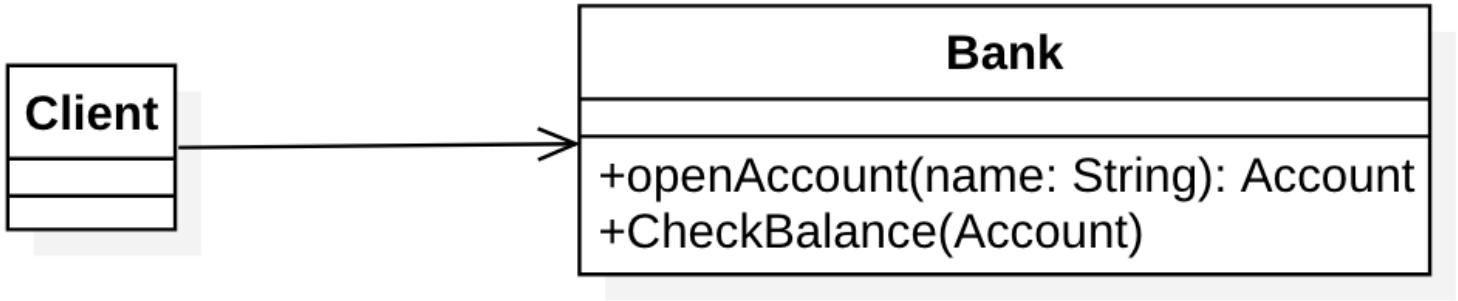

```
import java.rmi.*;
public class Client {
    public static void main (String[] args) {
        try {
            System.setSecurityManager(new RMISecurityManager());
            System.out.println("Looking up server...");
            AccountManager server =
                (AccountManager) Naming.lookup( "//" + args[0] +
                                                "/AccountManager );
            System.out.println("Server bound...");
            String name = "Jack B. Quick";
            Account account = server.open(name);
            float balance = account.balance();
            System.out.println ("The balance in " + name +
                                "'s account is $" + balance);
        } catch (Exception e) {  e.printStackTrace(); }
} }
```

# A simple example: a trivial prototype of a “banking application” - Low level design view



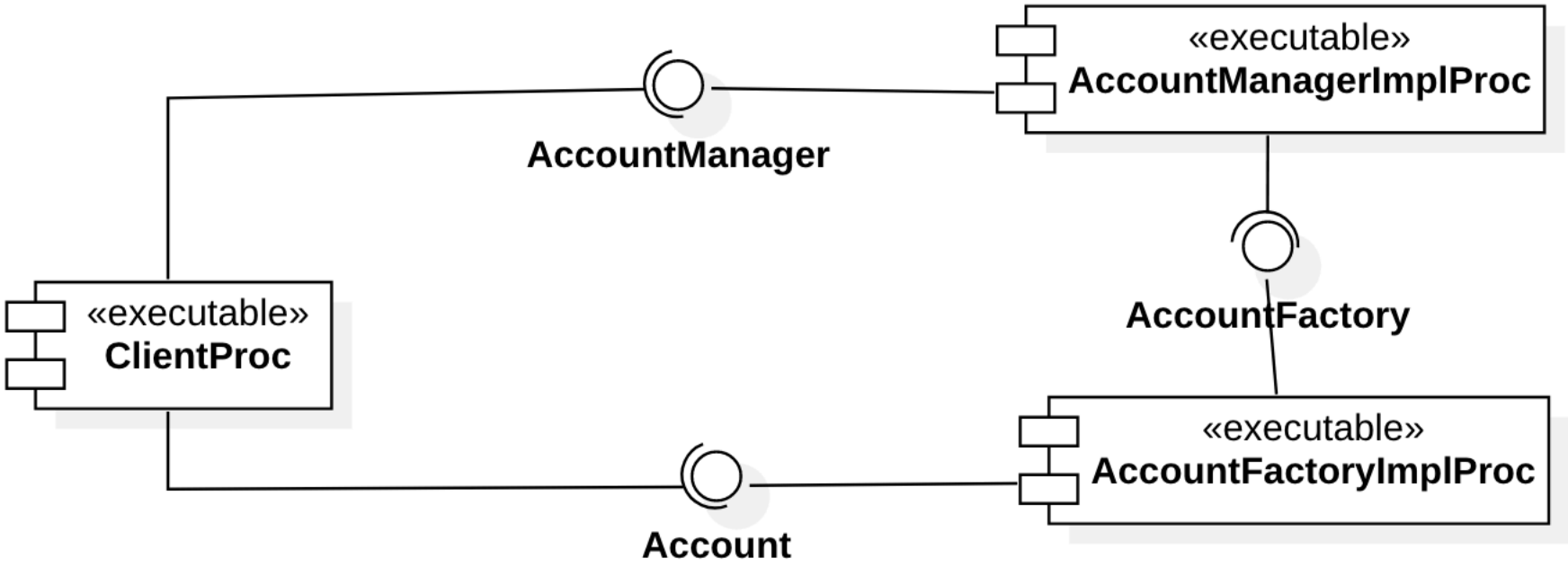

- What info do I get from this?
  - It uses remote objects
  - Client uses two interfaces
  - The other interface is used by AccountManager

