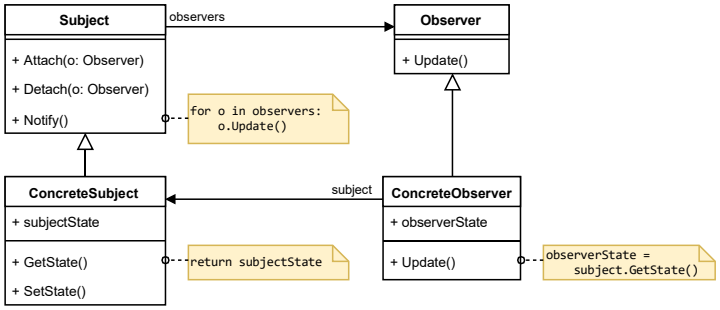
Design Patterns 448.058 (VO)

Michael Krisper
Georg Macher

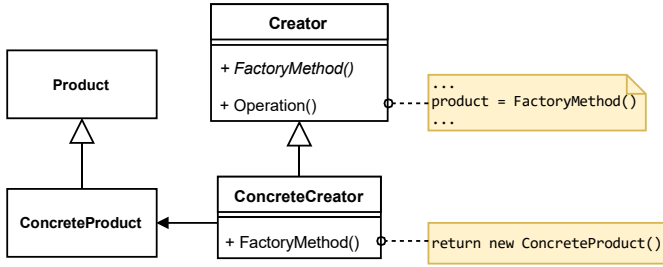
23.10.2018

Revision from last time…

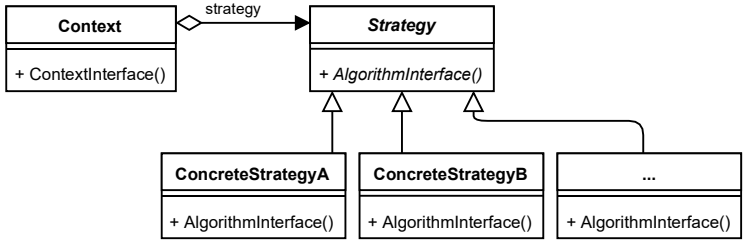
- OBSERVER
  Subjects notify registered observers.



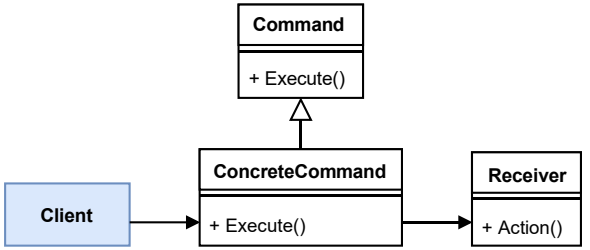
- FACTORY METHOD
  Delegate the creation of objects.



- STRATEGY
  Substitute a function/behaviour later.



- COMMAND
  Encapsulate a request in its executing context.



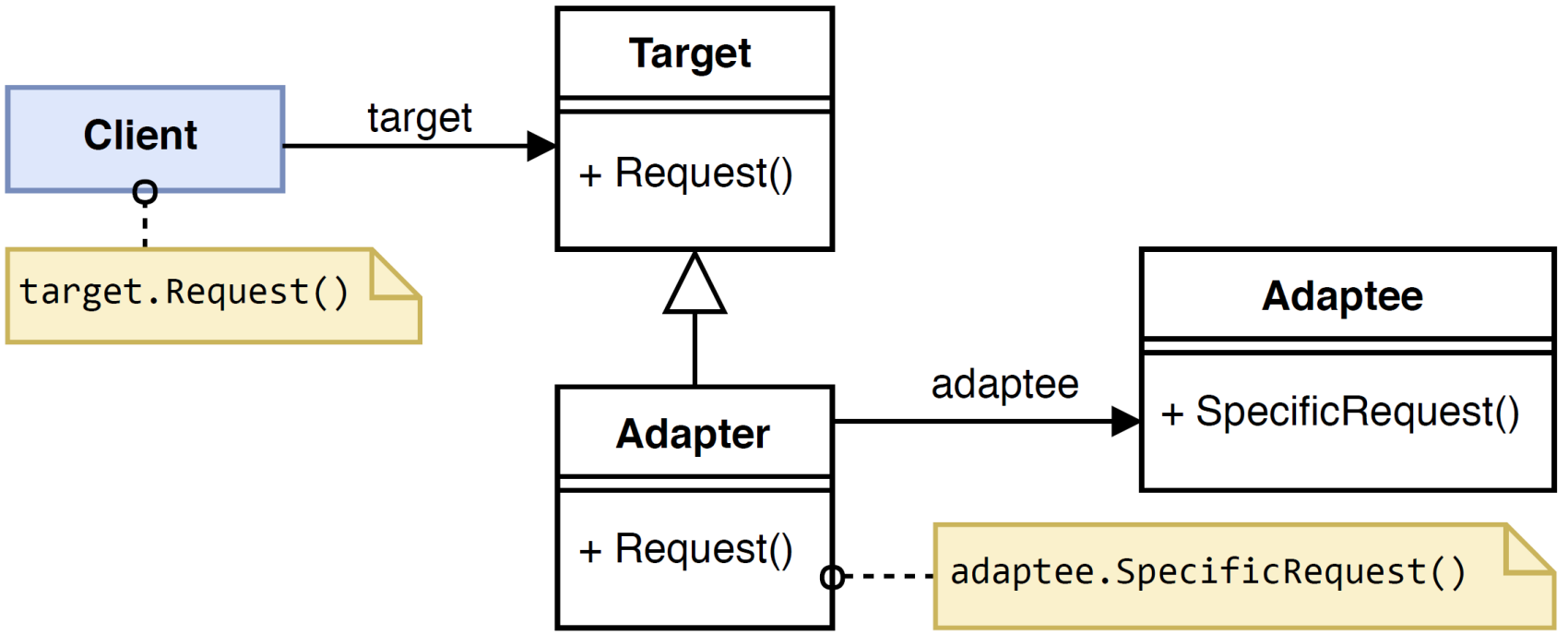
Learning Goals for Today

- Understanding and describing some wrapper design patterns:
  - Adapter
  - Wrapper Façade
  - Decorator
  - Proxy

- Understanding and describing Architecture Patterns:
  - Layers (Recap)
  - Pipes & Filters
  - Broker
  - Master-Slave
  - Client-Server
  - Leader-Follower
  - Model-View-Controller
  - Model-View-Presenter
  - Model-View-Viewmodel
  - Presentation-Abstraction-Control

Adapter

Wrap around a class to make it compatible to another interface.



