Architektur und Entwurfsmuster

Definition, Abgrenzung und ausgewählte Beispiele

W3L AG
info@W3L.de

2012

Zur Historie

- **Entwurf-durch-Routine**
  - Wiederverwendung bewährter Lösungen
- **Entwurf-durch-Innovation**
  - Erarbeitung neuer Lösungen
- **Erste Ansätze**
  - Gang of Four (GoF) 1995: Design Patterns: Elements of Reusable Object-Oriented Software
  - Erfahrungen aus GUI-Bereich; Sprachen C++ und Smalltalk
    - 23 Entwurfsmuster

Architektur- vs. Entwurfsmuster

- **Heute Unterscheidung zwischen Architektur- und Entwurfsmustern**

- **Architekturmuster**
  - „Architekturmuster *(architecture pattern)* beschreiben Systemstrukturen, die die Gesamtarchitektur eines Systems festlegen. Sie spezifizieren, wie Subsysteme zusammenarbeiten“
  - Auch: Basisarchitekturen, Architekturstil

- **Entwurfsmuster**
  - „Entwurfsmuster *(design patterns)* geben bewährte generische Lösungen für häufig wiederkehrende Entwurfsprobleme an, die in bestimmten Situationen auftreten. Sie legen die Struktur von Subsystemen fest.“
  - Auch: Mikroarchitektur

Klassifikation nach Zweck und Geltungsbereich

Zweck→ ↓Geltungsbereich	Erzeugendes Muster	Strukturelles Muster	Verhaltensmuster
Klasse	•Fabrikmethoden-Muster (factory method pattern)	•Adapterklasse	•Interpreter (interpreter) •Schablonenmethode (template)
Objekt	•Abstrakte Fabrik (abstract factory) •Erbauer (builder) •Prototyp (prototype) •Singleton (singleton)	•Adapter (adapter) •Das Brücken-Muster (bridge pattern) •Kompositum (composite) •Dekorierer (decorator) •Das Fassaden-Muster (facade pattern) •Fliegengewicht (flyweight) •Das Proxy-Muster (proxy pattern)	•Zuständigkeitskette (chain of responsibility) •Das Kommando-Muster (command pattern) •Iterator (iterator) •Vermittler (mediator) •Memento (memento) •Das Beobachter-Muster (observer pattern) •Zustand (state) •Das Strategie-Muster (strategy pattern) •Besucher (visitor)

Klassifikation nach Anwendungs- und Verteilungsarten

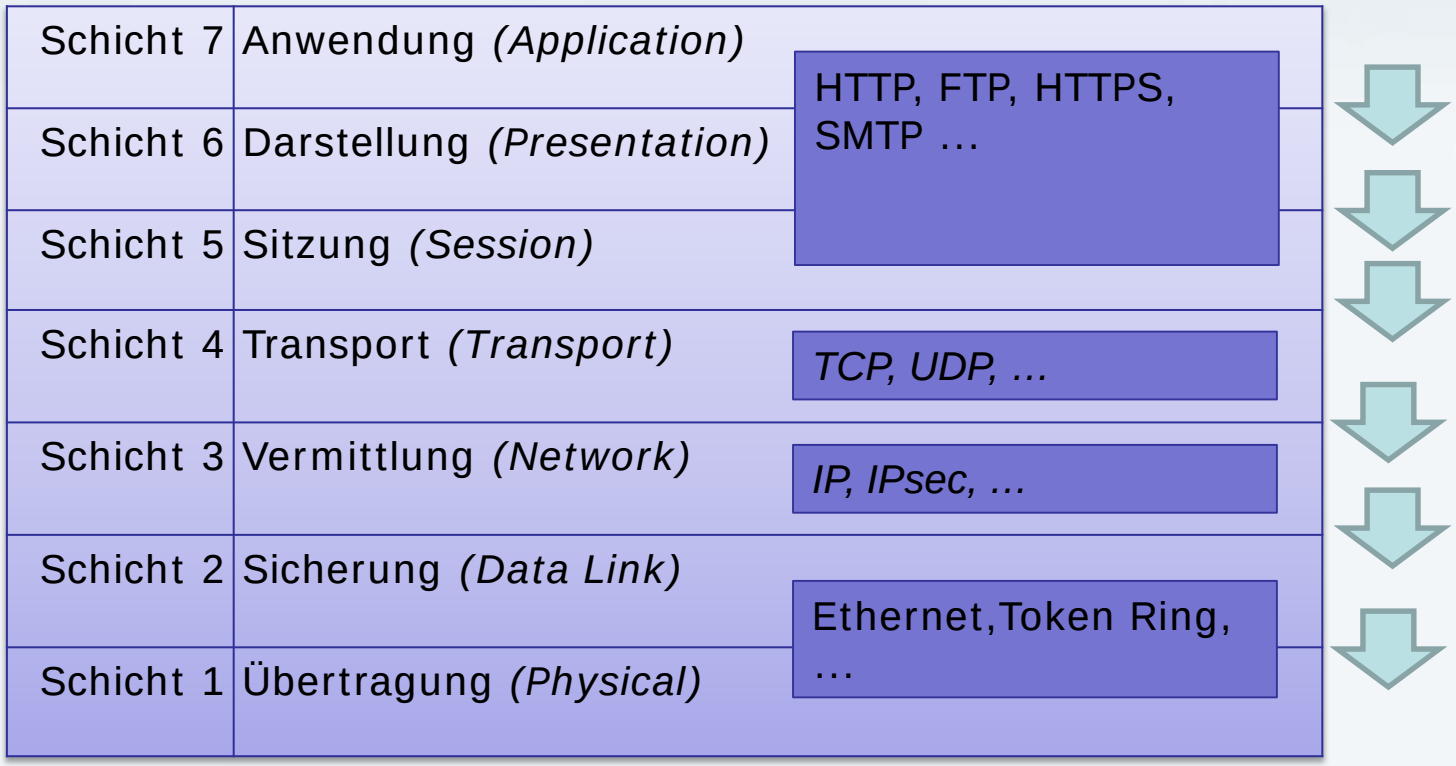
- **Muster für verteilte Systeme**
  - Starke, Gernot: *Pattern-Oriented Software Architecture Vol. 4: A Pattern Language for Distributed Computing*; John Wiley & Sons
    - über 250 Muster
  - Buschmann, F., Henney, K., Schmidt, D.: *Remoting Patterns - Patterns for Enterprise, Realtime and Internet Middleware*; John Wiley & Sons
- **Muster für sicherheitsrelevante Systeme**
  - Authentifizierung, Autorisierung und Vertraulichkeit
  - Voelter, M., Kircher, M., Zdun, U.: *Security Patterns: Integrating Security and Systems Engineering*; John Wiley & Sons
  - Schumacher, Markus, Fernandez-Buglioni, Eduardo, Hybertson, Duane, Buschmann , Frank, Sommerlad, Peter: *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*; Prentice Hall

