Verteilte Systeme – Übung

Stubs & Skeletons

Sommersemester 2021

Michael Eischer, Laura Lawniczak, Tobias Distler

Friedrich-Alexander-Universität Erlangen-Nürnberg
Lehrstuhl Informatik 4 (Verteilte Systeme und Betriebssysteme)
www4.cs.fau.de

Lehrstuhl für Verteilte Systeme
und Betriebssysteme

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

TECHNISCHE FAKULTÄT

Überblick

Aufgabe 2

Java Reflection API

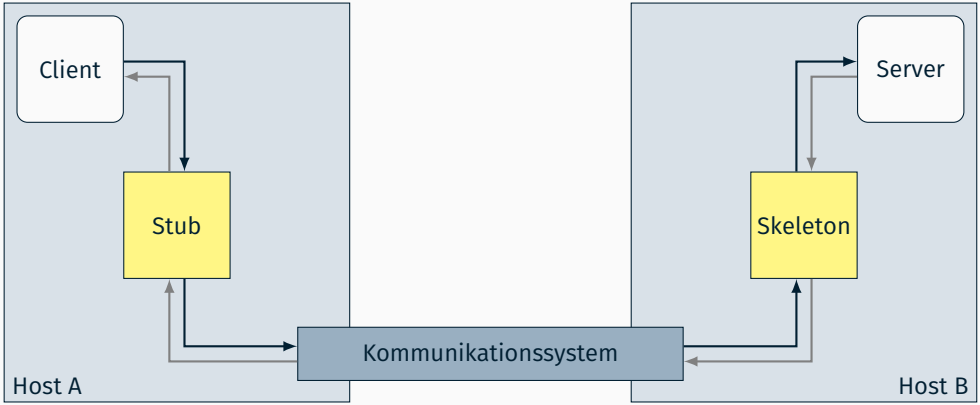
Stubs & Skeletons

- Dynamische Proxies als Stubs
- Generische Skeletons
- Identifikation von Remote-Objekten

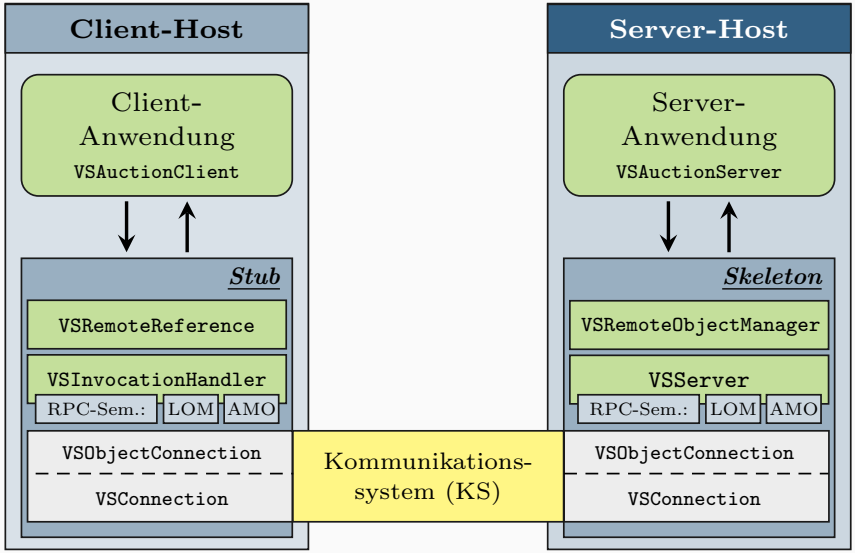
Aufgabe 2

Übungsaufgabe 2

- Dynamische Stubs und Skeletons
- Unterstützung von Rückrufen



Übungsaufgabe 2



Exceptions

- Ziel: Transparente Fernaufrufe
  - Normalfall: Rückgabe des Ergebnisses
  - Fehlerfall: Abbruch der Ausführung auf Server-Seite (Exception)
    → Fernaufrufsystem muss Exception zum Aufrufer propagieren

- Konsequenz für den Fehlerfall
  - Fangen der Exception beim Methodenaufruf auf Server-Seite → `InvocationTargetException`
  - Weiterleitung der Exception zur Client-Seite
  - Werfen der Exception im Stub

- Im Fernaufruf bedingte Fehler
  - Beispiele
    - Unerreichbarer Server
    - Verbindungsabbruch
    - ...

