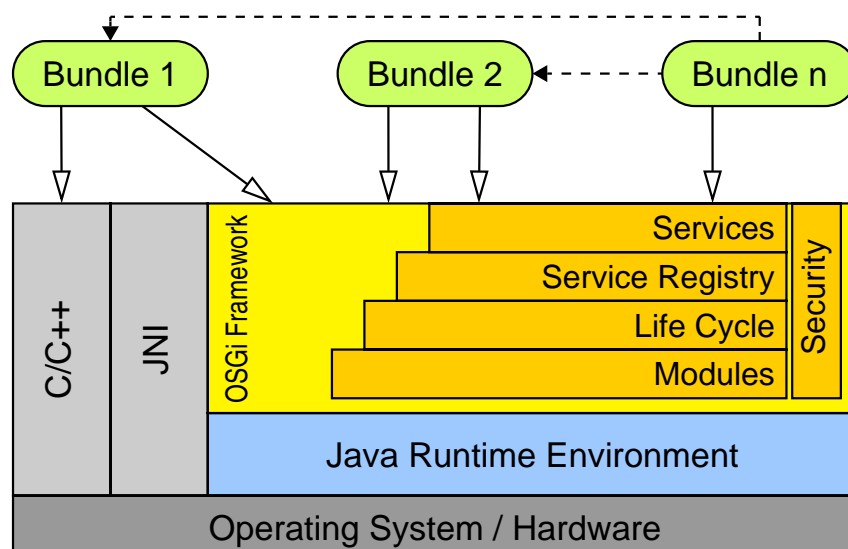


## J.1 OSGi

- OSGi (Open Services Gateway initiative)
  - ◆ Standardisierte, komponentenorientierte Softwareplattform
  - ◆ Entspricht dem SOA-Paradigma (Service Oriented Architecture)
  - ◆ OSGi Service Platform setzt auf der Java Virtual Machine auf
- Spezifikation
  - ◆ OSGi Service Platform Release 1 (2000), aktuell Release 4.1 (2007)
  - ◆ Beschreibt lediglich API und Testcases
- OSGi-Frameworks
  - ◆ Equinox Framework von Eclipse (<http://www.eclipse.org/equinox/>)
  - ◆ Apache Felix (ehemals Oscar) (<http://felix.apache.org/>)
  - ◆ Knopflerfish (Open Source / Gatespace) (<http://www.knopflerfish.org/>)
  - ◆ Diverse kommerzielle Produkte

## J.2 Framework

- Das OSGi Framework bietet eine universelle, sichere und zentral verwaltete Ausführungsumgebung für modulare Anwendungen (Bundles)
- Framework-Aufbau



# 1 Framework

## ■ Security Layer

- ◆ Bundles können aus verschiedenen Quellen stammen
- ◆ Infrastruktur für eine kontrollierte Ausführung (*controlled environment*), basierend auf der Java 2 Security Architecture. [1]
- ◆ Code Authentifizierung durch Signaturen (Herkunft, Unterzeichner)
- ◆ Security Layer ist optional und oft nicht implementiert

