

Software Komponenten

Szyperski

Eine Softwarekomponente ist ein wiederverwendbares Stück Software

- ▶ hat eine gut spezifizierte Schnittstelle
- ▶ kann in nicht-vorhergeplanten Kontexten eingesetzt werden
- ▶ ist eine eigenständig vermarktbare Einheit

Beispiele

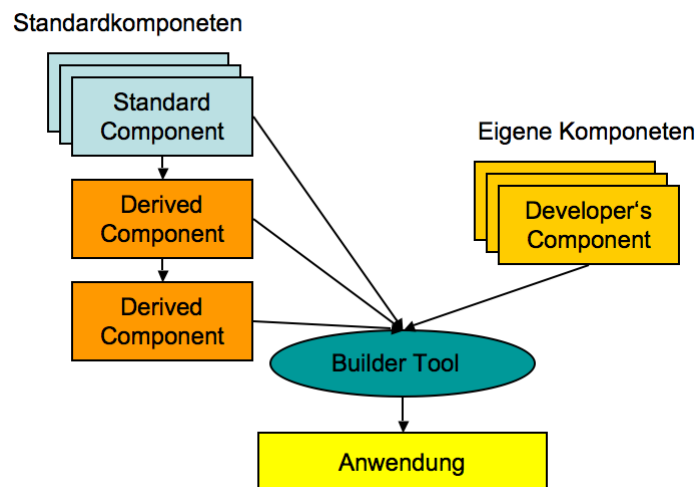
- ▶ Dialog einer graphischen Oberfläche
- ▶ Module einer Textverarbeitung (z.B. Rechtschreibkorrektur)
- ▶ Modul welches ein Gerät integriert (z.B. Drucker)

Klassen == Komponenten ?

Software Komponenten

- ▶ Standard-Konventionen für die Schnittstellen von Software-Komponenten
 - ▶ Ziele: Einfache Verwendbarkeit + Wiederverwendbarkeit
- ▶ Software-Komponenten sind selbst-beschreibend
 - ▶ Automatische Analyse von Schnittstelle und Eigenschaften möglich
 - ▶ Introspection / Reflection - Mechanismen
 - ▶ Interface-Repository
 - ▶ Namens-Konventionen
 - ▶ Zusammenstellung zu komplexen Anwendungen mit grafischen Werkzeugen (Builder-Tools)
 - ▶ Software-Baukasten

Software Komponenten



Software Komponenten

- ▶ Richtlinien für Software-Komponenten
 - ▶ Namenskonventionen
 - ▶ Repositories, Introspection-Mechanismus
 - ▶ Schnittstellen-Semantik
 - ▶ Programmiermethodik
- ▶ Beispiele Komponentenarchitekturen für Java
 - ▶ **JavaBeans**
 - ▶ EJB – Enterprise Java Beans
 - ▶ Spring
 - ▶ Jini
 - ▶ **OSGi**
 - ▶ CORBA Component Model

Java im Kontext von Komponentensystemen

Ziele der Sprache

- ▶ Lösung der verbreiteten Probleme bei der Entwicklung und Verteilung von Software
 - ▶ verschiedene Betriebssysteme (Unix, Windows, MacOS, ?)
 - ▶ verschiedene Hardware-Architekturen
- ▶ Java: Sprache und Ausführungsumgebung für sichere, schnelle und sehr robuste Anwendungen auf verschiedenen Plattformen in heterogenen verteilten Netzen

Wesentliche Eigenschaften für Komponenten

- ▶ objektorientiert
- ▶ Polymorphismus bei Methodenaufrufen
- ▶ sehr flexibel (dynamisches Laden und Binden)
 - ▶ Kann auch problematisch sein! ⇒ OSGi
- ▶ Reflection/Introspection Mechanismus

Java Beans

- ▶ Java Beans ist eine Schnittstellen-Spezifikation für wiederverwendbarer Software-Komponenten in Java
 - ▶ definiert Java-Komponenten
 - ▶ und wie sie zusammenarbeiten
- ▶ Eine Bean ist eine beliebige Java-Klasse, die den Java Beans-Konventionen folgt

Die *offizielle* Definition von Sun:

A Java Bean is a reusable software component that can be visually manipulated in builder tools.

