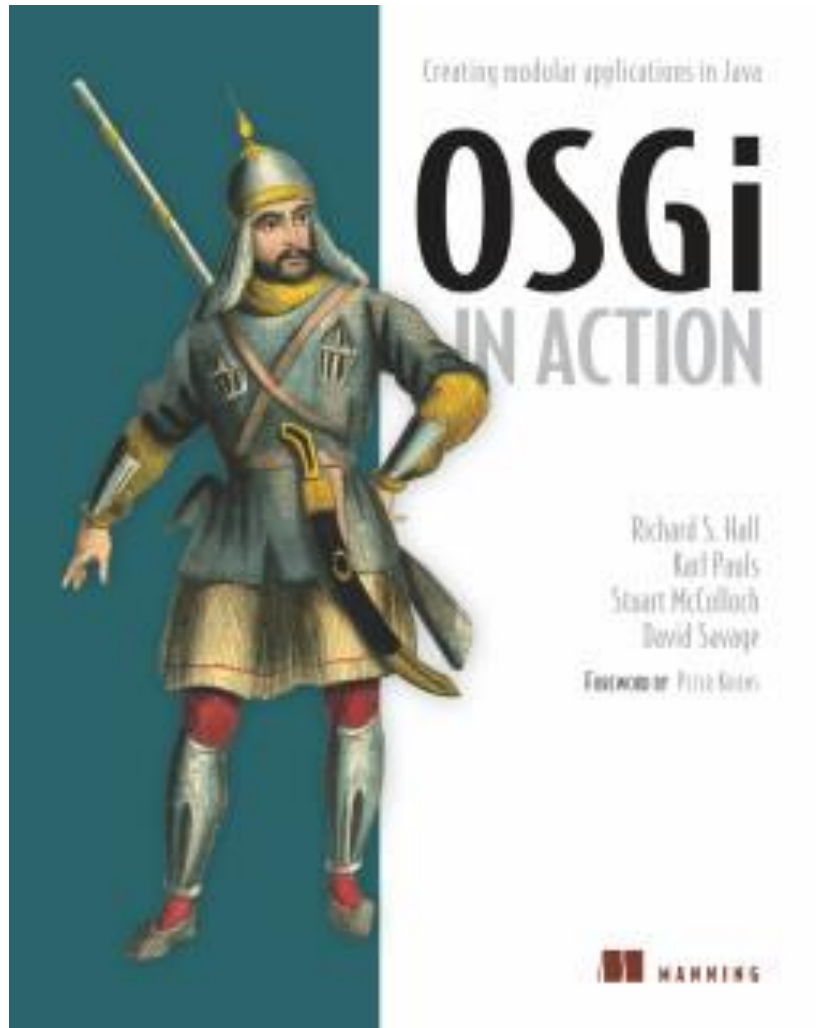


# OSGi

Eine Service-Plattform für Java



Apache Karaf is a small OSGi based runtime where applications can be deployed.

Here is a short list of features supported by the

- **Hot deployment:** Karaf supports hot code reloading. Each time a jar is copied in the repository, and changes will be handled automatically (blueprint and spring ones are included).
- **Dynamic configuration:** Services are configured on the fly. Configuration can be defined in Karaf using the configuration language and changes on the properties are monitored.
- **Logging System:** using a centralized logger (JDK 1.4, JCL, SLF4J, Avalon, Tomcat, Commons Logging).
- **Dependency Injection:** Dependency Injection of libraries.

**OVERVIEW**

- Home
- Getting Started
- FAQ
- Download

**RELEASES**

- Schedule
- 4.0.x
  - Overview
  - Quickstart
  - Update notes
  - Dependencies Matrix
  - Manual

- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
  - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
  - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
  - Musterbeispiele werden zur Verfügung gestellt
  - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
    - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung
  - Auch ein Git-Repository ist eingerichtet
    - <https://GitHub.com/JavacreamTraining/org.javacream.training.osgi>
- Konventionen
  - Befehle werden in `Courier-Schriftart` dargestellt
  - Dateinamen werden in *kursiver Courier-Schriftart* dargestellt
  - Links werden in unterstrichener Courier-Schriftart dargestellt

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

**Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks,  
der fotomechanischen und elektronischen Wiedergabe vorbehalten.**

Übersicht	6
Programmierung	24
Weiterführende Themen	71

# 1 ÜBERSICHT

1.1

## **WAS IST OSGI?**

- *„Die OSGi Alliance (früher Open Services Gateway initiative) spezifiziert eine hardwareunabhängige dynamische Softwareplattform, die es erleichtert, Anwendungen und ihre Dienste per Komponentenmodell („Bundle“/„Service“) zu modularisieren und zu verwalten („Service Registry“).“*
  - Der ursprüngliche Begriff der Open Services Gateway initiative ist nicht mehr gebräuchlich
- Die OSGi Alliance besteht aus einem Konsortium von mehr als 30 Firmen
  - u. a. Adobe Systems, Deutsche Telekom, Hitachi, IBM, Liferay, Oracle, Siemens, Software AG, TIBCO Software
- OSGi ist eine Spezifikation
  - Kein Produkt
  - OSGi-Provider stellen die Plattform zur Verfügung



- OSGi erweitert die Programmiersprache Java um ein sinnvolles Modul-Konzept
  - Export und Import von Packages als neue Form der Kapselung
  - Berücksichtigung einer Versions-Nummerierung
  - Einfaches API
- OSGi-Bundles sind spezielle Java-Archive
  - Die Manifest-Datei enthält zusätzliche OSGi-typische Informationen
- Die OSGi-Runtime erweitert die Java Virtual Machine
  - Dynamisches Installieren und Deinstallieren von Bibliotheken zur Laufzeit
  - Prüfen der Requirements einer Bibliothek vor Start durch ausgefeilten Resolve-Mechanismus
  - Bibliotheken können in verschiedenen Versionen parallel betrieben werden

- Betrieb einer modular konzipierten Anwendung
  - Hier werden die Anwendung in vielen kleinen, unabhängigen Module realisiert
    - Stichwort „Microservices“!
    - „OSGi als lokale SOA-Plattform“
  - Die einzelnen Module sind OSGi-Bundles
- Bereitstellung von Services parallel in verschiedenen Versionsständen
  - Beispiel Datenbanktreiber
  - Auch für Mandanten-fähige Systeme interessant
  - Weiche Releases mit Unterstützung der Vorgängerversion(en)
- Dynamische Anwendung
  - Durch Installation eines Bundles werden für andere Anwendungen neue oder zusätzliche Services freigeschaltet
    - Bei Bedarf mit dynamischer Aktualisierung
  - Continuous Delivery
  - Auch für Patches/Bug Fixes interessant

- Apache Group
  - Felix
  - Karaf
- Eclipse
  - Equinox
- Knopflerfish
- JEE Applikationsserver
  - JBoss
  - Glassfish
  - ...

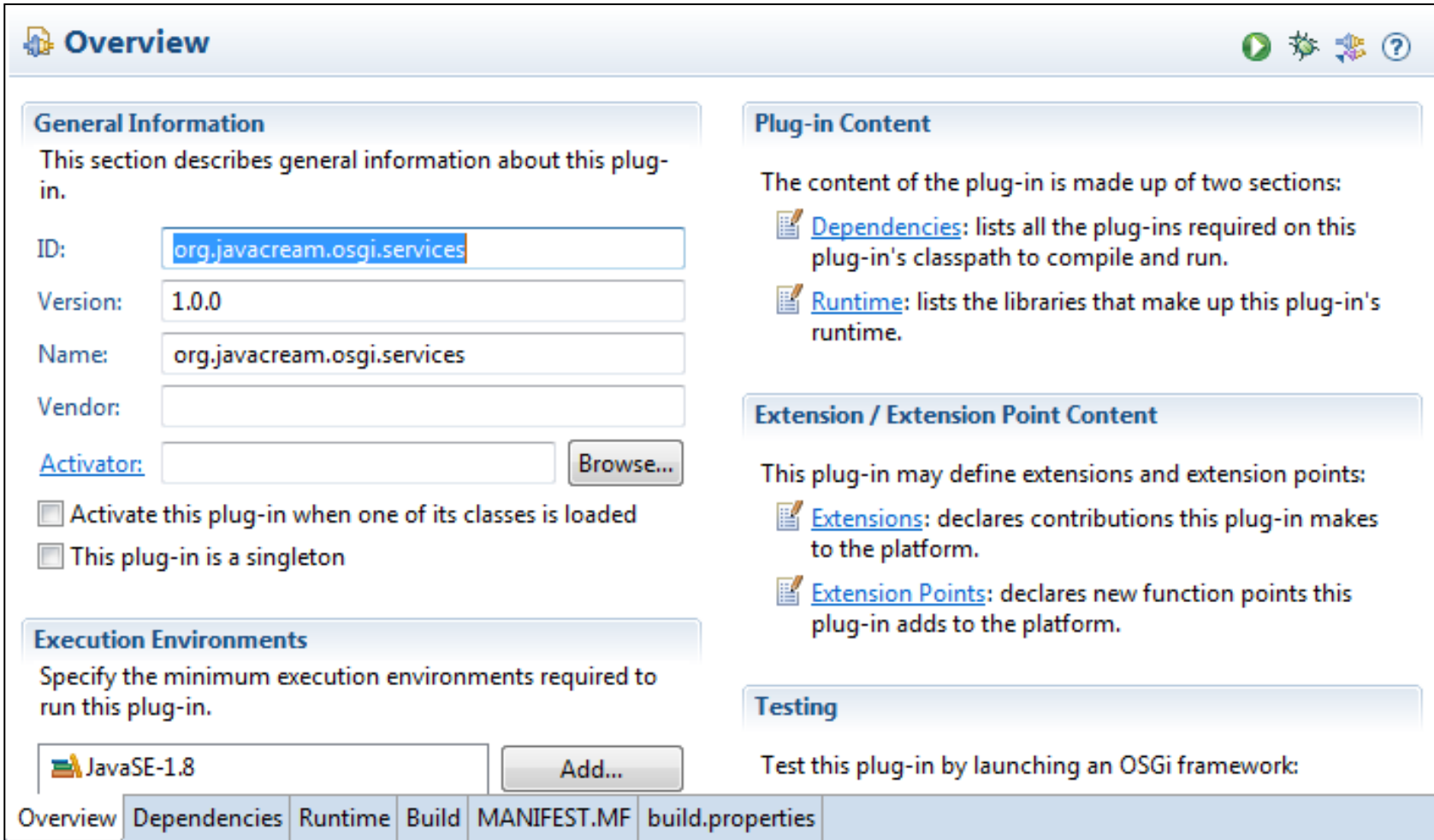
- Java-Server-Implementierungen
  - JEE Applikationsserver
  - Portal-Server
  - ESBs
  - Messaging Systeme
- Rich Client Platforms (RCPs)
  - Eclipse
  - Netbeans
- Java auf Embedded Systemen
  - Hier wird besonderer Wert auf eine leichtgewichtige Service-Plattform gelegt
- Klassische Unternehmens-Anwendungen
  - Eher wenig verbreitet (!)
    - Bei weitem nicht alle Projekte benötigen das von OSGi angebotene Level vom Modularisierung und Dynamik
  - Allerdings werden die Anwendungen häufig in auf OSGi basierenden Servern betrieben

- JBoss Modules
  - Grundlage des JBoss Applikationsservers
  - Support über RedHat
- Das Java-Modulkonzept
  - Eingeführt mit Java 9
  - Stellt allerdings erst einen Teil der OSGi-Funktionen zur Verfügung
- Service Oriented Architecture
  - Ausgelegt auf eine verteilte System-Architektur
  - Fein-granulare Microservices sind aber durchaus mit OSGi-Ansätzen vergleichbar

1.2

## **WERKZEUGE UND PROJEKTORGANISATION**

- Prinzipiell ist jede Java-Entwicklungsumgebung geeignet
  - Bundles sind normale Java-Archive
  - OSGi-Bibliotheken können als normale Abhängigkeiten integriert werden
- Allerdings ist es vorteilhaft, OSGi-spezifische Erweiterungen zu nutzen
  - Umgang mit Dependencies
  - Manifest-Editor
  - Projekt-Generatoren
  - Ausführen, Testen und Debuggen von Bundles
  - ...



