Software Architectures and Quality

Architectural Patterns and Styles

Javier González Huerta

javier.gonzalez.huerta@bth.se

BLEKINGE TEKNISKA HÖGSKOLA · BTH ·

in real life

Objectives

- Introduce the most some of the most common architectural patterns and styles
- Discuss their impact on quality
- Understand how can we use them when designing or transforming an architecture

What patterns (and styles) are

- A pattern is a recurring common solution (in terms of design elements) to a recurring problem in the real world or in software
    - ”Twice is a coincidence, three times is a pattern”
- Architectural Patterns (and Styles)
    - Collection of elements, relations and constraints
    - Can be seen as packaged strategies to solve a common, recurring problem
    - Exhibit known quality attributes, that can help us fixing some quality issues

Architectural Patterns and Styles

Quick Tour Through Some Qualities

Quick tour through some qualities

Runtime

- Availability
- Interoperability
- Security
- Performance
- Fault Tolerance
- Scalability
- Safety

Design Time

- Maintainability
- Modifiability
- Testability

Quick tour through some qualities: Runtime

Availability

- Whether the software is there and ready to carry out the tasks

Standard ISO/IEC25000 Definition: *“The degree to which a software component is operational and available when required for use.”*

- Ability of the system to mask, handle or repair faults so that the cumulative outage time does not exceed a value in a period of time
- Usually is measured as:

$$Availability = \frac{MTBF}{MTBF+MTTR}$$

Availability: The nines rule

Availability
99.0%
99.9%
99.99%
99.999%
99.9999%

Availability: The nines rule

Availability	Downtime in 90 days
99.0%	21 h 36 min
99.9%	2 h 10 min
99.99%	12 min 58 s
99.999%	1 min 18 s
99.9999%	8s

Availability: The nines rule

Availability	Downtime in 90 days	Downtime in a year
99.0%	21 h 36 min	3 d 15.6 h
99.9%	2 h 10 min	8 h 0 min 46 s
99.99%	12 min 58 s	52 min 34 s
99.999%	1 min 18 s	5 min 15 s
99.9999%	8s	32 s

Quick tour through some qualities: Runtime

Interoperability

- Whether a system can interchange meaningful information and cooperate with other systems

Standard ISO/IEC25000 Definition: *“The degree to which the software product can be cooperatively operable with one or more other software products.”*

Quick tour through some qualities: Runtime

Security

- Whether the software protect information and data to unauthorized access

Standard ISO/IEC25000 Definition: *“The protection of system items from accidental or malicious access, use, modification, destruction, or disclosure.”*

- Confidentiality: Data protected to unauthorized access
- Integrity: Data not subject to unauthorized manipulation
- Availability*: System ready for its legitimate user

*Availability is seen also as a component of security

Quick tour through some qualities: Runtime

Performance

- It is the ability of the software to meet the timing requirements

Standard ISO/IEC25000 Definition: *"The degree to which the software product provides appropriate response and processing times and throughput rates when performing its function, under stated conditions."*

- Usually simplified to *Latency Time*
  - Time elapsed between firing an event and receiving a response
- Sometimes we also talk about resource utilization (memory consumption & CPU utilization)

Quick tour through some qualities: Runtime

Fault Tolerance / Reliability

- The degree to which the system can handle failures

Standard ISO/IEC25000 Definition: “The degree to which the software product can maintain a specified level of performance in cases of software faults or of infringement of its specified interface.”

- It is somehow a component of Availability, but in this case we are more interested on whether the system is free of failure states
- Fault tree analysis is one way to analyze the sources of failure and their probabilities

Quick tour through some qualities: Runtime

Scalability

- How the system effectively manages [added] resources

Standard ISO/IEC25000 Definition*: "The degree to which the software product can be adapted for different specified environments without applying actions or means other than those provided for this purpose for the software considered.

*This version of ISO names scalability as adaptability

- Measures what happens for example when the number of requests grows
- If we are measuring how manages added resources, we will have to consider how impacts to other qualities

Quick tour through some qualities: Safety

Safety

- Safety Is about the ability of the software to not harm or kill users…

Standard ISO/IEC25000 Definition: “degree to which a product or system mitigates the potential risk to people in the intended contexts of use ”

- Usually measured by demonstrating the presence of fail-safe modes and their coverage

Quick tour through some qualities: Design Time

Modifiability/Maintainability

- The cost / effort required change the software system

Standard ISO/IEC25000 Definition (Maintainability): “The degree to which the software product can be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications. ”

- Usually measured as the cost in money or effort in time to analyze, plan, design, develop and test a given change on the system

Quick tour through some qualities: Design Time

Testability

- How easy is finding software faults in the system

Standard ISO/IEC25000 Definition (Maintainability): "The degree to which the software product enables modified software to be validated."

- Usually measured as the effort in time to test the software

Architectural Patterns and Styles

Architectural Styles

Architectural Styles

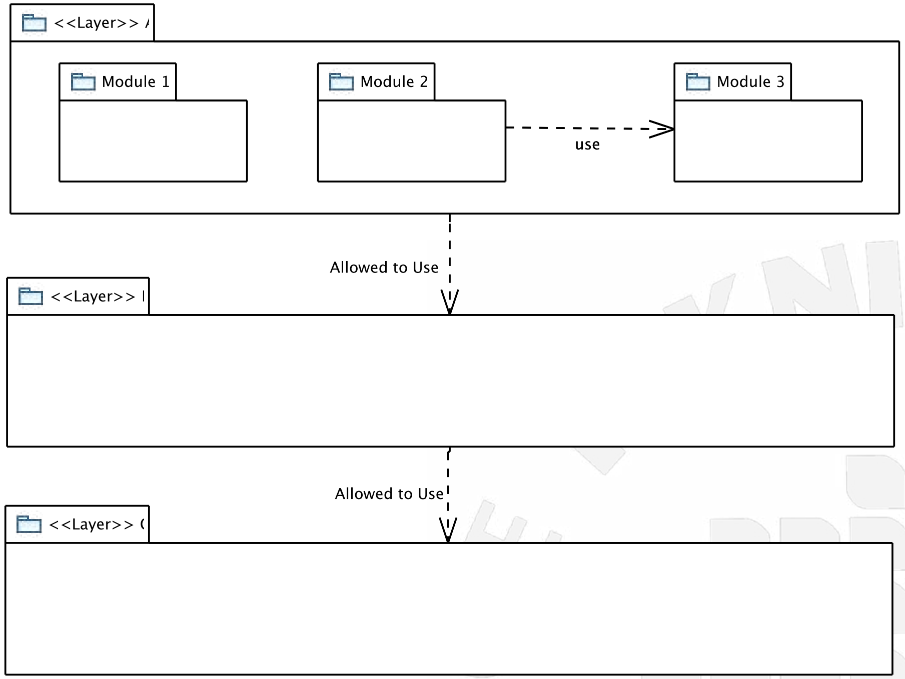
- Layered Style
- Model View Controller (MVC)
- Pipes and Filters
- Blackboard
- Microkernel
- Broker

Layered Style

- **Context:**
  - Complex systems that require decomposition
- **Problem:**
  - The software needs to be segmented, to be able to be developed and evolved separately, with clear separation of concerns.

Layered Style

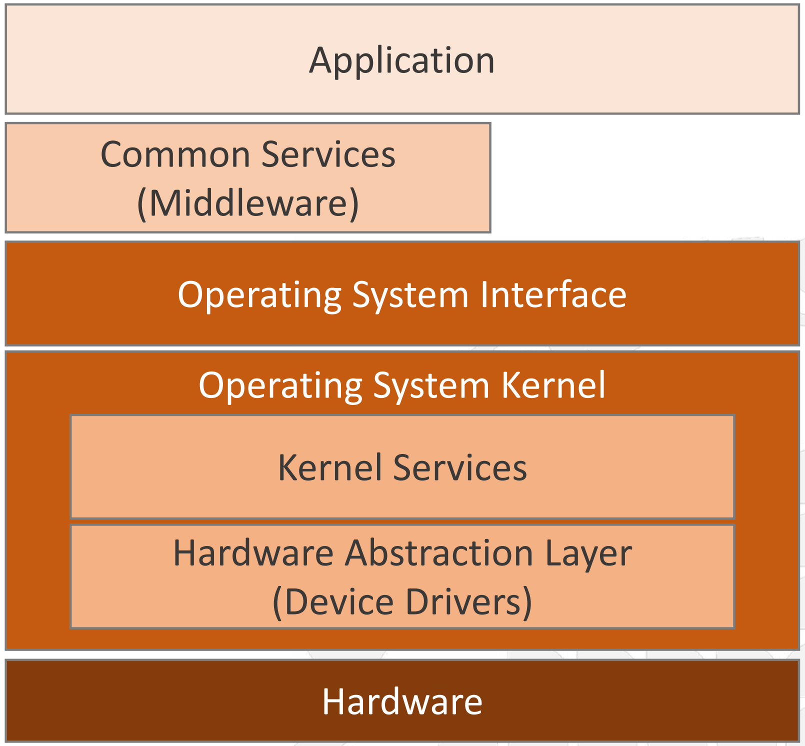
- **Solution:**
  - Layering helps structuring applications that can be decomposed into groups of subtasks that operate at different levels of abstraction
  - There is a well established separation of responsibilities
  - Each level and each module inside layers can be modified separately



Layered Style

- **Solution:**
  - Layering helps structuring applications that can be decomposed into groups of subtasks that operate at different levels of abstraction
  - There is a well established separation of responsibilities
  - Each level and each module inside layers can be modified separately

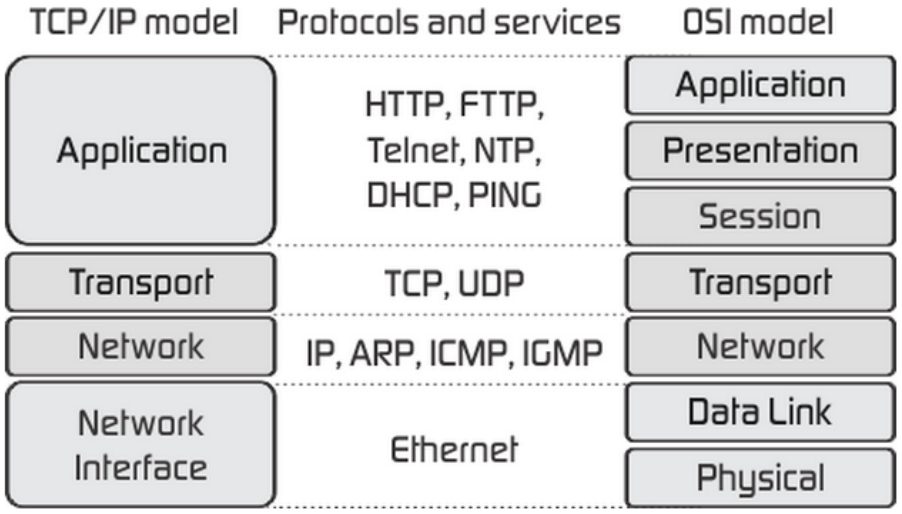
Operating System Architecture


Application
Common Services (Middleware)
Operating System Interface
Operating System Kernel
Kernel Services
Hardware Abstraction Layer (Device Drivers)
Hardware

