Programmierung mit Komponenten

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Grundidee: Teile und Herrsche

Das *Aufteilen eines Ganzen in wohldefinierte Teile* ist ein drittes Grundprinzip der Softwaretechnik.

Klassische Teile eines Systems:

- 1970: Funktionen
- 1980: Module
- 1990: Klassen (+ Aspekte)
- 2000: Komponenten

Wozu Komponenten?

Klassisch: Komponenten eines Systems werden zur Übersetzungszeit *gebunden* und können nicht länger getrennt oder ausgetauscht werden.

Dieser Ansatz genügt nicht mehr den heutigen Bedürfnissen:

- Erwünscht sind Kombinationen aus selbst geschaffenen und gekauften Komponenten (*components of the shelf, COTS*)
- Unterstützung von *Produktfamilien* aus *individuellen Zusammenstellungen* von Komponenten
- Verteilte Systeme aus *dynamisch austauschbaren* Komponenten

Was ist eine Komponente?

Eine Komponente stellt *Dienste* bereit (wie ein Modul oder ein Objekt).

Darüber hinaus aber

- kann eine Komponente als Einheit *unabhängig* verteilt werden.
- kann eine Komponente als Einheit *von Dritten* zusammengesetzt werden
- hat eine Komponente keinen persistenten Zustand

Unabhängigkeit

Eine Komponente kann als Einheit unabhängig verteilt werden.

Folgen:

- Eine Komponente ist getrennt von ihrer Umgebung (und anderen Komponenten)
- Eine Komponente kapselt alles ein, was sie benötigt
- Eine Komponente wird nicht teilweise verteilt

Zusammensetzung durch Dritte

Eine Komponente kann als Einheit von Dritten zusammengesetzt werden.

Folgen:

- Eine Komponente darf keine besonderen Konstruktionsschritte benötigen
- Eine Komponente muß spezifizieren, was sie braucht und was sie zur Verfügung stellt
- Eine Komponente darf nur über diese Schnittstellen kommunizieren
- Die benötigten Dienste der Komponente sollten konfigurierbar sein

Kein persistenter Zustand

Eine Komponente hat keinen persistenten Zustand.

Folgen:

- Eine Komponente kann nicht von ihrer Kopie unterschieden werden.

Beispiel: Unterscheide die

- *Datenbank* (die persistente Daten speichert) von der
- *Datenbank-Komponente* (die den Zugang bereitstellt)

Komponenten und Objekte

Was steckt in einer Komponente?

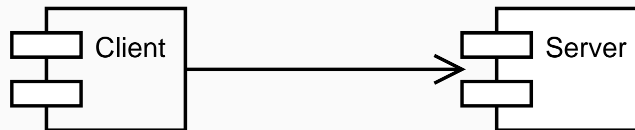
Komponenten enthalten in der Regel mehrere Klassen, aus denen *Objekte* erzeugt werden können.

Ein Objekt enthält eine *Identität*, einen *Zustand* und eine *Ursprungsklasse.*

Objekte sind grundsätzlich *verteilt.*

Referenzen

Ein *client*-Objekt verfügt über eine *Referenz* auf ein *Server*-Objekt, wenn es eine Dienstleistung des Servers in Anspruch nehmen möchte.



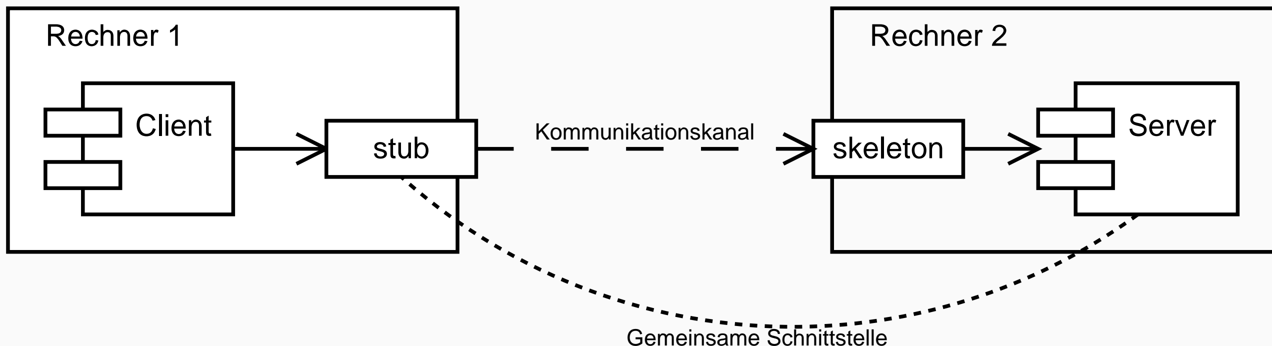
Diese Referenz erhält es als Ergebnis eines Dienstaufrufs – oder über den Namensdienst der Komponenten-Plattform.

