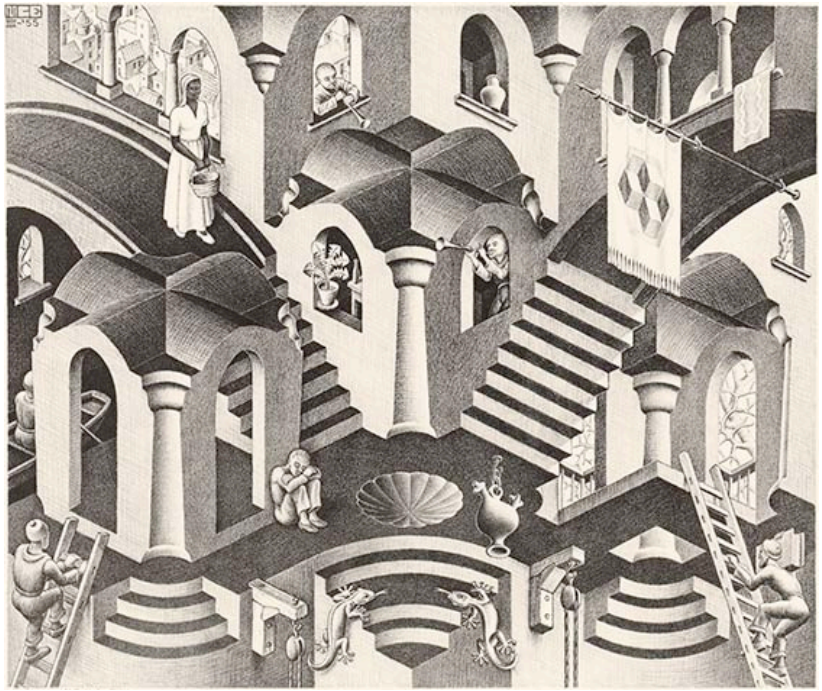


Software Architectures

Part 4: Architectural Patterns

Content

What are you about to learn?

Knowledge

- Can explain the architectural pattern ”layers”
- Can explain the architectural pattern ”pipes and filters”
- Can explain the architectural pattern ”plug-in”
- Can explain the architectural pattern ”broker”
- Can explain the architectural pattern ”service-oriented architecture”
- Can explain the architectural pattern ”model-view-controller”

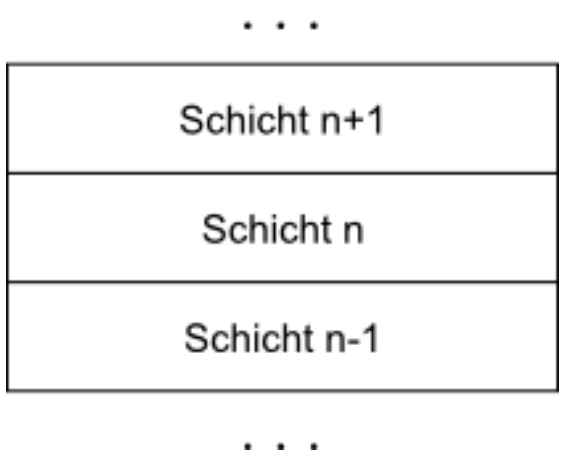
Layers

Problem

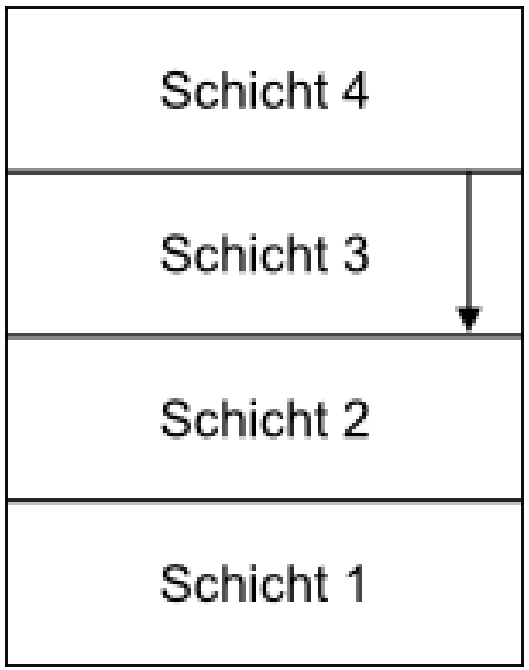
- Partition complex systems such as to minimize dependencies between their parts
- Structure the system horizontally such that a higher layer acts as a client that is being served by a lower layer server
- Lower level servers may act as client to their lower layers
- Each tier represents some abstraction

Solution

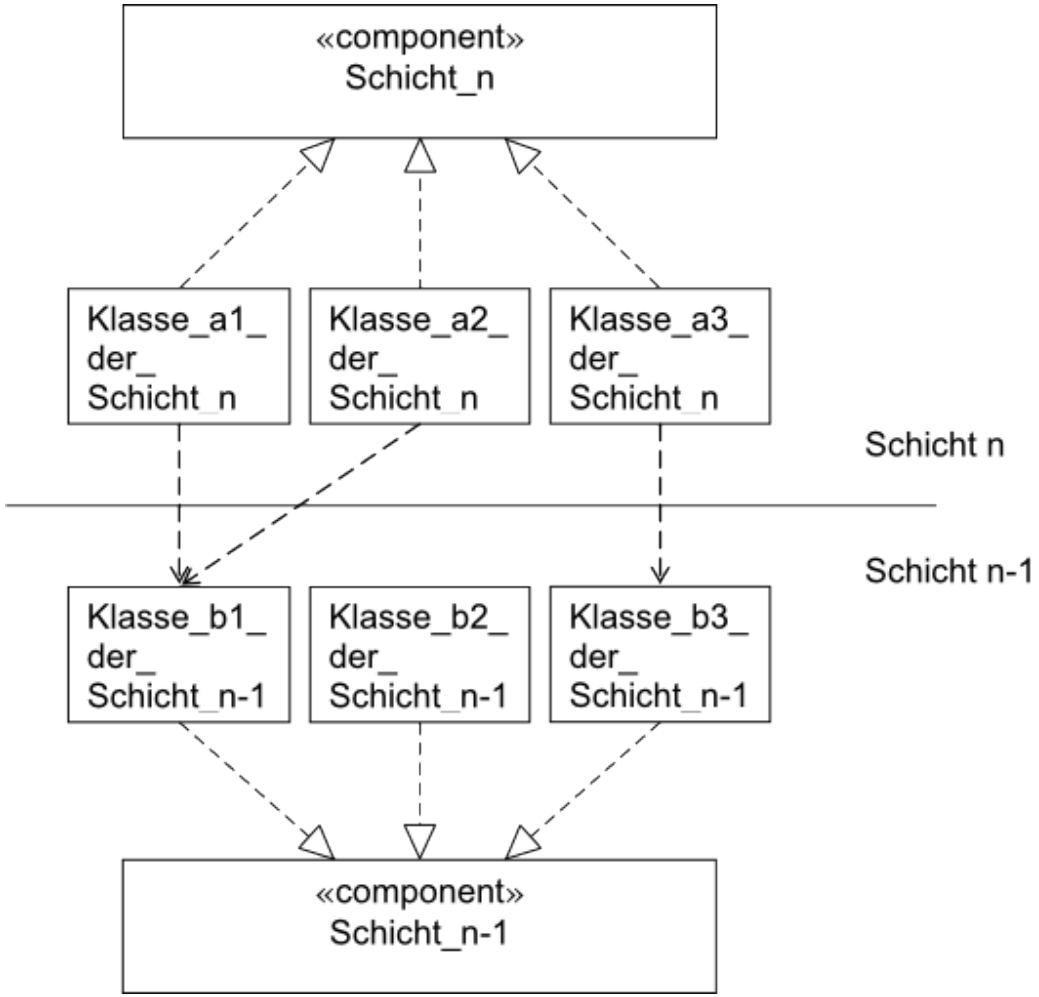
- Components are arranged in layers (tiers)
- Each tier serves a dedicated purpose, e.g. communication, data management
- Client-server model: lower layers offer services to higher layers

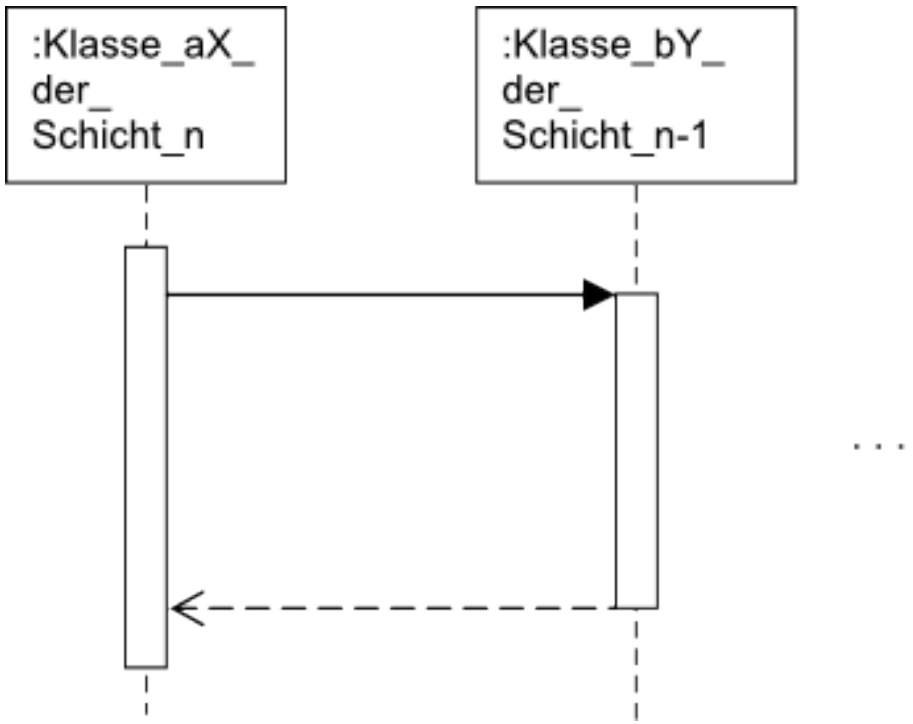


- Higher layers can call services of a lower layer
- Lower layers must not call services of a higher layer
- With layer bridging: higher layers can call services several layers lower
- Strict layering: Higher layer just call services from layer directly below

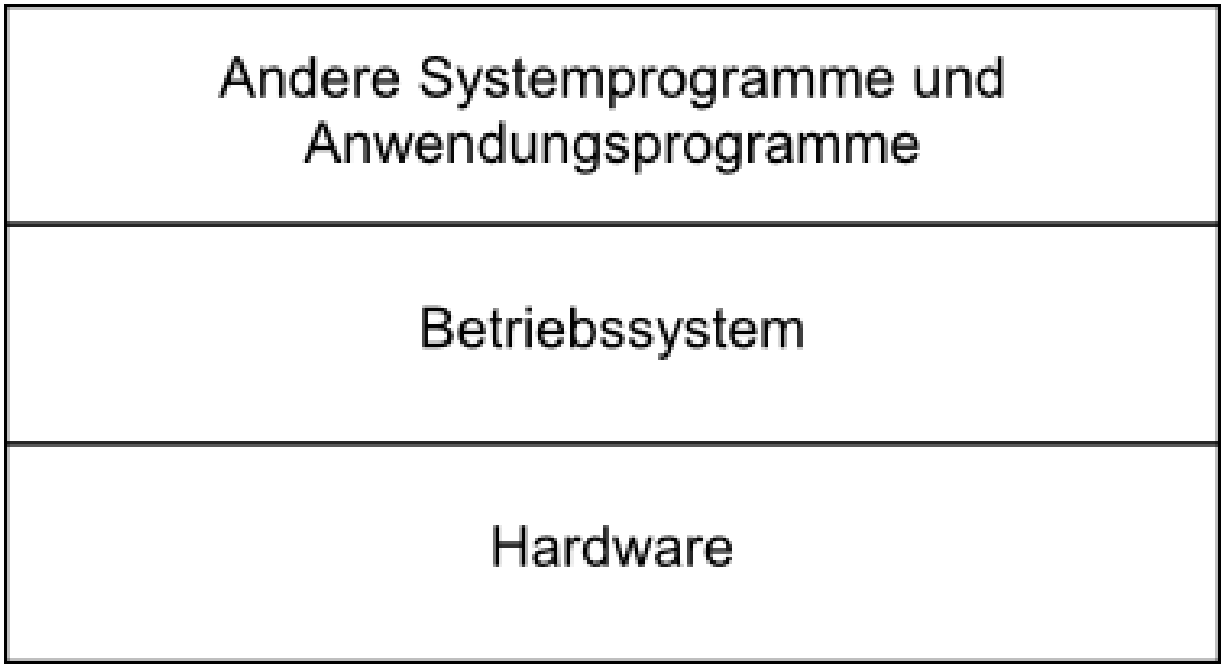


The Layers Pattern
Class Diagram



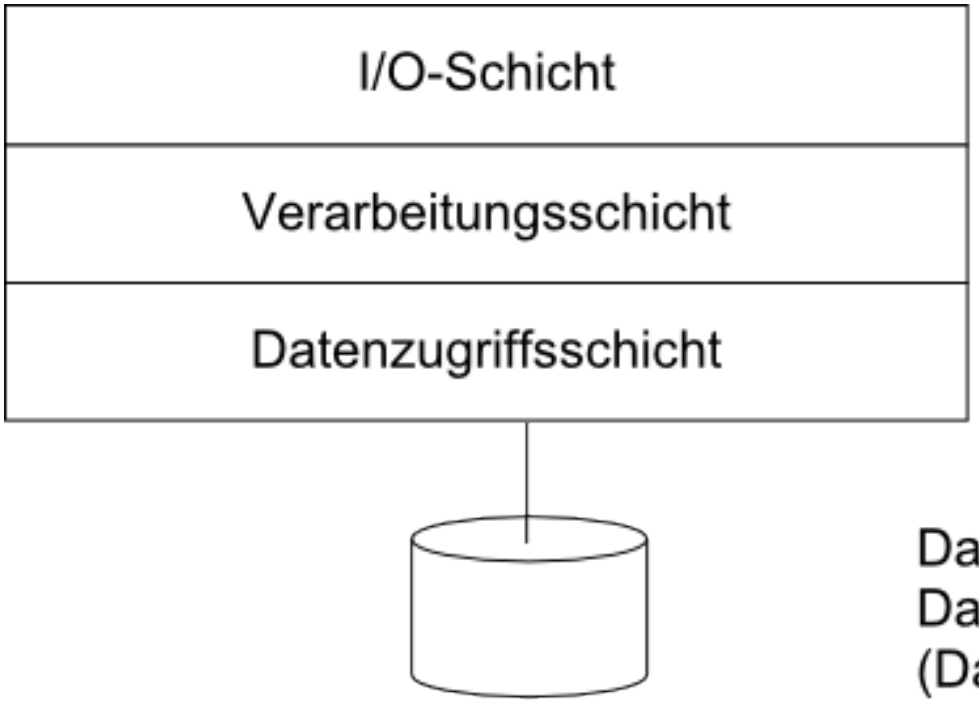


The Layers Pattern
Tier Structure of a Computer



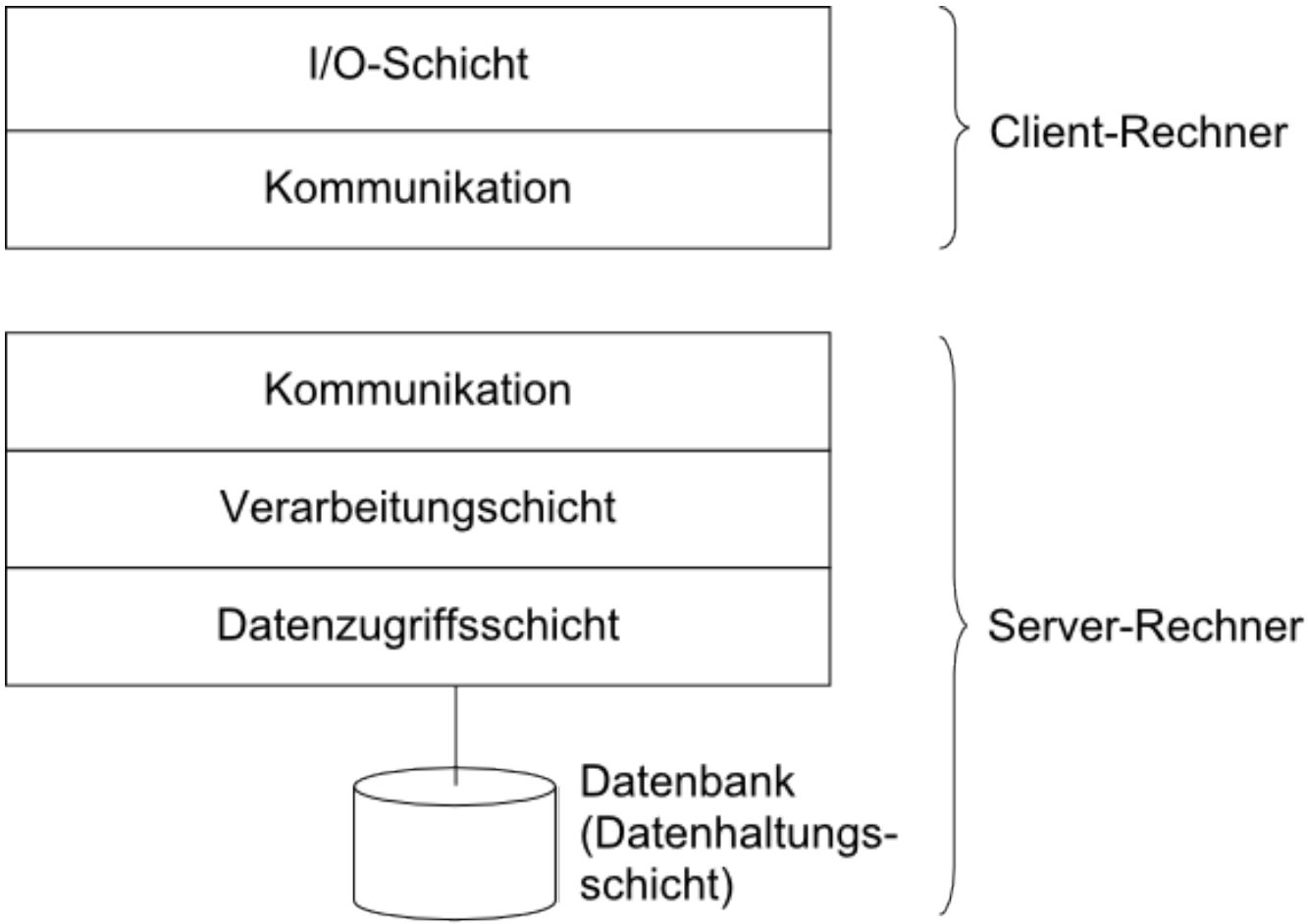
Examples: Torque, Hibernate, EclipseLink

