Architectural Patterns

Layers, broker, MVC, ...

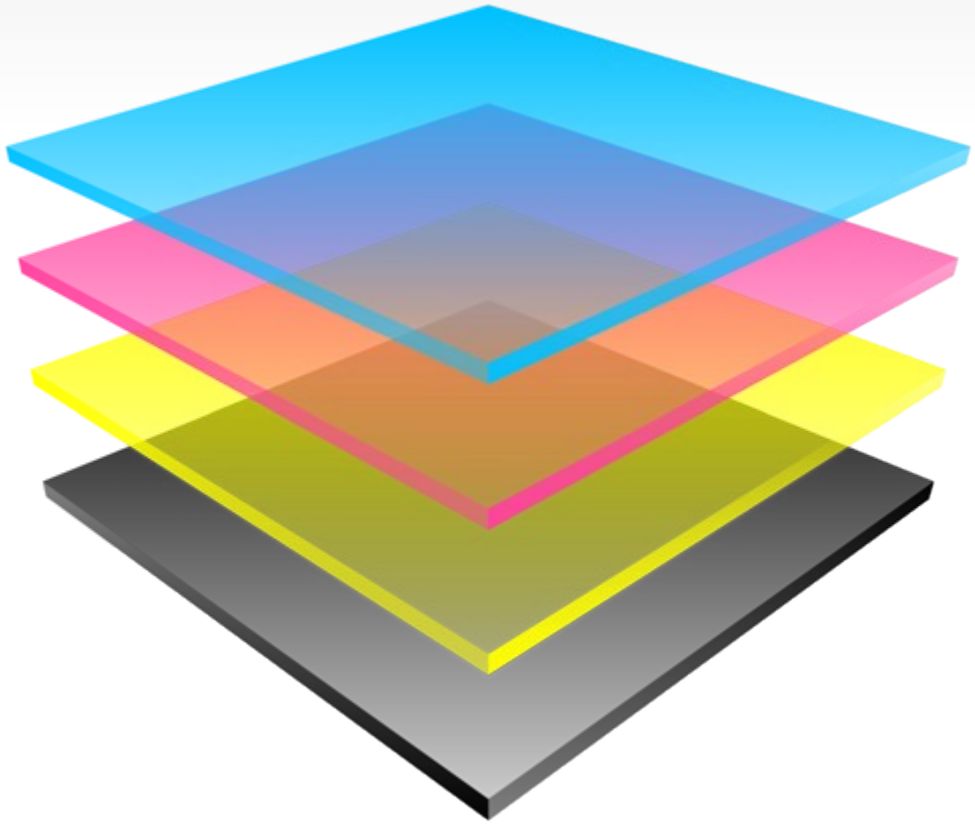
© R. Ošlejšek
Faculty of Informatics
Masaryk University
oslejsek@fi.muni.cz

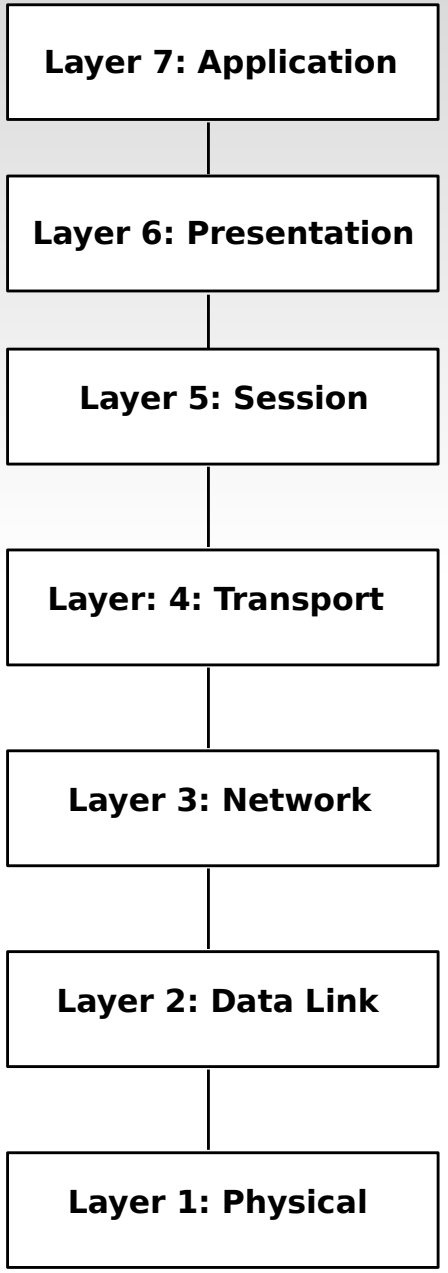
Architectural Patterns

Architectural pattern is named collection of architectural design decisions that are applicable to design problems appearing over and over again, which are parametrized under various contexts of software development. [Taylor et al.]

Layers

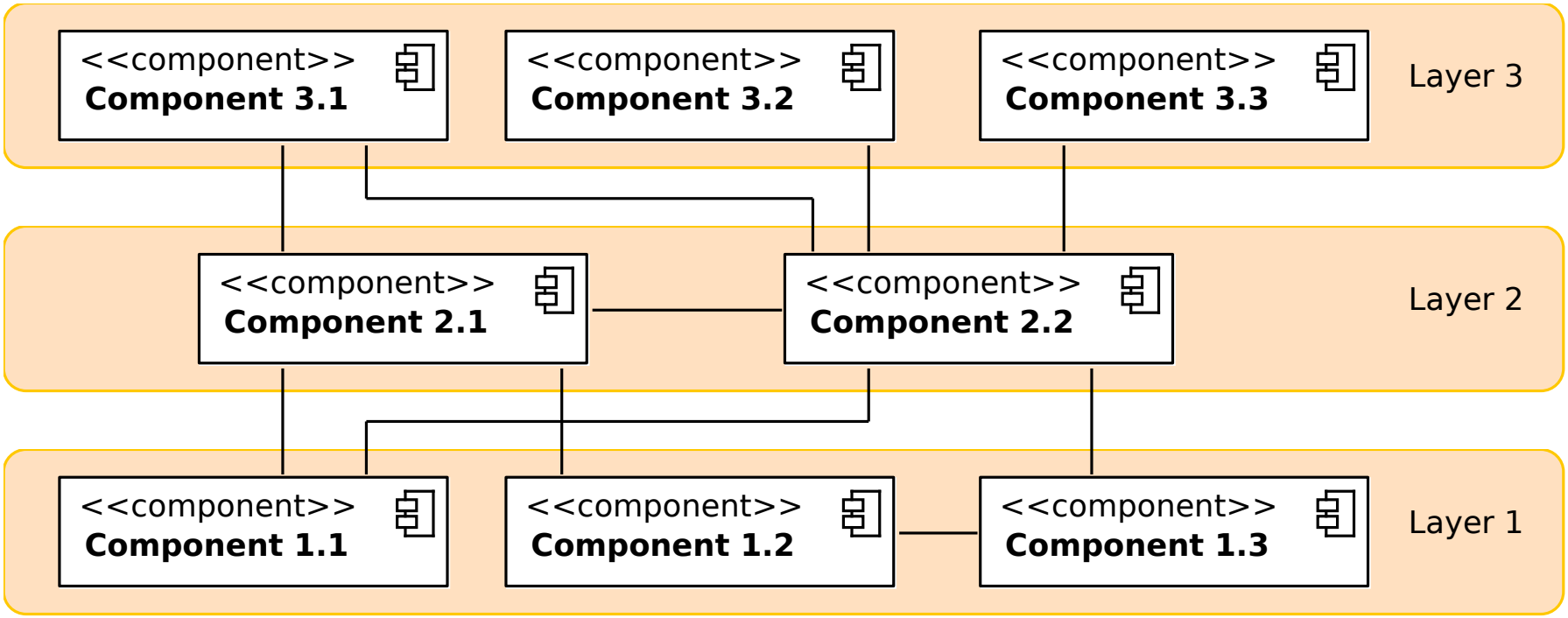
- Helps to structure applications that can be decomposed into *groups of subtasks* in which each group of subtask is at a *particular level of abstraction*.
- [Buschmann et al: Pattern-Oriented Software Architecture, 1997]





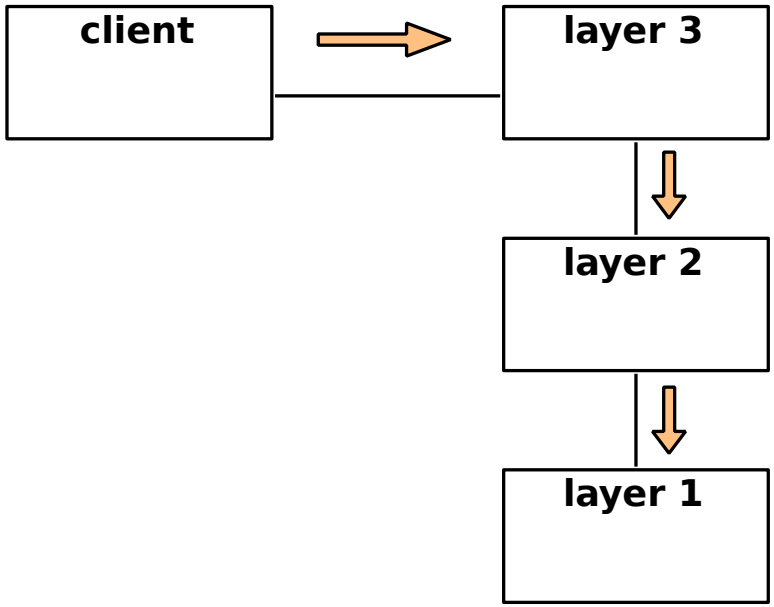
Layers – Structure

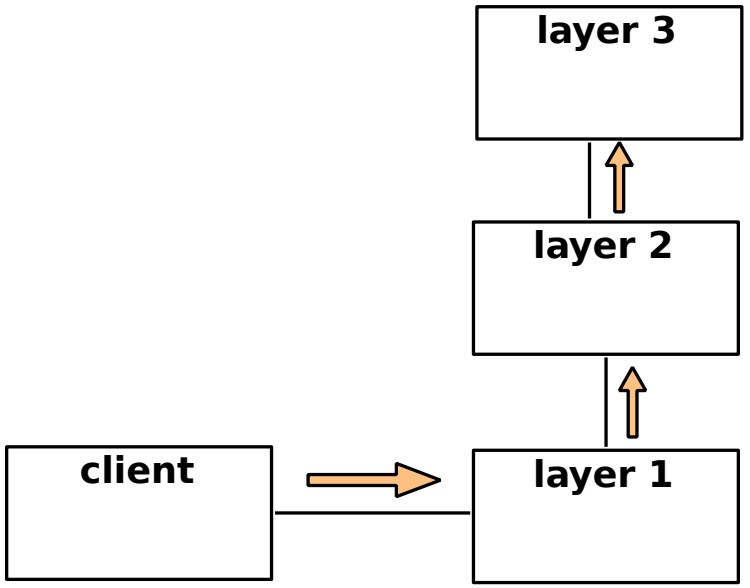
- Higher layers are on a higher level of abstraction (i.e. more general, higher level services).
- Layer N provides services used by layer N+1
- Layer N delegates subtasks to layer N-1
- Services of a layer N are only used by layer N+1 (there are no further direct dependencies between layers)
- Layers are complex entities consisting of different components



Layers – Dynamics

- Top-down communication: A client issues a request to layer N. Layer N calls the next layer N-1 for supporting subtasks and so on until layer 1 is reached. Layer N often translates a single request into several requests to layer N-1.
- Bottom-up communication: A chain of actions starts at the lower layer 1, e.g. when a device driver detects input. Upper layers are notified and provided data are interpreted by the upper layers.
- Communication through a subset of the layers: For example, if layer N acts as a cache which is able to satisfy a request without sending sub-requests to lower layers. Usually requires statefull implementation of components. This scenario is applicable to the both top-down and bottom-up communication directions.