# A simple example: a trivial prototype of a “banking application” - Higher level view

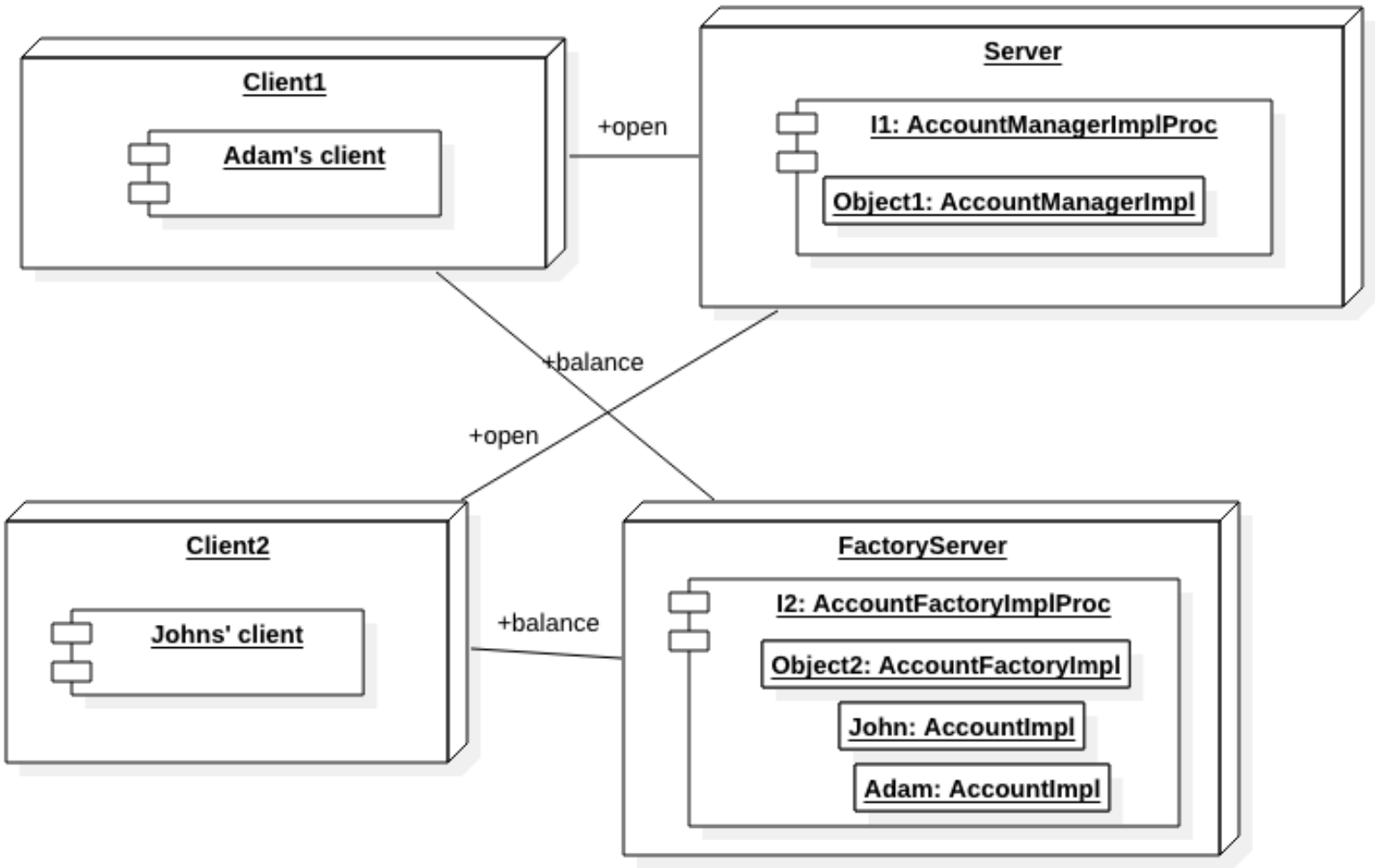



- A Bank Server is able to receive from clients two requests
  - Open account: it returns an account. If the account does not exist, the system creates it
  - Balance: it checks the balance of a previously opened account
- Other details are made transparent:
  - The factory
  - The distinction between services offered by the manager and those offered by the account

# A simple example: a trivial prototype of a “banking application” - Executables (*deployment units*)



# A simple example: a trivial prototype of a “banking application” - Processes and threads (*runtime units*)



# What happens if changes are applied on the source code?

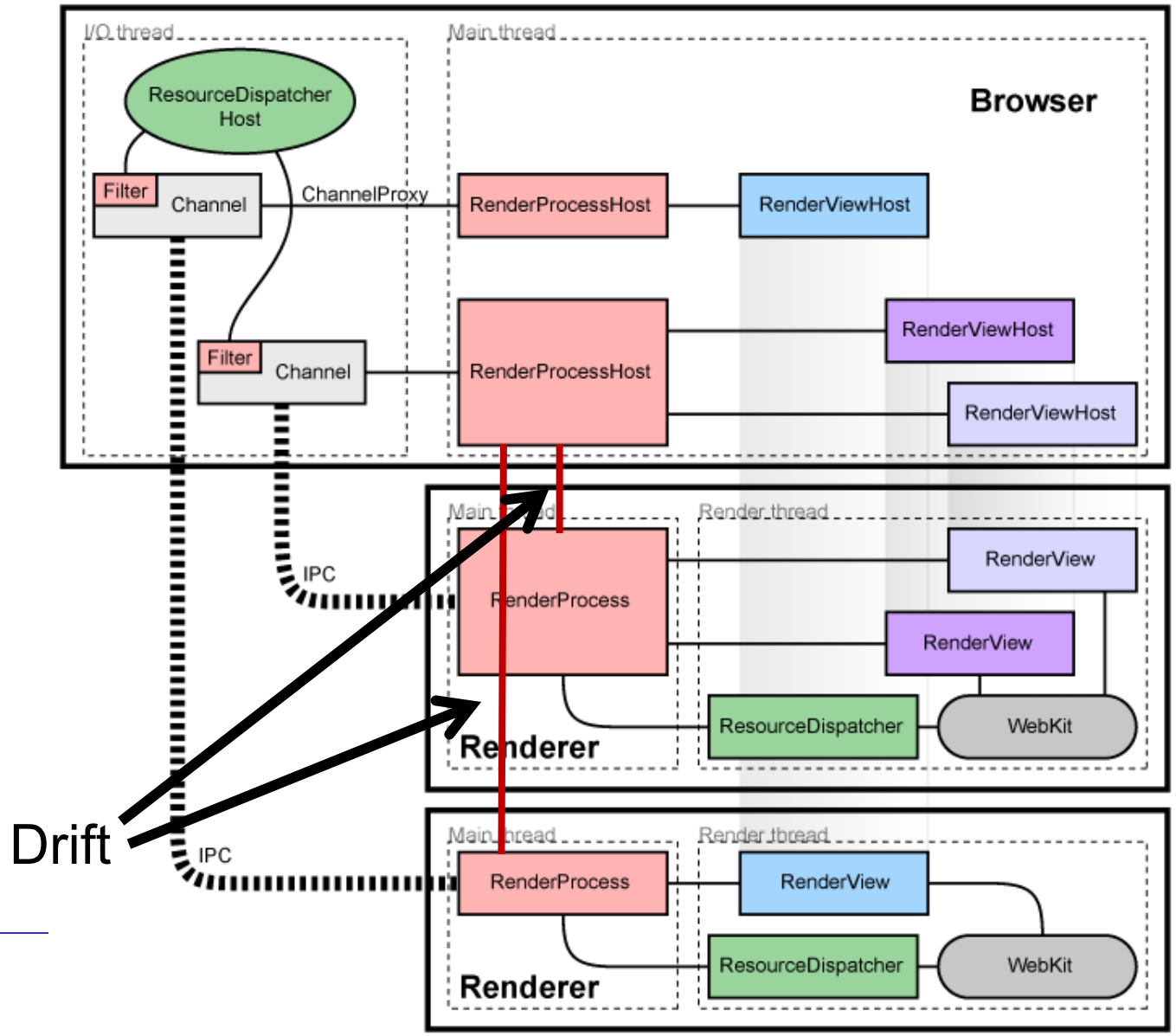

- Architectural degradation
  - Changes in the software system are not reflected in the corresponding architectural description
- Why does this happen?
  - Lack of a documented architecture
  - Inadequate processes and supporting tools
  - Developer sloppiness
  - Perception of short deadlines

# Drift and erosion


- Two kinds of degradation
  - **Drift:** the change is not included nor implied by the architecture but it does not necessarily imply its outright violation
  - **Erosion**: the change violates the described architecture