Adapter

Context: Working with multiple different frameworks or libraries.

Problem: How to make incompatible classes work together?

Forces:

- Existing class interface does not match the one you need.
- You want to reuse the functionality (not just copy it).
- Source code of used class may not be available (copying or changing it is not possible)
- Class may be sealed (inheritance is not possible)

Solution:

- Create an Adapter class which wraps around the Adaptee.
  Variant: **Class Adapter** (inherits from Adaptee)
  Variant: **Object Adapter** (contains Adaptee member)
- Implement the desired new interface using the methods of the Adaptee as underlying basis.

Consequences: (Class Adapter)

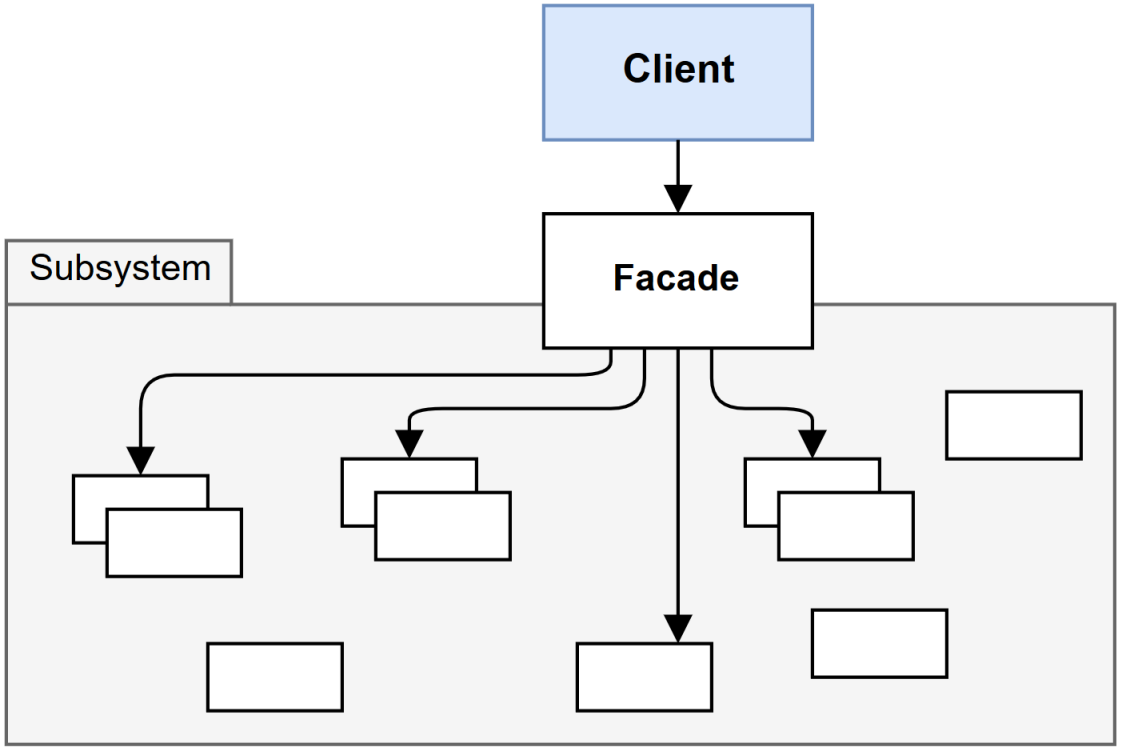
+ Allows to use override mechanisms (e.g. protected methods, V-table, access to protected members).
+ No additional indirection.
~ Inheritance approach (all methods of adaptee are inherited automatically, only changes have to be implemented)
- Won't work when we want to adapt a class and all its subclasses (liskov substitution!), because it is on a different branch of subclasses.

Consequences: (Object Adapter)

+ Works with base Adaptees and all subclasses (allows liskov substitution).
+ Adapter hides underlying type of Adaptee (breaks inheritance hierarchy, composition over inheritance!).
~ Explicit implementation approach (no methods inherited automatically, all needed methods have to be implemented explicitly)
- Adds additional layer of indirection.

Wrapper Façade

Encapsulate functions and data in a combined interface.



Wrapper Façade

Context: Working with a complex structure having many functions, maybe even with different programming paradigms (e.g. object-oriented vs. structured).

Problem:

- How to make it easier to use a complex system of functions, or to use functions of different programming paradigms in a more intuitive way?

Forces:

- Different programming paradigms have different ways of decomposition, approaches, and calling conventions.
- Developers are used to their own environments and conventions.
- Developing heterogenous paradigms makes programs more difficult to maintain.
- Concise and coherent code is more robust, easier to learn and maintain.
- Changing dependent software is often not possible (source code not available)
- Platform specific details should be hidden away.