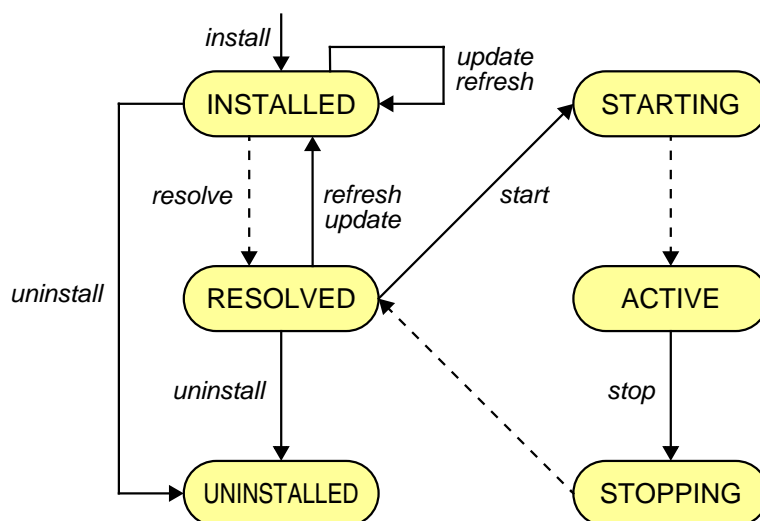
## ■ Module Layer

- ◆ Einheit der Modularisierung: *Bundle*
- ◆ Bundles werden als Java ARchive (JAR) Datei ausgeliefert
- ◆ Sie enthalten:
  - Alle für den Dienst notwendigen *Ressourcen* (inkl. weiterer JARs)
  - Eigenen Class-Path und Loader (Policy)
  - Bundlebeschreibung und Abhängigkeiten (z.B. Java Packages) in *Manifest Datei* (META-INF/MANIFEST.MF)

# 1 Framework

## ■ Life Cycle Layer

- ◆ Verwaltung des Lebenszyklus von Bundles
- ◆ Bundles können *installiert*, *gestartet*, *gestoppt* und *deinstalliert* werden
- ◆ Zusätzlich können Bundles zur Laufzeit *aktualisiert* und *überwacht* werden



# 1 Framework

## ■ Life Cycle Layer - Entitäten:

- ◆ *Bundle* - Repräsentiert ein installiertes Bundle
- ◆ *Bundle Context* - Ausführungskontext eines Bundles
  - Zugriff auf Informationen des Framework und die Service Registry
  - Installieren anderer Bundles
  - Wird beim starten/stoppen an *Bundle Activator* weitergereicht
- ◆ *Bundle Activator* - Interface implementiert durch eine Klasse des Bundles
  - Starten und stoppen des Bundles
- ◆ *Bundle Event* - Signalisiert Lebenszyklus Zustandsänderungen
- ◆ *Framework Event* - Signalisiert Framework Zustandsänderungen
- ◆ *Bundle/Framework Listener* - Listener für Events
- ◆ *Bundle Exception* - Ausnahme für Framework-Operationen
- ◆ *System Bundle* - Bundle-Repräsentation des Frameworks

# 1 Framework

## ■ Service Layer

- ◆ *Service Registry* zum registrieren, finden und binden von Diensten
- ◆ Dienste sind Java-Objekte (*Service Object*), registriert mit ihrem/ihren Interface(s) (*Service Interface*)
- ◆ Bundles können Dienste registrieren, suchen und ihren Zustand abfragen
- ◆ Mit dem stoppen des Bundles werden alle registrierten Dienste entfernt

## ■ Service Layer - Entitäten:

- ◆ *Service* - Registrierter Dienst in der Service Registry, das *Service Object* gehört zu und läuft in einem Bundle
- ◆ *Service Registry* - Verwaltet die Dienste
- ◆ *Service Reference* - Referenz auf einen Dienst
  - Zugriff auf Diensteigenschaften (auf *Service Object* über *Bundle Context*)
- ◆ *Service Registration* - Rückgabe der Registrierung, erlaubt aktualisieren und entfernen des Dienstes

# 1 Framework

- Service Layer - Entitäten (Forts.):
  - ◆ *Service Factory* - Ermöglicht es dem Anbieter, das *Service Object* auf den Nutzer anzupassen
  - ◆ *Service Event* - Informationen über Registrierung, Änderung und Deregistrierung von *Service Objects*
  - ◆ *Service Listener* - Listener für *Service Events*
  - ◆ *Filter* - Objekt für die Auswahl über Diensteigenschaften

# 2 Felix OSGi Framework

- Apache Top-Level-Projekt (<http://felix.apache.org>)
  - ◆ Aktuelle Release: Felix 1.0.1
  - ◆ Installation: Download und entpacken
- Felix starten und testen:

```
ulbrich@fau48a:/local/felix$ java -jar bin/felix.jar

Welcome to Felix.
=====

Enter profile name: integration_test

DEBUG: WIRE: 31.0 -> org.osgi.service.packageadmin -> 0
...
DEBUG: WIRE: 33.0 -> org.apache.felix.shell -> 31.0

->
```