Layers – Variants

- Relaxed Layered System:
  - Each layer may use the services of all layers bellow it, not only of the next lower layer.
  - Higher flexibility and performance
  - The loss of maintainability
- Layering Through Inheritance
  - Layers implemented as base classes.
  - A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.
  - Higher layer can modify lower-layer services
  - Closely ties the layers (recompilation of upper layer classes after the change in lower layers, etc.)

Layers – Known Uses

- Many information systems from the business software domain.
- Virtual machines: insulates higher levels from low-level details or varies hardware, e.g. JVM.
- APIs: Encapsulates lower layers of frequently used functionality.

Layers – Consequences

- Benefits:
  - *Reuse* of layers due to well-defined abstraction
  - *Support for standardization* due to clearly-defined and commonly-accepted levels of abstraction.
  - *Dependencies are kept local*, which supports testability, for instance.
    - Proxy pattern for remote connection of layers
  - *Exchangeability* of layers
    - Adapter pattern for interface adaptation
    - Bridge pattern for dynamic exchange (manipulating the pointer to the implementation at run-time)
- Liabilities:
  - *Cascades of changing behavior* of (lower) layers
  - *Lower efficiency* due to the communication overhead
  - *Unnecessary work*, e.g. duplicate work (error detection even if lower services are reliable)
  - Difficulty of establishing the correct *granularity of layers*. Too few layers restricts reusability, changeability and portability. Too many layers introduce complexity and communication overhead.

Layers or tiers?

- Logical **layers** are merely a way of organizing your code. Typical layers include Presentation, Business and Data – the same as the traditional 3-tier model. But when we're talking about layers, we're only talking about logical organization of code. In no way is it implied that these layers might run on different computers or in different processes on a single computer or even in a single process on a single computer. All we are doing is discussing a way of organizing a code into a set of layers defined by specific function.
- Physical **tiers** however, are only about where the code runs. Specifically, tiers are places where layers are deployed and where layers run. In other words, tiers are the physical deployment of layers.

Pipes and Filters

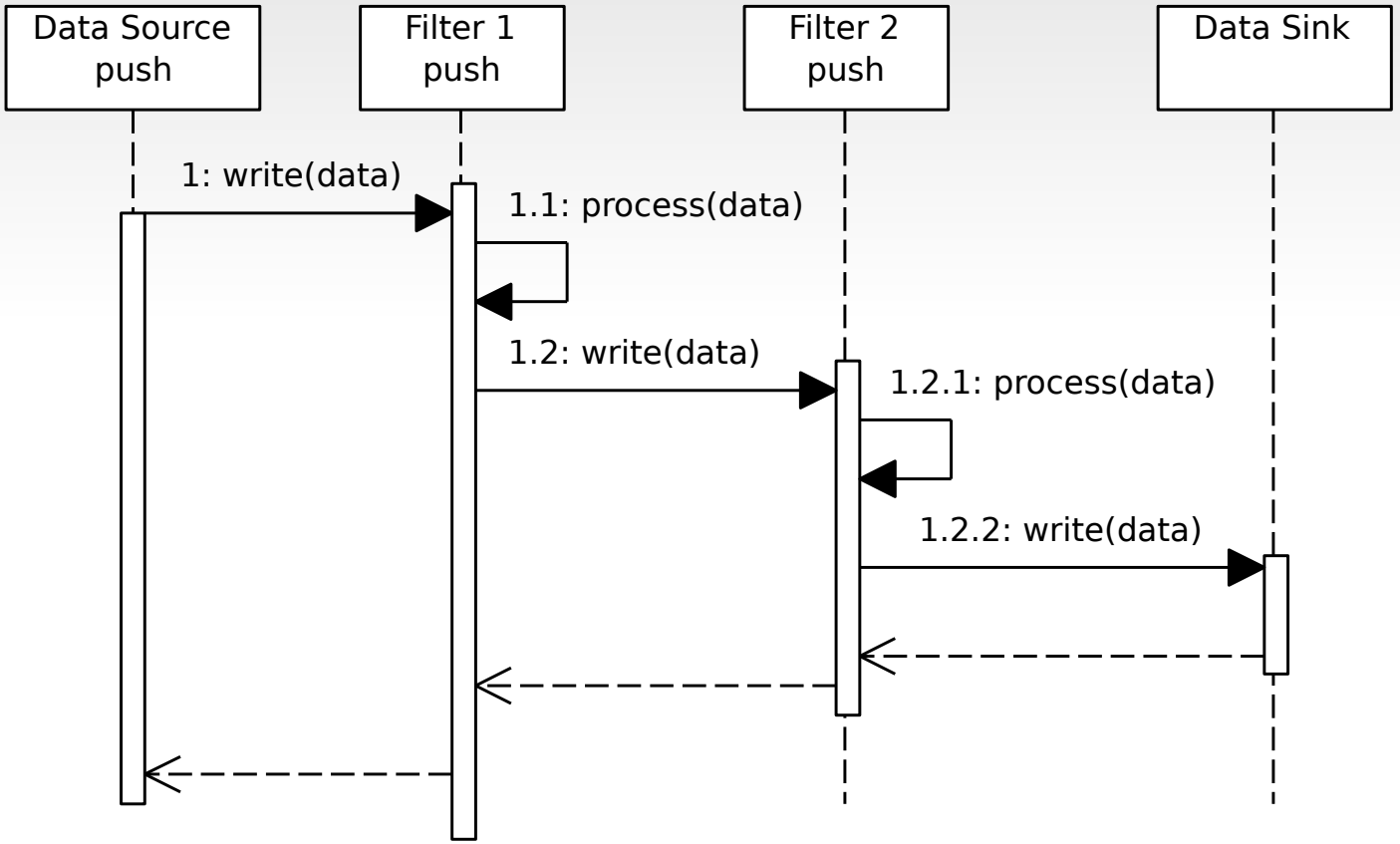
- Provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data are passed through pipes between adjacent filters. Recombining filters allows as to build families of related systems.
- [Buschmann et al: Pattern-Oriented Software Architecture, 1997]

Pipes and Filters – Structure

- Pipe: Denotes the connection between filters.
- Passive filter: The subsequent pipeline element pulls output data from the filter OR the previous pipeline element pushes new input data to the filter.
- Active filter: the filter is active in a loop, puling its input from and pushing its output.
- Data source: represents the input to the system
- Data sink: collects the results from the end of the pipeline.

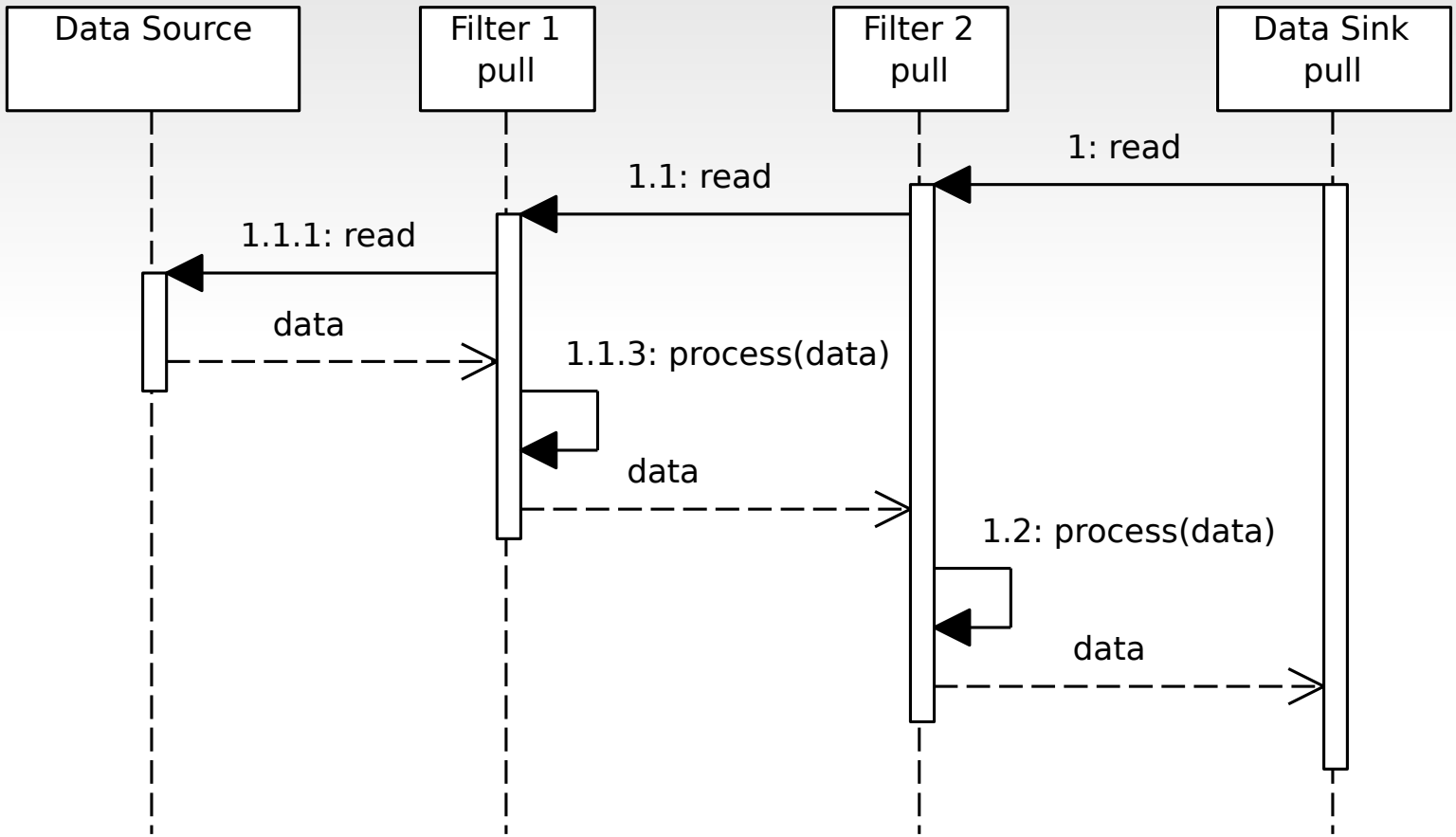
Pipes and Filters – Dynamics I

- Push pipeline (passive filters)