Da *Client* und *Server* nicht auf demselben Rechner liegen müssen, ist im Allgemeinen eine Abbildung von Referenzen auf Speicheradressen nicht möglich.

Stubs und Skeletons

Komponenten-Modelle realisieren Referenzen auf entfernte Objekte über Stubs (*Stummel*) und Skeletons (*Skelette*).

Statt auf das entfernte Objekt verweist eine Referenz auf ein lokales *Stub-Objekt*, das die gleiche Schnittstelle wie das entfernte Objekt bietet.



Stubs und Skeletons (2)

Ruft der Client eine Operation auf, *verpackt* das Stub-Objekt die Daten des Aufrufs und sendet sie an den Rechner, auf dem das Skeleton-Objekt liegt.

Dort werden die Daten empfangen und *entpackt*; Ausgabe-Parameter werden umgekehrt genauso übertragen.

Das Verpacken und Entpacken kostet natürlich Zeit; dies ist jedoch vernachlässigbar gegenüber den Transportkosten.

Vorteile

- Client und Server haben beide den Eindruck, mit lokalen Partnern zu kommunizieren
- Stubs und Skeletons können automatisch generiert werden
- Das Laufzeitsystem kann alle Aufrufe abfangen und überprüfen (z.B. zur Authentifizierung)

Standards für Komponenten

Komponenten benötigen *Standards* zum Zusammenarbeiten:

Diese Standards sichern folgende Eigenschaften:

Interoperabilität – Technische Plattform zur Kommunikation

Sicherheit – Benutzerverwaltung und Authentifizierung

Transaktionen – zur kontrollierten Veränderung eines Datenbestandes gemäß dem ACID-Prinzip (*atomic*, *consistent*, *isolated*, *durable*)

Ortstransparenz – mit Namensdienst

Standards für Komponenten (2)

Wir betrachten die wichtigsten Standards für komponentenbasierte Systeme:

- CORBA
- J2EE
- Sun ONE
- .NET

CORBA

CORBA (von *Common Object Request Broker Architecture*) ist der älteste Standard für komponentenbasierte Entwicklung.

Entwickelt von der *Object Management Group*, 1991

CORBA ist eine Spezifikation, kein Produkt!

Object Management Architecture

Anwendungsobjekte

Client-Objekte

Server-Objekte

Object Request Broker (ORB)

Spezielle Objektdienste:
- Namensgebung
- Objekte kopieren
- Objekte löschen...

Allgemeine Objektdienste:
- Hilfedienste
- Druckdienste
- Sicherheitsdienste...

Object Management Architecture (2)

Hauptkomponenten der Object Management Architecture (OMA):

Anwendungsobjekte können Clients (Dienstnutzer) als auch Server (Dienstanbieter) sein

Object Request Broker (ORB) vermittelt zwischen Objekten

- übermittelt Operationsaufrufe und Ergebnisse
- vergleichbar mit einer Telefonzentrale

Spezielle Objektdienste werden vom ORB zur Erfüllung seiner Aufgaben benötigt

Allgemeine Objektdienste stehen jeder Komponente zur Verfügung

Die Schnittstellen-Beschreibung

Der ORB benutzt die *Schnittstellen* von Client- und Server-Objekten, um

- Anforderungen von Client-Objekten an Server-Objekte weiterzuleiten und
- die Ergebnisse zurückzuliefern.