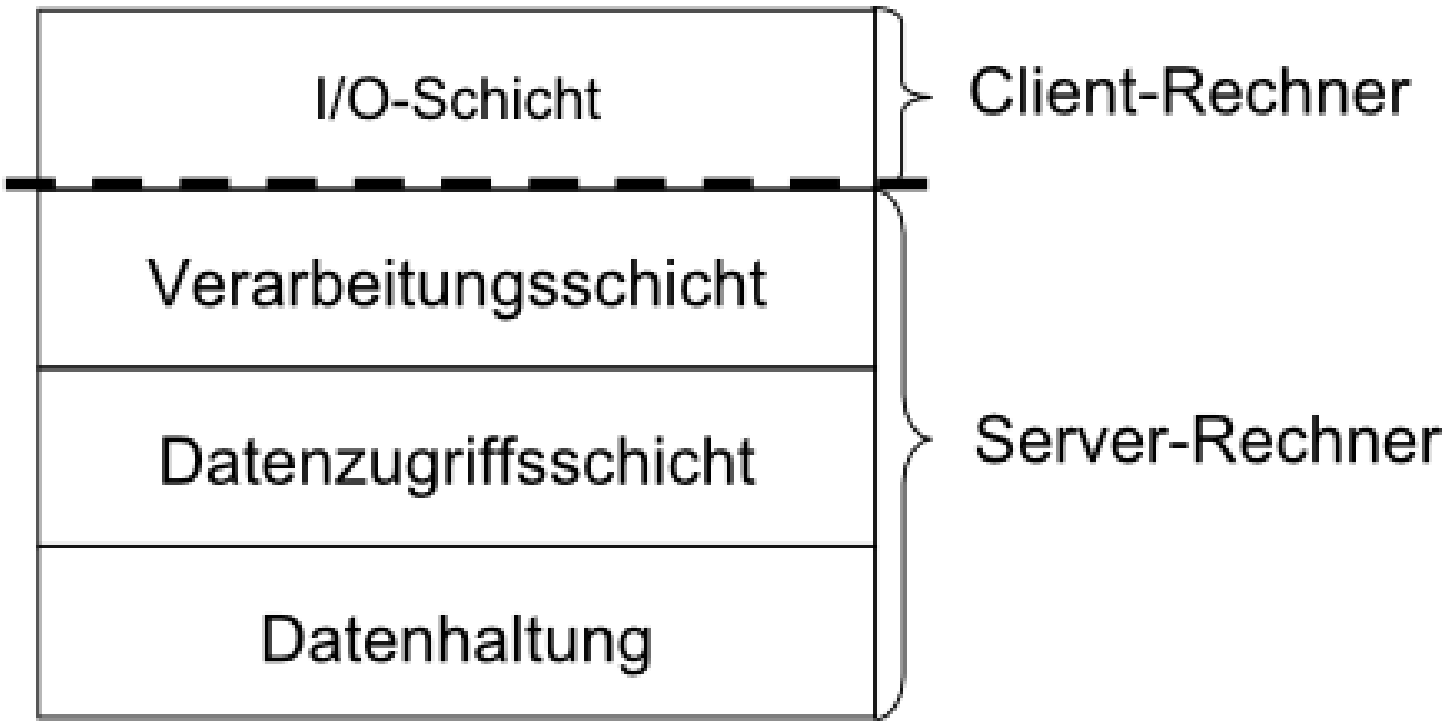
The Layers Pattern
Single Computer System

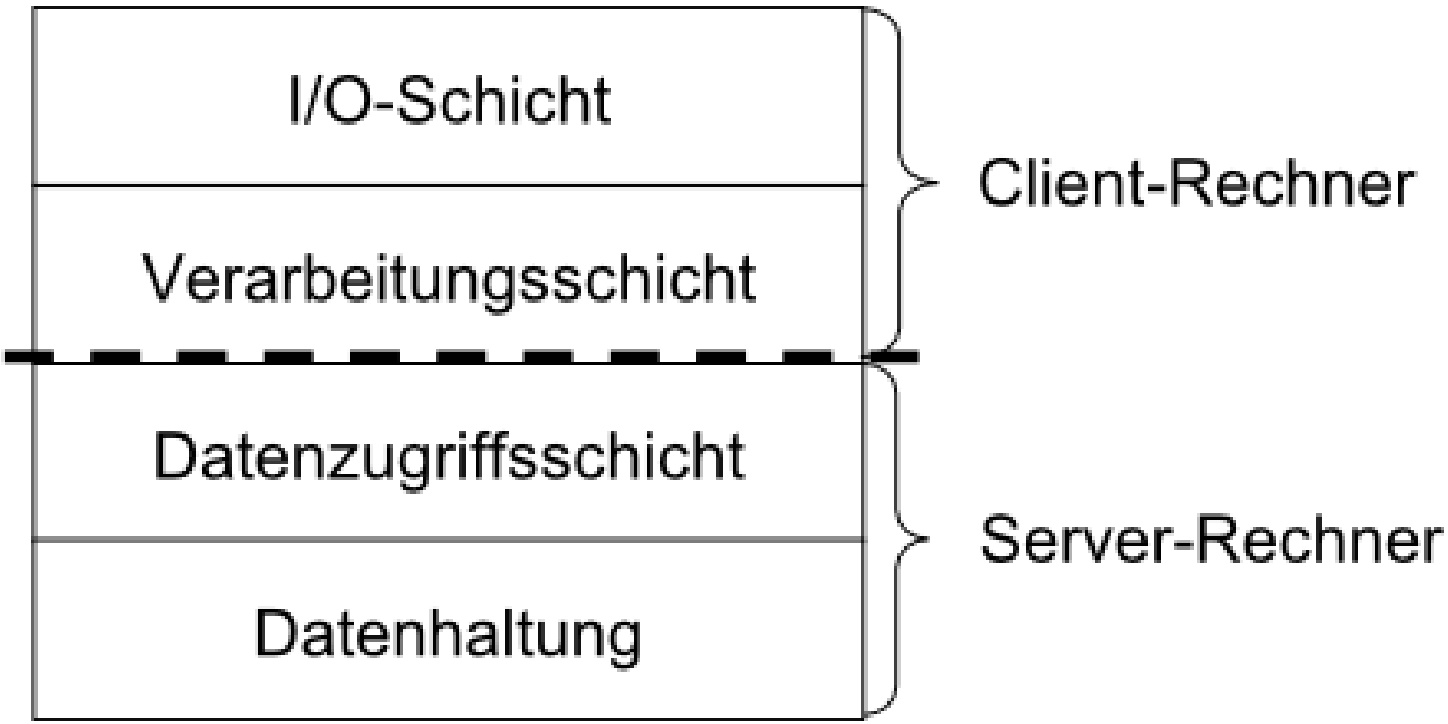


Datenbank zur persistenten Datenhaltung (Datenhaltungsschicht)

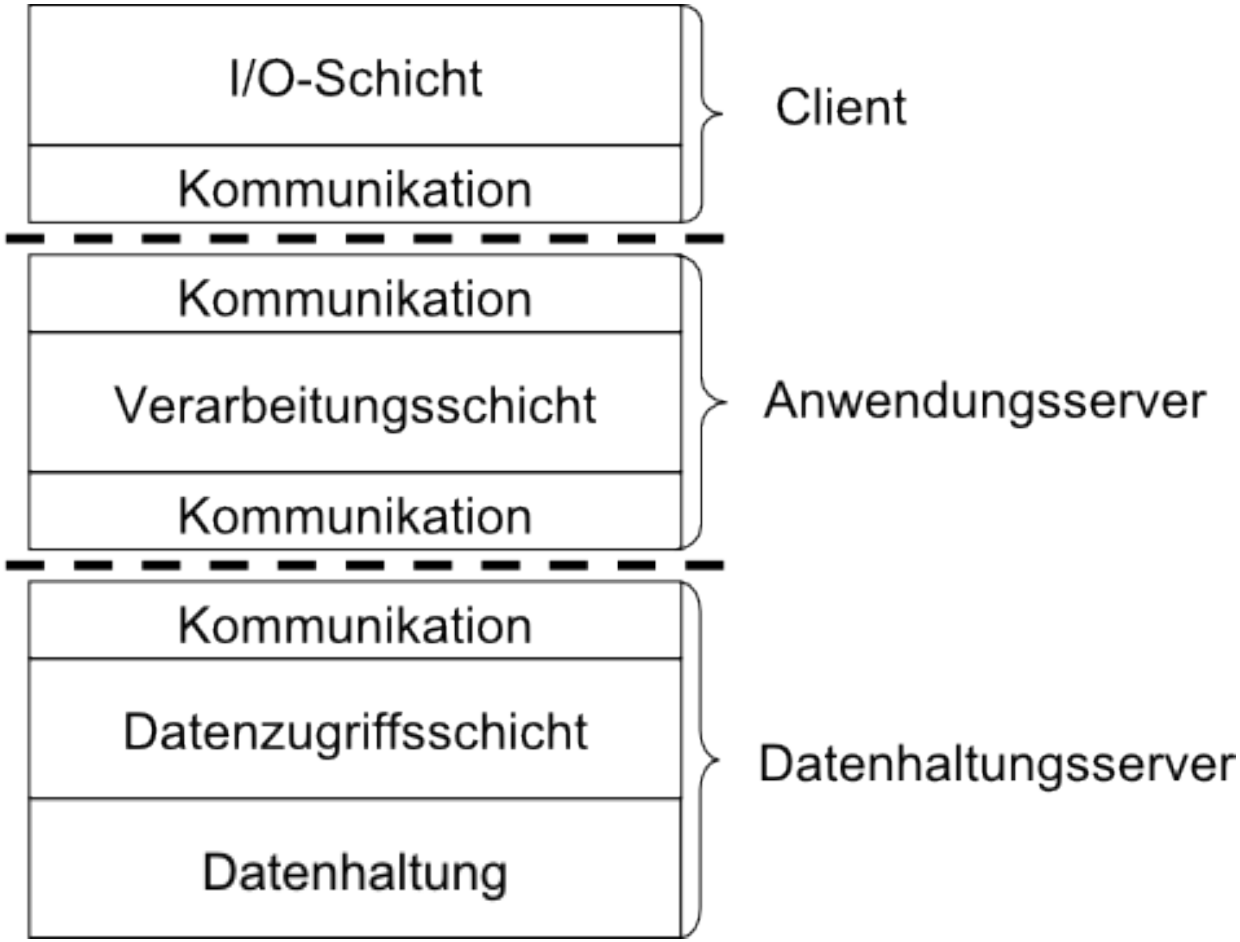
The Layers Pattern
Client-Server Tier Model



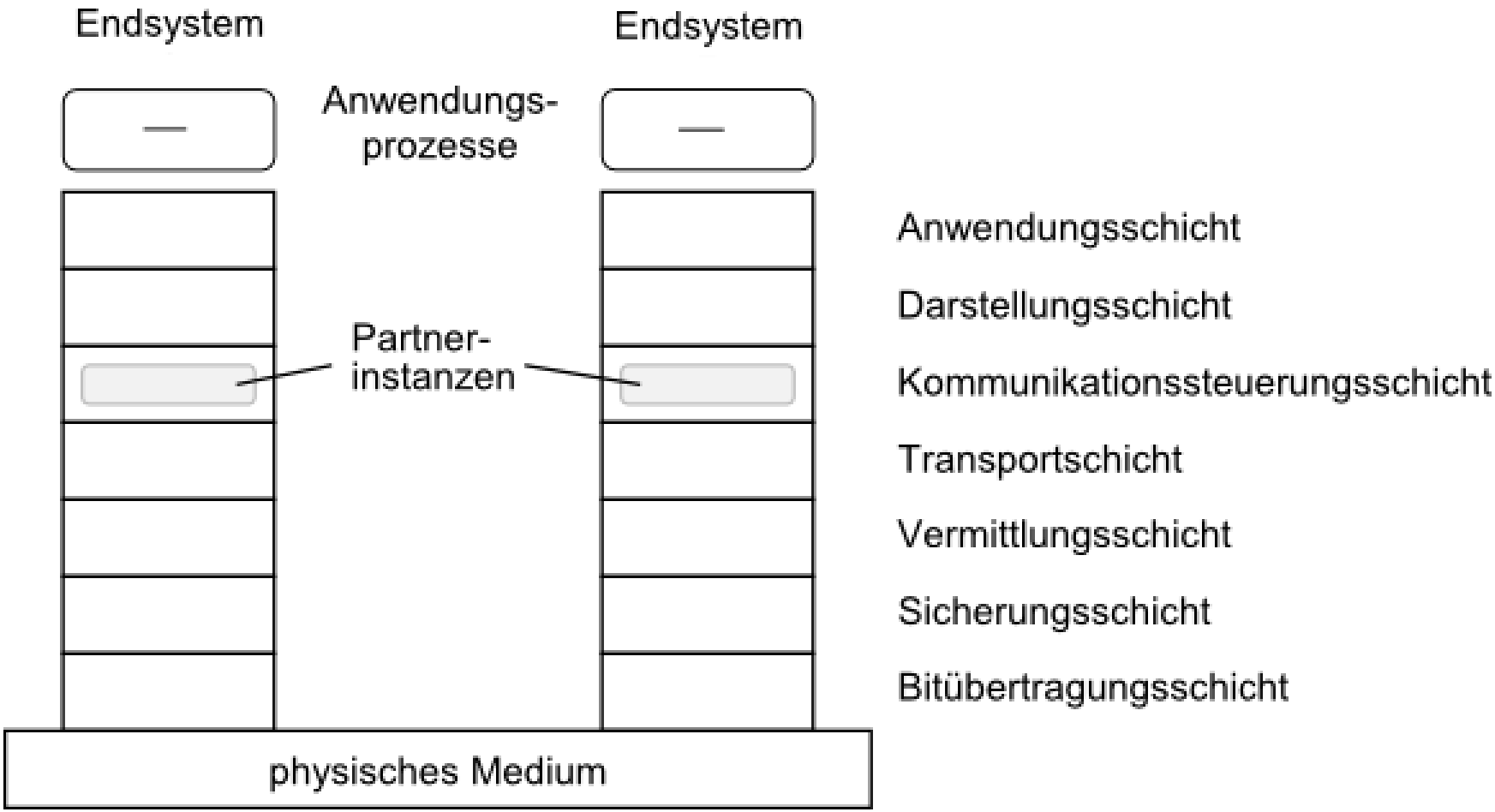




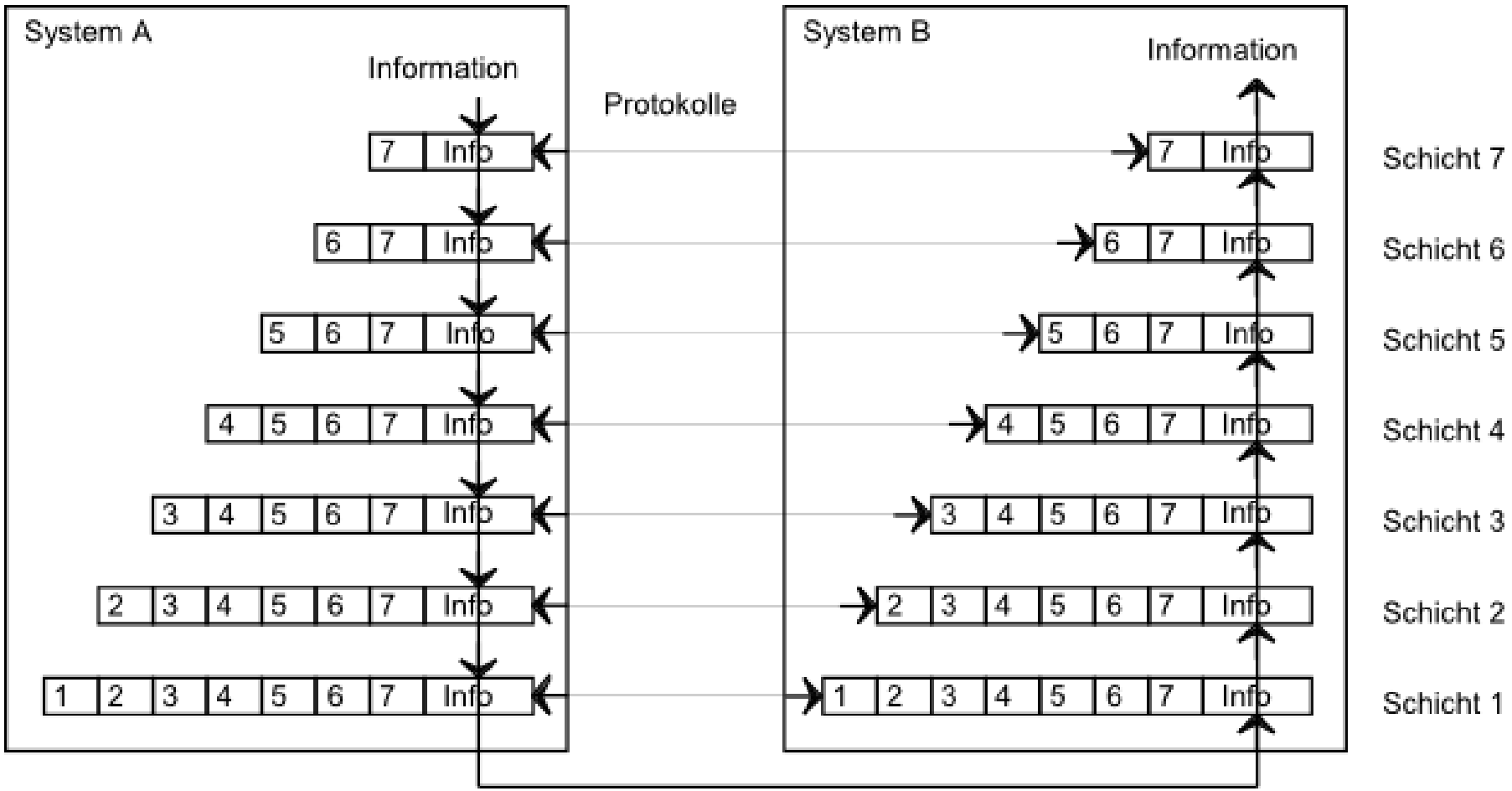
The Layers Pattern
Three-Tier Architecture



The Layers Pattern
The ISO/OSI Communication Layers Model



The Layers Pattern
The ISO/OSI Communication Path



Pipes and Filters

Problem

- Data stream oriented systems shall be partitioned into a number of steps
- The system shall be easily extendible
- The system shall support parallel processing as much as possible

Solution

- The system task is being decomposed into separate processing steps
- Each step is being implemented as a **filter**
- Each filter is connected to the following via a **pipe**
- Filters read data, transform them, and output the transformed data