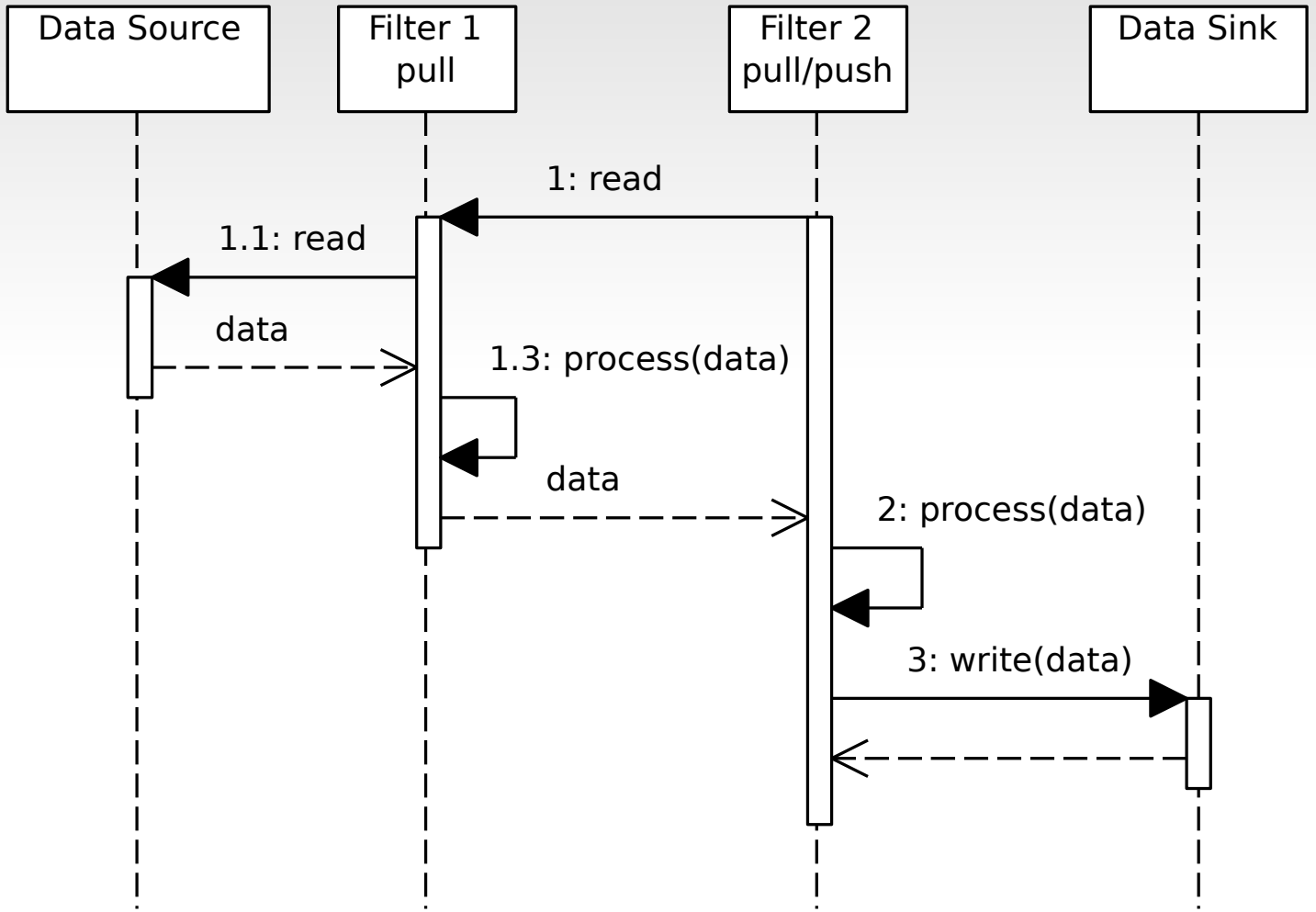
Pipes and Filters – Dynamics II

- Pull pipeline (passive filters)



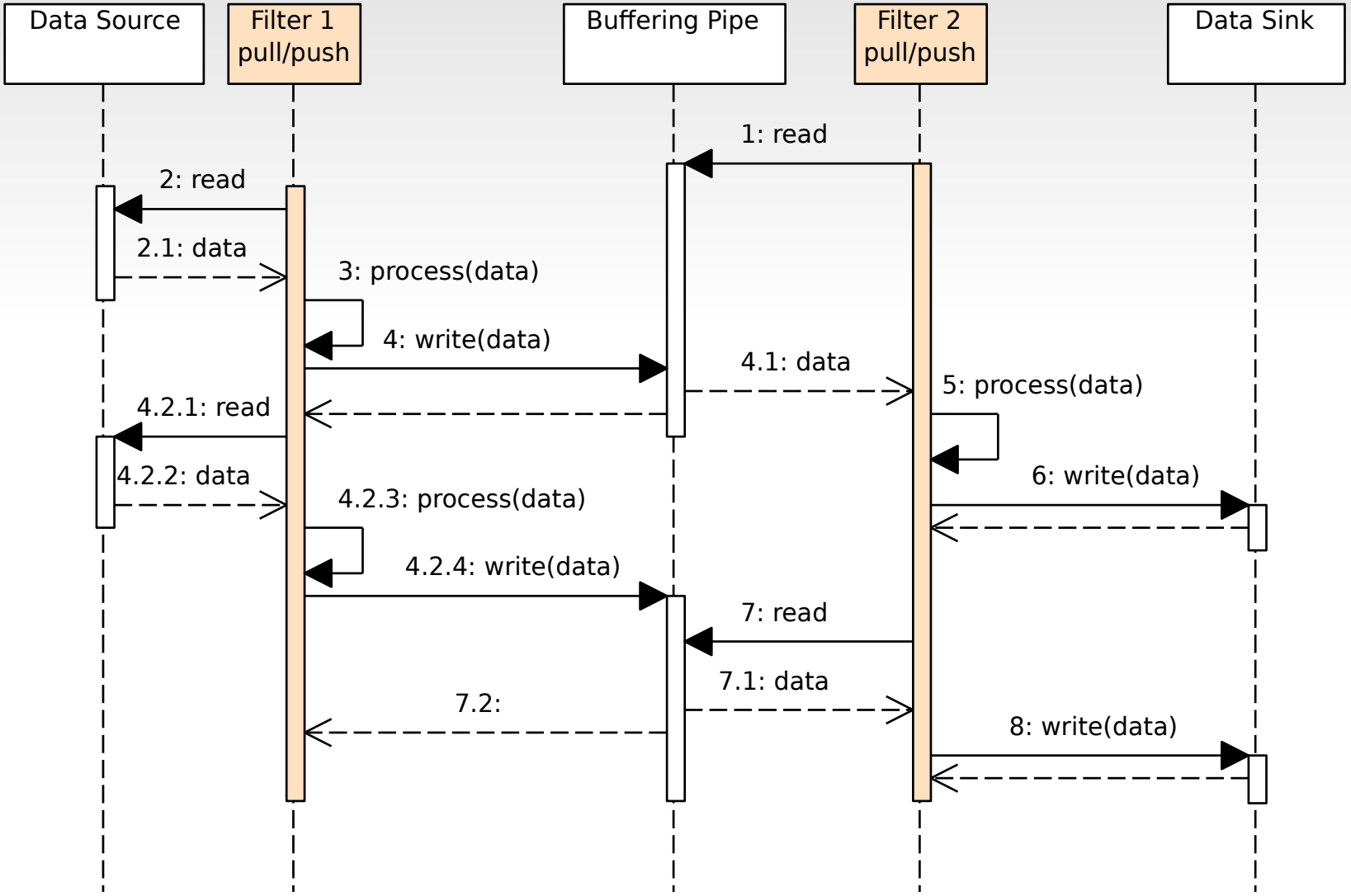
Pipes and Filters – Dynamics III

- Mixed push-pull pipeline (passive filters)



Pipes and Filters – Dynamics IV

- Active filters: The most common behavior, where all filters actively pull, compute, and push data in a loop.



Pipes and Filters – Known Uses

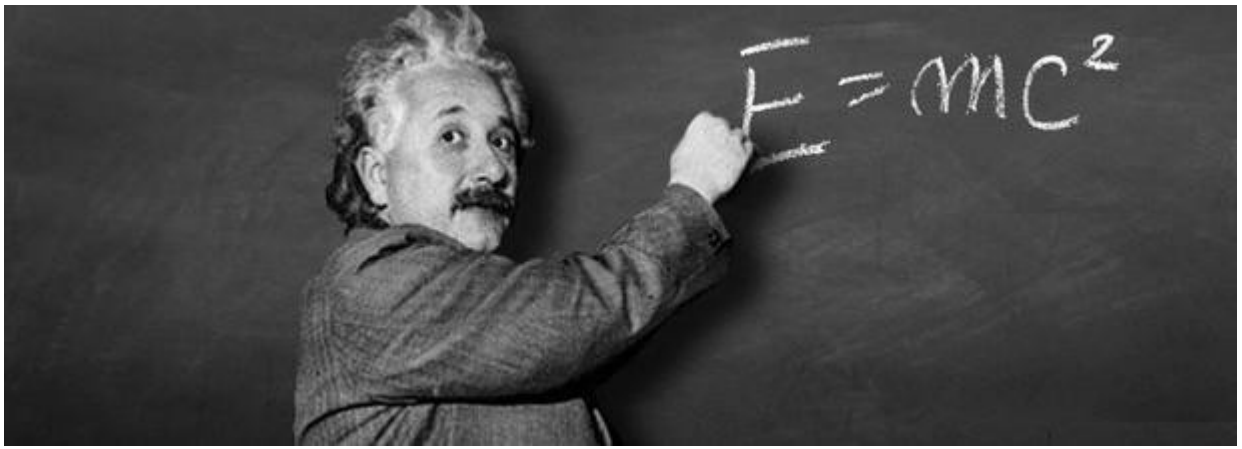
- UNIX
- complex event processing (CEP)
- graphics pipeline (OpenGL)
- ...