Die Schnittstelle wird mit Hilfe der *Interface Description Language* (IDL) beschrieben.

Die Schnittstellen-Beschreibung (2)

Schnittstelle
(geschrieben in IDL)

Server
code

IDL
stub

interface
repository

IDL
skeleton

implemen-
tation
repository

- benutzt vom Client -

- benutzt vom Server -

Die Schnittstellen-Beschreibung (3)

Aus einer IDL-Beschreibung werden automatisch generiert:

IDL stub – Funktionen, die der Client benutzt, um Dienste des Servers anzufordern

IDL skeleton – Code-Rahmen für den Server, der ausgeführt wird, wenn eine Anforderung eintrifft

Interface Repository – Speichert Informationen über die Schnittstelle

Implementation Repository – Verwaltet Server-Informationen, damit der ORB Server lokalisieren und starten kann

Der Object Request Broker

Client-Objekte

Server-Objekte

statisch

dynamisch

IDL stub

Dynamic Invocation Interface (DII)

ORB Interface

IDL skeleton

Object adapter

ORB core (Transportschicht)

Der Object Request Broker (2)

Clients interagieren mit dem ORB

- *statisch* über ihren *IDL stub oder*
- *dynamisch* über das *dynamic invocation interface*, die aus dem interface repository die Schnittstelle erfragt

Server interagieren mit dem ORB über den *object adapter*, der die Dienste im *IDL skeleton* aufruft. Die Schnittstelle zum object adapter heißt *POA* (*portable object adapter*).

Zusätzlich stehen Dienste des ORB über das *ORB interface* zur Verfügung.

Die Interface Description Language

Die *Interface Description Language* (IDL) beschreibt die *Schnittstelle* einer Komponente in *sprachunabhängiger Form.*

Es gibt *standardisierte Abbildungen* für C, C++, Smalltalk, COBOL, Ada, Java (sowie *nicht-standardisierte* für Eiffel, Modula-3, Perl, Tcl, Objective-C, Python . . .)

Die Interface Description Language

Die *Interface Description Language* (IDL) beschreibt die *Schnittstelle* einer Komponente in *sprachunabhängiger Form.*

Es gibt *standardisierte Abbildungen* für C, C++, Smalltalk, COBOL, Ada, Java (sowie *nicht-standardisierte* für Eiffel, Modula-3, Perl, Tcl, Objective-C, Python . . .)

Grundtypen: `short` · `long` · `float` · `double` · `char` · `boolean` · `string` · `octet` · `enum` · `any`

Strukturierte Typen: `structure` · `union` · `array` · `sequence` · `exception`

Die Interface Description Language (2)

`time.idl` beschreibt einen Dienst, um die aktuelle Uhrzeit abzufragen:

```
struct TimeOfDay {
    short hour;     // 0 - 23
    short minute;   // 0 - 59
    short second;   // 0 - 59
};

interface Time {
    TimeOfDay get_gmt();
}
```

Ein Zeit-Server

Wir übersetzen `time.idl` in C++-*stub*- und *skeleton*-Code:

```
$ idl time.idl
$ _
```

Wir erhalten

- `time.hh` – eine Header-Datei, die die Typen aus `time.idl` definiert
- `timeC.cc` – den Stub-Code
- `timeS.hh` und `timeS.cc` – den Skeleton-Code

Ein Zeit-Server (2)

So sieht die Implementierung von get_gmt im Server aus:

```
#include <iostream.h>
#include <time.h>
#include "timeS.hh"

class Time_impl: public virtual POA_Time {
public:
    virtual TimeOfDay get_gmt()
        throw(CORBA::SystemException) {
        struct tm *time_p = gmtime(time(0));

        TimeOfDay tod;
        tod.hour   = time_p->tm_hour;
        tod.minute = time_p->tm_min;
        tod.second = time_p->tm_sec;

        return tod;
    }
}
```

Ein Zeit-Server (3)

Unser Server erzeugt ein passendes Objekt und gibt dessen OID aus:

```
int main(int argc, char *argv[])
{
    // ORB starten
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Objekt erzeugen
    Time_impl time_servant;

    // OID ausgeben
    Time_var tm = time_servant._this();
    cout << orb->object_to_string(tm) << endl;

    // Und los...
    orb->run();
}
```

