Spring + Design Patterns

Michael Inden

Freiberuflicher Consultant und Trainer

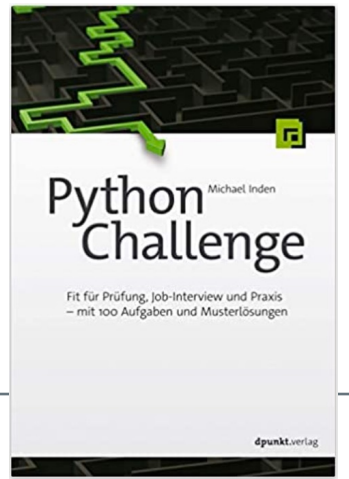
Speaker Intro

- Michael Inden, Jahrgang 1971
- Diplom-Informatiker, C.v.O. Uni Oldenburg

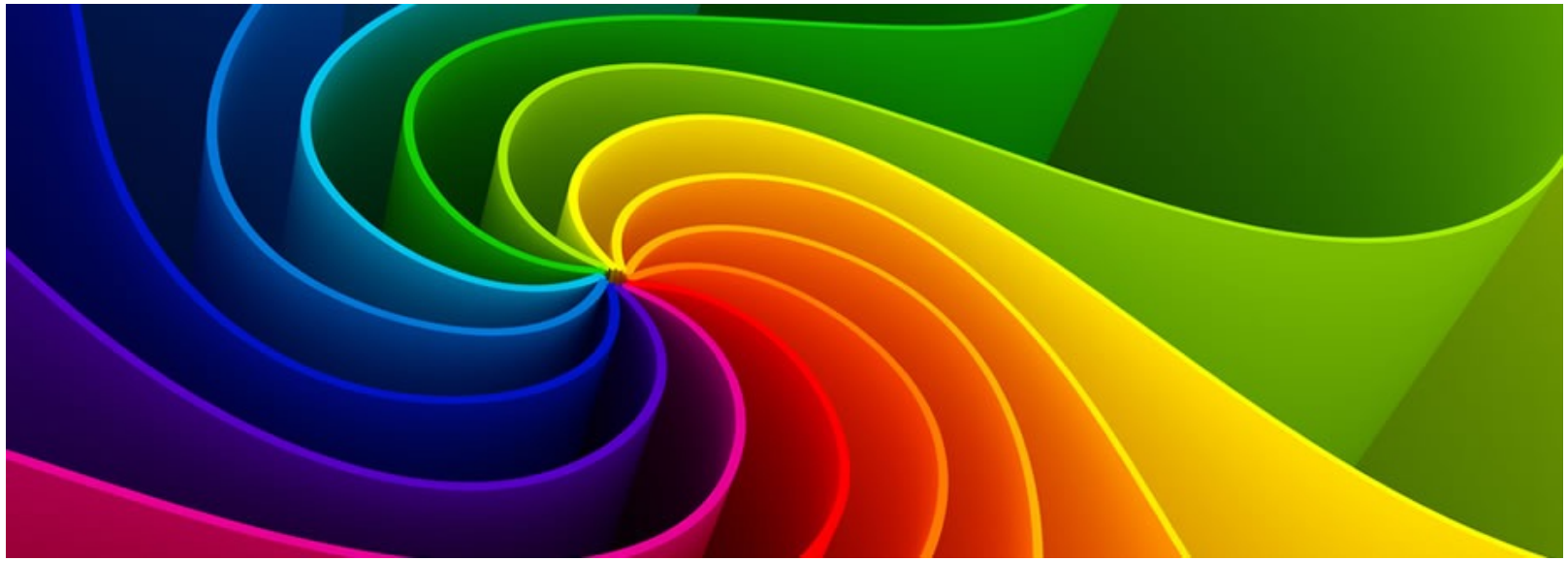
- ~8 ¼ Jahre **SSE** bei Heidelberger Druckmaschinen AG in Kiel
- ~6 ¾ Jahre **TPL, SA** bei IVU Traffic Technologies AG in Aachen
- ~4 ¼ Jahre **LSA / Trainer** bei Zühlke Engineering AG in Zürich
- ~3 Jahre **TL / CTO** bei Direct Mail Informatics / ASMIQ in Zürich
- **Freiberuflicher Consultant, Trainer und Konferenz-Speaker**
- Autor und Gutachter beim dpunkt.verlag

E-Mail: michael_inden@hotmail.com
Blog: https://jaxenter.de/author/minden
https://www.wearedevelopers.com/magazine/java-records

Kurse: Bitte sprecht mich an!



Design Prinzipien und Patterns in Spring



SOLID

Software Development is not a Jenga game

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

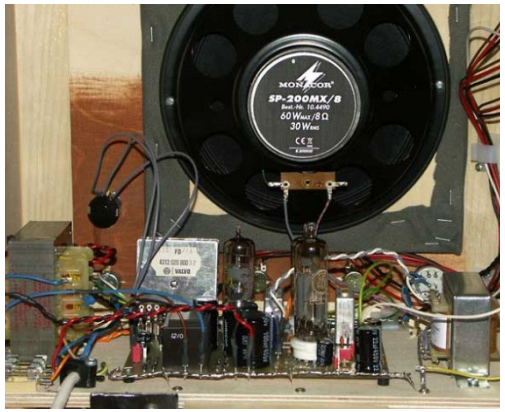
SOLID

- **S** R P – Single Responsibility Principle
- **O** C P – Open Closed Principle
- **L** S P – Liskov Substitution Principle
- I S P – Interface Segregation Principle
- **D** I P – Dependency Inversion Principle

Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

Open Closed Principle

Open for Modifications



Closed for Extensions

Open for Extensions. Closed for Modifications

O C P – Open Closed Principle

- leichte Erweiterbarkeit
- korrekte Kapselung sowie Trennung von Zuständigkeiten.
- *sollte sich eine Klasse nach ihrer Fertigstellung nur noch dann ändern müssen, wenn komplett neue Anforderungen oder Funktionalitäten zu integrieren sind oder aber Fehler korrigiert werden müssen*.

Open Closed Principle - Example

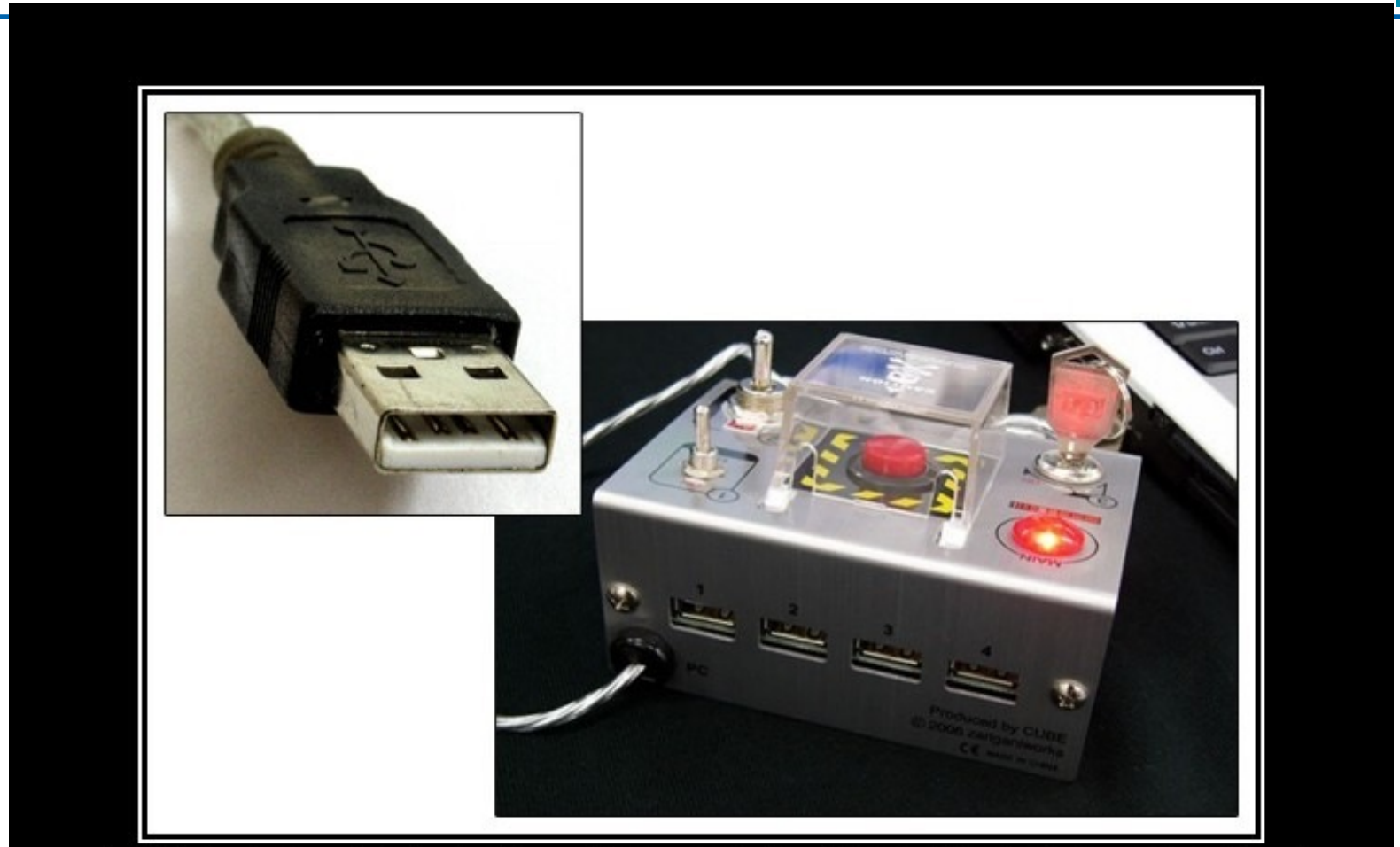
The following code does not conform to OCP:

```
public class Drawer {

 public void drawAll(List<Shape> shapes) {

   for (Shape shape : shapes) {

       if (shape instanceof Circle) {
              drawCircle((Circle) shape);
       }

       if (shape instanceof Square) {
              drawSquare((Square) shape);
       }

   }

 }

 ...
```