Pipes and Filters – Consequences

- Benefits:
  - No intermediate files necessary, but possible
  - Flexibility by filter exchange
  - Flexibility by recombination
  - Reuse of filter components
  - Rapid prototyping of pipelines
  - Efficiency by parallel processing
- Liabilities:
  - Sharing state information is expensive or inflexible
  - Efficiency gain by parallel processing may be an illusion (due to the cost for transferring data between filters, some filters consume all their input before producing any output, …)
  - Data transformation overhead (e.g. conversion of numbers to ASCII in every pipe)
  - Error handling, failure recovery

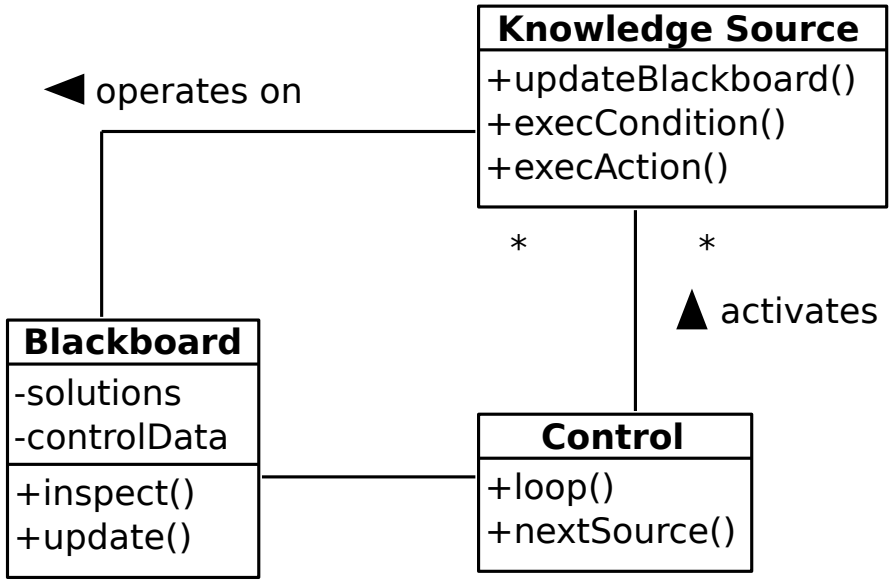
Blackboard

- Is useful for problems for which no deterministic strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial approximate solution.
- [Buschmann et al: Pattern-Oriented Software Architecture, 1997]
- Metaphor:
  - A group of specialists are seated in a room with a large blackboard. They work as a team to brainstorm a solution to a problem, using the blackboard as the workplace for cooperatively developing the solution.
  - The session begins when the problem specifications are written onto the blackboard. The specialists all watch the blackboard, looking for an opportunity to apply their expertise to the developing solution. When someone writes something on the blackboard that allows another specialist to apply their expertise, the second specialist records their contribution on the blackboard, hopefully enabling other specialists to then apply their expertise. This process of adding contributions to the blackboard continues until the problem has been solved.

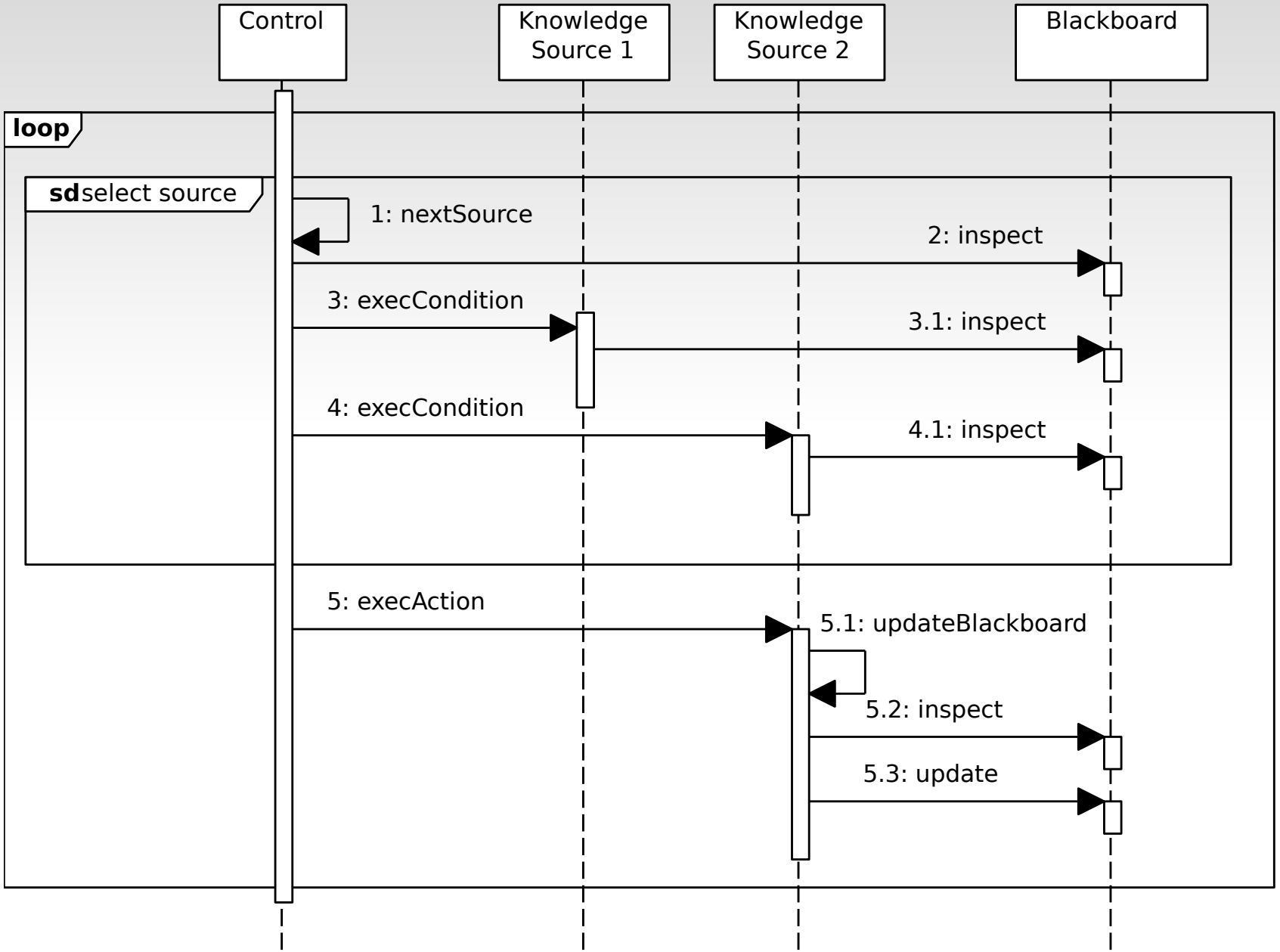


Blackboard – Structure

- Blackboard:
  - Central data store. We use the term *vocabulary* for the set of of all data elements that can appear on the blackboard.
  - Provides an interface that enables all knowledge sources to read from and write to it.
- Knowledge sources:
  - Separate independent subsystems that solve specific aspects of the overall problem.
  - They don't communicate directly but strictly throw the blackboard (they have to understand the vocabulary of the blackboard).
  - They usually operate on two levels of abstraction, transforming particular solution to a higher-level solution.
- Control:
  - Monitors blackboard.
  - Schedules knowledge source actions.



Blackboard – Dynamics



Blackboard – Known Uses

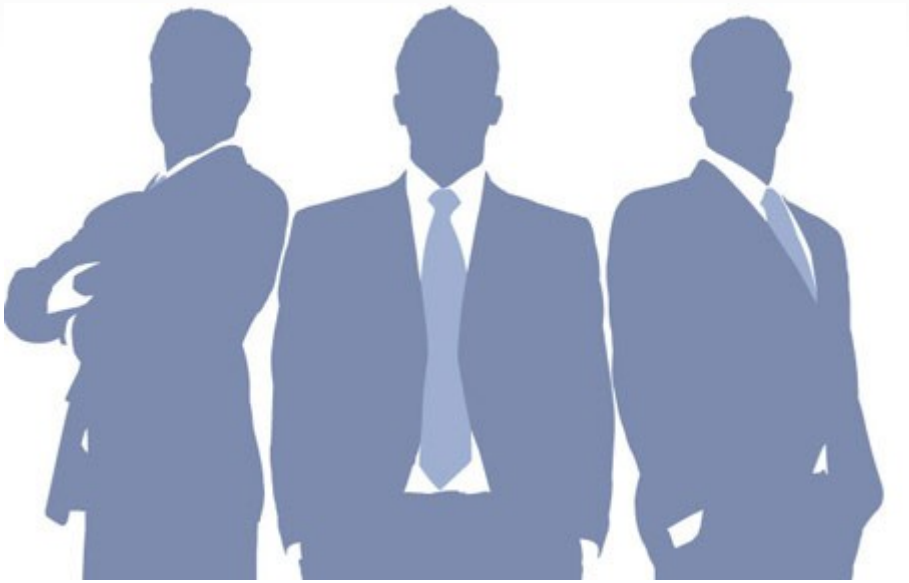
- Some modern *Bayesian machine learning* systems, using agents to add and remove Bayesian network nodes.

Blackboard – Consequences

- Benefits:
  - Experimentation, e.g. trying different control heuristics.
  - Support for changeability and maintainability due to the strict separation of individual knowledge sources, the control algorithm and the central data structure.
  - Reusable knowledge sources.
- Liabilities:
  - Difficulty of testing.
  - No good solution is guaranteed.
  - Difficulty to establish a good control strategy.
  - Low efficiency.
  - High development effort.
  - No support for parallelism.