Ein Zeit-Client

Der Client nutzt die eingelesene OID, um auf die Zeit zuzugreifen:

```
#include <iostream.h>
#include "time.hh"
int main(int argc, char *argv[])
{
    // ORB starten
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

    // Referenz suchen
    CORBA::Object_var obj = orb->string_to_object(argv[1]);
    Time_var tm = Time::_narrow(obj);

    // Zeit holen
    TimeOfDay tod = tm>get_gmt();
    cout << "Zeit: " << tod.hour << ":" << tod:minute << endl;

    return 0;
}
```

Zeit-Server und Zeit-Client

Beispiel-Lauf:

```
$ ./server &
```

Zeit-Server und Zeit-Client

Beispiel-Lauf:

```
$ ./server &
IOR:000000009c169adc4795ec49e4f9a9a03719db2a152b7a4d8cd78794d532fd44a973
da423a1afcf620a84871c9bfe29ef48a319496bf027d44c5b9f286b79aebce34e2c1d07a
3a1ba104c5b3dac2553e625911fd993a0317e041aa49d1ef9bce569c83e0e84ccaf4f1b7
c8ed647bf8569e0d266796643206e59008c64157c37be6415a3444282096eef27787ab8f
0a438e55436622de0788a7826d98f6fba28d0e8ae5a34943ecc67dbfb44469f44a52cf28
334e910f25820a9390750d8e57bada22b75e93382490a99d55f1ea288b56bf459711e023
d45ae8a0d0d656c328fafe3cf7495185ca9cb85f47e4d3755a78026b573eed93b247a59d
2252c423dc0df08243793f3da3d5db6bdf15a6130e9591781bb6d7b4ba60731185ff062c
8d76420beeea0000000000000000000
$
```

Zeit-Server und Zeit-Client

Beispiel-Lauf:

```
$ ./server &
IOR:000000009c169adc4795ec49e4f9a9a03719db2a152b7a4d8cd78794d532fd44a973
da423a1afcf620a84871c9bfe29ef48a319496bf027d44c5b9f286b79aebce34e2c1d07a
3a1ba104c5b3dac2553e625911fd993a0317e041aa49d1ef9bce569c83e0e84ccaf4f1b7
c8ed647bf8569e0d266796643206e59008c64157c37be6415a3444282096eef27787ab8f
0a438e55436622de0788a7826d98f6fba28d0e8ae5a34943ecc67dbfb44469f44a52cf28
334e910f25820a9390750d8e57bada22b75e93382490a99d55f1ea288b56bf459711e023
d45ae8a0d0d656c328fafe3cf7495185ca9cb85f47e4d3755a78026b573eed93b247a59d
2252c423dc0df08243793f3da3d5db6bdf15a6130e9591781bb6d7b4ba60731185ff062c
8d76420beeea0000000000000000000
$ ./client IOR:000000000...
Zeit: 9:59
$ _
```

Zugriff erfolgt Netz-transparent!

Objektdienste

CORBA definiert außerdem *Dienste*, die viele Klassen häufig benötigen:

Namensdienst – Abbildung von Referenzen (OIDs) auf Namen

Lebenzyklusdienst – Verwaltung von Objekten

Ereignismeldedienst – Benachrichtigen über Ereignisse

Persistenzdients – Dauerhaftes Speichern

Nebenläufiger Dienst – Synchronisierung konkurrierender Zugriffe

Transaktionsdienst – mit zweistufigem *commit*

Sicherheitsdienst – Autorisierungsfunktionen . . .

Alle Dienste sind in IDL spezifiziert.

Komponenten in CORBA

CORBA 2.0 stellt nur eine *Architektur* zur Verfügung, aber noch keine Komponenten:

- Keine Aussagen über Schnittstellen
- Keine Aussagen über Verteilung

Komponenten sind erst seit CORBA 3.0 definiert; sie richten sich im Wesentlichen nach Suns *Enterprise Java Beans (EJBs).*

Hintergrund 1: Vorhandene EJBs sollten als CORBA-Komponenten benutzt werden können.

