Frameworks

Christian Silberbauer

Definition

- (Brügge & Dutoit, 2004, S. 364):

> Eine wieder verwendbare Teilanwendung, die spezialisiert werden kann, um maßgeschneiderte Anwendungen zu erstellen.

- Frameworks bilden in der OOP ein Klassengerüst.

Aspekte

- Frameworks bestimmen den Kommunikationsfluss.
- Sie dienen einem bestimmten Zweck (z.B. UI, SOA, OR-Mapping, Logging).
- Es dient zur Wiederverwendung von Querschnittsbelangen für Anwendungen.
- Frameworks ermöglichen die Umsetzungen von Metaebenen/Schichten im System.

Dependency-Injection

- Die Kommunikation zwischen Framework und Anwendung erfolgt nach dem Hollywood-Prinzip:

> Don‘t call us, we call you.

- In der OOP kommuniziert das Framework über seine Schnittstellen mit der Anwendung.
- Somit ist die Anwendung an das Framework lose gekoppelt.
- Die Bindung wird per **Dependency-Injection** umgesetzt.

- Frameworks benutzen die Anwendung. Bibliotheken hingegen werden von der Anwendung benutzt.
- Beispiel: `Math.sqrt(16);`

Dependency-Injection

- Beispiel einer engen Kopplung:

```
class A {
   private B x;

   public A() {
      x = new B();
   }
}
```

```
class B {
   public void f() { }
}
```

- A ist statisch an B gebunden.

Dependency-Injection

- Beispiel einer losen Kopplung mittels DI:

```
class A {
   private I x;

   public void setDep(I x) {
      this.x = x;
   }
}
```

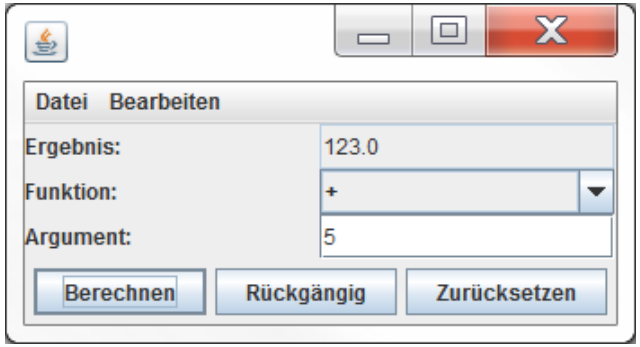
```
interface I {
   public void f();
}

class B implements I {
   public void f() { }
}
```

- A kann dynamisch an B gebunden werden.

Dependency-Injection

- Die Abhängigkeit zu B wird zur Laufzeit „injiziert“.
- B könnte zur Entwicklungszeit von A noch gar nicht programmiert sein…
- DI schafft eine typsichere, dynamische Adaptierbarkeit.
- Technische Grundlage für DI bilden Vererbung und Polymorphie
- Für eine ausführliche Beschreibung zu Dependency-Injection siehe folgenden Artikel von Martin Fowler: http://martinfowler.com/articles/injection.html