Solution:

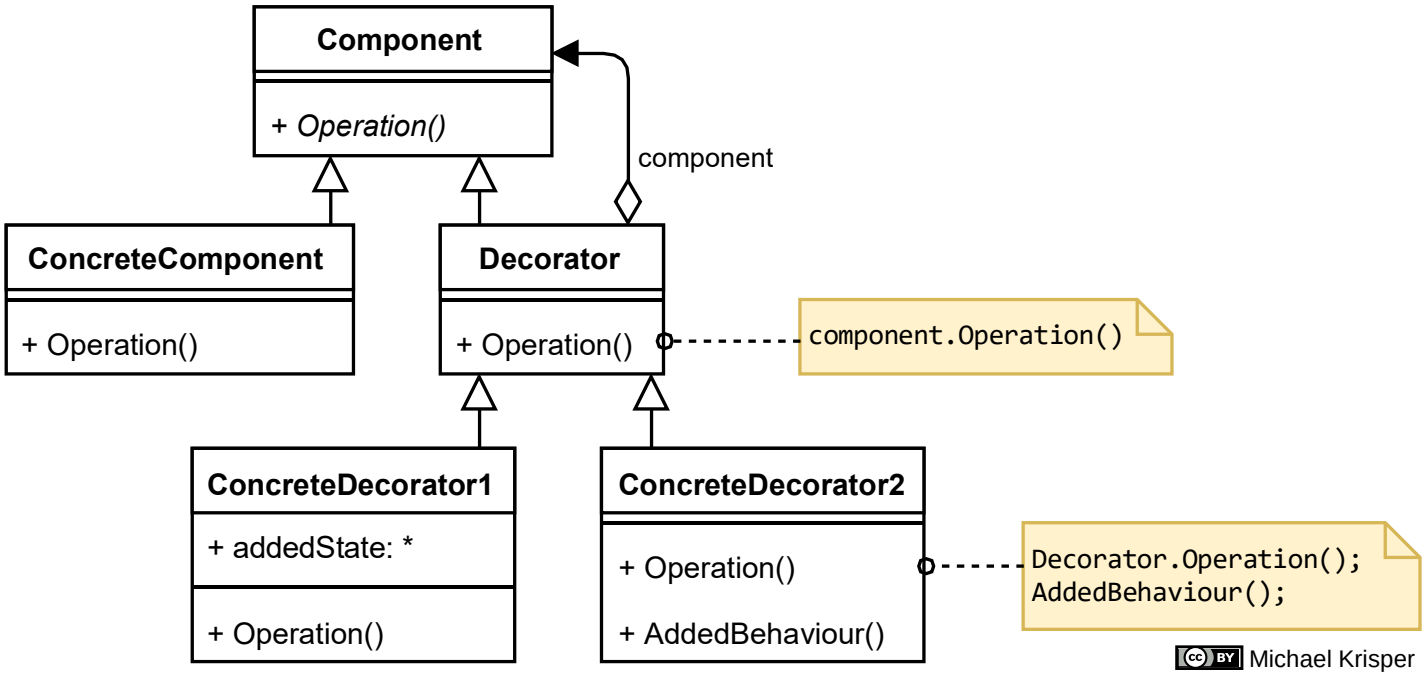
- Hide the complexities (implementation details) of the larger system and provide a simpler interface to the client.
- Encapsulate no-OO API data & functions within concise, robust, portable, maintainable, cohesive OO class interface.

Consequences:

+ Provides concise, cohesive and robust higher-level object-oriented programming interfaces.
+ Easier usability and maintainability.
- May diminish functionality and lose benefits of underlying paradigm
- Performance degradation by adding an additional layer of abstraction

Decorator

Extend the functionality of an object, while maintaining the same interface.



Decorator

Context: Functional extension of objects.

Problem: No arrangement last for long, we need to support adding or extending of functionalities.

There is nothing so stable as change – Bob Dylan

Forces:

- We want to add responsibilities to individual objects dynamically and transparently, without affecting other objects.
- We want to be able to withdraw responsibilities.
- The extension by subclassing is impractical:
  - large number of independent possible extensions.
  - hidden class definition or otherwise unavailable for subclassing

Solution:

- Define a Decorator which forwards requests to its Component object.
- The decorator may optionally perform additional operations before and after forwarding the request.

Consequences:

\+ More flexibility by adding responsibilities

\+ Flexibility responsibilities can be added and removed also at runtime

\+ Decorators also make it easy to add a property twice

\+ Avoids feature-laden classes high up in the hierarchy

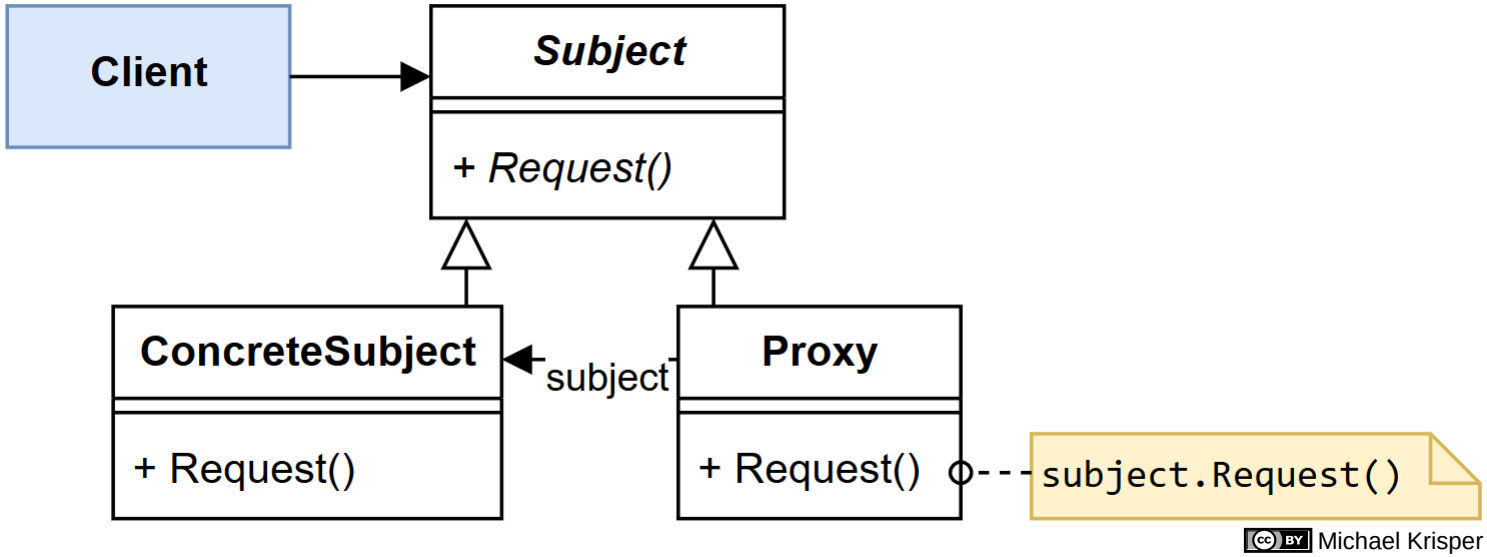
\+ Avoids the class explosion issue

\- Decorator and its component are not identically

\- Can be hard to learn and debug(lots of little objects only different in the way of their interconnection)

Proxy

Forward requests to a concrete subject.



Proxy

Context: Need for versatile references to objects.

Problem: How to provide means for access control for another object?

Forces:

- An object is in a **different address space** (remote proxy).
- An expensive object needs to be **created on demand** (virtual proxy).
- The **access** to the original object must be **supervised** (access rights! – protection proxy).
- A **smart reference** is needed as a replacement for a bare pointer that performs additional actions when an object is accessed.

Solution:

- Maintain **a reference** that lets the proxy **access the real subject and provide interface identical** to Subject
- **Control access to the real subject** (may also include creating and deleting) and **act like the real subject**.

Consequences:

+ Introduces a level of indirection when accessing an object (separation of housekeeping and functionality)
+ Remote Proxy decouples client and server
+ Virtual Proxy can perform hidden optimizations
+ Caching Proxy could reuse subjects
+ Security Proxy can control access
- Overkill via sophisticated strategies
- Less efficiency due to indirection

Learning Goals for Today

- Understanding and describing some wrapper design patterns:
  - Adapter
  - Wrapper Façade
  - Decorator
  - Proxy
- Understanding and describing Architecture Patterns:
  - Layers (Recap)
  - Pipes & Filters
  - Broker
  - Master-Slave
  - Client-Server
  - Leader-Follower
  - Model-View-Controller
  - Model-View-Presenter
  - Model-View-Viewmodel
  - Presentation-Abstraction-Control

Learning Goals for Today

- Understanding and describing some wrapper design patterns:
  - Adapter
  - Wrapper Façade
  - Decorator
  - Proxy

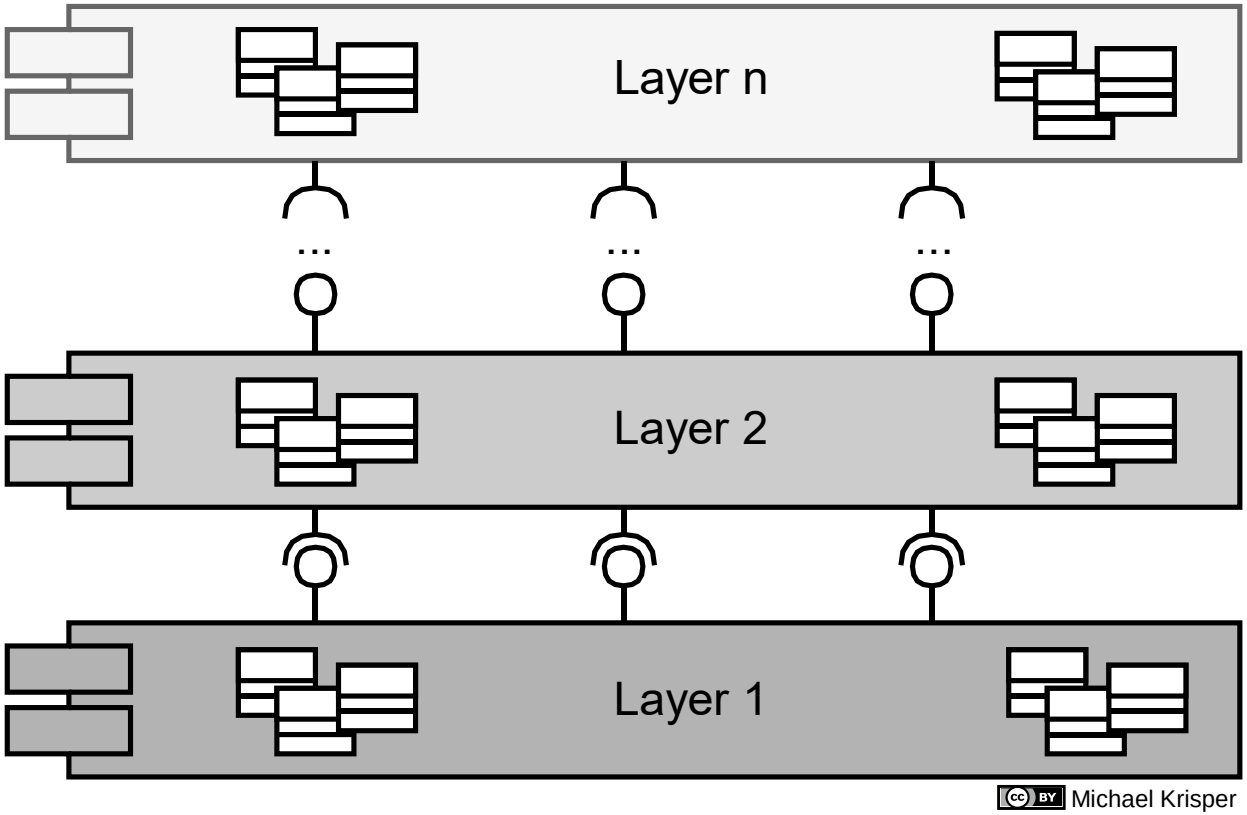
- Understanding and describing Architecture Patterns:
  - Layers (Recap)
  - Pipes & Filters
  - Broker
  - Master-Slave
  - Client-Server
  - Leader-Follower
  - Model-View-Controller
  - Model-View-Presenter
  - Model-View-Viewmodel
  - Presentation-Abstraction-Control