Broker

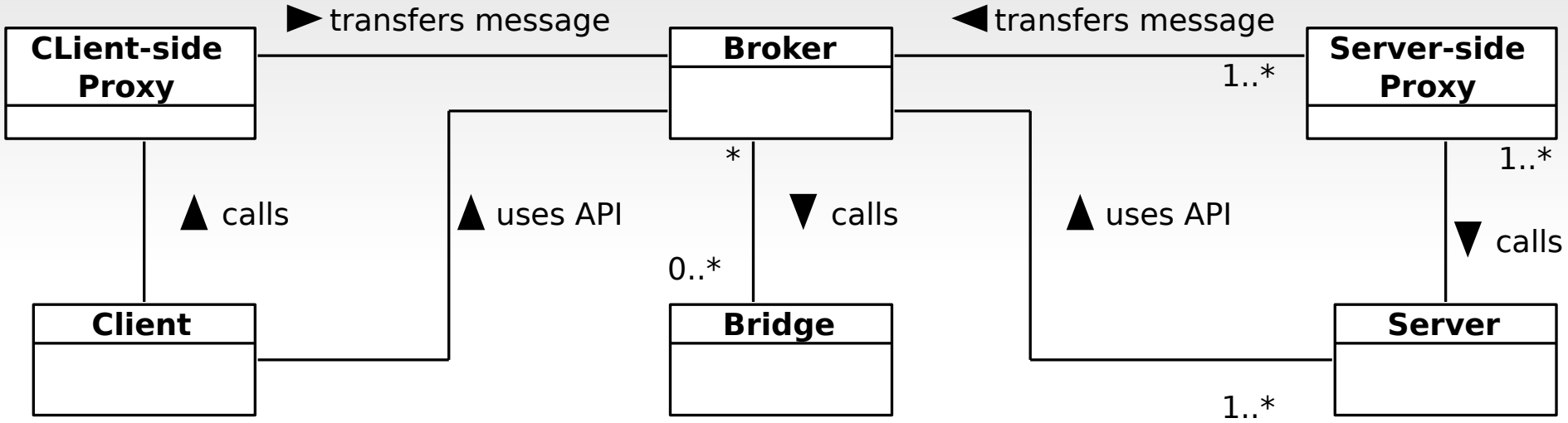
- Can be used to structure distributed software systems with decoupled components that interact by remote service invocation. A broker component is responsible for coordinating communication.
- [Buschmann et al: Pattern-Oriented Software Architecture, 1997]



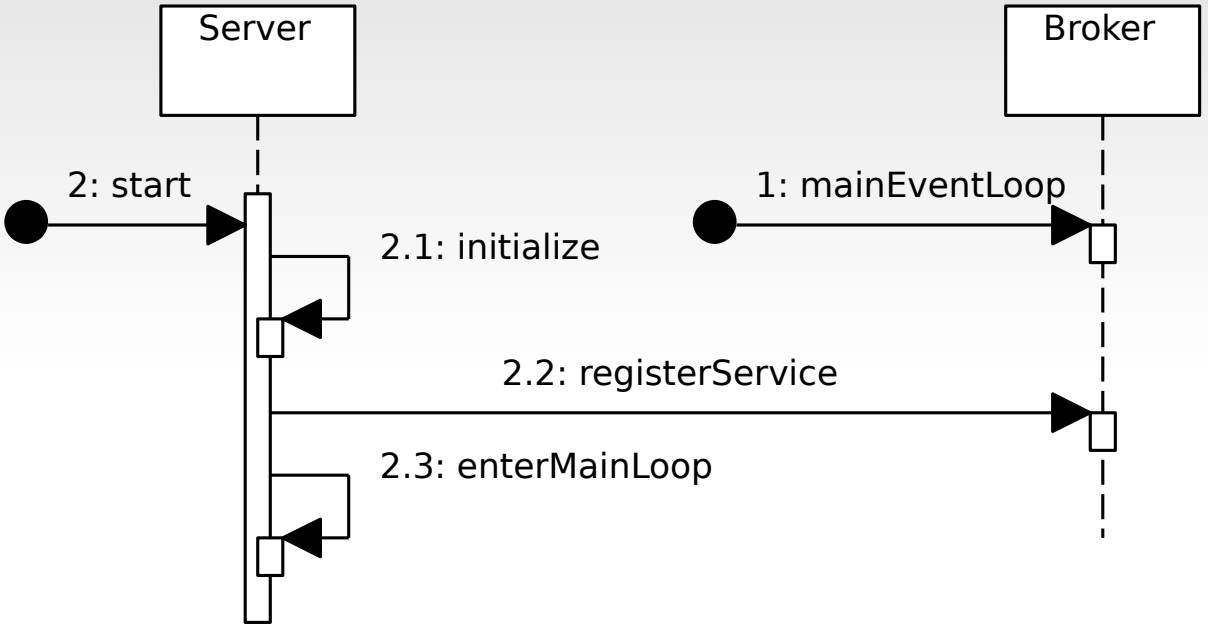
Broker – Structure

- Clients:
  - Implement user functionality. Send requests to servers. To call the remote service, clients forward requests to the broker. Client do not need to know the location of of the servers they access.
- Servers:
  - Implement services. Server may act as client.
- Brokers:
  - Messengers responsible for the transmission from clients to servers. If the specified server is hosted by another broker, the local broker finds a route to the remote broker.
- Bridges:
  - Optional components used for hiding implementation details when two broker cooperate.
- Client-side proxies:
  - Layer between clients and the broker providing transparency, i.e. a remote object appears to the client as a local one.
- Server-side proxies:
  - Analogous to Client-side proxies. Responsible for receiving requests.

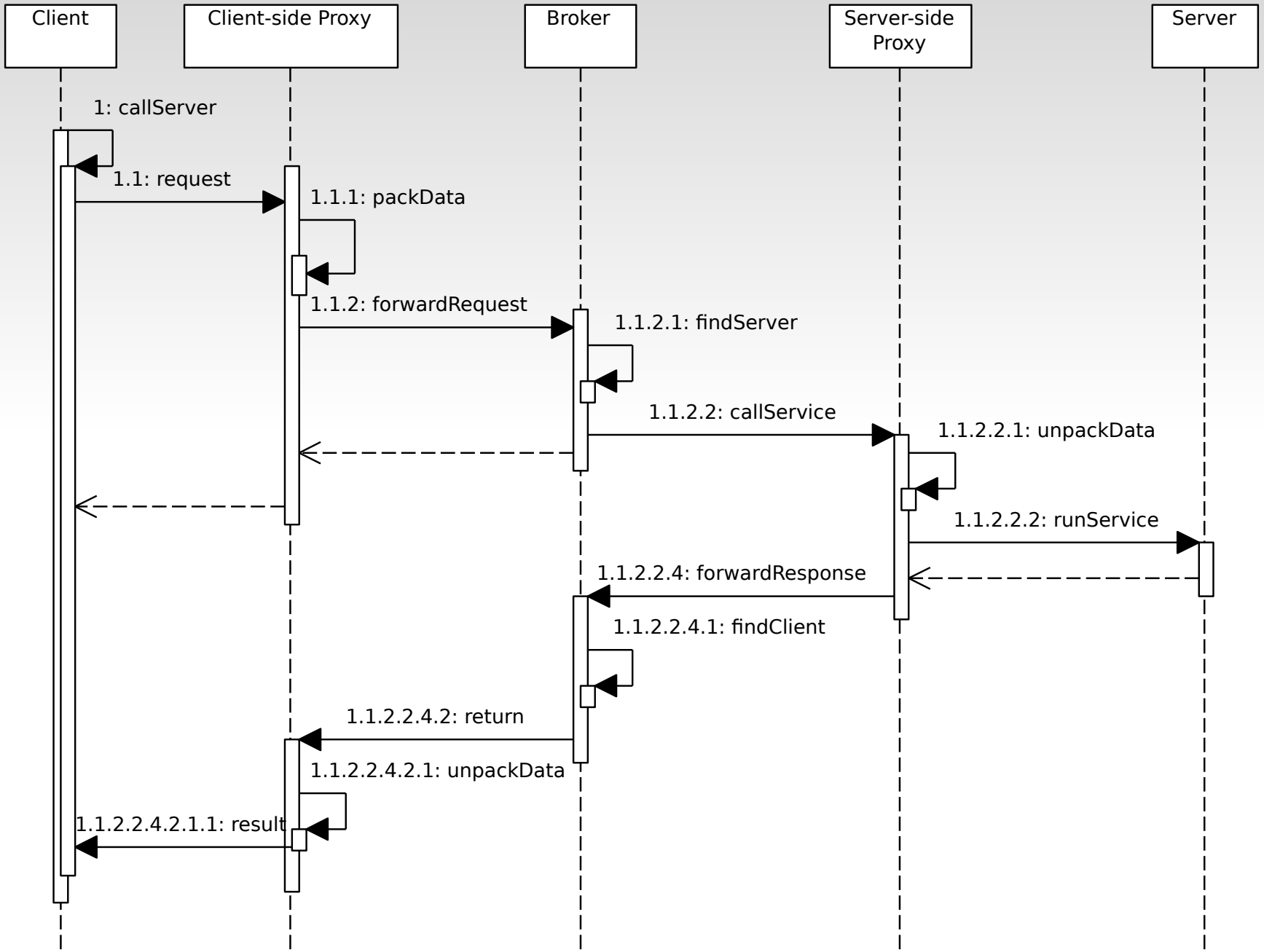
Broker – Structure (cont.)



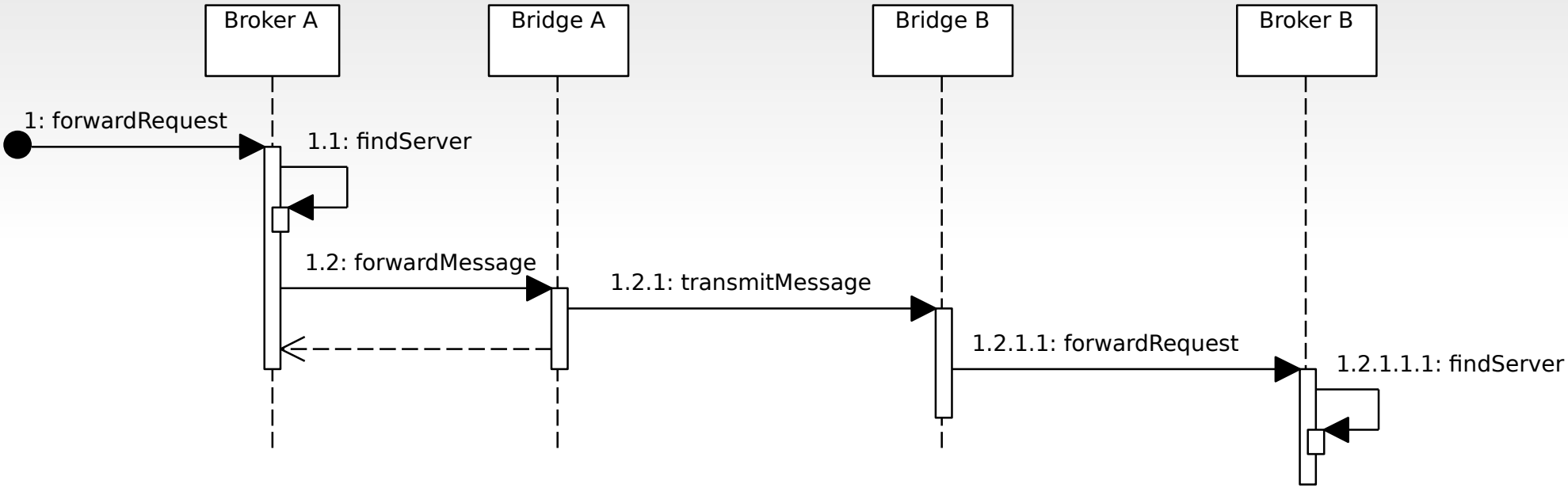
Broker – Dynamics (registration)



Broker – Dynamics (request to local srv.)