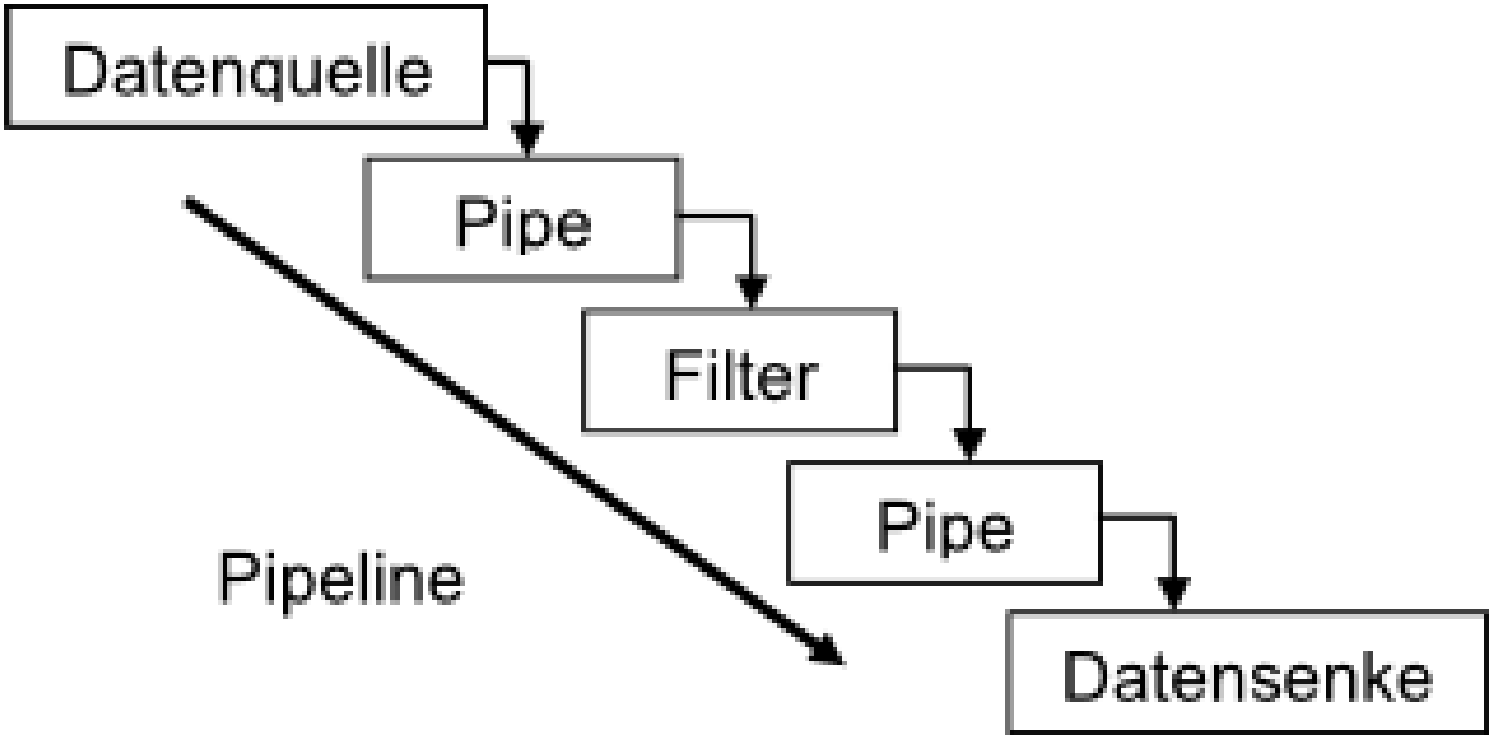
Filters

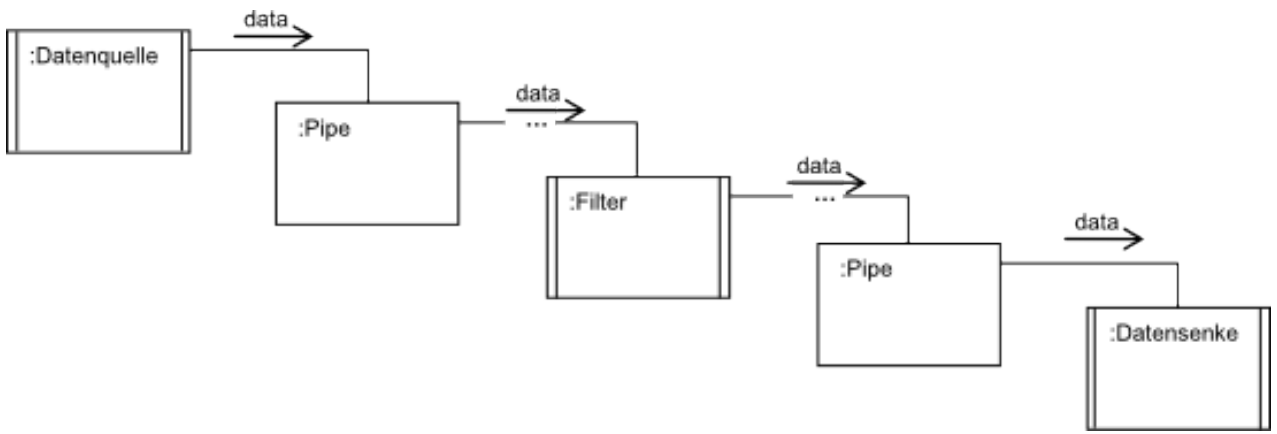
- can remove data from their input data stream
- can add data to their input data stream
- can modify data read from their input data stream

Pipes

- A pipe transfers and buffers data
- A pipe decouples the entities it connects
- Writing into a pipe and reading from a pipe can be asynchronous

- We call the arrangement of filters connected with pipes a **pipeline**
- A pipeline connects a **data source** with a **data sink**
- Typically active filters are being used, often implemented as separate parallel processes

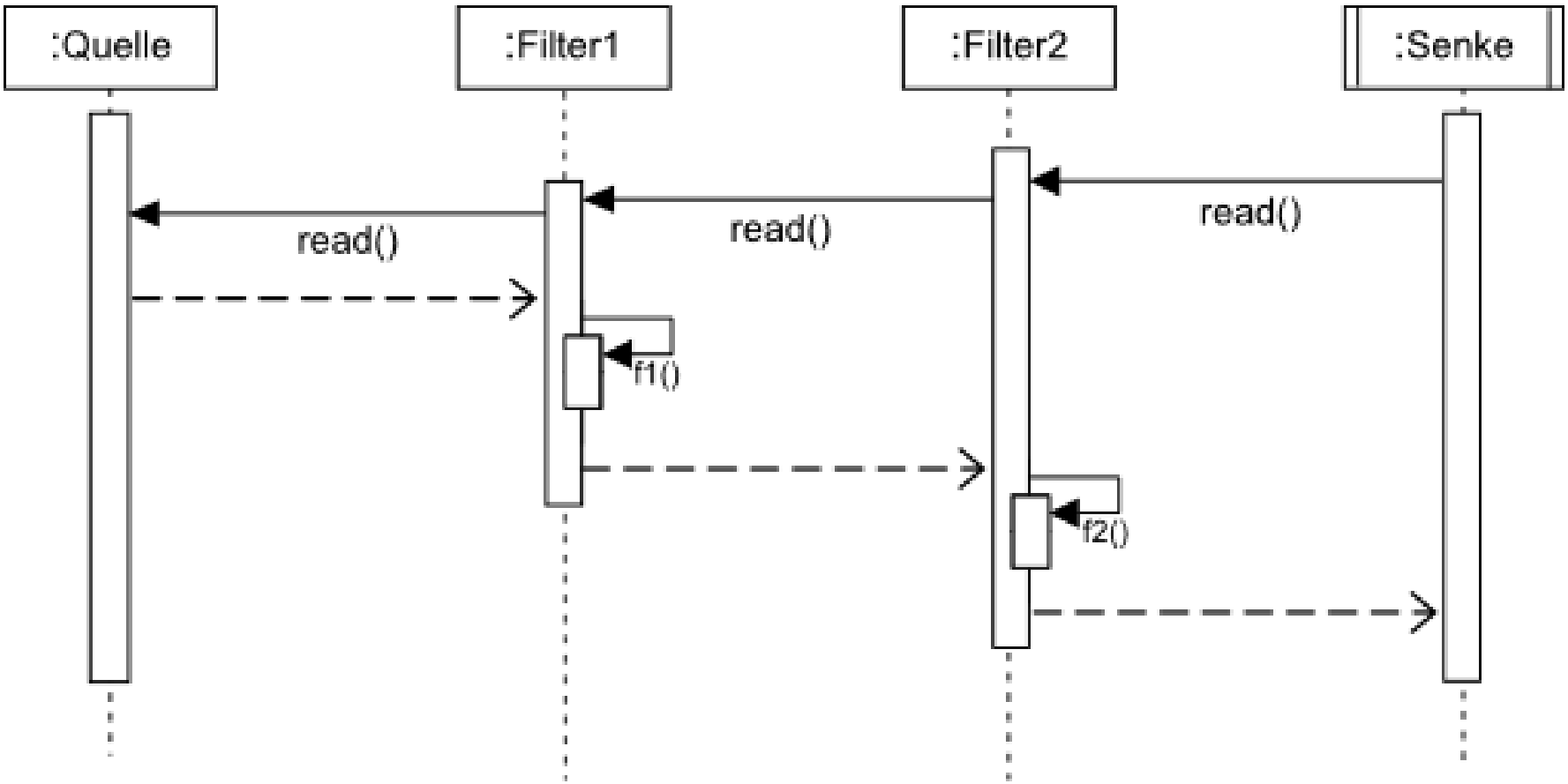




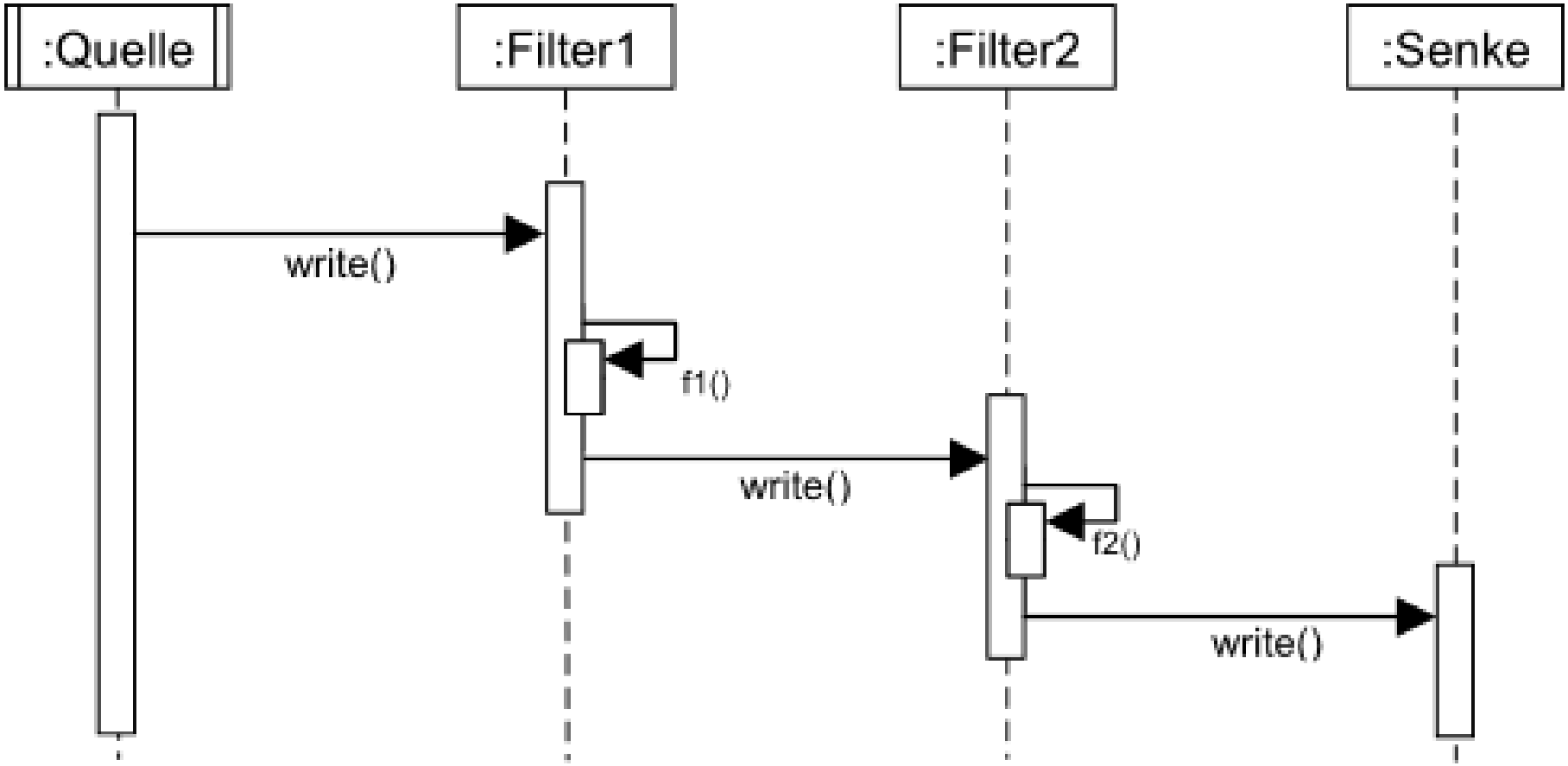
Participants

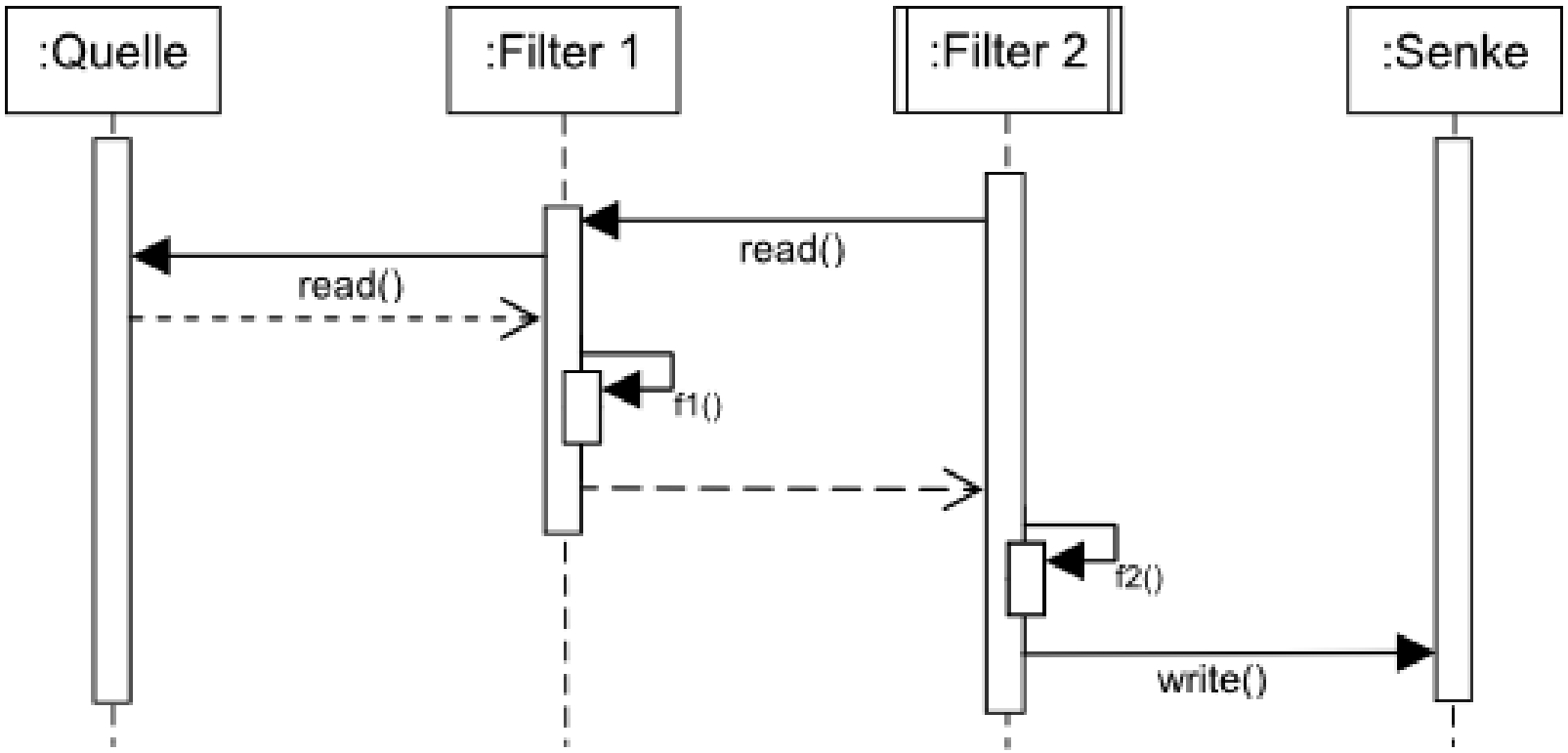
- Data source
- Pipe
- Filter
- Data sink

The Pipes and Filters Pattern
Pull Principle

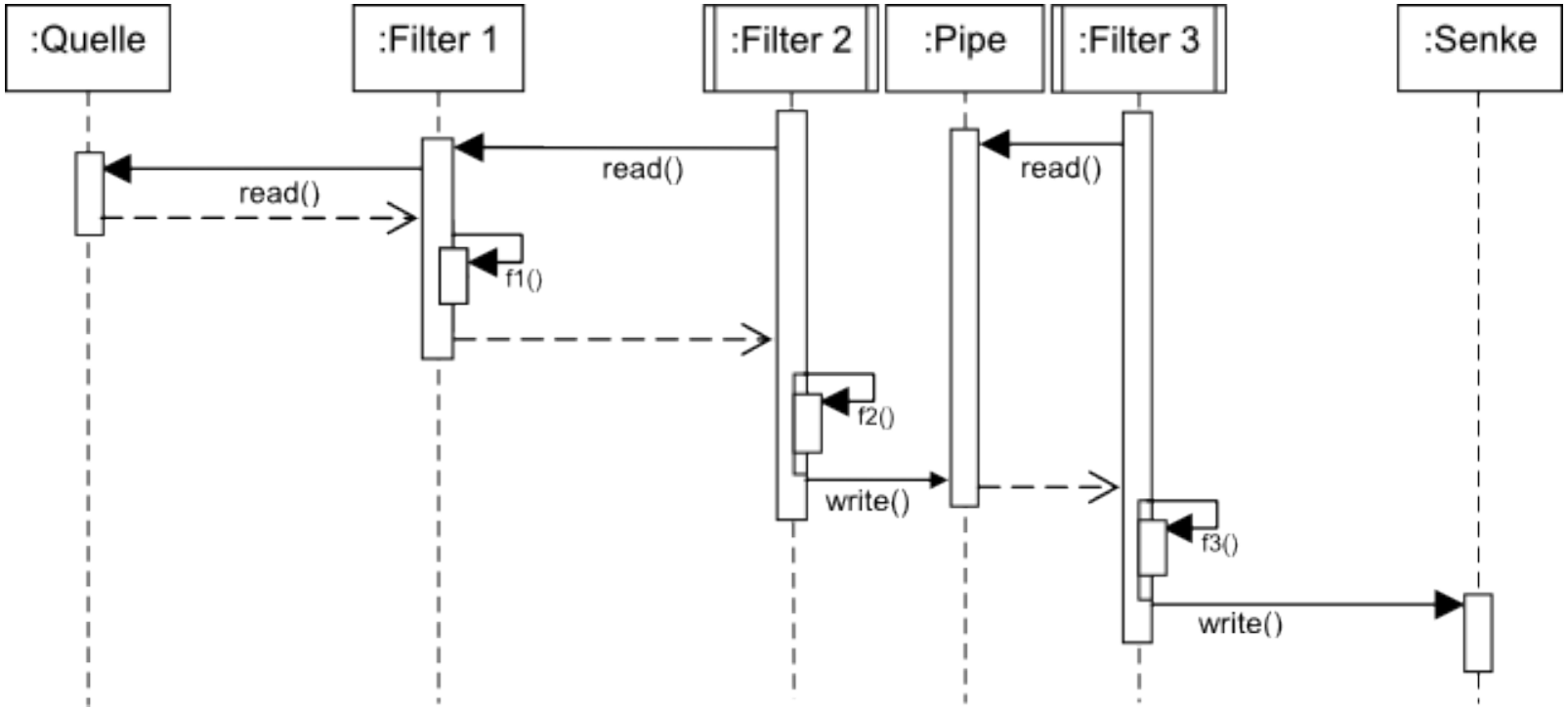


The Pipes and Filters Pattern
Push Principle





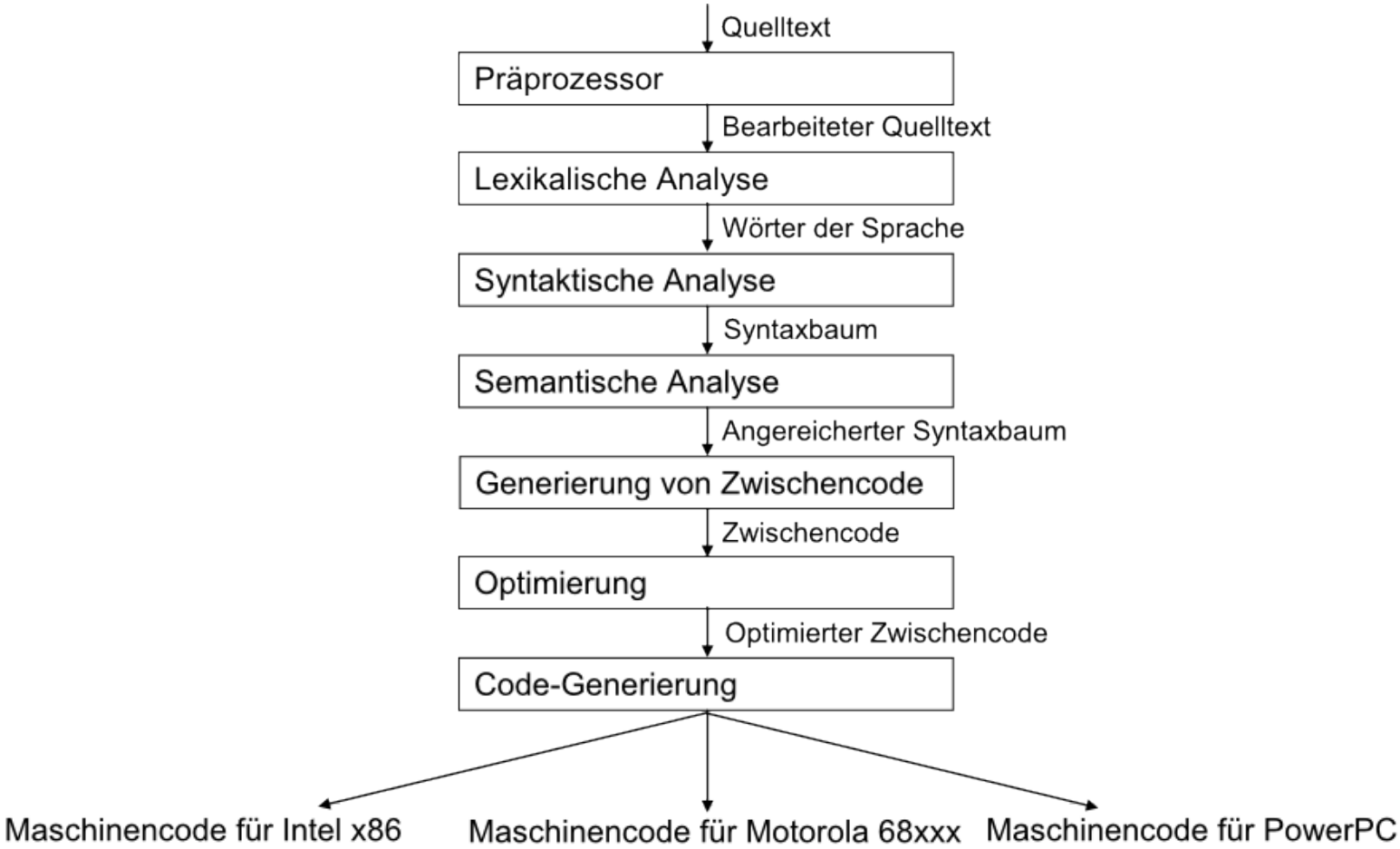
The Pipes and Filters Pattern
Asynchronous Decoupling of Two Filters