Klassifikation nach Anwendungs- und Verteilungsarten

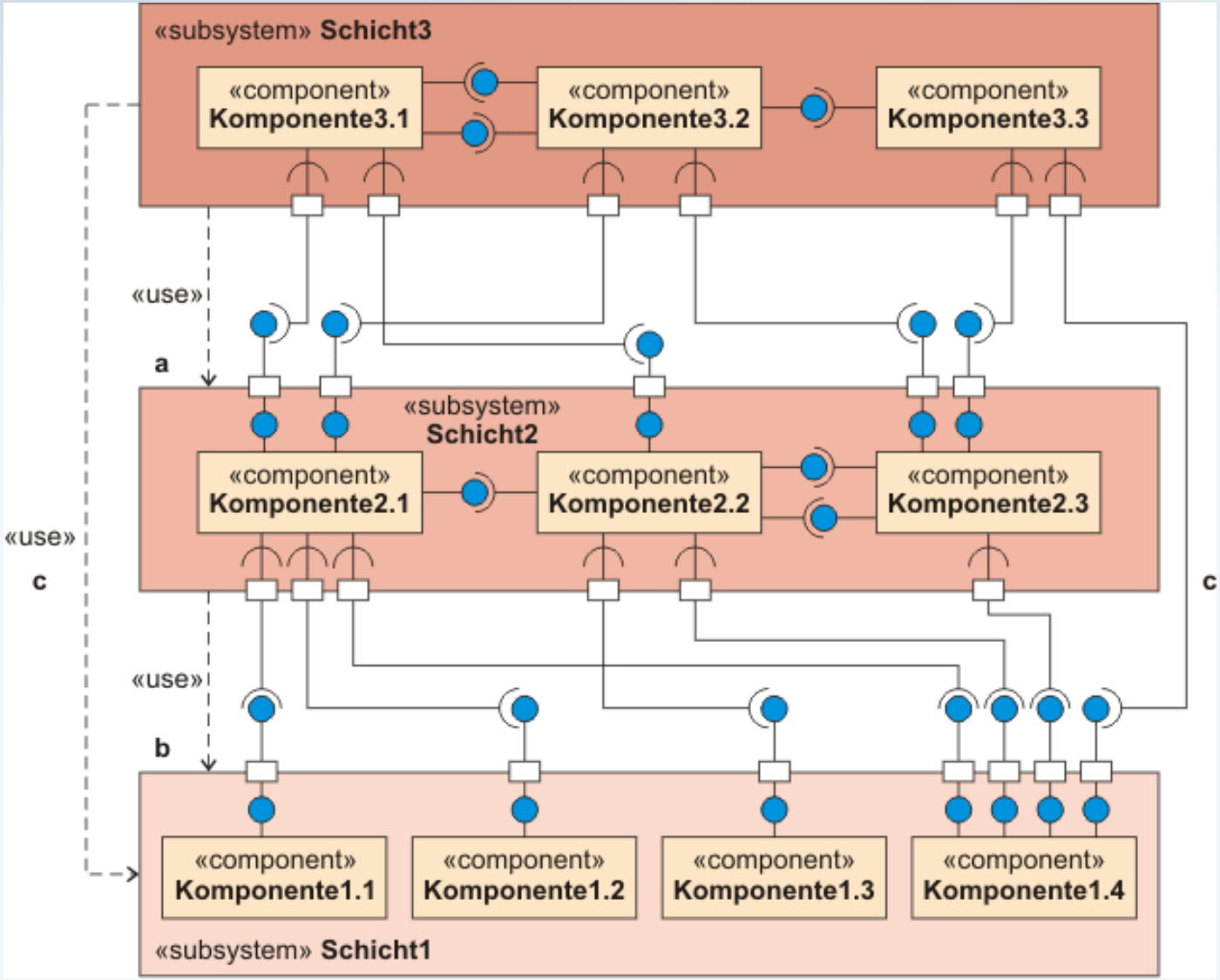
- **Muster für fehlertolerante Systeme**
  - Steel, Christopher: *Robust Communications Software: Extreme Availability, Reliability, and Scalability for Carrier-Grade Systems*; John Wiley & Sons
- **Muster für eingebettete Systeme**
  - Noble, J., Weir , C.: *Design Patterns for Distributed Real-Time Embedded Systems*; Springer
  - Buschmann, F., Henney, K., Schmidt, D.: *Small Memory Software: Patterns for Systems with Limited Memory*; Addison-Wesley