The screenshot shows the Eclipse Manifest Editor with the 'Overview' tab selected. The interface is divided into two main columns. The left column contains sections for 'General Information' and 'Execution Environments'. The right column contains sections for 'Plug-in Content' and 'Extension / Extension Point Content'. At the bottom, there is a 'Testing' section and a tabbed interface for switching between different views.

**Overview**

**General Information**  
This section describes general information about this plug-in.

ID:

Version:

Name:

Vendor:

Activator:

☐ Activate this plug-in when one of its classes is loaded

☐ This plug-in is a singleton

**Execution Environments**  
Specify the minimum execution environments required to run this plug-in.

**Plug-in Content**  
The content of the plug-in is made up of two sections:

- [Dependencies](#): lists all the plug-ins required on this plug-in's classpath to compile and run.
- [Runtime](#): lists the libraries that make up this plug-in's runtime.

**Extension / Extension Point Content**  
This plug-in may define extensions and extension points:











- [Extensions](#): declares contributions this plug-in makes to the platform.
- [Extension Points](#): declares new function points this plug-in adds to the platform.

**Testing**  
Test this plug-in by launching an OSGi framework:

Overview | Dependencies | Runtime | Build | MANIFEST.MF | build.properties

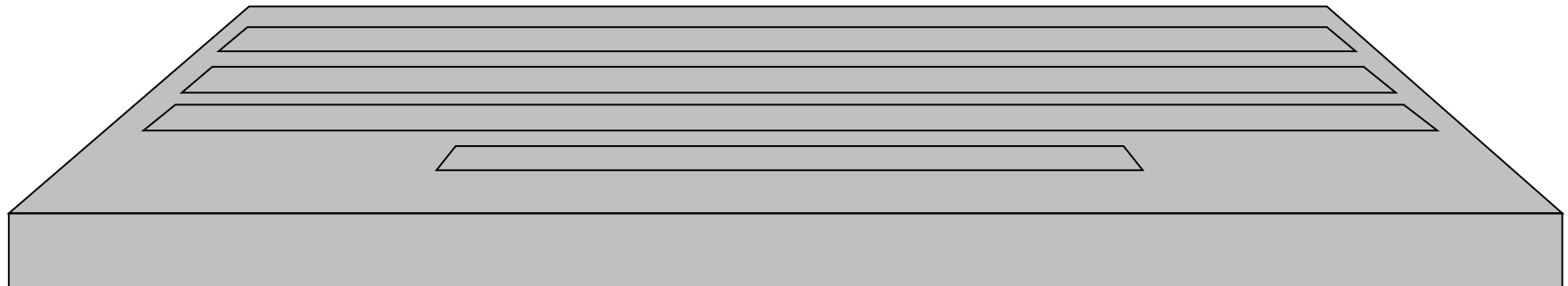


- OSGi motiviert stark zu einer fein-modularen Anwendungsarchitektur
  - Faustregel: Jeder zusammengehörige Use Case ist ein eigenes Bundle
- Die Benennung der Bundles erfordert eine hierarchische Struktur
  - Sinnvoll ist die Benutzung eines Paketnamens wie in Name
- OSGi-Bundles enthalten die Information einer Versionsnummer
  - Damit muss ein Bezug zum Versionsverwaltungssystem hergestellt werden

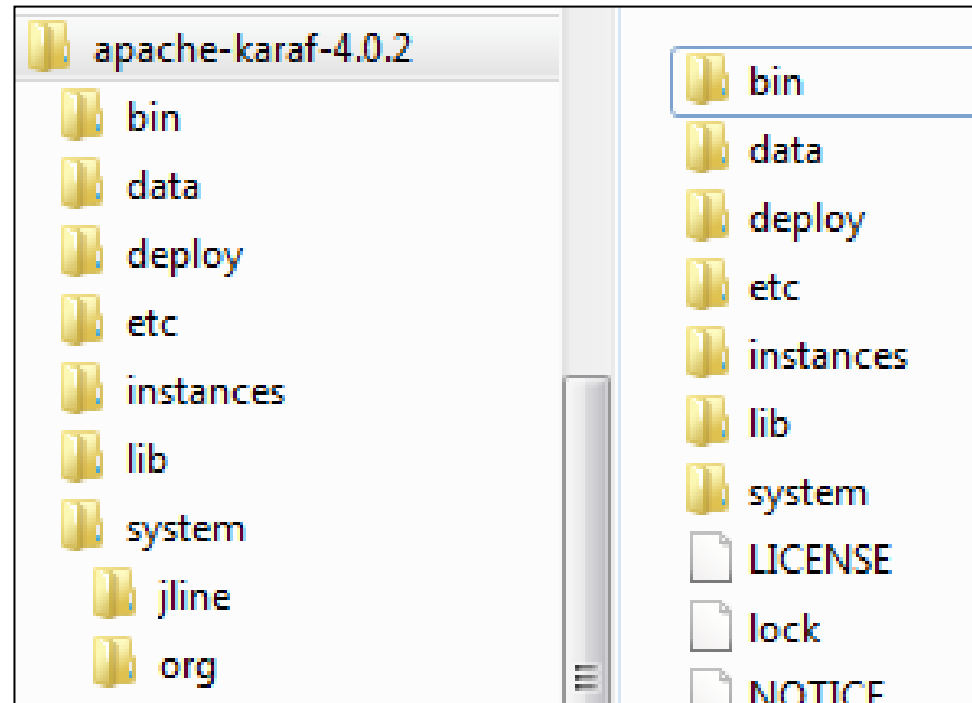
- ▷  com.javacream.storeservice.impl [git master]
- ▷  org.javacream.books.warehouse.actor [git master]
- ▷  org.javacream.books.warehouse.api [git master]
- ▷  org.javacream.books.warehouse.impl [git master]
- ▷  org.javacream.isbngenerator.actor [git master]
- ▷  org.javacream.isbngenerator.impl [git master]
- ▷  org.javacream.storeservice.actor [git master]
- ▷  org.javacream.storeservice.api [git master]
- ▷  org.javacream.storeservice.impl [git master]
- ▷  org.javacream.utils [git master]

- OSGi und Maven benutzen zumindest teilweise identische Konzepte
  - Dependencies entsprechen Maven-Koordinaten
    - Group-Id
    - Artefact-Id
    - Version
- Idee
  - Die OSGi-Metadaten werden aus dem Maven POM während des Build-Prozesses generiert
- Realisierung
  - bndtools (bndtools.org)
    - Name als Abkürzung von Bundle-Tools
    - Bestimmt die Abhängigkeiten durch Bytecode-Introspektion
  - Apache Felix mit dem BND-Plugin
    - beruht auf bndtools

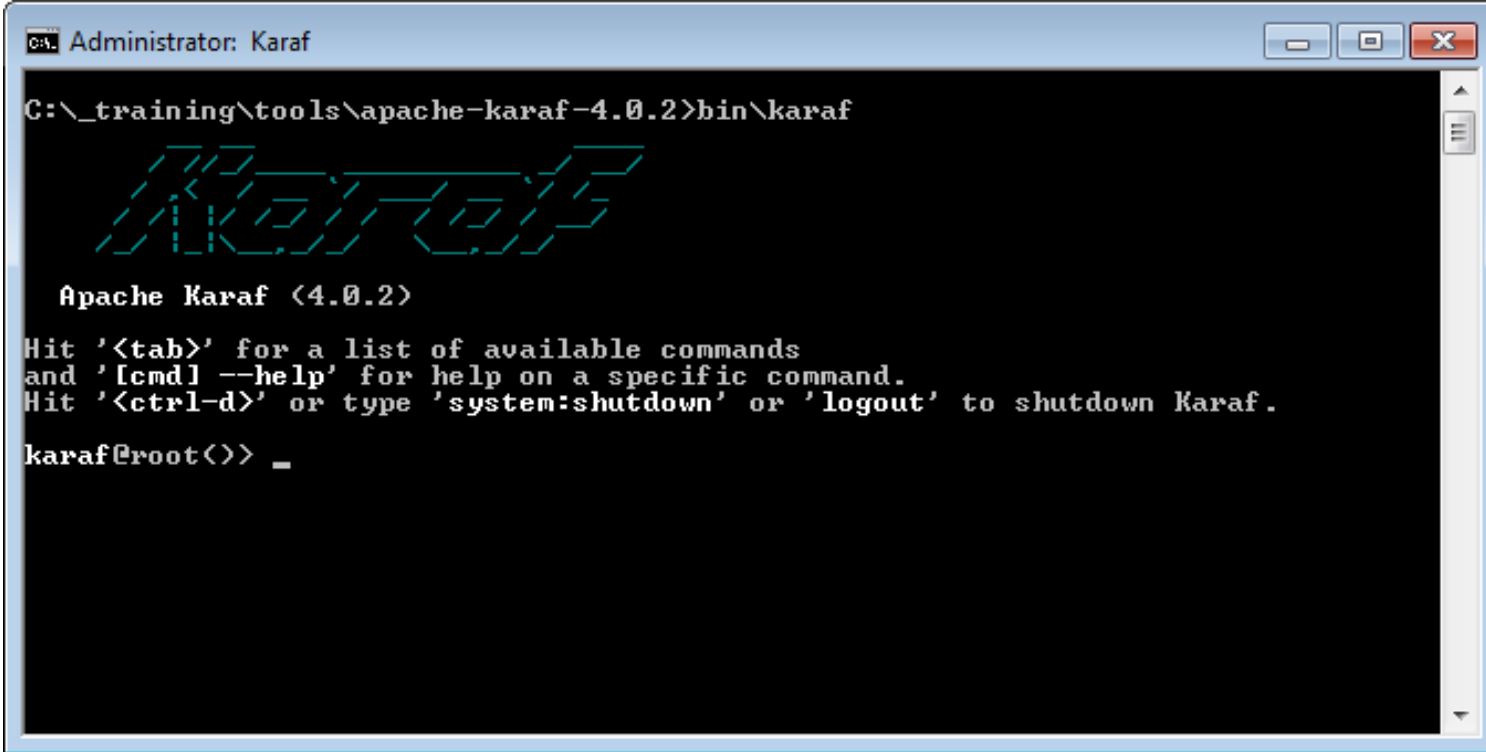
```
<plugin>
<groupId>org.apache.felix</groupId>
<artifactId>maven-bundle-plugin</artifactId>
<version>1.4.0</version>
<configuration><instructions>
<Bundle-Name>${project.name}</Bundle-Name>
<Bundle-Version>${project.version}</Bundle-
Version>
<Bundle-Activator>Foo</Bundle-Activator>
</instructions></configuration>
</plugin>
```



- Open Source Implementierung der OSGi-Spezifikation
- Distribution enthält eine sofort lauffähige Umgebung mit
  - Startskripte
  - OSGi Core
  - Vorinstallierten Bundles
  - Karaf-Konsole
  - Hot Deployment von Bundles
  - Logging
  - Einige Enterprise Services



# Gestartetes Karaf mit Konsole

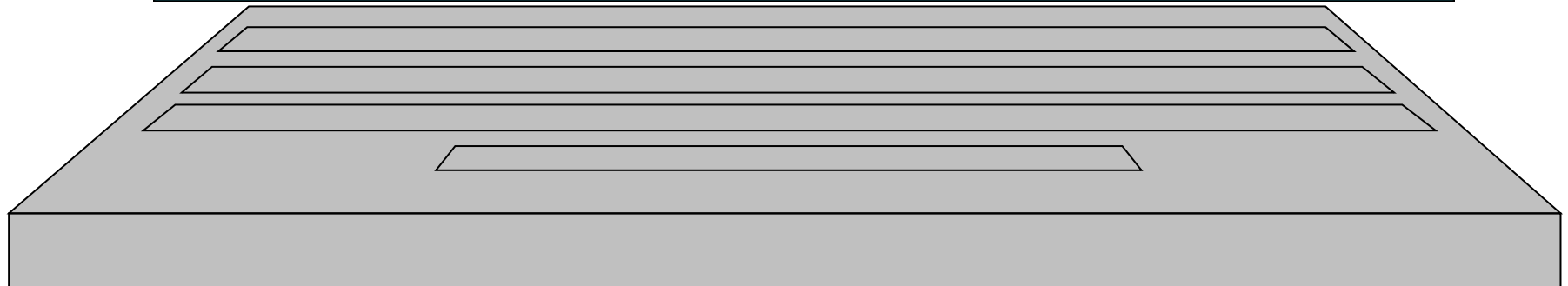


```
C:\_training\tools\apache-karaf-4.0.2>bin\karaf

  Apache Karaf <4.0.2>

Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type '<system:shutdown>' or '<logout>' to shutdown Karaf.

karaf@root(>>) _
```