Open Closed Principle - Example

The following code does **conform** to OCP:

```
public class Drawer {
 public void drawAll(List<Shape> shapes) {
   for (Shape shape : shapes) {
       shape.draw()
   }
 }
 ...
```



Quelle: http://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/

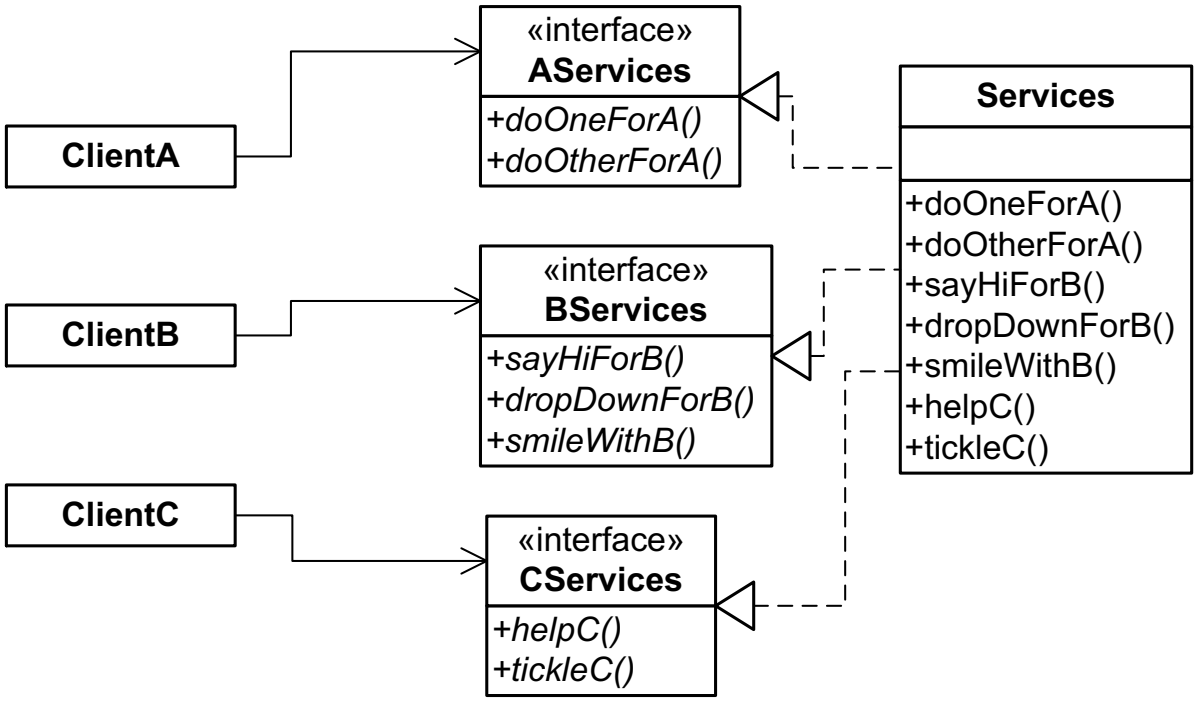
I S P – Interface Segregation Principle

- ***Erstelle möglichst spezifische, auf die jeweilige Aufgabe oder auf ihn als Klienten zugeschnittene Schnittstelle***
- Oftmals sieht man in der Praxis eher zu breite oder zu unspezifische Interfaces, die folglich fast immer auch Funktionalität anbieten, die ein Klient nicht benötigt, etwa wie folgt:

```
interface IUniversalFileCustomerAndPizzaService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;

    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);

    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

Interface Segregation Principle

Several client specific interfaces are better than one single, general interface



not only one interface for all clients

one interface per kind of client

I S P – Interface Segregation Principle

- der Entwurf einer gelungenen Schnittstelle ist gar nicht so leicht
- Es gilt, die **»richtige« Granularität** zu finden.
- Das benötigt etwas Erfahrung, Fingerspitzengefühl und auch ein wenig Ausprobieren – insbesondere auch eine Betrachtung aus Sicht möglicher Nutzer.

```
interface IFileService
{
    void scanDisk(final Drive drive) throws IOException;
    boolean rename(final File fileToRename,
                   final String newName) throws IOException;
}

interface ICustomerService
{
    Customer findCustomerByName(final String name);
    Iterable<Customer> getAllCustomers(final FilterCondition filterCondition);
}