Komponenten in CORBA

CORBA 2.0 stellt nur eine *Architektur* zur Verfügung, aber noch keine Komponenten:

- Keine Aussagen über Schnittstellen
- Keine Aussagen über Verteilung

Komponenten sind erst seit CORBA 3.0 definiert; sie richten sich im Wesentlichen nach Suns *Enterprise Java Beans (EJBs).*

Hintergrund 1: Vorhandene EJBs sollten als CORBA-Komponenten benutzt werden können.

Hintergrund 2: CORBA war kein großer kommerzieller Erfolg

J2EE

Die *Java 2 Platform Enterprise Edition* (J2EE) von Sun umfaßt Java-basierte Techniken für Unternehmenslösungen.

Hauptbestandteil der J2EE sind die *Enterprise Java Beans* (EJBs). Enterprise Java Beans haben (außer dem Namen) nichts mit Java Beans gemeinsam.

Auch EJBs sind eine Spezifikation, kein konkretes Produkt!

Aufbau einer EJB

EJB-Komponente (JAR-Datei)

remote interface

«interface»
Kunde

+operation_1()
+operation_2()

home interface

«interface»
KundeHome

+create()
+findByPrimaryKey()

bean class

KundeBean

+einAttribut:

+operation_1()
+operation_2()
+create()

deployment descriptor

```
<enterprise-beans>
...
</enterprise-beans>
<assembly-description>
...
```

Aufbau einer EJB (2)

Eine EJB besteht aus vier Teilen:

Aufruf-Schnittstelle (*remote interface*) beschreibt die Dienstleistungen der EJB als `interface`.

Verwaltungs-Schnittstelle (*home interface*) läßt EJBs erzeugen und im Netz finden

Bean-Klasse (*bean class*) implementiert die Operationen

Auslieferungs-Beschreibung (*deployment descriptor*) beschreibt die Schnittstelle im XML-Format

Aufruf- und Verwaltungssschnittstelle sind praktisch Skelett-Klassen.

Aufbau einer EJB (3)

Der Client kennt nur Aufruf- und Verwaltungsschnittstelle:

Client

RMI
CORBA-IIOP

home interface

remote interface

«interface»
EJBHome

«interface»
EJBObject

Bean-Klasse

EJB-Container

Aufruf-Schnittstelle

Wir betrachten die Realisierung eines Systems zur Seminarorganisation.

Die *Aufruf-Schnittstelle* beschreibt die Dienstleistungen:

```
package SemOrg.Schnittstellen;
import java.rmi.*;
import javax.ejb.*;

public interface Buchung extends EJBObject
{
    public void buchen(Kunde k, Seminartyp s)
        throws RemoteException;
}
```

Verwaltungs-Schnittstelle

Die *Verwaltungs-Schnittstelle* ermöglicht das Erzeugen der Objekte:

```
package SemOrg.Schnittstellen;
import java.rmi.*;
import javax.ejb.*;

public interface BuchungHome extends EJBHome
{
    public Buchung create()
        throws RemoteException, CreateException;
}
```

Bean-Klasse

```
package SemOrg.Server;
import java.rmi.*;
import javax.ejb.*;
import javax.naming.*;

public class BuchungBean implements SessionBean {
    public void ejbCreate() throws RemoteException {
        // Initialisierung des Objekts
    }
    public void ejbRemove() {
        // Objekt wird zerstört -- Aufräumarbeiten
    }
    public void setSessionContext(SessionContext ctx) {}
    public void ejbActivate() {} // Brauchen wir hier nicht
    public void ejbPassivate() {}
    public void buchen(Kunde k, Seminartyp s)
        throws RemoteException
    {
        // alle Schritte zur Durchführung einer Buchung
    }
}
```

Auslieferungs-Beschreibung

im XML-Format – ersetzt IDL in CORBA

```
<enterprise-beans>
  <session>
    <ejb-name>SemOrg/Buchung</ejb-name>
    <home>SemOrg.Schnittstellen.BuchungHome</home>
    <remote>SemOrg.Schnittstellen.Buchung</remote>
    <ejb-class>SemOrg.Server.BuchungBean</ejb-class>
    <session-type>Stateless</session-type>
    <transaction-type>Container</transaction-type>
  </session>
</enterprise-beans>
<assembly-description>
  <container-transaction>
    <method>
      <ejb-name>Buchung</ejb-name>
      <method-name>*</method-name>
    </method>
  </container-transaction>
</assembly-description>
```