- **Starten im Installationsverzeichnis**
  - `bin\karaf`
  - Nicht direkt im bin, sonst werden die Bundles nicht gefunden!
- **Einige Konsolenbefehle**
  - `install <Pfad zur Bundle-jar>`
    - Nach der Installation hat das Bundle eine interne fortlaufende Nummer als ID
  - `uninstall <Bundle-ID>`
  - `update <Bundle-ID>`
  - `start <Bundle-ID>`
  - `stop <Bundle-ID>`
  - `list`
- **Fehlermeldungen**
  - Nur interne Fehler tauchen in der Standard-Konfiguration auf der Konsole auf
  - **Log-Datei** `KARAF_HOME\data\log\karaf.log`

2

# PROGRAMMIERUNG



2.1

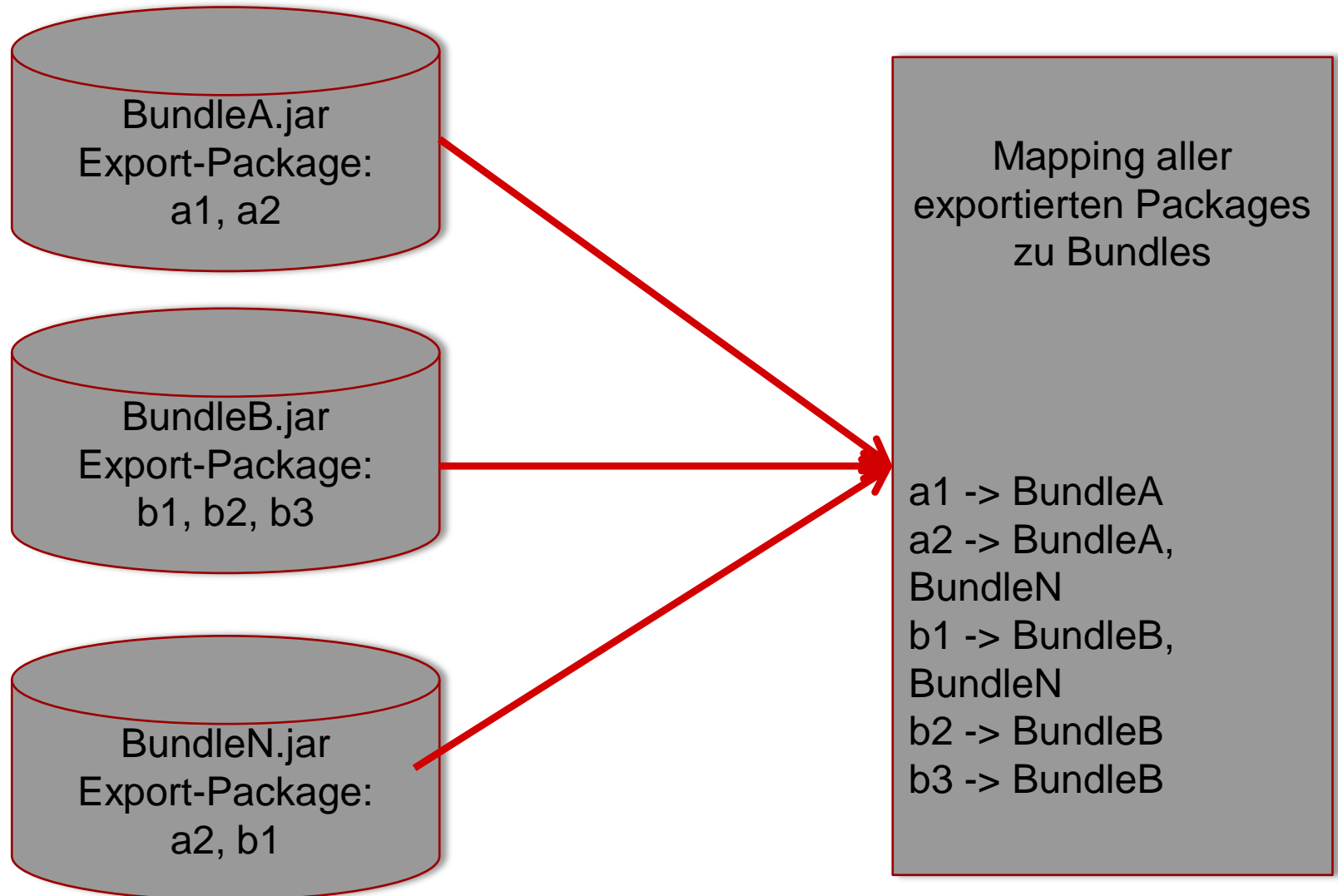
## **BUNDLES UND KLASSENLADER**

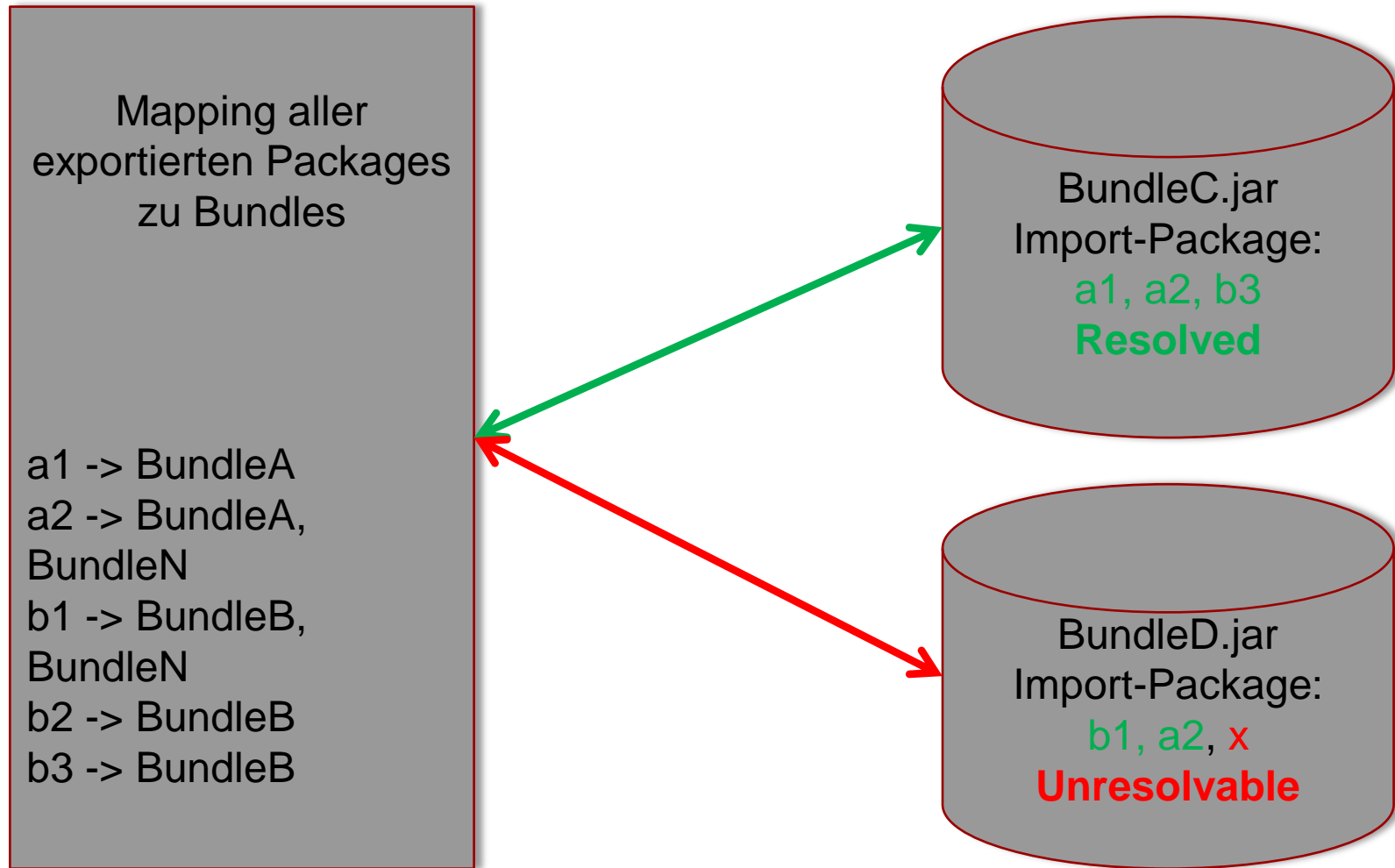
- Beschreibende Informationen
  - `Bundle-ManifestVersion: 2`
  - `Bundle-Name: org.javacream.demo.osgi.first`
  - `Bundle-SymbolicName: org.javacream.demo.osgi.first`
  - `Bundle-Version: 1.0.0`
  - `Bundle-Vendor: Javacream`
- Deklarative Informationen
  - `Bundle-RequiredExecutionEnvironment: JavaSE-1.8`
  - `Import-Package: org.osgi.framework;version="1.5.0"`
  - `Export-Package: org.javacream.demo.osgi.first`
- Optionale Activator-Klasse, die den Bundle-Lifecycle implementiert
  - `Bundle-Activator:`  
`org.javacream.demo.osgi.first.EchoActivator`
  - Der Activator fungiert als `main`-Methode des Bundles

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;