interface IPizzaService
{
    boolean orderPizza(final long customerId, final Pizza pizza);
}
```

Fabrikmethode
(Factory Method)

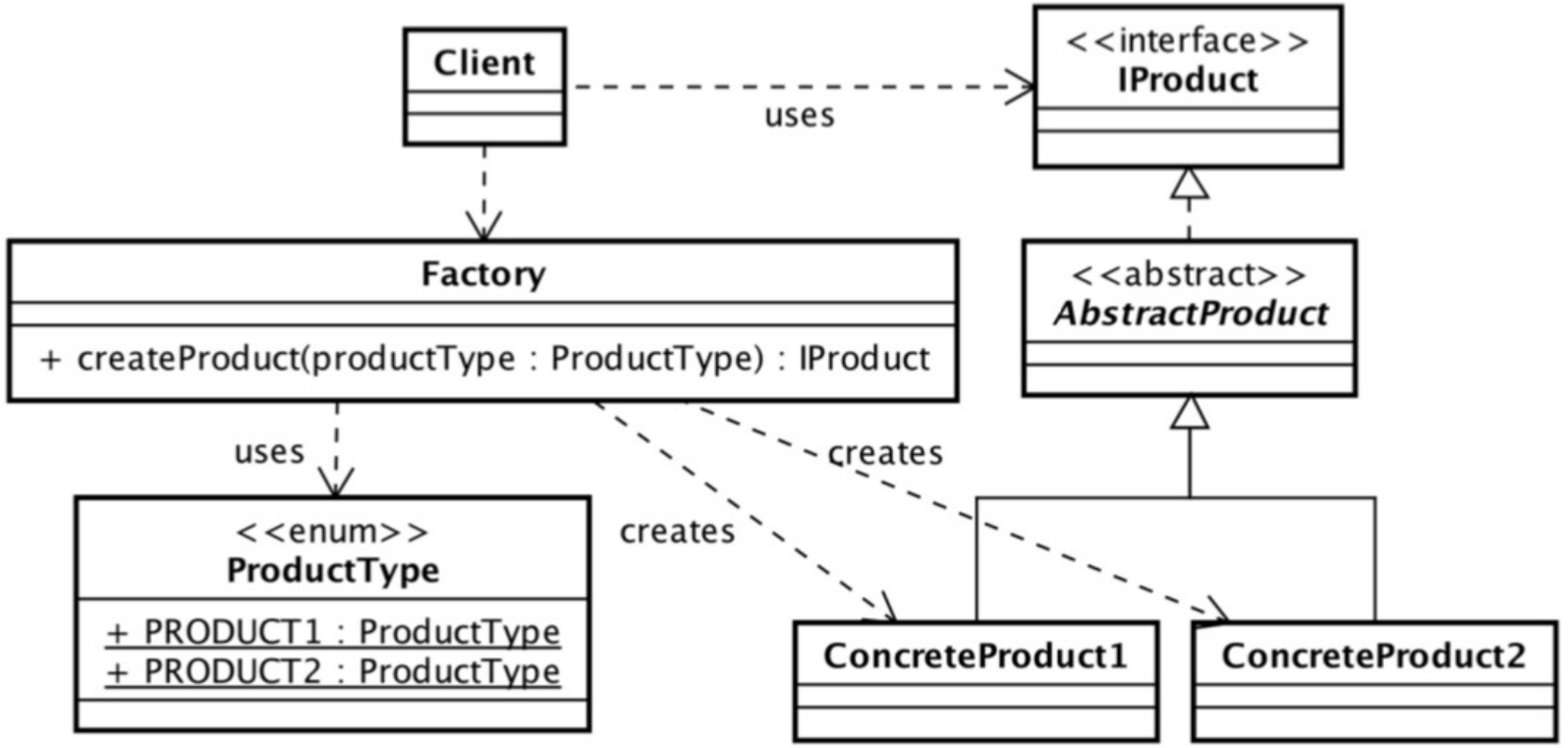
Motivation und Kurzbeschreibung Fabrikmethode

- Die Idee hinter dem Muster **FABRIKMETHODE** ist ähnlich einer **Produktion** in einer realen **Fabrik**, die aufgrund einer **Bestellung** gewünschte **Dinge produzieren** kann.

- Bei einer solchen **Bestellung** sind **lediglich** die **Artikelnummern** oder **Bezeichnungen** der Teile, nicht aber die konkreten Ausprägungen und Realisierungen bekannt.

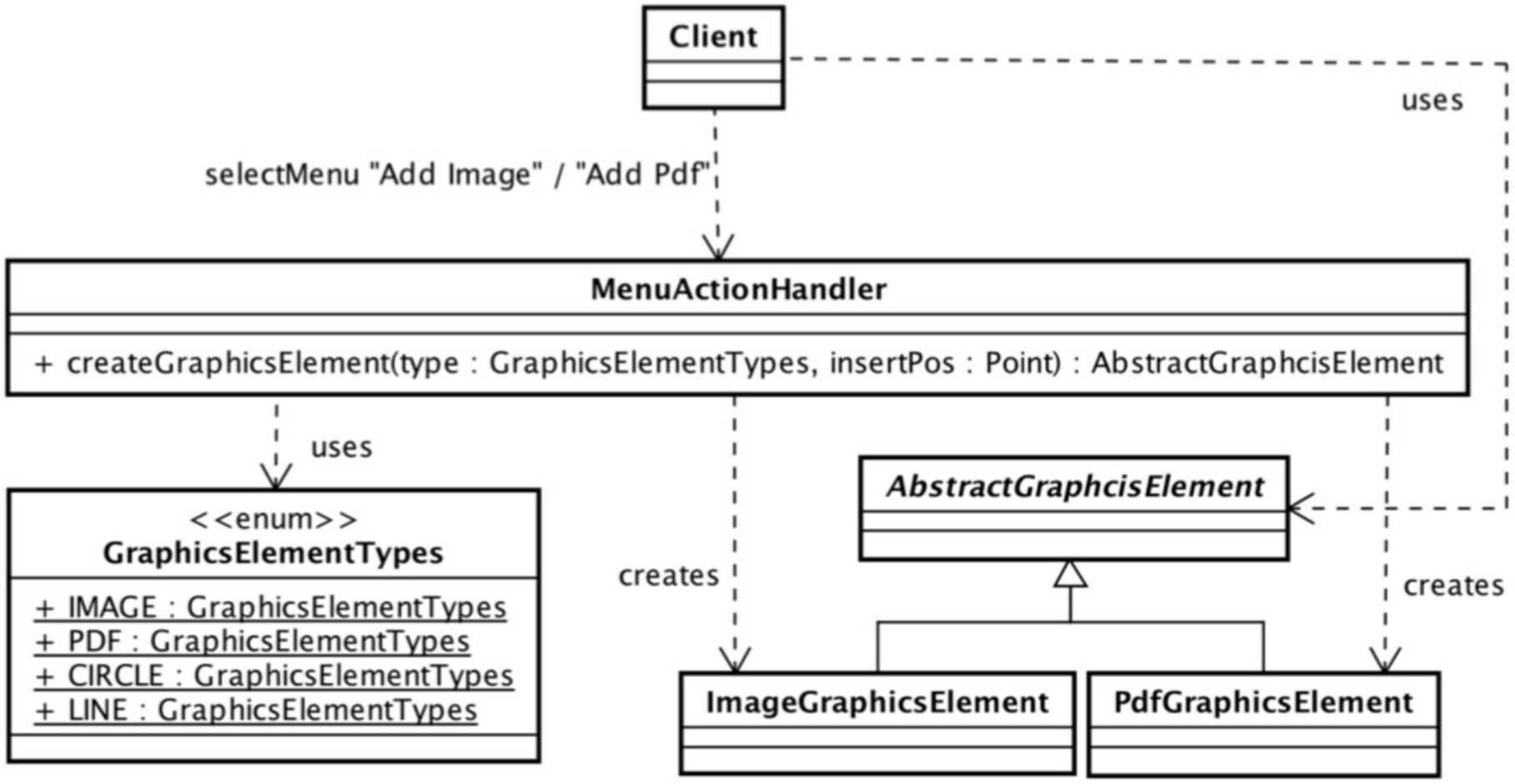
- Es findet eine **Kapselung** der **Objekterzeugung** statt: Objekte werden **nicht direkt per Konstruktoraufruf** erzeugt, sondern dieser Vorgang wird an eine spezielle Methode oder ein spezielles Objekt, eine sogenannte Fabrik, delegiert. Die dort definierten Fabrikmethoden erzeugen Objekte verschiedenen Typs.

UML



- Es wird eine Fabrikklasse mit einer Methode `createProduct(ProductType)` zum Erzeugen von Objekten definiert. Dort wird anhand eines Parameters entschieden, welcher konkrete Typ erzeugt wird.

Anwendungsbeispiel



Bewertung Fabrikmethode

\+ **Lesbarkeit** – Die Details des Konstruktionsprozesses werden versteckt => mehr Lesbarkeit

\+ **Abstraktion** – Der konkret erzeugte Objekttyp kann vor dem Aufrufer versteckt werden, indem lediglich eine Referenz auf ein Interface oder eine abstrakte Klasse des erzeugten Objekttyps zurückgegeben wird. Stärkere Kapselung und losere Kopplung.

\+ **Kapselung** – Die Kapselung verstärkt sich: Ein Klient nutzt lediglich eine Schnittstelle zur Erzeugung, wodurch die konkrete Realisierung einer Fabrikklasse ausgetauscht werden kann.

\+ **Konstruktionssicherheit** – Durch Konsistenzprüfung möglich können vollständig initialisierte Objekte erzeugt / garantiert werden bzw. Fehler entsprechend korrigiert oder behandelt werden.

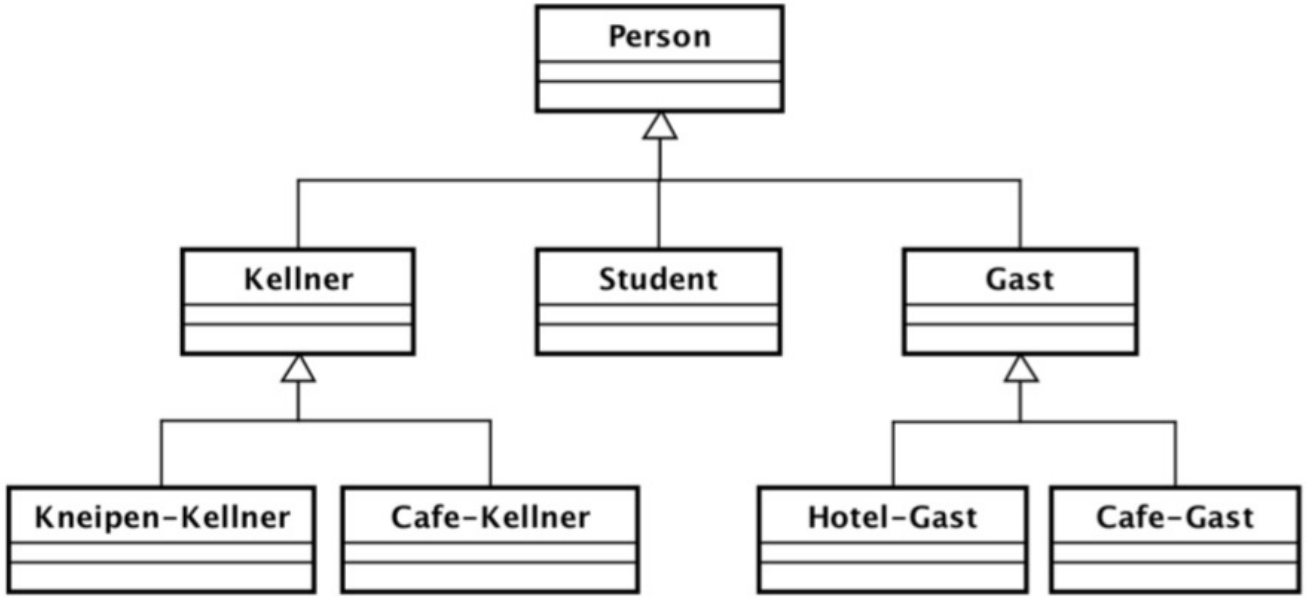
o **Mehraufwand** – Es entsteht ein wenig mehr Sourcecode.

o **Mehr Komplexität** – Geringfügig mehr Komplexität durch die zu realisierenden Klassen

Dekorierer (Decorator)

Probleme mit Vererbung

- **Selbst beim Einhalten der „is-a“-Eigenschaft kann es zu Problemen kommen:**

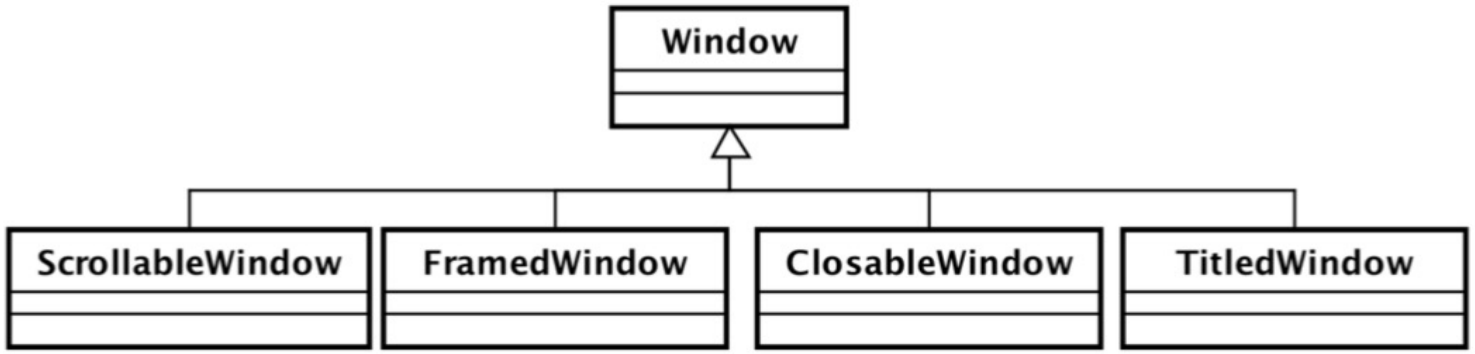


- **Sind alles Spezialisierungen**

- **ABER: Zielen mehr auf eine Rolle/Aufgabe, die zeitweilig ausgeübt wird**
  - **„is-a-role-played-by“ => Delegation**
  - **„can-act-like“ => Interface**

Probleme mit Vererbung

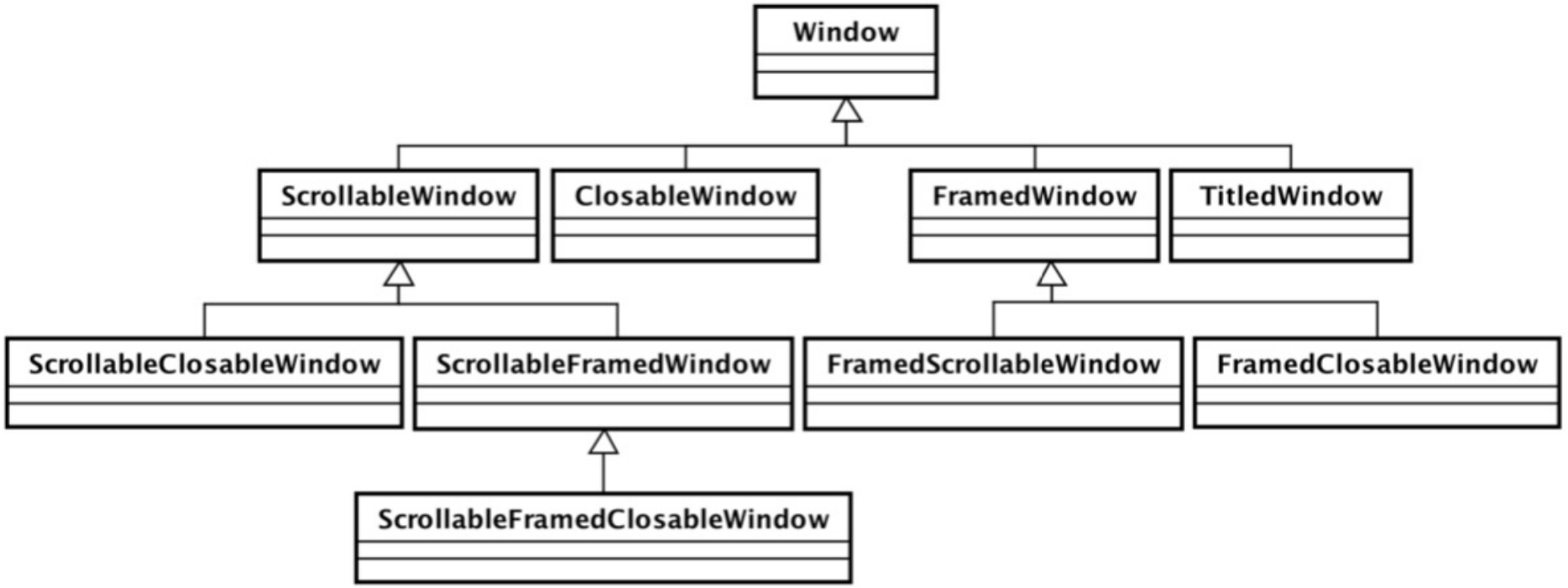
- **Eigenschaften per Vererbung hinzufügen:**



- **Sind alles Spezialisierungen**
- **ABER: Beschreiben mehr eine (boolesche) Eigenschaft**
- **Problem: Man möchte die Eigenschaften kombinieren**