Layered Style

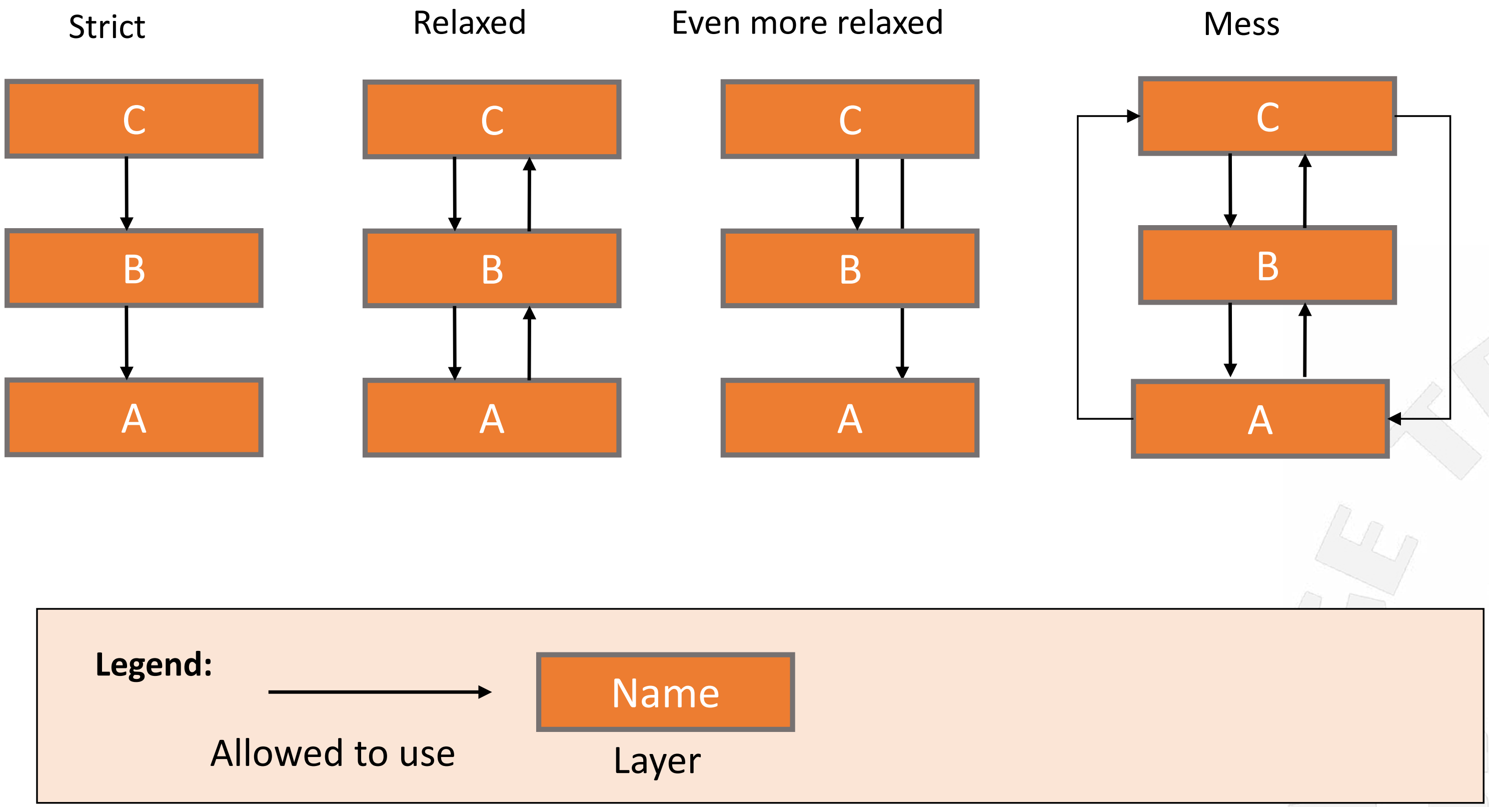
- **Solution:**
  - Layering helps structuring applications that can be decomposed into groups of subtasks that operate at different levels of abstraction
  - There is a well established separation of responsibilities
  - Each level and each module inside layers can be modified separately

ISO/OSI vs TCP Layers



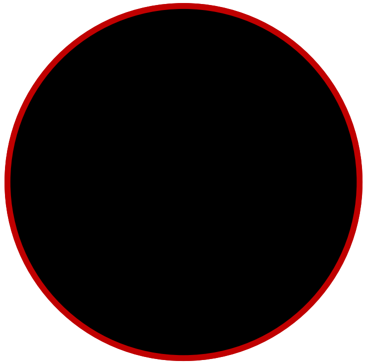
TCP/IP model	Protocols and services	OSI model
Application	HTTP, FTTP, Telnet, NTP, DHCP, PING	Application Presentation Session
Transport	TCP, UDP	Transport
Network	IP, ARP, ICMP, IGMP	Network
Network Interface	Ethernet	Data Link Physical

Layered Style



Reflection

- Discuss during 2 minutes in groups of 2-3 persons: Pros and cons of the Layered Style?



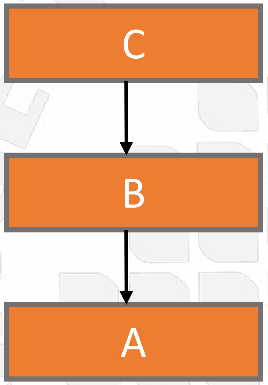
Layered Style: Impact on Qualities

Pros:

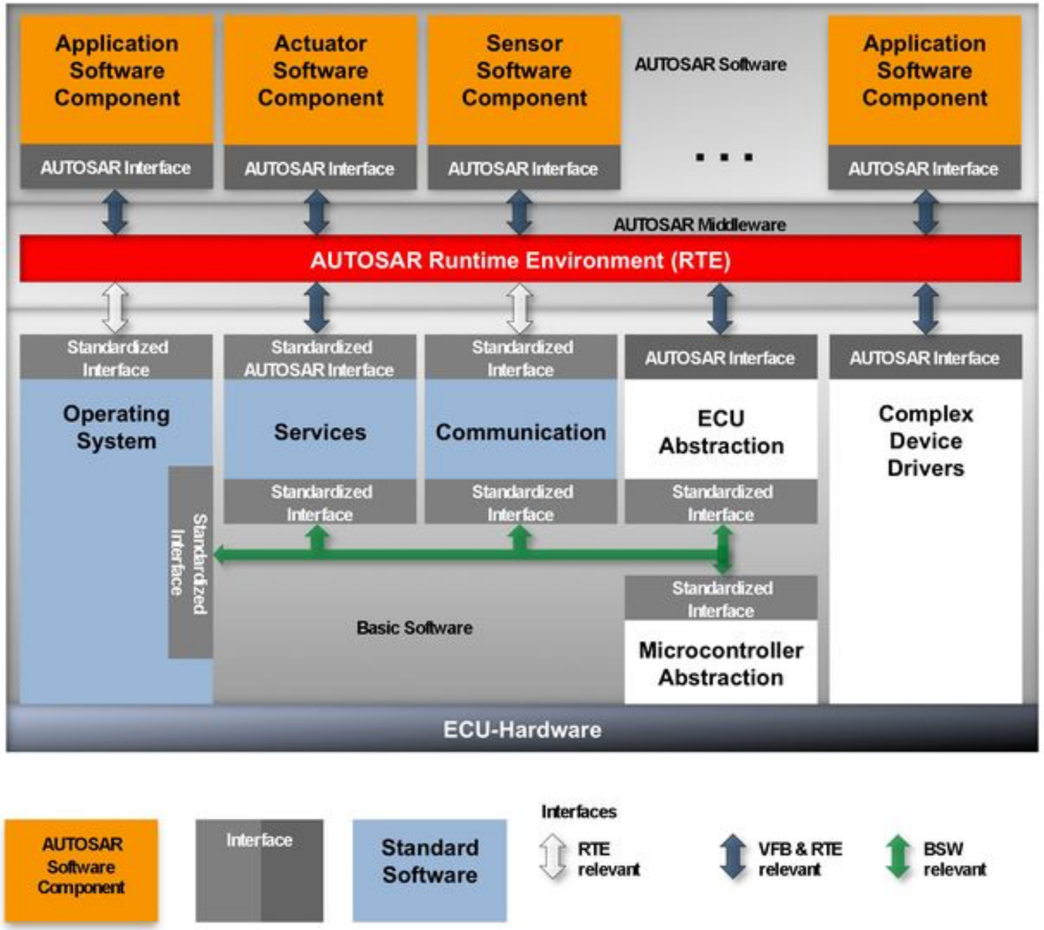
- **Modifiability /Maintainability**
  - Reduces coupling between layers and usually increases cohesion within the layer
  - If the functionality is well isolated in just one layer, improves maintainability

Cons:

- **Performance**
  - The layers introduce overhead adding a performance penalty
- **Fault Tolerance**
  - A failure in lower layers can lead to chain reactions



Layered Architecture - A Bigger Example: Autosar



Source: www.autosar.org

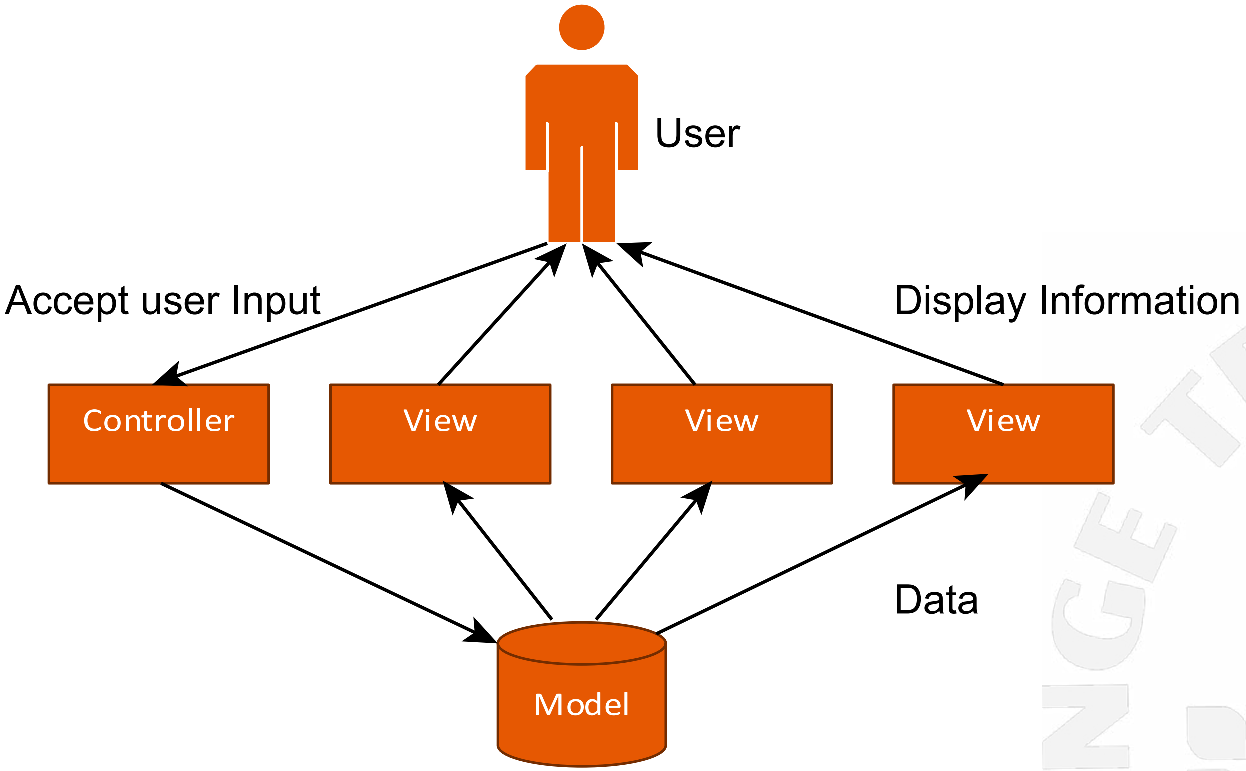
Model View Controller

- **Context:**
  - Interactive applications with a flexible Graphical User Interface (GUI)
- **Problem:**
  - How to decouple the GUI from the functionality & data?
  - How to have different GUI for the same functionality & data?
  - How to create, maintain and coordinate GUIs when the functionality and data change?