Schichten-Muster *(layers pattern)* - Beispiele

- **Beispiel: ISO/OSI-7-Schichtenmodell: Schichten-Architektur mit linearer Ordnung**

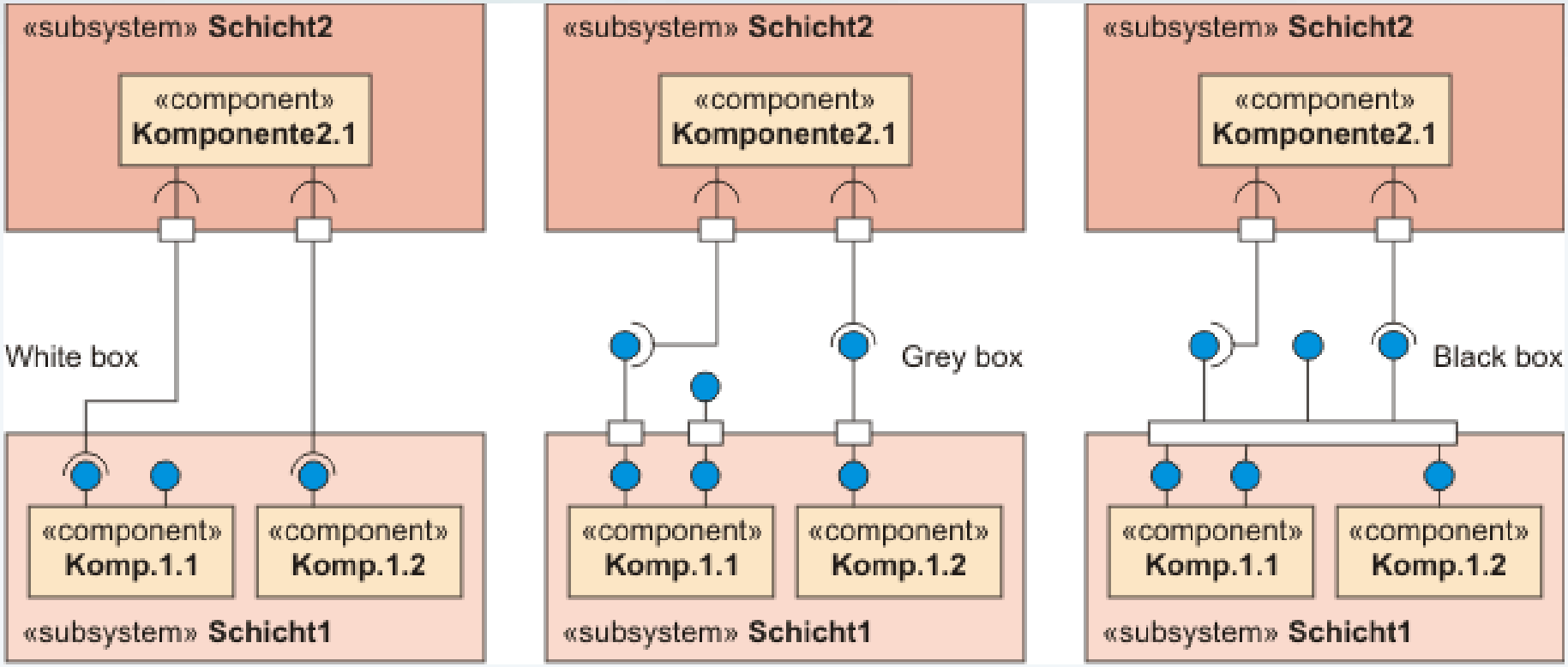


Architekturmuster: Schichtenmuster



Architekturmuster: Schichtenmuster

- **White box, Grey box, Black box**



Zusammenhang: Architektur- und Entwurfsmuster

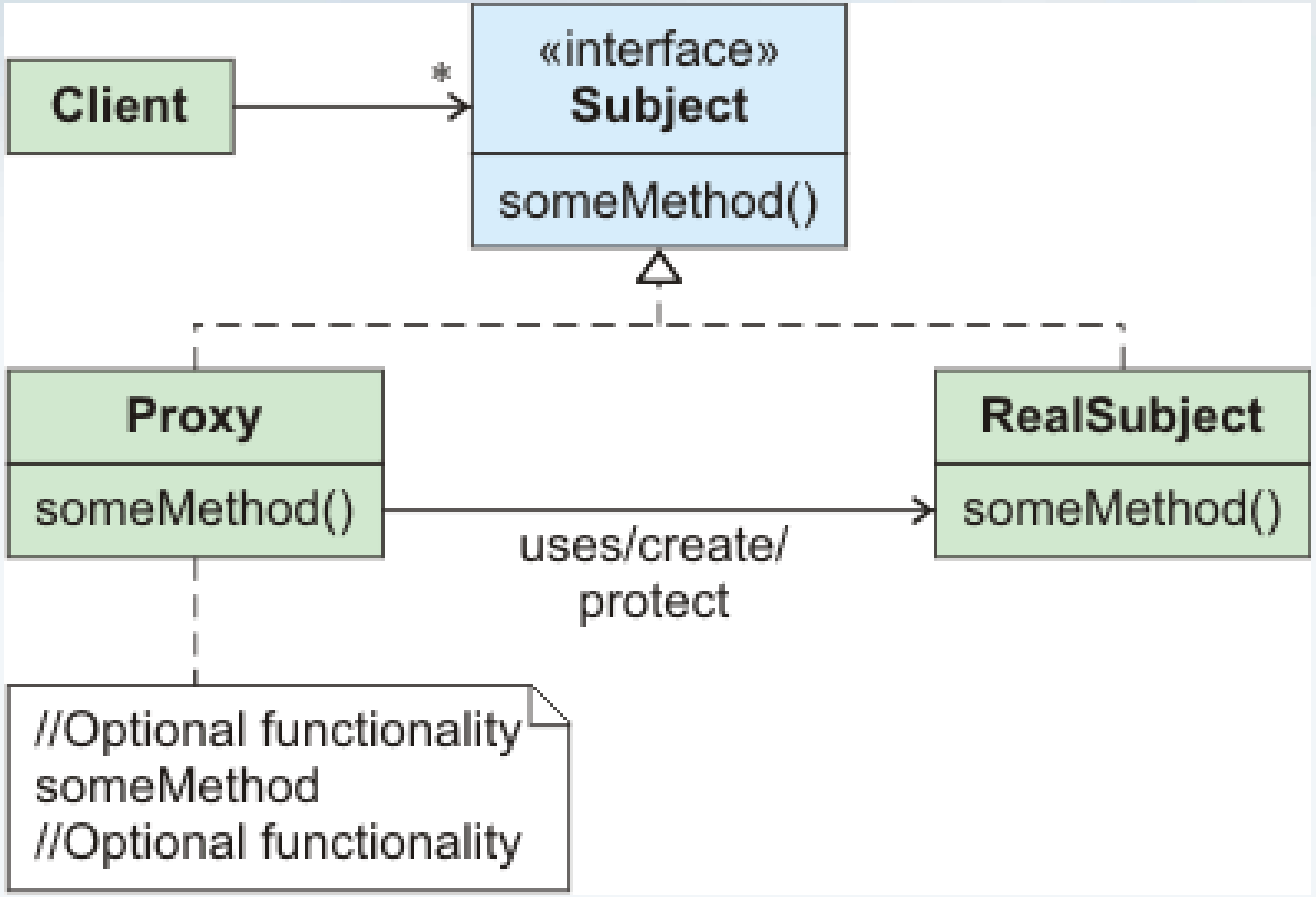
- **Schichtenarchitektur: Entkoppeln der Schichten durch Entwurfsmuster**
  - Black-box-Ansatz
    - Fassaden-Muster
  - Abhängigkeiten nur „von oben nach unten“
    - Callback, Beobachter-Muster
  - Entkoppeln von konkreten Funktionen
    - Kommandomuster
  - Eliminierung von „Tangling“ und „Scattering“ aufgrund von querschneidenden Belangen *(Cross Cutting Concerns)*
    - Proxy-Muster
- **Drei-Schichten-Architektur**
  - GUI, Fachkonzept- und Datenhaltungsschicht
    - MVC, Beobachter-Muster, Callback

Entwurfsmuster: Proxy-Muster

- **Proxy-Muster *(proxy pattern)***
  - Auch: Stellvertreter-Muster *(placeholder pattern, surrogate pattern)*

- **Objekt soll bzw. darf nicht direkt angesprochen werden**
  - Sicherheit
    - Interception, Schutz-Proxy
  - Verteilung
    - Remote-Proxy
  - Performanz
    - Lazy-Loading, virtueller Proxy
    - Cache-Proxy
  - Nebenläufigkeit
    - Synchronisations-Proxy
    - z.B. Umsetzung Copy-On-Write

Entwurfsmuster: Proxy-Muster



Entwurfsmuster: Proxy-Muster

- **Beispiel: Sicherheit**

```
class Kunde
{
        private String name;

        public String getName()
        {
                if (!Application.CurrentUser.HasAccess(this,“name“))
                        throw new SecurityException();
                return name;
        }
        …
}
```

Entwurfsmuster: Proxy-Muster

- **Beispiel: Sicherheit**

```
interface IKunde
{
        public String getName();
}

class Kunde implements IKunde
{
        private String name;

        public String getName()
        {
                return name;
        }
        ...
}
```