  => weitere Ableitungen notwendig
  => Kombinatorische Explosion

Probleme mit Vererbung

- **Kombinatorische Explosion:**



- **Gibt es Unterschiede bei verschiedenen Ableitungsreihenfolgen?**
- **Für orthogonale (voneinander unabhängige) Eigenschaften: => Decorator-Pattern: Füge Funktionalität dynamisch ohne Vererbung hinzu**

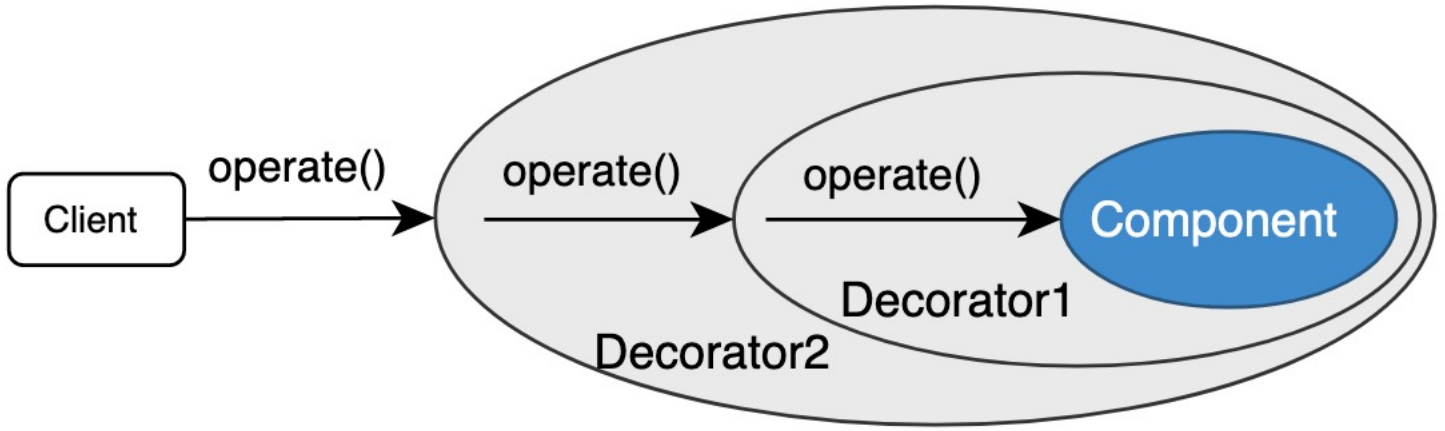
Motivation und Kurzbeschreibung Dekorierer

- zusätzliches Verhalten transparent zur Verfügung stellen
- Keine Modifikation der ursprünglichen Klasse
- Dies ist dann praktisch, wenn entweder eine zu erweiternde Klasse nicht als Sourcecode vorliegt oder dieser nicht verändert werden darf.
- Ein erster Gedanke ist häufig, eine Subklasse zu bilden und dort die gewünschten Erweiterungen vorzunehmen. Vererbung hat aber so ihre Tücken
- Dekorierer arbeitet ohne Vererbung
- Kann neue Funktionalität zur Laufzeit, also dynamisch, bereitstellen

Unglaublich!
Wie soll das den gehen?

Dekorierer

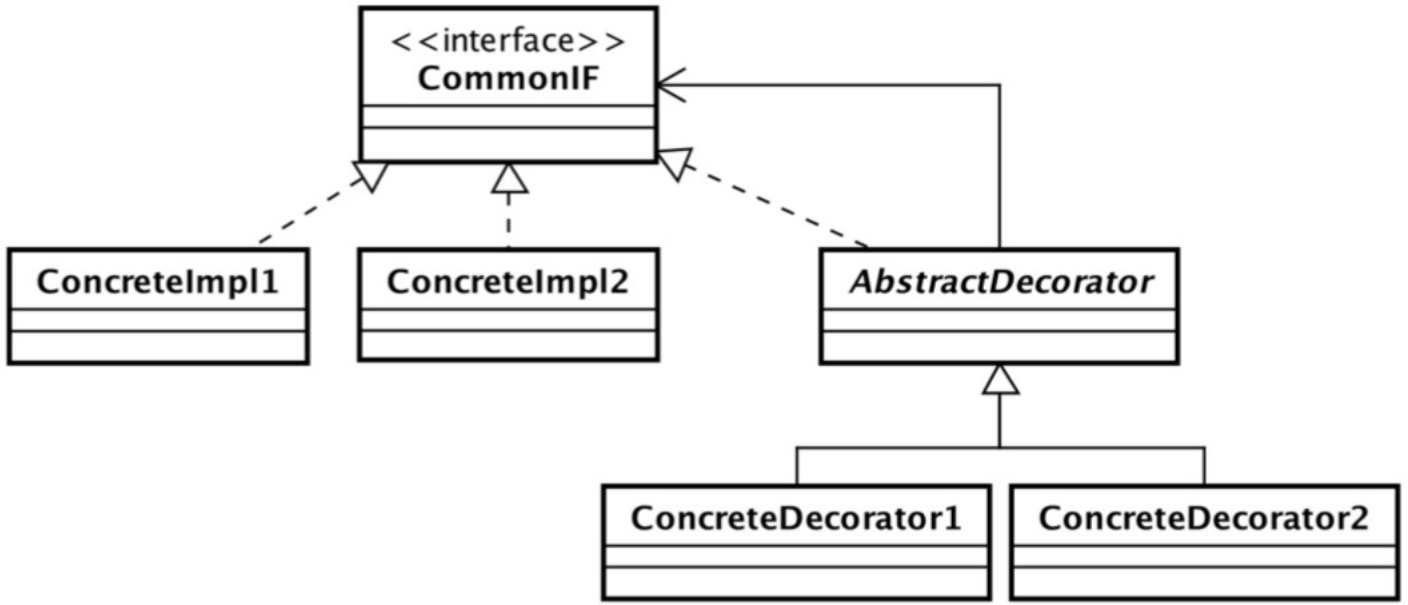
- Über das Erfüllen eines Interfaces, Ummanteln mit Funktionalität
- Erweiterungen in der Funktionalität werden wie ein Mantel / eine Schicht um die vorhandene Funktionalität¨t gelegt. Daher manchmal auch Wrapper genannt.



- Jede Dekoriererklasse realisiert nur einen Teil der Gesamtfunktionalität.
- Aber: Keine Kontrolle, welche Funktionalität hinzugefügt wird
- Kombination mit z. B. Factory Method

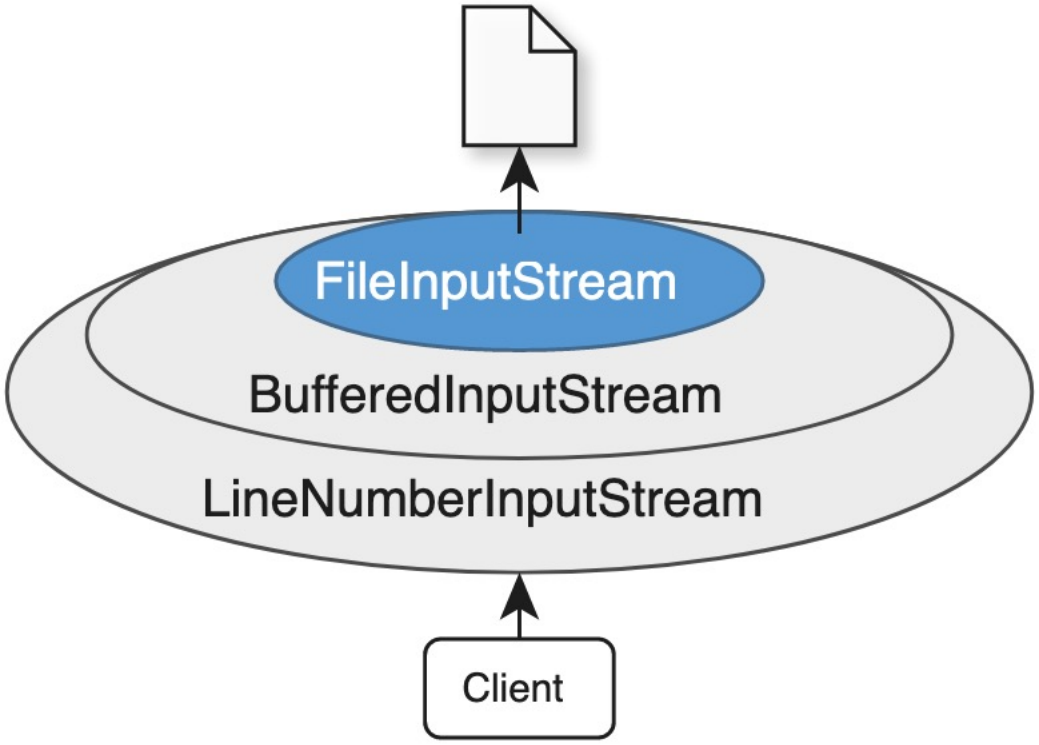
https://www.philipphauer.de/study/se/design-pattern/decorator.php

UML



- **Grundlage ist ein gemeinsamer Basistyp** (CommonIF)
- Bei Aufruf von Methoden eines Dekoriererobjekts werden **korrespondierende Methoden** des referenzierten, zu dekorierenden Objekts aufgerufen und somit der Auftrag delegiert
- Problemlos können auch mehrere **Dekoriererobjekte hintereinander geschaltet** werden, d. h., es findet dann eine **mehrfache Ummantelung** statt.

Anwendungsbeispiel I



https://www.philipphauer.de/study/se/design-pattern/decorator.php

Bewertung Dekorierer

\+ **Transparente Ergänzung zusätzlicher Funktionalität** – Funktionalität lässt transparent hinzufügen. Dies ist sogar zur Laufzeit möglich, wodurch eine Realisierung gemäß diesem Muster einer puren statischen Vererbung überlegen ist.

\+ **Hintereinanderschaltung** – Mehrere unterschiedliche Dekoriererklassen lassen sich hintereinander schalten, um komplexere Funktionalitäten zu realisieren.

\+ **Flexibilität** – Die zu dekorierende Klasse ist nicht festgelegt, da lediglich gegen eine gemeinsame Schnittstelle programmiert wird. Dekoriererklassen können für verschiedene zu dekorierende Klassen genutzt / wiederverwendet werden. Statische Vererbung erlaubt das nicht.

