Zürcher Hochschule
für Angewandte Wissenschaften

School of
Engineering
InIT Institut für angewandte
Informationstechnologie

Bachelor of Science (BSc) in Informatik

Modul Software-Entwicklung 1 (SWEN1)

LE 08 – Entwurf mit Design Pattern I
Zusammenfassung

SWEN1/PM3 Team:
R. Ferri (feit), D. Liebhart (lieh), K. Bleisch (bles), G. Wyder (wydg)

Ausgabe: HS24

Um was geht es?

- Entwurfsmuster:
  - Was ist das? Warum?
  - Aufbau
  - Anschauen:
    - Adaptor
    - Factory
    - Singleton
    - Dependency Injection
    - Proxy
    - Chain of Responsibility

Um was geht es?

- Vor und während dem Programmieren müssen laufend Entscheide gefällt werden, die nicht nur das unmittelbare Umfeld des Ortes betreffen, an dem gerade Code hinzugefügt wird, sondern in einem grösseren Zusammenhang betrachtet werden sollten, wie zum Beispiel:
  - Welche Klasse(n) müssen für eine neue Funktionalität ergänzt oder neu entwickelt werden?
  - Wie mache ich einen Algorithmus austauschbar?
  - Wie erzeuge ich eine neue Instanz, wenn der Typ konfigurationsabhängig ist?
- Entwurfsmuster bieten dafür bewährte Lösungen an.

Lernziele LE 08 – Entwurf mit Design Patterns I

- Sie sind in der Lage:
  - Den allgemeinen Aufbau und Zweck von Design Patterns (Entwurfsmuster) zu erklären.
  - Den Aufbau und Einsatz der folgenden Design Patterns zu erklären und anzuwenden:
    - Adaptor
    - Factory
    - Singleton
    - Dependency Injection
    - Proxy
    - Chain of Responsibility

Agenda

1. **Einführung in Design Patterns**
2. Repetition GRASP
3. Design Pattern
4. Wrap-up und Ausblick

Design Pattern: Was ist das? Warum?

Bewährte Lösungsschablonen
für wiederkehrende Entwurfsprobleme

- Rad nicht neu erfinden
- Gemeinsame Sprache/Verständnis
- Best-practices lernen

Aufbau Design Patterns

- Beschreibungsschema
  - Name
  - Beschreibung Problem
  - Beschreibung Lösung
  - Hinweise für Anwendung
  - Beispiele

GRASP und GoF

- GRASP
  - Grundlegende Prinzipien der Zuweisung von Verantwortlichkeiten, formuliert als Design Patterns
- GoF
  - Buch «Design Patterns», herausgekommen 1995
  - 23 Patterns, 15 sind gebräuchlich
  - Erste systematische Publikation von Design Patterns
  - Können als Spezialisierungen von GRASP interpretiert werden

GoF Design Patterns Catalog

- Creational Patterns
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Structural Patterns
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy
- Behavioral Patterns
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

s. Cheat Sheet Design Patterns [5]

Anwendung von Design Patterns

- Design Patterns sind ein wertvolles Werkzeug, um bewährte Lösungen für wiederkehrende Probleme schnell zu finden.
- Sie helfen, im Team effizient über Lösungsmöglichkeiten zu kommunizieren.
- Ihre Anwendung stellt aber immer einen Trade-Off zwischen Flexibilität und Komplexität dar.
- Es ist keineswegs so, dass ein Programm automatisch besser wird, wenn mehr Patterns angewendet werden.

Agenda

1. Einführung in Design Patterns
2. **Repetition GRASP**
3. Design Pattern
4. Wrap-up und Ausblick

Denkpause

Aufgabe 8.1 (5’)

Diskutieren Sie in Murmelgruppen folgende Fragen:

- Wie heissen die 9 GRASP Design Patterns?
- Welche dieser 9 Pattern decken sehr häufige Entwurfsentscheidungen ab?
- Wie sieht ein UML Klassen- und Sequenzdiagramm aus?

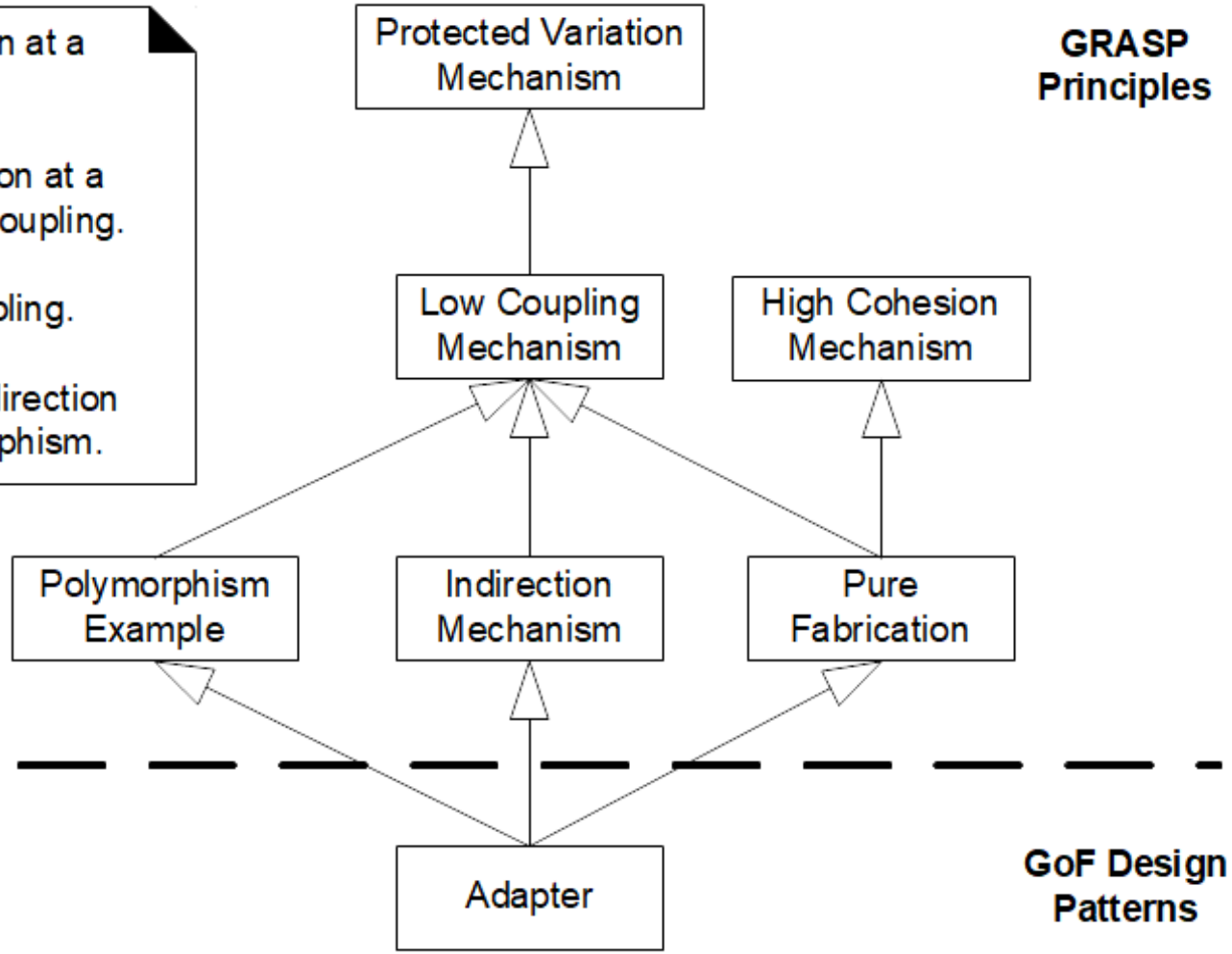
GRASP Prinzipien und GoF Design Patterns

Low coupling is a way to achieve protection at a variation point.

Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.

An indirection is a way to achieve low coupling.

The Adapter design pattern is a kind of Indirection and a Pure Fabrication, that uses Polymorphism.



GoF Patterns sind Spezialfälle der GRASP Prinzipien

Agenda

1. Einführung in Design Patterns
2. Repetition GRASP
3. **Design Patterns**
4. Wrap-up und Ausblick

Liste der Design Patterns für die Lerneinheit

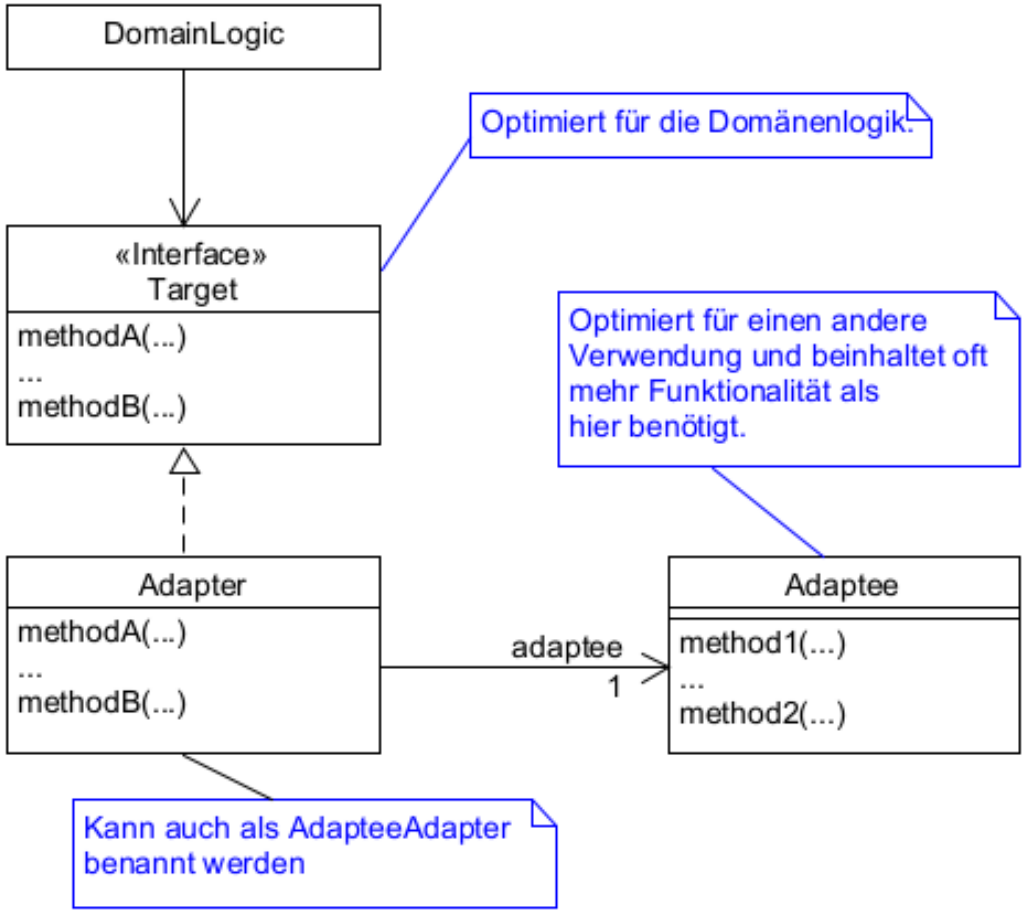
- Adapter
- Simple Factory
- Singleton
- Dependency Injection
- Proxy
- Chain of Responsibility

Erläuterungen zu den Beispielen

- Die Design Patterns werden basierend auf den Diagrammen und Beschreibungen in [4] (GoF Buch) eingeführt und das dortige Beispiele ebenfalls erklärt.
- Java SDK: In der Standard-Klassenbibliothek werden fast alle Design-Pattern angewendet. Es werden die Klassennamen angegeben, aber keine weiteren Angaben zur Umsetzung. Es wird empfohlen, die angegebenen Klassen im Quelltext anzuschauen.
- Larman [1], POST: In der Point of Sales Terminal Anwendung werden fast alle Design-Patterns angewendet. Diese Beispiele werden mit Diagrammen im Detail erläutert.
- Wo möglich werden Code-Beispiele der Umsetzung der Design-Patterns gezeigt.