## 2 Felix OSGi Framework

### ■ Felix Shell:

```
-> help
bundlelevel <level> <id> ... | <id> - set or get bundle start level
cd [<base-URL>]                - change or display base URL
headers [<id> ...]             - display bundle header properties
help                           - display impl commands
install <URL> [<URL> ...]      - install bundle(s)
obr help                       - OSGi bundle repository
packages [<id> ...]           - list exported packages
ps [-l | -s | -u]              - list installed bundles
refresh [<id> ...]            - refresh packages
resolve [<id> ...]            - attempt to resolve specified bundle
services [-u] [-a] [<id> ...] - list registered or used services
shutdown                      - shutdown framework
start <id> [<id> <URL> ...]    - start bundle(s)
startlevel [<level>]          - get or set framework start level
stop <id> [<id> ...]           - stop bundle(s)
uninstall <id> [<id> ...]      - uninstall bundle(s)
update <id> [<URL>]            - update bundle
version                       - display version of framework
->
```

## 2 Felix OSGi Framework

### ■ Konfigurationsdateien von Felix

- ◆ **conf/system.properties** und **conf/config.properties**
- ◆ Bundles können über **config.properties** konfiguriert werden
- ◆ Zugriff über **BundleContext.getProperty()**
- ◆ Alternative Standorte:

```
java -Dfelix.system.properties=file:/local/felix/conf/system.properties
bzw.
java -Dfelix.config.properties=file:/local/felix/conf/config.properties
```

### ■ Einige Parameter:

- ◆ **felix.auto.install** - Bundles die automatisch installiert werden sollen
- ◆ **felix.auto.start** - Bundles die automatisch gestartet werden sollen
- ◆ **felix.cache.dir** - Verzeichnis des Bundle-Cache (default **~/.felix/**)
- ◆ **felix.cache.profile** - Name des Profiles (innerhalb des Bundle-Cache)

### ■ Integration in Eclipse siehe: <http://felix.apache.org/>

## 2 Felix OSGi Framework

- Life Cycle - Manuelles installieren und starten eines Bundles:

```
-> install file:bundle/simple.jar
Bundle ID: 10
-> ps
START LEVEL 1
  ID    State      Level  Name
[  0] [Active      ] [  0] System Bundle (1.1.0.SNAPSHOT)
...
[ 10] [Installed   ] [  1] Simple Bundle (1.0.0)
-> start 10
Simple bundle 10 has started.
From native: Hello!
->
```

- Stoppen, aktualisieren und deinstallieren von Bundles analog
  - ◆ Achtung: Bei Abhängigkeiten evtl. ein `refresh` notwendig

## J.3 Manifest

- Beschreibt das Bundle und seine Abhängigkeiten  
Notwendige Parameter:
  - ◆ **Manifest-Version** - Manifest Spezifikation (1.0)
  - ◆ **Bundle-Name** - Name des Bundles
  - ◆ **Import-Package** - Von anderen Bundles importierte Pakete (zwingend notwendig ist `org.osgi.framework`)
- Wichtige Parameter:
  - ◆ **Bundle-Activator** - Klasse die das Interface `BundleActivator` implementiert, andernfalls dient das Bundle nur als Bibliothek
  - ◆ **Bundle-SymbolicName** - Eindeutiger Name (meist in Domain-Form)
  - ◆ **Export-Package** - Pakete die von diesem Bundle exportiert werden
  - ◆ **Bundle-Version** - Version
  - ◆ **Bundle-Classpath** - Intra-Bundle Classpath für eingebettete JARs
  - ◆ **Require-Bundle** - Importiert alle Pakete der angegebenen Bundles

# 1 Manifest

- Activator Klasse ist "main" des Bundles und wird beim starten ausgeführt
- Jedes Bundle hat eigenen Classloader und *Class Space*
  - ◆ Classpath ergibt sich aus: RTE + Framework + Bundle Classpath