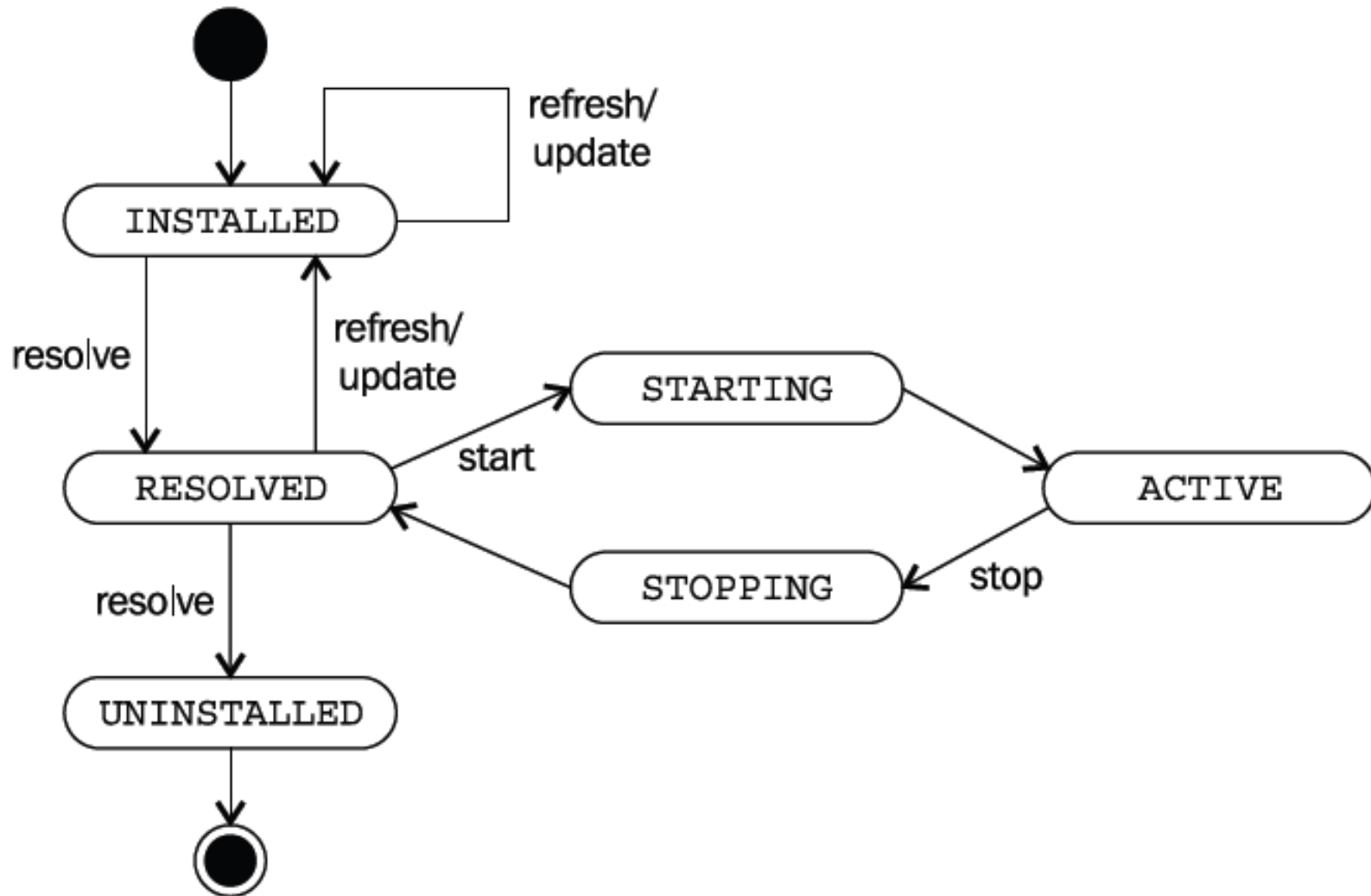
public class SimpleActivator implements
BundleActivator {
    public void start(BundleContext bundleContext) throws
Exception {
        System.out.println("Hello, starting " + this);
    }
    public void stop(BundleContext bundleContext) throws
Exception {
        System.out.println("Hello, stopping " + this);
    }
}
```

- Ein eigenes Verfahren zur Kapselung von Klassen
- Im Gegensatz zu Standard-Archiven muss ein Bundle explizit
  - alle intern benötigten Pakete importieren
  - alle Pakete, die das Bundle anderen Bundles anbietet, exportieren
- Dies funktioniert nur
  - in einer OSGi-Runtime
  - in einer OSGi-kompatiblen Entwicklungsumgebung
- Ein Bundle kann auch in einer Nicht-OSGi Anwendung benutzt werden
  - Dann liegt es aber als normales Java-Archiv im Klassenpfad
  - Keinerlei Berücksichtigung der Manifest-Datei
- OSGi prüft für jedes Bundle, ob dessen Imports von Exports abgedeckt werden können
  - Resolve
  - In neuen OSGi-Versionen wird allgemeiner von „Requirements“ und „Capabilities“ eines Bundles gesprochen



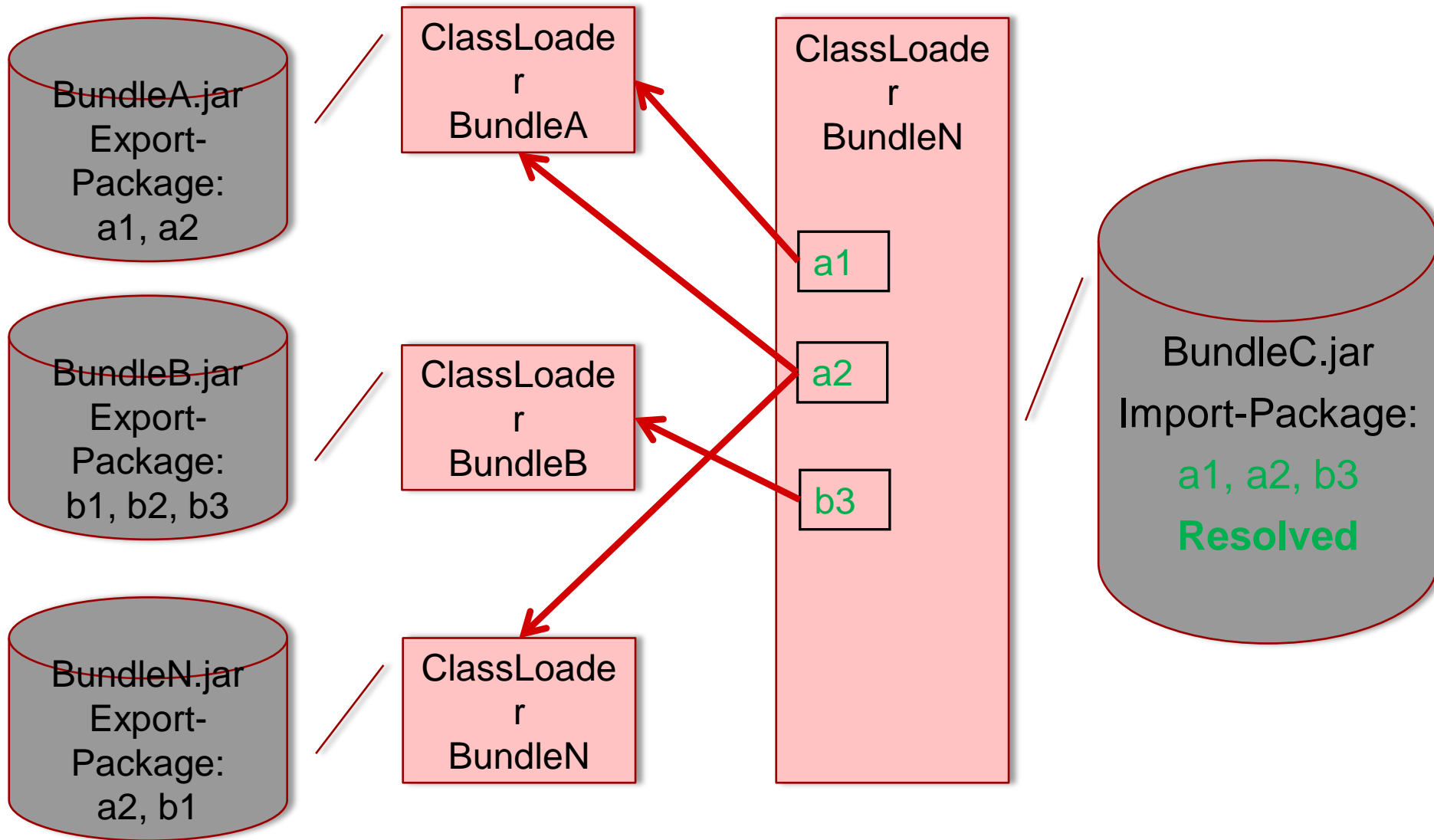


- Die Übergänge erfolgen durch
  - Installation des Bundles
  - Starten des Bundles
    - Hier wird die `start`-Methode des Activators aufgerufen
  - Stoppen des Bundles
    - Hier wird die `stop`-Methode des Activators aufgerufen
- Hinweise
  - Ein einmal ausgeführtes Resolve kann durch ein Update des Bundles wiederholt werden
  - Die OSGi-Spezifikation enthält noch weitere States wie `FAILED`
  - OSGi-Provider merken sich den Zustand eines Bundes und stellen diesen nach einem Neustart wieder her

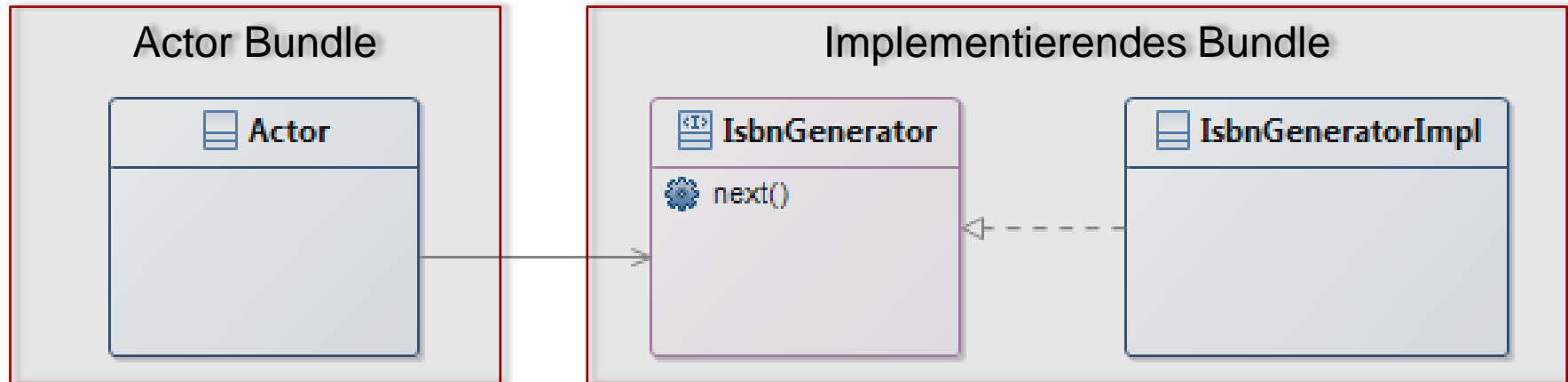




- Für jedes Bundle wird ein eigener Klassenlader benutzt
- OSGi verwaltet für alle installierten Bundles eine Liste der exportierten Pakete
  - Ein Paket kann auch von mehreren unterschiedlichen Bundles exportiert werden
- Wird ein Bundle gestartet wird geprüft, ob die Runtime-Umgebung alle importierten Pakete zur Verfügung stellen kann
  - Resolve-Phase
  - Im Fehlerfall kann das Bundle nicht aktiviert werden
- Zum Laden der Klassen eines importierten Paketes delegiert der Bundle-Klassenlader an den oder die Klassenlader der exportierenden Bundles

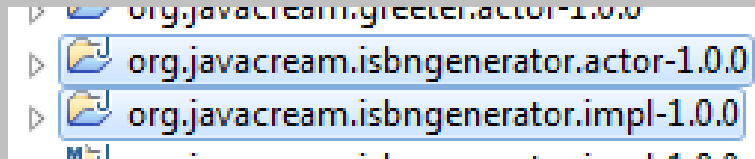


- Ein einfaches Klassendiagramm
  - Erzeugung einer eindeutigen ISBN-Nummer



- Implementierung mit 2 Bundles
  - Actor-Bundle
  - Impl-Bundle mit
    - Interface
    - Implementation
    - Factory/Context

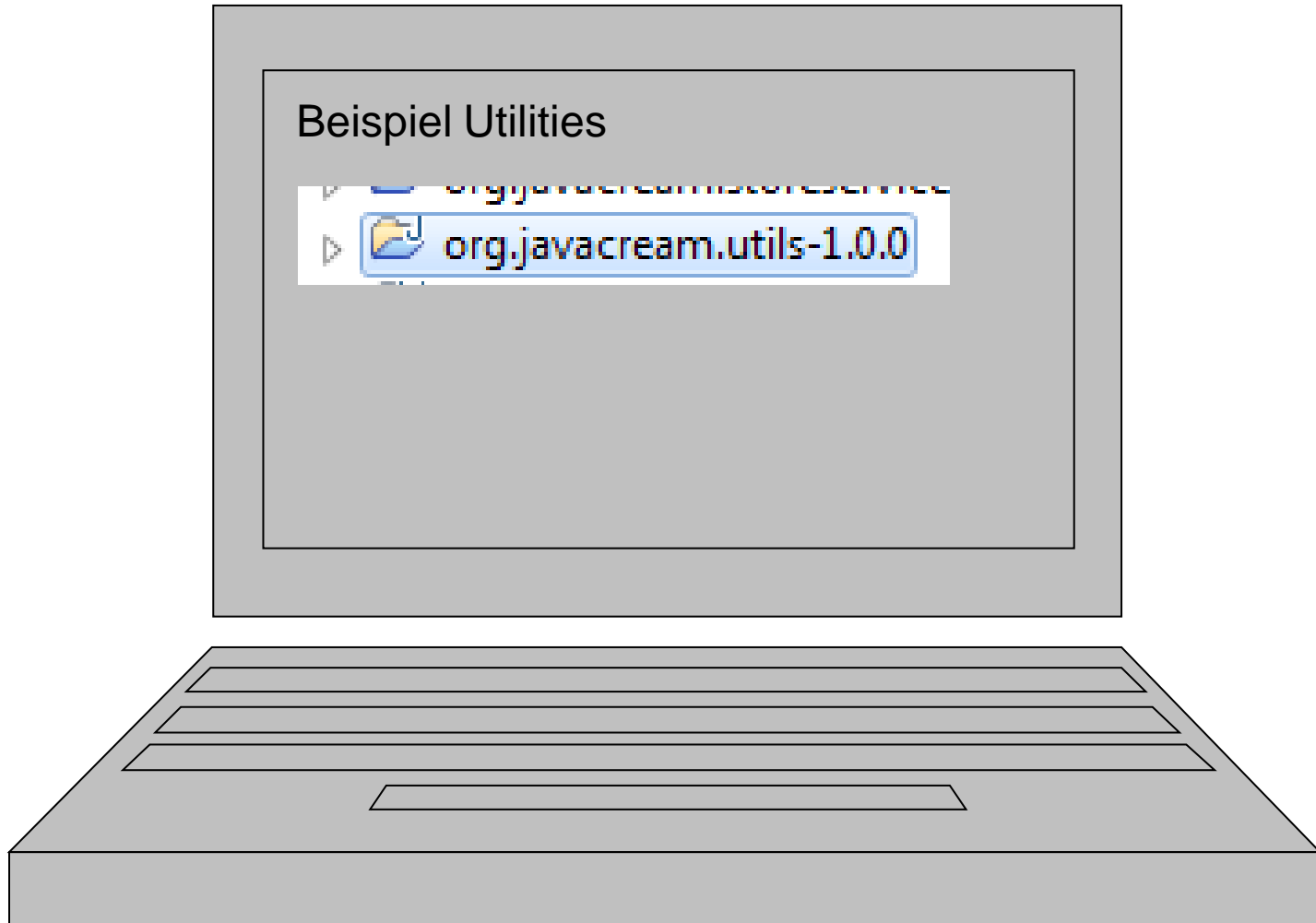
## Beispiel IsbnGenerator



org.javacream.greeter.actor-1.0.0  
org.javacream.isbngenerator.actor-1.0.0  
org.javacream.isbngenerator.impl-1.0.0

- Standardwert: „.“
- Ein Bundle kann intern weitere Bibliotheken als normale Java-Archive mitbringen, die im Bundle-Classpath eingetragen werden
  - Vorsicht: Der Standard wert „.“ wird dabei überschrieben und muss deshalb explizit angegeben werden
- Sinnvoller Einsatz
  - Soll der Build-Prozess unabhängig von OSGi sein, wird ein normales JAR erzeugt, das dann einem Bundle hinzugefügt wird
  - Third-Party-Libraries sollen als Bundle benutzt werden, sind aber nicht selber Bundles
    - Die benötigten Packages der Bibliothek werden dann exportiert
    - Vorsicht: Nicht alle Bibliotheken können OSGi-konform benutzt werden!
      - Falls diese intern selber ClassLoader benutzen kann es in einer OSGi-Umgebung zu Fehlern kommen

- Ein Bundle kann Dynamic Imports definieren
  - `DynamicImport-Package: *`
- Damit werden Packages mit Platzhaltern importiert
- Vorsicht: Mit dem oben gezeigten \*-Import geht das feingranulare Konzept der OSGi-Kapselung verloren
  - Dynamic Imports deshalb nur mit Bedacht zu verwenden
  - Beispiel: Framework mit Reflection-Benutzung und allgemeinen Hilfsklassen



## 2.2

# DER BUNDLECONTEXT

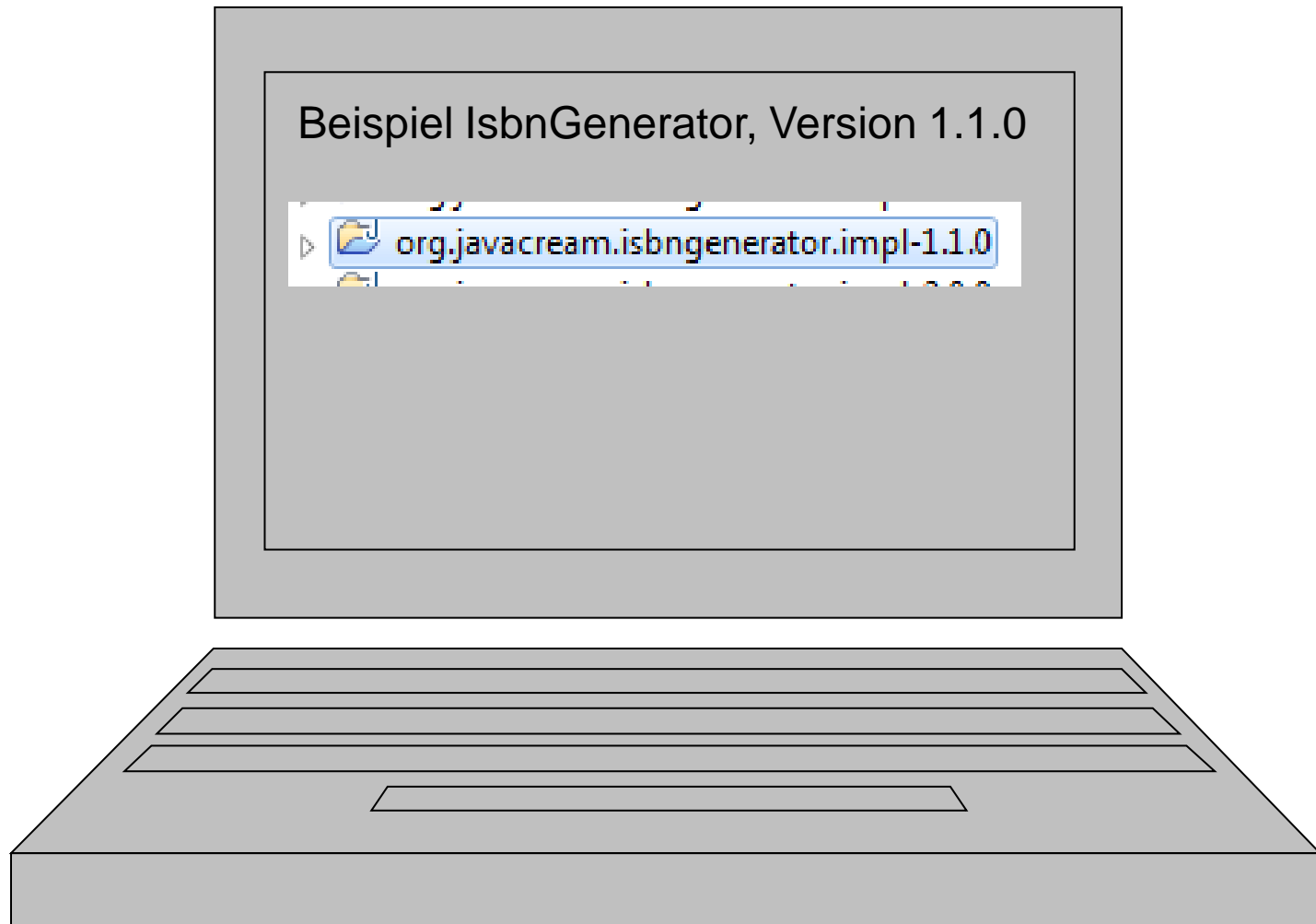


- Der `BundleContext` ist eine OSGi-Hilfsklasse, die auch Aufgaben von `java.lang.Class` und `java.lang.ClassLoader` übernimmt
  - Laden von Ressourcen
  - Laden von Klassen
- Darüber hinaus
  - Zugriff auf das Bundle-Manifest
    - Auslesen der Header
  - Registrieren von Services
  - Nachschlagen von Services
  - OSGi-Events
    - `BundleListener`
    - `FrameworkListener`
    - `ServiceListener`
  - Prinzipiell Zugriff auf jedes installierte Bundle
    - Kann bei aktiviertem `SecurityManager` unterbunden werden