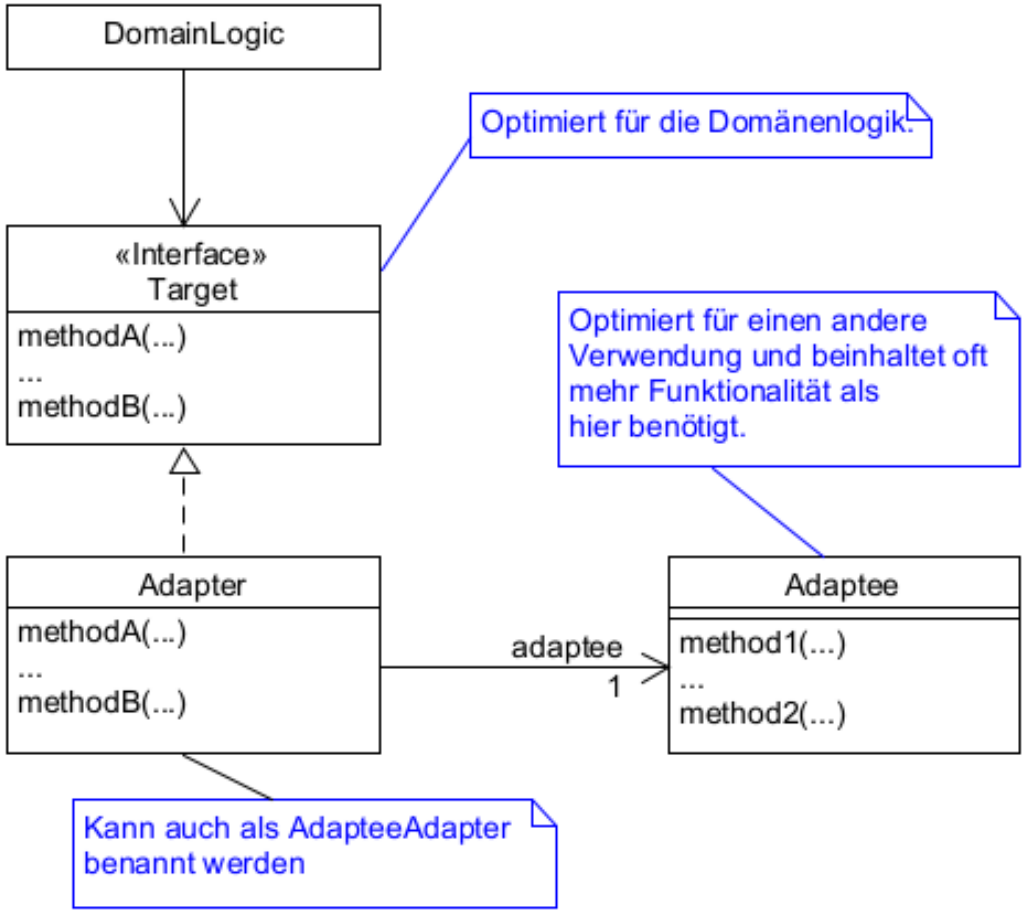
Adapter: Problem und Lösung

- Problem
  - Eine Klasse soll eingesetzt werden, die aber inkompatibel mit einem bereits definierten domänen-spezifischem Interface ist.
- Lösung
  - Eine eigene Adapter Klasse wird dazwischengeschaltet.



Adapter: Hinweise

- Hinweise
  - Oft wird so ein externer Dienst in die eigene Anwendung integriert, insbesondere wenn der Dienst austauschbar sein soll.
  - Das Target Interface ist bewusst für die Domänenlogik optimiert, während der Adaptee oft von extern bezogen wird.
  - Falls es sich beim Adaptee um einen externen Dienst handelt, kann im Adapter allenfalls die Kommunikation integriert werden.



Adapter: Beispiele (1/2)

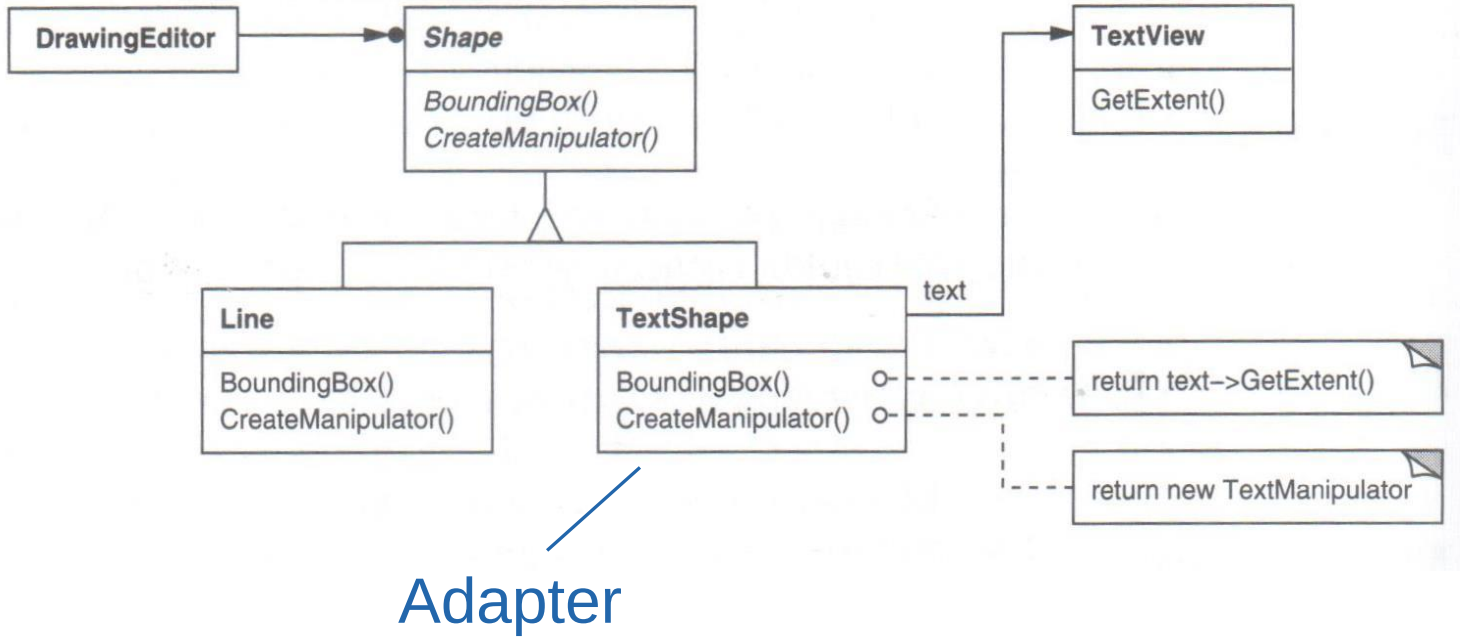
- JDK
  - java.awt.Component ist die Basisklasse des Abstract Window Toolkit, eine visuelle Klassenbibliothek, die das Erstellen eines GUI für verschiedene Plattformen unterstützt. Während Component den unabhängigen Teil darstellt, gibt es eine getPeer() Methode, die java.awt.peer.ComponentPeer zurückliefert, worin sich dann der plattformspezifische Code befindet. Somit übernimmt ComponentPeer die Rolle eines Adapters.
  - javax.xml.parsers.SAXParser ist ein Adapter auf ein org.xml.sax.XMLReader Objekt.

Adapter: Beispiele (2/2)

- GoF Beispiel (siehe nachfolgende Folie)
  - TextShape ist ein Adapter auf ein TextView Objekt.
- Larman, Point Of Sale Terminal (siehe nachfolgende Folie)
  - Adapter für die Ansteuerung von verschiedenen Buchhaltungsprogrammen.
- Allgemein
  - Überall, wo externe Dienste angesprochen werden und diese Dienste **austauschbar** sein sollten, kommen Adapter zum Einsatz.
- Code Beispiel
  - FileVisitor (java.nio.file) Implementation, die die Dateinamen in einen BufferedWriter schreibt.

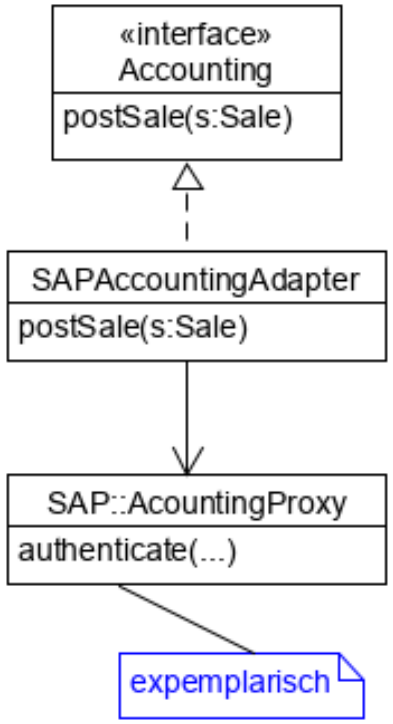
Adapter: Beispiel GoF

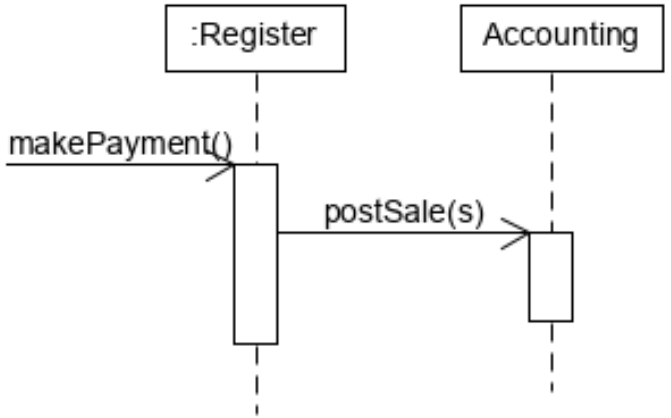
- Das Framework hat unterschiedliche Vererbungshierarchien für Shape (einfache geometrische Figuren) und Views (komplexe Komponenten).
- Die fehlende TextShape Klasse wird nun als Adapter realisiert, der die Shape Methoden auf TextView Methoden anpasst.