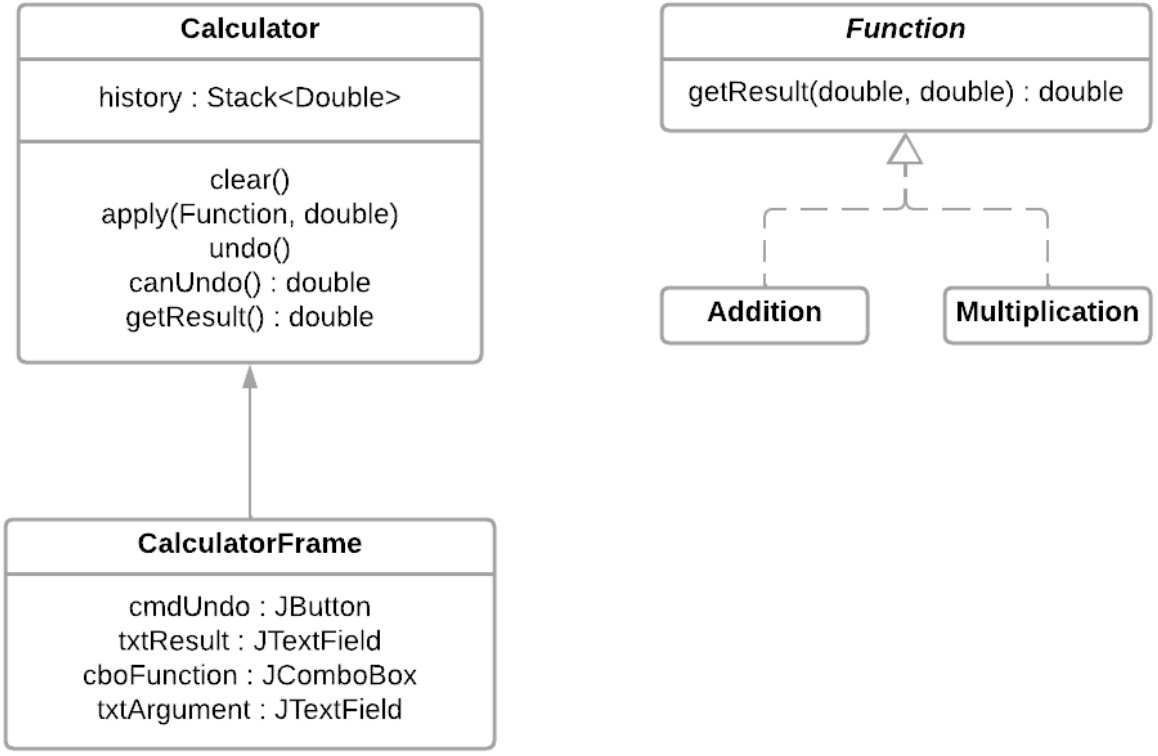
Taschenrechner-Beispiel



- (siehe auch Übungsblatt)

- Ein Taschenrechner soll implementiert werden.
- Die Klasse `Calculator` definiert hierfür den Kern.
- Diese kann mit `Function`s parametrisiert werden (z.B. `Addition`, `Multiplication`).
- `Calculator` erlaubt Berechnen, Zurücksetzen und Rückgängig.
- Die Klasse `CalculatorFrame` definiert das UI auf Basis von Java-Swing.

Architektur



Erläuterung

- `Calculator` und `Function` stellen den Kern des Taschenrechner-Frameworks dar.
- `Addition` und `Multiplication` sind gegebene Entitäten des Frameworks zur Verwendung in Anwendungen.
- Weitere solcher Klassen können im Rahmen der Framework- und Anwendungsentwicklung erstellt und adaptiert werden ohne dass eine Anpassung von `Calculator` notwendig ist.
- CalculatorFrame implementiert eine Java-Swing-UI unter Verwendung des Taschenrechner-Frameworks.

Bewertung

- Pro:
  - Taschenrechner und Taschenrechner-Funktionen sind lose gekoppelt.
  - Taschenrechner-Framework und UI-Framework sind kompatibel (keine funktionalen Überschneidungen)
- Contra:
  - Enge Kopplung an Java.

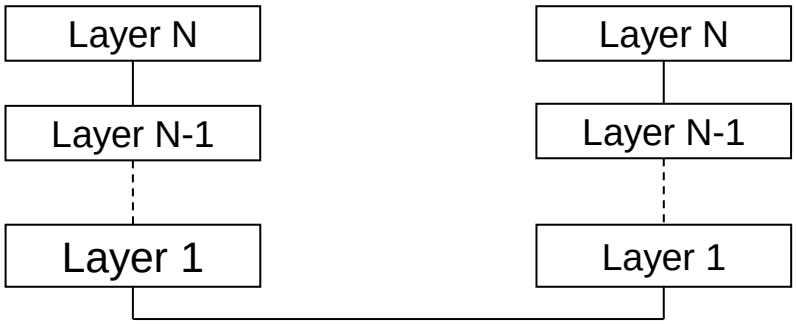
Schichtenmodell

- Frameworks ermöglichen die Umsetzungen von Metaebenen/Schichten im System.
- Virtuelle Kommunikation einer (darüberliegenden) Schichten wird so möglich.
- Die darunterliegende Schicht ist transparent.