```
public void start(BundleContext bundleContext) throws  
Exception {  
    Properties props = new Properties();  
    Bundle bundle = bundleContext.getBundle();  
    URL resource =  
    bundle.getResource("/isbngenerator.properties");  
    props.load(resource.openStream());  
    //...
```

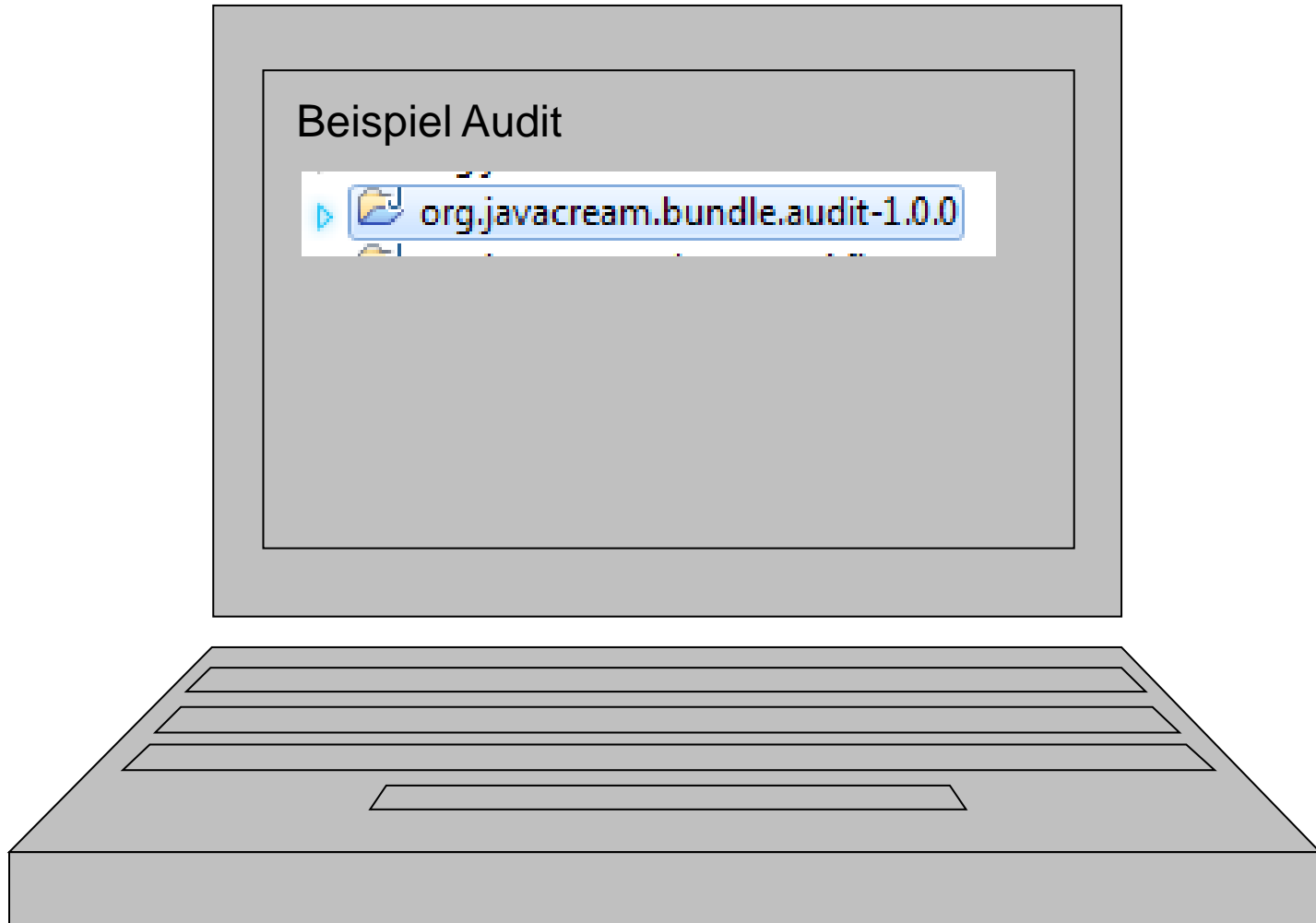
```
public void start(BundleContext bundleContext) throws  
Exception {
```

```
    Properties props = new Properties();  
    Bundle bundle = bundleContext.getBundle();  
    Dictionary<String, String> headers = bundle.getHeaders();  
    String strategy = headers.get("IsbnGenerator-Strategy");  
    //...
```



```
class SimpleBundleListener implements BundleListener{
    @Override
    public void bundleChanged(BundleEvent bundleEvent) {
        System.out.println("Received bundle event: " + bundleEvent);
        System.out.println("Type: " + bundleEvent.getType());
        System.out.println("Bundle: " + bundleEvent.getBundle());
        System.out.println("Origin: " + bundleEvent.getOrigin());
    }
}
```