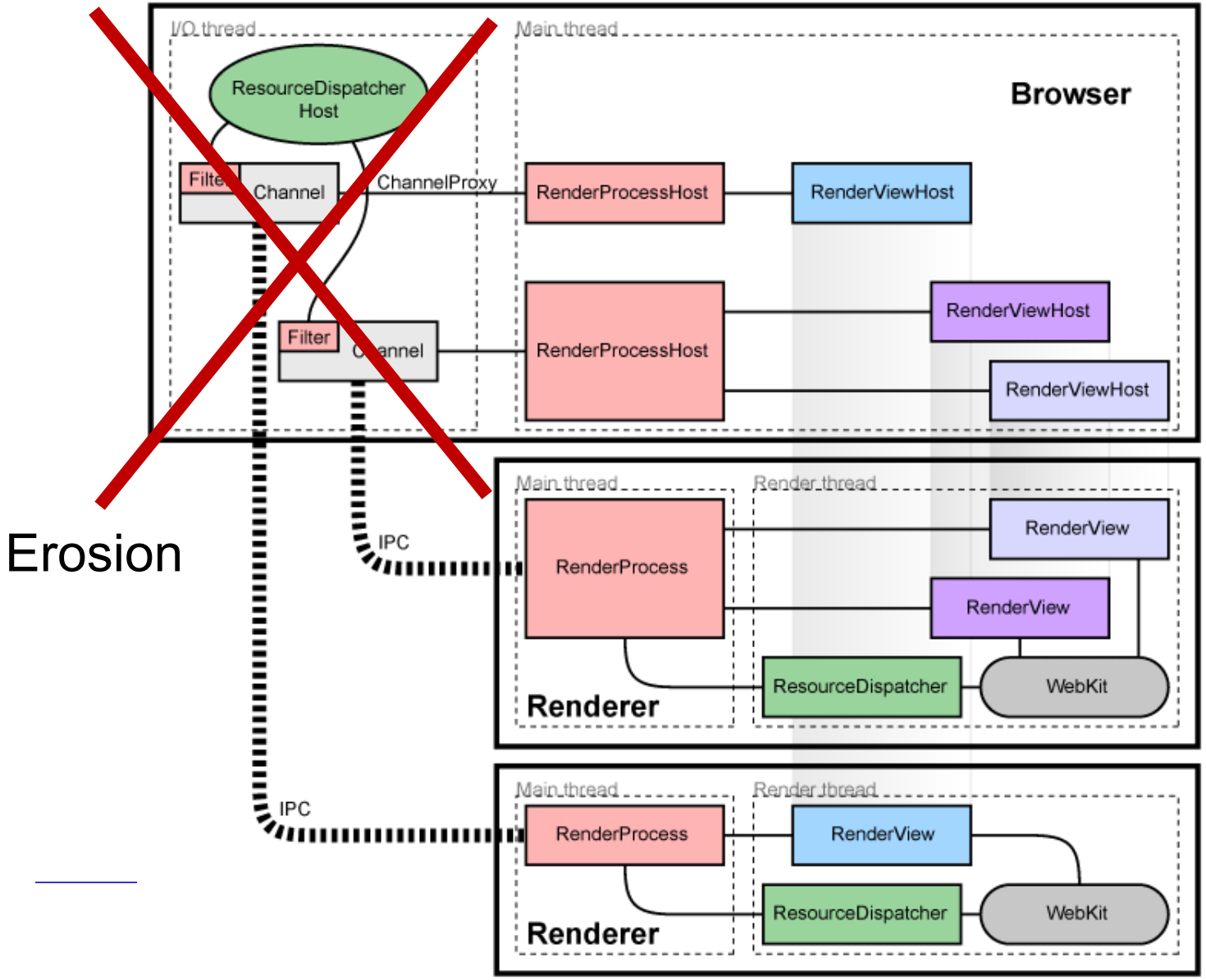
# Example – Chromium Architecture

https://www.chromium.org/developers/design-documents/multi-process-architecture



# Example – Chromium Architecture

https://www.chromium.org/developers/design-documents/multi-process-architecture



# Software Architecture & Quality


- The notion of quality is central in software architecting: a software architecture is devised to gain insight in the qualities of a system at the earliest possible stage.
- Some qualities are observable via execution: performance, security, availability, functionality, usability
- And some are not observable via execution: modifiability, portability, reusability, integrability, testability

Politecnico
di Milano

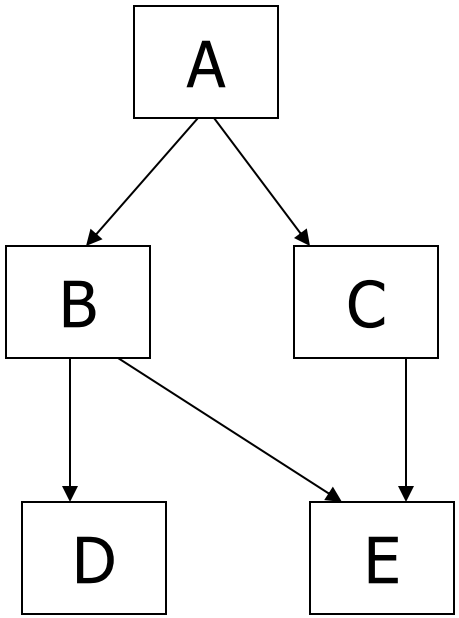

# Architectural styles

# Architectural Style

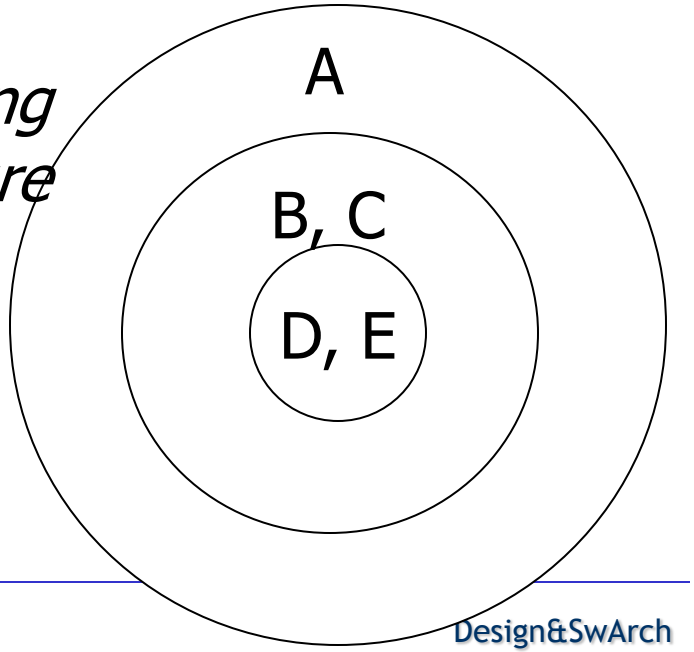
- […] an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition.” [Garland & Shaw]
- [Garland & Shaw] David Garlan and Mary Shaw, January 1994, CMU-CS-94-166, see "*An Introduction to Software Architecture*" at http://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf]

# Layered style

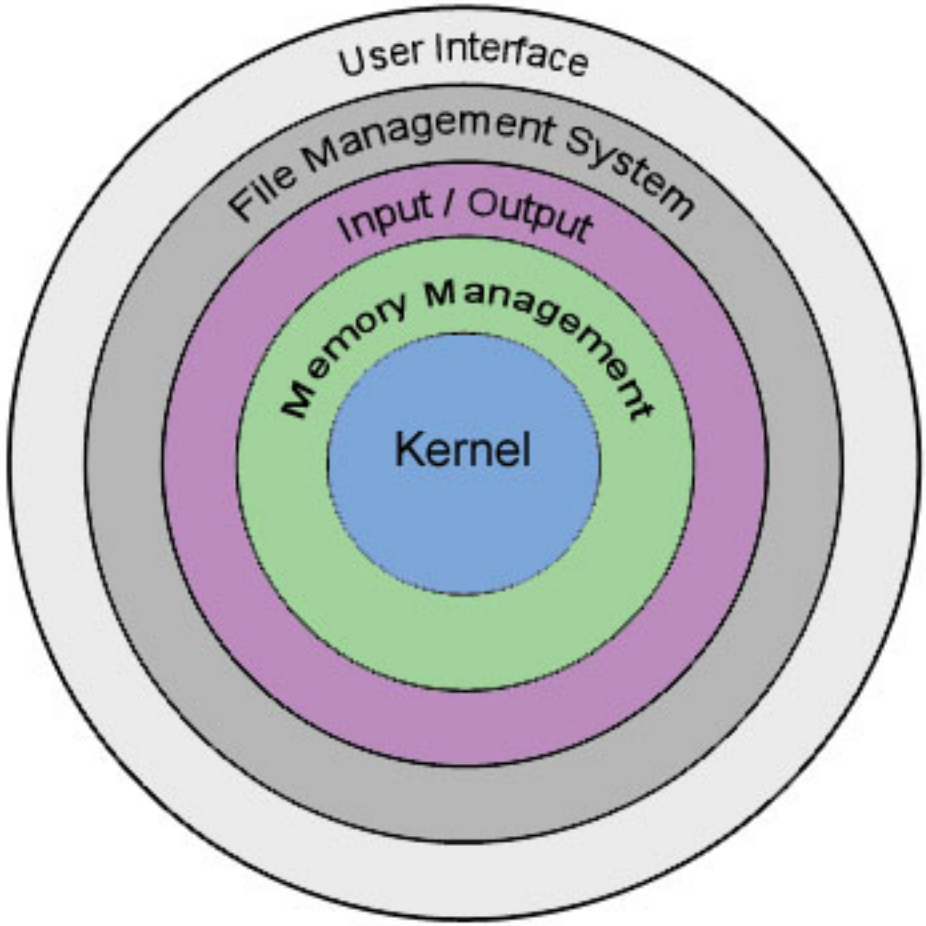
- The system is organized through abstraction levels, as a hierarchy of abstract machines
- Hierarchy is given by the USE relation