Layers-Architekturmuster

- (Buschmann, 1996):

> The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.



- (Goll, 2014):

> Das Architekturmuster Layers modelliert die Struktur eines Systems in Schichten mit Client/Server-Beziehungen zwischen den Schichten.

Architektur- und Entwurfsmuster

- Architekturmuster:

> An architectural pattern expresses a fundamental structural organization schema for software systems. It provides a set of predefined subsystems, specifies their repsonsibilities, and includes rules and guidelines for organizing the relationships between them.

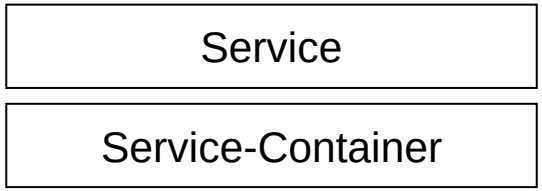
- Entwurfsmuster:

> A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communication components that solves a general design problem within a particular context.

- Patterns-Übersicht: http://www.hillside.net/patterns

- Quelle: (Buschmann, 1996, S. 12)

Service-Orientierte Architekturen



- In Service-Orientierten Architekturen (SOA) ist die Kommunikation eines Services durch einen Service-Container gekapselt.
- So gilt für den Service *Ortstransparenz*.

Counter-Beispiel

- Ein Counter-Service ist anzubieten.
- Für den Client sollen dafür entfernte Methodenaufrufe transparent sein.

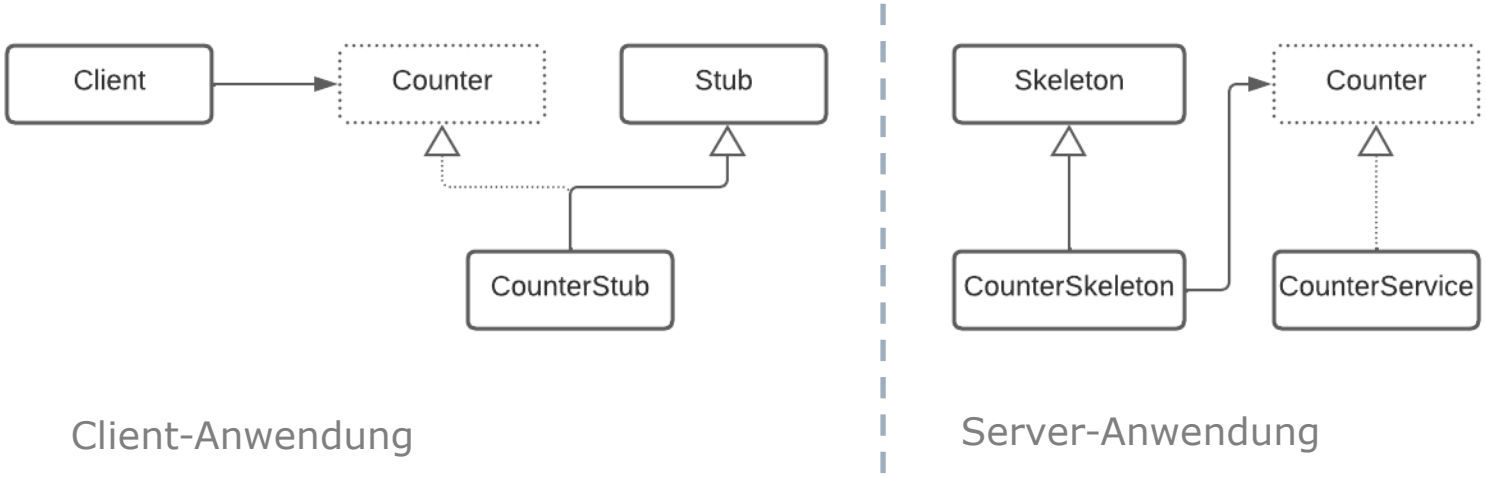
Das Interface `Counter`

```
public interface Counter {
   void setValue(int value) throws IOException;
   int getValue() throws IOException;
   void increment() throws IOException;
   void decrement() throws IOException;
}
```