Broker – Dynamics (brokers interaction)



Broker – Known Uses

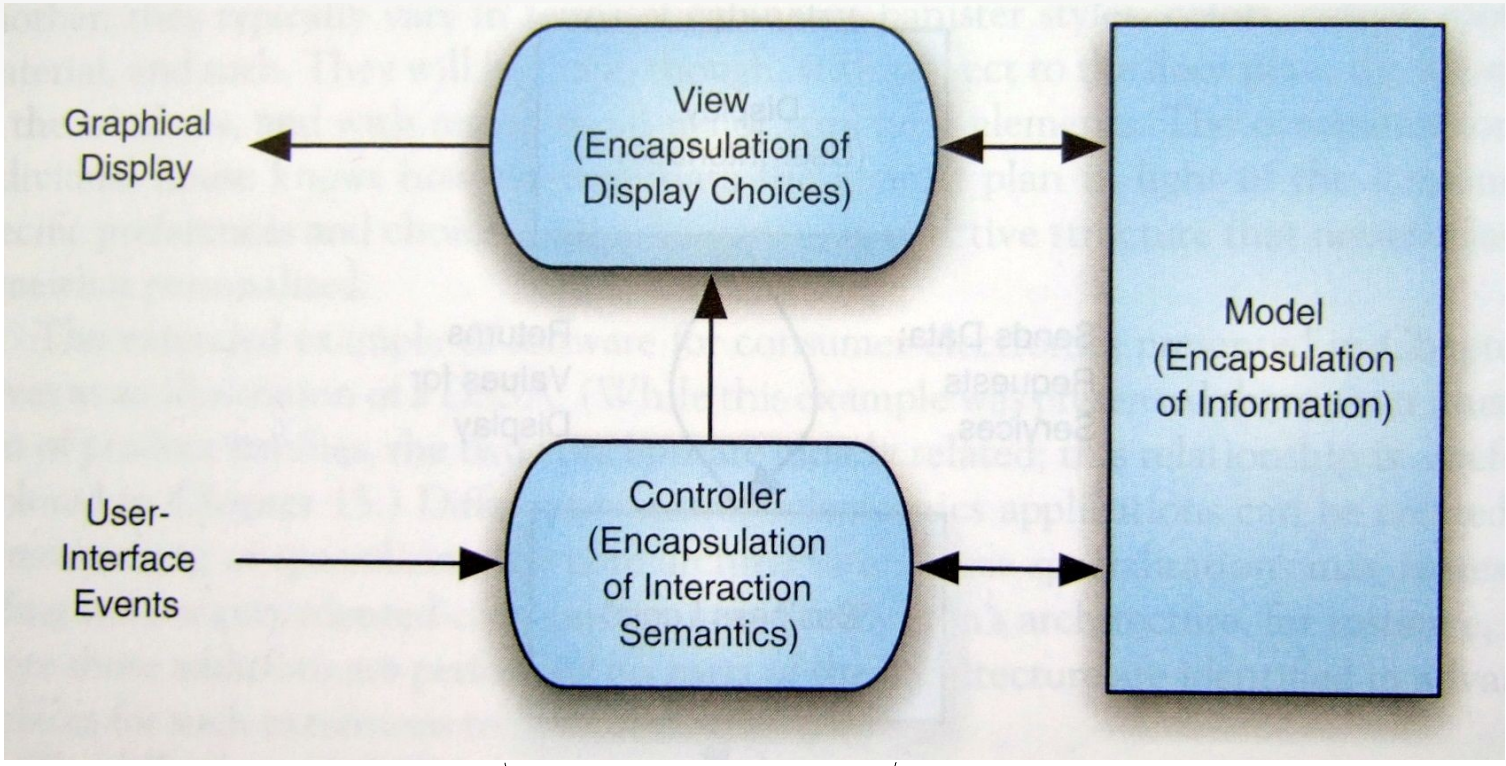
- CORBA
- Microsoft's OLE 2.x
- Modern integration platforms, e.g. Apache Camel or Mule ESB

Broker – Consequences

- Benefits:
  - Location transparency.
  - Changeability and extensibility of components.
  - Portability of a Broker system, which hides operating system and network system details from clients and servers by using indirection layers such as APIs, proxies and bridges.
  - Interoperability between different Broker systems.
  - Reusability.
- Liabilities:
  - Restricted efficiency.
  - Lower fault tolerance (compared with non-distributed software system)
  - Testing and debugging.

Model-View-Controller, MVC

- Pattern solving interaction (strictly separated input and output)
- [Buschmann et al: Pattern-Oriented Software Architecture, 1997]
- The devil is in the details [http://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/]
  - Should be the presentation logic included in View or Controller? Should be the core functionality of the application included in Controller or Model? How tight the connection between the View and Controller should be? Could View connect the Model directly?

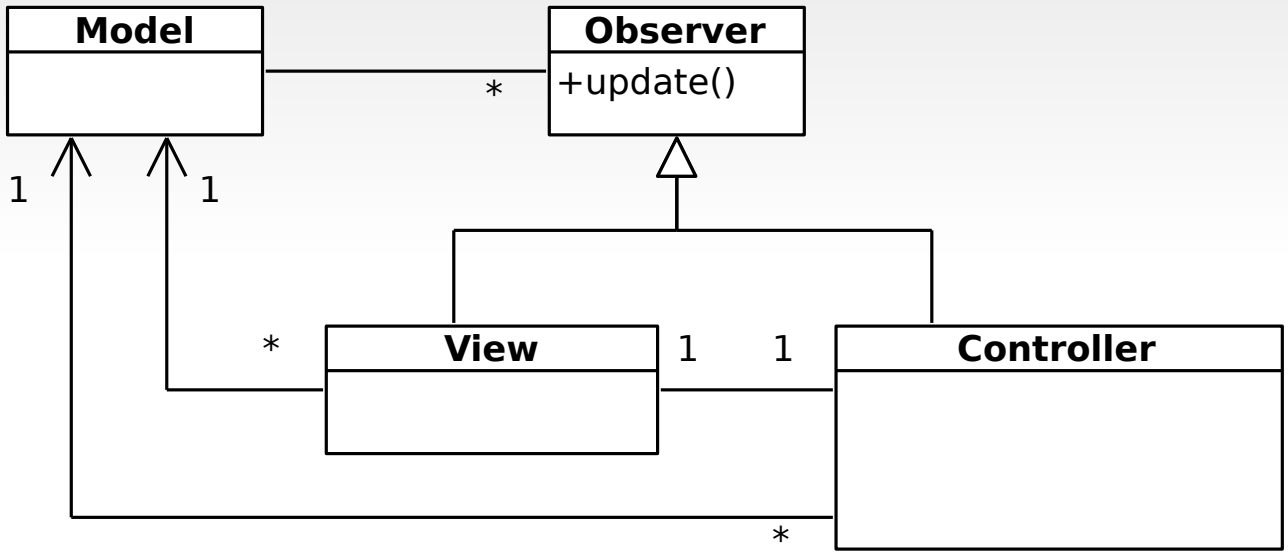


[Taylor et al. 2009]