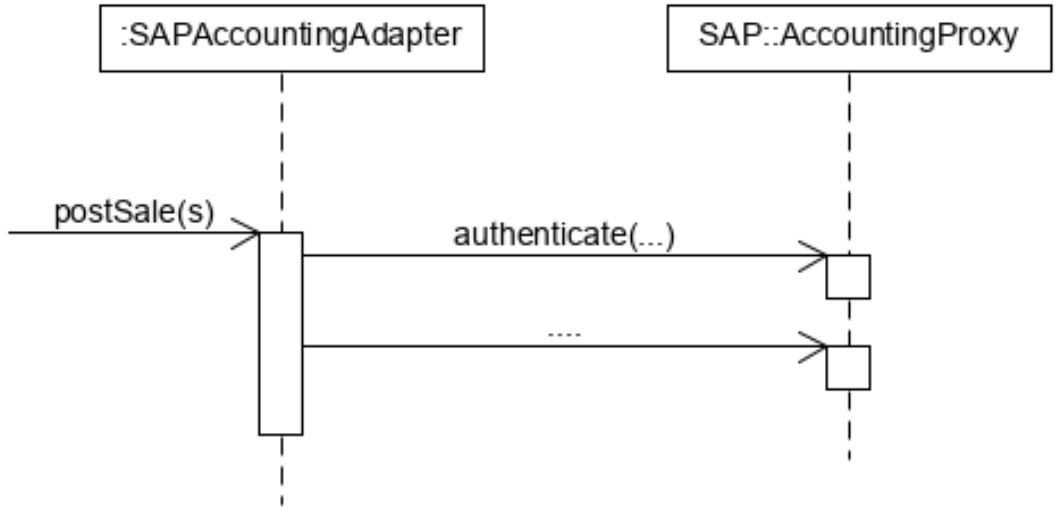


Adapter: Beispiel Point Of Sale Terminal

Die Anforderungen verlangen, dass dem Buchhaltungsprogramm am Schluss die Daten eines Kaufs übermittelt werden. Dabei sollen verschiedene Buchhaltungsprogramme unterstützt werden.







Adapter: Code Beispiel

Das Adapter Pattern ist so allgemein, dass es kein typisches Codebeispiel gibt.
Bei diesem Beispiel geht es darum, dass alle Dateien unterhalb eines Wurzelverzeichnisses aufgelistet und in eine Datei geschrieben werden.
Der FileVisitor kann so als Adapter auf einen BufferedWriter interpretiert werden.

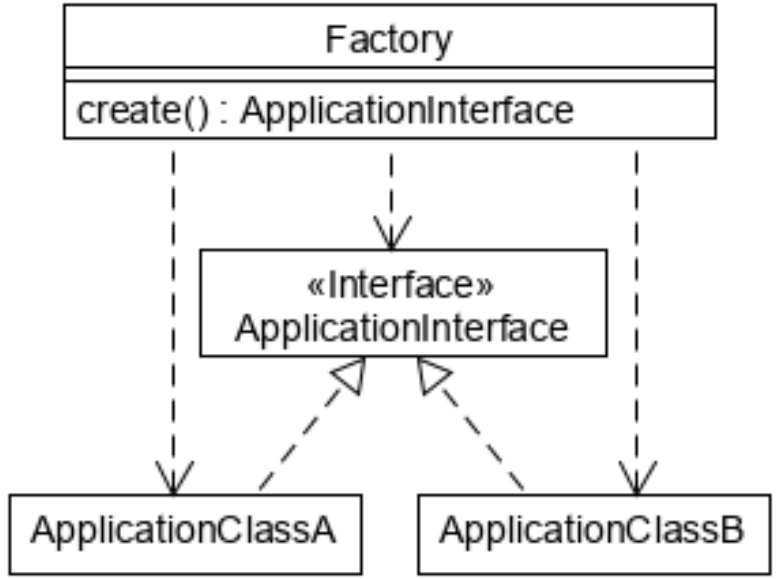
```
public static class FileVisitorToTextFileAdapter implements FileVisitor<Path> {
    private BufferedWriter writer;

    public FileVisitorToTextFileAdapter(BufferedWriter writer) {
        super();
        this.writer = writer;
    }

    @Override
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        writer.write(file.toString() + "\n");
        return FileVisitResult.CONTINUE;
    }
```

Anpassung der eingehenden Parameter an die Bedürfnisse des Ziels

Die weiteren Methoden von FileVisitor wurden weggelassen

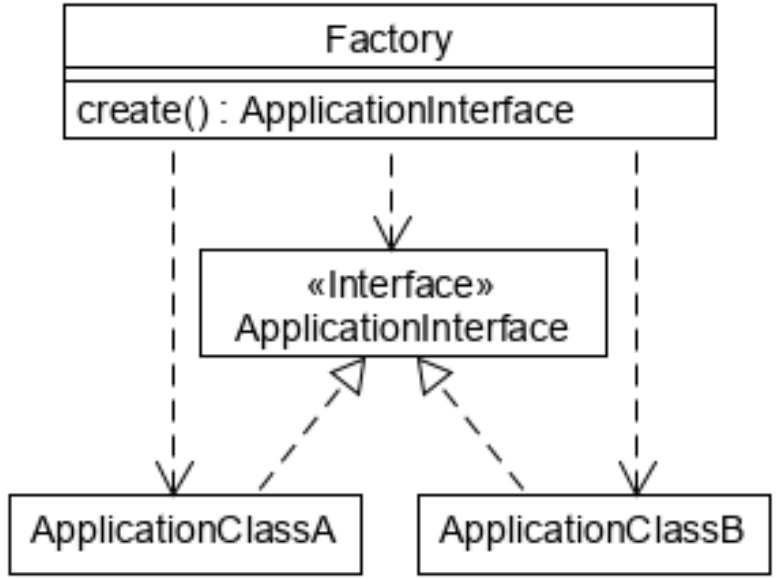
Simple Factory: Problem und Lösung

- Problem
  - Das Erzeugen eines neuen Objekts ist aufwändig.
- Lösung
  - Eine eigene Klasse für das Erzeugen eines neuen Objekts wird geschrieben.



Simple Factory: Hinweise

- Hinweise
  - Oft ist die Erzeugung des neuen Objekts von irgendeiner Art von Konfiguration abhängig.
  - Es ist auch möglich, die create() Methode mit Parametern zu ergänzen.
  - Die Factory kann allenfalls die erzeugten Objekte zwischenspeichern und später wiederverwenden.



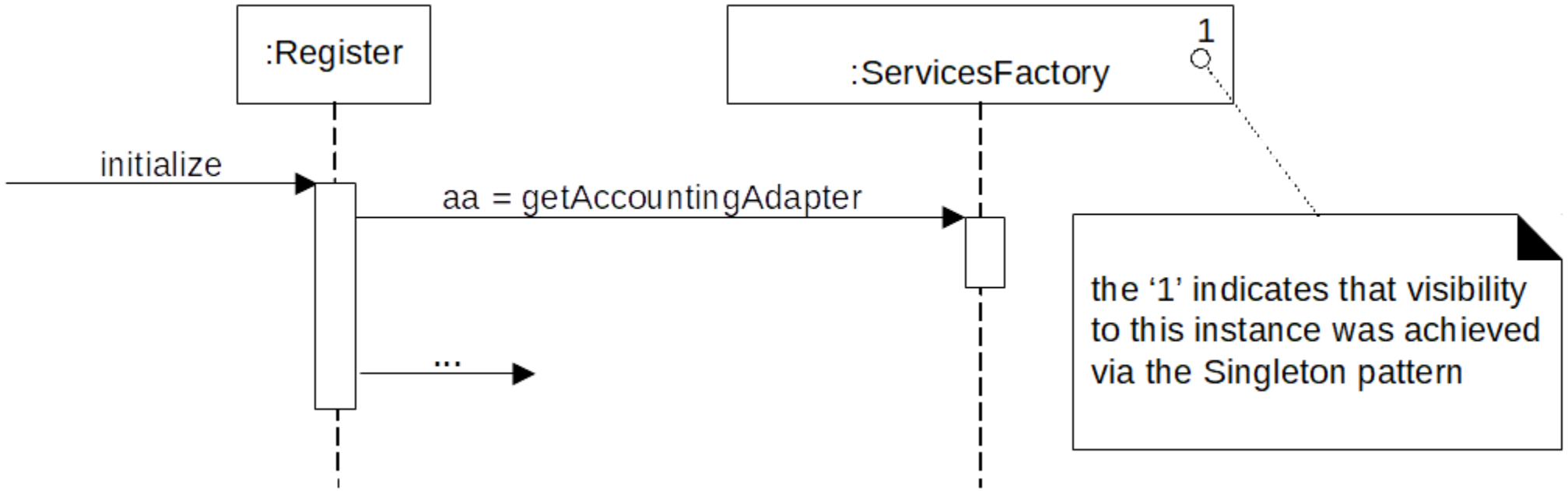
Simple Factory: Beispiele

- JDK
  - java.net.SocketFactory
  - javax.xml.parsers.SAXParserFactory
  - javax.crypto.SecretKeyFactory
- Larman, Point Of Sale Terminal (siehe nachfolgende Folien)
  - Der Adapter zum Buchhaltungsprogramm wird von einer Factory erzeugt, da es verschiedene Adapter für die verschiedenen Buchhaltungsprogramme gibt. Der Kunde wird bei der Installation dann z.B. über eine Konfigurationsdatei festlegen, welches Buchhaltungsprogramm und damit welche Adapter-Klasse er verwendet.
- Code Beispiel: Factory, die einen HTTP Request erzeugt

Simple Factory: Alternativen

- Alternativen
  - GoF Abstract Factory: Verantwortlich für die Erzeugung einer Familie von verwandten Objekte.
  - GoF Factory Method: In der Basisklasse wird eine abstrakte Methode definiert, die die einzige Verantwortung hat, ein Objekt eines bestimmten Interfaces zu erzeugen. In abgeleiteten Klassen wird dann diese Methode überschrieben und diese erzeugt dann das gewünschte Objekt.

Simple Factory: Beispiel Point Of Sale Terminal



Simple Factory: Code Beispiel

Um einen HTTP Request zu erzeugen, müssen viele Parameter festgelegt werden. Viele dieser Parameter (z.B. ein Header Eintrag für den API Key) ändern sich für ein bestimmtes Ziel nicht, und nur ein kleiner Teil (z.B. der Body) ändert sich von Aufruf zu Aufruf. Daher kann eine Factory dafür eingesetzt werden.

```
public class HttpRequestFactory {
    private String host;
    private int port;

    public HttpRequestFactory(String host, int port) {…}

    public HttpRequest createHttpClient(String body) throws IOException, InterruptedException {
        URI uri = URI.create("http://" + host + ":" + port + "/naturallanguageanalyzer/handleparagraph");
        HttpRequest request = HttpRequest.newBuilder(uri)
            .header("accept", "text/*")
            .header("apikey", "demokey")
            .method("GET", BodyPublishers.ofString(body))
            .build();
        return request;
    }
```

Constructor Code weggelassen

Denkpause

Aufgabe 8.2 (5’)

Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen:

- Sie haben die Idee, eine generische SimpleFactory Klasse zu schreiben, die von einem Property-File einen Klassennamen liest, diese Klasse lädt und damit ein neues Objekt erzeugt. Skizzieren Sie den Aufbau der create() Methode.
- Welche JDK-Klassen können dafür eingesetzt werden?
- Was spricht dafür, den (Java-) Klassennamen im Property-File zu definieren, und was dagegen? Wer wird so ein Property-File schreiben oder verändern?