- Easy to add new filters or to remove them
- Easy to change the order of filters in a pipeline
- Filters can be developed independently, facilitating rapid prototyping
- Only adjacent filters share data, all others are decoupled
- Storing intermediate information is unnecessary
- Filters can run in parallel
- Filters can be reused easily

- It is difficult to handle errors properly since there is no single system state
- The slowest filter in the pipeline determines the throughput
- Data format transformations may be required if filter input and output formats do not match

The Pipes and Filters Pattern
Application Areas: Compiler



Plug-In

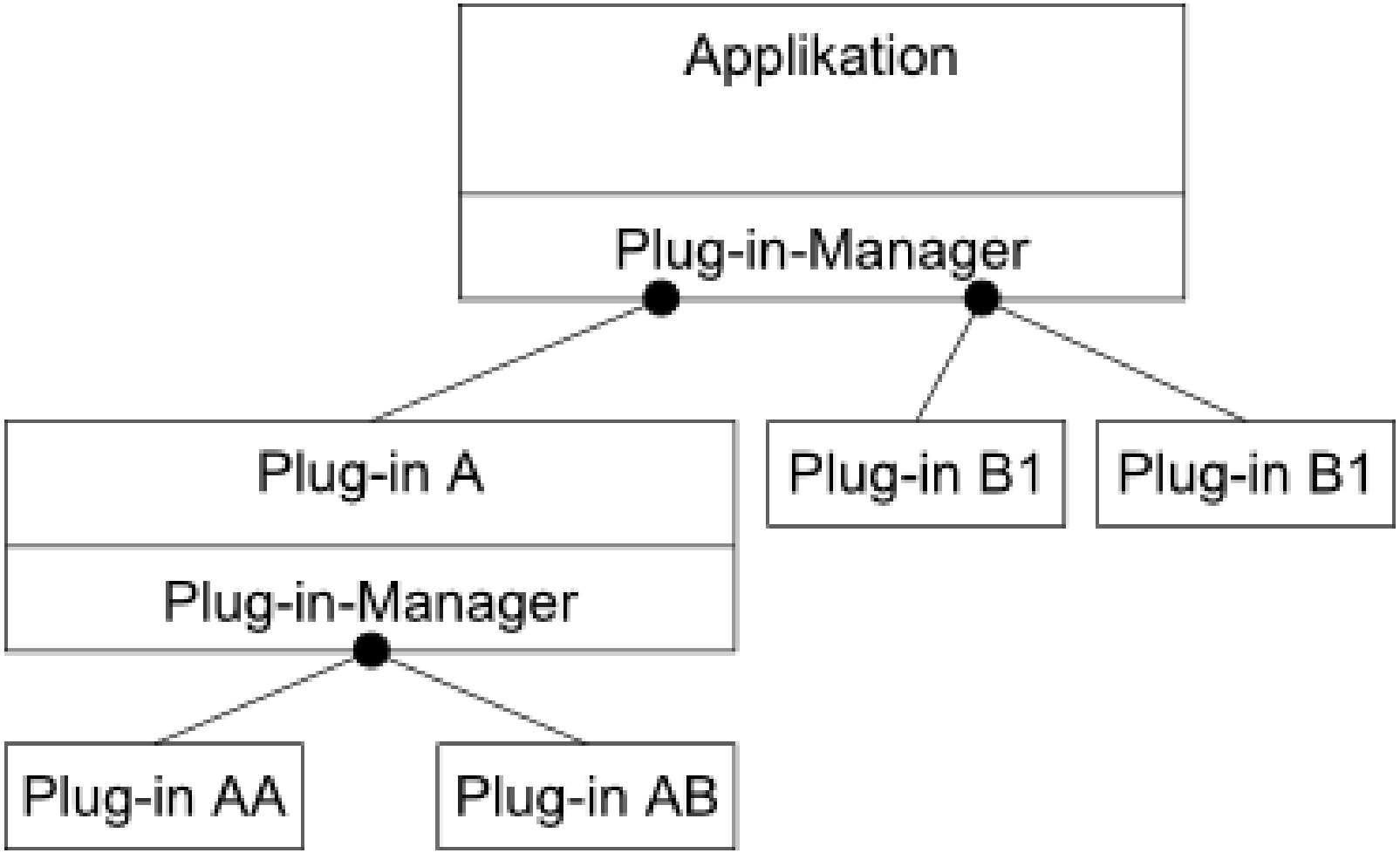
Problem

- It shall be possible to add new functionality to existing software without having to modify it
- New functionality shall be addable by third parties
- The system shall work without add-ons, but shall provide more functionality with them

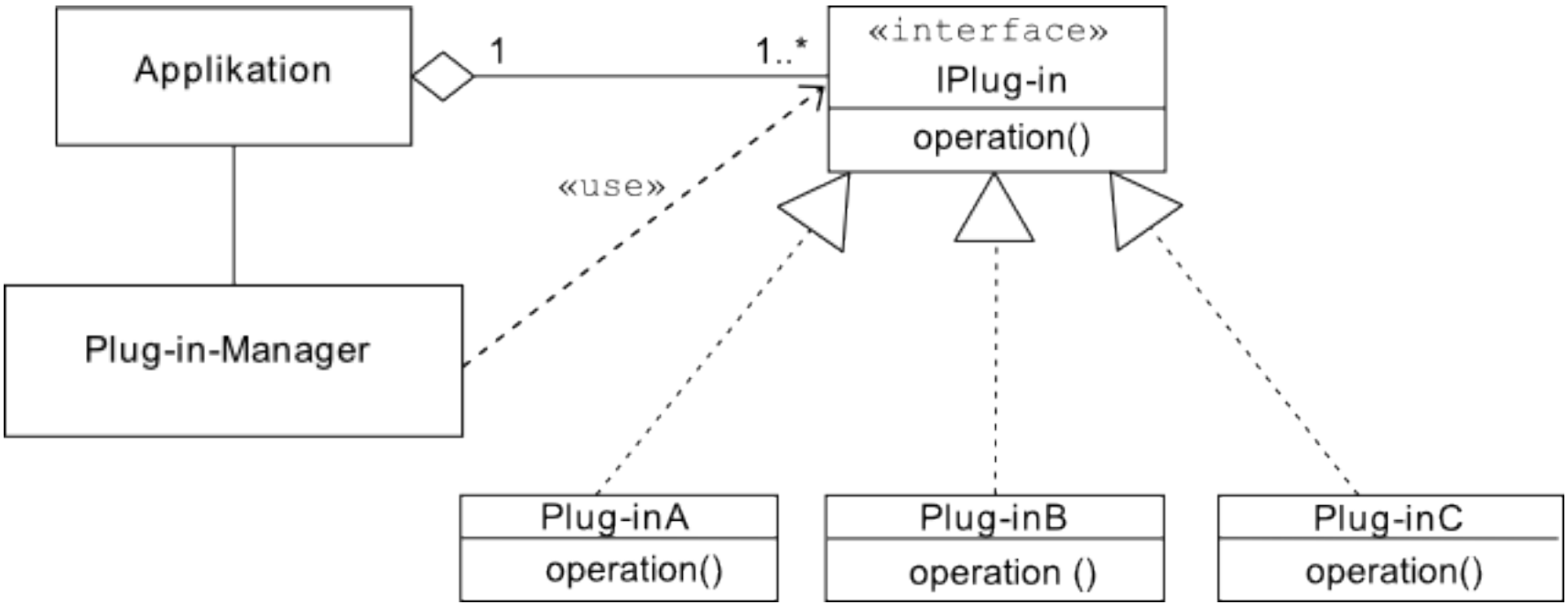
Solution

- There are interfaces defined as extension points
- Plug-ins implement these interfaces and are registered with the core system
- During runtime plug-ins replace the interfaces with real implementations

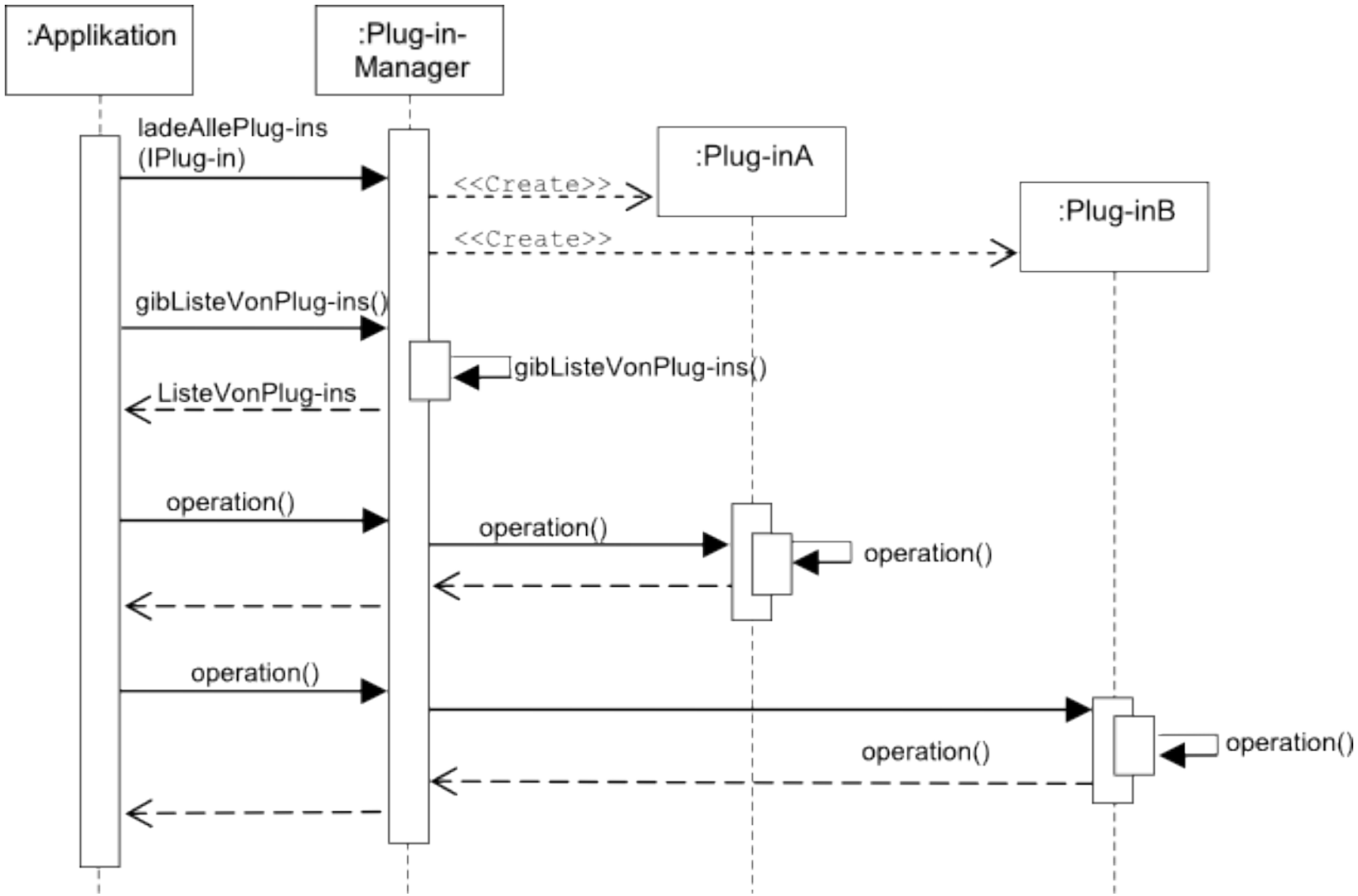
The Plug-In Pattern Pattern
Plug-in Architecture



The Plug-In Pattern Pattern
Class Diagram



The Plug-In Pattern Pattern
Dynamic Behaviour



Broker

Problem

- Large systems must be scalable (more users, more transactions)
- System components must be distributable to several computers
- Components must thus be loosely coupled so that they can be distributed
- It wouldn't be feasible to have every component know all the others

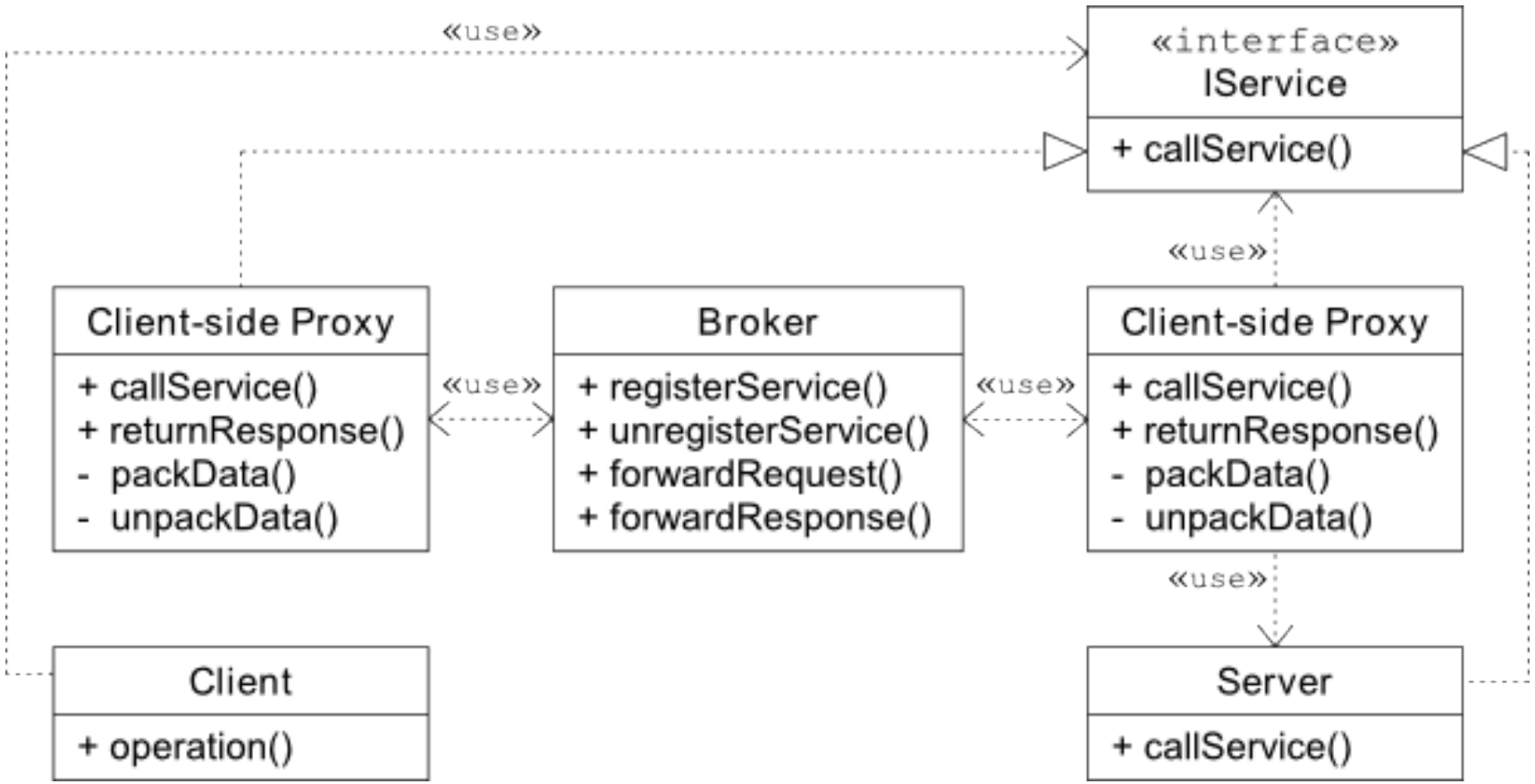
Solution

- Components are classified by their role in inter-component communication
- Server components provide one or more services
- Client components consume one or more server component services
- These roles can change during operation, i.e. server can become clients
- The broker pattern puts an intermediate layer between client and server components in a distributed system

- Server register their services with the broker and then wait for requests
- Clients request a service from the broker
- The broker transfers the request to one of the servers
- The server responds to the broker who in turn transfers the response to the client
- The client does not need to know the physical location of the server
- To the client it is immaterial which physical server instance it is served by
- The broker is the only one that has to have knowledge of the physical instances of clients and servers