Die Klasse `CounterService`

```
public class CounterService implements Counter {

   private int value = 0;

   public void setValue(int value) {
      this.value = value;
   }

   public int getValue() {
      return value;
   }

   public void increment() {
      value++;
   }

   public void decrement() {
      value--;
   }
}
```

Stub-Skeleton-Kommunikation



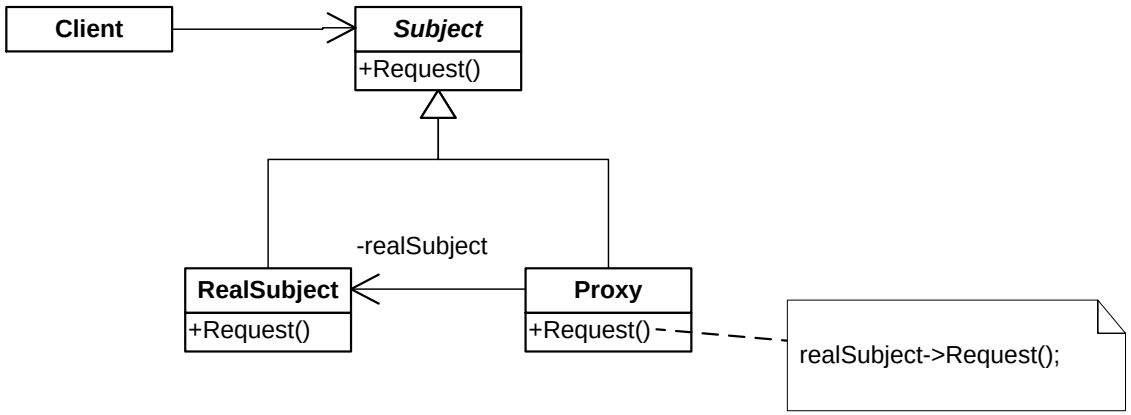
- Der `CounterStub` ermöglicht die Server-Kommunikation zum `CounterSkeleton`.
- Für den `Client` ist die Remote-Kommunikation transparent.
- Er erhält die `CounterStub`-Instanz vom `ServiceProvider`.

Proxy-Muster

- Zweck:

> Bietet einen Stellvertreter oder Platzhalter für ein anderes Objekt, um den Zugriff darauf zu kontrollieren.

- Struktur:



Proxy-Muster

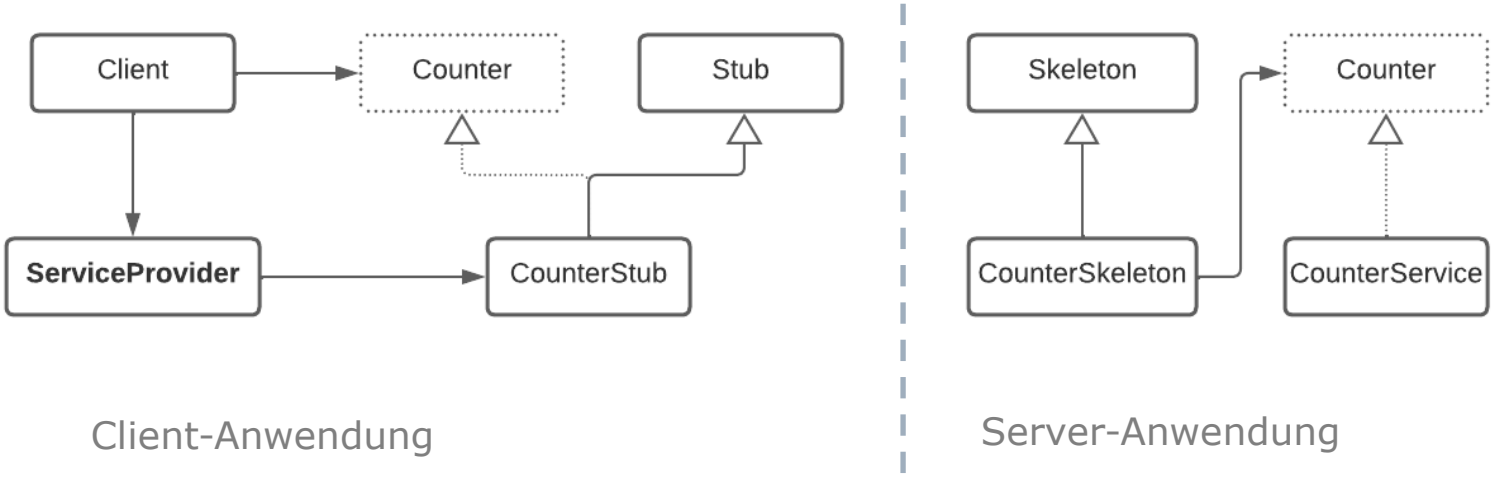
- Anwendbarkeit:

1. **Remote-Proxy**: Bietet eine lokale Repräsentation eines Objekts in einem anderen Adressbereich.
2. **Virtual-Proxy**: Erstellt ressourcenintensive Objekte bei Bedarf. Z.B. bei Zugriff auf ein Bild mit hoher Auflösung („ImageProxy“).
3. **Protection-Proxy**: Kontrolliert Zugriff auf das Originalobjekt.
4. **Smart-Reference**: Ergänzt zusätzliche Funktionalität beim Zugriff auf ein Objekt. Es zählt z.B. die Anzahl der Referenzen und erlaubt ggf. Schreibzugriff nur für eine Referenz.

Die Klasse `Client`

```
public class Client {

   public static void main(String[] args) throws Exception {

      Counter counter =
         ServiceProvider.instance().createService(ICounter.class);
      counter.setValue(1000);

      for (int i = 0; i < 3; i++) {
         counter.increment();
      }

      int r = counter.getValue();
      System.out.println(r);
   }
}
```

Dependency-Injection



- Er erhält die `CounterStub-Instanz` vom `ServiceProvider`.
- Die *dynamische* Abhängigkeit des `Client`s zum `CounterStub` erfolgt per Dependency-Injection.

Die Klasse `ServiceProvider` (Client)

```
public class ServiceProvider {

   private final static String HOST = "localhost";
   private final static int PORT = 8011;
   private static ServiceProvider theInstance = null;

   private ServiceProvider() { }

   public static ServiceProvider instance() {
      if (theInstance == null) theInstance = new ServiceProvider();
      return theInstance;
   }
   public <T> T createService(Class<T> cls) throws Exception {
      try (Socket socket = new Socket(HOST, PORT)) {
         DataInputStream in = new DataInputStream(socket.getInputStream());
         DataOutputStream out = new DataOutputStream(socket.getOutputStream());
         if (cls.equals(Counter.class)) {
            return cls.cast(new CounterStub(in, out));
         }
         return null;
      }
   }
}
```

Zum `ServiceProvider` (Client)

- Implementiert eine **Simple-Factory**.
- Es kapselt das Anlegen von Stubs und Sockets.