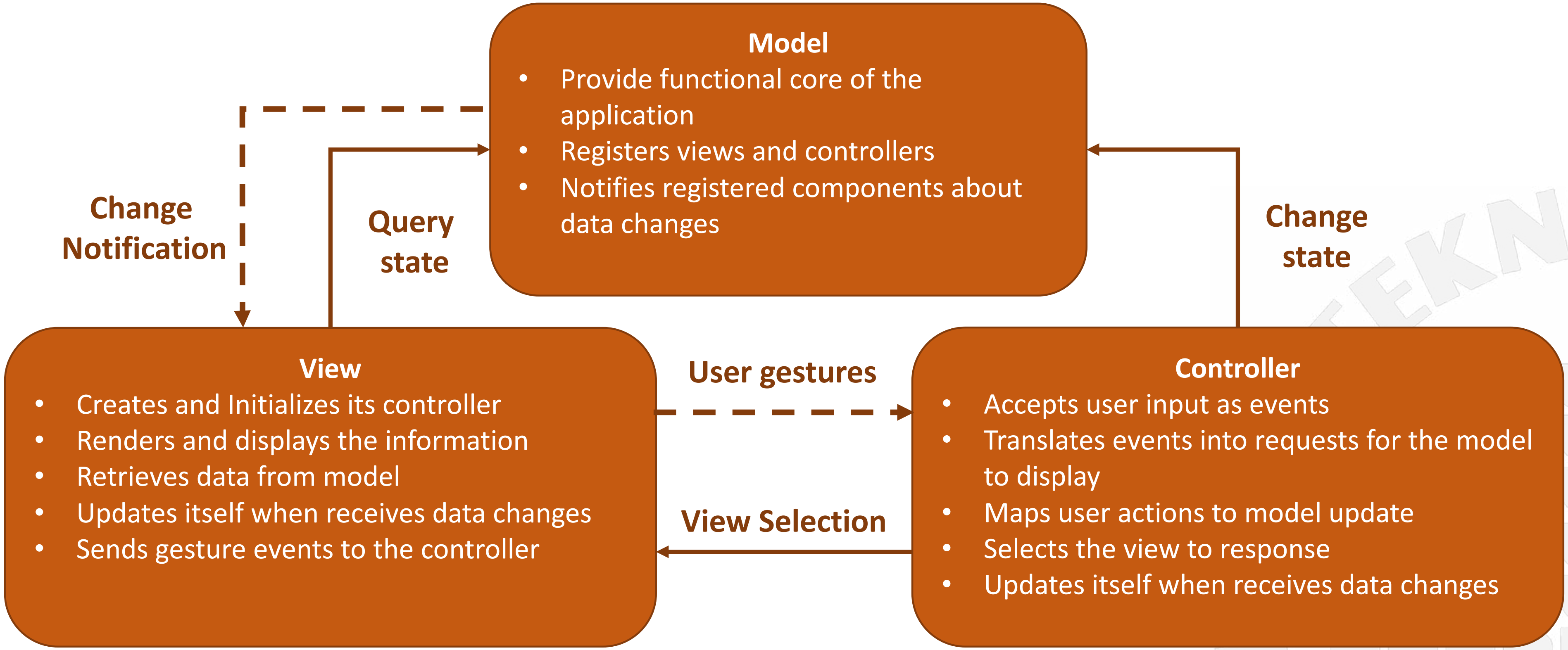
Model View Controller

- **Solution:**
  - MVC divides an interactive application into three components
    - The model contains the core functionality and data
    - The view displays the information to the user
    - The controllers handle the user input
  - Views and controller together comprise the GUI (Behavior + Presentation)
  - Changes are automatically propagated to assure consistency between the GUI and the model

Model View Controller



Model View Controller



Model

- Provide functional core of the application
- Registers views and controllers
- Notifies registered components about data changes

View

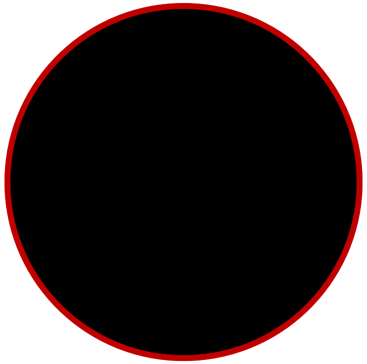
- Creates and Initializes its controller
- Renders and displays the information
- Retrieves data from model
- Updates itself when receives data changes
- Sends gesture events to the controller

Controller

- Accepts user input as events
- Translates events into requests for the model to display
- Maps user actions to model update
- Selects the view to response
- Updates itself when receives data changes

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: Pros and cons of the MVC?**



Model View Controller: Impact on Qualities

Pros:

- **Testability and Modifiability:**
  - Since the three components re loosely coupled it is easy to modify and test the system.
  - Changes in one component do not affect the others.
  - Easy to add new views
  - Easy to add new controllers
  - Easy to add new look-and-feel

Cons:

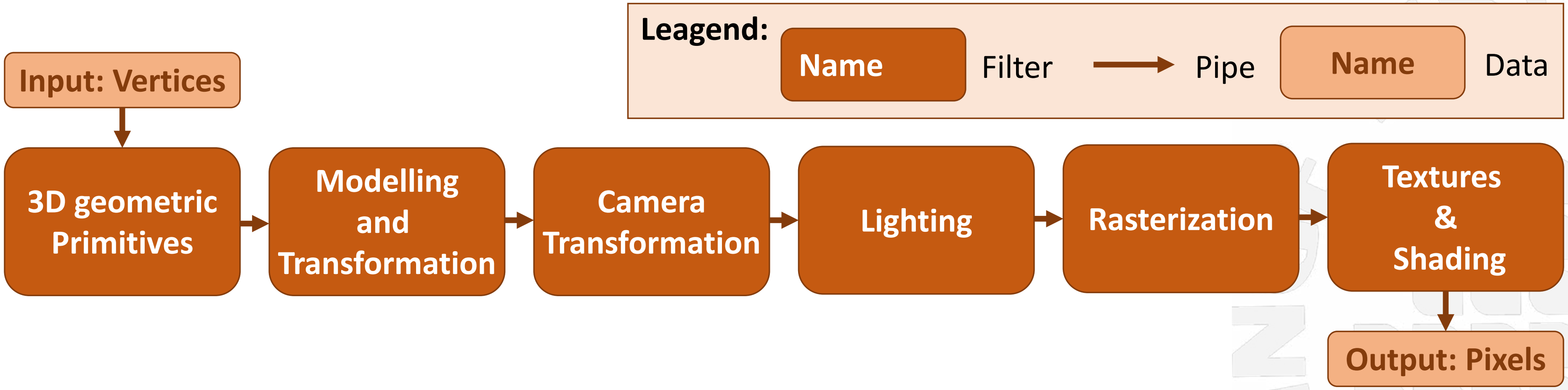
- **Performance:**
  - Might have low performance (many updates), unchanged data
- Increased complexity
- Difficulties in re-using views separate from controllers
- Platform-dependent code in view and controller

Pipes and Filters

- **Context:** Process streams of discrete data items from input to output with several intermediate steps
- **Problem:** How to divide a system into reusable, loosely coupled components with simple interaction mechanisms?

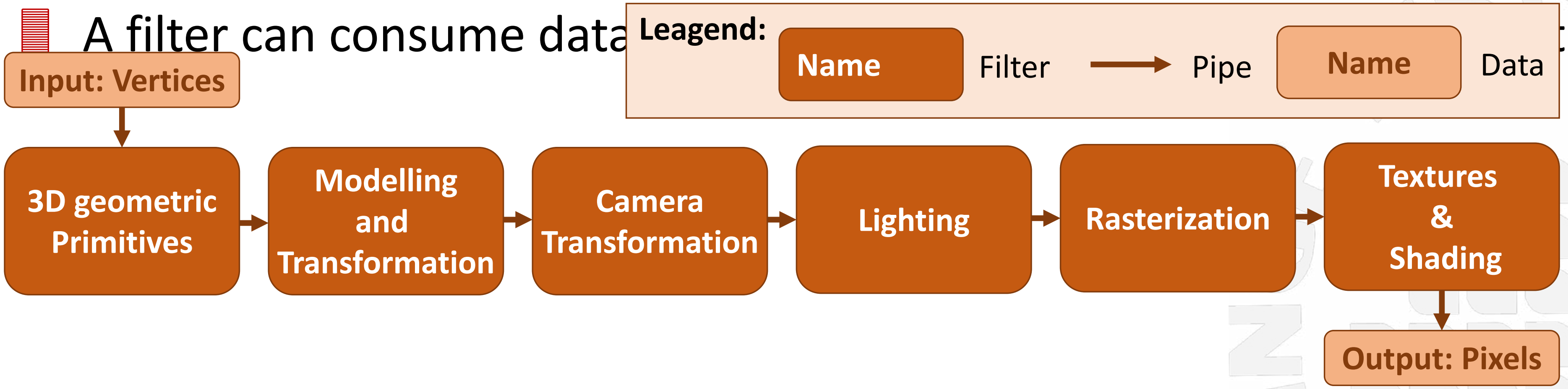
Pipes and Filters

- Provides a structure of systems that process, through a transformation chain, a stream of data
- Each processing step is encapsulated in a filter component
- Data arrives to a filter input who transforms the data
- Data is passed through via its output port through a pipe to next filter



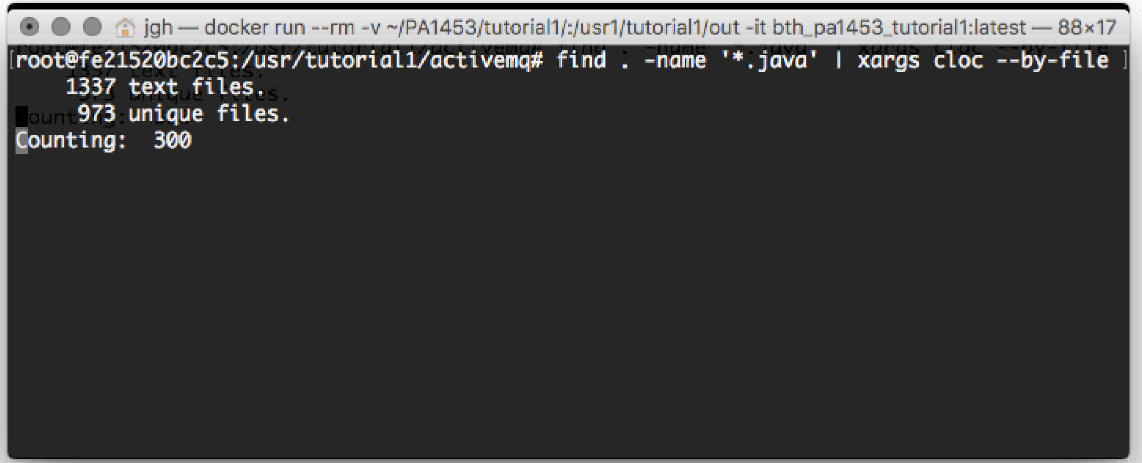
Pipes and Filters

- Data arrives to a filter input
- The filter transforms the data
- Data is passed through via its output port(s) through a pipe to next filter
- A filter can consume data ts



Pipes and Filters

- Provides a structure of systems that process, through a transformation chain, a stream of data
- Each processing step is encapsulated in a filter component
- Data is passed through pies between adjacent filters



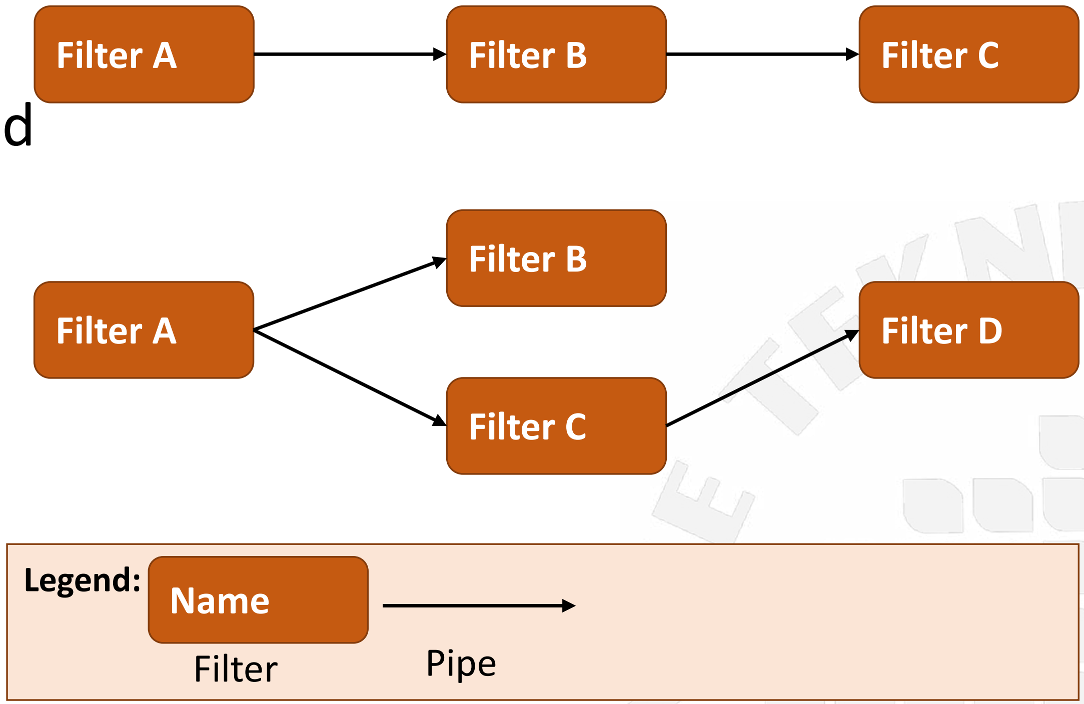
jgh — docker run --rm -v ~/PA1453/tutorial1/:/usr1/tutorial1/out -it bth_pa1453_tutorial1:latest — 88×17

```
root@fe21520bc2c5:/usr/tutorial1/activemq# find . -name '*.java' | xargs cloc --by-file
    1337 text files.
     973 unique files.
Counting:  300
```