\+ **Vereinfachung von Vererbungshierarchien** – Komplexe und unübersichtliche Vererbungshierarchien lassen sich durch Einsatz dieses Musters vermeiden.

o **Gemeinsamer Basistyp benötigt** – Es ist ein gemeinsamer Basistyp erforderlich, der die öffentliche Schnittstelle für Dekoriererklassen und für zu dekorierende Objekte definiert.

Bewertung Dekorierer

o **Fehlende Kontrolle** – Eine Kontrolle, wer welche Funktionalität wie und wann hinzufügt und ob dies sinnvoll ist, bleibt der Disziplin des Entwicklers überlassen. Eine mehrfache Hintereinanderschaltung von Objekten derselben Dekoriererklasse, beispielsweise mehrmals Instanzen von `BufferedInputStream` oder `ReverseComparator<T>`, ist somit möglich, aber meistens nicht sinnvoll.

- **Zugriff auf Spezialisierungen schwieriger möglich** – Die Funktionalität wird durch Dekoriererobjekte transparent hinzufügt, wodurch ein Aufrufer nicht direkt darauf zugreifen kann, da dieser nur eine Referenz auf die allgemeine Dekoriererklasse hält. Als Lösung kann man eine Referenz auf eine konkrete Dekoriererklasse speichern, um deren Zusatzfunktionalität explizit nutzen zu können. Beispiel `BufferedInputStream`

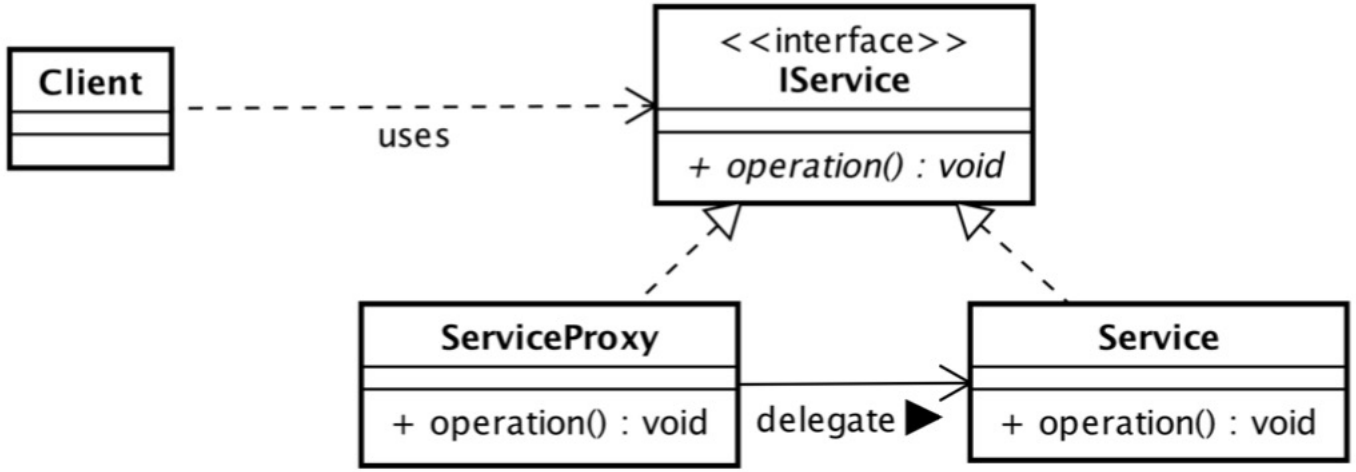
- **Implementierungsaufwand** – Enthält das zu ummantelnde Interface relativ viele Methoden, so müssen all diese implementiert werden.

Proxy

Motivation und Kurzbeschreibung Proxy

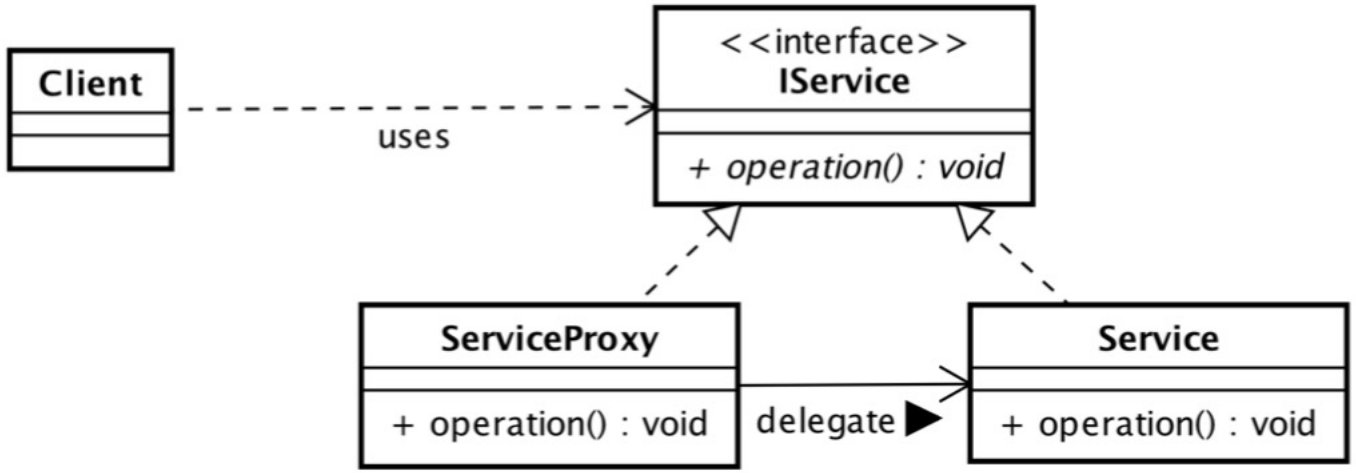
- **Verlagern** der **Kontrolle** über ein **Objekt** auf ein **Stellvertreterobjekt**, den sogenannten ***Proxy***.
- **Nutzer wenden sich immer an den Proxy und nicht direkt an das vertretene Objekt.**
- Ein **Klient kennt** das eigentliche **Objekt nicht**.
- Dadurch kann der **Proxy** sowohl die **Erzeugung** als auch den **Zugriff** auf das eigentliche **Objekt kontrollieren**:
  - **Zugangskontrolle** realisieren oder
  - **Remote Calls** verstecken
  - **Vereinfachung** von **Lazy Initialization**

UML



- Die **Basis** dieses Musters ist eine **gemeinsame Schnittstelle** `IService`. Diese stellt sicher, dass sowohl der **Proxy** in Form der Klasse `ServiceProxy` als auch das **eigentliche Objekt** vom Typ `Service` **nach außen gleich behandelt** werden können. Zur Durchführung der eigentlichen Aufgaben **verwaltet** der **Proxy** eine **Referenz** auf das **Objekt**.

UML



- Der Proxy ermöglicht es, **zusätzlich** zum **Verhalten** der Methoden eines Objekts **verschiedene weitere Aktionen auszuführen**. Ein Proxy kann die **Aktionen** sowohl **vor** als auch **nach** der **Delegation** des **Methodenaufrufs** ausführen. Dadurch ist es möglich, **Funktionalität** zu **ergänzen** und **sogar** zu **entfernen** (indem ein Methodenaufruf nicht weitergeleitet wird).

Anwendungsbeispiel