Architectural Patterns

- How are responsibilities distributed in a system?
- Who communicates with whom?
- Relations & Dependencies between Objects

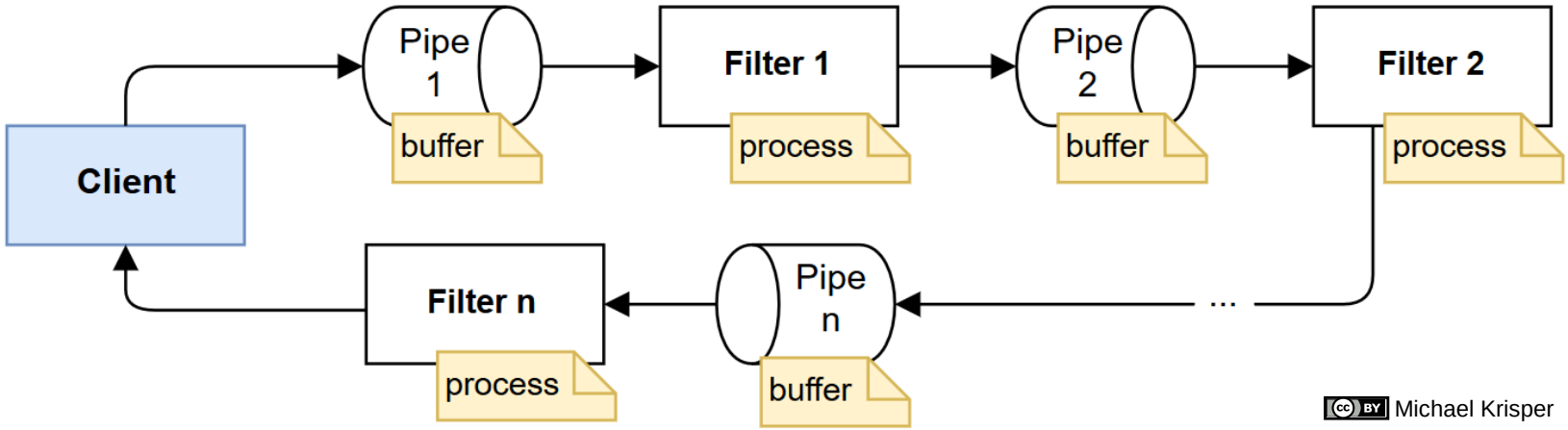
Layers

Split your system into layers based on abstraction levels



Pipes & Filters

Form a sequence of processing steps using a common interface.



Pipes & Filters

Context: Processing of data streams.

Problem: How to can data streams be decomposed into several processing stages.

Forces:

- **Exchanging or reordering of processing steps** shall be possible (future system enhancements).
- Small processing steps are **easier to reuse** than larger.
- Probably different sources of input data exist (file, network, sensor,..)
- Results shall be storable in different ways.
- Explicit **storage of interim steps** shall be possible.
- **Multiprocessing** shall be enabled.

Solution:

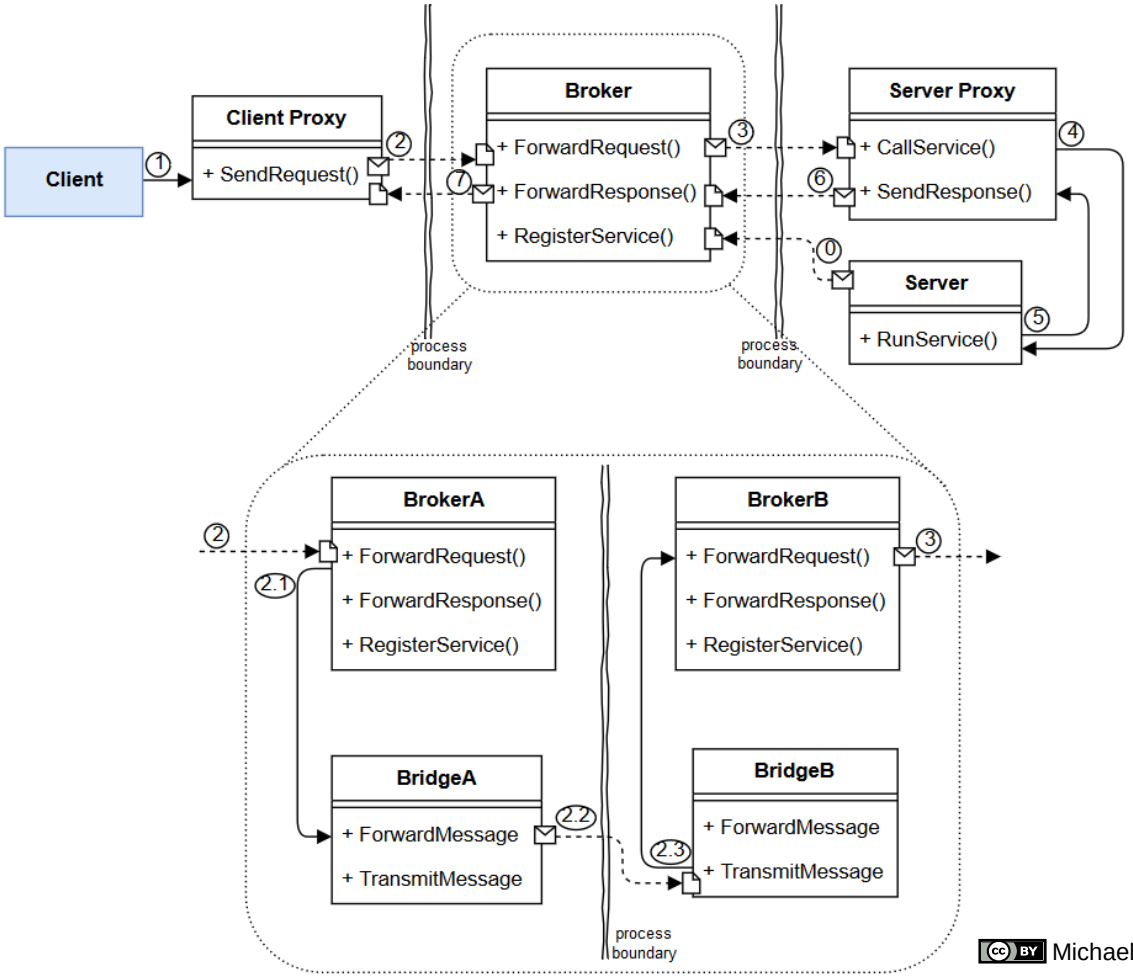
- **Divide** System task **into a sequence** of processing steps (dependent only on output of predecessor and connected by the dataflow)
- **Define a data format** to be passed along each pipe.
- Implement each pipe connection **either push or pull**
- Filter design and implementation
- Design **Error handling** (min. error detection)
- Setup processing pipeline