Architektur

- ▶ Eigenschaften (Properties)
 - ▶ initiale Einstellung von Eigenschaften (Zustand) einer Bean
- ▶ Methoden Ereignisse (Events)
 - ▶ die Verknüpfungspunkte für Beans
- ▶ Adapter
 - ▶ zur Schnittstellenanpassung, wenn Beans nicht einfach zusammenpassen
- ▶ Introspection
 - ▶ statt eines Interface-Repositories: einfach in die Bean reinschauen was sie kann

Beispiele

- ▶ Grafische Beans einer Benutzeroberfläche
 - ▶ Knöpfe, Regler, Textfenster
 - ▶ HTML rendering Bean
- ▶ Unsichtbare Beans
 - ▶ Datenbank-Schnittstelle
 - ▶ Timer
 - ▶ löst Ereignisse in bestimmten Abständen aus
 - ▶ kann komplexe Zeit-Logik kapseln
- ▶ Softwarekomponenten, die Geräte/Geräteteile repräsentieren
 - ▶ Schalter
 - ▶ Sensoren, Aktoren
 - ▶ Video-Rekorder
 - ▶ Kaffeemaschine

Properties

- ▶ Beschreiben Eigenschaften von Komponenten
- ▶ Jede Property hat
 - ▶ einen **Namen** – symbolische, aussagekräftige Beschreibung der Eigenschaft (Color, Font, usw.)
`private Color color;` (Instanzvariable der Bean-Klasse)
 - ▶ einen **Typ** – Java-Klasse, kapselt den Wert
 - ▶ **Constraints** – optionale Einschränkungen (z.B. read-only)
- ▶ Namens-Konventionen für Methoden zum Zugriff (Methoden der Bean- Klasse)
 - ▶ get-Methode zum Lesen
`public Color getColor() return color;`
 - ▶ set-Methode für Modifikationen

```
public void setColor(Color newColor) {
    color = newColor;
    repaint();
}
```

Properties

- ▶ Simple properties
 - ▶ repräsentieren einen einzelnen Wert, Zugriff mit set/get-Methoden
- ▶ Indexed properties
 - ▶ repräsentieren ein Feld, set/get-Methoden haben einen index-Parameter
- ▶ Bound properties
 - ▶ informieren andere Objekte über Zustandsänderungen (PropertyChange event)
- ▶ Constrained properties
 - ▶ nicht erlaubte Zustandsänderungen können zurückgewiesen werden

Events

Umsetzung des Source-Listener Design-Patterns

- ▶ Source-Objekt informiert Listener über Zustandsänderungen
 - ▶ EventSource hat Methoden für Listener, um sich zu registrieren/abzumelden
 - ▶ Beispiel:


```
public void addTimerListener(TimerListener l)
public void removeTimerListener(TimerListener l)
```

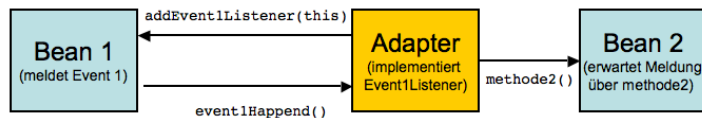
 - ▶ add/remove
 - ▶ gleicher Name Methode wie Parameter
 - ▶ Parameter muss EventListener Schnittstelle implementieren
- ▶ EventObject
 - ▶ kapselt Informationen ber Ereignis
- ▶ EventListener
 - ▶ Klasse die EventListener-Interface implementiert (Subtyp von `java.util.EventListener`)

Events

- ▶ PropertyChangeSupport
 - ▶ automatisches Versenden von notification-Events, immer wenn sich der Wert einer Property ändert
- ▶ VetoableChangeSupport
 - ▶ ermöglicht das Zurückweisen von Property-Werten, die außerhalb des gültigen Bereichs liegen

Adaptor

- ▶ Anpassung von Methoden einer Bean an Ereignisse einer anderen
 - ▶ Adapter implementiert passendes Event-Listener Interface
 - ▶ Adapter registriert sich für Events von Bean 1
 - ▶ Event 1 wird ausgelöst : Bean 1 ruft Methode die im event-Interface festgelegte Methode `event1Happend()` bei allen registrierten Event-Listener-Objekten auf
 - ▶ `event1Happend()` im Adapter ruft `method2()` an Bean 2 auf
- ▶ Adapter-Objekt kann Daten evtl. manipulieren (z. B. umrechnen)
- ▶ Einfache Adapter können automatisch erzeugt werden



Introspection

- ▶ Automatische Analyse von Eigenschaften einer Bean
- ▶ Ab Java 1.1 Reflection API
 - ▶ Analyse von Java-Klassen zur Laufzeit
 - ▶ Instanzvariablen: Name & Typ
 - ▶ Methoden: Name, Parameter, Ergebnis-Typ
- ▶ JavaBeans Namens-Konventionen
 - ▶ get/set Methoden für Properties
 - ▶ add/remove Methoden für Events
 - ▶ andere Methoden für normale Methoden
- ▶ Alternative: Information in einer BeanInfo-Klasse (ähnlich wie ein Interface-Repository)
 - ▶ Entwickler kann Properties und Events explizit spezifizieren