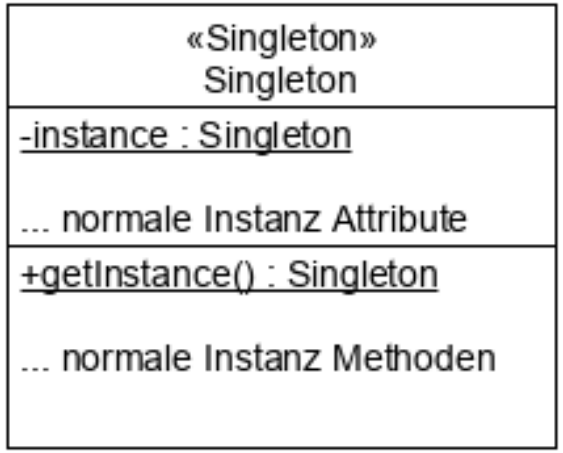
Singleton: Problem und Lösung

- Problem
  - Man benötigt von einer Klasse nur eine einzige Instanz.
  - Diese Instanz muss global sichtbar sein.
- Lösung
  - Klasse mit einer statischen Methode, die immer dasselbe Objekt zurückliefert.
  - Statische Methode wird public deklariert.

```
public class Singleton {

private static Singleton instance = new Singleton();

// privater Konstruktor, verhindert direkte Instanziierung.
private Singleton() {
}

// Instanzvariablen ...

public static Singleton getInstance() {
    return instance;
}

// Instanz-Methoden ...
}
```

Singleton: Hinweise

- Hinweise
  - Globale Sichtbarkeit wird heutzutage sehr kritisch betrachtet.
  - Lazy Creation für die Instanz ist möglich, dann sollte aber die getInstance() Methode synchronisiert werden.



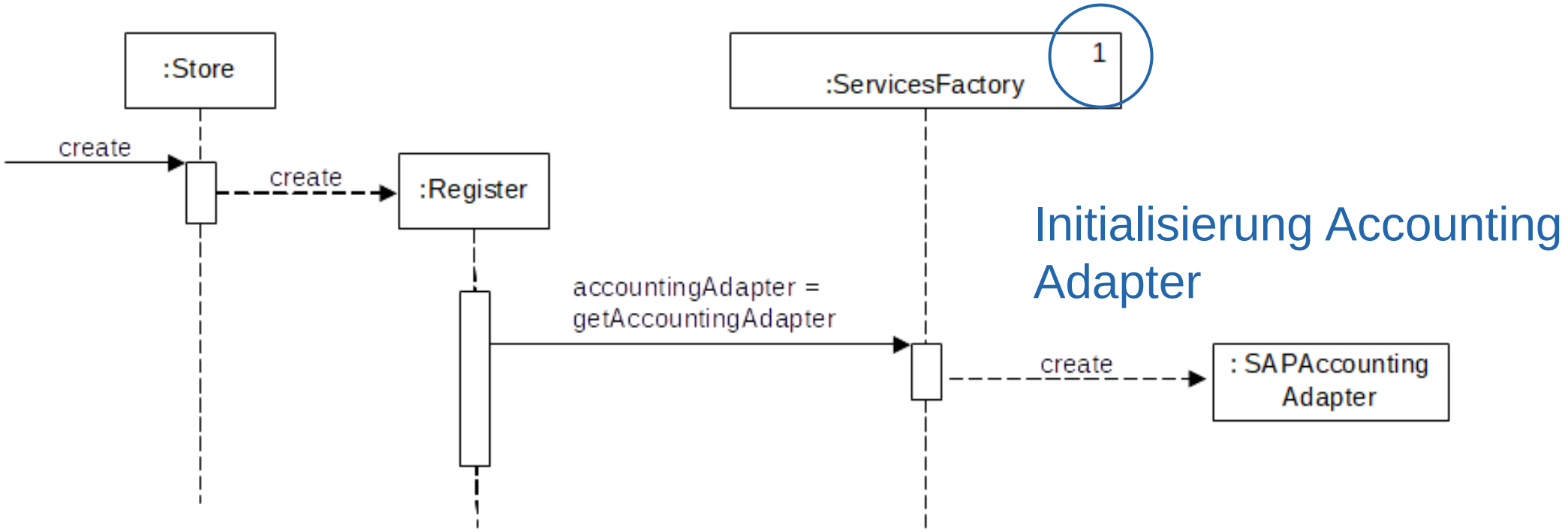
Singleton: Beispiele

- JDK
  - ClassLoader.getSystemClassLoader()
  - SocketFactory.getDefault()
  - Runtime.getRuntime()
- Larman, Point Of Sale Terminal
  - Die Factory für den Adapter zum Buchhaltungsprogramm ist ein Singleton.

Singleton: Bemerkungen

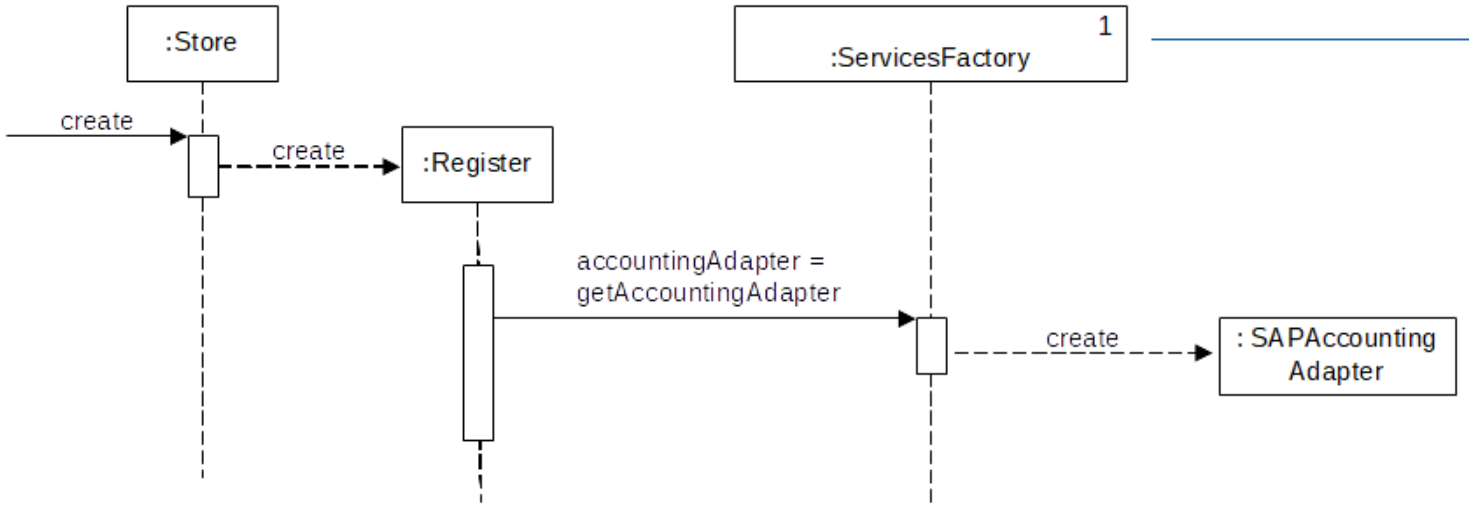
- Allgemein
  - Singletons sind dann wichtig, wenn es genau einen zentralen Ort braucht, um Ressourcen zu verwalten.
  - Es kann ebenfalls von Vorteil sein, dass Speicherplatz gespart wird, wenn es nur ein Objekt gibt. Allerdings ist dieser Vorteil bei «normalen» Anwendungen eher unbedeutend.
  - Die Java enum Instanzen sind ebenfalls Singletons.
  - Die globale Sichtbarkeit ist eher problematisch.

Singleton: Beispiel Point Of Sale Terminal (1/2)



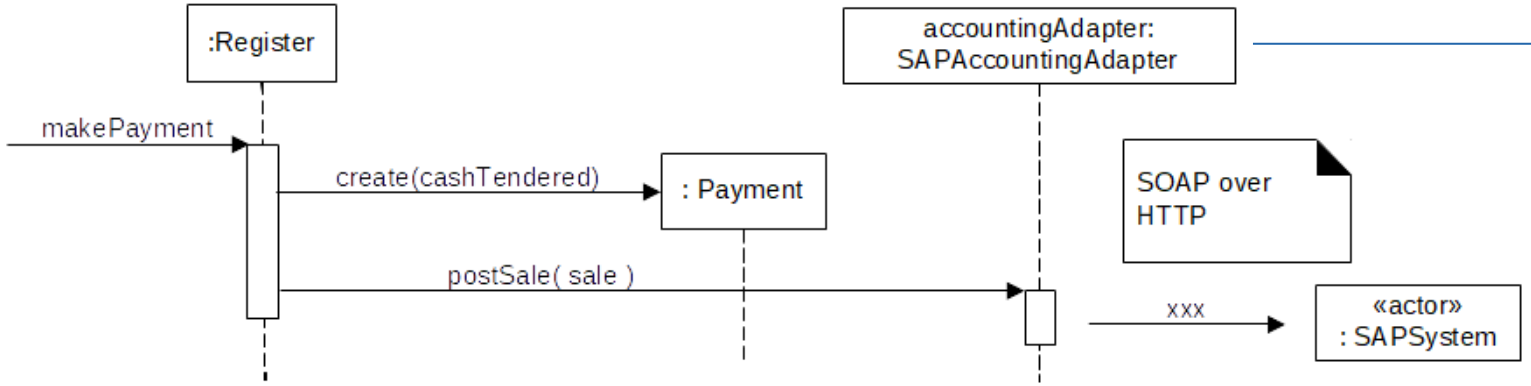
Initialisierung Accounting Adapter

Singleton: Beispiel Point Of Sale Terminal (2/2)



Singleton und Simple Factory

Kombination von mehreren Design Patterns



Adapter

Denkpause

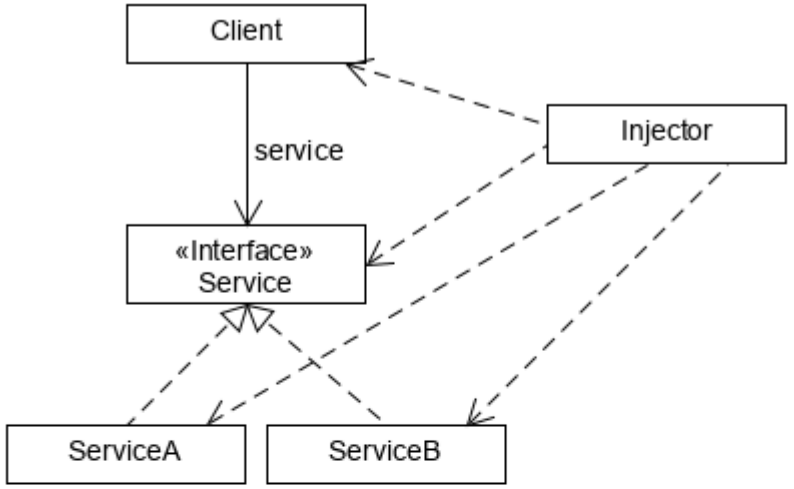
Aufgabe 8.3 (5’)

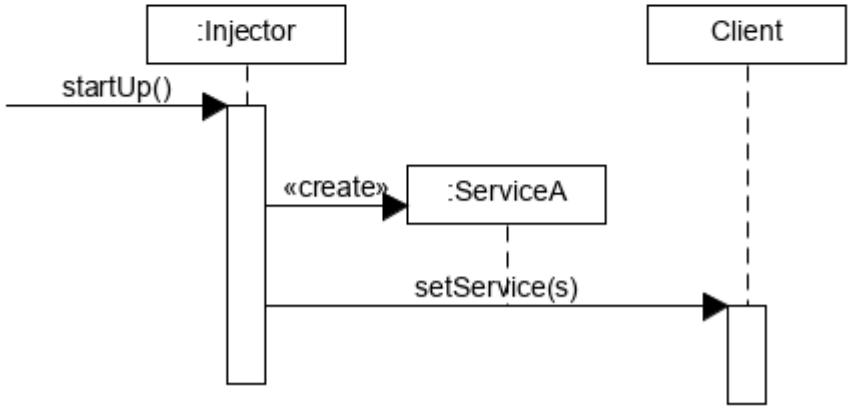
Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen:

- Basierend auf den vorhergehenden Diagrammen skizzieren Sie das DCD (ohne Methoden und Attribute) von POST, das die folgenden Klassen enthält: Store, Register, ServicesFactory, AccountingAdapter und SAPAccountingAdapter.
- Wer hat eine Assoziation auf ServicesFactory? Welche Klassen haben eine Dependency davon?
- Welchen Typ hat das Attribut Register.accountingAdapter?

