Modulare Webanwendungen mit Java
Theorie und Praxis

Jan Paul Buchwald

18. Java Forum Stuttgart, 09.07.2015

Agenda

Modularisierung

Motivation
Vor- und Nachteile
Definition Modul

Theorie

Module im Java Kontext
Modularisierungsansätze für Java (Web) Anwendungen

Praxis

Modularity Maturity Model
Patterns, Best Practices
Zuschnitt, Abhängigkeiten

Monolithen

Build Monolith
Deployment Monolith
Runtime Monolith

Quelle: http://commons.wikimedia.org/wiki/File:Uluru_(Helicopter_view)-crop.jpg

Modulare Entwicklung

Quelle: https://commons.wikimedia.org/wiki/File:Lego_Color_Bricks.jpg

Zusammensetzung standardisierter, wiederverwendbarer Software-Bausteine ergibt Gesamt-Lösung

Warum Modularisierung?

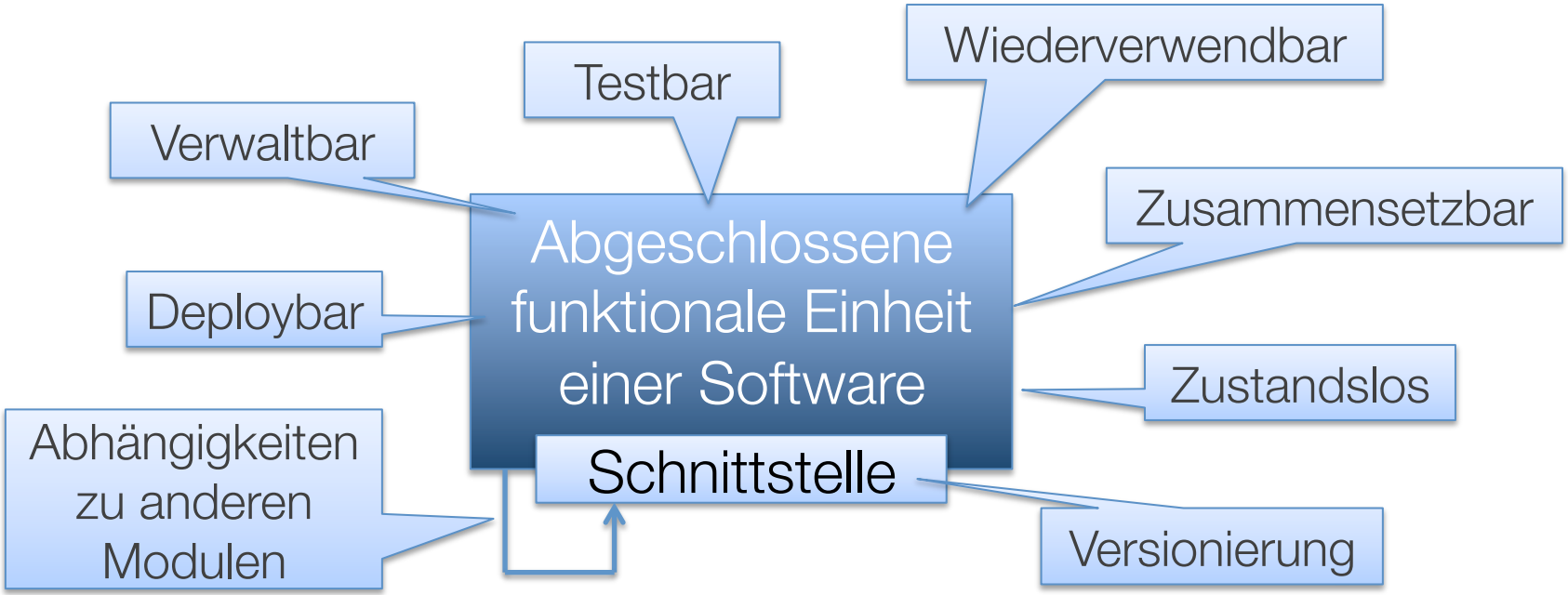
Bessere Wartbarkeit
- Verständlichkeit
- Lokalität