Zusammenfassung

- ▶ Beans sind zusammensteckbare Software-Bausteine
 - ▶ Zustand = Properties
 - ▶ Eingänge = Methoden
 - ▶ Ausgänge = Events
 - ▶ wenn Stecker-Buchse nicht passen: Adapter
- ▶ Probleme Defizite
- ▶ Zusammenstecken einfacher Komponenten zu komplexen Komponenten
 - ▶ hierarchische Beans
- ▶ Ankoppeln neuer Software-Komponenten an laufende Anwendungen
- ▶ Software-Komponenten im verteilten System
- ▶ Aktualisierung von laufenden Komponenten
- ▶ Keine Unterstützung für die Verwendung von Code verschiedener Hersteller/Entwickler

OSGi – Open Services Gateway initiative

- ▶ OSGi ist ein dynamisches Modulsystem für Java
- ▶ Entwicklung und Standardisierung durch ein Konsortium
 - ▶ Z.B. IBM, Siemens, Nokia, SAP, etc.
- ▶ Ursprünglich für Java-basierte eingebettete Systeme gedacht
- ▶ Aktuell Entwicklung zum Standard zur Modularisierung von komplexen Java-Anwendungen
 - ▶ Z. B. Eclipse und Websphere
- ▶ Spezifikation unter <http://www.osgi.org/>

OSGi website

OSGi technology provides a service-oriented, component-based environment for developers and offers standardized ways to manage the software lifecycle. These capabilities greatly increase the value of a wide range of computers and devices that use the Java platform.

Klassen Laden in Java

- ▶ Alle Klassen werden durch einen Class Loader in die Java Laufzeitumgebung geladen
- ▶ Komplexe Applikationen verfügen oft über mehrere Class Loader
 - ▶ Integration von Programmcode bzw. Komponenten verschiedener Firmen und Entwickler
 - ▶ Laden von Klassen über das Netzwerk bzw. zur Integration von neuen Funktionen

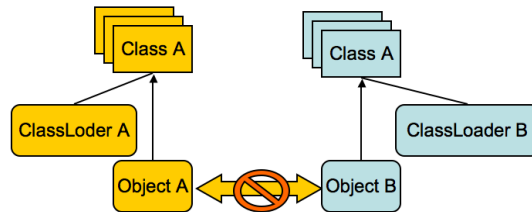
Eigene ClassLoader Implementierungen

- ▶ Ableitung von `java.lang.ClassLoader`
- ▶ `loadClass()` lädt Klassen bleibt aber erhalten

```
public synchronized Class loadClass(String name) throws ClassNotFoundException {
    Class c = findLoadedClass(name); // Already loaded this class?
    if (c == null) { // Can the class be loaded by a parent?
        try {
            if (parent == null) {
                c = VMClassLoader.loadClass(name, resolve);
                if (c != null)
                    return c;
            } else {
                return parent.loadClass(name, resolve);
            }
        } catch (ClassNotFoundException e) { }
        // Still not found, we have to do it ourself.
        c = findClass(name);
    }
    resolveClass(c);
    return c;
}
```

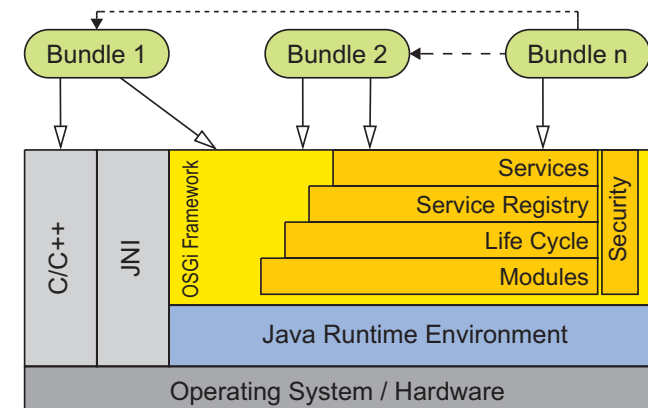
- ▶ `findClass()` wird implementiert
 - ▶ Legt fest welche Klassen geladen werden