  → Fernaufrufsystem muss Fehler (soweit möglich) behandeln
  [Näheres in Übungsaufgabe 3]

Unterstützung von Rückrufen

- Parameterübergabe (analog zu Java RMI)
  - Call-by-Value (Standard): Übertragung einer Kopie des Parameters
  - Call-by-Reference: Übertragung eines Stub für den Parameter
    - Parameter implementiert `Remote`-Schnittstelle
    - Parameterobjekt wurde zuvor exportiert

- Implementierung
  - Erweitertes Marshalling im Invocation-Handler des Stub
    - Analyse der Aufrufparameter
    - Unterscheidung der Parameterübergabearten
  - Beachte: Call-by-Reference ist auch relevant für Rückgabewerte

- `Class.isAssignableFrom()`: Überprüfung, ob ein Objekt `o` eine bestimmte Schnittstelle (z. B. `Serializable`) implementiert

```
Object o = [...];
if(Serializable.class.isAssignableFrom(o.getClass())) {[...]}
```

Java Reflection API

- Bietet die Möglichkeit, das Laufzeitverhalten von Applikationen zu analysieren und es gegebenenfalls sogar zu beeinflussen
- Tutorial: `http://docs.oracle.com/javase/tutorial/reflect/index.html`

> "[...] This is a relatively advanced feature and **should be used only by developers** who have a **strong grasp of the fundamentals of the language**. [...]"

- Ermöglicht zur Laufzeit
  - Analyse von Attributen, Konstruktoren, Methoden, ...
  - Erzeugung neuer Objekte
  - Modifikation bestehender Objekte
  - Dynamische Methodenaufrufe
  - ...

Class-Objekte

- Zentrale Klasse: `java.lang.Class`
  - Pro Objekttyp existiert ein unveränderliches `Class`-Objekt
  - Beispiel

```
String x = "x";
String y = "y";
boolean b = (x.getClass() == y.getClass());  // -> b == true
```

- Zugriff auf `Class`-Objekte
  - Allgemein: Per `class`-Attribut [Funktioniert auch bei primitiven Datentypen.]

```
Class c = <Klassenname>.class;
```

  - Über existierendes Objekt mit `getClass()`

```
Class c = <Objekt>.getClass();
```

  - Über Klassenname mit `forName()`

```
Class c = Class.forName(<Klassenname>);
```

- Analyse einer Klasse

```
public class Class<T> {
    public Class<? super T> getSuperclass();
    public Class<?>[] getInterfaces();
    public Method[] getMethods();
    [...]
}
```

`getSuperClass()` Zugriff auf `Class`-Objekt der Oberklasse

`getInterfaces()` Zugriff auf `Class`-Objekte der von dieser Klasse direkt implementierten Schnittstellen

`getMethods()` Rückgabe der öffentlichen Methoden dieser Klasse

- Beispiel: Ausgabe aller Methoden der implementierten Schnittstellen

```
Class<?> c = <Objekt>.getClass();
do {
    for(Class<?> intf: c.getInterfaces()) {
       for(Method m: intf.getMethods()) System.out.println(m);
    }
} while((c = c.getSuperclass()) != null);
```

- Analyse einer Methode: `java.lang.reflect.Method`

```
public class Method {
    public String getName();
    public Class<?>[] getParameterTypes();
    public Class<?> getReturnType();
    public Class<?>[] getExceptionTypes();
    public String toGenericString();
    [...]
}
```

`getName()`	Rückgabe des Methodennamens
`getParameterTypes()`	Zugriff auf `Class`-Objekte der Parameter
`getReturnType()`	Zugriff auf `Class`-Objekt des Rückgabewerts
`getExceptionTypes()`	Zugriff auf `Class`-Objekte der Exceptions
`toGenericString()`	Rückgabe der kompletten Methodensignatur

- Dynamischer Aufruf einer Methode

```
public class Method {
    public Object invoke(Object obj, Object... args);
}
```