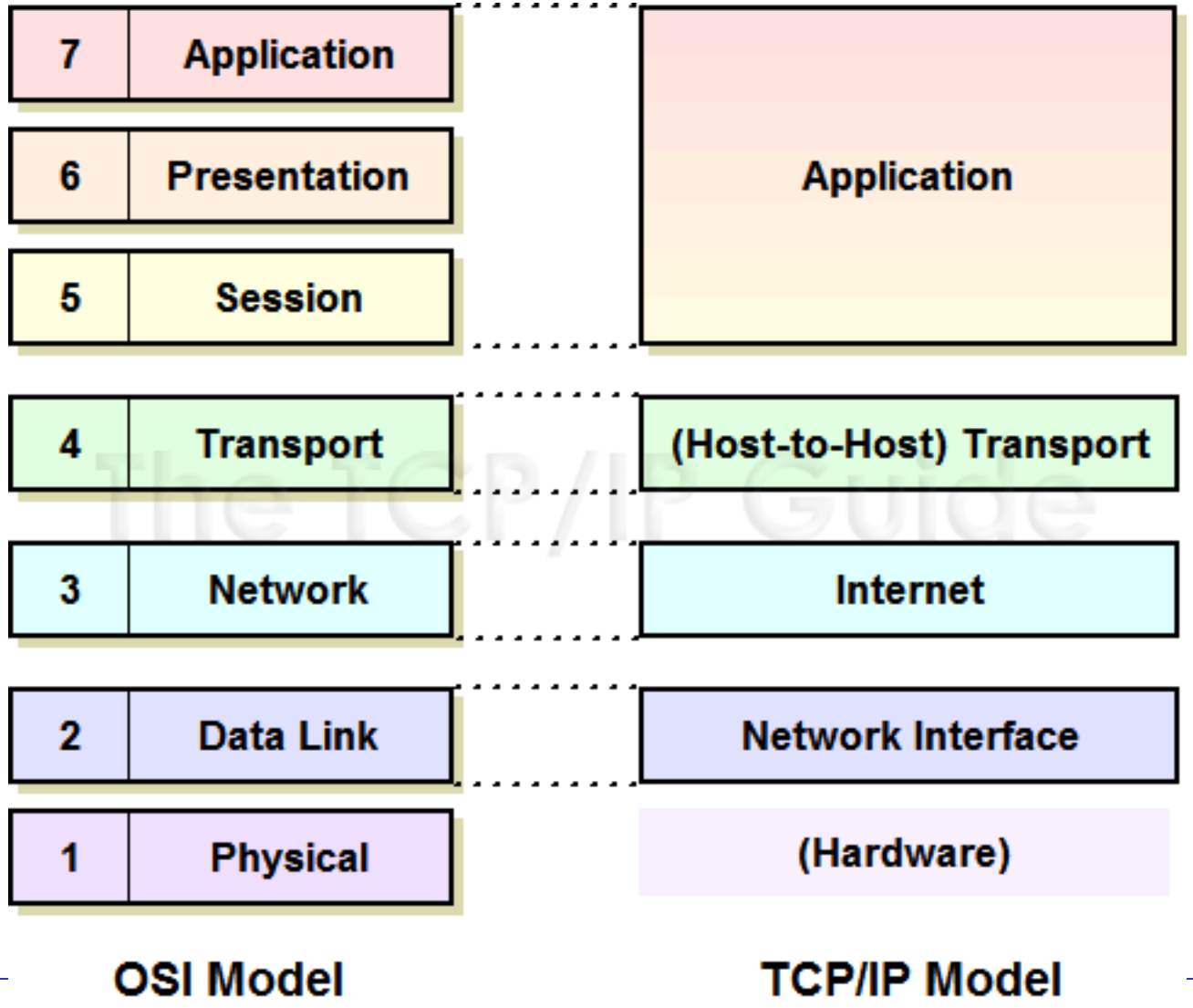

*the onion-ring structure*



# Well-known layered systems: An Operating System



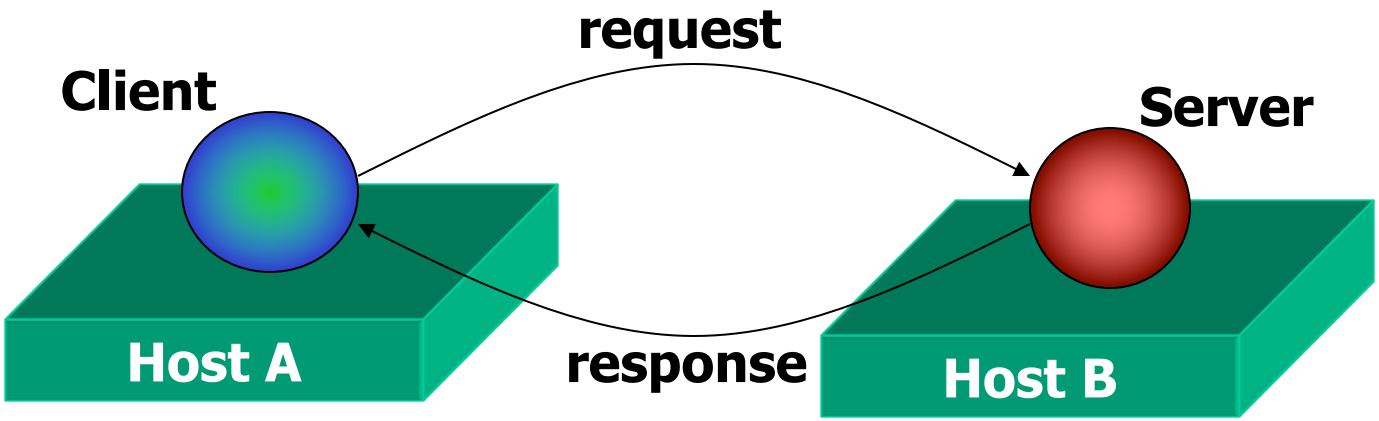
# Well-known layered systems: The OSI and TCP/IP models