```
public class Service implements IService
{
    @Override
    public void doSomething()
    {
        System.out.println("doSomething");
    }

    @Override
    public String calculateSomething(final int value)
    {
        System.out.println("calculateSomething");
        try
        {
            TimeUnit.SECONDS.sleep(3);
        }
        catch (final InterruptedException e)
        {
            // can't happen here, no other thread to interrupt us
        }
        return "" + value;
    }
}
```

```
public interface IService
{
    void doSomething();
    String calculateSomething(int value);
}
```

Varianten Proxy

- **Access Control Proxy** – Kann den **Zugriff** auf gewisse **Daten** steuern. Beispielsweise lässt sich **vor** jeder **Methodendelegation** eine **Rechteprüfung** ausführen, die bei Bedarf die Eingabe eines Passworts verlangt.

- **Decorating/Interceptor Proxy** – **Fügt** weitere **Funktionalität transparent** gemäß dem DEKORIERER-Muster **hinzu**. Häufig spricht man auch von INTERCEPTOR.

- **Lazy Init Proxy oder Virtual Proxy** – **Vermittelt** den **Eindruck**, ein **Objekt** stehe schon zur **Verfügung**, **bevor** es **tatsächlich erzeugt** wurde. **Objektbestandteile** werden erst in dem Moment **konstruiert**, in dem diese auch tatsächlich benutzt werden. Ein **virtueller Proxy fungiert** als **Platzhalter**, etwa beim Zugriff auf Objekte, die aufwendig zu konstruieren oder zu laden sind, **beispielsweise** werden für **Bilder** zunächst **Platzhalter** dargestellt. Das kann dies zu signifikanten **Performance-Verbesserungen** führen.

- **Remote Proxy** – Agiert als **Stellvertreter** für Objekte, die über ein **Netzwerk** (remote) angesprochen werden. Für Nutzer bleibt diese Tatsache (weitgehend) **transparent**. Ein Remote Proxy kann den **Eindruck erwecken**, ein **entferntes Objekt** wäre ein **lokales**.

Anwendungsbeispiel: Decorating Proxy