```
class SimpleFrameworkListener implements FrameworkListener {  
    public void frameworkEvent(FrameworkEvent frameworkEvent) {  
        System.out.println("Bundle: " + frameworkEvent.getBundle());  
        System.out.println("Type: " + frameworkEvent.getType());  
        System.out.println("Throwable: " + frameworkEvent.getThrowable());  
    }  
}
```



2.3

## **OSGI SERVICES**



- Im Prinzip eine statische HashMap
- Bundle registrieren einen Service unter der verpflichtenden Angabe eines Interfaces
- Zusätzlich kann ein Dictionary mit beliebigen key-value-Paaren zur weiteren Differenzierung benutzt werden
- Der Zugriff auf Services erfolgt
  - über die Schnittstelle
  - über einen Filter
    - ein LDAP-Suchausdruck
  - Eine Suche kann potenziell mehrere Treffer ergeben

- Services werden damit zur Laufzeit
  - hinzugefügt
  - geändert
- Allerdings muss das Actor-Bundle immer noch administrativ verwaltet werden
  - `start-stop`
  - Wäre die Implementierung im selben Bundle wie das Interface würde das keinen Effekt haben!
    - Das Actor-Bundle hat die Klassendefinition der Schnittstelle und der Implementierung aus dem selben Klassenlader geladen
- Beispiel: Workflow zum Update der Service-Implementierung
  - update auf Implementierungs-Bundle
  - `stop` des Actor-Bundles
  - `start` des Actor-Bundles

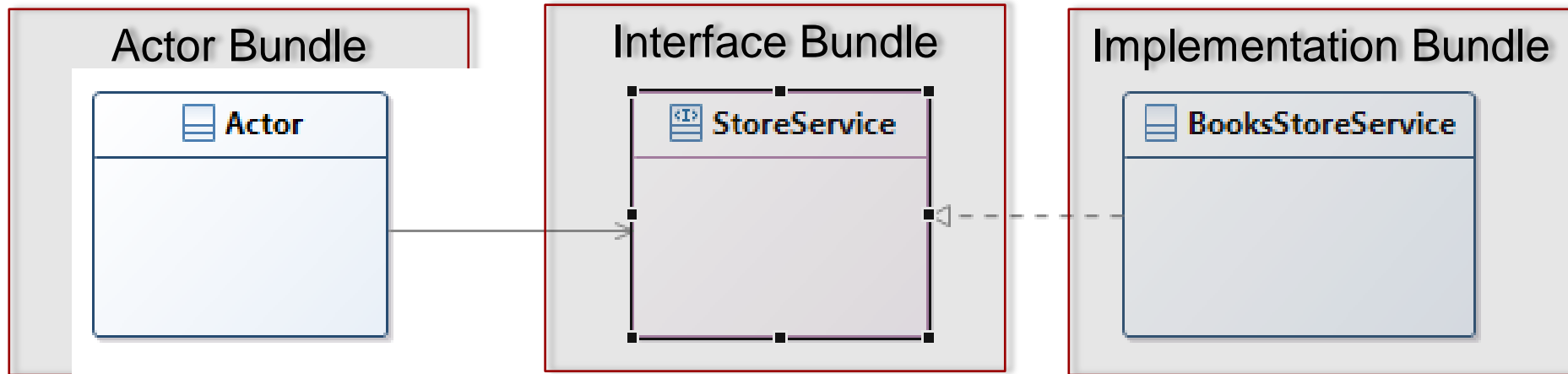
```
public class StoreServiceActivator implements BundleActivator {
    private ServiceRegistration<StoreService> serviceReference;

    public void start(BundleContext bundleContext) throws
Exception {
        BooksStoreService booksStoreService = new BooksStoreService();
        Hashtable<String , String> dictionary =
            new Hashtable<String, String>();
        serviceReference = bundleContext.registerService(
            StoreService.class, booksStoreService, dictionary);
    }

    public void stop(BundleContext bundleContext) throws Exception
    {
        serviceReference.unregister();
    }
}
```

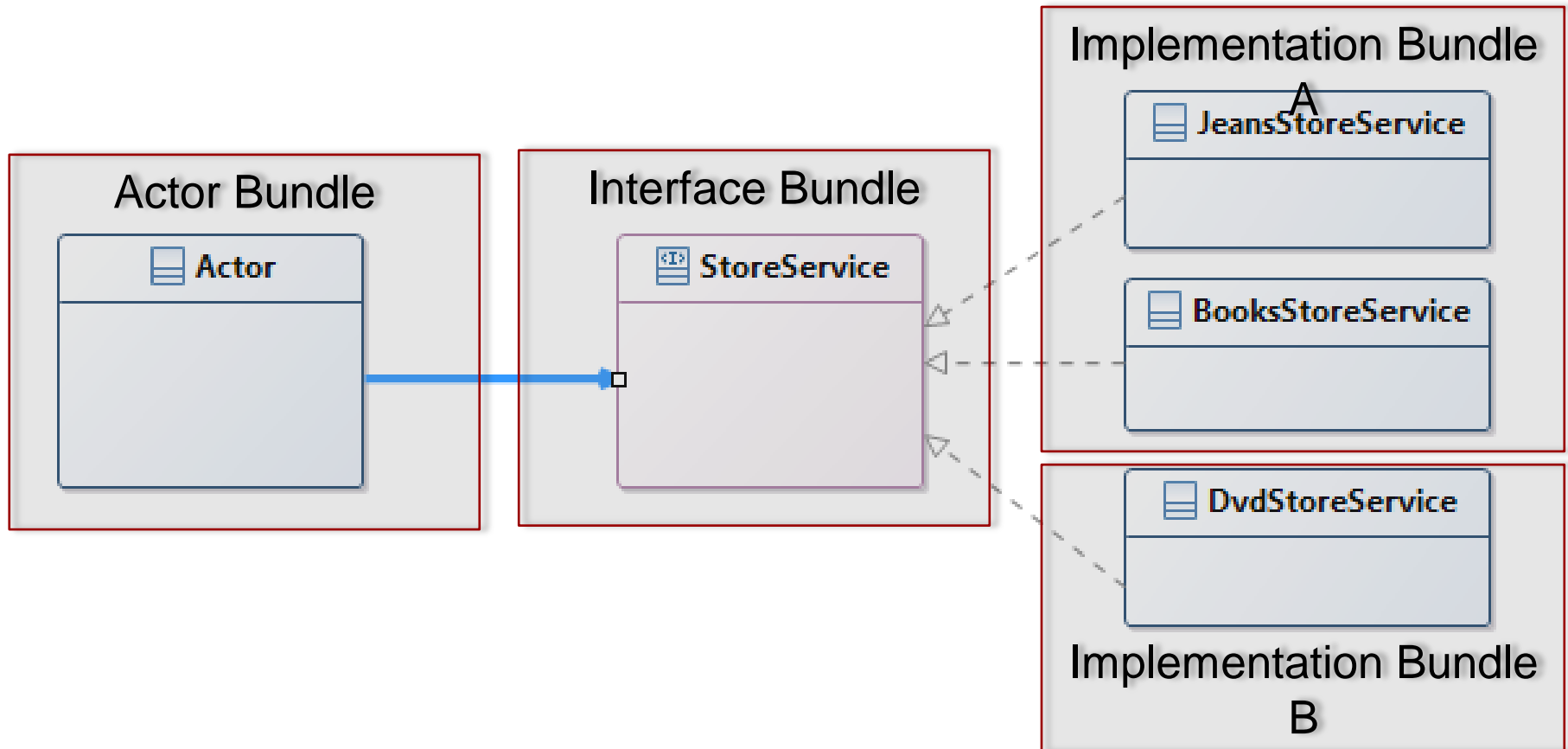
```
public void start(BundleContext bundleContext) throws Exception {  
    ServiceReference<StoreService> storeServiceServiceReference =  
        bundleContext.getServiceReference(StoreService.class);  
    StoreService storeService =  
        bundleContext.getService(storeServiceServiceReference);  
    //...
```

- Der Zugriff auf die Services erfolgt zwingend über ein Interface
- Falls das Interface vom implementierenden Bundle getrennt wird, sind Services dynamisch verwendbar
  - Actor-Bundle
  - Interface-Bundle
  - Implementierung-Bundle








- Service-Implementierungen können in der Registry durch die Angabe eines Dictionaries mit key-value-Paaren differenziert werden
- Service-Implementierungen können aus verschiedenen Bundles stammen
- Beim Lookup kann der Actor
  - Eine Liste aller Services anfordern
  - Mit einem Filter einen bestimmten Service auswählen

# Mehrere Service-Implementierungen





## Beispiel StoreService

- ▷  org.javacream.storeservice.actor-1.0.1
- ▷  org.javacream.storeservice.actor-2.0.0
- ▷  org.javacream.storeservice.api-1.0.0
- ▷  org.javacream.storeservice.impl-1.0.0
- ▷  org.javacream.storeservice.impl-2.0.0

- Um eine echte Dynamik zu Erreichen kann ein `ServiceListener` benutzt werden
  - Dieser wird von der OSGi-Runtime aufgerufen, sobald ein Service registered oder unregistered wird
- Vorsicht!
  - Eine stabile Implementierung eines dynamischen Services ist relativ aufwändig
    - Ein gerade registrierter Service kann theoretisch sofort wieder verschwinden
    - Multithreading-Bedarf, da Listener von einem einzigen Thread aufgerufen werden und deshalb keinesfalls blockieren dürfen
      - Genau wie beim start/stop des BundleActivators

- Der `ServiceTracker` ist Bestandteil der OSGi-Spezifikation
- Er realisiert die aufwändige Dynamik des Service-Lookups

```
ServiceTracker<StoreService, StoreService> storeServiceTracker =  
    new ServiceTracker<StoreService, StoreService>  
        (bundleContext, StoreService.class, null);
```