# Client/Server

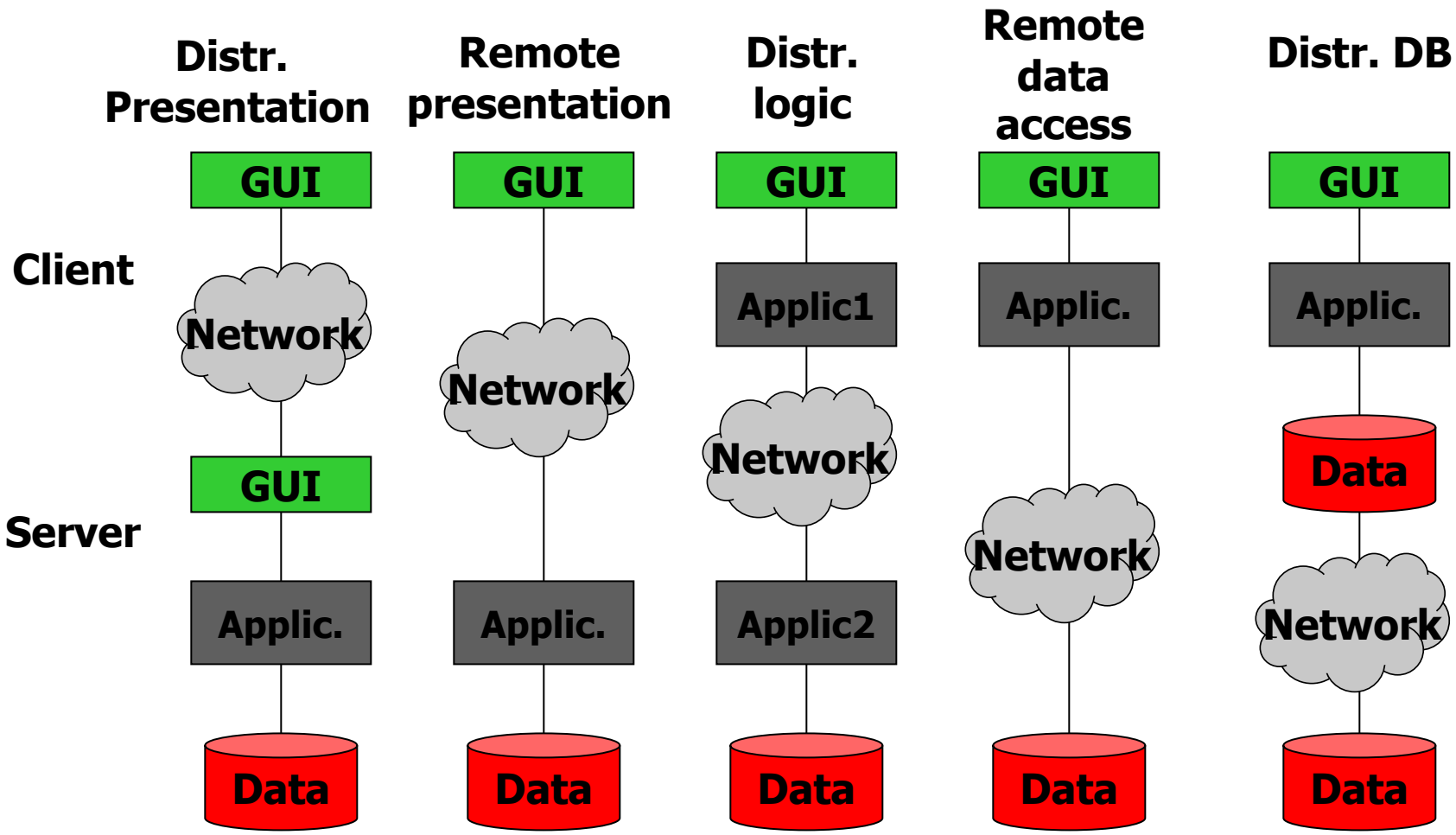

- The best-known and most-used architectural style for distributed applications
- Client & server are different processes with well-defined interface
  - Accessible only through interface
  - Client & server may be defined by a set of hw/sw modules
- Client & server play different roles
  - Server invoked to provide one of a set of services
  - Client uses the provided services and initiates communication
  - Comunication through messages or through remote invocations

# Client /Server: a general scheme



# Client/Server = two-tier architecture

NOTE: also here we have layers!