- Beispiel (META-INF/manifest.mf)

```
Manifest-Version: 1.0
Bundle-Name: simplebundle
Bundle-SymbolicName: gruppe0.osgi.simplebunde
Bundle-Version: 1.0.0
Bunde-Description: Demo Bundle
Bunde-Activator: gruppe0.osgi.simplebundle.impl.Activator
Import-Package: org.osgi.framework
```

- Auf korrekte Formatierung achten!
  - ◆ Bezeichnen : Wert / Listen kommasepariert / Umbruch mit Leerzeichen einrücken / Zeilenumbruch am Ende der Datei

# 2 Auflösen der Abhängigkeiten

- Auflösen der Abhängigkeiten (*Resolving*) zwischen Bundles
  - ◆ Verdrahten von Importeuren und Expoteuren anhand von Bedingungen
- Die Bedingungen werden statisch definiert, durch:
  - ◆ Importierte und exportierte Pakete
  - ◆ Benötigte Bundles (import aller Pakete)
  - ◆ Fragmente (Teil eines größeren logischen Bundles)
- Ein Bundle kann aufgelöst werden wenn
  - ◆ Alle Importe sind verdrahtet
  - ◆ Alle benötigten Pakete sind verfügbar und ihre Exporte verdrahtet
- Nach dem Auflösen kann Bundle geladen und ausgeführt werden

## J.4 Build file

- Bauen eines OSGi Bundles
  - ◆ Übersetzen der Quellen
  - ◆ Packen aller Ressourcen und des Manifestes zu einem JAR

- Beispiel für ein Ant `build.xml`:

```
<?xml version="1.0"?>
<project name="simplebundle" default="all">
  <target name="all" depends="compile,jar"/>
  <target name="compile">
    <javac destdir = "./classes" srcdir = "./src"></javac>
  </target>
  <target name="jar">
    <jar basedir = "./classes"
      jarfile = "./build/simplebundle.jar"
      includes = "**/*"
      manifest = "./meta-inf/MANIFEST.MF"
    />
  </target>
</project>
```

## J.5 Dienste - Aktivator

- Die Aktivator Klasse implementiert `BundleActivator` Interface
- Beispiel:

```
public class Activator implements BundleActivator {
  public static BundleContext bc = null;
  public void start(BundleContext bc) throws Exception {
    System.out.println("stopping...");
    Activator.bc = bc;
  }
  public void stop(BundleContext bc) throws Exception {
    System.out.println("stopping...");
    Activator.bc = null;
  }
}
```

- Methoden `start()` und `stop()` müssen implementiert werden
  - ◆ `BundleContext` sollte mit starten gesetzt und mit stoppen gelöscht werden
  - ◆ In `start()` wird das Bundle hochgefahren (z.B. Threads gestartet)
  - ◆ In `stop()` muss alles wieder aufgeräumt werden

# 1 Dienste anbieten

- Erweiterung des Bundles um einen Dienst
  - ◆ Bundle erhält ein Interface (*Service Interface*) für den Dienst
  - ◆ Das Manifest wird erweitert

- Manifest Erweiterung:

```
Export-Package: gruppe0.osgi.simpleservice; version="1.0.0"
```

- Dienst Interface:

```
package gruppe0.osgi.simpleservice;
import java.util.Date;

public interface SimpleService {
    public String getFormattedDate(Date date);
}
```

- Dienst Implementierung (*Service Implementation*) analog

# 1 Dienste anbieten

- Aktivator registriert den Dienst
  - ◆ `BundleContext.registerService( SInt, SImpl, Param<K:V> ) ;`
  - ◆ `ServiceRegistration` zur Verwaltung der Registrierung

- Beispiel:

```
public class Aktivator implements BundleActivator {
    public static BundleContext bc = null;
    public void start(BundleContext bc) throws Exception {
        System.out.println("starting...");
        Aktivator.bc = bc;
        SimpleService service = new SimpleServiceImpl();
        ServiceRegistration registration =bc.registerService(
            SimpleService.class.getName(),service, new Hashtable());
    }
    public void stop(BundleContext bc) throws Exception {
        System.out.println("stopping...");
        Aktivator.bc = null;
    }
}
```