Dependency Injection (DI): Problem und Lösung

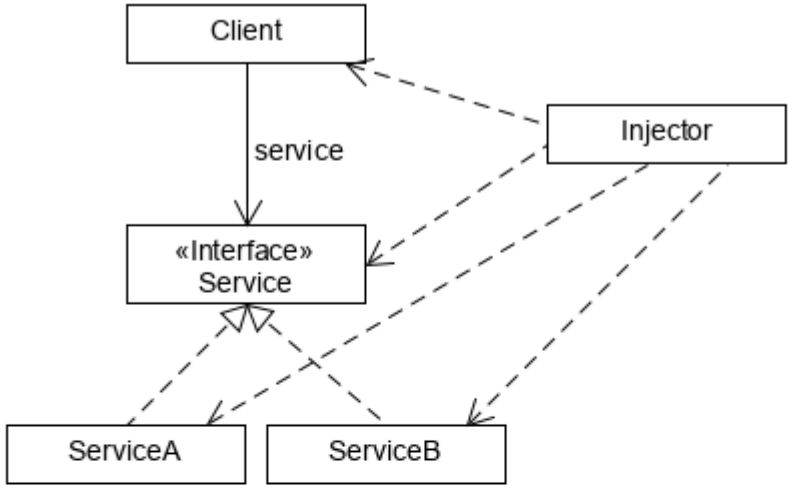
- Problem
  - Eine Klasse braucht eine Referenz auf ein anderes Objekt. Dieses Objekt muss ein bestimmtes Interface definieren, je nach Konfiguration aber mit einer anderen Funktionalität.
- Lösung
  - Anstelle, dass die Klasse das abhängige Objekt selber erzeugt, wird dieses Objekt von aussen (Injector) gesetzt.

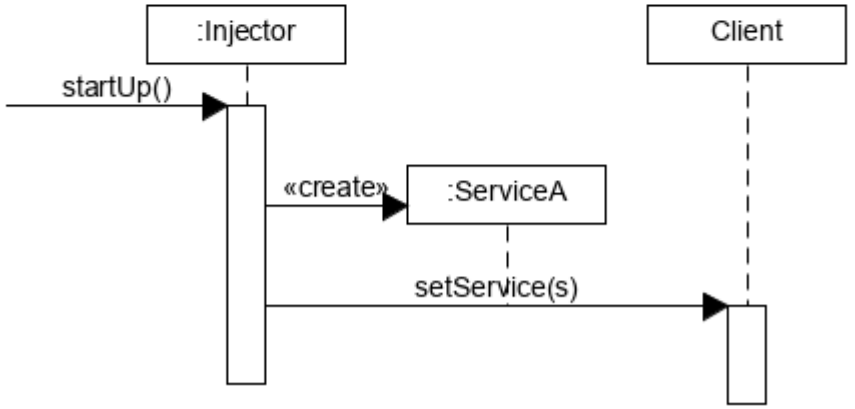




Dependency Injection (DI): Hinweise (1/2)

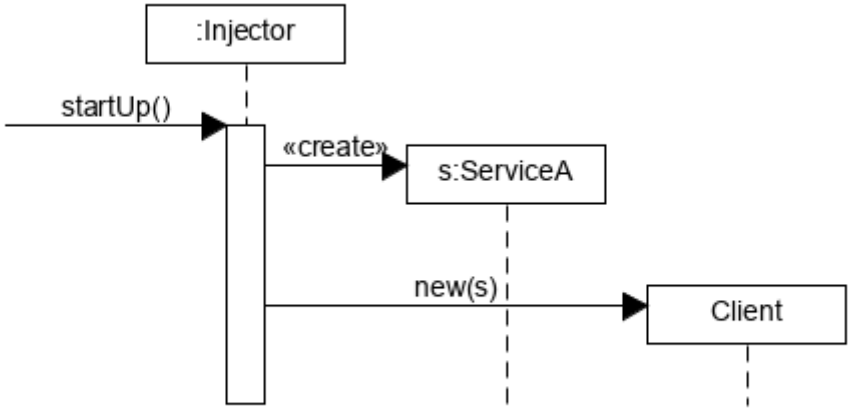
- Hinweise
  - Ersatz für das Factory Pattern.
  - Direkter Widerspruch zum GRASP Creator Prinzip.
  - Viele Frameworks unterstützen inzwischen DI (z.B. Spring), kann aber problemlos auch ohne Framework angewendet werden.
  - Erleichtert das Schreiben von Testfällen, insbesondere den Gebrauch von Mocks.





Dependency Injection (DI): Hinweise (2/2)

- Varianten
  - Das Beispiel auf der vorhergehenden Folie zeigt DI über setter Methoden.
  - Alternativ kann der Service bei Constructor vom Client übergeben werden.
  - Diese Idee kann so weit geführt werden, dass der Injector die ganze Anwendung initialisiert.
- Frameworks verwenden Annotationen, um anzuzeigen, welche Attribute über DI gesetzt werden sollen.



Denkpause

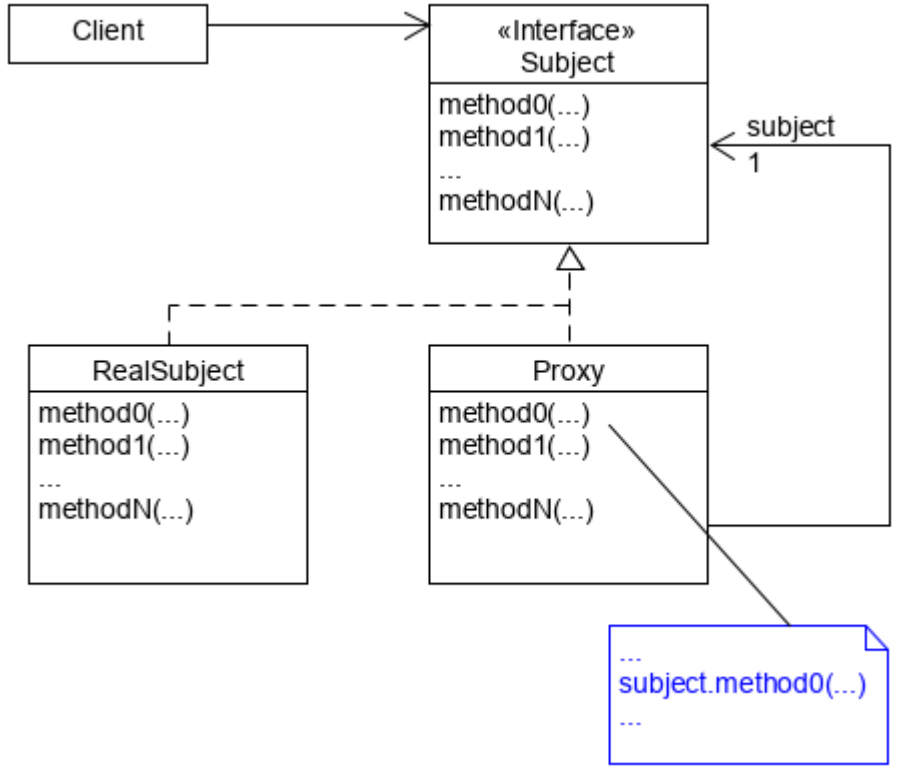
Aufgabe 8.4 (5’)

Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen:

- Betrachten Sie das DCD von der letzten Aufgabe. Welche Klasse erhält die Aufgabe des «Injectors»? Welche Klassen fallen weg? Wohin geht diese Verantwortlichkeit?
- Wer übernimmt die Rolle des Injectors im Falle von automatischen Testfällen?

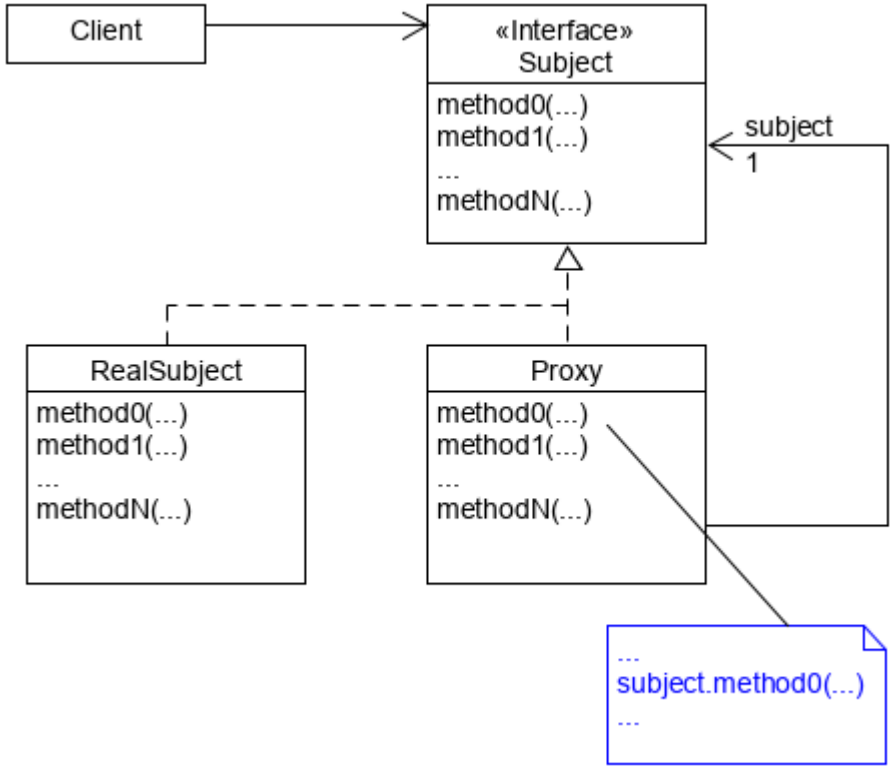
Proxy: Problem und Lösung

- Problem
  - Ein Objekt ist nicht oder noch nicht im selben Adressraum verfügbar.
- Lösung
  - Ein Stellvertreter Objekt («Proxy») mit demselben Interface wird anstelle des richtigen Objekts verwendet.
  - Das «Proxy» Objekt leitet alle Methodenaufrufe zum richtigen Objekt weiter.