Entwurfsmuster: Proxy-Muster

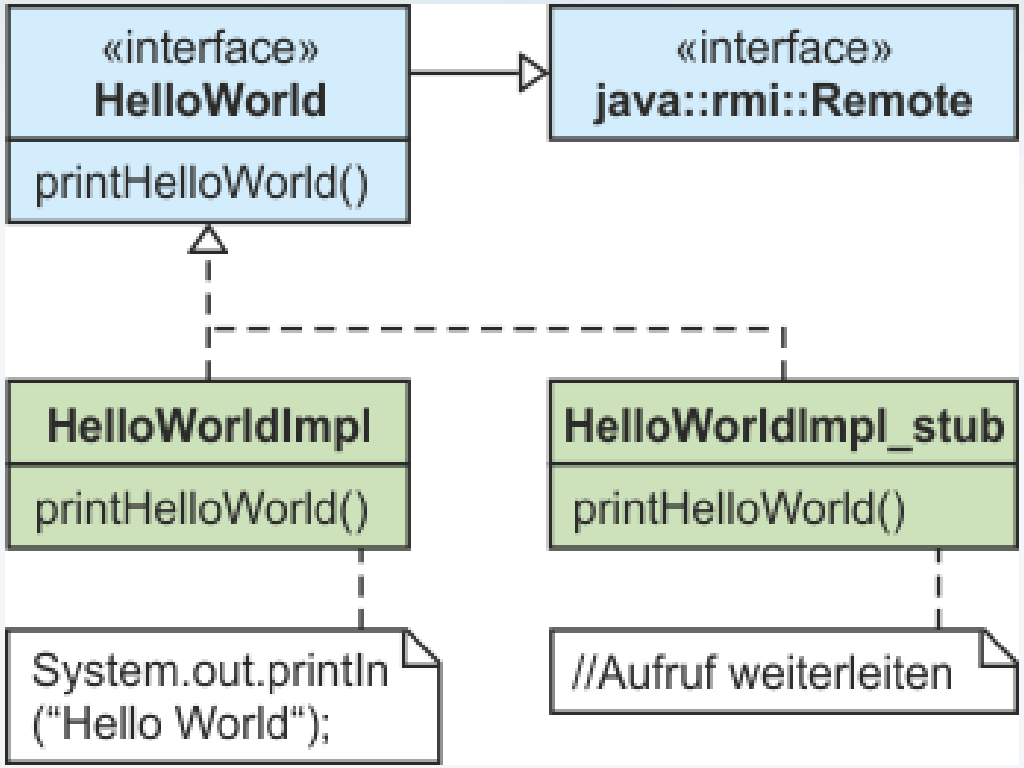
- **Beispiel: Sicherheit**

```
class KundeProxy implements IKunde
{
        private Kunde kunde;

        public String getName()
        {
                if (!Application.CurrentUser.HasAccess(this,“name“))
                        throw new SecurityException();
                return kunde.getName();
        }
        …
}
```