```
storeServiceTracker.open();
```

- Für eine echte Dynamik wird noch eine Proxy-Klasse benötigt
  - Diese holt sich bei jedem fachlichen Methodenaufruf vom `ServiceTracker` den aktuell gültigen Service
  - Ab jetzt genügt es, das implementierende Bundle zu aktualisieren
    - Ein Restart des Actors ist nicht mehr notwendig

Beispiel StoreService-Actor mit  
ServiceTracker

org.javacream.storeservice.actor-2.5.0

2.4

## **VERSIONIERUNG VON BUNDLES**

- Export- und Import-Package unterstützen Qualifier
  - Beliebige Menge von `key=value`-Paaren
  - Über Semikolon abgetrennt
    - `Import-Package: org.osgi.framework;version="1.3.0"`
- Die Versionsnummer ist ein spezieller Qualifier
  - Unterstützung von Version-Ranges
    - `[1.0, 2.0)`
      - eine beliebige 1.x-Version
    - `[1.0, 1.1]`
      - 1.0.x bis 1.1 inklusive
    - Ohne Angabe eines Ranges ist die Obergrenze „unendlich“
  - Default-Version ist 0.0.0
    - Nicht die Bundle-Version!

- Falls keine fixe Versionsnummer angegeben ist benutzt OSGi beim Resolve stets
  - eine passende, bereits aufgelöste Version
  - die höchste passende Version
- Vorsicht:
  - Dies kann beim Neustart der Umgebung zu einem geänderten Verhalten führen!

# Ein Resolve-Problem: Ausgangs-Szenarium

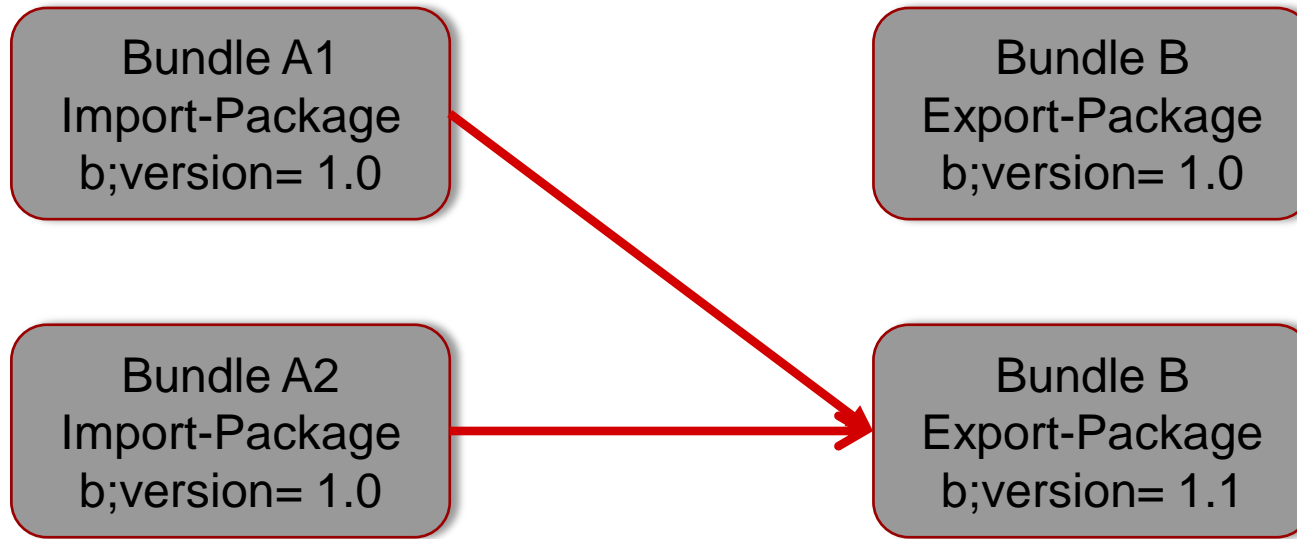
Bundle A1  
Import-Package  
b;version= 1.0

Bundle B  
Export-Package  
b;version= 1.0

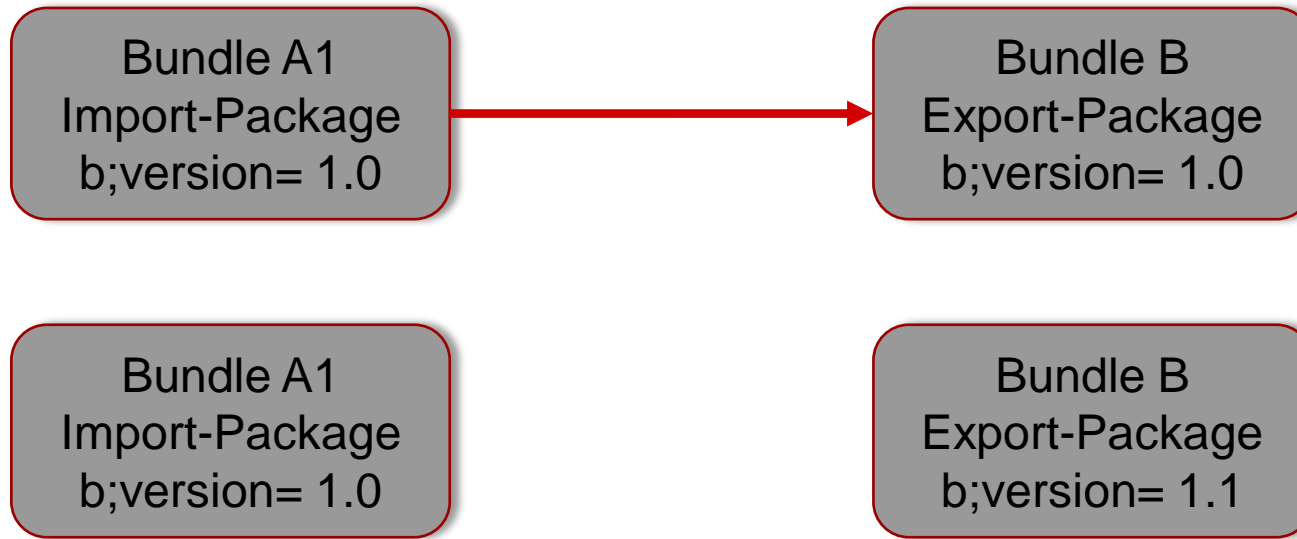
Bundle A2  
Import-Package  
b;  
version= 1.0

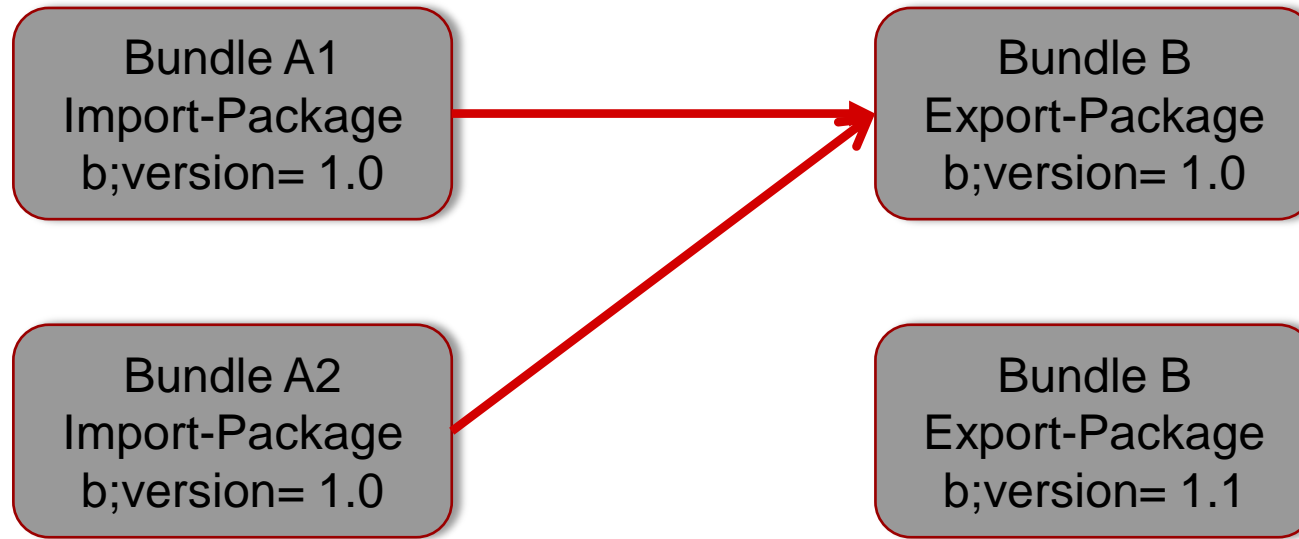
Bundle B  
Export-Package  
b;version= 1.1

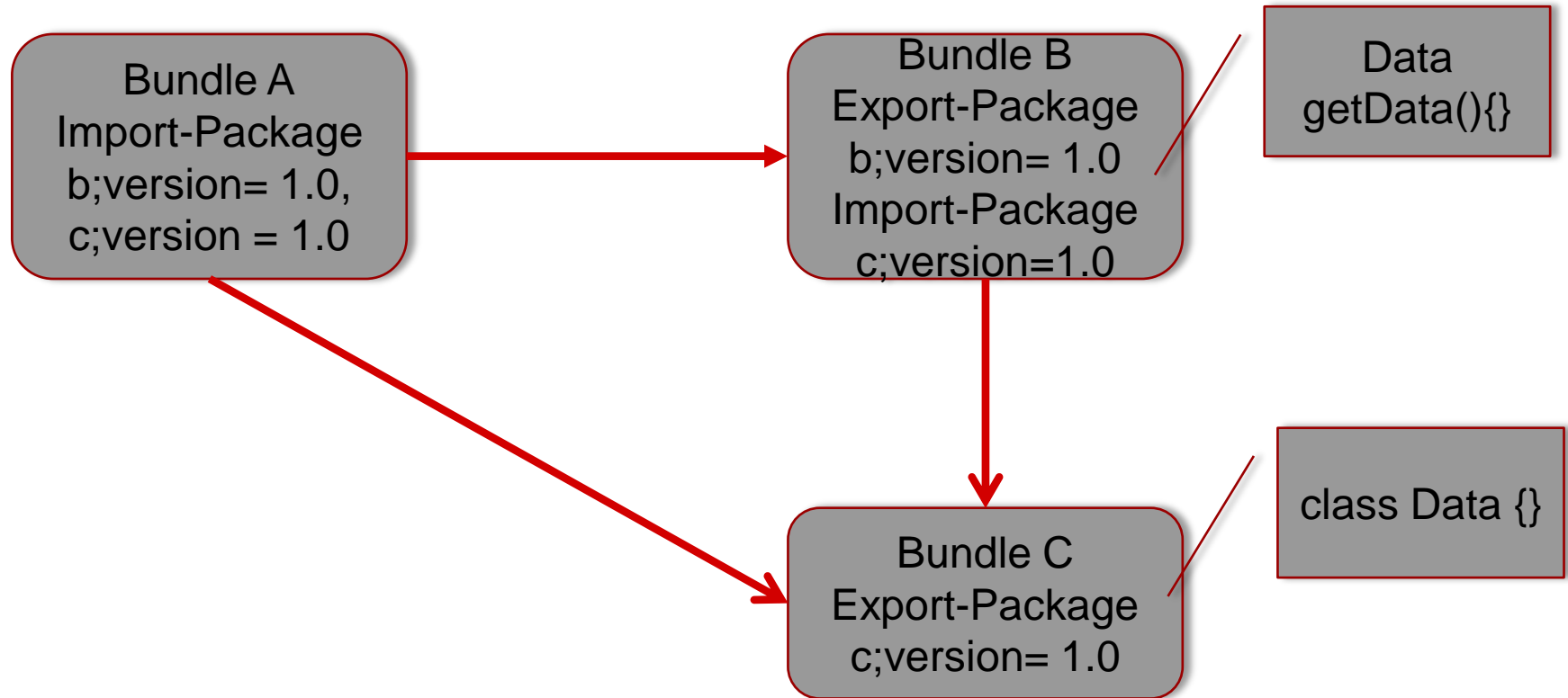


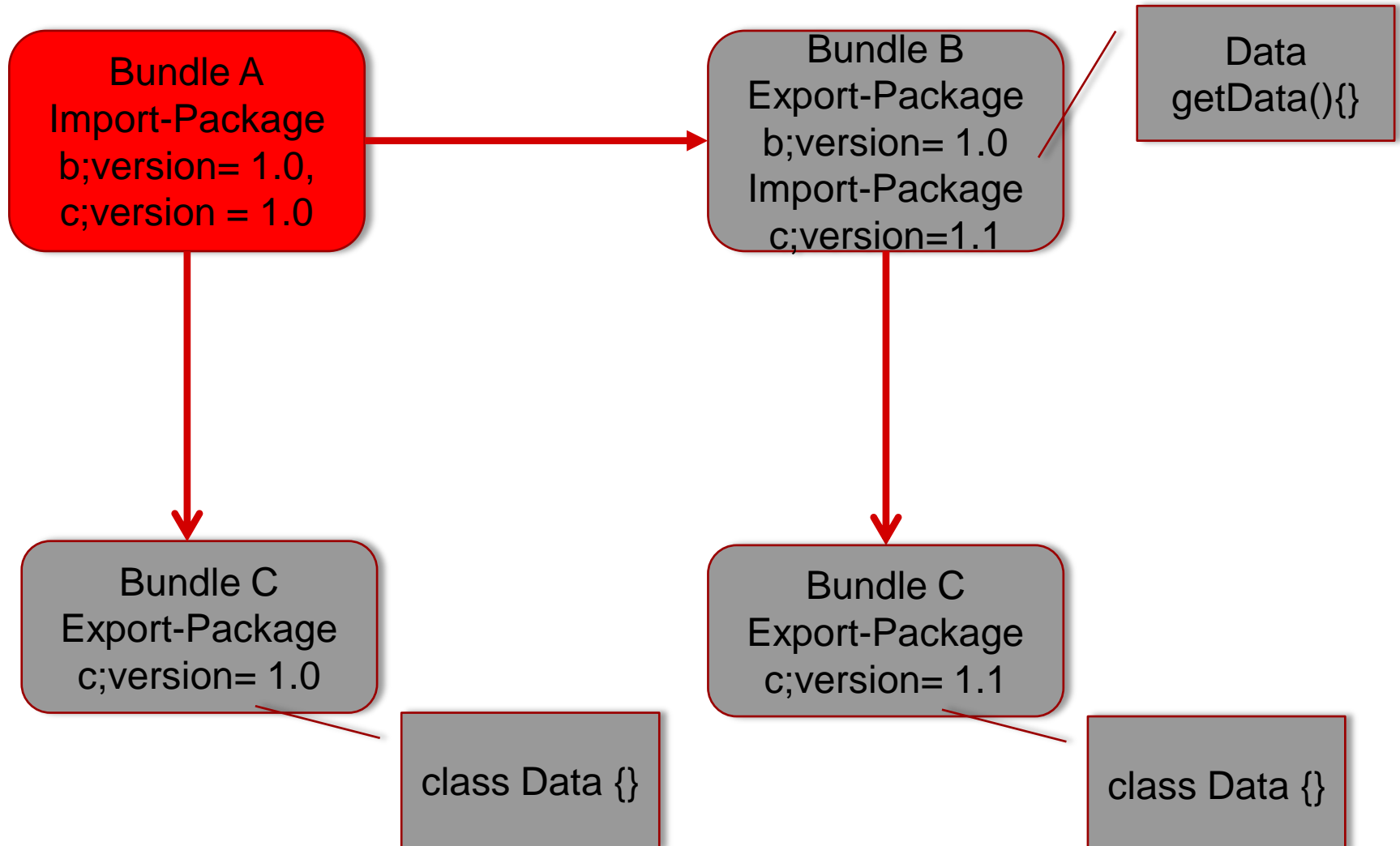


# Ein Resolve-Problem: Ausgangs-Szenarium 2









- In dieser der eben dargestellten Situation bekommt Bundle A das Klassenobjekt Data von zwei unterschiedlichen Klassenladern geliefert
  - Dies führt in jedem Fall zu einem Fehler
  - Erläuterung:
    - Der Resolve-Mechanismus hat keine Kenntnis darüber, dass das Bundle B eine Methode enthält, die als Rückgabewert einen Typ des Packages c liefert
- Lösungsansatz
  - Bundle B benutzt die `uses`-Klausel
  - Damit wird dem Resolve-Mechanismus diese verborgene Abhängigkeit signalisiert
    - Eine Lösung des Problems ist nur möglich, wenn die Version-Ranges von Bundle A und Bundle B bezüglich des Imports von Package c kompatibel sind!

3

## WEITERFÜHRENDE THEMEN

3.1

## **BLUEPRINT**



- Bei der Installation eines Bundles erweitert der Extender dieses Bundle
  - Der Extender selber ist ein eigenes Bundle,
  - der einen `BundleListener` benutzt
- Durch den `BundleContext` hat der Extender den Vollzugriff auf das neu installierte Bundle
  - Insbesondere können Klassen geladen werden
  - Zugriff auf alle Ressourcen
  - Zugriff auf den Manifest-Header

- Für Web-Anwendungen
- Ablauf
  - Hat das Bundle eine Datei WEB-INF/web.xml?
  - Falls ja: Parsen der Datei und Registrieren der darin definierten Web Komponenten beim http-Server

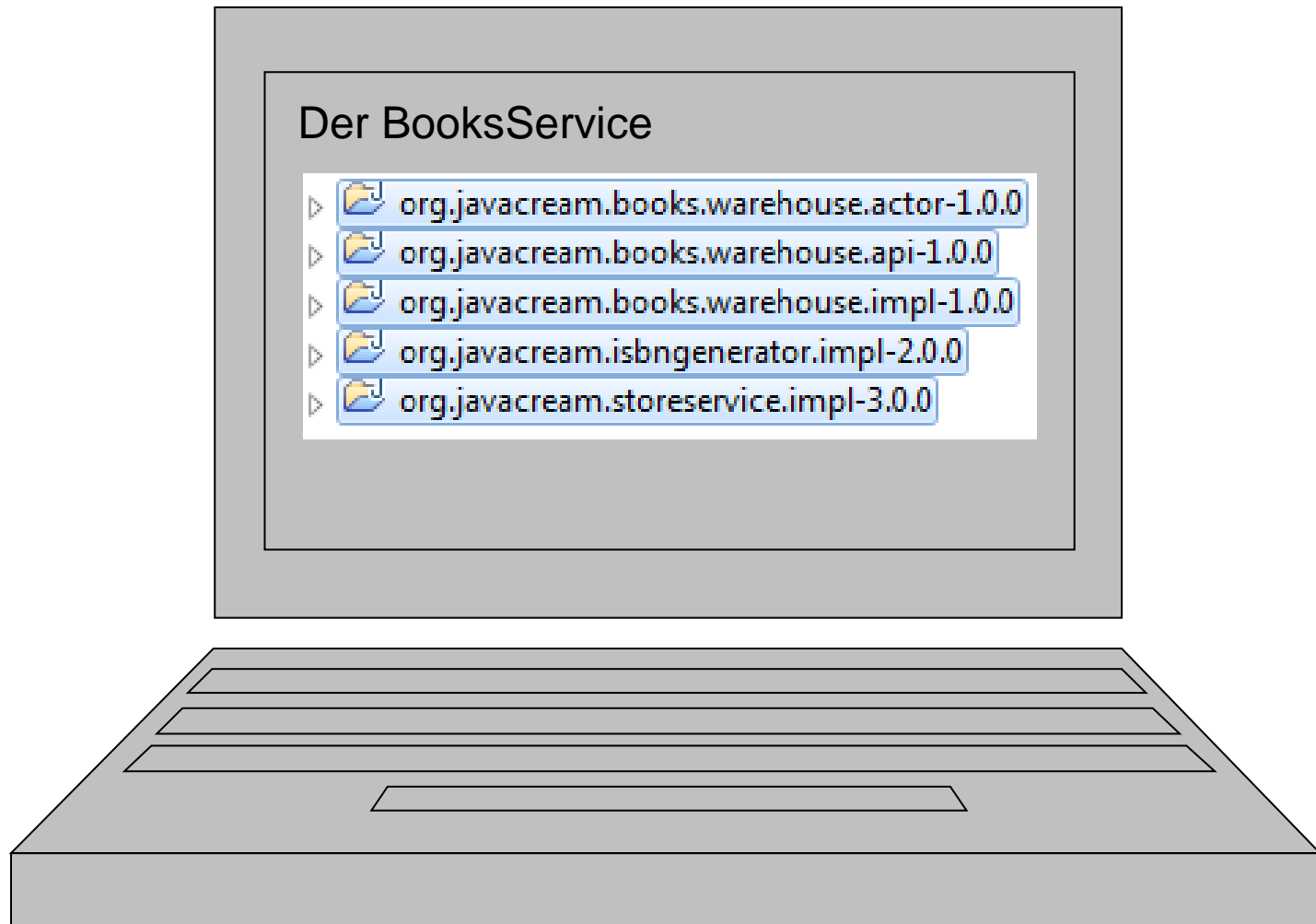
- Dependency Injection Framework der OSGi Alliance
  - Vormals Spring Dynamic Modules
  - Von der Spring Community allerdings nicht mehr weiter verfolgt, sondern an die OSGi-Alliance übergeben
- XML-basierte Definition von Beans und deren Properties
  - Damit wird das Objekt-Geflecht der Anwendung definiert
- XML-basierte Registrierung von Services
- XML-basierter Zugriff auf Services über reference