Consequences:

+ Intermediate files possible
+ Flexible via filter exchange
+ Flexible via recombination
+ Efficient for parallel processing
- Sharing state infos is expensive
- Data transformation overhead
- Error handling is crucial

Broker

Manage dynamic communication between clients and servers in distributed systems.



Broker

Context: Distributed / heterogeneous systems with independent cooperating components.

Problem: You want to build complex SW systems as a set of decoupled and interoperating components

Forces:

- Remote method invocation shall be supported
- The architecture shall support location transparency
- The addition, exchange, or removal of services shall be supported dynamically
- System details shall be omitted for developer

Solution:

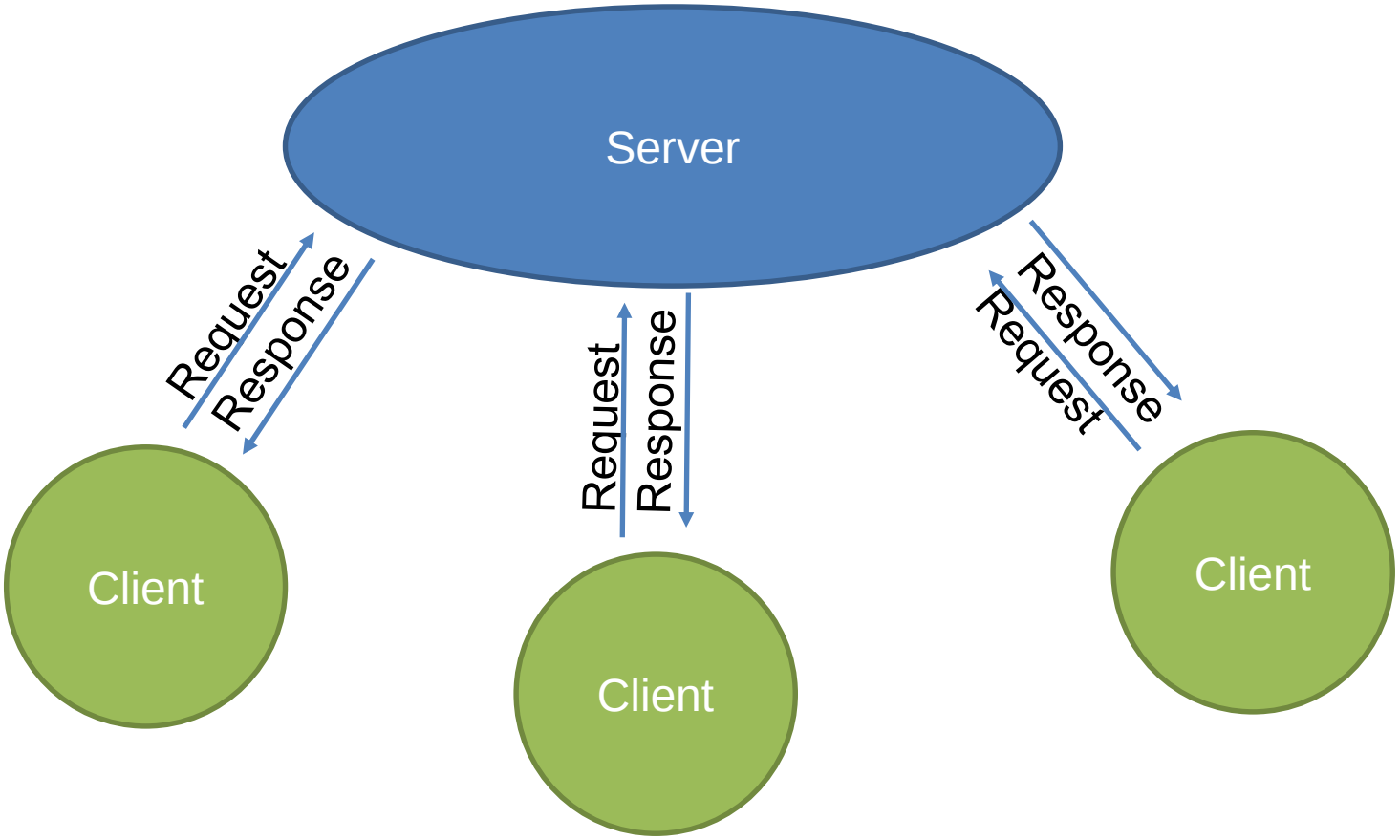
- **Define an object model**, or use an existing model (use e.g. CORBA, OLE/COM/.NET…)
- Decide upon component **operability**
- **Specify broker API** (client side and server side)
- Use proxy object to hide implementation details
- Design the broker component (protocol between client & server-side proxies, between brokers, consider failures in comp and communication)

Consequences:

+ Broker is **responsible for locating a server** (location transparency)
+ **Changeability** & extensibility of components (due proxies & bridges)
+ Broker hides OS& network details (**portability**)
+ **Interoperability between different broker**
+ Reusabilitiy of components
- Restricted efficiency (communication overhead)
- **Lower fault tolerance** (server/client may fail independently)
- **Hard to test & debug** (many components involved)

Client-Server

Clients send requests to central server which answers with responses.