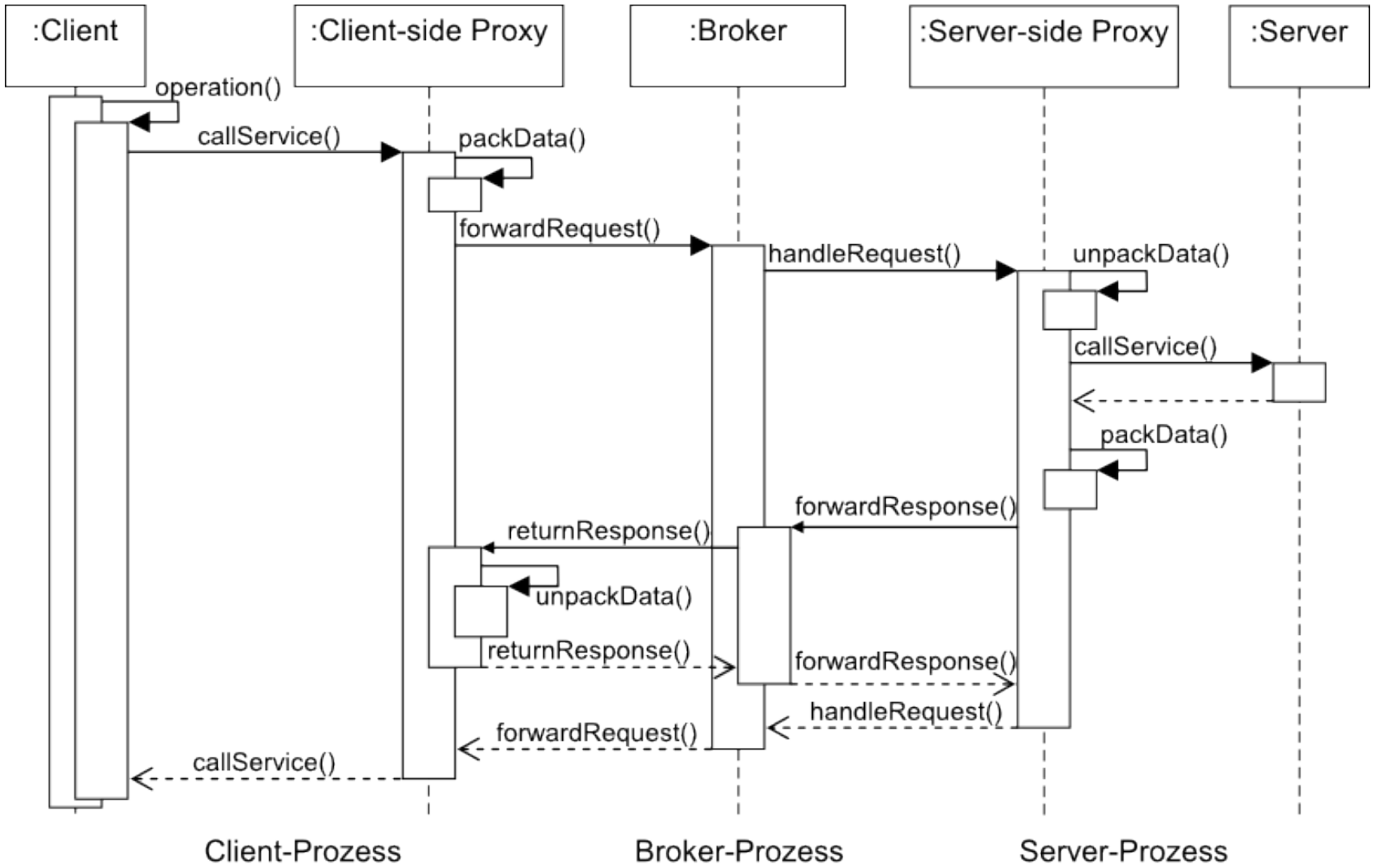
- A client class calls a method of a server class
- This works only if server and client classes reside in the same computer or VM
- In a distributed system, the method calls need to be serialized for transmission over a communication network (**marshalling**)
- At the receiving side the serialized method call needs to be converted back into a standard method call (**unmarshalling**)
- For this purpose we introduce a **client-side proxy** and a **server-side proxy**
- The client thus call a client-side proxy method which interacts with the broker
- The broker transfers the clients call to the server-side proxy which interacts with the server

The Broker Pattern
Structure

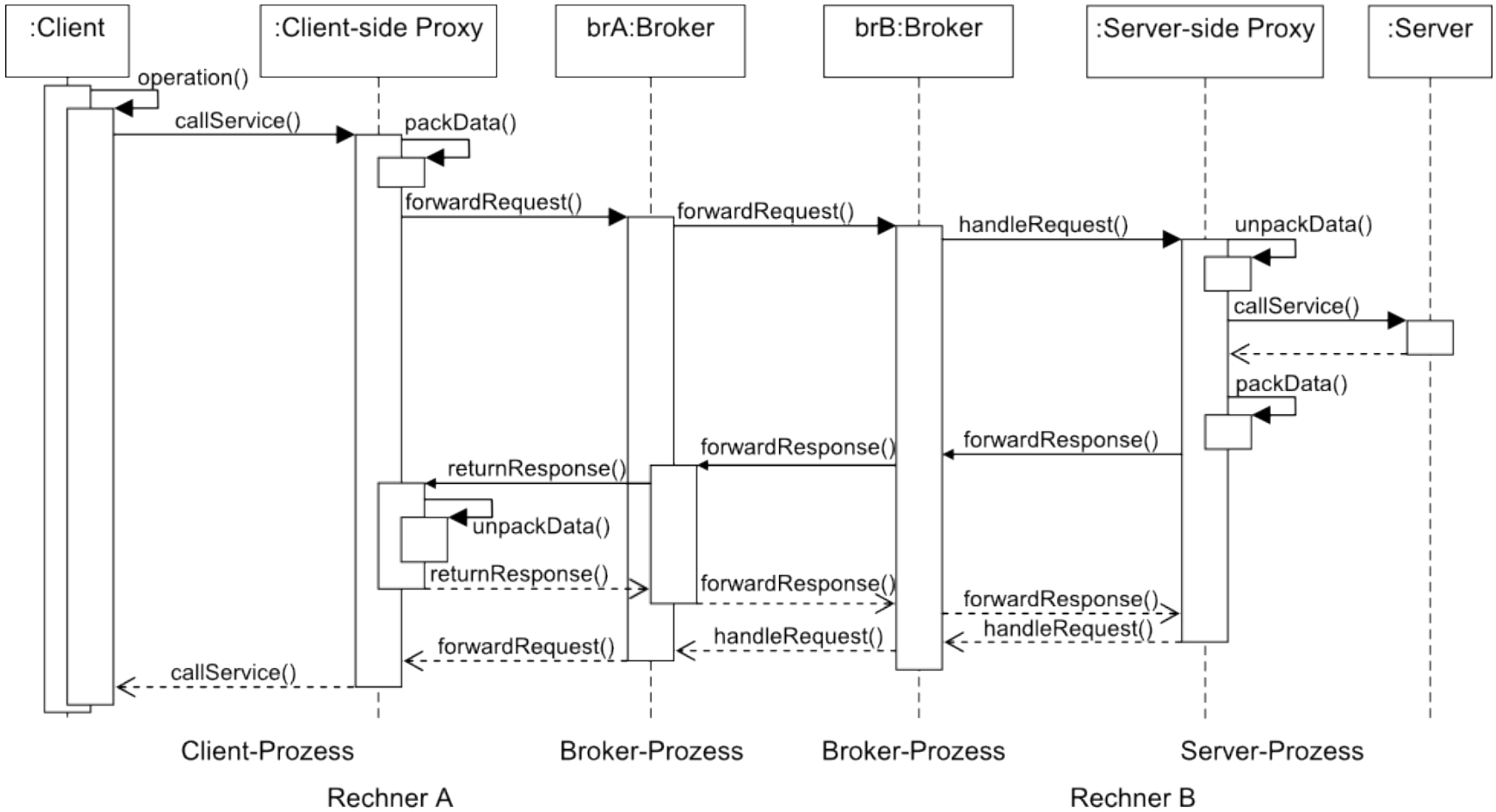


- Client
- Client-side proxy
- Broker
- Server-side proxy
- Server

The Broker Pattern
Dynamic Behaviour, Single Computer



The Broker Pattern
Dynamic Behaviour, Several Computer



- The broker pattern separates communication between client and server from functionality (separation of concerns)
- Server implementation can change as long as interface remains stable
- Client has no need to know anything about the physical server instance, it only needs to know the local broker and the logical service it requests
- The same is true for the server
- Client and server are platform-independent. They can be implemented in different programming languages
- The broker pattern enables very large software systems distributed across many computers

- The local broker constitutes a central point of failure in case anything goes wrong
- Performance is penalized by marshalling and unmarshalling operations
- Broker can become a performance bottleneck
- Brokers and proxies are tightly coupled.

- The internet with a domain name server (DNS)
- Performance is penalized by marshalling and unmarshalling operations
- Broker can become a performance bottleneck
- Brokers and proxies are tightly coupled.

Service-Oriented-Architecture

Problem

- Business processes need to be supported by IT
- As business processes change frequently, IT system need to change as well
- If business process architecture is tied too closely to IT architecture, modifications in the former may require architectural changes on the latter
- Architectural changes in software systems are expensive and take a long time

Solution

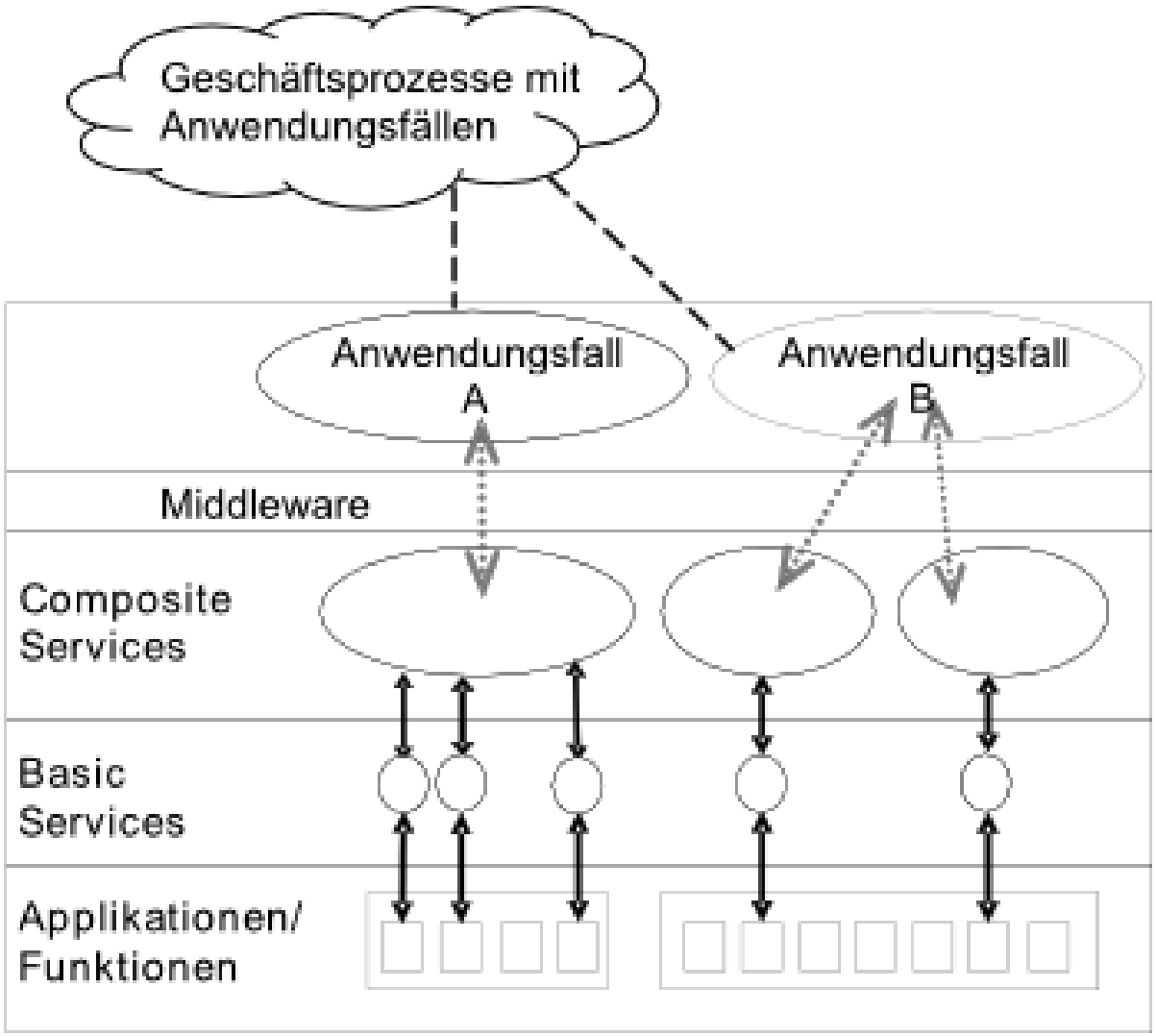
- We encapsulate business processes or parts thereof in components
- A **service** or **application service** implements a **use case**

The Service-Oriented-Architecture Pattern
Properties

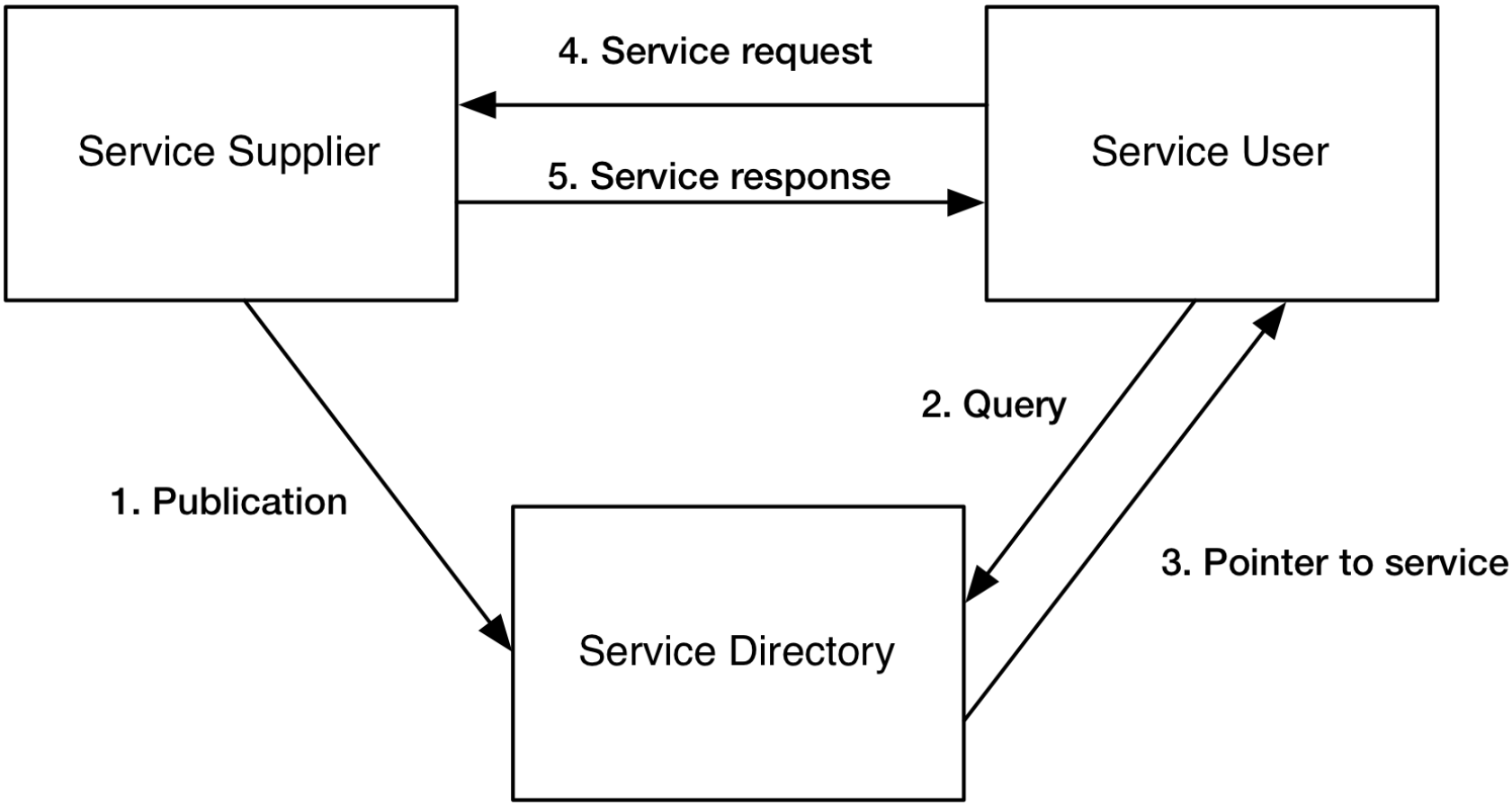
- Distribution - a service is available in a network
- Component appearance, can be called independently
- Stateless - service starts in the same state with each call
- Loosely coupled - services do not depend on each other
- Exchangeability - standardized interfaces promote this property
- Location transparency - the actual hardware is immaterial
- Platform independence - operating system and hardware are immaterial
- Access via interface - knowledge of interface suffices to use service
- Service directory, service register, service broker

- A **business process** is characterized by a number of use cases
- A use case is supported by a **composite service**
- Often services are composed of **basic services**
- A composite service is comprised of basic services and may use other services

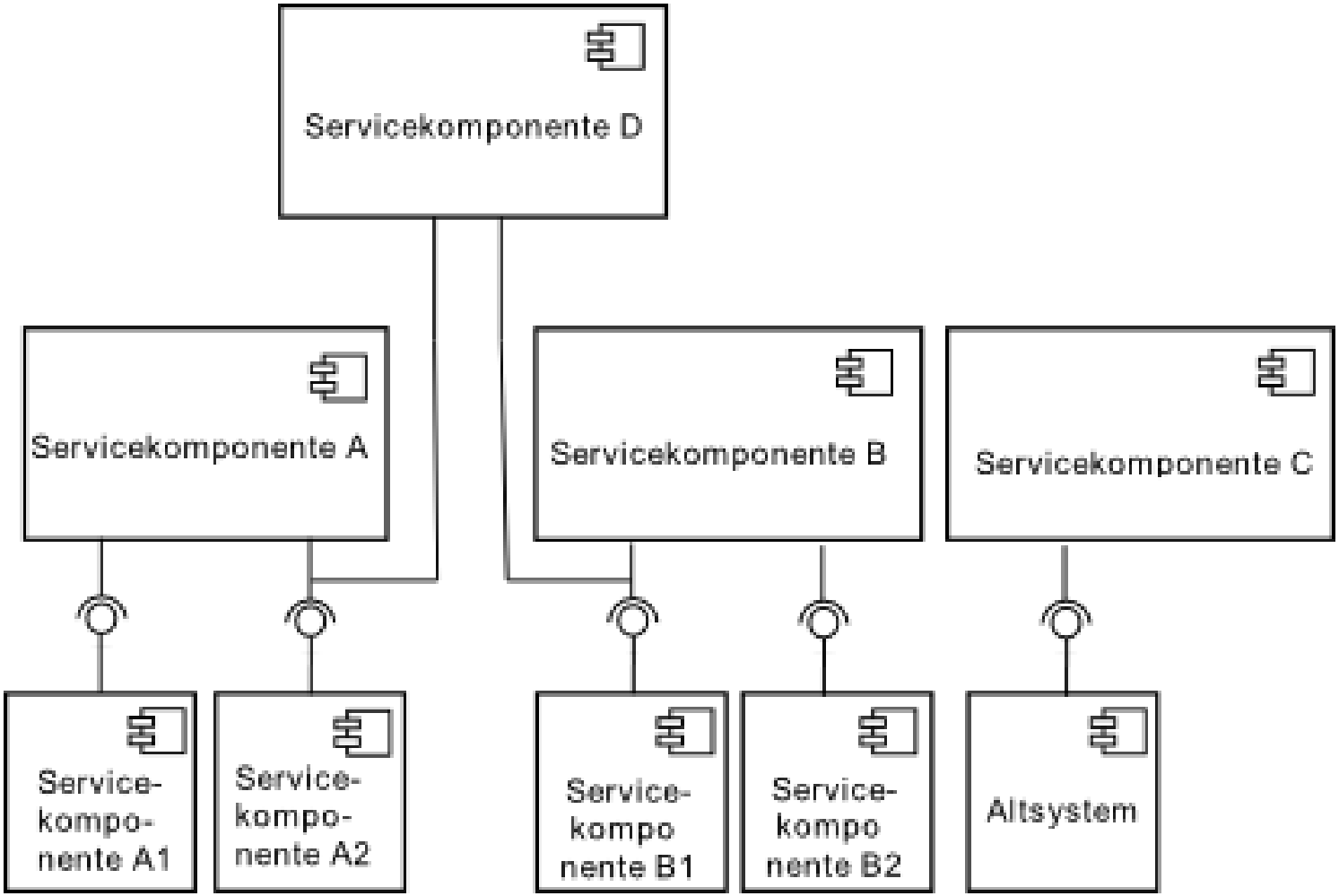
The Service-Oriented-Architecture Pattern
Layer Diagram for Services