```
<?xml version="1.0" encoding="UTF-8"?>
<blueprint>
  <bean id="mapBooksService"
class="org.javacream.books.warehouse.business.MapBooksService"
init-method="init">
    <property name="storeService" ref="storeService" />
    <property name="isbnGenerator" ref="isbnGenerator" />
  </bean>

  <service id="booksService" interface="org.javacream.books.warehouse.BooksService"
ref="mapBooksService">
  </service>

  <reference id="storeService" interface="org.javacream.storeservice.StoreService" />
  <reference id="isbnGenerator" interface="org.javacream.isbngenerator.IsbnGenerator"
/>

</blueprint>
```



3.2

## **OSGI SERVICES DER SPEZIFIKATION**

- OSGi Core
  - Das eigentliche Framework
  - Die Service-Registry
  - Hilfsklassen wie der Service-Tracker
- OSGi-Compendium
  - Definiert Spezifikationen für zusätzliche Standard-Services
    - Zur Benutzung eines Services muss aber ein Provider-Bundle installiert werden
  - Compendium Services umfassen
    - EventAdmin
    - ConfigAdmin
    - http

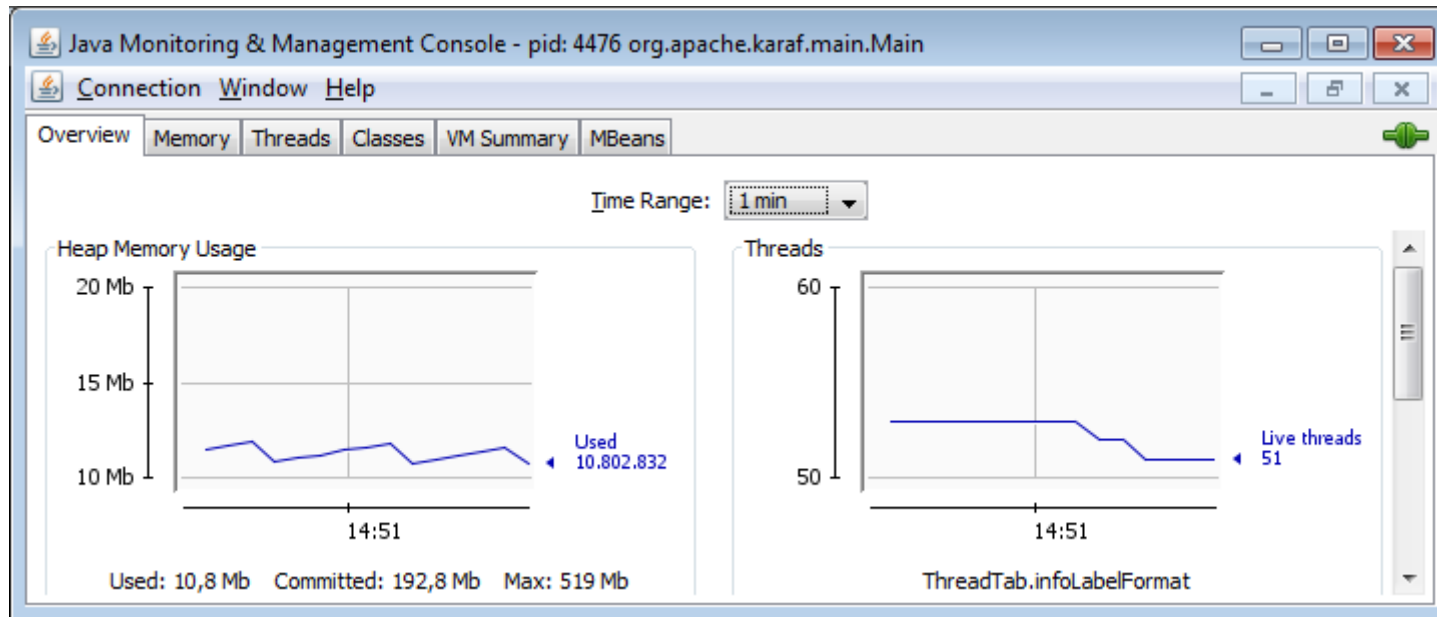
- Services für Enterprise Anwendungen
  - Analog zur Java Enterprise Edition
- Enterprise OSGi umfasst
  - DataSources
  - EntityManager des JPA
  - Transaction Manager
  - Security
  - Blueprint
- Provider implementieren damit Bundles, die den Funktionsumfang eines JEE-Applikationsservers realisieren können
  - Apache Aries
  - Apache Karaf
  - ...



3.3

## **RUNTIME UND INFRASTRUCTURE**

- OSGi-Runtimes erweitern eine Java Virtual Machine nur minimal
  - Eine laufende Apache Karaf-Instanz



- Abgespeckte Runtimes besitzen keine Konsole, kein Logging etc.
  - Alle diese Features werden durch Bundles zur Verfügung gestellt

- OSGi-Konsole
  - Installation von Bundles
  - Lifecycle
  - Abhängigkeiten (Requirements) von Bundles
  - Zur Verfügung gestellte Services
- OSGi-Repositories
  - Analog zu einem Maven-Repository werden OSGi-Bundles zentral abgelegt
  - Features definieren Gruppen von Bundles, die zusammen installiert werden müssen
    - Angabe der benötigten Bundles inklusive Version
  - OSGi-Runtimes können Features installieren
    - Eventuell mit automatischem Update

- Administration
  - Web Console
  - Scripting
- Überwachung und Monitoring
  - Hier bietet sich eine JMX-basierte Lösung an