```
public class ServicePerformanceProxy implements IService
{
    private final IService service;

    ServicePerformanceProxy(final IService service)
    {
        this.service = service;
    }

    @Override
    public String calculateSomething(int value)
    {
        final long startTime = System.nanoTime();

        final String result = service.calculateSomething(value);

        printExecTime("calculateSomething", System.nanoTime() - startTime);

        return result;
    }

     ...
```

Anwendungsbeispiel: Decorating Proxy

```
public class ServicePerformanceProxy implements IService
{
    ...

    @Override
    public void doSomething()
    {
        final long startTime = System.nanoTime();

        service.doSomething();

        printExecTime("doSomething", System.nanoTime() - startTime);
    }

    private void printExecTime(final String methodName, final long duration)
    {
        System.out.println("Method call of '" + methodName + "' took: " +
                           TimeUnit.NANOSECONDS.toMillis(duration) + " ms");
    }
}
```

Anwendungsbeispiel: Decorating Proxy

```
public class StaticProxyExample
{
    public static void main(String[] args)
    {
        final IService service = createService();
        service.calculateSomething(42);
        service.doSomething();
    }

    private static IService createService()
    {
        final IService service = new Service();
        return new ServicePerformanceProxy(service);
    }
}
```

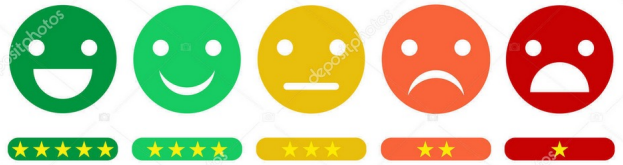
```
calculateSomething
Method call of 'calculateSomething' took:
3000 ms
doSomething
Method call of 'doSomething' took: 0 ms
```

Anwendungsbeispiel II: Access Control Proxy

```
public class RestrictedAccessMap<K, V> implements Map<K, V>
{
    private final Map<K, V> underlyingMap = new HashMap<>();

    public void ensureAccessGranted() throws InvalidAccessRightsException {
        if (!LoggedInUserService.INSTANCE.getLoggedInUser().equals("ADMIN"))
            throw new InvalidAccessRightsException("Invalid User");
    }

    @Override
    public V put(final K key, final V value) {
        ensureAccessGranted();
        return underlyingMap.put(key, value);
    }

    @Override
    public int size() {
        ensureAccessGranted();
        return underlyingMap.size();
    }

    @Override
    public boolean isEmpty() {
        ensureAccessGranted();
        return underlyingMap.isEmpty();
    }
...
```

Wie kann man die Prüfung dynamisch realisieren und nicht immer in jeder Methode selbst aufrufen?
=> Übung Dynamic Proxy

Bewertung Proxy



\+ **Steuerung von Funktionalität** – Ähnlich zum DEKORIERER-Muster **lässt** sich die eigentliche **Anwendungsfunktionalität** um weitere Funktionalität **ergänzen**. Bei diesem Muster steht jedoch der **steuernde Charakter** im **Vordergrund**.

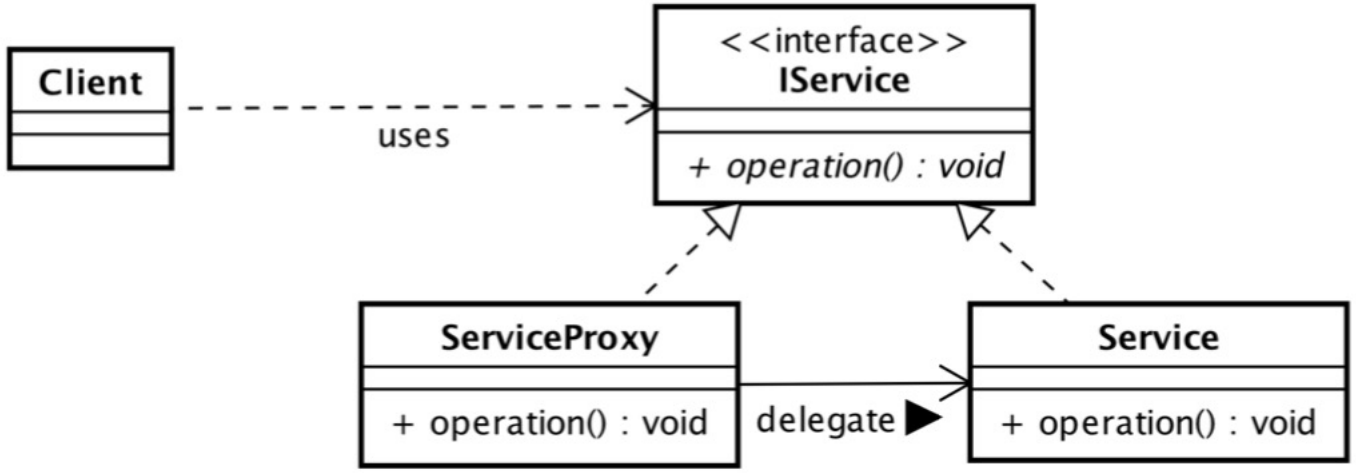
o **Aufwand durch Delegation** – Besitzt ein **Originalobjekt viele Methoden**, so ist die **Realisierung** des Proxy-Objekts durch die vielen notwendigen Delegationen **aufwendig**. Dafür können **Dynamic Proxies** hilfreich sein, deren Basis ins JDK integriert ist. Dadurch **lässt** sich der **durch Delegation verursachte Aufwand** mitunter **deutlich reduzieren**.

Dynamic Proxy

Motivation und Kurzbeschreibung Dynamic Proxy

- dynamische Proxys als spezielle Form von Proxys und damit auch ein Stellvertreter für ein anderes Objekt
- Recap: Nutzer wenden sich immer an den Proxy und nicht direkt an das vertretene Objekt.
- Recap: Proxy kann kontrollieren bzw. steuern, wann, wie und von wem auf das vertretene Objekt zugegriffen wird.
- Problem: Der Proxy muss all diejenigen öffentlichen Methoden implementieren, die das ursprüngliche Objekt in seiner Schnittstelle definiert. Das kann ziemlich aufwendig werden.

UML des statischen Proxy + Motivation Dynamic Proxy



Wenn man einen Proxy selbst implementiert, sind schnell relativ viele Methoden zu schreiben.

Man spricht dann auch von einem **statischen Proxy**, da dieser bereits **während** der **Kompilierung** vorliegt. Manchmal ist es aber wünschenswert, einen **Proxy** zu einem bestimmten Interface **erst** zur **Laufzeit** zu **erstellen**. Einen solchen nennt man dann **dynamischen Proxy**.

Dynamic Proxy im JDK

- Das JDK bietet im Package `java.lang.reflect` die **Klasse `Proxy`** zur **Konstruktion** dynamischer Proxys
- Das **Interface `InvocationHandler`**, um **dynamischen Proxy-Funktionalität** zu **implementieren**
- Zentrale Methode