## 2 Dienste nutzen

### ■ Manifest für die Nutzung eines Dienstes:

```
Manifest-Version: 1.0
Bundle-Name: serviceuser
Bundle-SymbolicName: gruppe0.osgi.serviceuser
Bundle-Version: 1.0.0
Bunde-Description: Demo Bundle
Bunde-Activator: gruppe0.osgi.serviceuser.impl.Activator
Import-Package: org.osgi.framework, gruppe0.osgi.simpleservice
```

### ■ Anforderung eines Dienstes

- ◆ `BundleContext.getServiceReference( ServiceInterface );`
- ◆ `BundleContext.getService( ServiceReference );`
- ◆ Framework erlaubt dynamisches installieren, aktualisieren und löschen
- ◆ `ServiceReference` muss daher immer überprüft werden
- ◆ Die `ServiceReference` nach Verwendung schnell wieder freigeben

## 2 Dienste nutzen

### ■ Beispiel:

```
import gruppe0.osgi.SimpleService;
public class Activator implements BundleActivator {
    public static BundleContext bc = null;
    public void start(BundleContext bc) throws Exception {
        Activator.bc = bc;
        ServiceReference reference = bc.getServiceReference
            (SimpleService.class.getName());
        if (reference != null) {
            SimpleService service =
                (SimpleService)bc.getService(reference);
            System.out.println(
                service.getFormattedDate(new Date()));
            bc.ungetService(reference);
        } else {
            System.out.println("Service unavailable!!");
        }
    }
    public void stop(BundleContext bc) throws Exception {
        Activator.bc = null;
    }
}
```

## 2 Dienste nutzen

### ■ Dynamisches binden von Diensten

- ◆ Problem mit erstem Ansatz falls Dienst beim Start nicht verfügbar
- ◆ Lösung: Regelmäßig prüfen oder `ServiceListener`

### ■ `ServiceEvent` und Filter nutzen

- ◆ Ein Filter ist ein LDAP ähnlicher String über den Events gefiltert werden
- ◆ Um den Filter erweiterte `start()` Methode des Aktivators:

```
public void start(BundleContext bc) throws Exception {
    Activator.bc = bc;
    String filter = "(objectclass="+SimpleService.class.getName()+")";
    bc.addServiceListener(this, filter);
    ServiceReference references[] = bc.getServiceReferences
        (null, filter);
    for (int i = 0; references != null && i < references.length; i++)
    {
        this.serviceChanged(
            new ServiceEvent(ServiceEvent.REGISTERED, references[i]));
    }
}
```

## 2 Dienste nutzen

### ■ Der Aktivator erhält das `ServiceListener` Interface

- ◆ Zusätzliche Methode `serviceChanged(ServiceEvent event)`
- ◆ Hilfsmethoden (z.B. `stopUsingService()/startUsingService()`)

### ■ Beispiel:

```
public void serviceChanged(ServiceEvent event) {
    switch (event.getType()) {
        case ServiceEvent.REGISTERED:
            this.service = (SimpleService) Activator.bc.getService(
                event.getServiceReference());
            this.startUsingService();
            break;
        case ServiceEvent.MODIFIED:
            this.stopUsingService();
            this.service = (SimpleService) Activator.bc.getService(
                event.getServiceReference());
            this.startUsingService();
            break;
        case ServiceEvent.UNREGISTERING:
            this.stopUsingService();
            break;
    }
}
```

## 2 Dienste nutzen

- Dienste können mit Eigenschaften registriert werden
  - ◆ Zusätzliche Beschreibung des Dienstes
  - ◆ Ermöglicht "best fit" Auswahl bei mehreren Dienst Anbietern

- Beispiel - Dienstanbieter:

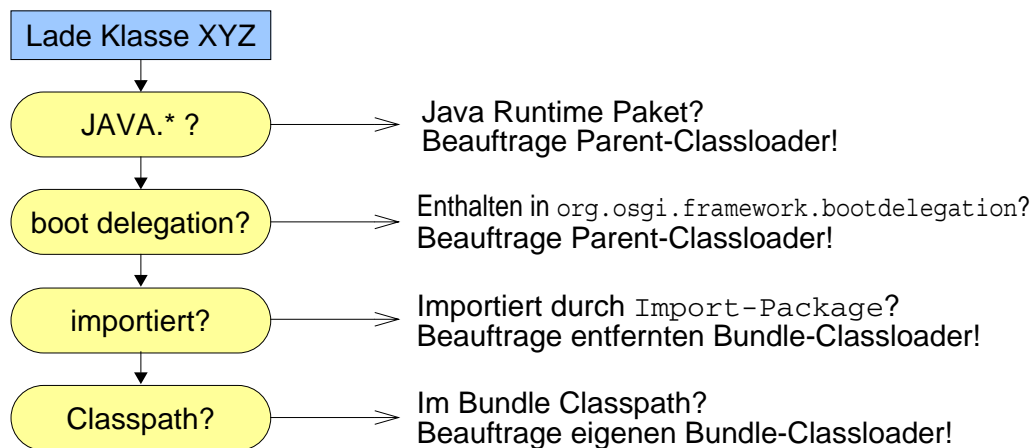
```
Hashtable properties = new Hashtable();
properties.put("color", Boolean.FALSE);
properties.put("A3paper", Boolean.TRUE);
bundleContext.registerService(
    "inf4.LaserJet", new Laserjet(), properties);
```

- Beispiel - Dienstnutzer:

```
ServiceReference[] srefs =
    bundleContext.getServiceReferences(
        "inf4.LaserJet", "(A3paper=true), (color=false)");
```

## J.6 Classloading

- Jedes Bundle besitzt einen eigenen *ClassLoader*
  - ◆ Dadurch verfügt jedes Bundle über eigenen *Namespace*
  - ◆ **Bundle-ClassPath** (Manifest) beschreibt intra Bundle Classpath
- Suchreihenfolge



# 1 Classloading

- Kann zu nicht offensichtlichen Effekten führen
  - ◆ System-Classloader kann Klasse laden, wird aber ggf. wegen Regelwerk nicht "weitergereicht".
  - ◆ Framework Konfiguration bestimmt welche Klassen exportiert werden
- Trickreiches Beispiel:
  - ◆ Klasse XYZ wird durch zwei Classloader C1 und C2 geladen
  - ◆  $XYZ_{C1}$  ist nicht gleich  $XYZ_{C2}$  !!
  - ◆ Kein Cast möglich: Unterschiedliche Typen!
- Classloader halten Referenzen
  - ◆ Auf alle geladenen Klassen (Selbst nach `uninstall`)
  - ◆ Framework `refresh`

## J.7 Stale reference

- In OSGi kann der Besitzer eines Objektes "verschwinden"
- "Abgestandene" Referenzen: Referenz auf ein Objekt
  - ◆ wobei dessen Classloader bzw. Bundle gestoppt wurde
  - ◆ dessen Dienst unregistriert wurde
- Beispiel:
  - ◆ Bundle **A** nutzt Service von Bundle **B**
  - ◆ **A** hält eine Referenz - erhalten z.B. über den Dienst - auf Objekt **o** der Klasse **K**, geladen durch **B**'s Classloader
  - ◆ Wird Bundle **B** gestoppt ist Referenz auf **o** *Stale Reference*
- Problematik:
  - ◆ Klassen eines gestoppten Bundles können nicht vom Garbage Collector eingesammelt werden
  - ◆ Problemen beim aktualisieren

## J.8 Hinweise zu Aufgabe 7

- MW-Library Server aus Aufgabe 4 als OSGi Bundle
- Modularisierung von Server und Datenbank
  - ◆ Zwei Bundles: `server` und `database`
- Vorgaben
  - ◆ Felix OSGi Framework: `/local/felix`
  - ◆ Konfiguration: `/proj/i4mw/pub/Aufgabe7`
- Konfiguration von Felix
- Classloading der Standard Java Klassen

## J.8 Literatur

- [1]: Java Security Architecture  
<http://java.sun.com/j2se/1.5.0/docs/guide/security/spec/security-spec.doc1.html>