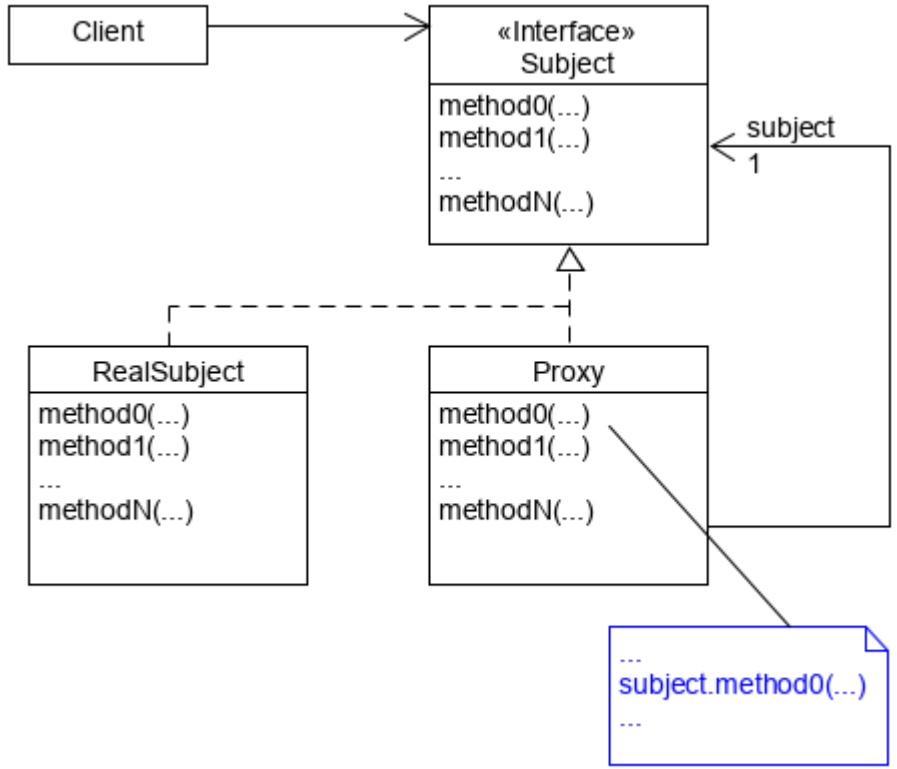
Proxy: Einsatz

- Einsatz als (Struktur ist dieselbe!)
  - «Remote Proxy» ist ein Stellvertreter für ein Objekt in einem anderen Adressraum und übernimmt die Kommunikation mit diesem.
  - «Virtual Proxy» verzögert das Erzeugen des richtigen Objekts auf das erste Mal, dass dieses benutzt wird.
  - «Protection Proxy» kontrolliert den Zugriff auf das richtige Objekt.



Proxy: Hinweise

- Sieht ähnlich aus wie ein Adapter, der Unterschied ist aber, dass der «Adaptee», in diesem Fall das RealSubject, auch dasselbe Interface implementiert wie der «Adapter» resp. Subject
- Vom Aufbau her identisch mit dem Decorator Pattern (siehe LE09), hat aber einen anderen Zweck.



Proxy: Beispiele

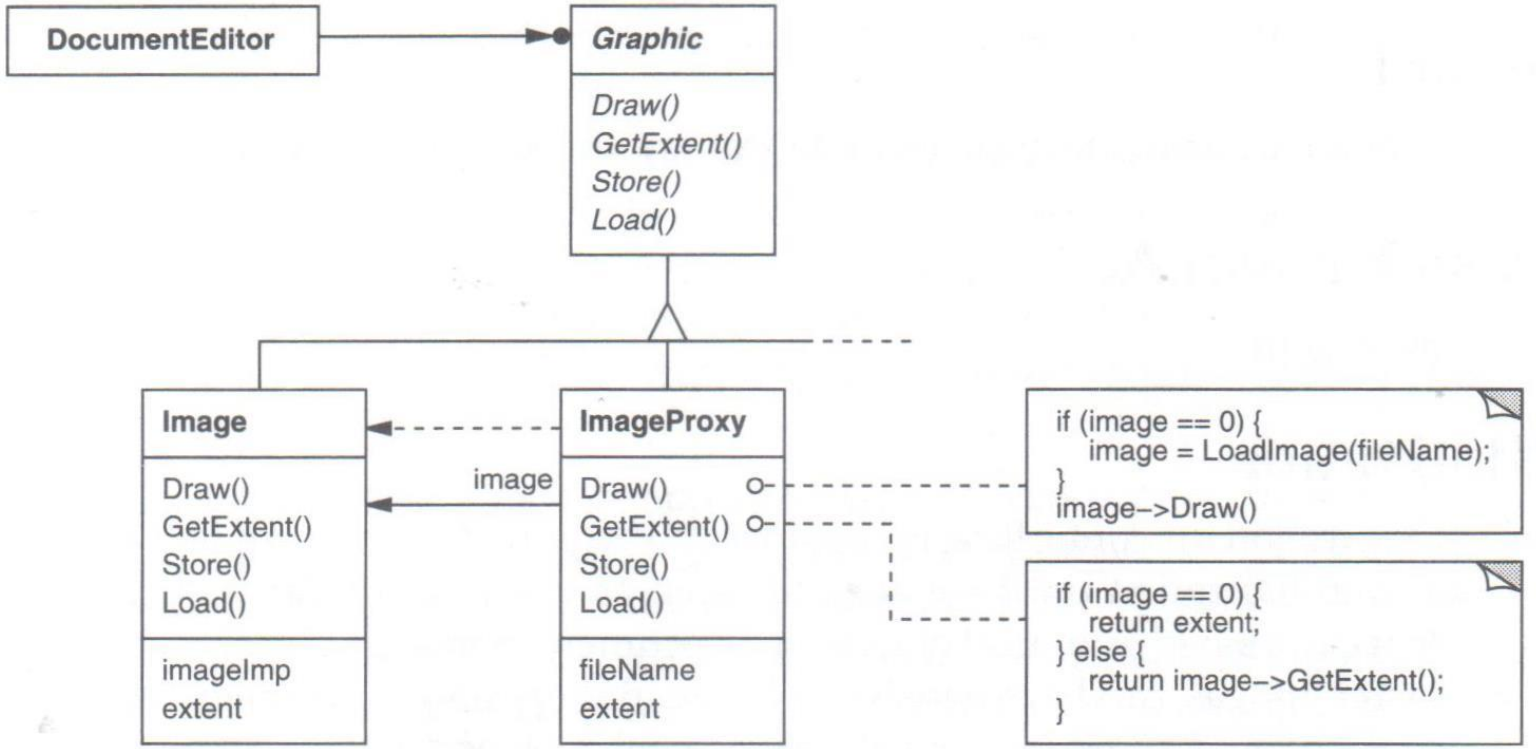
- JDK
  - RMI (Remote Method Invocation) Framework basiert auf Proxies (s. Java RMI in LE10).
  - Klasse Proxy erlaubt das dynamische Implementieren eines Interfaces.
- GoF
  - Ein Virtual Proxy für Bilder (Image).
- Point Of Sale Terminal
  - Das Umschalten auf einen lokalen Dienst wird mit einem Proxy gemacht, das zuerst den entfernten Dienst anspricht, und nur wenn dies nicht gelingt, mit dem lokalen Dienst weiterfährt.
- Code Beispiel : Remote Proxy für einen Service

Proxy: Hinweise

- Ein «Decorator» hat denselben Aufbau, aber eine andere Bedeutung. Ein «Decorator» fügt zusätzliche Funktionalität hinzu (siehe LE09)
- Persistenzframeworks verwenden Virtual Proxy Objekte, um das Erzeugen von Objekten und damit das Herunterladen der Daten zu verzögern, bis die Daten wirklich gebraucht werden.

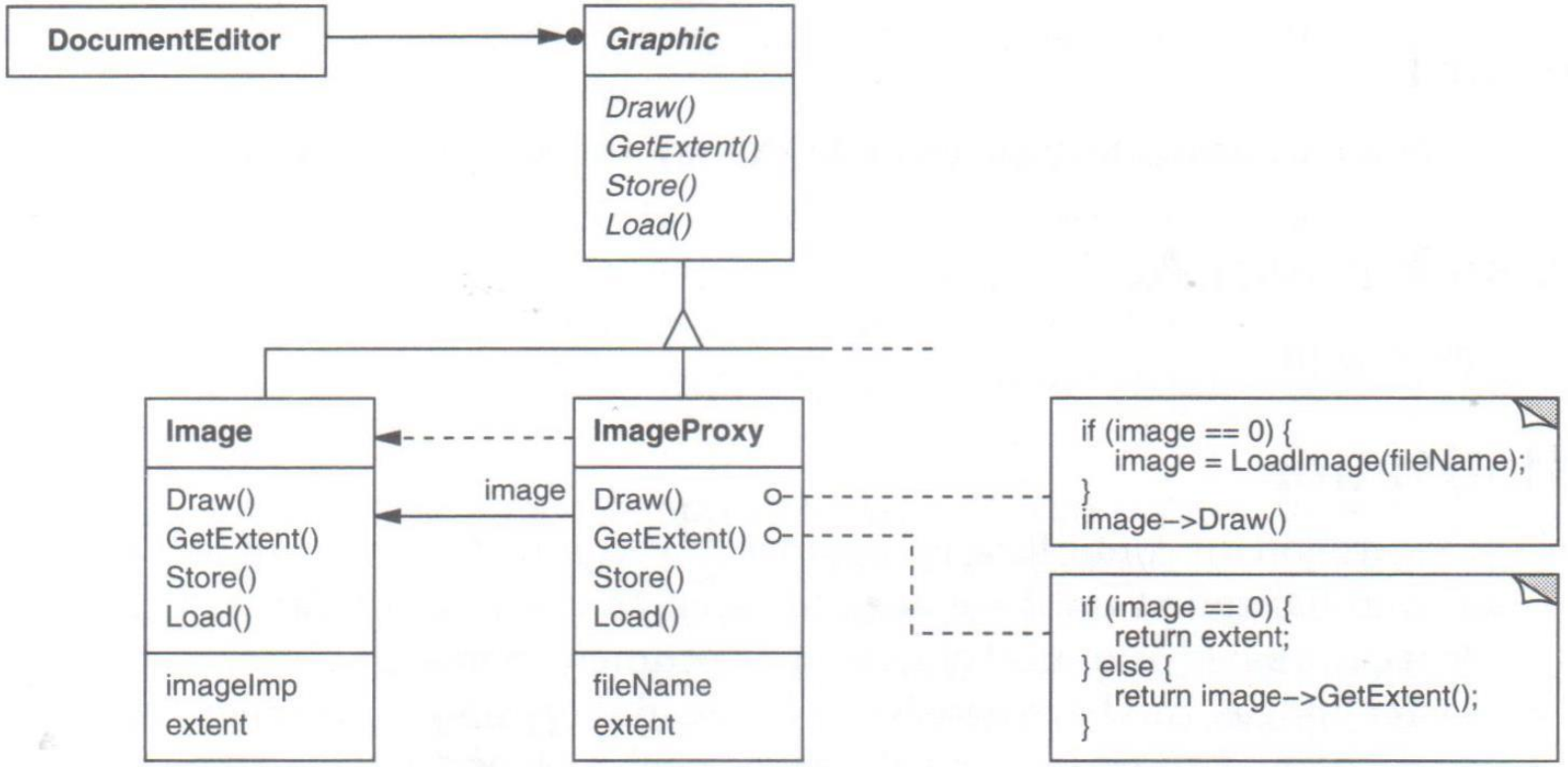
Proxy: Beispiel GoF (1/2)

- Bilder benötigen im Vergleich zu reinem Text viel mehr Speicher. Daher kann es Sinn machen, dass nur die Bilder wirklich geladen werden, die auch angezeigt werden.
- Eine ImageProxy Klasse wird gebildet, die wie die Image Klasse das Interface Graphic implementiert.