Client-Server

Context: Distributed application.

Problem: You want to cooperate (share resources, content or service function) with multiple distributed clients.

Forces:

- Availability of services (resources, functions,..) is limited, but required by multiple requesters.
- Service might be provided by only one dedicated provider (centralized system).
- Simpler clients might be required
- Number of possible service-requester might be unknown.

Solution:

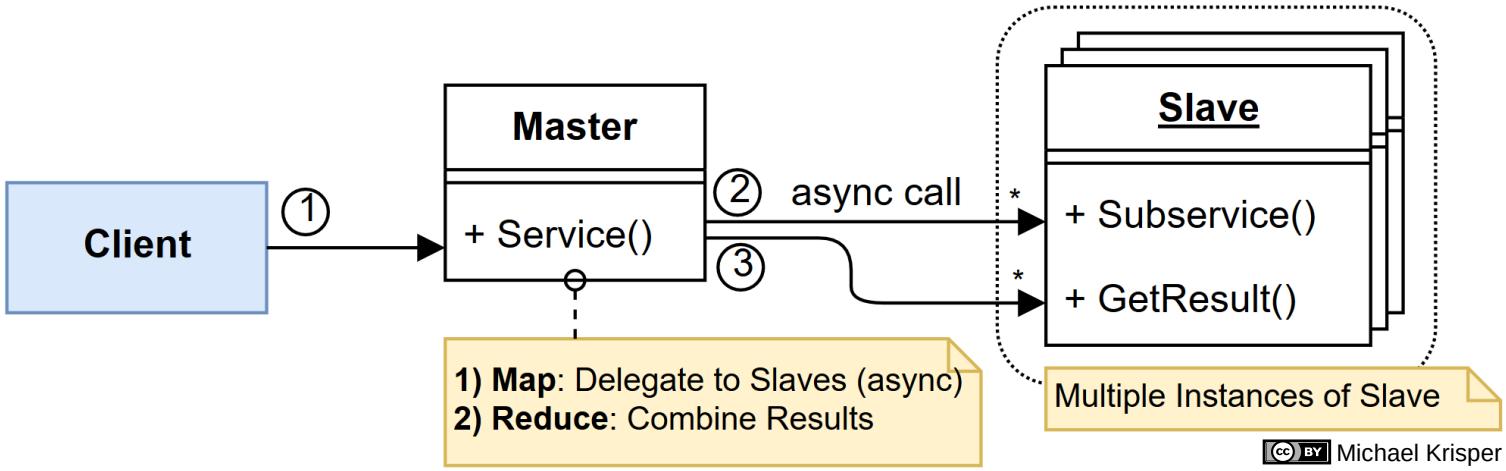
- Service-Interface: Define a protocol for serving a request/response communication.
- Server-Side Implementation: Implement a Listener which waits for requests from potentially multiple clients and individually answers with responses.
- Client-Side Implementation: Implement a Client which sends requests and waits for responses.

Consequences:

+ Encourages Service-Oriented Architectures
+ Centralization of specific services
+ Services get available for many clients
+ Doesn’t need to know exact number of clients
+ Workload gets moved to server. Clients are free to do something else
+ Exchangeability and extensibility
- Server could get overloaded
- Single-Point-Of-Failure, Denial-Of-Service Attacks are possible
- Communication overhead

Master-Slave

A master distributes work amongst some helpers.



Master - Slave

Context: Partitioning of work into semantically-identical sub-tasks.

Problem: You want to solve instances of the same problem, **partition identical work** and separate concerns.

Forces:

- Processing of sub-tasks should not depend on algorithms for partitioning work and assembling the result
- Sub-tasks might **need coordination**
- Many **instances of the same problem** must be **solved**
- **Different algorithm implementation** may be required
- **Multi-threaded applications** may be wanted

Solution:

- Introduce a **coordination instance** between clients of the service and the processing of individual sub-tasks
- The master component divides work into equal sub-tasks, distributes these sub-tasks to Slave components & combines results (**maintaining slaves**)
- Provide all slaves with a common interface. The clients will **only communicate with the Master**

Consequences:

\+ **Exchangeability** and extensibility

\+ **Separation of concerns**

\+ Fault tolerance – several replicated implementations can detect and handle failures