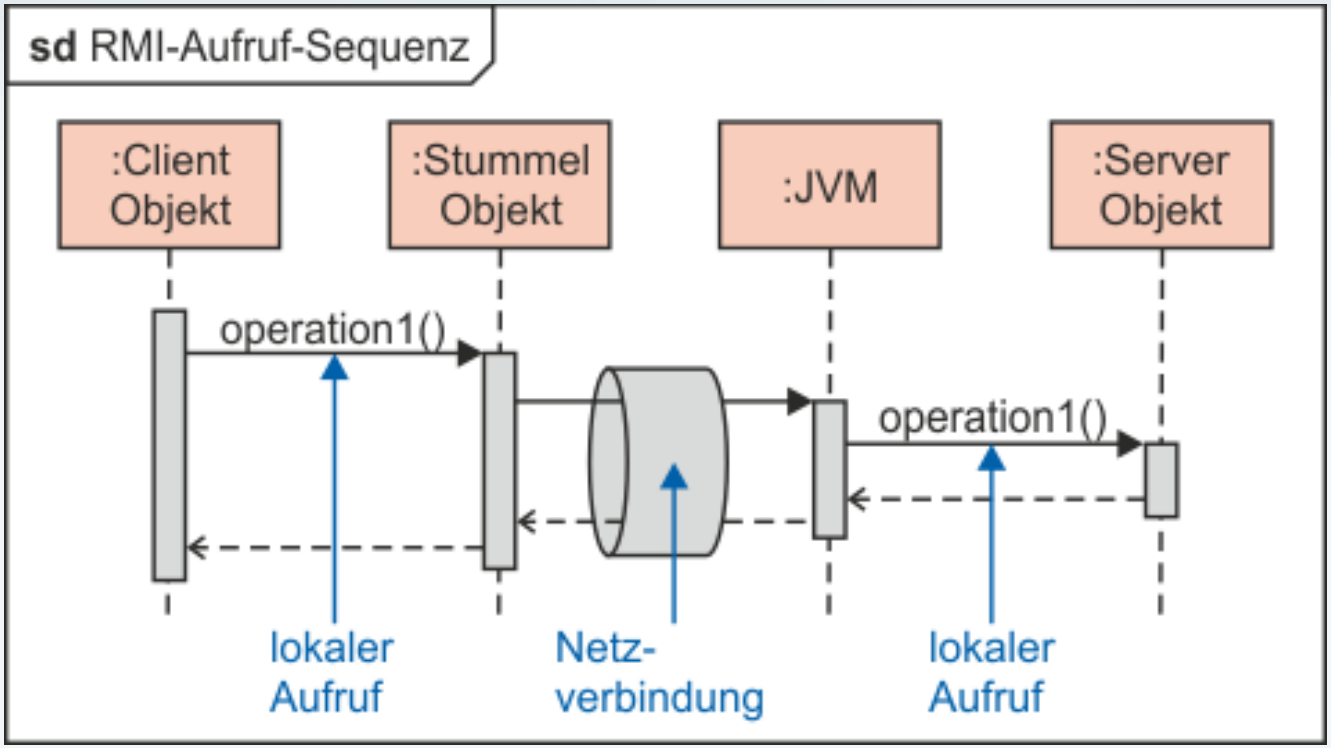
Entwurfsmuster: Proxy-Muster

- **Beispiel: Transparenter Remote-Zugriff, Java RMI**



Entwurfsmuster: Proxy-Muster

- **Beispiel: Transparenter Remote-Zugriff, Java RMI**



Entwurfsmuster: Proxy-Muster

- **Proxy wird häufig generiert z.B. auf Basis von Deklarationen**
  - Beispiele
    - RMI: rmic -v1.2 HelloWorldImpl
    - Annotations (Java); Attribute (.Net)
    - Externe Konfigurationsdateien (z.B. Hibernate XML-Datei(en))

```
class Kunde implements IKunde
{
        private String name;

        @Security(UserAccessLevel = true)
        public getName()
        {
                return name;
        }
        ...
}
```

Kommando-Muster *(command pattern)*

- **Kommando-Muster *(command pattern)***
  - Ojektbasiertes Verhaltensmuster *(behavioral pattern)*
  - Auch: Befehls-, Aktions- oder Transaktions-Muster

- **Grundidee**
  - Entkopplung Sender (Auslöser, *invoker*) vom Empfänger *(receiver)*
  - Sender ruft Methode auf
  - Empfänger reagiert auf Kommando/Methodenaufruf
  - Ein Kommando ist ein Objekt
    - z.B. Implementierung von *(undo)* und *(redo)* möglich

Kommando-Muster *(command pattern)*

- **Weitere Anwendungsbeispiele**
  - Aufrufe sollen asynchron ausgeführt werden (z.B. Speicherung von Kommandos in Warteschlange)
  - Ausgeführte Kommandos sollen protokolliert werden (Logbuch)
  - Registrierung eines Kommandos bei unterschiedlichen Stellen (z.B. Menü, Button)
  - Ermöglichung einer Makro-Aufzeichnung

Kommando-Muster *(command pattern)*

- **Beispiel**

```
//Receiver
public class MyInteger
{
        private int value;

        public MyInteger(int v){ value = v; }

        public void plus(int s){ value = value + s; }
        public void minus(int s){ value = value - s; }
        public int getValue(){ return value; }
        public int setValue(int v){ value = v; }
}
```

Kommando-Muster *(command pattern)*

Beispiel

```
//Command
public interface Operation
{
        void do();
        void undo();
}
```

```
//ConcreteCommand: s1 wird um s2 erhoeht
public class Plus implements Operation
{
        private MyInteger s1, s2;

        public Plus(MyInteger s1, MyInteger s2)
        {
                this.s1 = s1;
                this.s2 = s2;
        }

        public void do()
        {
                s1.plus(s2.getValue());
        }

        public void undo()
        {
                s1.minus(s2.getValue());
        }
}
```

Kommando-Muster *(command pattern)*

- **Beispiel**

```
//Invoker
public class Calculator
{
        Stack<Operation> ops = new Stack<Operation>();

        public void invoke(Operation op)
        {
                ops.push(op);
                op.do();
        }

        public void undo()
        {
                //letzte Operation rueckgaengig machen
                Operation op = ops.pop();
                op.undo();
        }
}
```

Kommando-Muster *(command pattern)*

- **Beispiel**

```
//Client
public static void main(String args[])
{
        Calculator calc = new Calculator();

        MyInteger s1 = new MyInteger(5);
        MyInteger s2 = new MyInteger(4);
        Plus p = new Plus(s1, s2);

        //ab hier keine Abhaengigkeiten mehr zu MyInteger
        calc.invoke(p); //s1 == 9

        calc.invoke(p); //s1 == 13

        ...

        //letzte Operation rueckgaengig machen
        calc.undo(); //s1 == 9
}
```