- Socketverbindungen könnten wiederverwendet werden.
- Dies suggeriert das **Object-Pool-Entwurfsmuster**.
- (Siehe auch Übungsblatt)

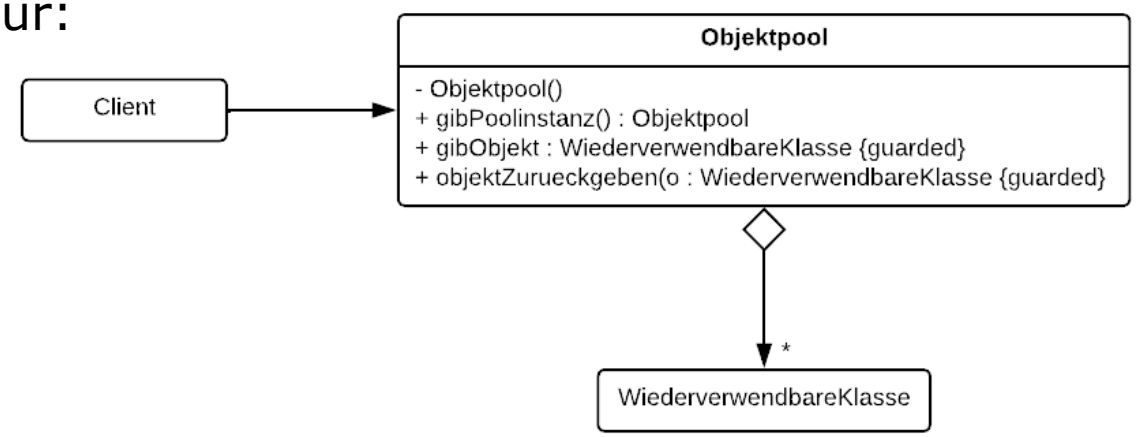
- Zudem implementer `ServiceProvider` das **Singleton-Muster**.
- So wird sichergestellt, dass es nur eine `ServiceProvider`-Instanz gibt und dass diese erst bei Bedarf erstellt wird.
- Eine *Simple-Factory* nutzt üblicherweise *Singleton* oder aber die `create`-Methode wird statisch implementiert.

Object-Pool-Muster

- Zweck:

  Ermöglicht es, Objekte aus einem Pool zur Verfügung zu stellen und wiederzuverwenden.

- Struktur:



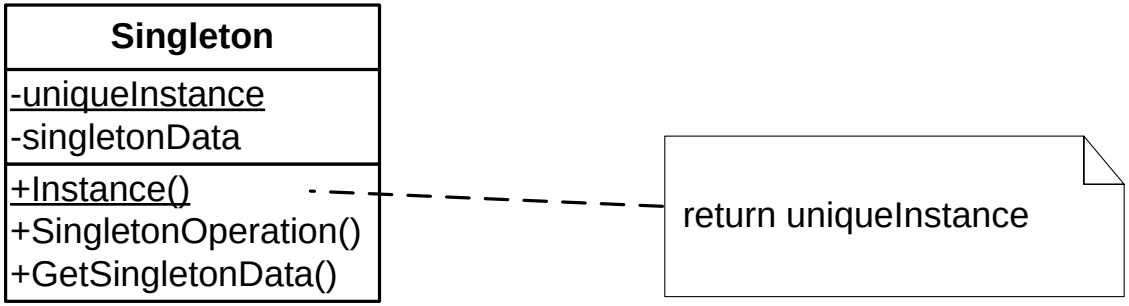
- Beispiele: Connection-, Thread-, Service-Pool

Singleton-Muster

- Zweck:

  Stellt sicher, dass es von einer Klasse nur eine Instanz gibt und bietet einen globalen Zugriff auf dieses Objekt.

- Struktur:



Die Klasse `CounterStub`

```
public class CounterStub extends Stub implements Counter {

   public CounterStub(DataInputStream in, DataOutputStream out)
                                                    throws IOException {
      super(in, out);
      out.writeInt(ServiceId.Counter.ordinal());
   }
   public void setValue(int x) throws IOException {
      out.writeInt(CounterMethodId.SetValue.ordinal());
      out.writeInt(x);
   }
   public int getValue() throws IOException {
      out.writeInt(CounterMethodId.GetValue.ordinal());
      return in.readInt();
   }
   public void increment() throws IOException {
      out.writeInt(CounterMethodId.Increment.ordinal());
   }
   public void decrement() throws IOException {
      out.writeInt(CounterMethodId.Decrement.ordinal());
   }
}
```