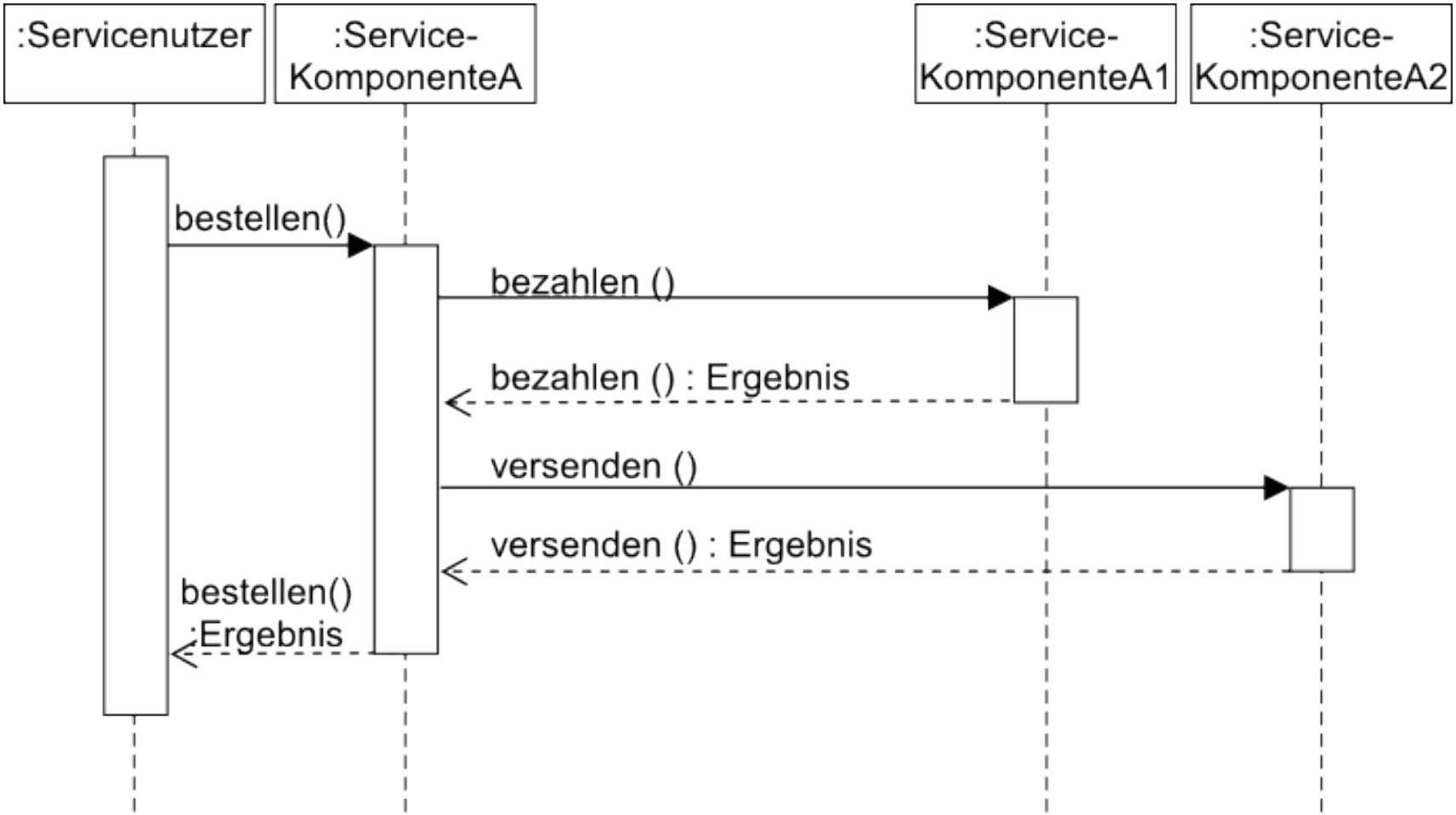
The Service-Oriented-Architecture Pattern
Cooperation in a SOA



The Service-Oriented-Architecture Pattern
SOA Component Diagram



The Service-Oriented-Architecture Pattern
Example for Sequence Diagram



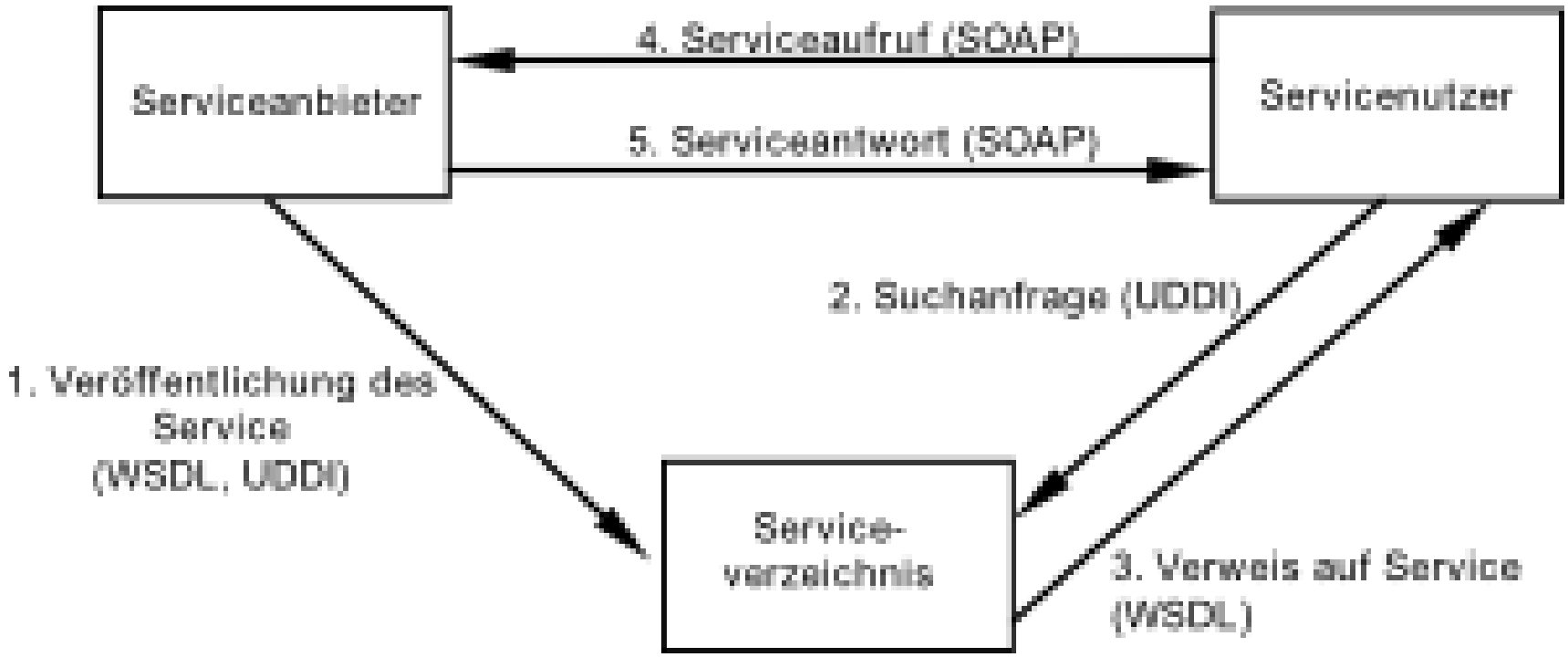
- Mapping of business processes to services gives a good overview of all services required and interfaces used
- Complexity of distributed systems is reduced due to the partitioning
- Services and basic services can be reused
- Services can be exchanged as long as the interface remains compatible

- If services are made too small complexity increases
- There is an overhead due to the required communication through several layers
- The commonly used protocols place require considerable network bandwidth
- A SOA is only effective if the associated business processes are well defined

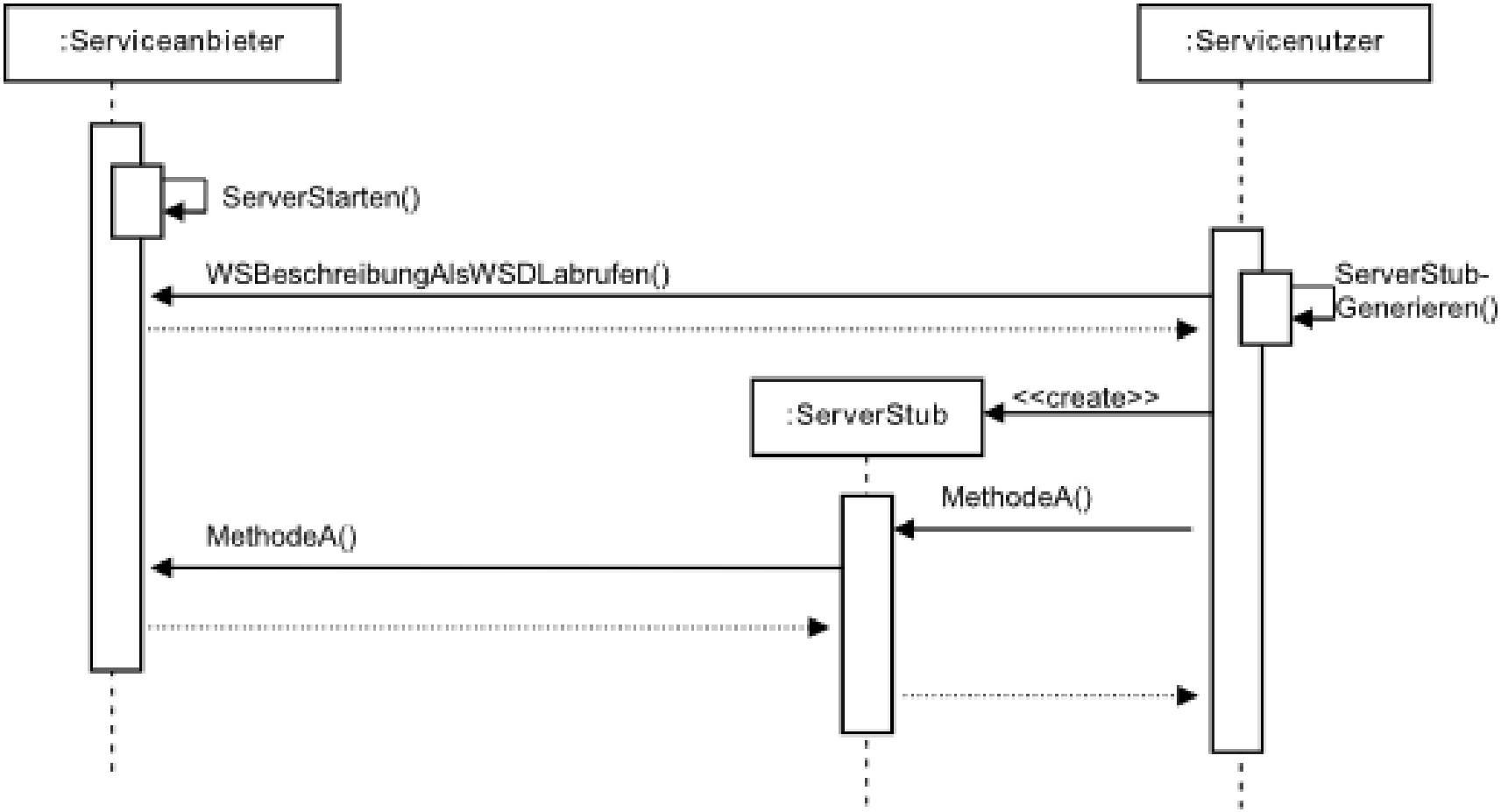
The Service-Oriented-Architecture Pattern
Implementations

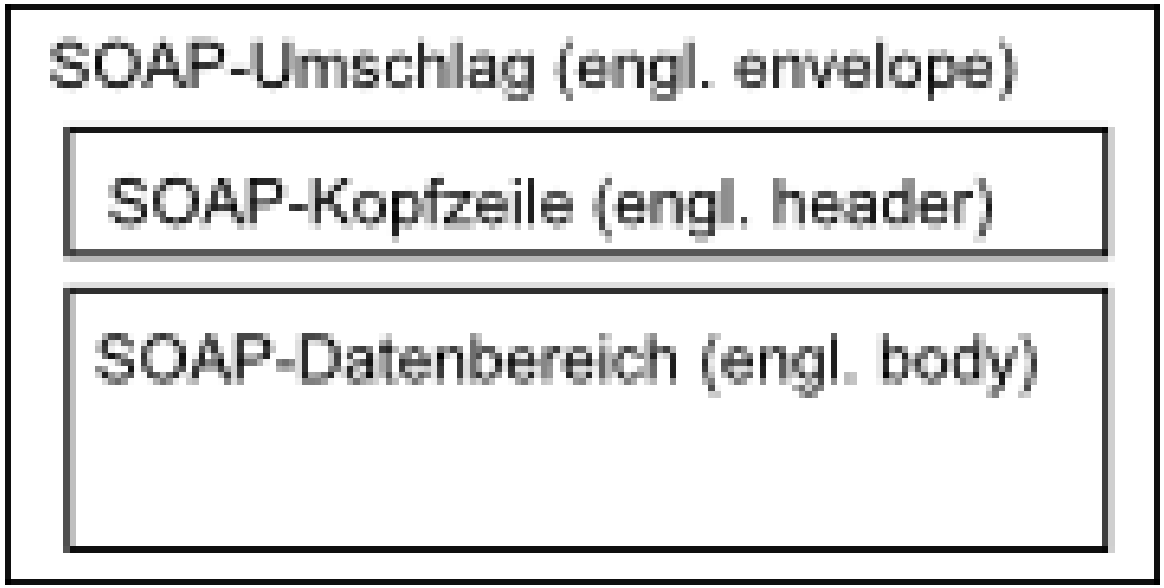
- XML based web services
- REST (**Re**presentational **S**tate **T**ransfer) web services
- CORBA (**C**ommon **O**bject **R**equest **B**roker **A**rchitecture)
- OSGI

The Service-Oriented-Architecture Pattern
XML-based Web Service

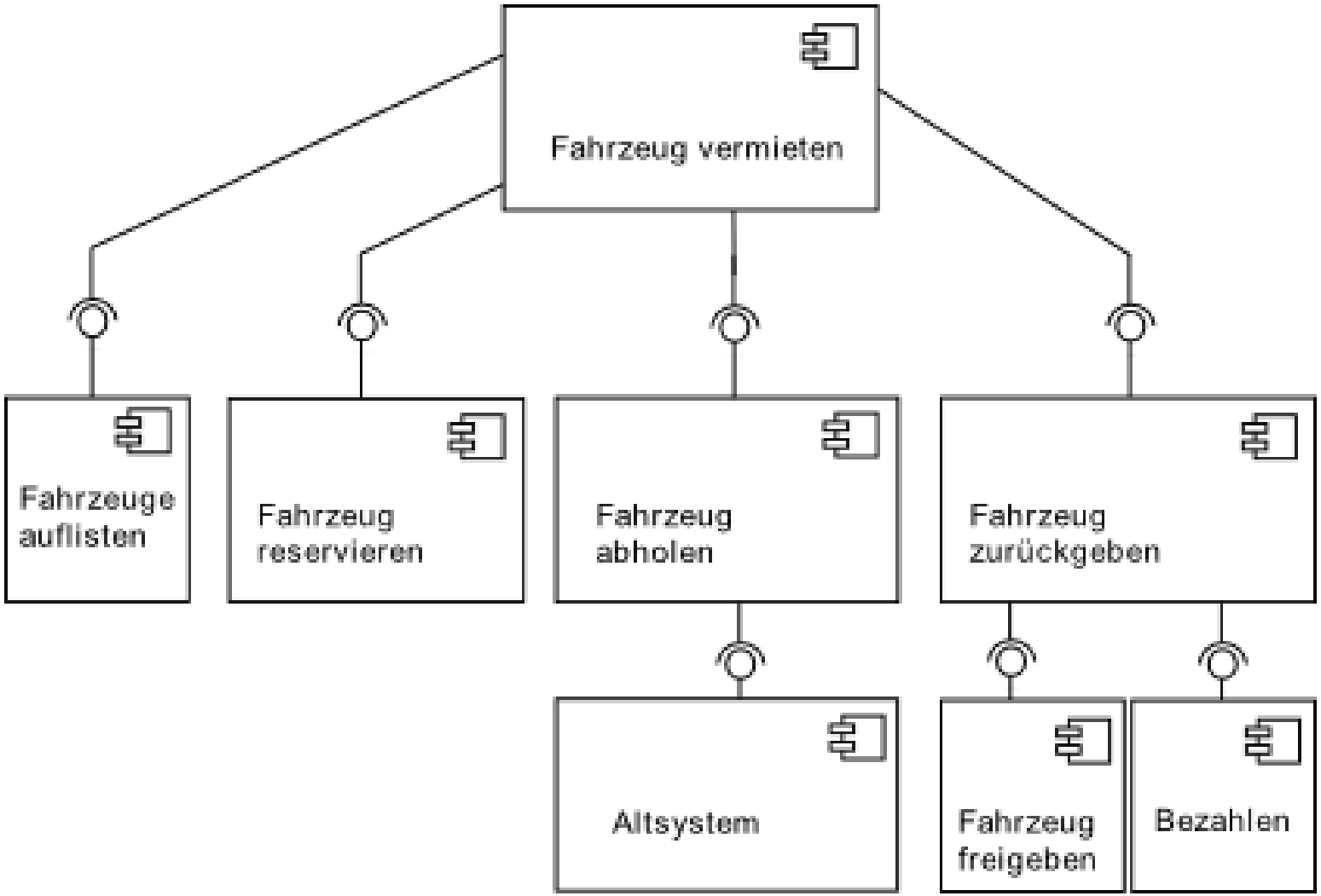


The Service-Oriented-Architecture Pattern
JAX-WS Sequence Diagram





The Service-Oriented-Architecture Pattern
Component Diagram of a SOAP Implementation



Model-View-Controller

Problem

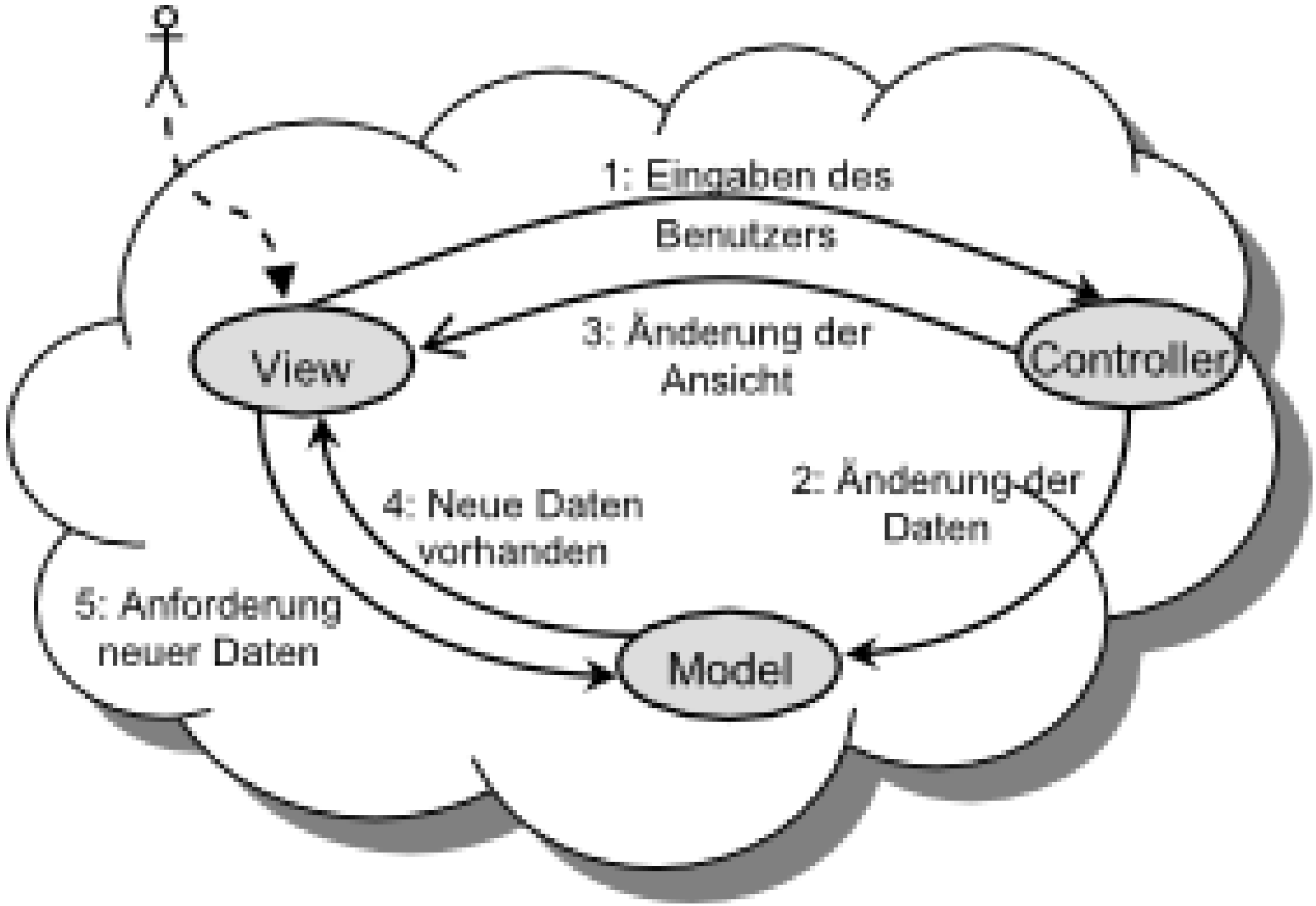
- The user interface is a component that often changes
- The same information may have to be presented in different ways
- Any presentation shall be updated when the underlying information changes
- Example: Spread sheet with graphical and tabular presentation of data