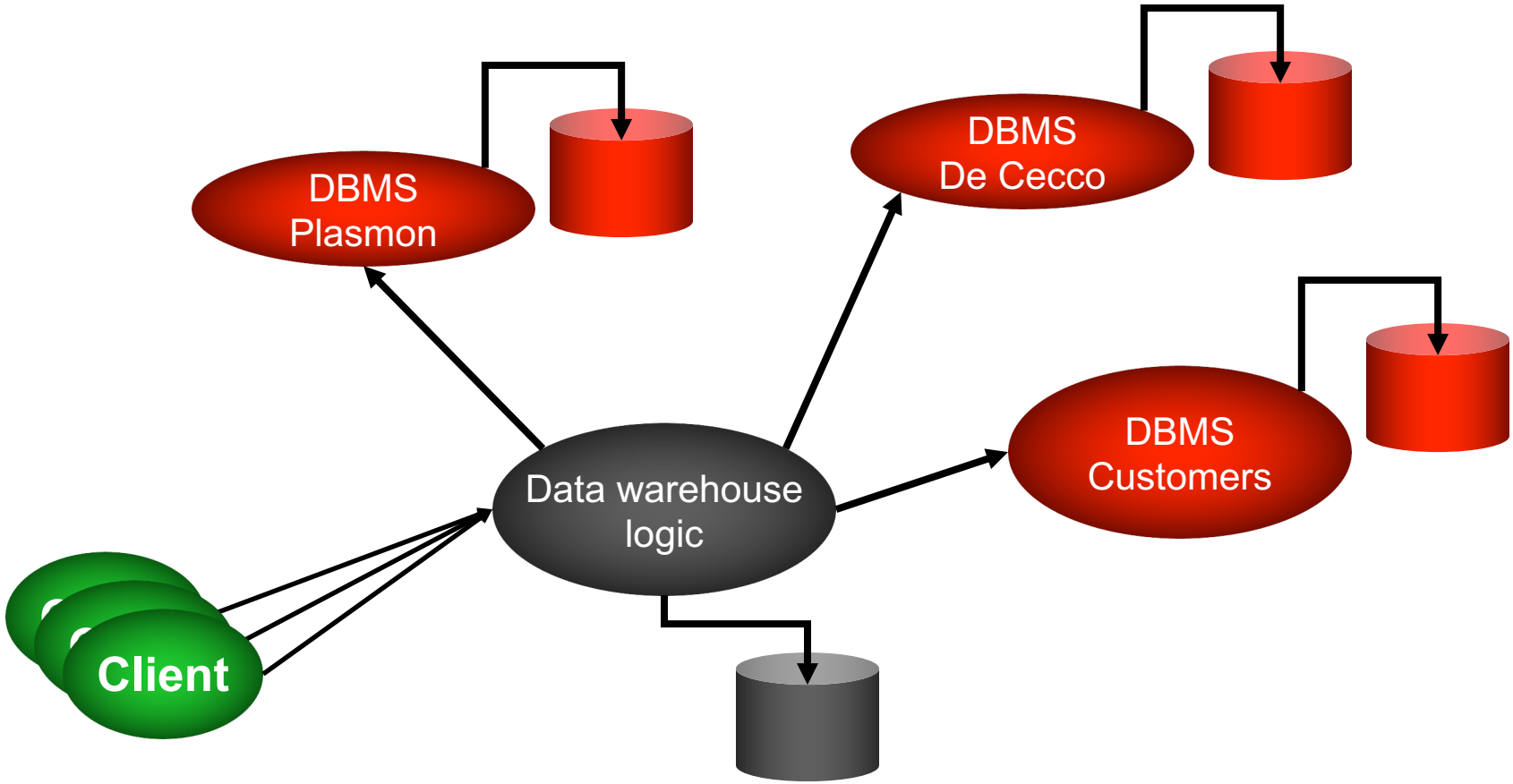


# Three-tier architecture

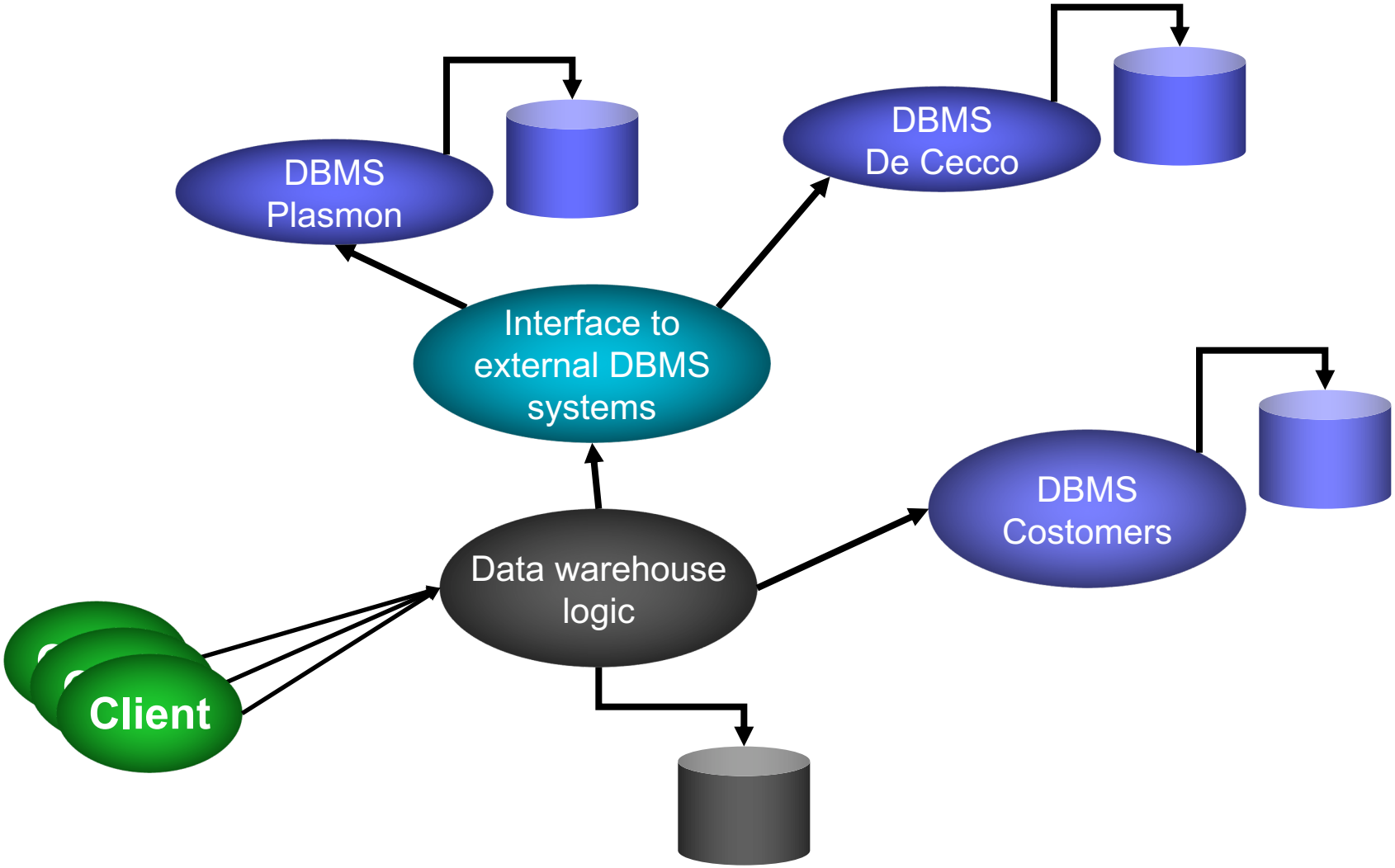

- In the typical scheme, the mid level has all the application logic
- Advantage
  - decoupling of logic and data, logic and presentation

# An example

Data warehouse for a supermarket



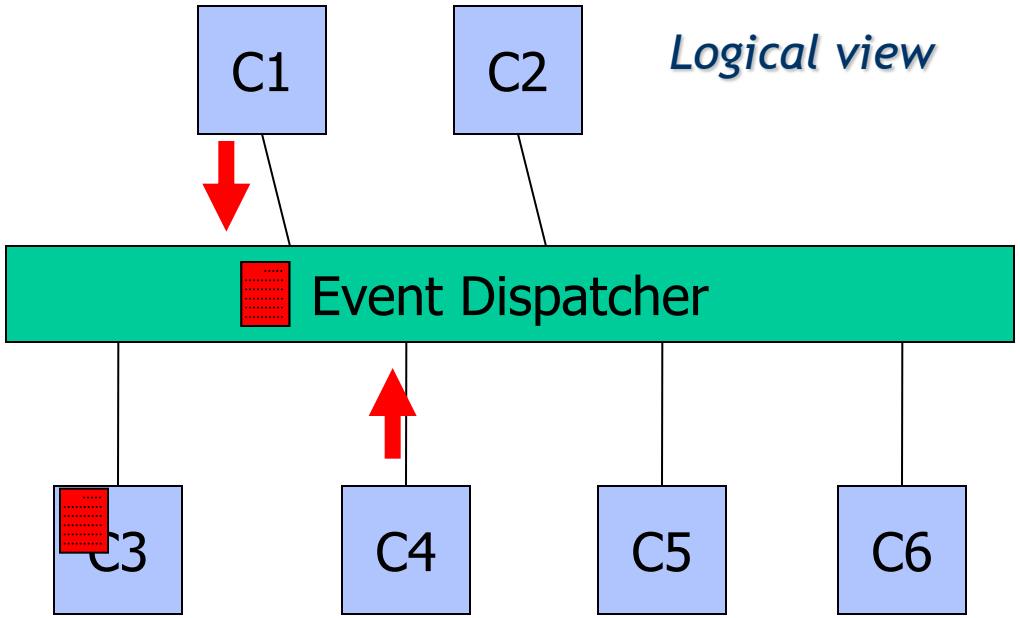
# From 3 to n levels



# Event-based systems

Registration

Event




- Events are broadcast to all registered components
- No explicit naming of target component

# Event-based paradigm


- Often called *publish-subscribe*
- Publish
  - event generation
- Subscribe
  - declaration of interest
- Different kinds of event languages possible

# Event-based systems


- \+ Increasingly used for modern integration strategies
- \+ Easy addition/deletion of components
- – Potential scalability problems
- – Ordering of events

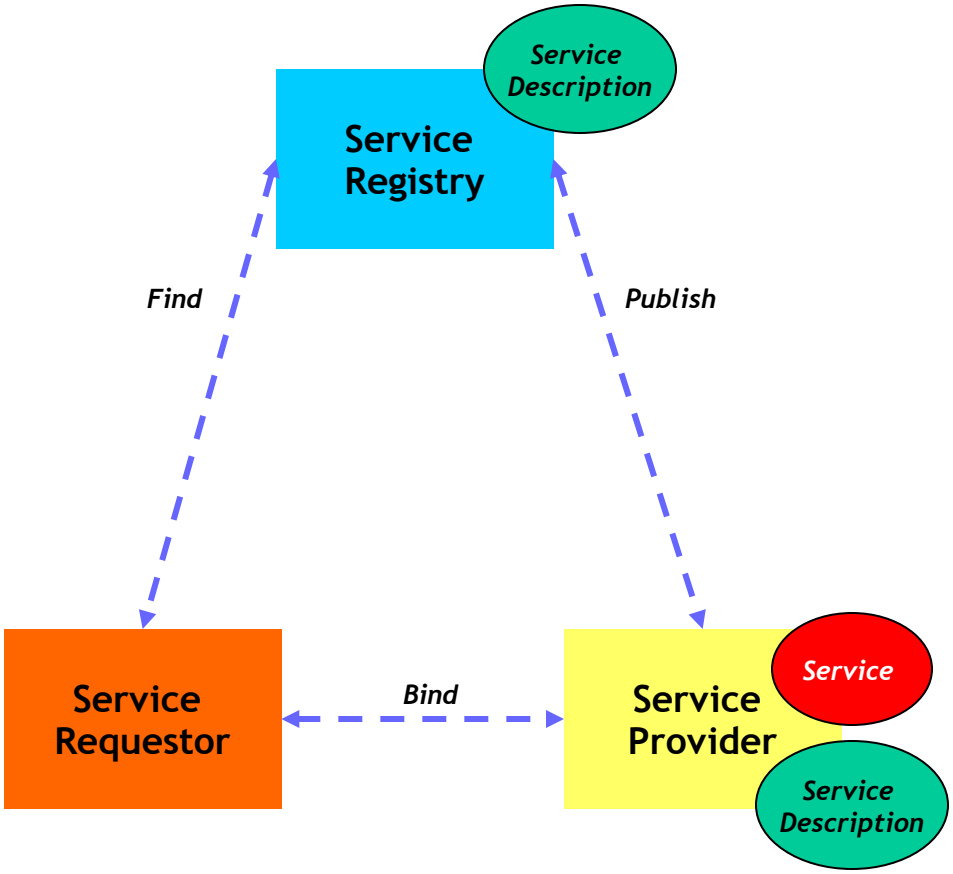
# Common characteristics

- Asynchrony
  - send and forget
- Reactive
  - computation driven by receipt of message
- Location/identity abstraction
  - destination determined by receiver, not sender
- Loose coupling
  - senders/receivers added without reconfiguration
  - one-to-many, many-to-one, many-to-many