Die Klasse `Stub`

```
public abstract class Stub {

   protected DataInputStream in;
   protected DataOutputStream out;

   public Stub(DataInputStream in, DataOutputStream out) {
      this.in = in;
      this.out = out;
   }
}
```

ServiceId und CounterMethodId

```
public enum ServiceId {
    Unknown,
    Counter;
}
```

```
public enum CounterMethodId {
    Unknown,
    SetValue,
    GetValue,
    Increment,
    Decrement;
}
```

Die Klasse `Server`

```
public class Server {
   private final static int PORT = 8011;

   public static void main(String args[]) throws Exception {

      try (ServerSocket serverSocket = new ServerSocket(PORT)) {
         System.out.println("Server started.");

         while (true) {
            Socket socket = serverSocket.accept();
            Thread thread = new ServerThread(socket);
            thread.start();
         }
      }
   }
}
```

Die Klasse `ServiceThread`

```
public class ServiceThread extends Thread {

   private DataInputStream in;
   private DataOutputStream out;

   public ServiceThread(Socket socket) throws IOException {
      in = new DataInputStream(socket.getInputStream());
      out = new DataOutputStream(socket.getOutputStream());
   }

   …
}
```

Die Klasse `ServiceThread`

```
public class ServiceThread extends Thread {
   …
   public void run() {
      try {
         Skeleton skeleton;
         int serviceId = in.readInt();
         ServiceId sid = ServiceId.values()[serviceId];

         switch (sid) {
            case Counter: skeleton = new CounterSkeleton(in, out); break;
            default:      return;
         }
         while (true) {
            int methodId = in.readInt();
            skeleton.onRequest(methodId);
         }
      }
      catch (SocketException e) { }
      catch (IOException ex) {
         throw new RuntimeException(ex);
      }
   }
}
```

Die Klasse `CounterSkeleton`

```
public class CounterSkeleton extends Skeleton {

   private Counter service;

   public CounterSkeleton(DataInputStream in, DataOutputStream out) {
      super(in, out);
      this.service =
         ServiceProvider.instance().createService(Counter.class);
   }

   public void onRequest(int methodId) throws IOException {
      CounterMethodId mid = CounterMethodId.values()[methodId];
      int value;

      switch (mid) {
         case SetValue:  value = in.readInt();
                         service.setValue(value); break;
         case GetValue:  value = service.getValue();
                         out.writeInt(value); break;
         case Increment: service.increment(); break;
         case Decrement: service.decrement(); break;
      }
   }
}
```

Die Klasse `ServiceProvider` (Server)

```
public class ServiceProvider {

   private static ServiceProvider theInstance = null;

   private ServiceProvider() { }

   public static ServiceProvider instance() {
      if (theInstance == null) theInstance = new ServiceProvider();
      return theInstance;
   }

   public <T> T getService(Class<T> cls) {

      if (cls.equals(Counter.class)) {
         return cls.cast(new CounterService());
      }
      return null;
   }
}
```

Bewertung

- Pro:
  - Die Netzwerkkommunikation ist für die Client-Anwendung transparent.
- Contra:
  - Die Service-Einbindung ist statisch und damit redundanzbehaftet.
  - Es wird eine Socket-Verbindung pro Service und nicht pro Client erzeugt.
  - (Siehe Übungsblatt zur Behebung)

Architektur

Anwendung	`Client`	`CounterService`

Middleware	`ServiceProvider`
	`CounterStub`
		`ServiceProvider`
		`CounterSkeleton`
		`Server`
		`ServiceThread`