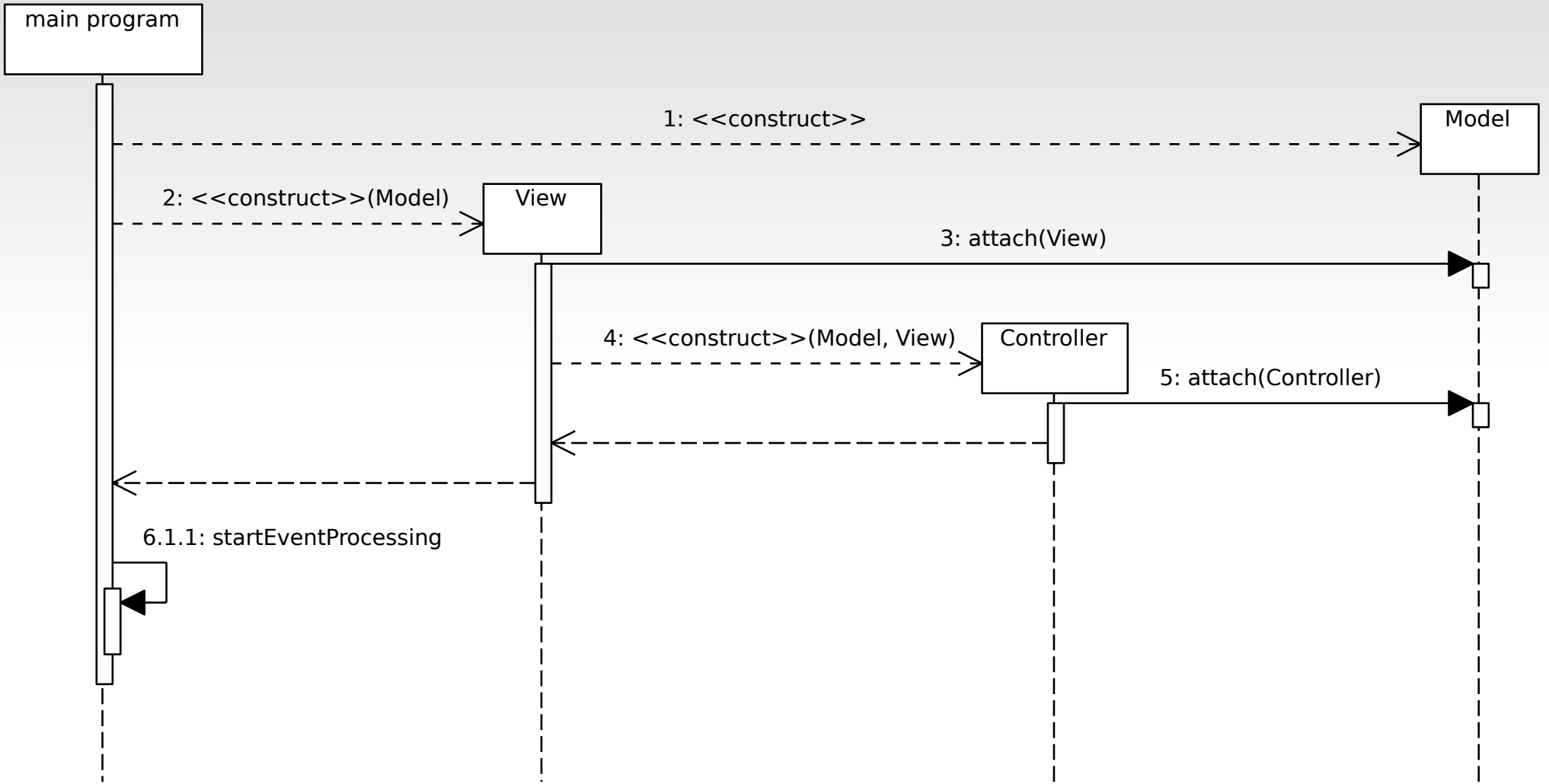
MVC – Structure

- Model:
  - Provides functional core of the application. Encapsulates the appropriate data.
  - Registers dependent view and controllers.
  - Notifies dependent components about data change.
- View:
  - Presents information to the user.
  - Different views can present the model in different ways.
  - Each view creates a suitable controller (1:1 relationship)
  - Views often offer functionality that allows controllers to manipulate the display.
- Controller:
  - Accepts user input as events. Events are translated into requests for model or the associated view.
  - If the behavior of the controller depends on the state of the model, the controller registers itself with the change-propagation mechanism and implements an update procedure.

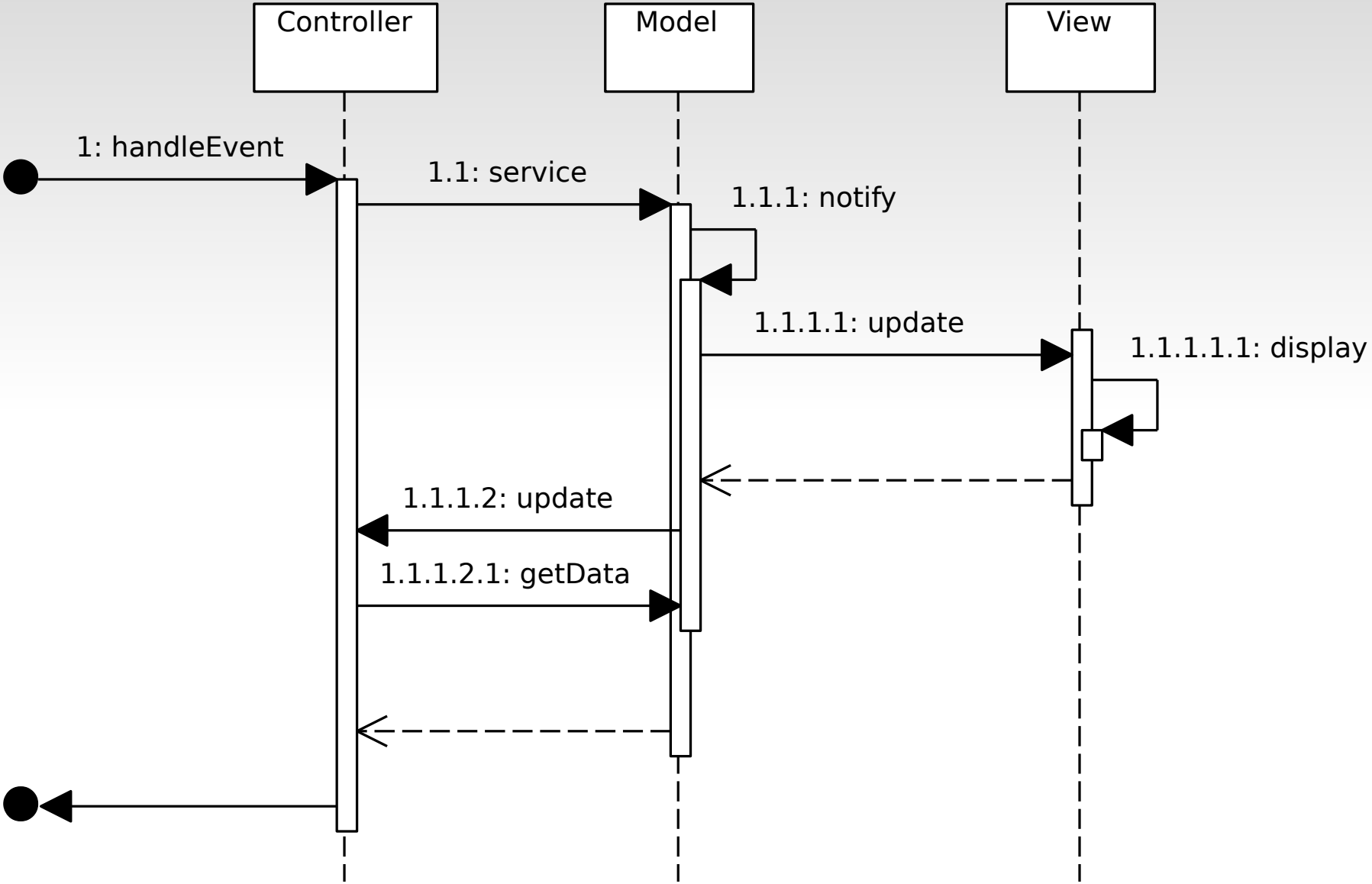
MVC – Structure (cont.)



MVC – Dynamics (initialization)



MVC – Dynamics (event)



MVC – Known Uses

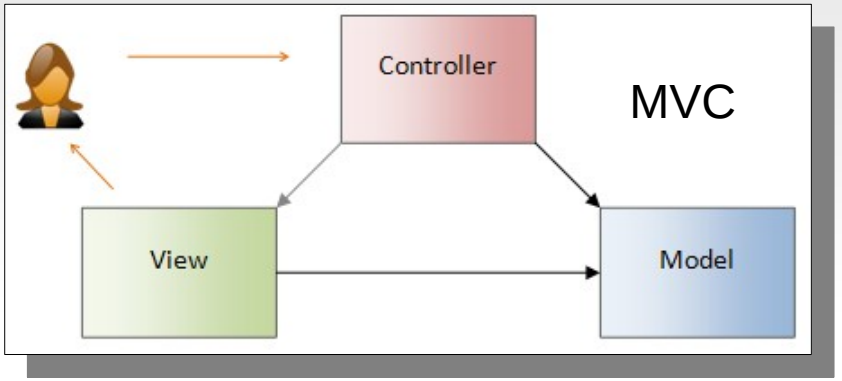
- Historical frameworks with terminals.
- WWW: Static HTML pages or pages with server-side scripting.
- Spring MVC, CakePHP, Zend and other MVC frameworks.

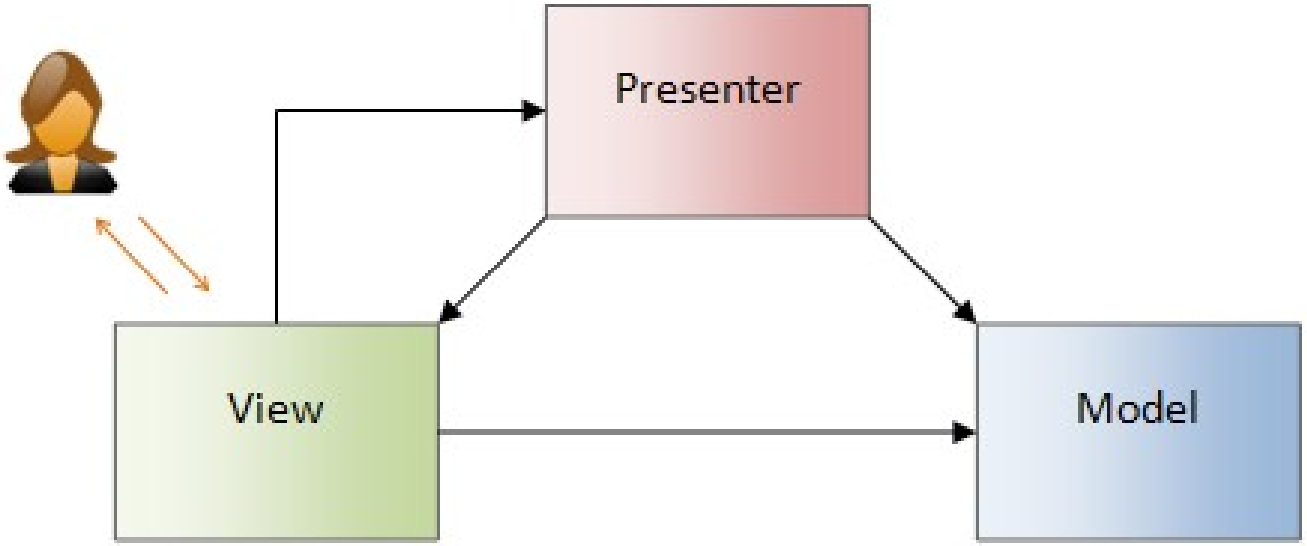
MVC – Consequences

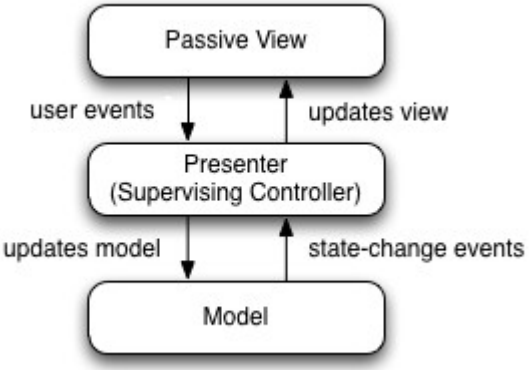
- Benefits:
  - Multiple view of the same model.
  - Synchronized views.
  - 'Pluggable' views and controllers.
  - Exchangeability of 'look and feel'.
- Liabilities:
  - Increase complexity.
  - Potential for excessive number of updates.
  - Intimate connection between view and controller.
  - Close coupling of views and controllers to a model (changes to the model's interface are likely to break the code of both view and controller).