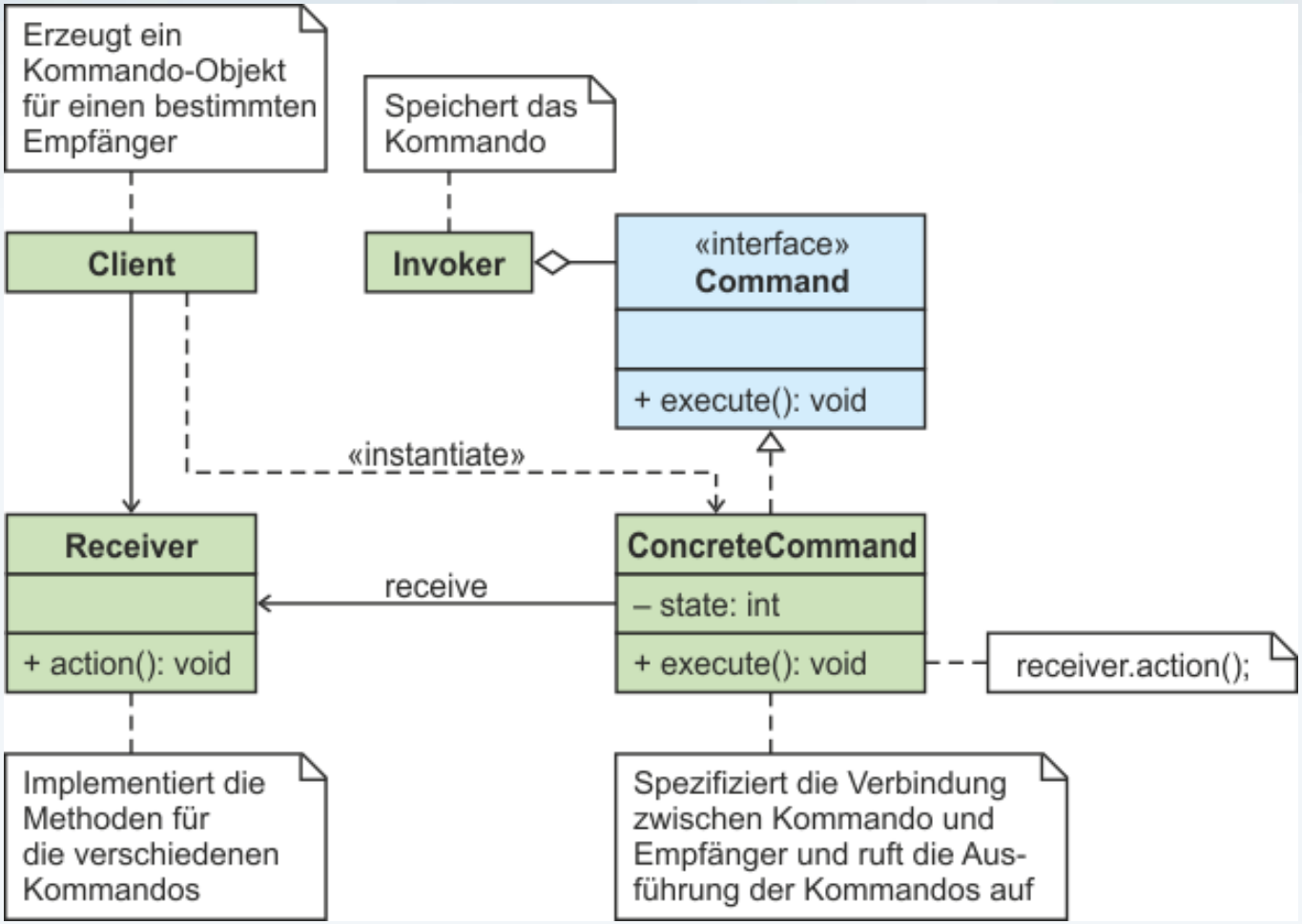
Kommando-Muster *(command pattern)*

- **Weitere Beispiele**
  - Unit-Tests (z. B. JUnit) werden als Kommandos gekapselt
  - Java: Paket javax.swing.undo: Unterstützung von Undo/Redo-Kommandos

Kommando-Muster	**Swing-Bibliothek**
Command	AbstractAction
ConcreteCommand	ConcreteAction
Unterklasse von Command	Unterklasse von AbstractAction
execute()	actionPerformed()