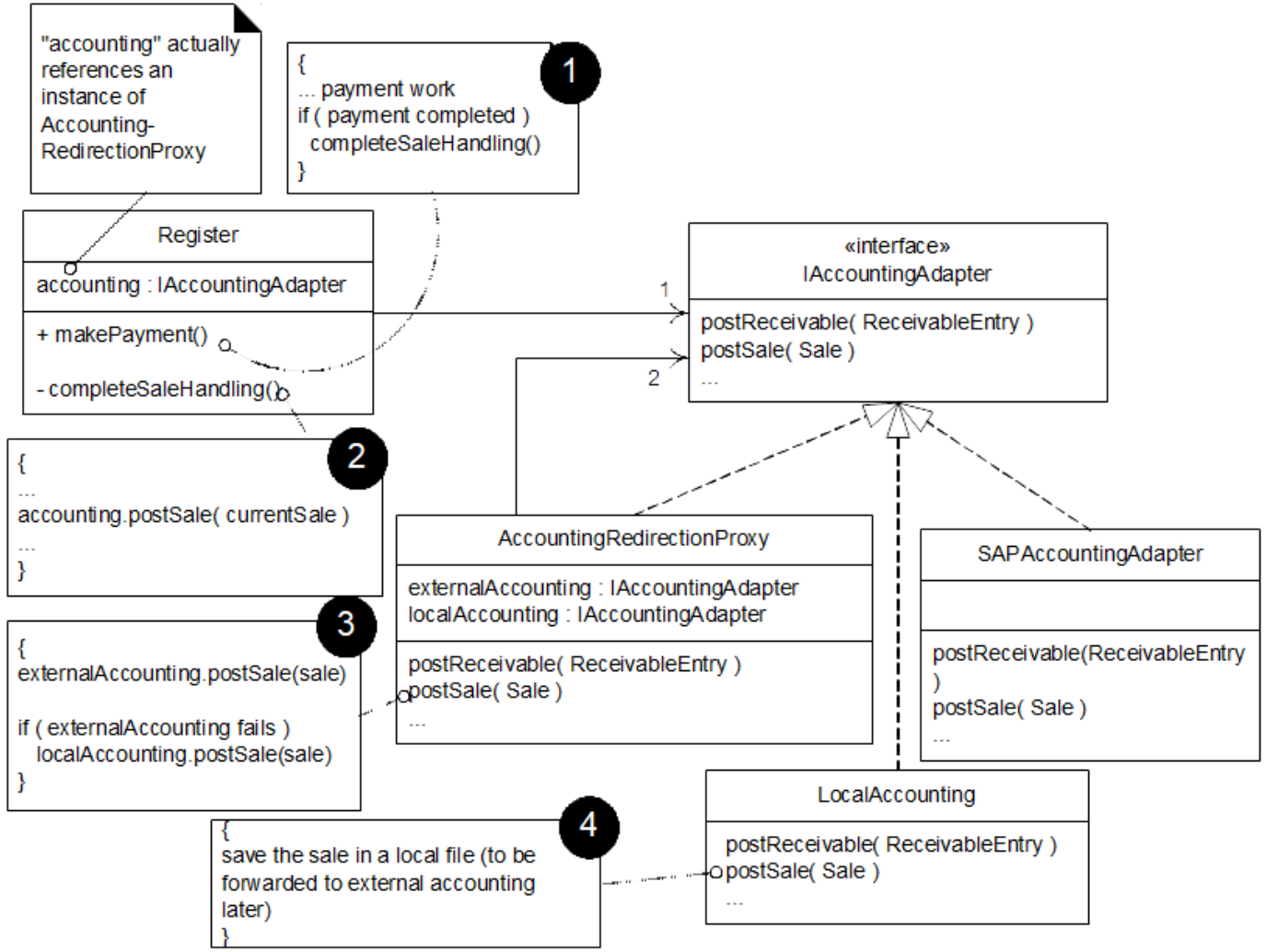
Proxy: Beispiel GoF (2/2)

- Die ImageProxy Klasse delegiert alle Methodenaufrufe an die richtige Image Klasse weiter, wobei zuerst sichergestellt wird, dass das Bild wirklich im Speicher ist.



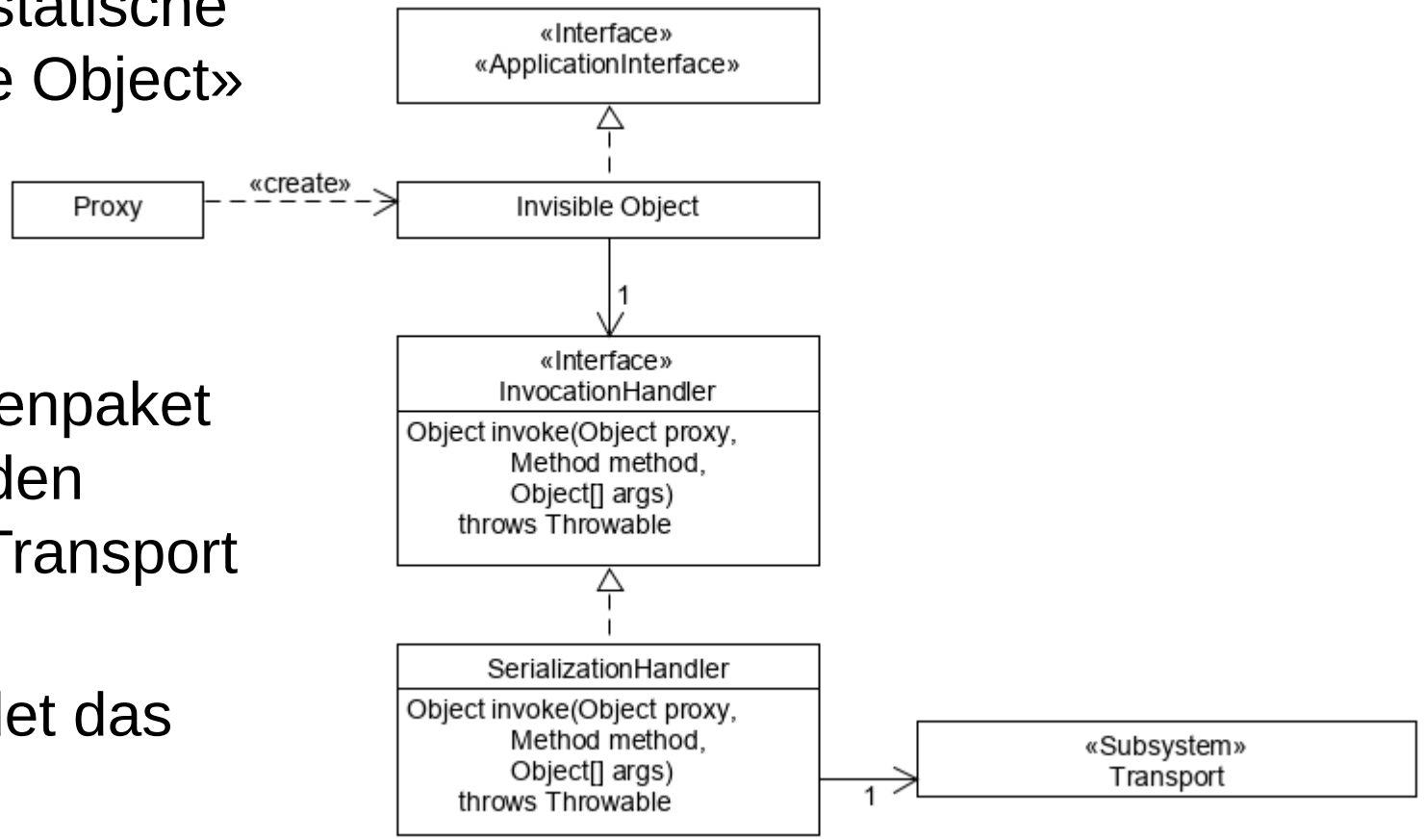
Proxy: Beispiel Point Of Sale Terminal

- Um die Ausfallsicherheit zu erhöhen, wird ein RedirectionProxy erzeugt, der entweder den richtigen Adapter (SAPAccountingAdapter) oder den lokalen Zwischenspeicher (LocalAccounting) ansteuert.



Proxy Beispiel: Remote Proxy mit JDK Proxy Klasse

- Die Proxy Klasse selber bietet statische Methoden an, um das «Invisible Object» zu erzeugen.
- Dabei wird eine Instanz eines InvocationHandler übergeben.
- Dieser Handler erzeugt ein Datenpaket mit dem Methodennamen und den Parametern und ruft damit ein Transport Subsystem auf.
- Das Transport Subsystem sendet das Paket zum Remote Computer.