- Beispiel: `registerAuction()`-Methodenaufruf am `VSAuctionService` aus Übungsaufgabe 1
  - Gewöhnlicher `registerAuction()`-Methodenaufruf

```
VSAuctionService service  = new VSAuctionServiceImpl();
service.registerAuction(new VSAuction("Testauktion", 1), 42, null);
```

  - `registerAuction()`-Methodenaufruf mit Java Reflection API

```
VSAuctionService service  = new VSAuctionServiceImpl();

// Holen des Methoden-Objekts fuer registerAuction()
Class<?> c = service.getClass();
Class<?>[] paramTypes = new Class<?>[]{ VSAuction.class, int.class, VSAuctionEventHandler.class };
Method m = c.getMethod("registerAuction", paramTypes);

// Zusammenstellung der Parameter und Aufruf der Methode
Object[] params = new Object[]{ new VSAuction("Testauktion", 1), 42, null };
m.invoke(service, params);
```

[Wie das Beispiel verdeutlicht, gibt es keinen Grund, für den Aufruf einer Methode die Java Reflection API zu verwenden, solange alles Mögliche unternommen wurde, dies zu verhindern.]

Stubs & Skeletons

Dynamische Proxies als Stubs

- Stellvertreter des entfernten Objekts beim Aufrufer einer Methode
  $\rightarrow$ Implementierung der **Schnittstelle des entfernten Objekts**

- Zentrale Aufgabe: Umwandlung eines lokalen Methodenaufrufs am Stub in einen Fernaufruf am entfernten Objekt
  - Erzeugung einer Anfragenachricht
    - Eindeutige Kennung des Server-Prozesses
    - Eindeutige Kennung des entfernten Objekts
    - Eindeutige Kennung der aufzurufenden Methode
    - Einpacken der Aufrufparameter
  - Senden der Anfragenachricht über das Kommunikationssystem
  - Empfang einer Antwortnachricht über das Kommunikationssystem
  - Auspacken des Rückgabewerts
  - Übergabe des Rückgabewerts an den Aufrufer

- Schnittstelle

```
public interface VSHelloInterface {
    public void setName(String name);
    public String getName();
    public void sayHello();
}
```

- Implementierung

```
public class VSHelloImpl implements VSHelloInterface {
    private String name;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void sayHello() {
        System.out.println("Hallo " + name);
    }
}
```

Manuelle Implementierung von Stubs

- Stub für Beispiel-Schnittstelle

```
public class VSHelloStub implements VSHelloInterface {
    public void setName(String name) {
        // Anfrage (IDs, Parameter) erstellen und senden
        // Fuer synchronen Aufruf: Antwort empfangen
    }
    public String getName() {
        // Anfrage (IDs) erstellen und senden
        String s = [...] // Antwort empfangen und auspacken
        return s;
    }
    public void sayHello() {
        // Anfrage (IDs) erstellen und senden
        // Fuer synchronen Aufruf: Antwort empfangen
    }
}
```

- Nachteile einer manuellen Implementierung
  - Hoher Implementierungsaufwand (vor allem bei Schnittstellenänderungen)
  - Code-Duplikation
  - Fehleranfällig

- Grundidee
  - Zur Laufzeit generierte Stellvertreterobjekte
    → Konfigurierbare Schnittstellen
  - Umleitung von lokalen Methodenaufrufen am Proxy auf einen zuvor registrierten **Invocation-Handler**
    → Anwendungsspezifische Implementierung des Invocation-Handler
  - Weiterführende Informationen

    Dynamic Proxy Classes

    `http://docs.oracle.com/javase/7/docs/technotes/guides/reflection/proxy.html`

    Dynamic Proxies – Short Tutorial

    `http://www.javaspecialists.eu/archive/Issue005.html`

- Dynamische Proxies als Stubs
  - Implementierung beliebiger Schnittstellen
    → Proxies können als Stellvertreter für entfernte Objekte dienen
  - Abfangen von lokalen Methodenaufrufen
    → Umwandlung in Fernaufrufe

- Implementierung eines Invocation-Handler
  - Bereitstellung einer `invoke()`-Methode, an die sämtliche am Proxy getätigten Methodenaufrufe delegiert werden
  - Wissen über Methodenname und -parameter des ursprünglichen Aufrufs
  - Rückgabewert von `invoke()` $\rightarrow$ Rückgabewert des ursprünglichen Aufrufs