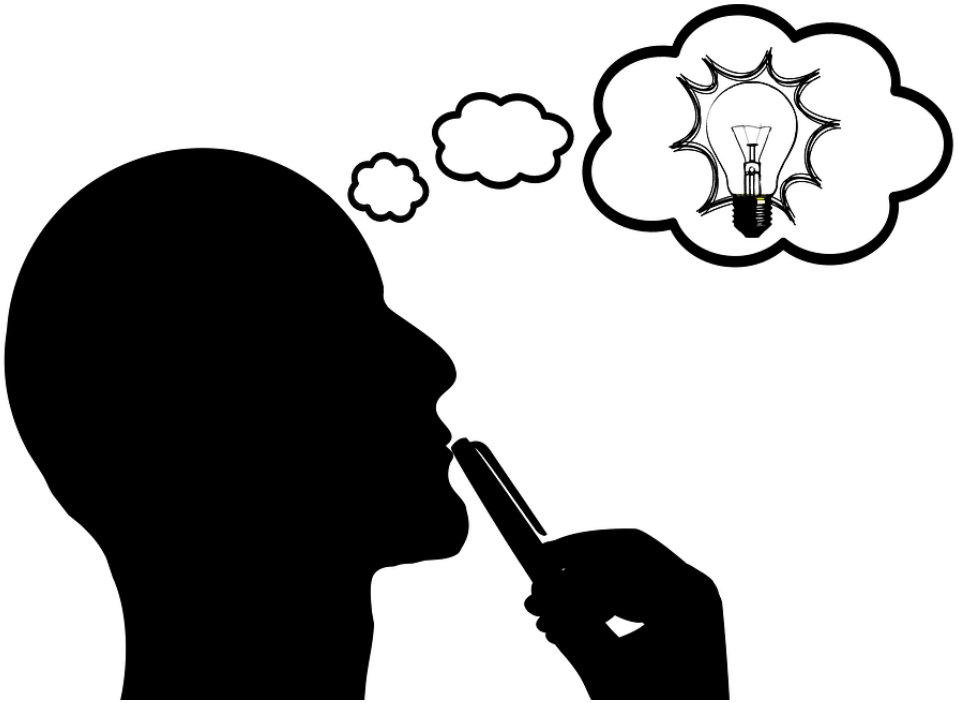
Architektur

Anwendung	`Client`	`CounterService`

Middleware	`ServiceProvider`
	`CounterStub`
		`ServiceProvider`
		`CounterSkeleton`
		`Server`
		`ServiceThread`

- Die Middleware-Schicht sollte frei sein von manuell zu entwickelnden Service-spezifischen Klassen.
- Sie sollte auch frei sein von manuellen Service-spezifischen Anpassungen (beide `ServiceProvider` und `ServiceThread`)

Was tun?



Das tun!

Merksatz: **Achte auf Standards!**

Remote Method Invocation (RMI)

- RMI ist der Quasi-Standard für entfernte Methodenaufrufe.
- Hierzu ein Beispiel…

RMI – Hello World

```
public class Server {
   public static void main(String args[]) throws Exception {
      LocateRegistry.createRegistry(1097);
      Naming.rebind("rmi://localhost:1097/HelloRmi",
         new HelloRmiService());
      System.out.println("Server started.");
   }
}
```

```
public class Client {
   public static void main(String[] args) throws Exception {
      HelloRmi helloRmi =
         (HelloRmi)Naming.lookup("rmi://localhost:1097/HelloRmi");
      System.out.println(helloRmi.sayHello());
   }
}
```

RMI – Hello World

```
public interface HelloRmi extends Remote {
   String sayHello() throws RemoteException;
}
```

```
public class HelloRmiService extends UnicastRemoteObject
                             implements HelloRmi {

   private static final long serialVersionUID = -2066553550108422861L;

   public HelloRmiService() throws RemoteException { }

   public String sayHello() {
      return "Hello, RMI!";
   }
}
```

RMI

- Weitere Informationen sind hier zu finden: https://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html

Literatur

- Brügge B. & Dutoit A.H., 2004. *Objektorientierte Softwaretechnik*. Person Studium.
- Buschmann, F. et al., 1996. *Pattern-Oriented Software Architecture, Vol. 1: A System of Patterns.* s.l.:Wiley, West Sussex, England.
- Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1995. *Design Patterns.* Boston: Addison-Wesley.
- Goll, J., 2014. *Architektur- und Entwurfsmuster der Softwaretechnik, 2. Auflage*.