Solution

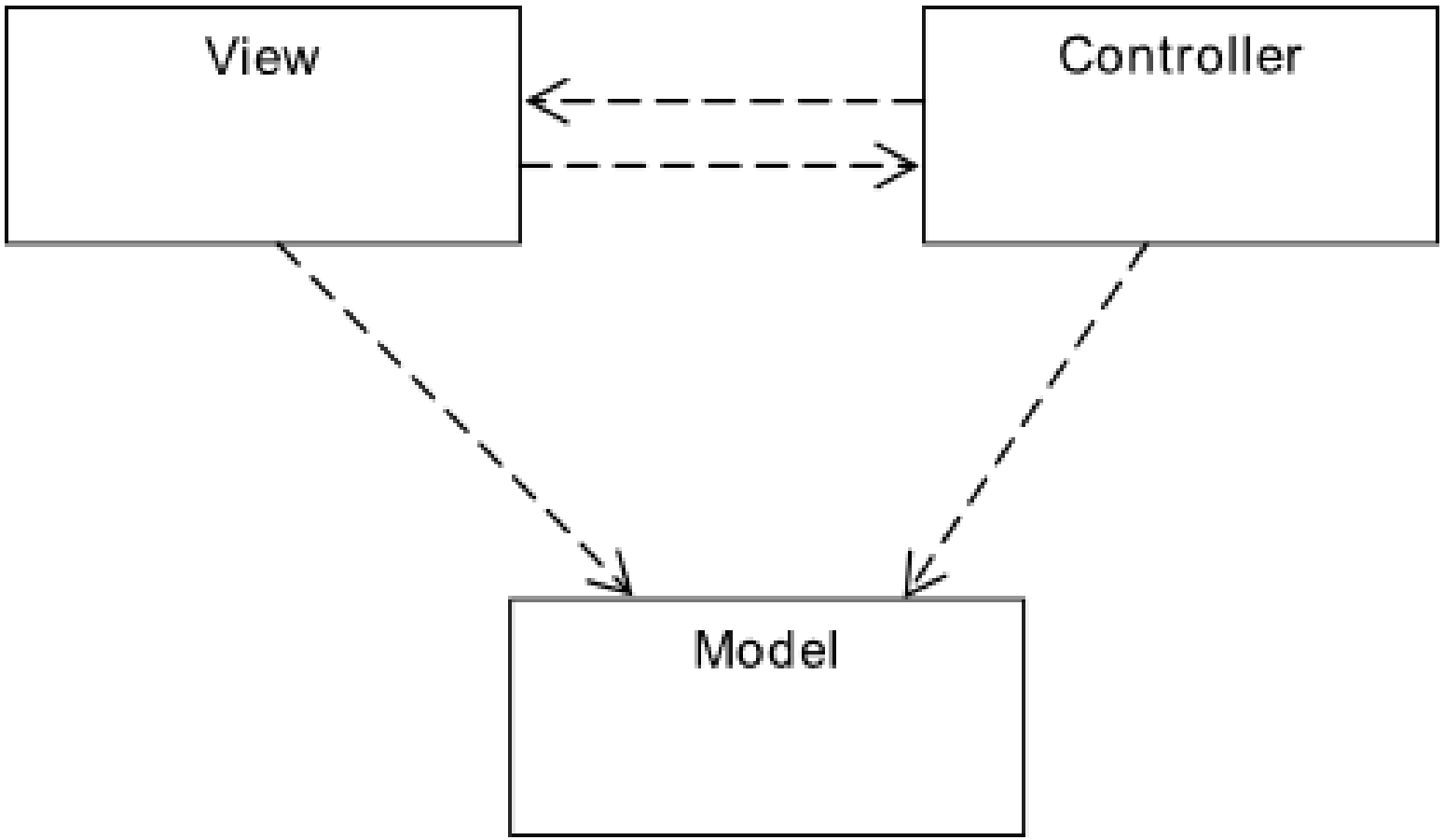
- Data storage and processing (model)
- Presentation logic (view)
- User input processing (controller)

are assigned to dedicated components

The Model-View-Controller Pattern
Component Interaction



The Model-View-Controller Pattern
Components and their Relationships

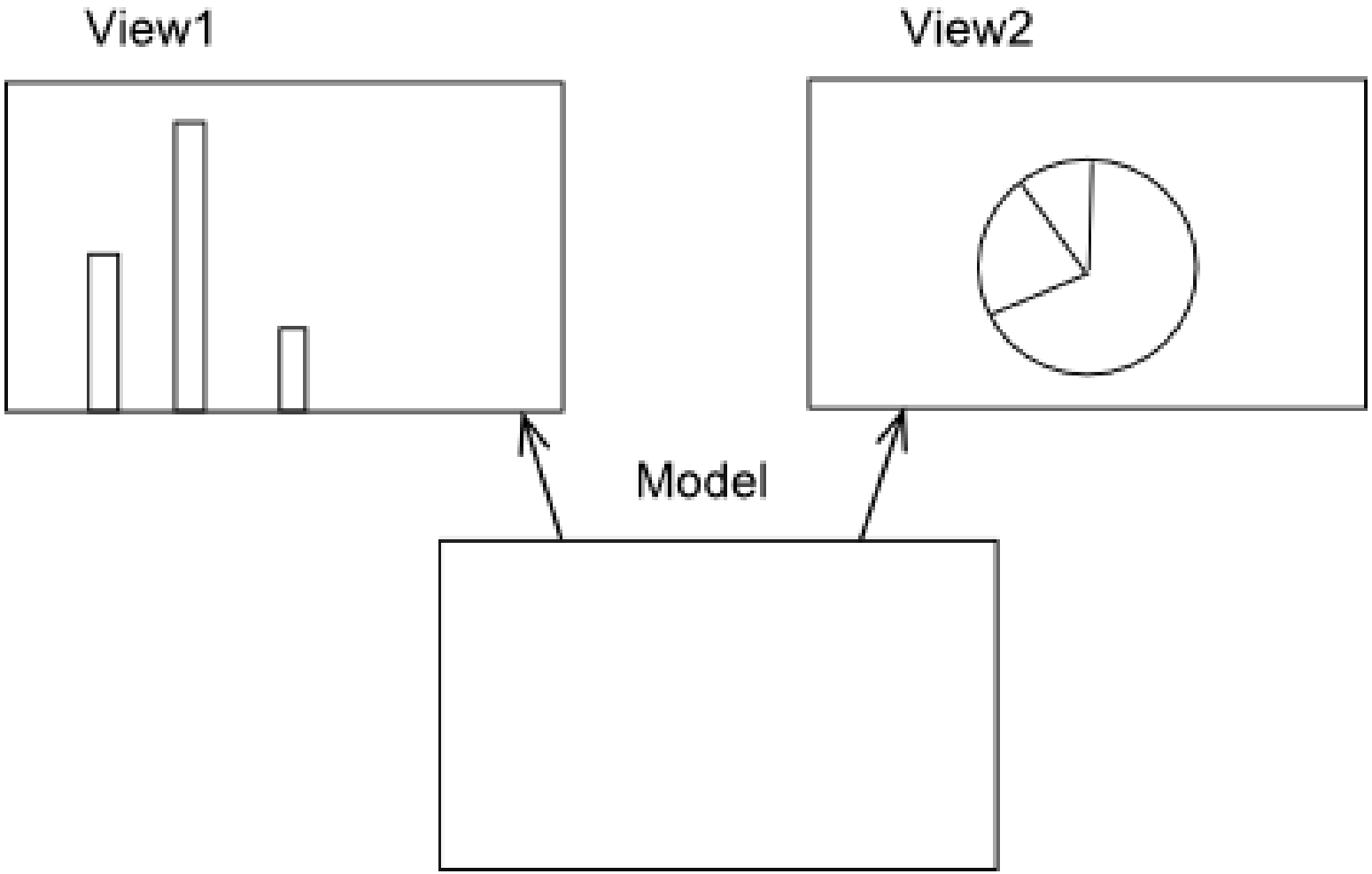


- The model stores the data to be presented in a view
- In simple systems the model may contain some business logic
- The model shall be independent of views and controllers
- In case model data changes the controller may inform the views (passive model)
- In case model data changes the model informs the views (active model)
- In push mode the model actively sends the data to the view
- In pull mode the model informs the view that there is new data

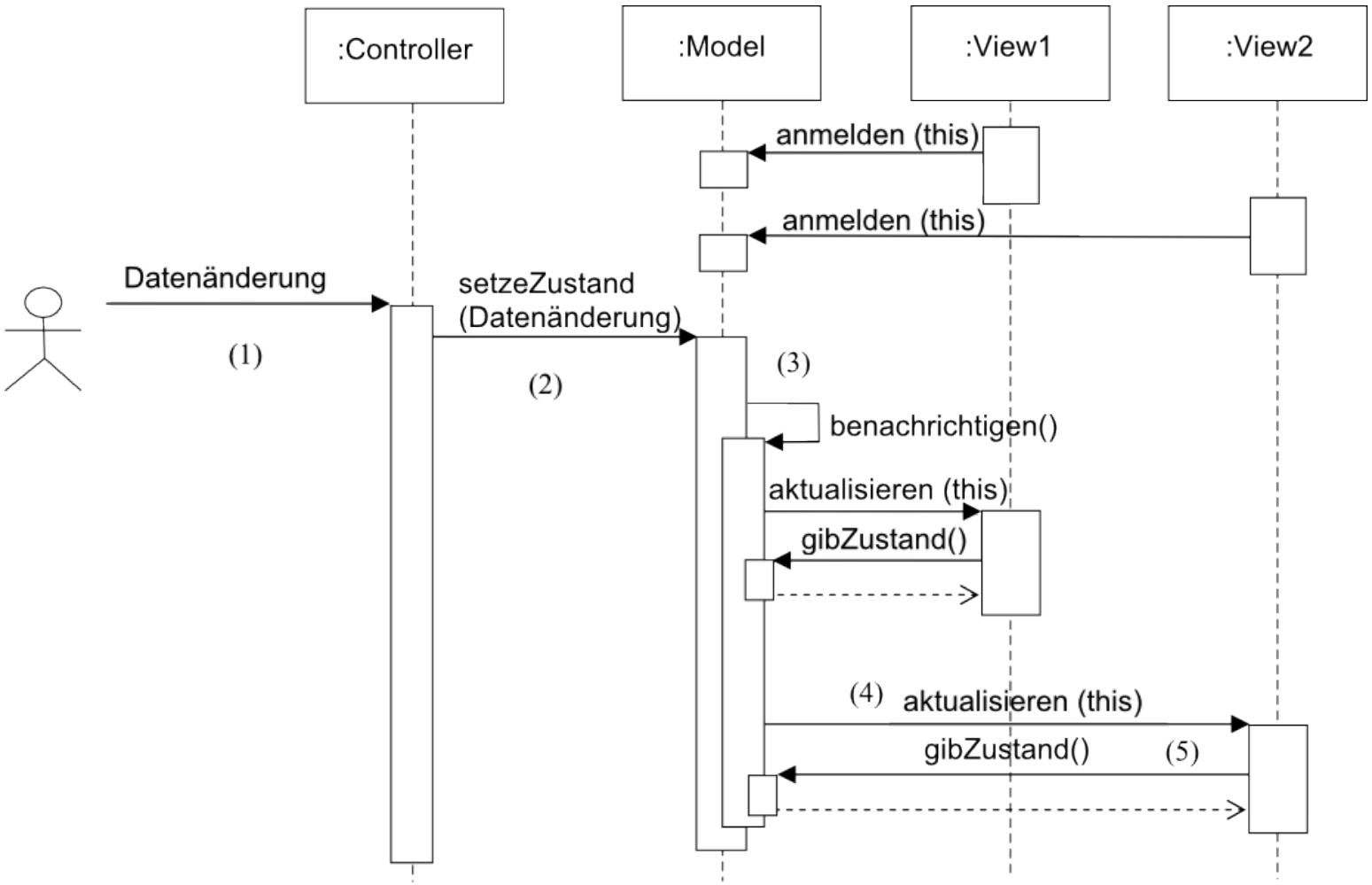
- The view serves to present model data to a user
- There can be many views on the same model data
- All views are updated in case of a model data change
- A view may furthermore present interactive elements like buttons
- Interaction with these elements creates events that usually are handled by the controller

- The controller controls the model and view state based on user input
- The controller transforms events caused by user actions into method calls of the model
- The controller controls the state of the view, e.g. to activate or deactivate control elements

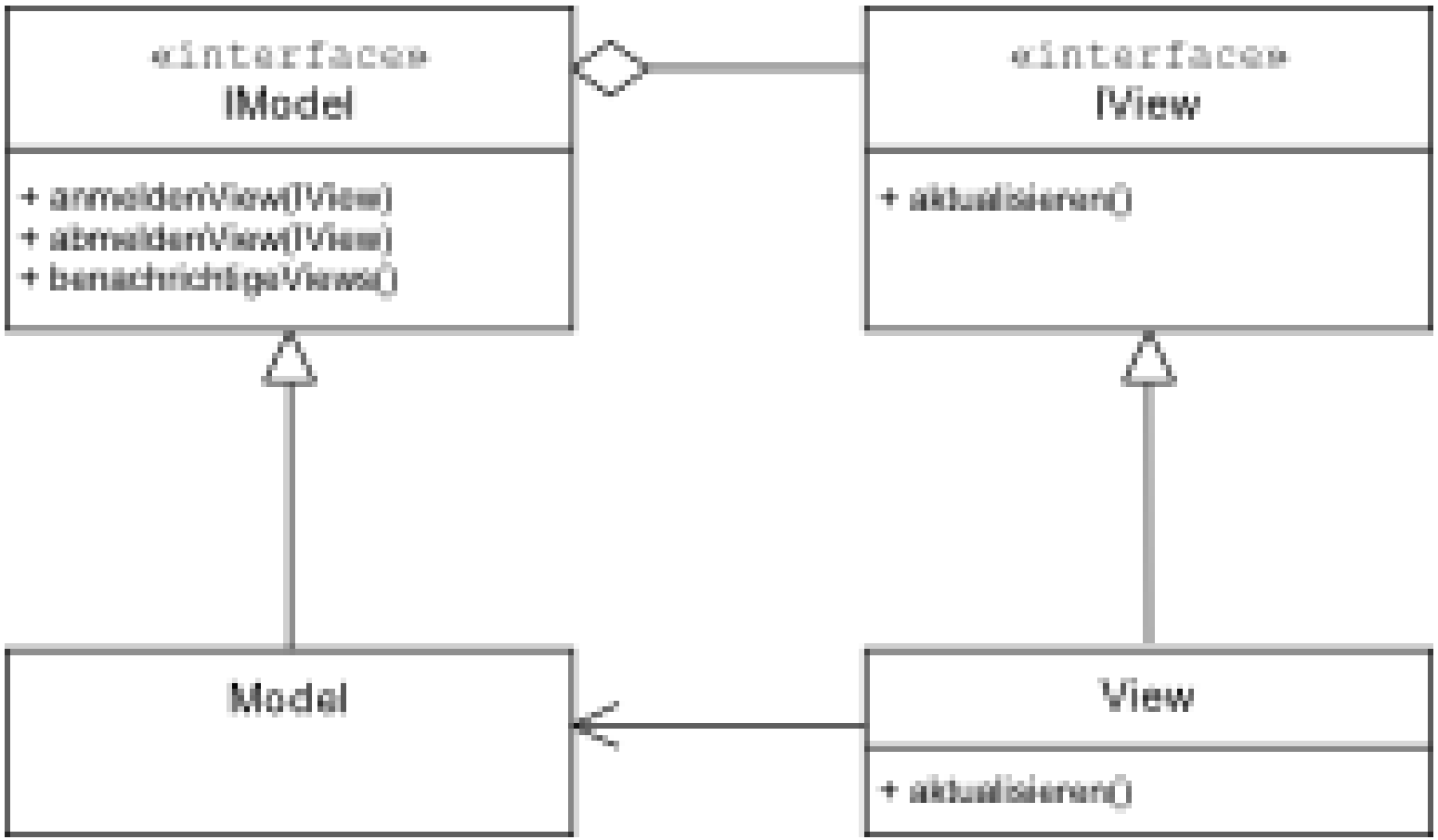
The Model-View-Controller Pattern
One Model and two Views