# Service-oriented architecture (SOA)

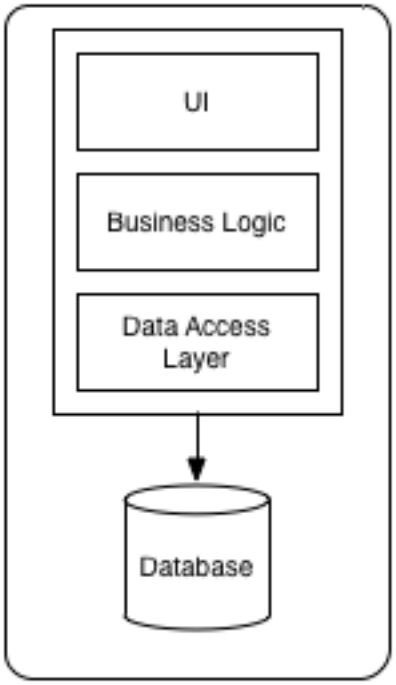

- The SOA defines three agents
  - Service requestors: they request the execution of a service
  - Service providers: they offer services
  - Intermediaries (service brokers, meta-information providers, service registries): they provide information (metadata) about other information/services
- The interaction among these roles results in the execution of the following operations
  - Service description publication
  - Service discovery
  - Service binding

# Roles and operations

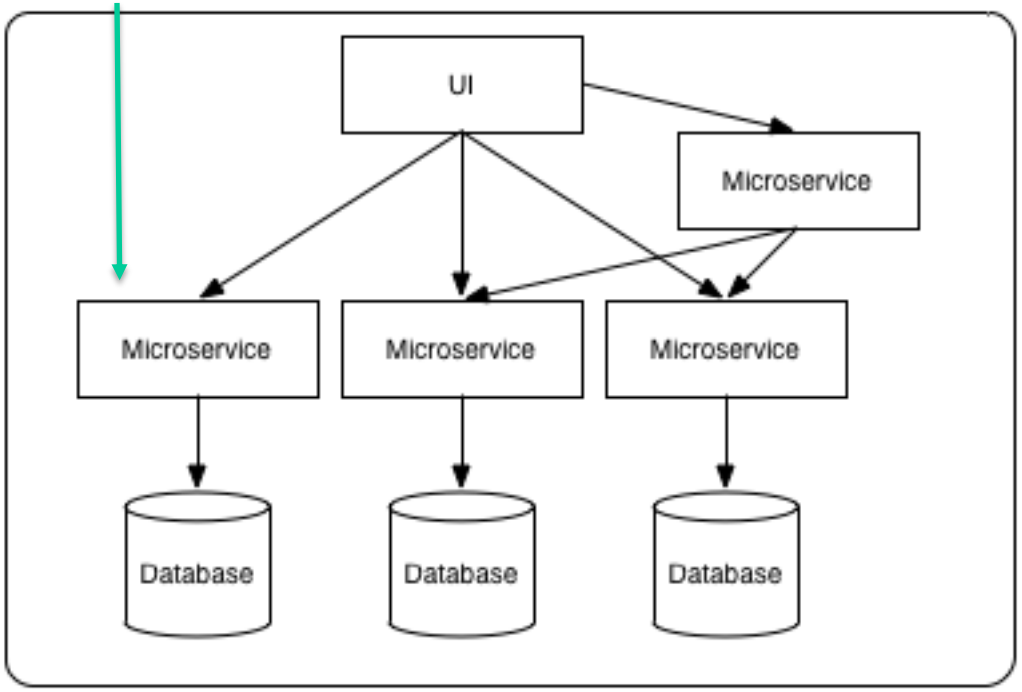


# Microservices

Stateless
Available in multiple instances
No data sharing among microservices
Each microservice may use a specific database




Monolithic Architecture




Microservices Architecture

Politecnico
di Milano


# … More Architectural Styles: Cloud Patterns

http://www.cloudcomputingpatterns.org

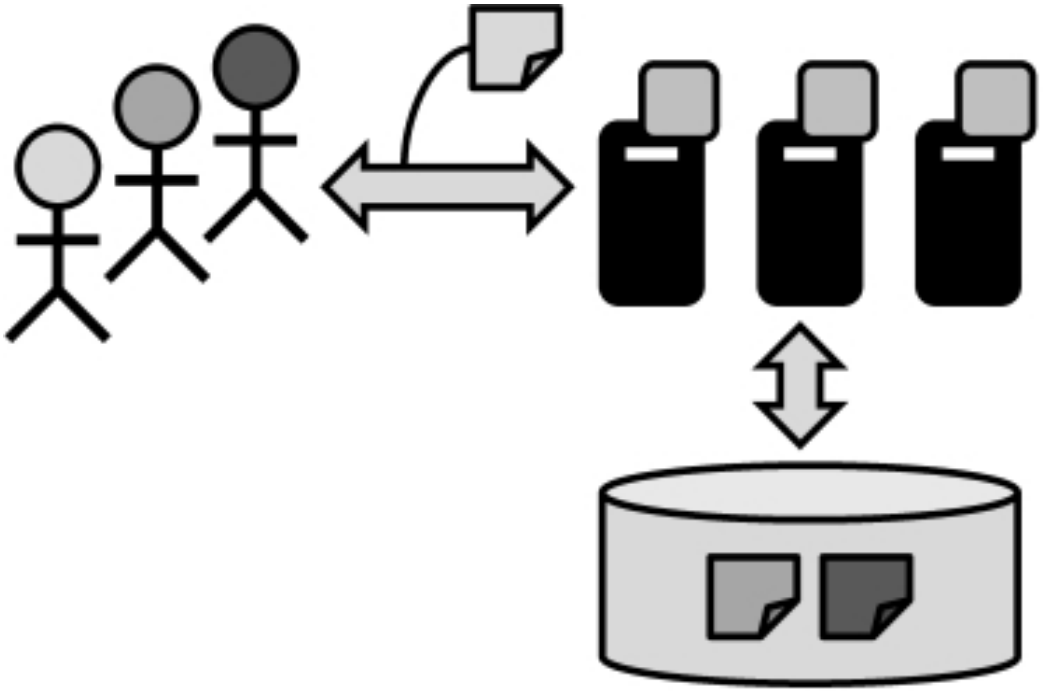
# Classification of patterns


- Basic Architectural Patterns
  - Composite Application
  - Loose Coupling
  - Stateless Component
  - Idempotent Component
- Elasticity Patterns
  - Map Reduce
  - Elastic Component
  - Elastic Load Balancer
  - Elastic Queue
- Availability Patterns
  - Watchdog
  - Update Transition
- Multi-Tenancy Patterns
  - Single Instance Component
  - Single Configurable Instance Component
  - Multiple Instance Component

# Stateless components


- Context
  - A componentized application subsumes components that can fail.
- Challenges
  - Cloud resources can display low availability.
  - Component instances can be added and removed regularly when the demand changes.
- Solution
  - Implement the components in a way that they do not contain any internal state, but completely rely on external persistent storage.