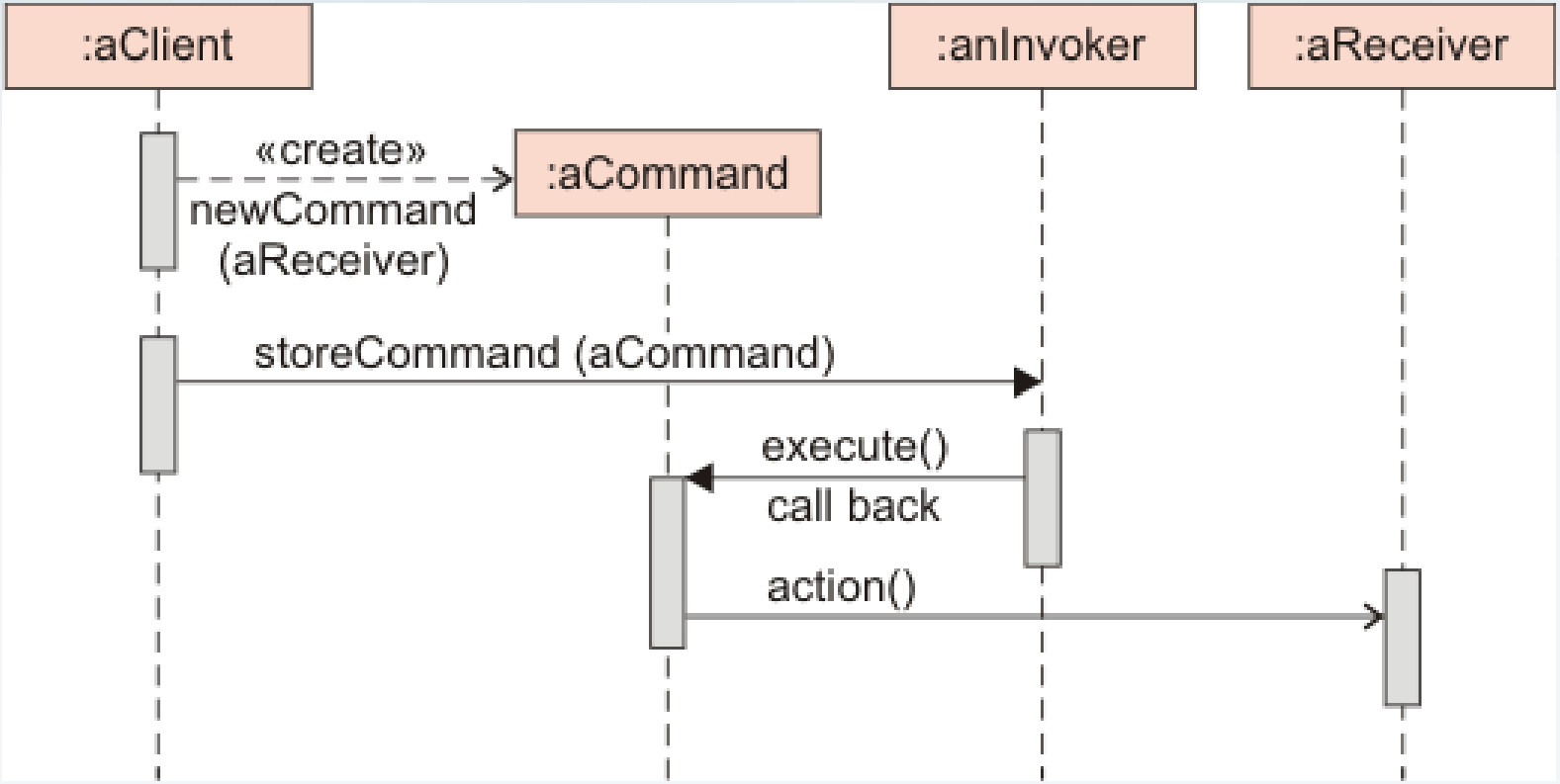
Kommando-Muster *(command pattern)*

Struktur



Kommando-Muster *(command pattern)*

- **Dynamik**



Kommando-Muster *(command pattern)*

- **Hinweise zur Implementierung**
  - Keine Intelligenz
    - Kommando delegiert Ausführung an Empfänger
  - Keine Delegation
    - Kommando implementiert alle Funktionalitäten selbst
  - Keine Delegation wenn
    - Kommandos unabhängig von existierenden Klassen sein sollen
    - kein passender Empfänger existiert

Kommando-Muster *(command pattern)*

- **Vorteile**
  - Aufrufer müssen nicht „wissen“, welche Objekte die Kommandos letztendlich ausführen
    - Aufrufer sind nur von der Schnittstelle Command abhängig
  - Kommandoobjekte können manipuliert und erweitert werden, wie jedes andere Objekt auch
  - Kommandoobjekte können leicht hinzugefügt werden, da keine existierenden Klassen geändert werden müssen
  - Das *Composite*-Muster kann verwendet werden, um existierende Kommandos zu einem neuen Kommando zu gruppieren
    - Übung: Kommando- und Composite-Muster

Kommando-Muster *(command pattern)*

- **Nachteile**
  - Die Systemstruktur ist schwieriger zu verstehen.
  - Es werden viele zusätzliche kleine Klassen benötigt.

- **Zusammenhänge**
  - Unterstützung der Weiterentwickelbarkeit und Erweiterbarkeit
  - Bessere Benutzbarkeit durch Undo- und Redo-Funktionalität
  - Kommandoobjekte sind ein objektorientierter Ersatz für Callbacks
  - Förderung des Prinzips der Abstraktion
  - Reduzierung der Sichtbarkeit

Neues

„Das Neue daran ist nicht gut, und das Gute daran ist nicht neu.“
(Johann Heinrich Voß (1751-1826), dt. Dichter)

Vielen Dank!

Literatur

- **Balzert, H.: *Lehrbuch der Softwaretechnik: Entwurf, Implementierung, Installation und Betrieb*; 3. Aufl., Spektrum Akademischer Verlag 2011**

Inhouse-Schulungen

Wir bieten Inhouse-Schulungen und Beratung durch unsere IT-Experten und –Berater.

Schulungsthemen

- Softwarearchitektur (OOD)
- Requirements Engineering (OOA)
- Nebenläufige & verteilte Programmierung

Gerne konzipieren wir auch eine individuelle Schulung zu Ihren Fragestellungen.

Sprechen Sie uns an!
Tel. 0231/61 804-0, info@W3L.de

W3L-Akademie

In Zusammenarbeit mit der Fachhochschule Dortmund bieten wir

zwei Online-Studiengänge

- B.Sc. Web- und Medieninformatik
- B.Sc. Wirtschaftsinformatik

und 7 Weiterbildungen im IT-Bereich an.

Besuchen Sie unsere Akademie!
http://Akademie.W3L.de