Tip: I recommend you to come back to this slide if you are going to measure modifiability for your architecture

Pipes and Filters

- Provides reusability since filters can be recombined to obtain new systems
- If the components are defined as independent, can be run in parallel



Pipes and Filters: Impact on Qualities

Pros:

- **Performance**
  - Improves performance since some filters can be run in parallel
  - … only if filters keep an efficient size
  - Having a lot of independent filters can introduce substantial overhead
- **Maintainability**
  - Improves maintainability, filters are modular
  - … only if the changes are local to one filter
  - It is easy to add and replace filters
  - Allows for runtime reconfiguration
- **Security**
  - Is easy to introduce security filters

Cons:

- **Fault Tolerance**
  - Can hinder fault tolerance, if a filter fails, everything fails

Pipes and Filters: Impact on Qualities

Pros:

- **Performance**
  - Improves performance since some filters can be run in parallel
  - … only if filters keep an efficient size
  - Having a lot of independent filters can introduce substantial overhead
- **Maintainability**
  - Improves maintainability, filters are modular
  - … only if the changes are local to one filter
  - It is easy to add and replace filters
  - Allows for runtime reconfiguration
- **Security**
  - Is easy to introduce security filters

Cons:

- **Fault Tolerance**
  - Can hinder fault tolerance, if a filter fails, everything fails

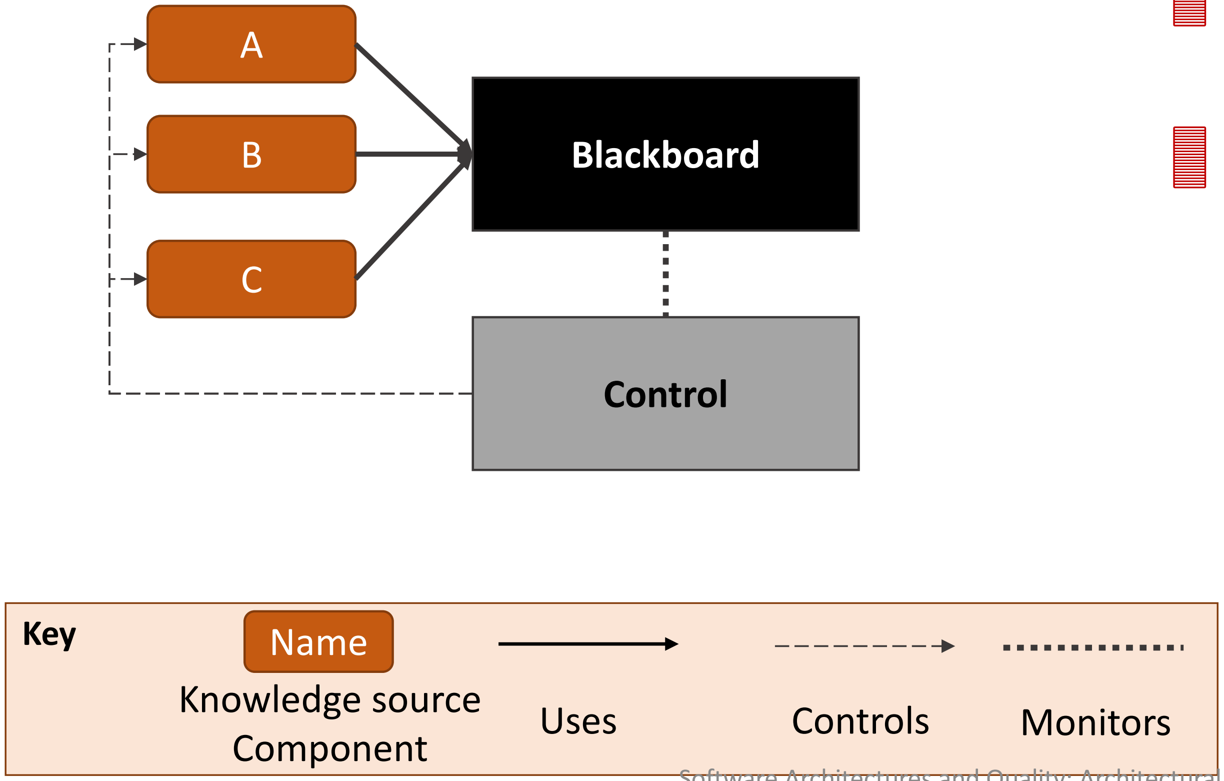
Pipes and Filters: Impact on Qualities

- Pipes and filters might be not recommended for highly interactive systems
- Pipes and filers is not the best option if there are long running computations

Blackboard

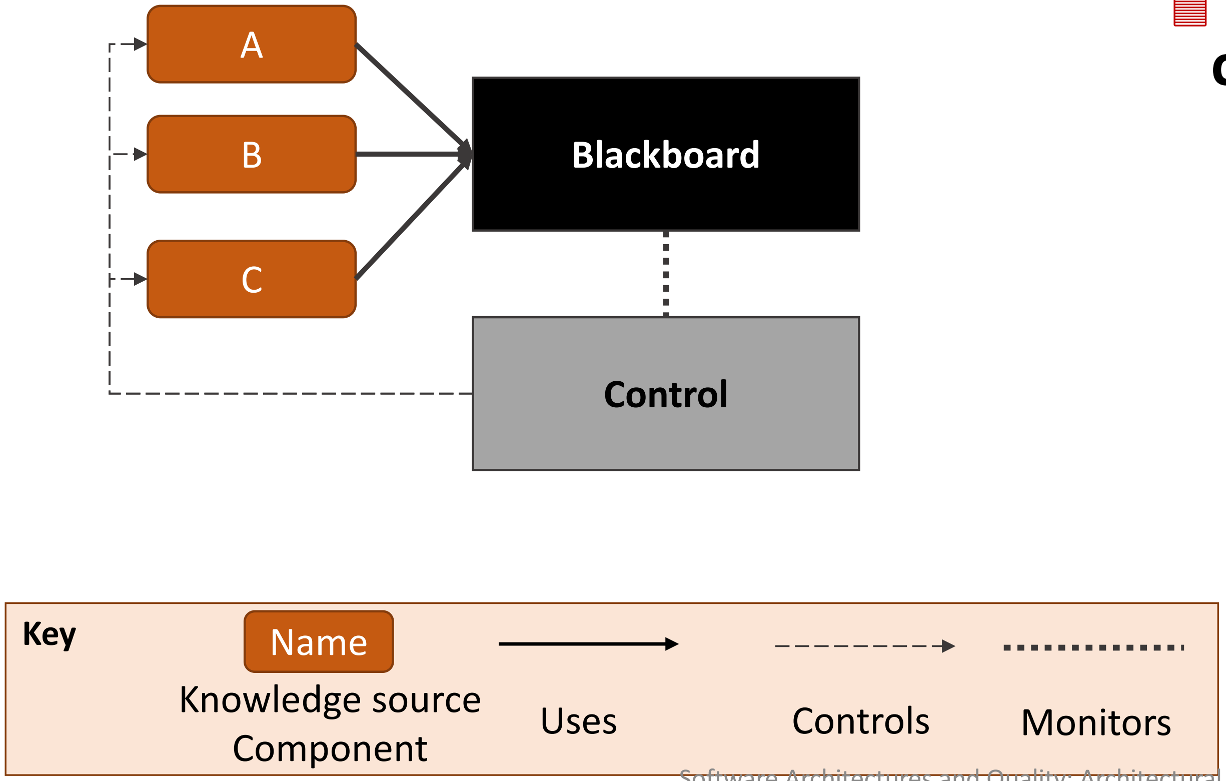
- **Context:**
  - A problem in which no close approach to a solution is well known or feasible
- **Problem:**
  - How to build a feasible non-deterministic solution for a problem of transforming raw data into high level structures, such as speech or image recognition?
  - The problem can be decomposed into several subtasks
  - Each subtask can generate several alternative solutions
  - No determined order to execute tasks
- This pattern has evolved in the last years to be applied into modern games

Blackboard



- **Blackboard:**
  - Repository of central Data
- **Control:**
  - Monitors the blackboard
  - Schedules Knowledge source activations

Blackboard



- **Knowledge sources (KS) components:**
  - Experts that solve part of the problem
  - Evaluates its own applicability
  - Computes a result
  - Updates the blackboard with hypothesis and conclusions

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: Pros and cons of the Blackboard?**

Blackboard: Impact on Qualities

Pros:

- **Varying performance**
  - Depends highly on implementation.
  - Control lowers performance even more.
- **Good maintainability**
  - Easy to add or remove knowledge sources.
  - Control makes it a bit more difficult.
- **Fault tolerance**
  - A failure in one KS does not affect the others
  - Might not even affect the result

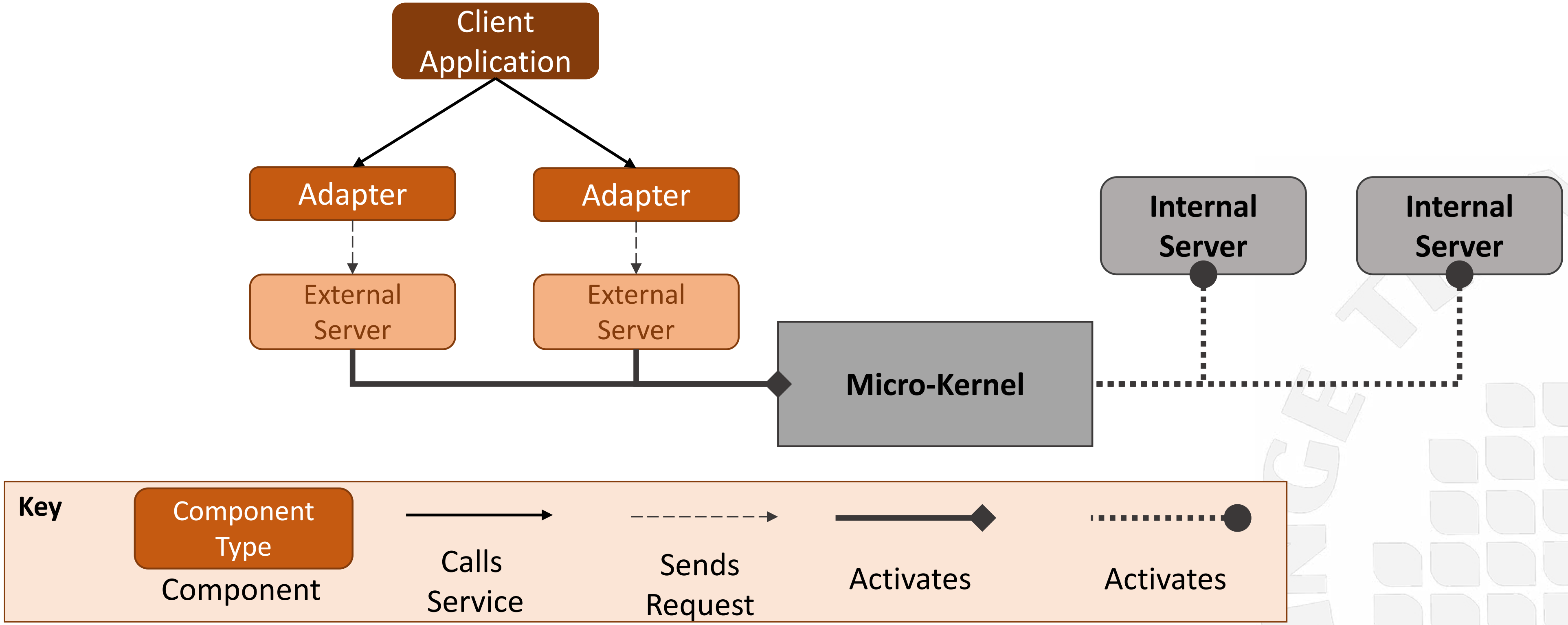
Cons:

- **Low Testability**
  - Is extremely difficult to test since there is no deterministic sequence
- **Low reliability**
  - Non-deterministic execution.
- **Low security**
  - All components can access all data.