- Schnittstelle: `java.lang.reflect.InvocationHandler`

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

[Nicht zu verwechseln mit der `invoke()`-Methode der Java Reflection Klasse `Method`]

  - Parameter:

`proxy`	Der Proxy, an dem die `invoke`-Methode aufgerufen wurde
`method`	Das `Method`-Objekt der aufgerufenen Proxy-Methode
`args`	Array mit den Parametern des ursprünglichen Methodenaufrufs [Falls kein Parameter übergeben wurde: `args == null`]

  - Die `invoke()`-Methode darf nur die Exceptions (`Throwable`) werfen, die in der Signatur der aufgerufenen Methode enthalten sind

- Proxy-Erzeugung mittels `Proxy.newProxyInstance()`

```
static Object newProxyInstance(ClassLoader loader, Class[] interfaces, InvocationHandler handler);
```

`loader` Class-Loader für die Proxy-Klasse
[Typischerweise der Class-Loader der zu implementierenden Schnittstelle; dieser lässt sich durch den Aufruf von `getClassLoader()` am `Class`-Objekt der Schnittstelle bestimmen.]

`interfaces` Array der zu implementierenden Schnittstellen-Klassen

`handler` Instanz des Invocation-Handler

- Nach der Erzeugung des Proxy-Objekts kann dieses als Stellvertreter für die eigentliche Implementierung der vom Proxy bereitgestellten Schnittstellen genutzt werden

- Umleitung eines Methodenaufrufs auf ein lokales Objekt

```
public class VSHelloInvHandler implements InvocationHandler {
    private VSHelloInterface object;

    public VSHelloInvHandler(VSHelloInterface object) {
        this.object = object;
    }

    // Handler-Methode fuer alle lokalen Aufrufe am Proxy
    public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
        System.out.println("[Proxy] Methode: " + method.getName());
        if(args != null) {
           System.out.println("[Proxy] Args: " + args.length);
        }
        return method.invoke(object, args);  // Eigentlicher Aufruf
    }
}
```

- Main-Methode (der Klasse `VSHelloTest`) zum Testen des Proxy

```
public static void main(String[] args) {
    // Erzeugung des eigentlichen Objekts
    VSHelloInterface object = new VSHelloImpl();

    // Erzeugung eines Invocation-Handler
    VSHelloInvHandler handler = new VSHelloInvHandler(object);

    // Proxy-Erzeugung
    ClassLoader ldr = ClassLoader.getSystemClassLoader();
    Class<?>[] intfs = new Class[] { VSHelloInterface.class };
    VSHelloInterface proxy = (VSHelloInterface) Proxy.newProxyInstance(ldr, intfs, handler);

    // Test: Methodenaufrufe am Proxy
    proxy.setName("Benutzer");
    proxy.sayHello();
    System.out.println(proxy.getName());
}
```

- Beispiel-Ausführung

```
> java VSHelloTest

[Proxy] Methode: setName
[Proxy] Args: 1
[Proxy] Methode: sayHello
Hallo Benutzer
[Proxy] Methode: getName
Benutzer
```

- Jeder Aufruf einer Methode an dem Objekt `proxy` wird durch den dynamisch generierten Proxy an die `invoke()`-Methode von `VSHelloInvHandler` weitergegeben
- Im verteilten Fall erfolgt im Invocation-Handler der Fernaufruf am entfernten Objekt

Stubs & Skeletons

Generische Skeletons

- Stellvertreter des Aufrufers einer Methode beim eigentlichen Objekt
  $\rightarrow$ Imitieren des Verhaltens eines lokalen Aufrufers

- Zentrale Aufgabe: Ausführung des eigentlichen Methodenaufrufs
  - Empfang einer Anfragenachricht über das Kommunikationssystem
    - Auspacken der Kennung des (jetzt lokalen) Objekts
    - Auspacken der Kennung der aufzurufenden Methode
    - Auspacken der Aufrufparameter
  - Bestimmung des Objekts mittels Kennung
  - Methodenaufruf am Objekt
  - Erzeugen einer Antwortnachricht mit dem Rückgabewert
  - Senden der Antwortnachricht über das Kommunikationssystem