Probleme bei der Verwendung von mehreren Class Loader



- ▶ Klassen dürfen innerhalb eines Anwendungskontextes nur einmal geladen werden
 - ▶ ClassLoader weisen jeder Klasse eine Id zu die bei gleichem Namen zur `ClassCastException` führt
 - ▶ Wo werden statische Variablen verwaltet?
- ▶ Ziel von OSGi:
 - ▶ Sichere Verwendung von Java Modulen unterschiedlicher Hersteller und Versionen
 - ▶ Kontrollierte Verwendung von Class Loader

Framework



Framework

Security Layer

- ▶ Ergänzung zur Java 2 Security Architektur
- ▶ Code Authentifizierung durch Signaturen (Herkunft, Unterzeichner)

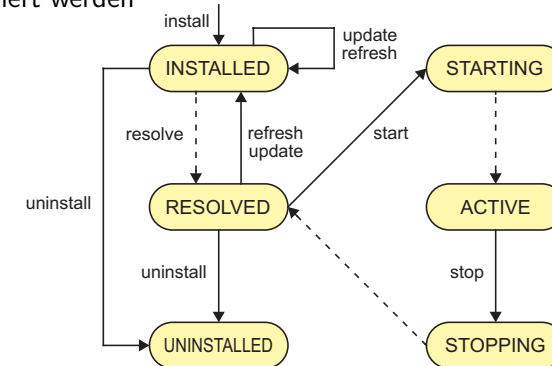
Module Layer

- ▶ Einheit der Modularisierung: **Bundle**
- ▶ Bundles werden als Java ARchive (JAR) Datei ausgeliefert
 - ▶ Enthlt alle nötigen Ressourcen (evtl. inkl. weiterer JARs)
 - ▶ Bundlebeschreibung und Abhängigkeiten als Manifest Datei (META-INF/MANIFEST.MF)

Framework

Life Cycle Layer

- ▶ Verwaltung des Lebenszyklus von Bundles
- ▶ Bundles können zur Laufzeit installiert, gestartet, gestoppt und deinstalliert werden



Framework

Life Cycle Layer

- ▶ Bundle - Repräsentiert ein installiertes Bundle
- ▶ Bundle Context - Ausführungskontext eines Bundles
 - ▶ Zugriff auf Informationen des Framework und die Service Registry
 - ▶ Installieren anderer Bundles
- ▶ Bundle Activator
 - ▶ Starten und Stoppen des Bundles

Service Layer

- ▶ Service Registry zum Registrieren, Finden und Binden von Diensten
- ▶ Dienste sind Java-Objekte, registriert mit ihrem/ihren Interface(s)
- ▶ Bundles können Dienste registrieren, suchen und ihren Zustand abfragen

Manifest

Manifest enthält alle Information zur Integration eines Bundels

- ▶ Activator Klasse ist der Einstiegspunkt
- ▶ Jedes Bundle hat eigenen Classloader und Class Space
 - ▶ Classpath ergibt sich aus: RTE + Framwork + Bundle Classpath
 - ▶ Bundle-Classpath - Intra-Bundle Classpath für eingebettete JARs
 - ▶ Import-Package - Importierte Pakete von anderen Bundles
 - ▶ Require-Bundle - Importiert alle Pakete des angegebenen Bundles
- ▶ Export-Package - Pakete die von diesem Bundle exportiert werden

Auflösen der Abhängigkeiten zwischen Bundles

- ▶ Auflösung aller im Manifest beschriebenen Abhängigkeiten
 - ▶ Importierte und exportierte Pakete
 - ▶ Benötigte Bundles (import aller Pakete)
- ▶ Fragmente (Teil eines größeren logischen Bundles)
- ▶ Ein Bundle ist *resolved* wenn
 - ▶ Alle Importe verdrahtet sind
 - ▶ Alle benötigten Pakete verfügbar sind und ihre Exporte verdrahtet
- ▶ Nach dem Auflösen kann Bundle geladen und ausgeführt werden

Zusammenfassung

- ▶ Einfaches Komponentensystem
- ▶ Löst *deployment*-Probleme
 - ▶ Es können Komponenten aus verschiedenen Quellen miteinander kombiniert werden
- ▶ Entwickelt sich zum de facto Standard für Java
- ▶ Offene Fragen
 - ▶ Verteiltes Interaktionsmodell
 - ▶ Dynamisches Laden von entfernten Code
 - ▶ Bisher zentrale proprietäre Lösungen

Auflösen der Abhängigkeiten zwischen Bundles

- ▶ Jedes Bundle besitzt einen eigenen Classloader
 - ▶ und verfügt damit über einen eigenen Namensraum

Suchreihenfolge