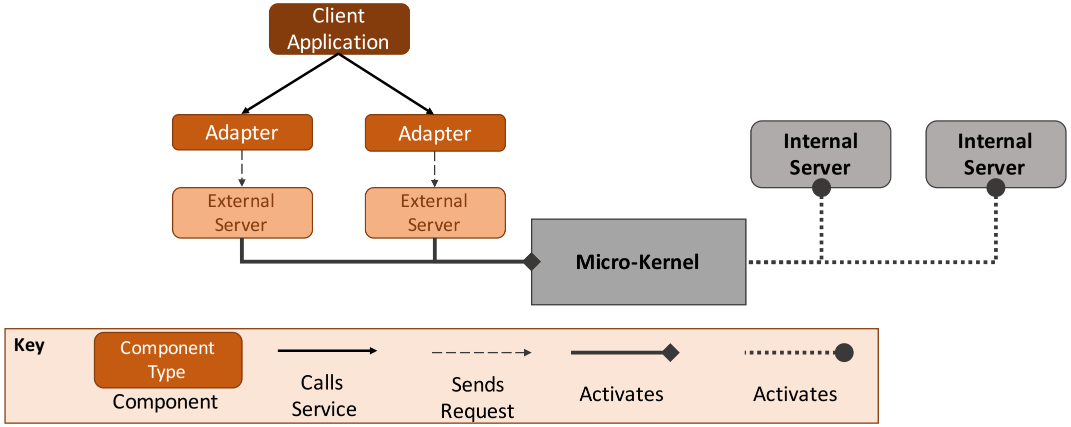
Microkernel

- **Context:**
  - Software systems that must be able to adapt to changing, evolving requirements
- **Problem:**
  - How to develop systems that cope with continuous hardware evolution?
  - The application platform should be portable, extensible and adaptable to new emerging technologies

Microkernel

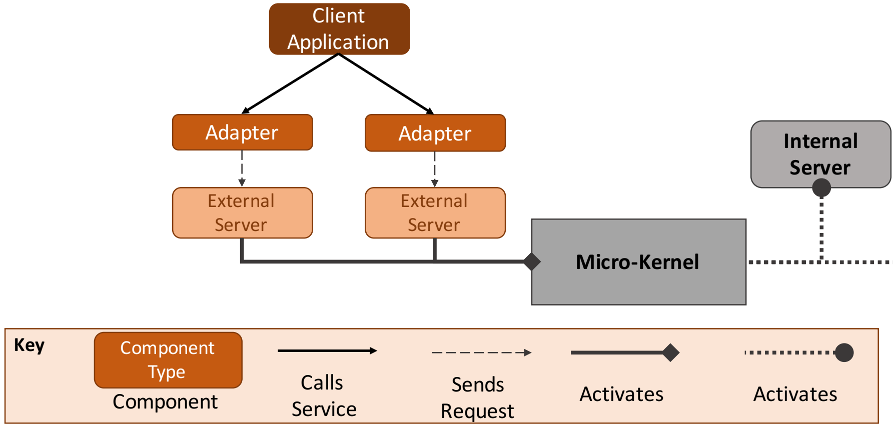


Microkernel



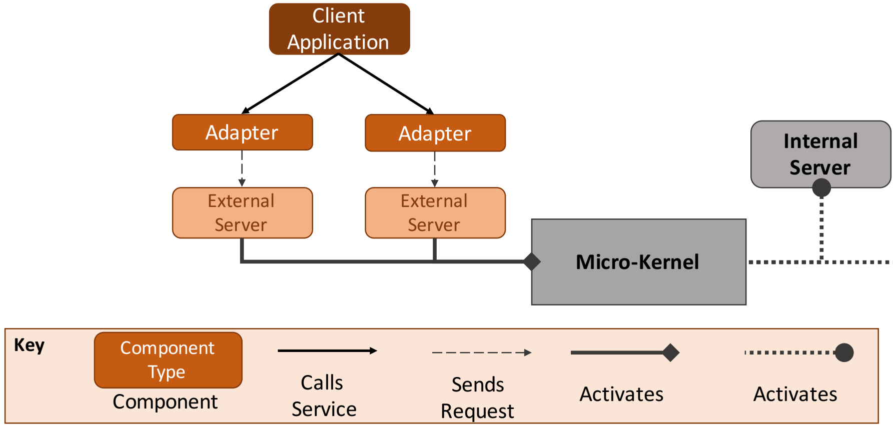
- **Microkernel:**
  - Provide core mechanisms
  - Provide communication mechanisms
  - Encapsulates system dependencies
  - Manage and Control Resources

Microkernel



- **Internal Server**
  - Implement additional services
  - Encapsulate the system specifics
- **External Server**
  - Provide programming interfaces for clients

Microkernel



- **Adapters**
  - Hide internal dependencies from the client
  - Invoke external server methods
- **Client**
  - User application

Microkernel: Impact on Qualities

Pros:

- **Portability**
  - Separate hardware-dependent functionality in separate subsystems
  - Just need to port the a small part of the microkernel
- **Flexible and Extensible**
  - Only adding new servers
- **Maintainability**
  - Separate policies from mechanisms
- **Security and Reliability**
  - The Microkernel runs in a protected address space
  - Failure in servers do not affect the whole

Cons:

- **Performance:**
  - Overhead due to communication
  - Less performance as compared to monolithic
- Complex systems
  - Microkernel is complex to design and develop

Broker

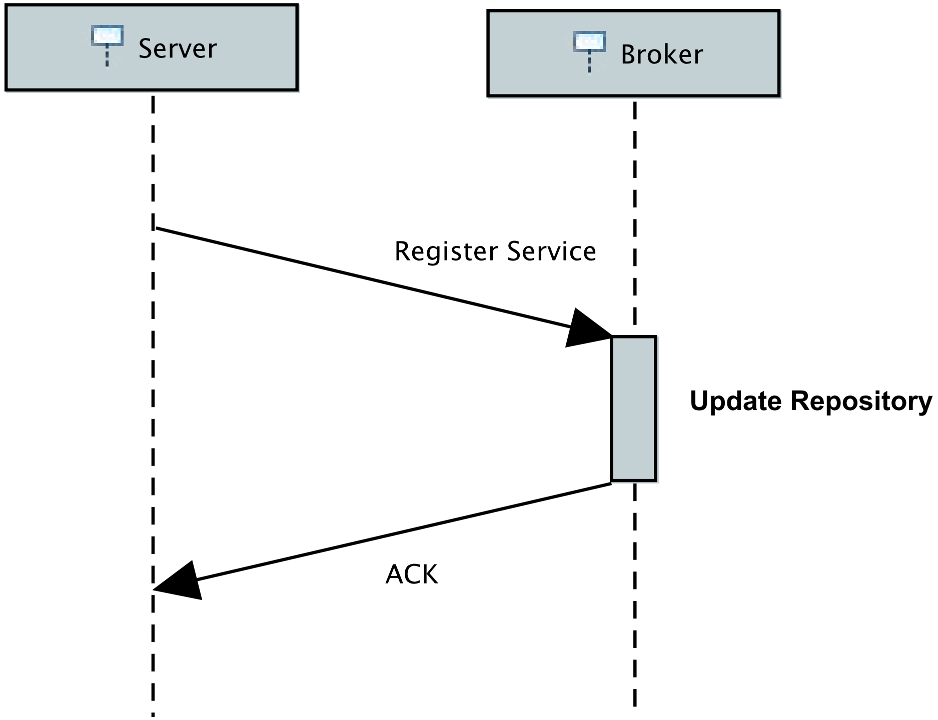
- **Context:** Systems build from several services distributed across servers
- **Problem:** How to hide details about the location of service providers, letting us change the binding dynamically?

- **Note:** Some authors refer to Broker as a pattern instead to as an style

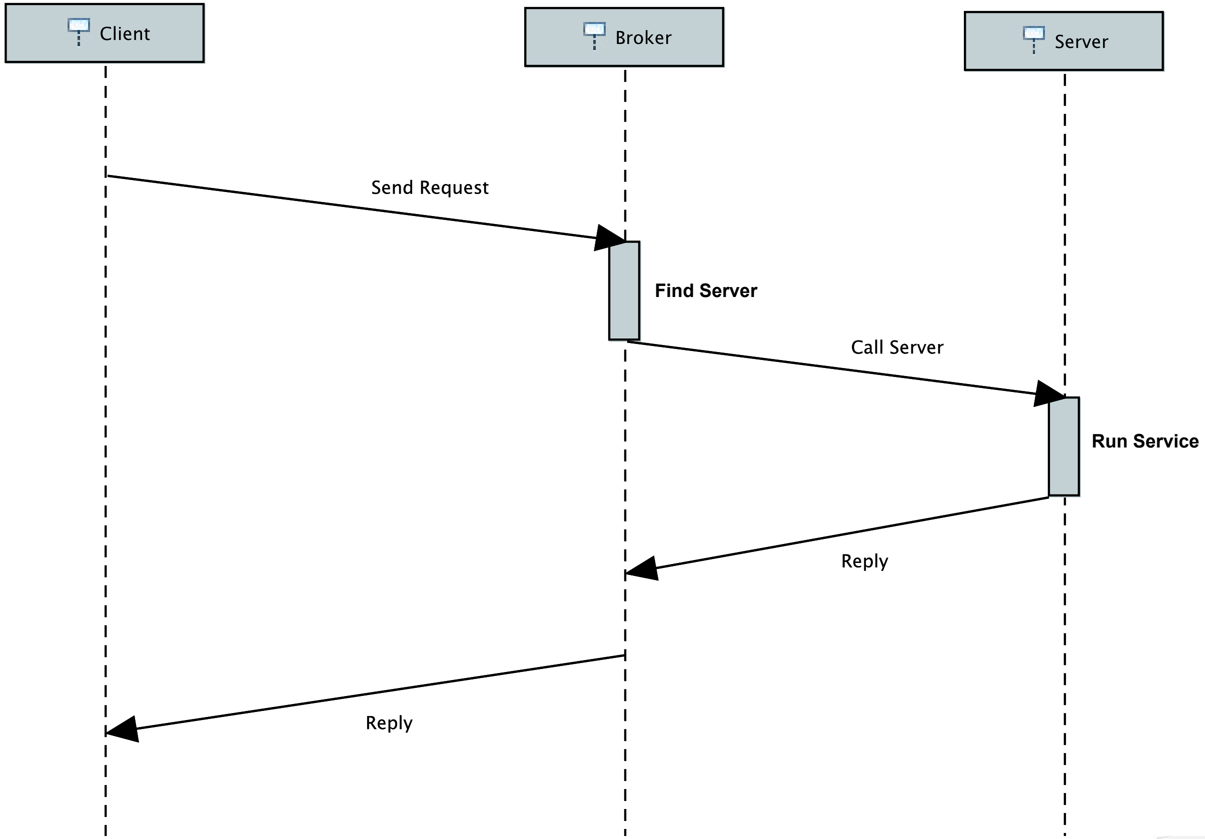
Broker

- **Solution:**
  - The broker pattern can be used to structure distribute systems with decoupled components
  - A broker component is added in the middle to mediate the communication between a number of clients and servers
    - Handles requests
    - Transforms / Encodes the inputs & outputs
    - Transmits results
    - Transmits exceptions

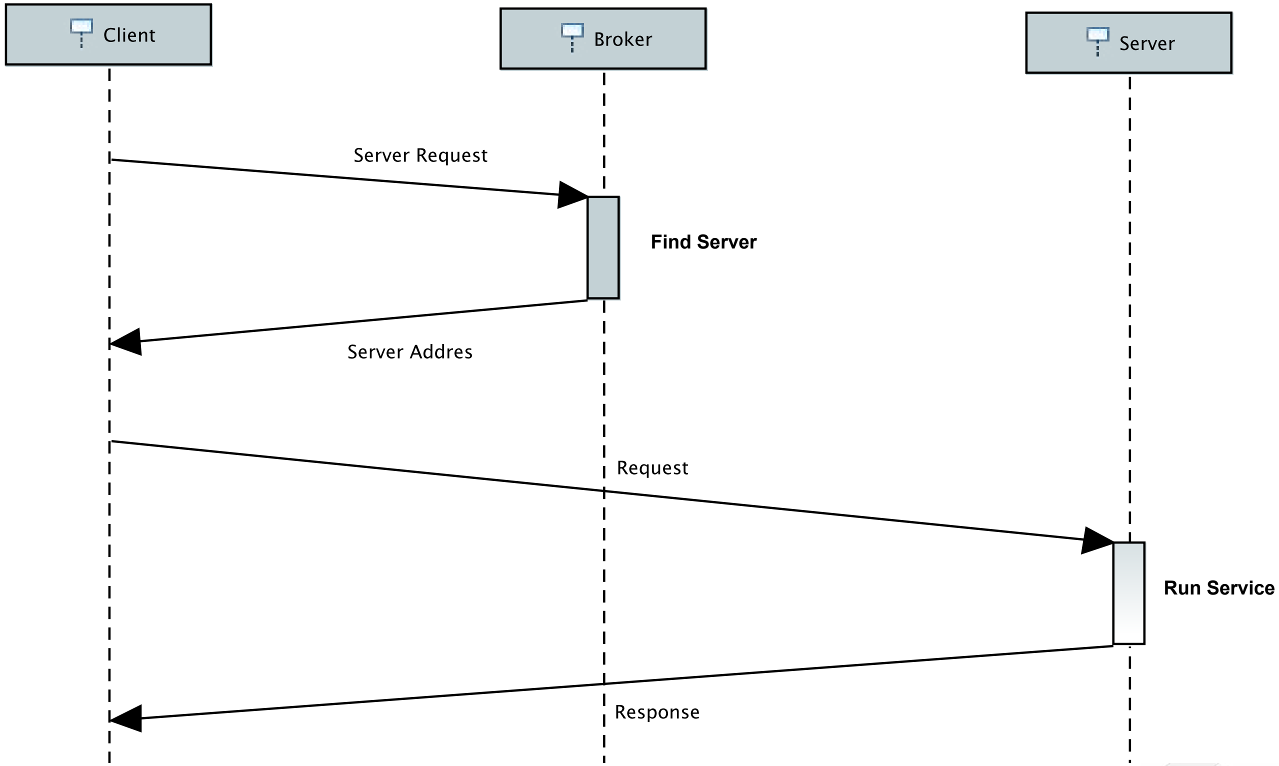
Broker



Broker


Client
Broker
Server
Send Request
Find Server
Call Server
Run Service
Reply
Reply