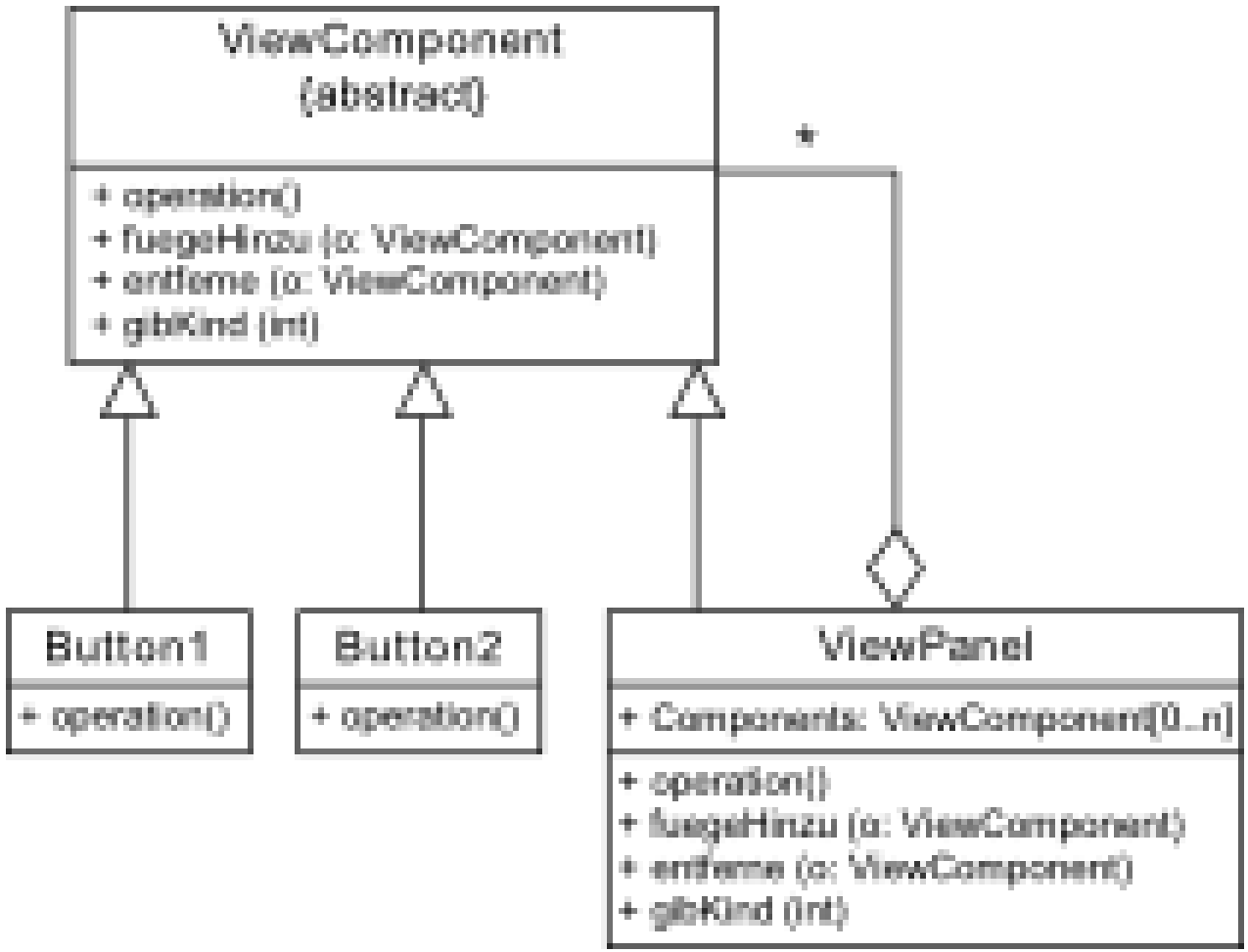


The Model-View-Controller Pattern
Updating the Model

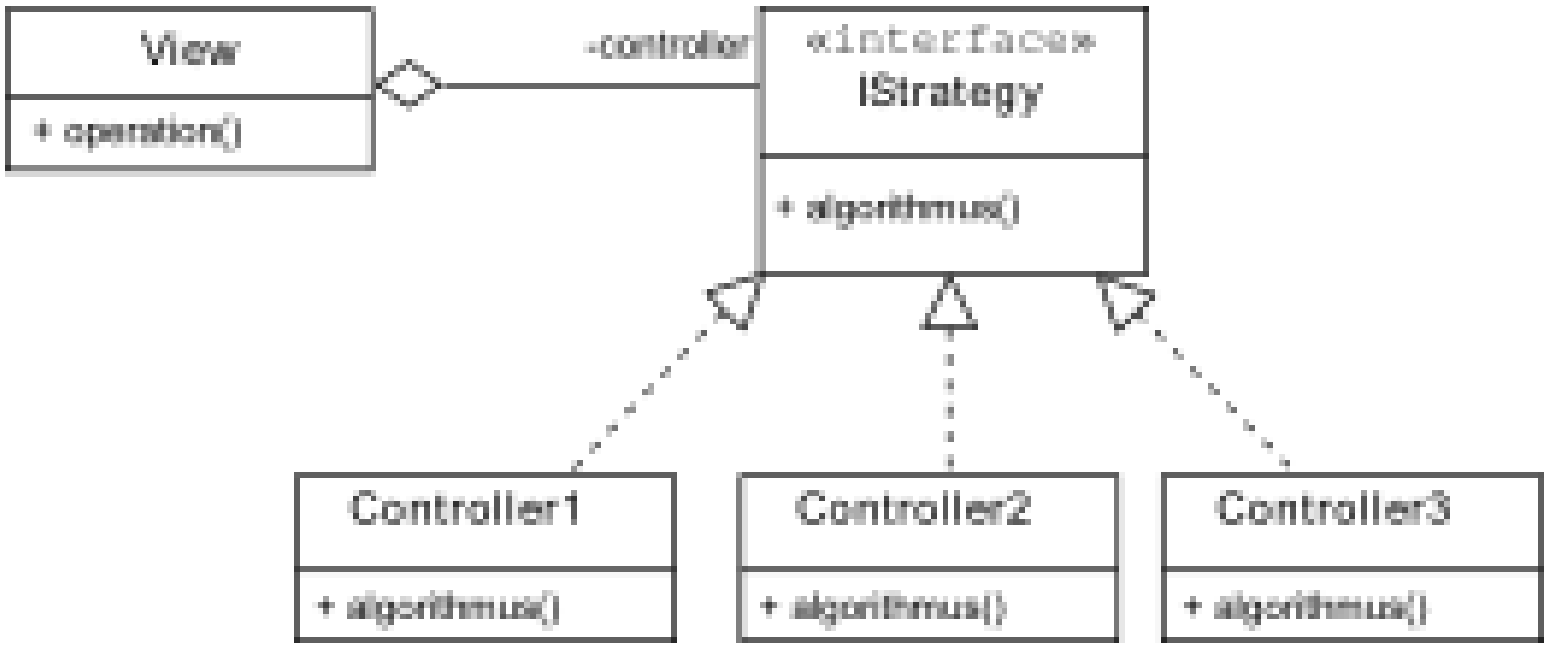


The Model-View-Controller Pattern
MVC and Observer Pattern in Pull Mode



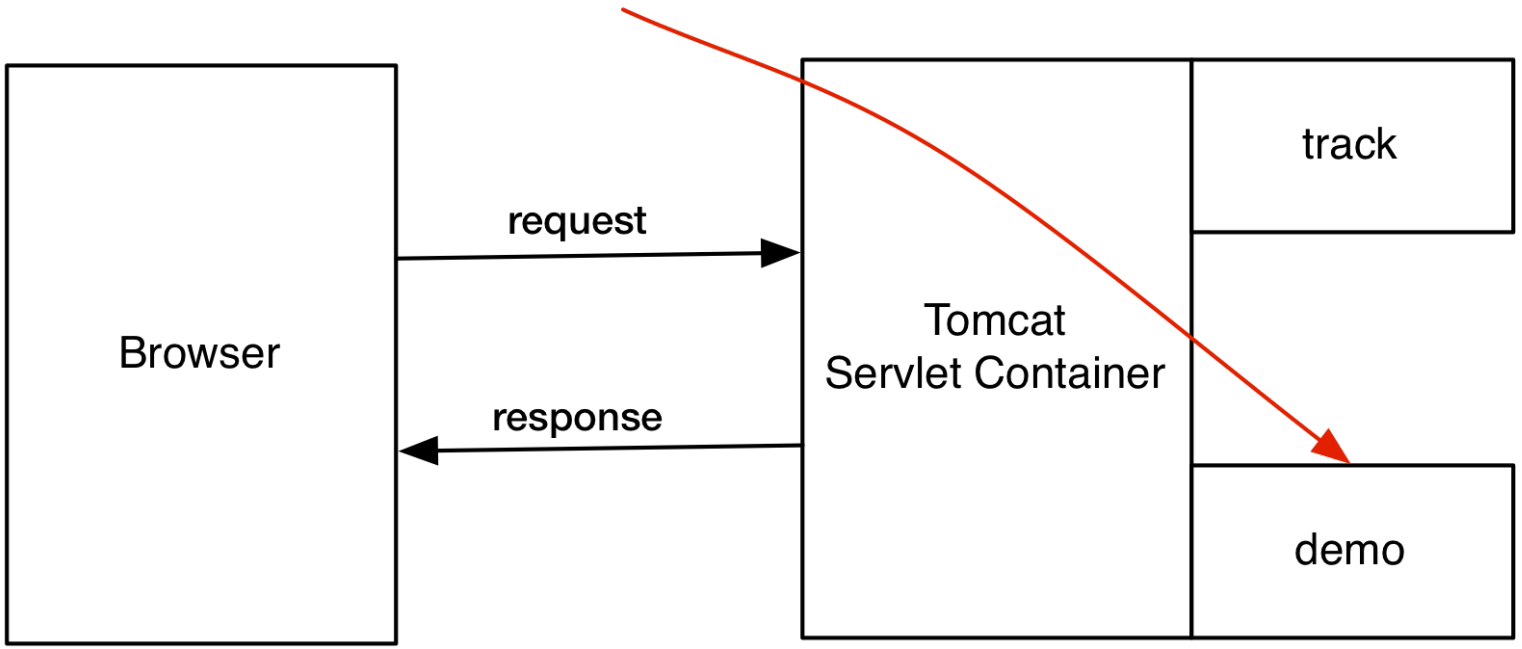


The Model-View-Controller Pattern
MVC and Strategy Pattern

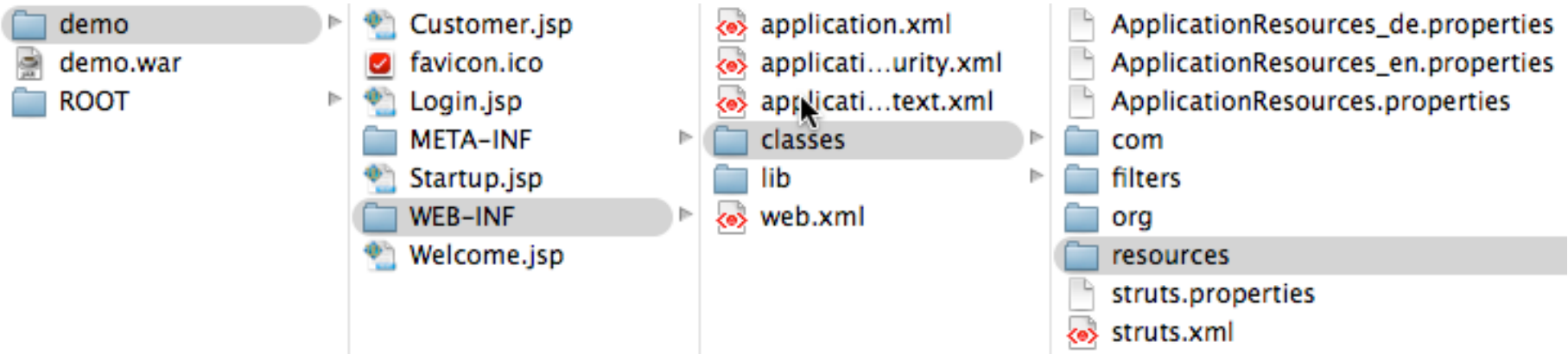


http://www.mydomain.com/demo/Startup.jsp

http://www.mydomain.com/demo?login.action&username="admin"&password="123"



The Model-View-Controller Pattern
Servlet Container Directory Structure



MVC2 Pattern:web.xml

```
1<?xml version="1.0" encoding="UTF-8"?>
2<web-app xmlns="http://java.sun.com/xml/ns/javaee"
3    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4    xsi:schemaLocation="..." version="2.5">
5    <filter>
6        <filter-name>struts2</filter-name>
7        <filter-class>org.apache.struts2.dispatcher.ng.filter.
8                            StrutsPrepareAndExecuteFilter</filter-class>
9    </filter>
10
11   <filter-mapping>
12       <filter-name>struts2</filter-name>
13       <url-pattern>*.action</url-pattern>
14   </filter-mapping>
15
16   <!-- The Welcome File List -->
17   <welcome-file-list>
18       <welcome-file>startup.jsp</welcome-file>
19   </welcome-file-list>
20</web-app>
```

MVC2 Pattern:web.xml

MVC2 Pattern:struts.xml

```
1<?xml version="1.0" encoding="UTF-8"?>
2<! DOCTYPE struts  PUBLIC
3  "-// ApacheSofwareFoundaJon// DTD  Struts  Configuration2.0//EN"
4  "http://struts.apache.org/dtds/-struts2.0.dtd">
5<struts>
6     < constant name="struts.enable.DynamicMethodInvocation"
7               value="false" />
8     < constant name="struts.devMode"  value="false" />
9     < constant name="struts.custom.i18n.resources"
10        value="ApplicationResources" />
11     <package  name="default"  extends="-strutsdefault"
           namespace="/">
12        < action name="login"  class="Demo.LoginAction">
13            <- interceptorrefname="loggingStack"></-interceptorref>
14            < result name="success">Welcome.jsp</result>
15            <result  name="error">Login.jsp</result>
16        </action>
```

MVC2 Pattern:struts.xml

```
17    -     -                          -       --
18          <action name="customer"
19               class="Demo.CustomerAction">
20               <result name="success">SuccessCustomer.jsp</result>
21               <result name="input">Customer.jsp</result>
22          </action>
23     </package>
24</struts>
```

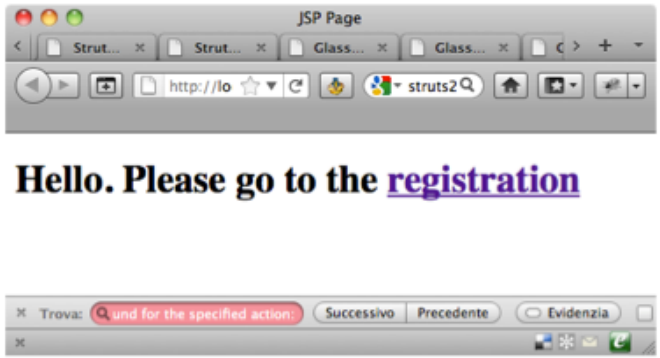
ApplicationResources.properties

```
label.username= Username
label.password= Password
label.login= Login
error.login= Invalid Username/Password. Please try again.

name= Name
age= Age
email= Email
telephone= Telephone
label.add.customer=Add Customer

errors.invalid=${getText(fieldName)} is invalid.
errors.required=${getText(fieldName)} is required.
errors.number=${getText(fieldName)} must be a number.
errors.range=${getText(fieldName)} is not in the range ${min} and ${max}
```

Startup.jsp



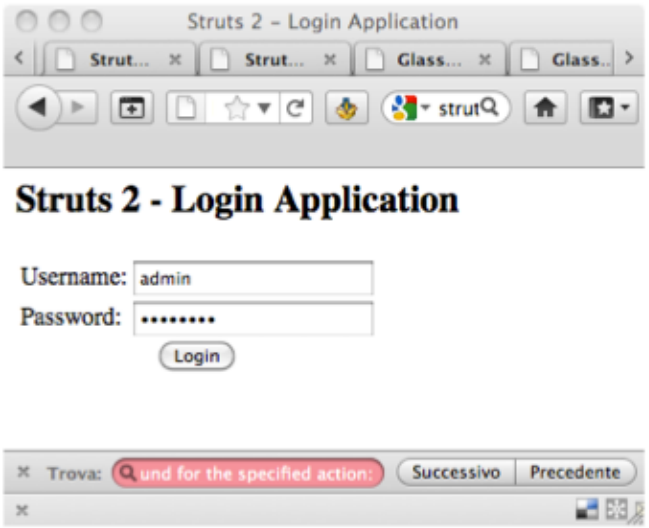

```
<%@page contentType="text/html"
    pageEncoding="UTF-8"%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">

<html>
    <head>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
        <title>JSP Page</title>
    </head>
    <body>
        <h1>Hello. Please go to the <a href="Login.jsp">registration</a></h1>
    </body>
</html>
```

Login.jsp

```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head>
<title>Struts 2 - Login Application </title>
</head>

<body>
<h2>Struts 2 - Login Application</h2>
<s:actionerror />
<s:form action="login" namespace="/" method="get">
   <s:textfield name="username" key="label.username" size="20" />
   <s:password name="password" key="label.password" size="20" />
   <s:submit method="execute" key="label.login" align="center" />
</s:form>
</body>
</html>
```



Data from the bundle

MVC2 Pattern:LoginAction.java

```
 1 package Demo;
 2
 3 import com.opensymphony.xwork2.ActionSupport;
 4
 5 public class LoginAction extends ActionSupport {
 6     private String username;
 7     private String password;
 8
 9     public String execute() {
10         if (this.username.equals("admin") &&
               this.password.equals("admin123")) {
11             return "success";
12         } else {
13             addActionError(getText("error.login"));
14             return "error";
15         }
16     }
17
18     public String getUsername() {
19         return username;
20     }
```

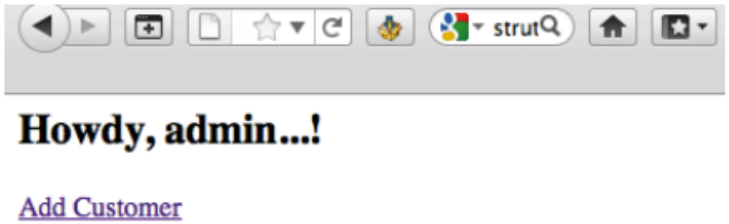
MVC2 Pattern:LoginAction.java

```
21
22      public void setUsername(String username) {
23          this.username = username;
24      }
25
26      public String getPassword() {
27          return password;
28      }
29
30      public void setPassword(String password) {
31          this.password = password;
32      }
33  }
```

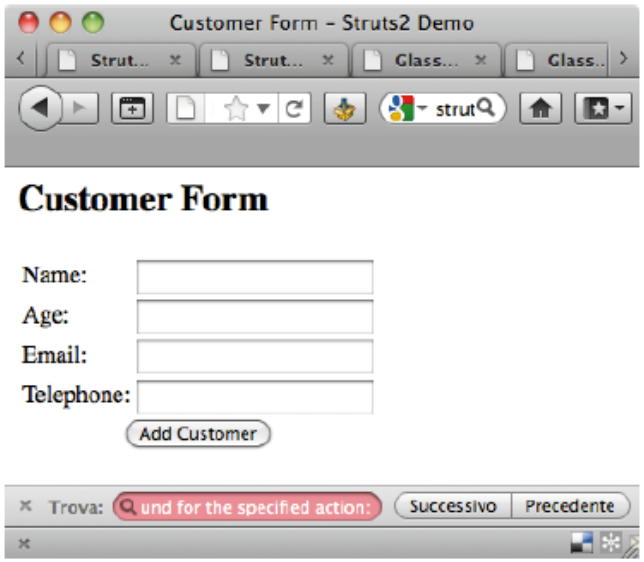
Welcome.jsp

```
<%@ page contentType="text/html;
charset=UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head>
<title>Welcome</title>
</head>

<body>
   <h2>Howdy, <s:property value="username" />...!
</h2>
   <s:a href="Customer.jsp">Add Customer</s:a>
</body>
</html>
```



Customer.jsp




```
<%@ page contentType="text/html; charset=UTF-8"%>
<%@ taglib prefix="s" uri="/struts-tags"%>
<html>
<head>
<title>Customer Form - Struts2 Demo </title>
</head>

<body>
<h2>Customer Form</h2>

<s:form action="customer" namespace="/" method="post" validate="false">
   <s:textfield name="name" key="name" size="20" />
   <s:textfield name="age" key="age" size="20" />
   <s:textfield name="email" key="email" size="20" />
   <s:textfield name="telephone" key="telephone" size="20" />
   <s:submit method="addCustomer" key="label.add.customer" align="center" />
</s:form>
</body>
</html>
```

MVC2 Pattern:CustomerAction.java

```
 1 package Demo;
 2
 3 import com.opensymphony.xwork2.ActionSupport;
 4
 5 public class CustomerAction extends ActionSupport {
 6     private String name;
 7     private Integer age;
 8     private String email;
 9     private String telephone;
10
11     public String addCustomer() {
12         return SUCCESS;
13     }
14
15     public String getName() {
16         return name;
17     }
```

MVC2 Pattern:CustomerAction.java

```
18
19      public void setName(String name) {
20          this.name = name;
21      }
22
23      public Integer getAge() {
24          return age;
25      }
26
27      public void setAge(Integer age) {this.age = age;
28
29          public String getEmail() {
30              return email;
31          }
32
33          public void setEmail(String email) {
34              this.email = email;
```

MVC2 Pattern:CustomerAction.java

```
35          }
36
37          public String getTelephone() {
38              return telephone;
39          }
40
41          public void setTelephone(String telephone) {
42              this.telephone = telephone;
43          }
44      }
45}
```

MVC2 Pattern:MyLoggingInterceptor.java

```
1  package Demo.Interceptors;
2
3  import com.opensymphony.xwork2.ActionInvocation;
4  import com.opensymphony.xwork2.interceptor.Interceptor;
5
6  public class MyLoggingInterceptor implements Interceptor {
7      private static final long serialVersionUID = 1L;
8
9      public String intercept(ActionInvocation invocation) throws
           Exception {
10         String className =
               invocation.getAction().getClass().getName();
11         long startTime = System.currentTimeMillis();
12         System.out.println("Before calling action: " + className);
13         String result = invocation.invoke();
14         long endTime = System.currentTimeMillis();
15         System.out.println("Afer calling action: " + className
16                             + " Time taken: " + (endTime - startTime)
                                   + " ms");
17         return result;
18     }
```

MVC2 Pattern:MyLoggingInterceptor.java

```
19
20      public void destroy() {
21          System.out.println("Destroying MyLoggingInterceptor...");
22      }
23
24      public void init() {
25          System.out.println("Initializing MyLoggingInterceptor...");
26      }
27  }
```