Model-View-Presenter

MVP is a variation of MVC suitable for „widget“ applications (button is drown but also handles I/O), or for modern web applications which have the ability to process GUI events on client side (ASP.NET, Flex, JFace, AJAX, Swing, Google Web Toolkit, Nette, ...)

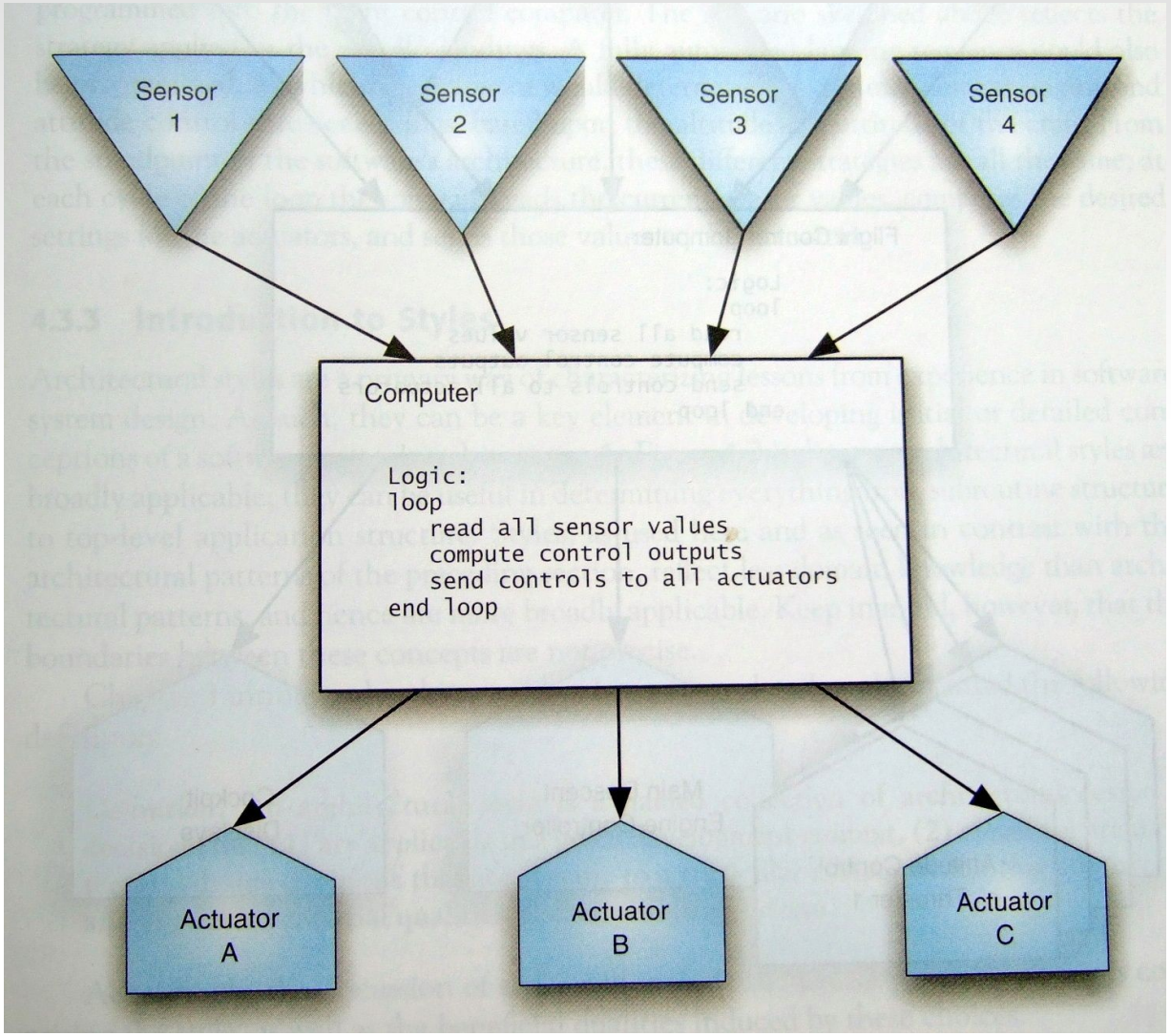






Another examples of architectural patterns (I)

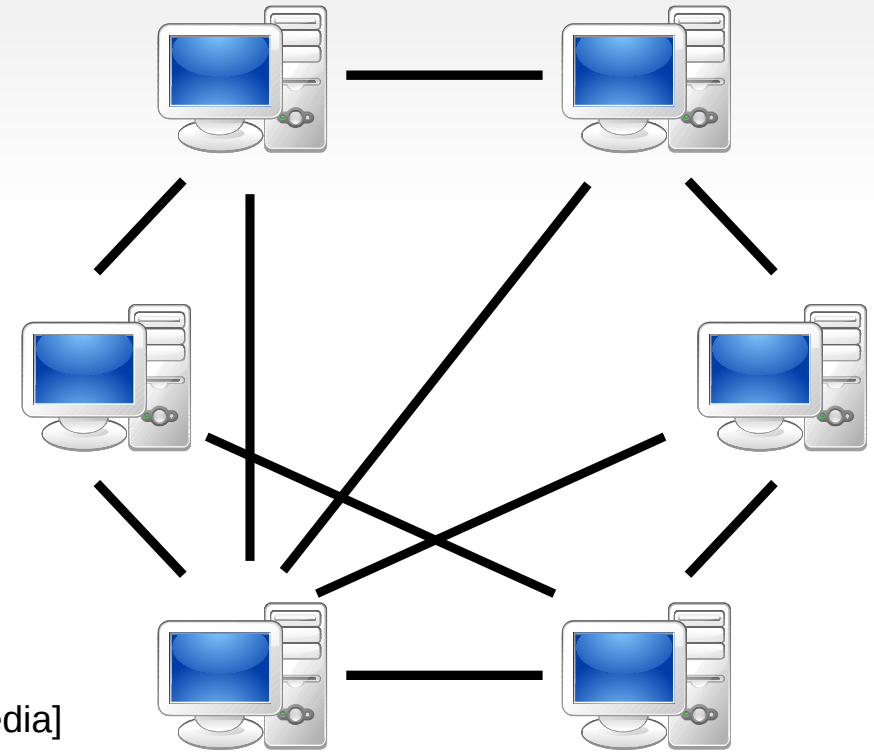
Sense-Compute-Control (embedded systems)



[Taylor et al. 2009]

Another examples of architectural patterns (III)

Peer-to-peer architecture is a distributed application architecture that partitions tasks or work loads between peers. Peers are equally privileged, equipotent participants in the application.



[wikipedia]

Architectural styles

Often confused with architectural patterns!

> **Architectural style** is named collection of architectural design decisions that (1) are applicable in given context of the development, (2) restricts architectural design decisions which are specific for concrete system within concrete context and (3) provide certain qualities in every proposed system. [Taylor et al.]

Simple examples:

From the point of view of programming language

- The main program and procedures/functions
- Object-oriented paradigm

From the communication point of view

- Procedure/method/service invocation
- Message delivery
- Publish-subscribe notification

Architectural patterns and styles

Design patters

- Generic solutions for problems on design and coding level

Architectural styles

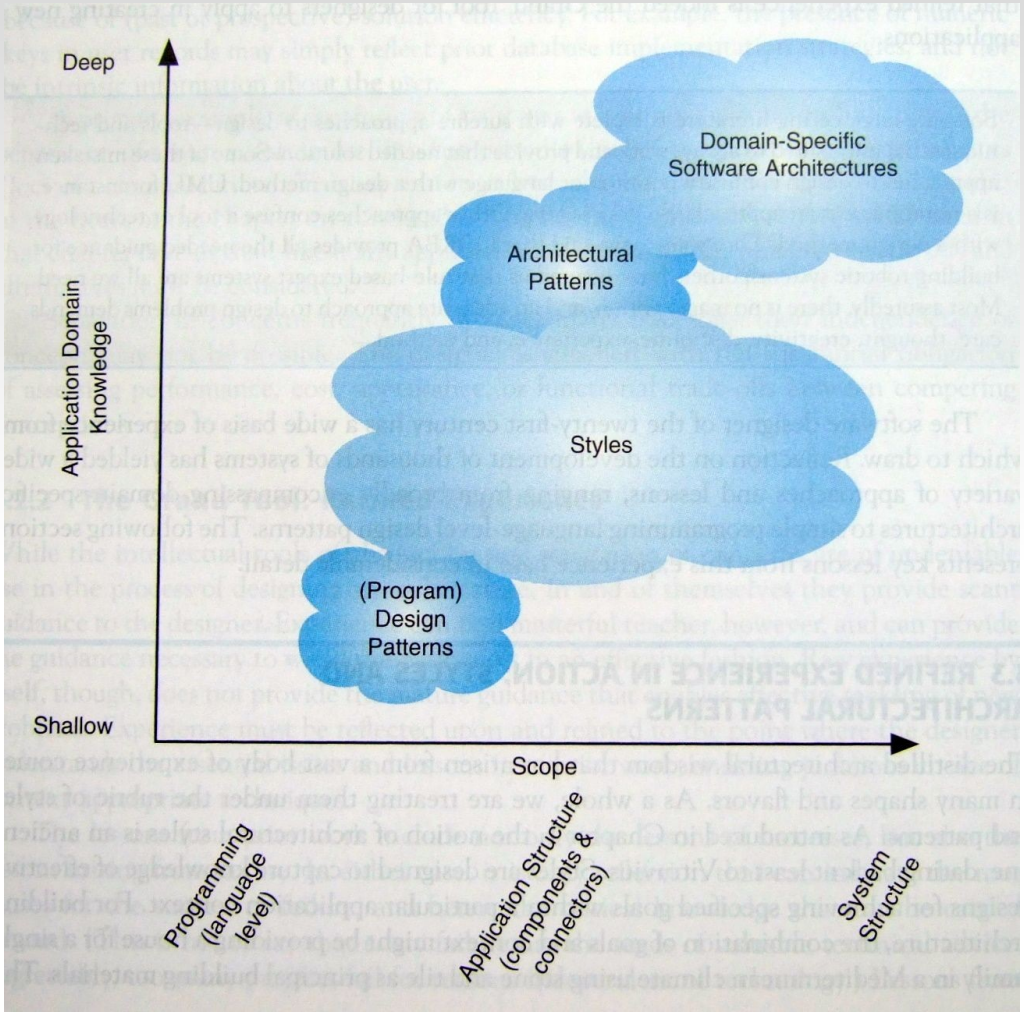
- Summarizing architectural principles affecting the code

Architectural patterns

- Generic solutions of architecture design

Domain-specific software architectures

- Design of complete structure of the application according to selected domain



[Taylor et al. 2009]

Questions?