Broker: Variations



Broker: Impact on Qualities

Pros

- **Modifiability:**
  - The *use-an-intermediary* improves modifiability, since breaks the dependency client-server
  - We can modify/substitute a server without clients noticing
- **Availability:**
  - We can substitute the server without clients noticing
  - Promotion of backup servers by the broker

Cons

- **Performance**
  - Communication overheads might have severe impact, hindering latency

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: Can the broker have benefits on the performance?**

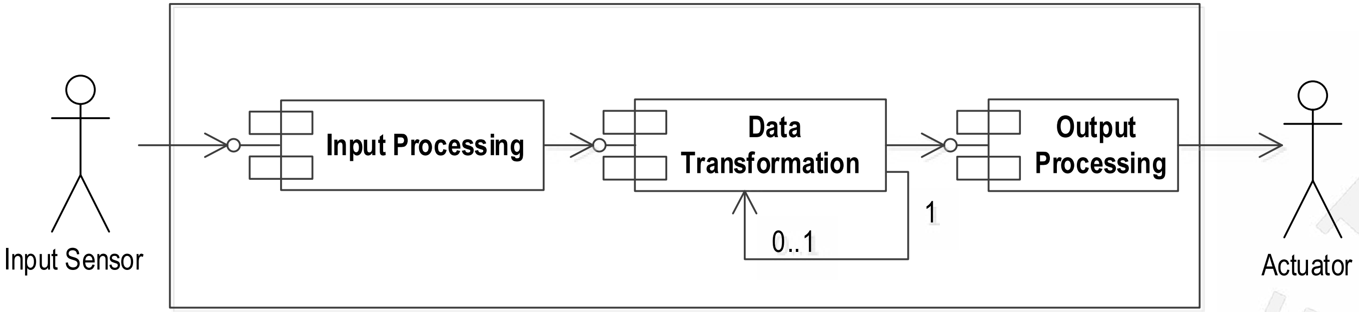
Architectural Patterns and Styles

Fault-Tolerance / Safety Critical Architectural Patterns

Safety-Critical Architectural Patterns

- Watchdog
- Sanity-Check
- Homogenous Redundancy Pattern
- Triple Redundancy Pattern

Typical Real-Time System



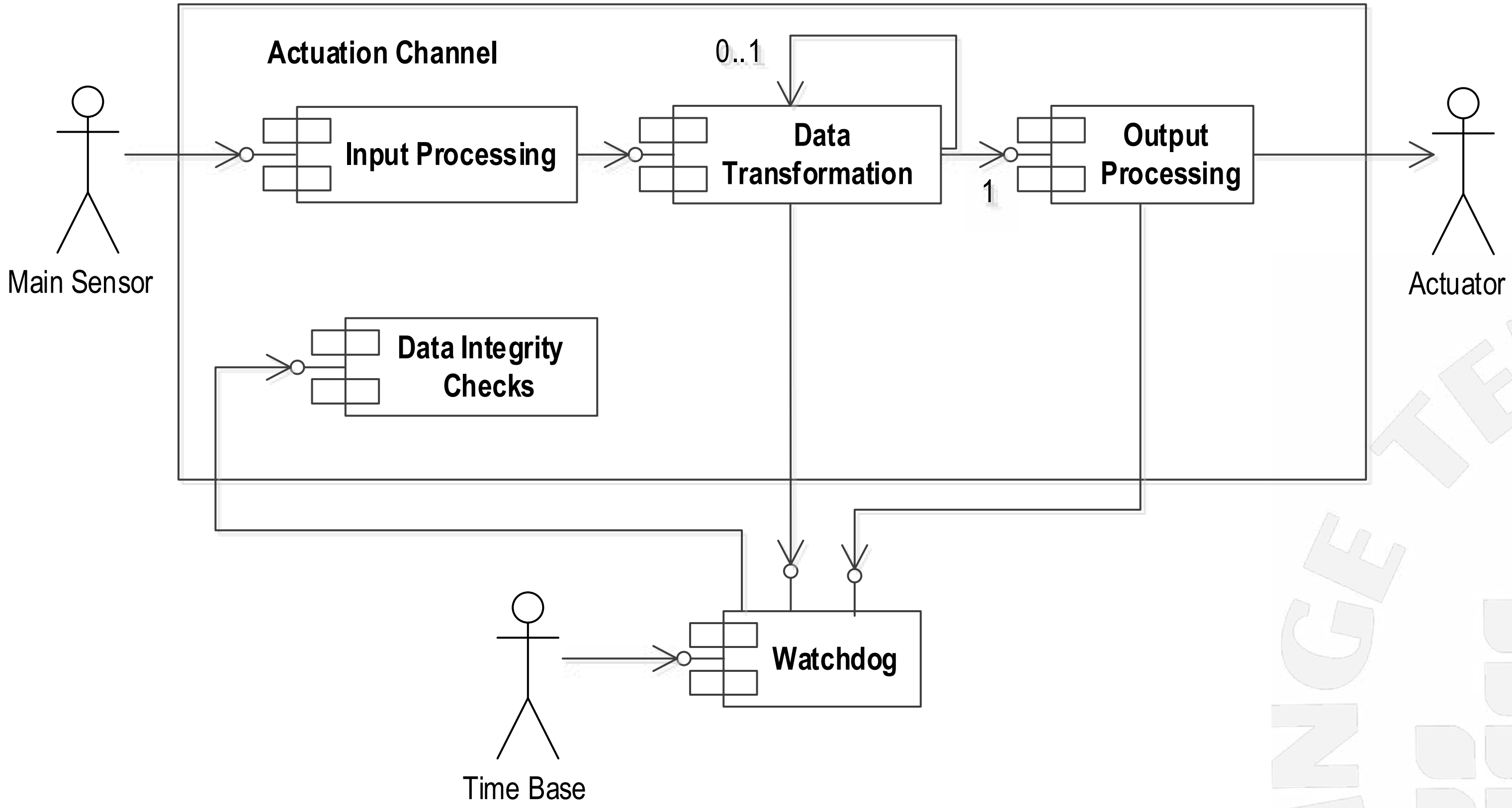
Watchdog Pattern

- **Context**
  - Real-time systems are predictable timely
  - The computations have a deadline by which they must be applied
  - If the computation occurs after that deadline, the result may either be erroneous or irrelevant
  - If the output comes too late, the system cannot be controlled
    - the system will be in an unstable region.
- **Problem:**
  - How to implement systems in which we have mechanisms to detect when the calculations are being performed?

Watchdog Pattern

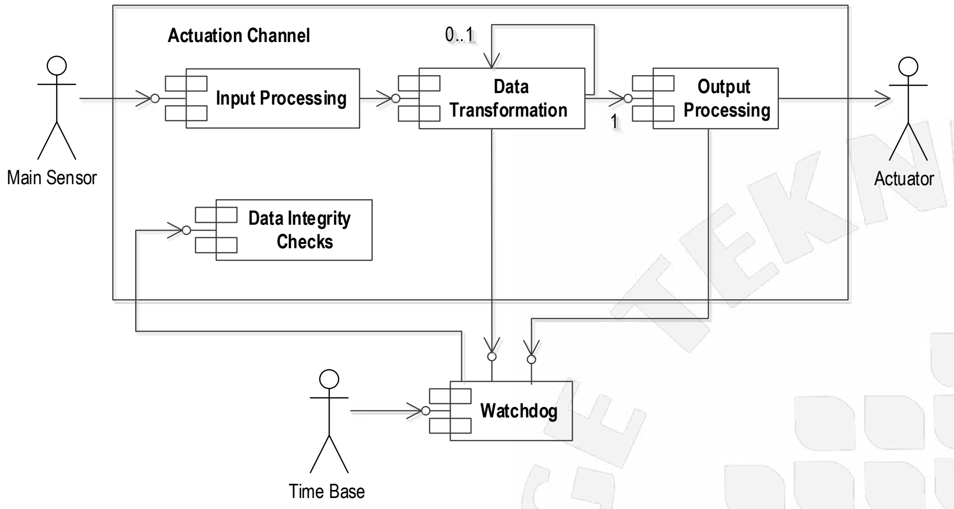
- Lightweight pattern that provides minimal coverage against faults
- Only checks that the internal computational processing is proceeding as expected
- This means that its coverage is minimal
  - A broad set of faults will not be detected.

Watchdog Pattern



Watchdog Pattern

- The actuator channel sends a liveness messages
  - AKA: stroking the watchdog
- The watchdog uses the timeline of the strokes to monitor if a fault has occurred
- It is able to report the fault to the actuation channel



Watchdog: Impact on Qualities

Pros:

- Fault-Tolerance
  - Its coverage for fault detection is minimal
  - The Watchdog Pattern is a very lightweight pattern that is rarely used alone in safety-critical systems

Cons:

- Performance
  - Since is very lightweight the impact on performance is minimal

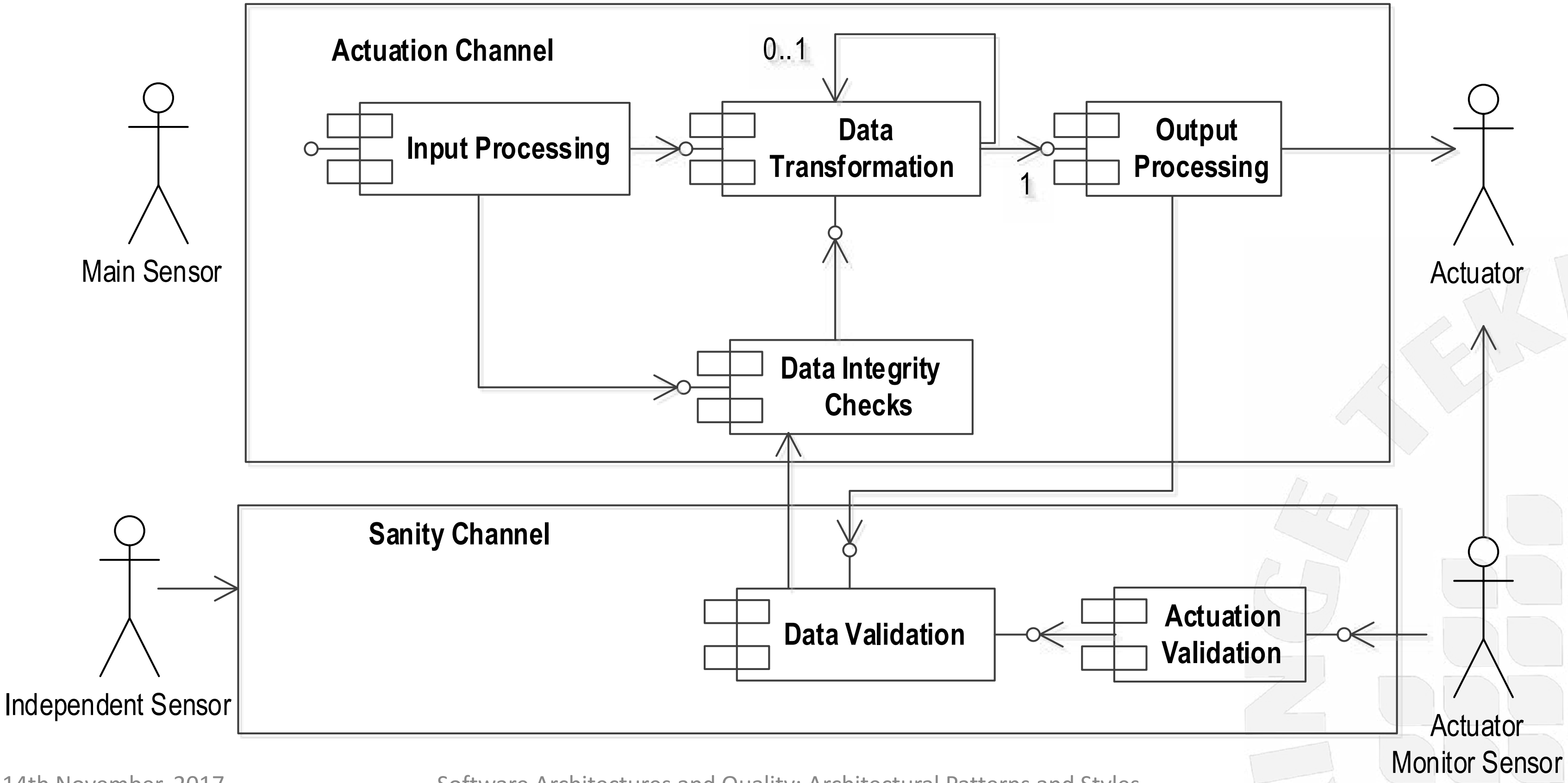
Sanity-Check

- Context:
  - In certain RT systems the outputs are *roughly* predictable (at least in order of magnitude)
- Problem:
  - How to assure that the system is more or less doing something reasonable, even if not quite correct
  - This is useful when the output is not critical if performed correctly (such as an optional enhancement) but is capable of doing harm if it is done incorrectly