```
Object invoke(Object proxy, Method method, Object[] args) throws Throwable;
```

- **Dynamischer Proxy** wird erst zur **Laufzeit** (daher dynamisch) per Aufruf von `Proxy.newProxyInstance()` **erzeugt**
- Basiert auf der **Angabe** zu **erfüllender Interfaces und eines `InvocationHandler`**

Anwendungsbeispiel

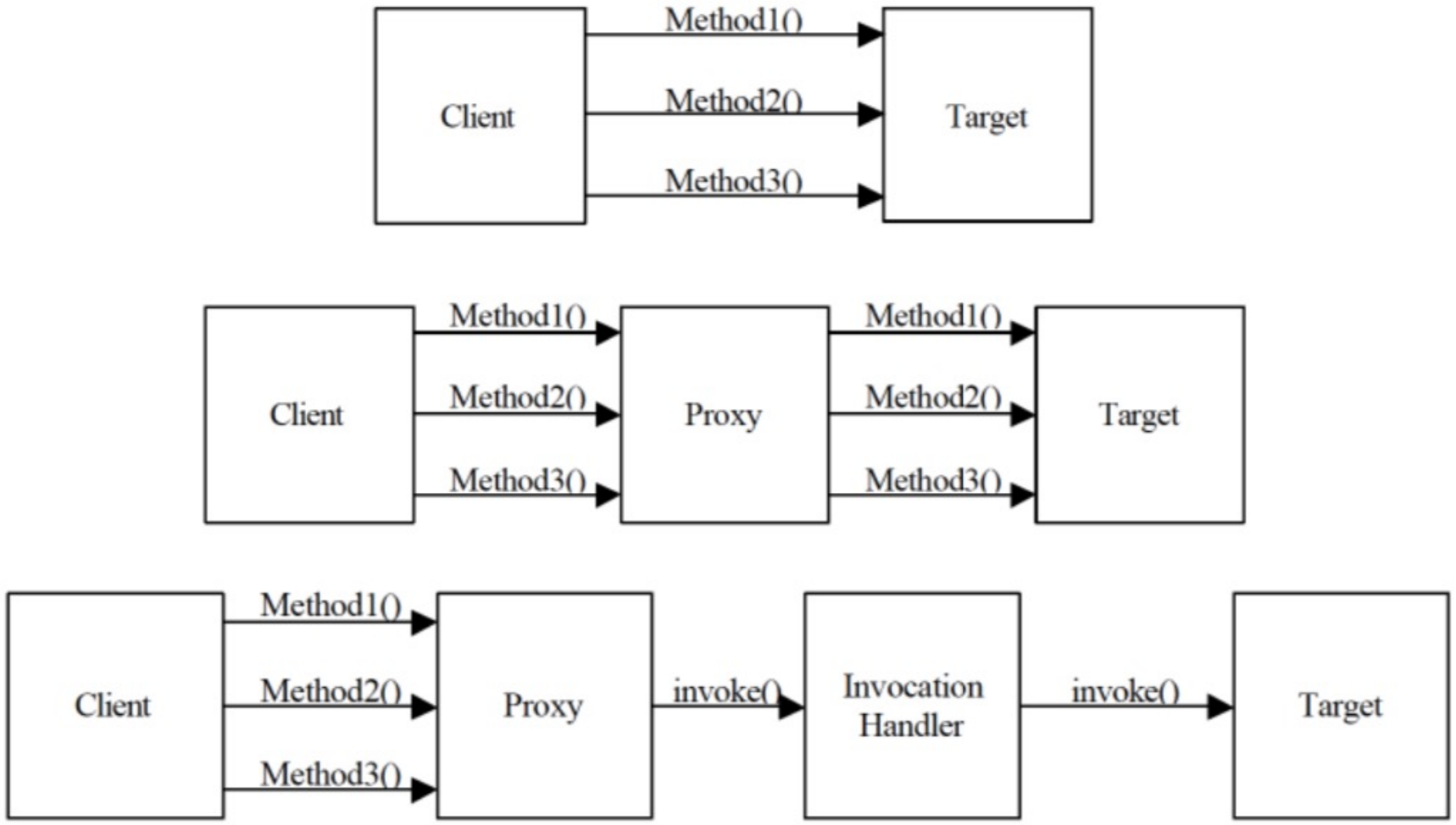
```
public class PerformanceMeasureInvocationHandler implements InvocationHandler
{
    private final IService service;

    public PerformanceMeasureInvocationHandler(final IService service)
    {
        this.service = service;
    }

    @Override
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
    {
        final long startTime = System.nanoTime();

        Object result = null;
        try
        {
            // Achtunng hier nicht versehentlich proxy übergeben
            result = method.invoke(service, args);
        }
        catch (InvocationTargetException ex)
        {
            throw ex.getTargetException();
        }
        printExecTime("calculateSomething", System.nanoTime() - startTime);
        return result;
    }
```

Anwendungsbeispiel

```
public class DynamicProxyExample
{
    public static void main(final String[] args)
    {
        final IService service = createService();
        service.calculateSomething(42);
    }

    private static IService createService()
    {
        final IService service = new Service();

        final InvocationHandler handler = new PerformanceMeasureInvocationHandler(service);

        final Class<?>[] proxyInterfaces = { IService.class };
        return (IService) Proxy.newProxyInstance(Service.class.getClassLoader(),
                                                 proxyInterfaces, handler);
    }
}
```

```
private static IService createService()
{
    final IService original = new Service();
    return new ServicePerformanceProxy(original);
}
```

Aufrufe Direkt + Proxy + Dynamic Proxy



Anwendungsbeispiel II

```
public class LoggingInvocationHandler implements InvocationHandler
{
    private Object target;

    public LoggingInvocationHandler(final Object target)
    {
        this.target = target;
    }

    @Override
    public Object invoke(final Object proxy, final Method method,
                         final Object[] args) throws Throwable
    {
        System.out.println("Invoking " + method.getName() + " " +
                           Arrays.toString(args));
        return method.invoke(target, args);
    }
}
```

Anwendungsbeispiel II: Dynamic Proxy in Kombination

```
public class DynamicProxyCombinationExample
{
    public static void main(final String[] args)
    {
        final IService service = createService();
        service.calculateSomething(42);
    }

    private static IService createService()
    {
        IService origService = new Service();

        InvocationHandler handler1 = new PerformanceMeasureInvocationHandler(origService);
        Class<?>[] proxyInterfaces = { IService.class };
        IService proxiedService = (IService) Proxy.newProxyInstance(Service.class.getClassLoader(),
                                                                     proxyInterfaces, handler1);

        InvocationHandler handler2 = new LoggingInvocationHandler(proxiedService);
        return (IService) Proxy.newProxyInstance(Service.class.getClassLoader(),
                                                 proxyInterfaces, handler2);
    }
}
```

Fazit zu dynamischen Proxys

- **bestehende Klassen lassen sich mit dynamischen Proxys ohne viel Mühe um Funktionalität erweitern**
- **Vor allem Querschnittsfunktionalitäten sind ideale Kandidaten, die mit dynamischen Proxys realisiert werden können, etwa**
  - Performance-Messung,
  - Logging usw.
- **Zudem lassen sich dynamische Proxys einfach kombinieren.**
- **Überlegen Sie einmal, wie viel Sourcecode Sie schon für die hier gezeigten beiden Proxys hätten in Ihren Applikationsklassen schreiben müssen – besser noch, man kann die Funktionalität nahezu überall wiederverwenden.**

Questions?

Thank You