# Stateless components

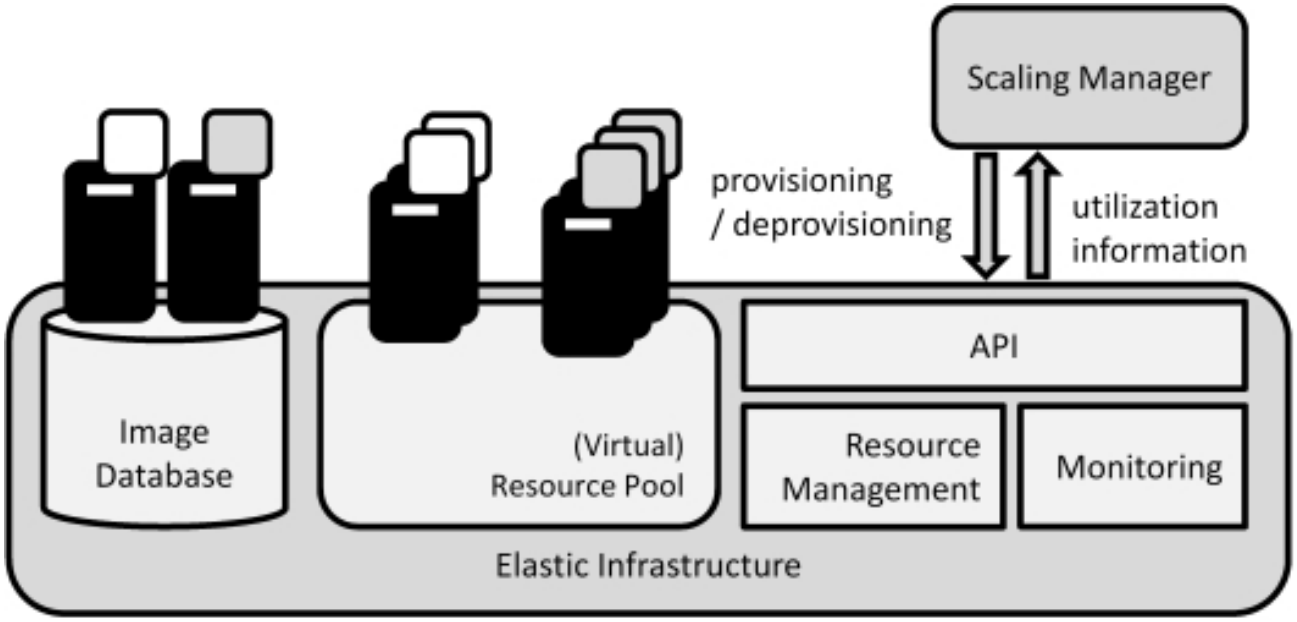


- No data is lost if an instance fails
- Multiple components have a common internal state
- The scalability of the central data store becomes the major challenge
  - Distributed data stores

# Elastic component


- Context
  - A componentized application uses multiple compute nodes provided by an *elastic infrastructure*.
- Challenges
  - To benefit from the elastic infrastructure, the management process to scale-out a componentized application has to be automated.
  - Manual resource scaling would not support pay-per-use nor quickly changing workload
- Solution
  - Monitor the utilization of compute nodes that host application components and automatically adjust their numbers using the provisioning functionality provided by the elastic infrastructure.

# Elastic component

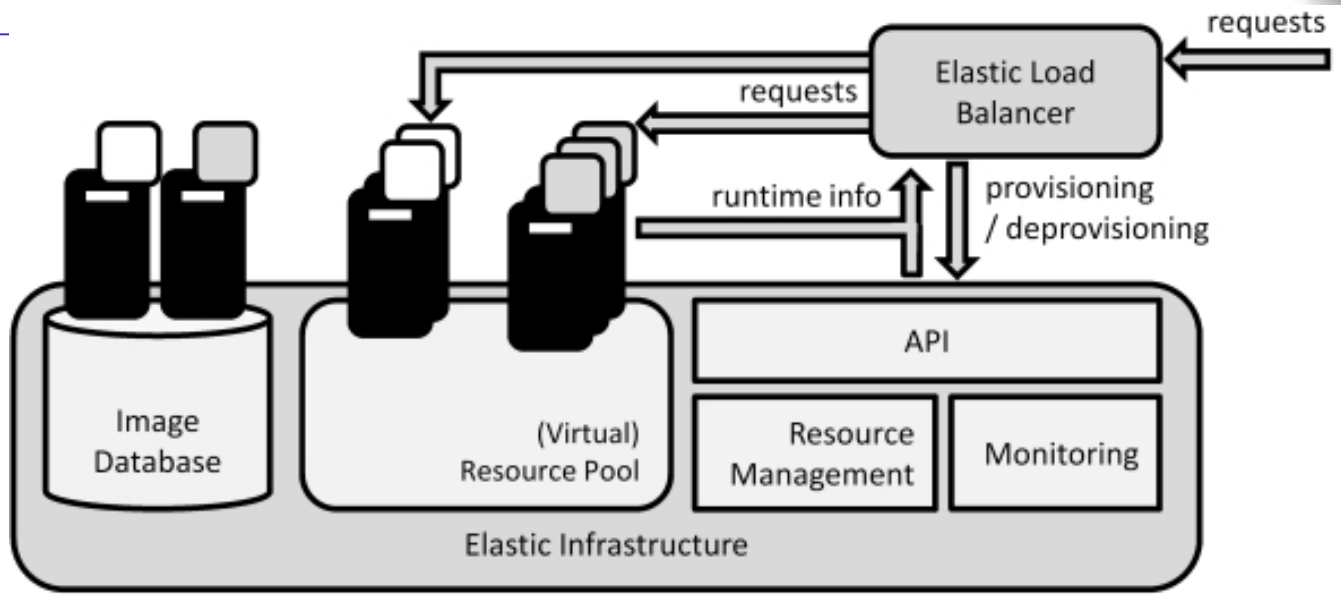



- If the utilization of compute nodes exceeds a specified threshold, additional hosting components are provisioned that contain the same application component.
- If the components are implemented as *stateless components*, the operations for adding and removal of components are significantly simplified.

# Elastic load balancer


- Context
  - A componentized application uses multiple compute nodes provided by an elastic infrastructure.
- Challenges
  - As for elastic component
  - Service requests are a good indicator of workload and therefore shall be used as a basis for scaling decisions
- Solution
  - Use an elastic load balancer that determines the amount of required resources based on numbers of requests and provisions the needed resources accordingly using the elastic infrastructure's API

# Elastic load balancer




- The number of requests that can be handled by a computing node is a crucial design parameter
- It has to be adjusted at run time
- Information about the time needed to provision new compute nodes can also be necessary to deduct effective scaling actions

# Assignments for the next class


- Watch the videos you find here
  - On software design descriptions and principles: https://polimi365-my.sharepoint.com/:v:/g/personal/10143828_polimi_it/EaE5ix1izNxGiLLtzBsr7GABI6M_BqvtBPeH4qtuVMOnhQ?e=hydFes
  - On the design process: https://polimi365-my.sharepoint.com/:v:/g/personal/10143828_polimi_it/EcbU-pgqCblMiw2hIZIUFSABffi1zSZCjXuYrgt1cQfntw?e=ebYZC9
    - Questionnaire available here: https://forms.gle/d2eHhiPT2vySFyQu5
- They will be used as a basis for discussion in class next time