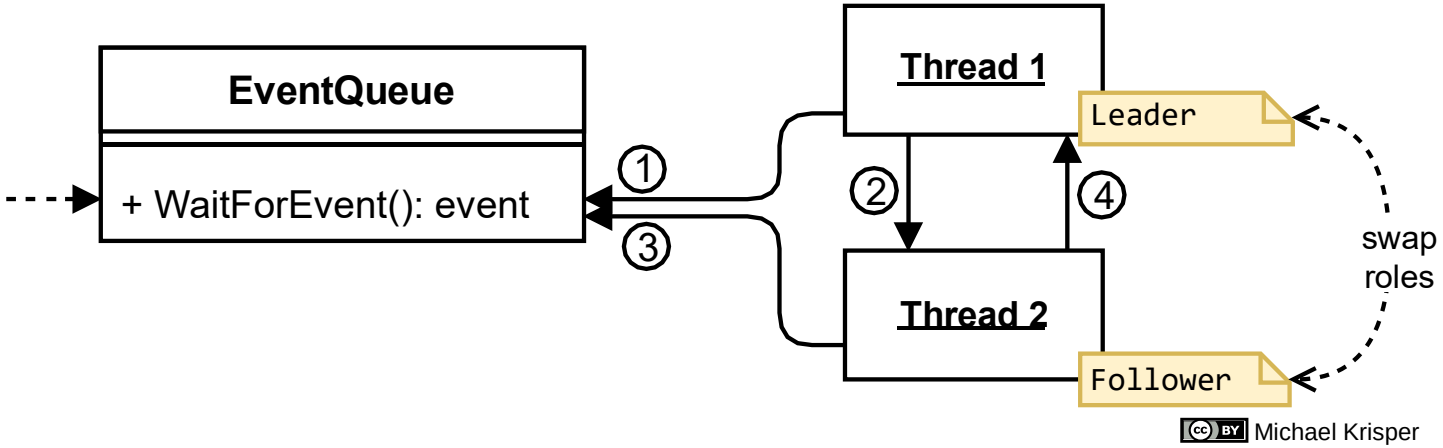
\+ **Efficiency** (support of parallel computation)

\- Not always feasible

\- **Partitioning & control can be tricky**

Leader-Follower

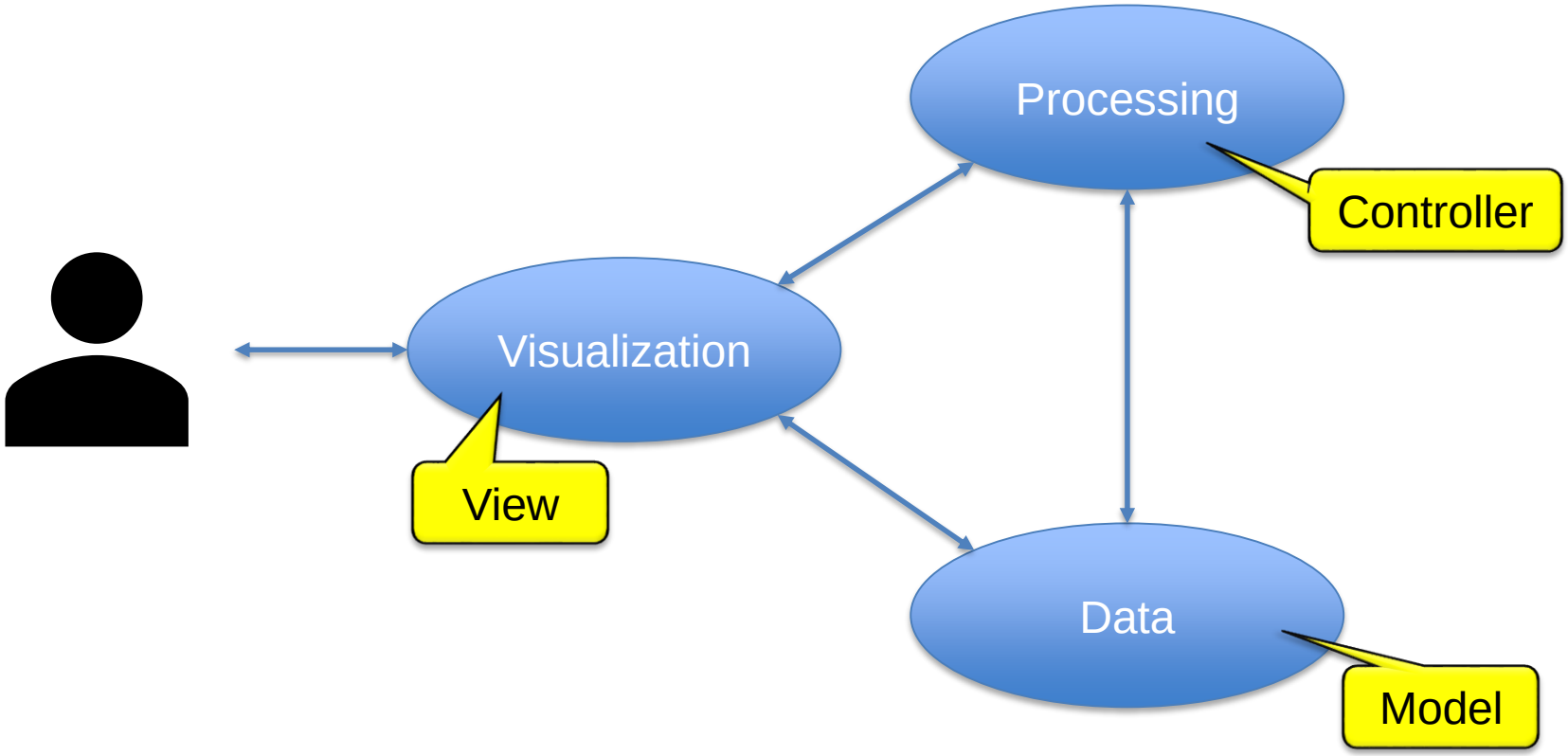
Multiple executors take turns in processing tasks, always switching the leader role.



Examples: ThreadPool, Background Workers

Model-View-Controller (MVC) / Model-View-Presenter (MVP) / Model-View-Viewmodel (MVVM)

Separate the responsibilities of visualizing, processing and data management for GUI applications.



Summary of Today

- Wrapper design patterns:
  - Adapter
  - Wrapper Façade
  - Decorator
  - Proxy
- Architecture Patterns:
  - Layers (Recap)
  - Pipes & Filters
  - Broker
  - Client-Server
  - Master-Slave
  - Leader/Follower
  - MVC / MVP / MVVM / PAC