Ein einfacher Client

Hier via *remote method invocation* (RMI) statt CORBA realisiert:

```
package SemOrg.Client;
import java.rmi.RemoteException;
import javax.naming.context;
import javax.naming.InitialContext;
import SemOrg.Schnittstellen.*;

public class Client {
    public static void main(String[] args) {
        Client einClient = new Client();
        einClient.start();
    }
    ...
}
```

Ein einfacher Client (2)

```
...
public class Client {
    public static void main(String[] args) ...
    public void start() {
        Kunde einKunde; Seminar einSeminar;
        Context ctx = new InitialContext();
        Object temp =
            ctx.lookup("java:comp/env/Buchung");
        BuchungHome home = (BuchungHome)
            javax.rmi.PortableRemoteObject.
            narrow(temp, BuchungHome.class);
        Buchung bean = home.create();
        bean.buchen(einKunde, einSeminar);
        bean.remove();
    }
}
```

Sun ONE

ONE = *Open Network Environment*, Nachfolger von J2EE

Realisiert *Web Services:* komplette Infrastruktur für moderne, verteilte Software-Systeme

Ziel: Aus Webseiten sollen programmierbare *Dienste* werden

Basiert auf

- HTTP als Transportprotokoll
- *Simple Object Access Protocol* (SOAP) und XML zum Aufruf und Verpacken der Daten
- *Web Services Description Language* (WSDL) zur Selbstbeschreibung

SOAP und WSDL werden derzeit standardisiert

.NET

.NET von Microsoft realisiert ebenfalls *Web-Services*

Grundlage: *Microsoft Intermediate Language* (MSIL) statt Java-Bytecode; zahlreiche Sprachen (insbesondere C#) können in MSIL übersetzt werden

.NET Assemblies entsprechen EJBs (einschließlich Selbstbeschreibung) und lösen die berüchtigten DLLs ab

Im Wesentlichen ähnliche Konzepte wie Sun ONE

Komponentenmodelle im Vergleich

Kriterium	SunONE/J2EE	Corba Components	.NET
Plattform-unabhängigkeit	ja	ja	im Wesentlichen *Windows*

Komponentenmodelle im Vergleich

Kriterium	**SunONE/J2EE**	**Corba Components**	**.NET**
Plattform-unabhängigkeit	ja	ja	im Wesentlichen *Windows*
Sprach-unabhängigkeit	Java	ja	ja (vorwiegend C#)

Komponentenmodelle im Vergleich

Kriterium	SunONE/J2EE	Corba Components	.NET
Plattform-unabhängigkeit	ja	ja	im Wesentlichen *Windows*
Sprach-unabhängigkeit	Java	ja	ja (vorwiegend C#)
Hersteller-unabhängigkeit	ja (Spezifikation in der Hand von Sun)	ja	Hersteller ist Microsoft

Konzepte

- *Komponenten* ermöglichen freie Konfigurierbarkeit und dynamische, verteilte Systeme
- Komponenten verfügen über wohldefinierte *Schnittstellen*
- Die Kommunikation zwischen Objekten geschieht über *Referenzen*, *Stubs* und *Skeletons*
- *CORBA* ist der älteste Standard für komponentenbasierte Systeme
- Aus der *CORBA Interface Description Language* werden automatisch *Stubs* und *Skeletons* erzeugt

Konzepte (2)

- *Enterprise Java Beans* aus Suns J2EE sind Komponenten, die CORBA und RMI zur Kommunikation nutzen
- *Sun ONE* und *.NET* realisieren Web Services, die aus Webseiten Komponentendienste machen

Literatur

Balzert, Lehrbuch der Softwaretechnik – Bd. 1 (2. Auflage), LE 29 „Verteilte objektorientierte Anwendungen“

www.google.de – Suche nach „J2EE“, „Sun ONE“ und „.NET“