Remote Proxy: Code Beispiel

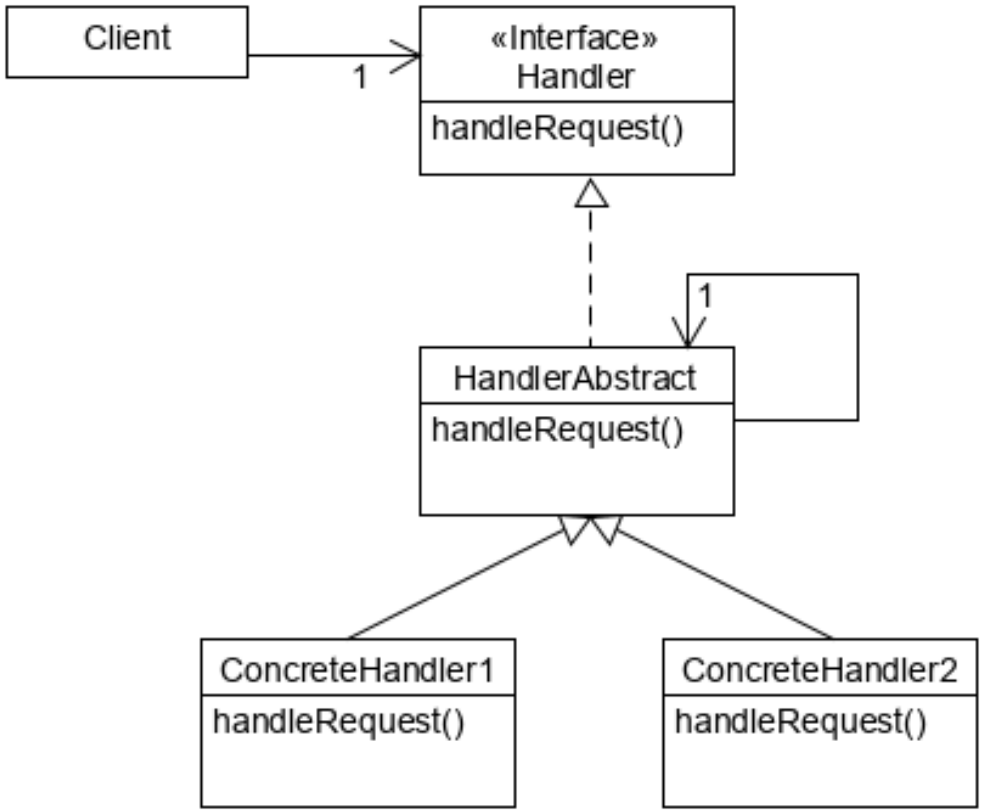
Der Zugriff auf einen Service über das HTTP Protokoll kann in einem Remote Proxy abgekapselt werden. Dabei orientiert sich die Schnittstelle des Remote Proxy an den Namen und Typen des Services. So gibt handleSentence einen String zurück, obwohl es logisch eigentlich ein boolean Wert wäre.

```
public class RemoteProxy {
    private HttpRequestFactory requestFactory;
    private HttpClient client;

    public String handleSentence(String sentence) throws IOException, InterruptedException {
        HttpRequest request = requestFactory.createHttpClient(sentence);
        HttpResponse<String> response = client.send(request, BodyHandlers.ofString());
        return response.body();
    }

    public String handleParagraph(List<String> paragraph) throws IOException, InterruptedException {
        String body = String.join("\n", paragraph);
        HttpRequest request = requestFactory.createHttpClient(body);
        HttpResponse<String> response = client.send(request, BodyHandlers.ofString());
        return response.body();
    }
```

Constructor weggelassen

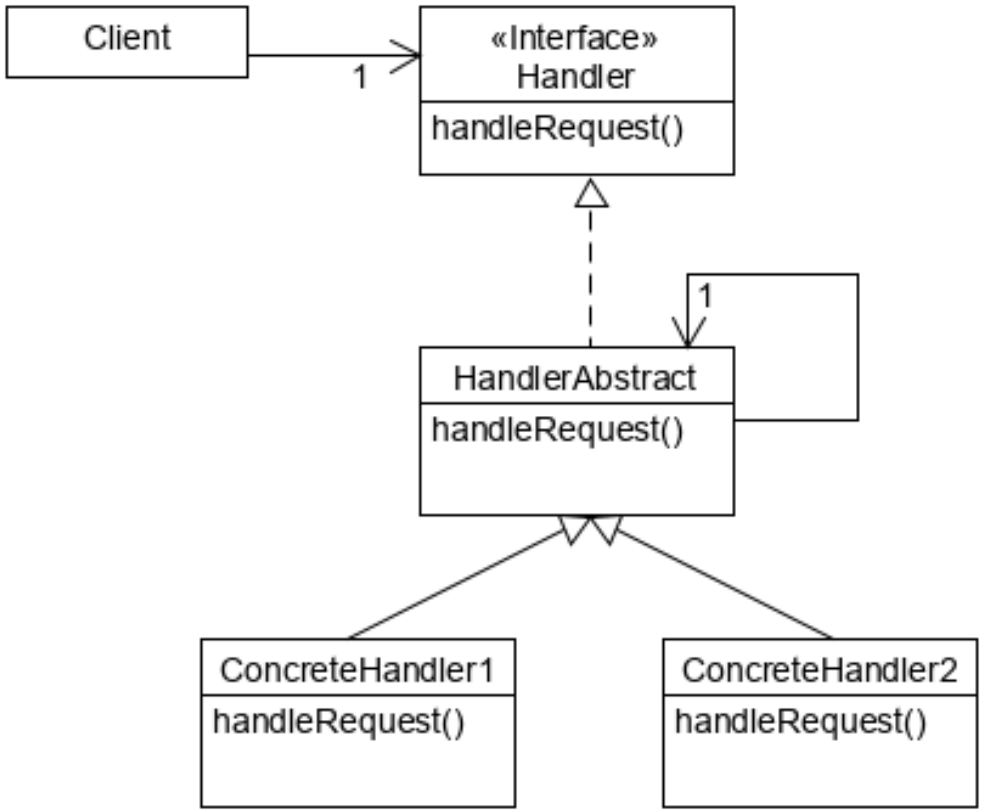
Chain of Responsibility: Problem und Lösung

- Problem
  - Für eine Anfrage gibt es potentiell mehrere Handler, aber von vornherein ist es nicht möglich (oder nur sehr schwer), den richtigen Handler herauszufinden.
- Lösung
  - Die Handler werden in einer einfach verketteten Liste hintereinandergeschaltet.
  - Jeder Handler entscheidet dann, ob der die Anfrage selber beantworten möchte oder sie an den nächsten Handler weiterleitet.



Chain of Responsibility: Hinweise

- Hinweise
  - Als Variante davon leitet jeder Handler die Anfrage an den nächsten Handler weiter, unabhängig davon, ob er sie selber behandelt oder nicht.
  - Es könnte sein, dass gar kein Handler die Anfrage behandelt.



Chain of Responsibility: Beispiele

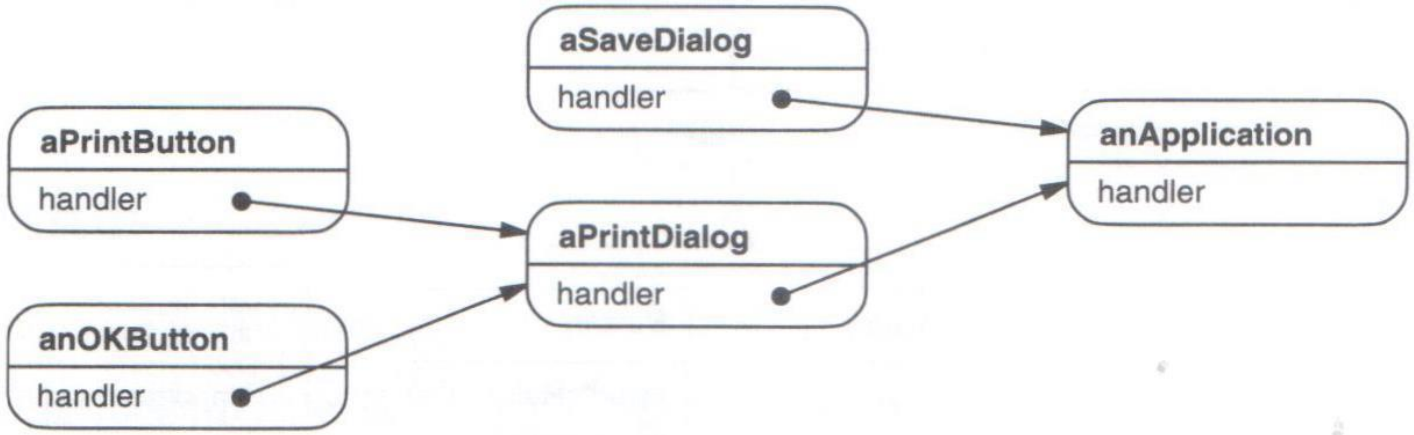
- JDK
  - Ein ClassLoader delegiert zuerst an seinen «parent» ClassLoader, bevor er selber versucht eine Klasse zu laden.
  - Die Logger Klasse sendet einen Log-Record nicht nur zu seinen eigenen Handlern, sondern auch zu seinem Parent Logger.
- GoF
  - HelpHandler, bei dem jedes Element eines Benutzerinterfaces entscheidet, ob es die Help-Anfrage selber behandelt oder an das umschliessende Element weiterleitet.
- Code Beispiel: Ein String wird von mehreren Komponenten überprüft.

Chain of Responsibility: Beispiele

- Allgemein
  - Bei allen hierarchischen Strukturen, wo eine Referenz auf den Parent-Node vorhanden ist, kann eine Anfrage zum Parent weitergeleitet werden, wenn der Node dies selber nicht bearbeiten kann.

Chain of Responsibility Beispiel: GoF

- In einem GUI Framework wird die Behandlung einer Hilfeanfrage zuerst von der visuellen Komponente behandelt, die gerade den Fokus hat.
- Ist dort keine Hilfe-Funktionalität vorhanden, wird die Anfrage der umgebenden Komponente weitergereicht, bis am Schluss die Anwendung an der Reihe ist.



Denkpause

Aufgabe 8.5 (5')

Diskutieren und bearbeiten Sie in Murmelgruppen folgende Fragen:

- Betrachten Sie das DCD von «Chain of Responsibility». Welchen Code könnte in HandlerAbstract plaziert werden, der vermutlich in allen ConcreteHandler vorhanden ist?
- Schreiben/Skizzieren Sie den Code für diese Erweiterungen.
- Wo werden neuen Handler eingefügt?
- Wie stellen Sie sicher, dass jede Anfrage behandelt wird?

Chain of Responsibility: Code Beispiel

Ein String soll überprüft werden (für was auch immer). Dafür werden verschiedene Komponenten eingesetzt, und die erste negative Überprüfung führt zu einem negativen Gesamtresultat.

```
public class ChainOfRespHandlerImpl implements ChainOfRespHandler {
    private ChainOfRespHandler next;

    @Override
    public boolean evaluateText(String text) {
        if (!evaluateHere(text)) {
            return next.evaluateText(text);
        }
        return false;
    }

    protected boolean evaluateHere(String text) {
        return text != null && text.length() > 5;
    }
}
```

Constructor weggelassen

„Irgendein Test“

Agenda

1. Einführung in Design Patterns
2. Repetition GRASP
3. Design Pattern
4. **Wrap-up und Ausblick**

Wrap-up

- Design Patterns sind wichtige Werkzeuge um gut strukturierten, wartbaren Code zu schreiben.
- Die Kombination von Singleton, Factory und Adapter wurde traditionell oft eingesetzt, um externe Dienste anzusprechen.
- Anstelle von Singleton und Factory ist vermehrt Dependency Injection (DI) vorzuziehen.
- Ein Proxy kapselt den Zugriff auf ein anderes Objekt vollständig ab und ist wie ein Stellvertreter.
- Chain of Responsibility ist dann angebracht, wenn eine Aufgabe potentiell von mehreren Handlern übernommen werden kann, aber für eine konkrete Aufgabe im voraus nicht klar ist, welcher Handler wirklich zuständig ist.

Ausblick

- In der nächsten Lerneinheit werden wir:
  - weitere Design Patterns kennenlernen und anwenden.

Quellenverzeichnis

[1] Larman, C.: UML 2 und Patterns angewendet, mitp Professional, 2005

[2] Seidel, M. et al.: UML @ Classroom: Eine Einführung in die objektorientierte Modellierung, dpunkt.verlag, 2012

[3] Martin, R. C.: Clean Architecture: A Craftsman's Guide to Software Structure and Design, mitp Professional, 2018

[4] Gamma, E et al.: Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley Longman, 1995

[5] McDonald, J: DZone Refcardz: Design Patterns, www.dzone.com, 2008