Generische Skeletons

- Idee
  - Gemeinsame Skeleton-Implementierung für alle Fernaufrufe
  - Methodenaufrufe per Java Reflection API
- Problemstellung: Wie finde ich die richtige Methode?
  - Methodenname ist nicht eindeutig
  - Parameteranzahl ist nicht eindeutig
  - → Kombination aus Methodename sowie Anzahl und Typen aller Parameter
- Lösungsansatz
  - Eindeutige Kennung per `Method.toGenericString()`
  - Beispiel: `VSAuctionService.registerAuction()`

```
public abstract void vsue.rmi.VSAuctionService.registerAuction(  // Methodenname
        vsue.rmi.VSAuction, int, vsue.rmi.VSAuctionEventHandler) // Parameter
    throws vsue.rmi.VSAuctionException, java.rmi.RemoteException // Exceptions
```

  - Bestimmung und Verwendung des richtigen `Method`-Objekts
    1. Abfrage aller Remote-Schnittstellen des Remote-Objekts
    2. Abfrage aller Methoden dieser Schnittstellen
    3. Vergleich der generischen Methoden-Strings mit dem in der Anfrage
    4. Aufruf von `invoke()` am gefundenen Methoden-Objekt

Stubs & Skeletons

Identifikation von Remote-Objekten

Referenzierung und Implementierung der Server-Seite

- Remote-Referenz: `VSRemoteReference`

```
public class VSRemoteReference implements Serializable {
    private String host;
    private int port;
    private int objectID;
}
```

`host`	Host-Name des Servers
`port`	Port-Nummer des Servers für Verbindungsaufbau
`objectID`	Objekt-ID für Zugriff auf Remote-Objekt

- Verwaltung von Verbindungen für Remote-Objekte: Anpassung von `VSServer`
  - Empfangen und Bearbeiten von Anfragen
  - Erzeugen und Senden von Antworten

Verwaltung von Remote-Objekten


```
public class VSRemoteObjectManager {
    public static VSRemoteObjectManager getInstance();
    public Remote exportObject(Remote object);
    public Object invokeMethod(int objectID, String genericMethodName, Object[] args);
}
```

- Implementierung als *Singleton*
  - Nur eine Instanz pro Java Virtual Machine
  - Zugriff über statische `getInstance()`-Methode
- Export von Objekten
  - Bereitstellung dynamischer Proxies für Fernaufrufe
  - Verwaltung der exportierten Remote-Objekte
- Aufruf von Methoden an exportierten Objekten
  - Suche des Objekts anhand der Objekt-ID
  - Bestimmung der Methode über ihren generischen Namen
  - Aufruf der Methode mit den übergebenen Parametern
  - Rückgabe des Rückgabewerts der aufgerufenen Methode

`ObjectInputFilter` der RMI-Registry

- `LocateRegistry` prüft seit Java `1.8.0_121` `bind`-Aufrufe von Clients

```
Remote vsproxy = VSRemoteObjectManager.getInstance().exportObject([...]);
Registry registry = LocateRegistry.getRegistry([...]);
registry.bind("name", vsproxy);
```

  Beispiel wirft eine `InvalidClassException` wegen `VSInvocationHandler`:

  - Proxy-Objekt `vsproxy` enthält den `VSInvocationHandler`
  - `getRegistry` gibt einen **Registry-Stub** zurück
  - Aufruf von `bind` löst einen Fernaufruf aus
  - `LocateRegistry` akzeptiert bei Fernaufrufen nur primitive Datentypen, Strings, Objekte, die die Remote-Schnittstelle implementieren, bestimmte RMI-Klassen sowie Arrays all dieser Datentypen

- Problemlösungen
  - `bind` direkt auf der Registry aufrufen (siehe Folie 1–11)

```
Registry registry = LocateRegistry.createRegistry(12345);
registry.bind(name, vsproxy);
```

  - Eigene Klassen erlauben. Vor Aufruf von `createRegistry` einfügen:

```
System.setProperty("sun.rmi.registry.registryFilter", "vsue.**");
```