Sanity-Check

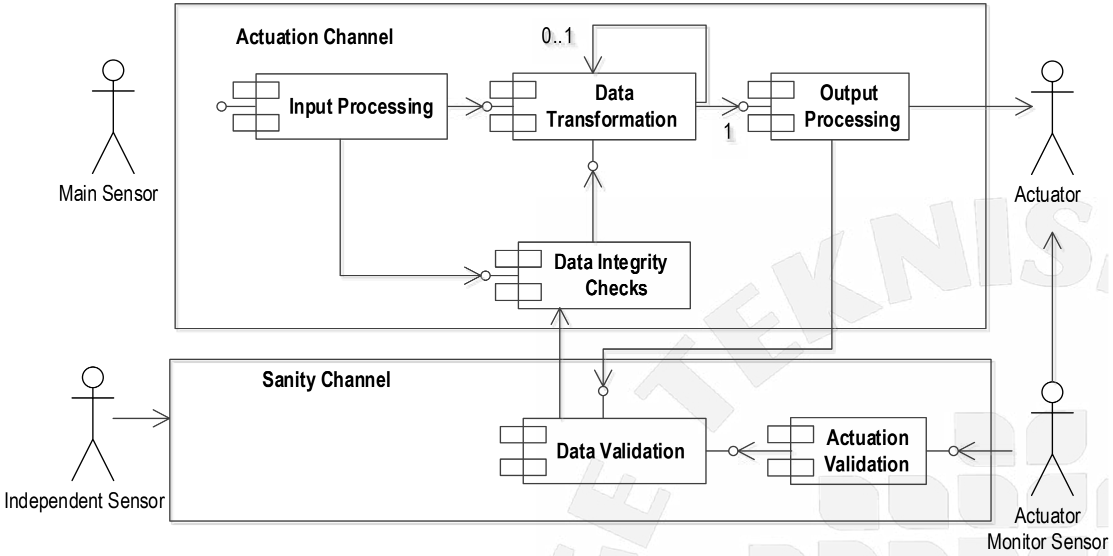
- The sanity check pattern is another lightweight pattern
- Makes sure the "system does no harm" when minor, or moderate, deviations from the commanded set point have no safety impact,
- Providing this minimal level of protection at a very low recurring and design cost.

Sanity Check



Sanity Check

- Makes use of an additional sensor and the actuator checking that the proper value that is really being applied
  - Usually just a simple verification that the commanded set point is somewhere in a relatively broad range
  - does not attempt to replicate the accuracy of the actuation channel
- Both channels (primary and sanity-check) run independently and simultaneously.



Sanity Check: Impact on Qualities

Pros:

- **Safety & Fault-Tolerance**
  - Yet another simple, inexpensive solution to improve safety and fault-tolerance

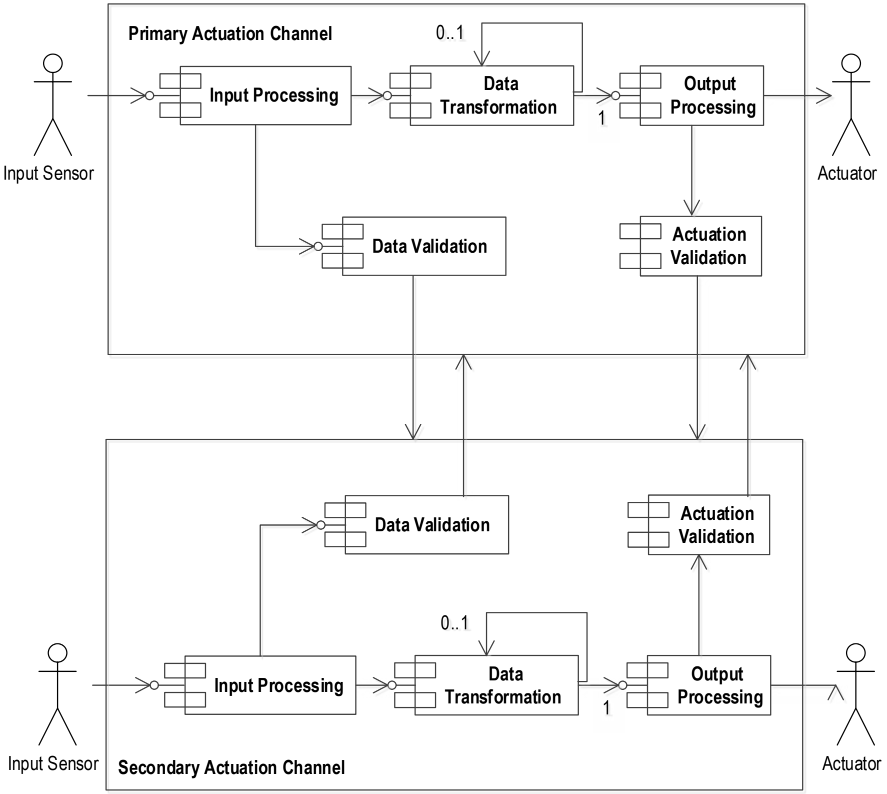
Cons:

- **Performance**
  - Since is very lightweight the impact on performance is minimal

Homogenous Redundancy Pattern

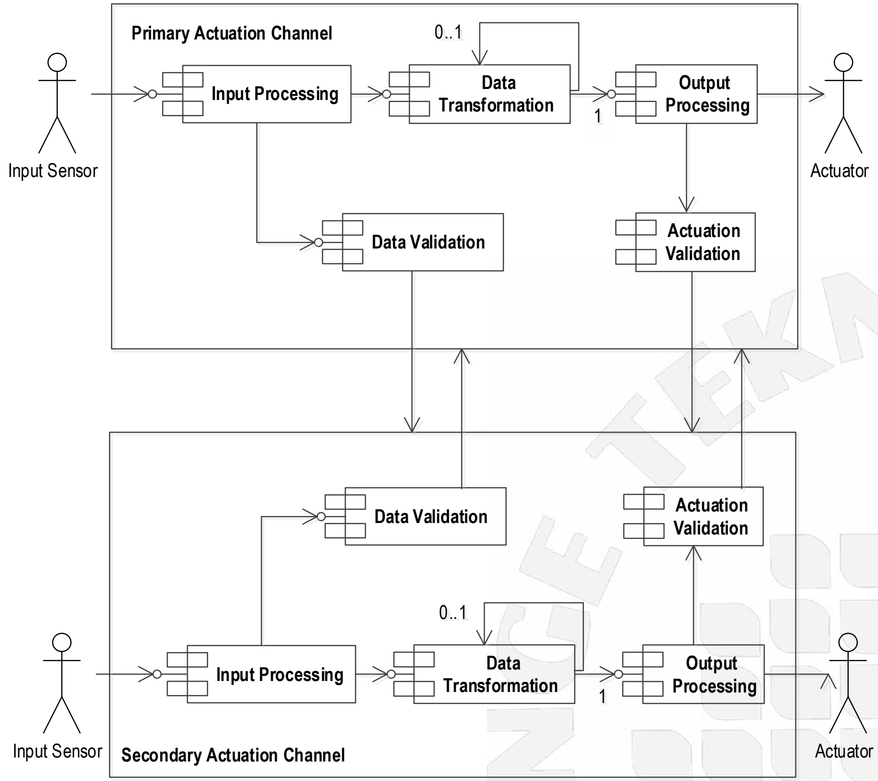
- **Context:**
  - In some safety critical systems, being able to delay the output
  - An obvious solution to avoid that something breaks is provide a copy
- **Problem:**
  - How to provide coverage to random faults?
  - How to continue functioning in the presence of failures?

Homogenous Redundant Pattern



Homogenous Redundancy Pattern

- Provides protection against random faults in the system execution
- The channels operate in sequence (never at the same time)
- Allows continue providing functionality in the presence of a failure.
- The primary channel continues running as long as there are no problems.
- In case of failure within the channel, the system is able to detect the fault and switch to the backup channel.



HR Pattern: Impact on Qualities

Pros:

- **Fault-Tolerance**
  - The system is able to continue operating even in case of faults

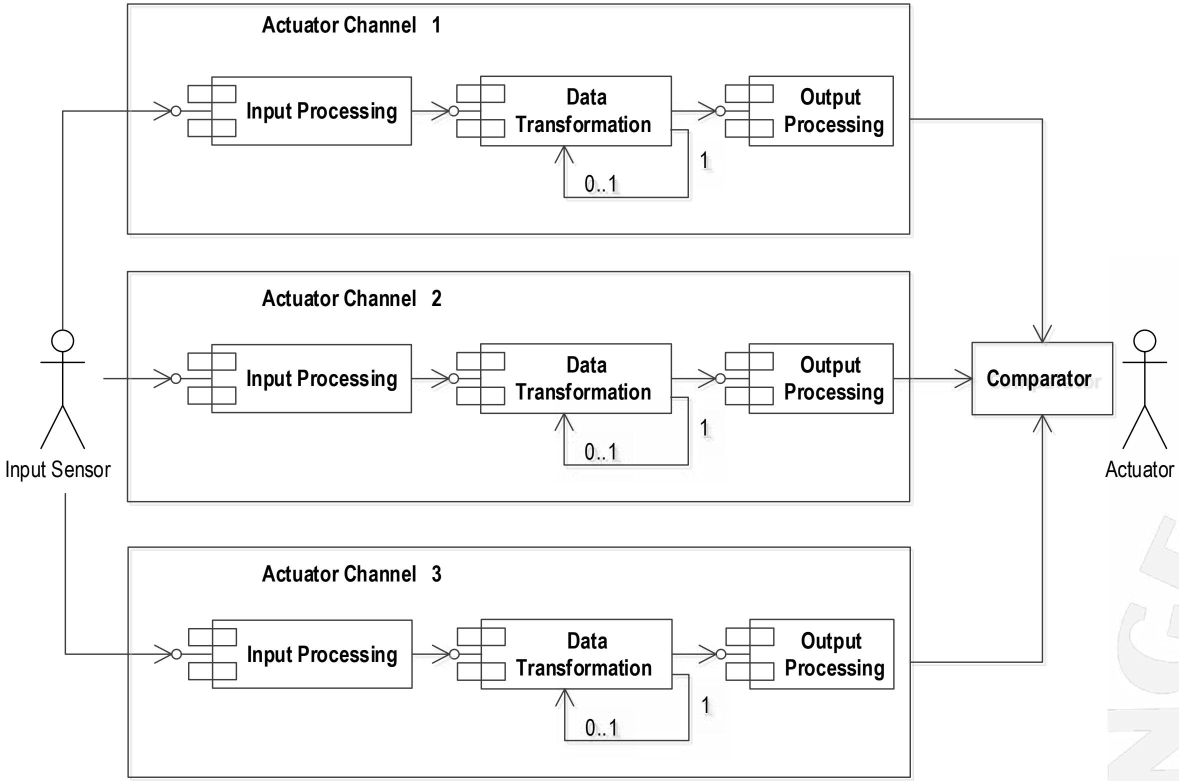
Cons:

- **Performance**
  - We can at least double the latency in case of failure
- **Resource consumption**
  - In this case we are doubling the resource consumption in terms of memory
  - We add some overhead for the data validation and actuator validation

Triple-Modular Redundancy (TMR) Pattern

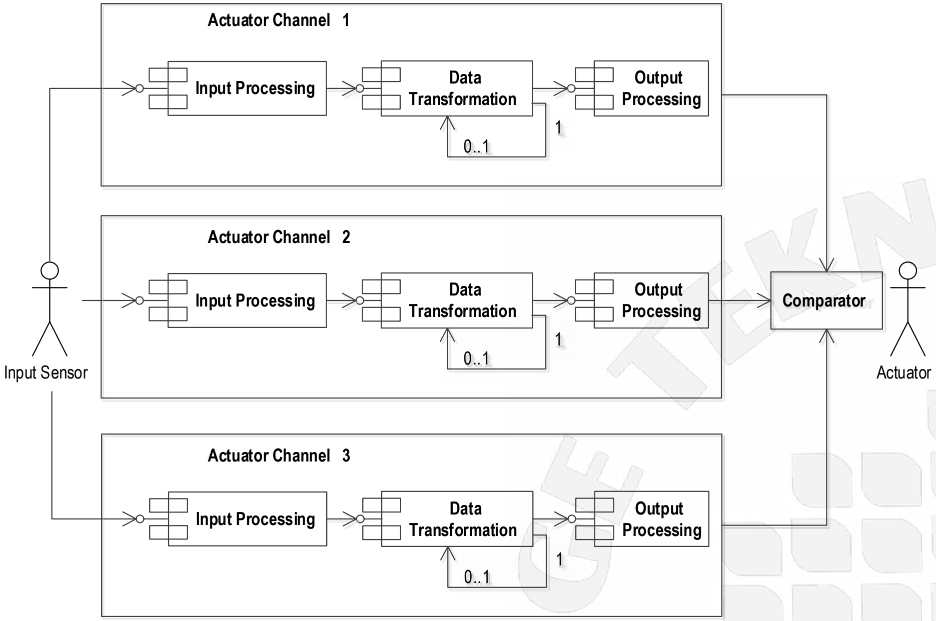
- **Context**
  - In some systems the output cannot be delayed another whole cycle
  - We cannot lose the input (is critical to provide an output per execution cycle)
- **Problem:**
  - How to provide protection against random faults? (with some additional constraints)
    - When a fault is detected, the input data should not be lost
    - No additional time be required to provide a response

TMR Pattern


Actuator Channel 1
Input Processing
Data Transformation
Output Processing
0..1
1
Actuator Channel 2
Input Processing
Data Transformation
Output Processing
0..1
1
Actuator Channel 3
Input Processing
Data Transformation
Output Processing
0..1
1
Input Sensor
Comparator
Actuator

TMR Pattern

- Adding three copies running in parallel and a comparator
- As long as two channels agree on the output, then any deviating computation of the third channel is discarded
- The system can operate in the presence of a fault



TMR Pattern: Impact on Qualities

Pros:

- **Fault-Tolerance & Safety**
  - Improves fault tolerance as HR was doing

Cons:

- **Performance**
  - We add some extra latency time for the comparator
- **Resource consumption**
  - In this case he have more than three times of memory consumptions
  - We also have more than triple CPU consumption since all the systems run in parallel

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: Do you think Homogenous Redundancy and Triple-Modular provides coverage for any kind of faults? What happens with systematic, recurrent faults?**

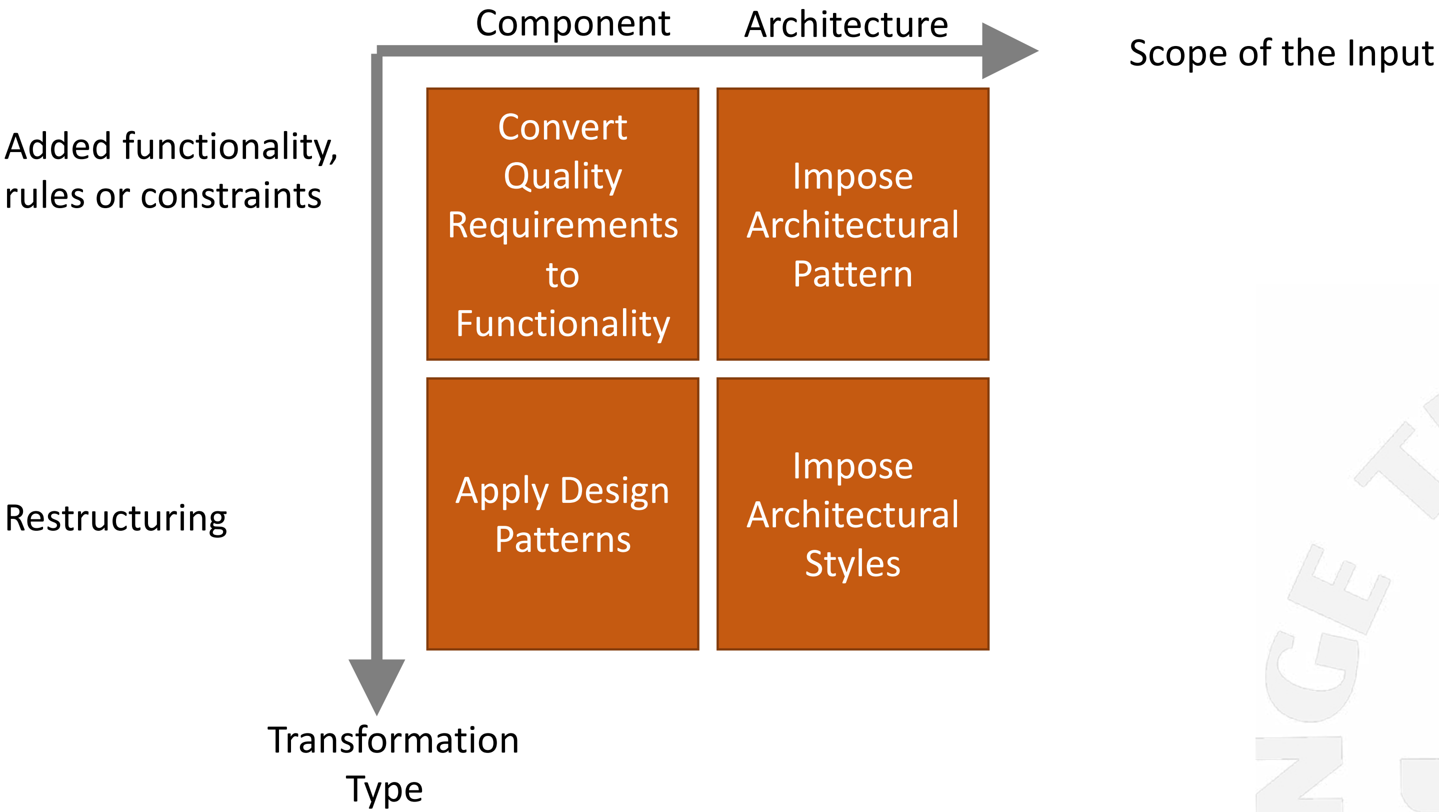
Architectural Patterns and Styles

Summing Up Patterns

Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: What do you think is the difference between architectural styles, architectural patterns and design patterns?**

The Map



	Component	Architecture
Added functionality, rules or constraints	Convert Quality Requirements to Functionality	Impose Architectural Pattern
Restructuring	Apply Design Patterns	Impose Architectural Styles

J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley Publishing Co., 2000.

Choosing Architectural Patterns

- Will depend on:
  - System´s nature
  - Quality attribute requirements:
    - always a tradeoff between alternatives
- Consider alternate patterns
- Often a single pattern will not be enough for the whole system.

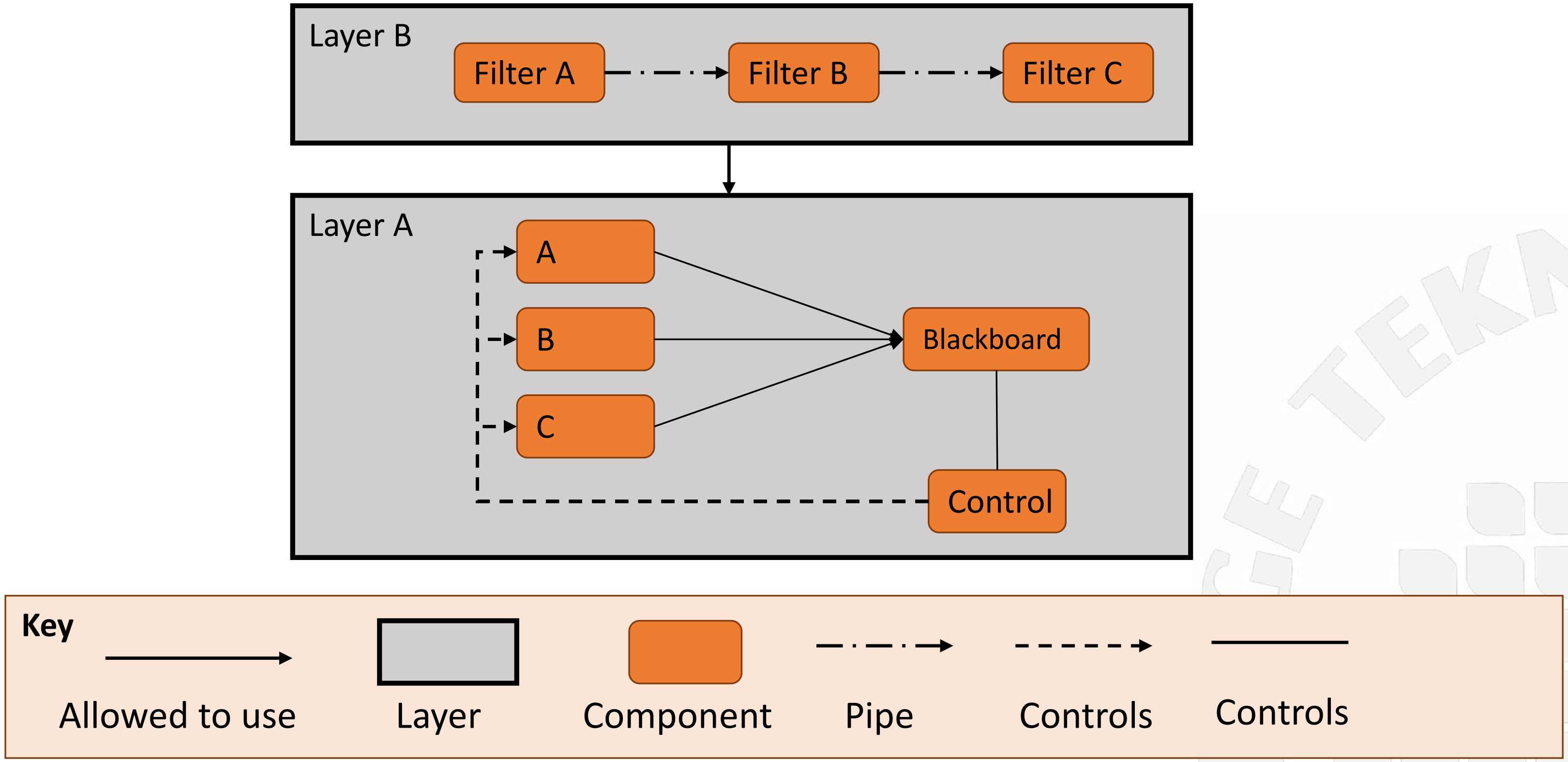
Reflection

- **Discuss during 2 minutes in groups of 2-3 persons: Do you think styles and patterns can be combined when designing an architecture?**

Combining Patterns

- Pattern can be combined *if*:
  - Conflicts between constraints can be resolved
  - If styles are isolated in separate components / parts of the architecture
- Some styles are orthogonal and can be merged (since they do the constraints don’t collide between them).
- We will have one dominant style for the whole architecture and local solutions (styles or patterns) applied in certain components / areas of the architecture

Combining Patterns



In the next episodes…

- We will go back to patterns and styles during the architecture evaluation and transformation, and during the design of your architecture

Software Architectures and Quality

Architectural Patterns and Styles

Javier González Huerta

javier.gonzalez.huerta@bth.se