Wiederverwendbarkeit
- Kombinierbarkeit

Dauer und Flexibilität
- Parallele Entwicklung
- Austauschbarkeit

Höhere Systemkomplexität
- Overhead durch Entkopplung
- Verwaltung von Abhängigkeiten
- Integrationsaufwand

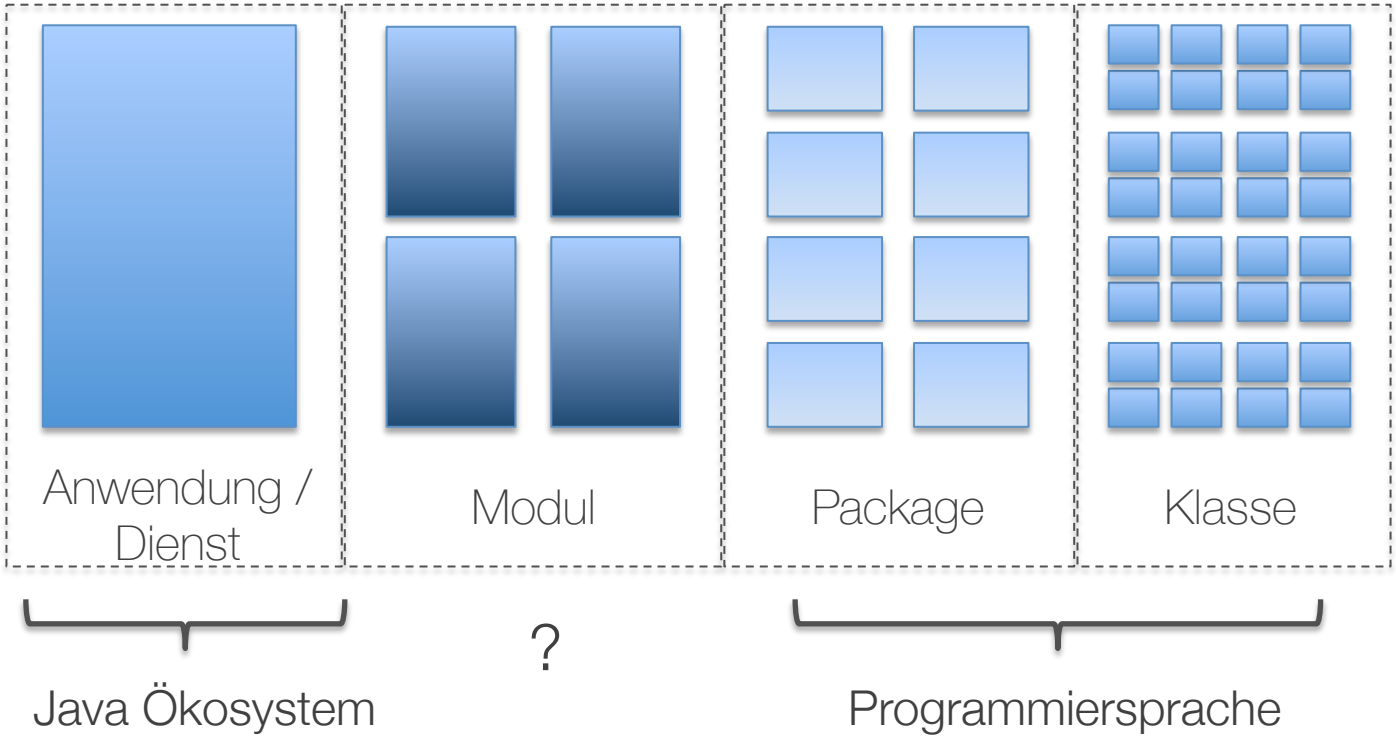
Definition Modul



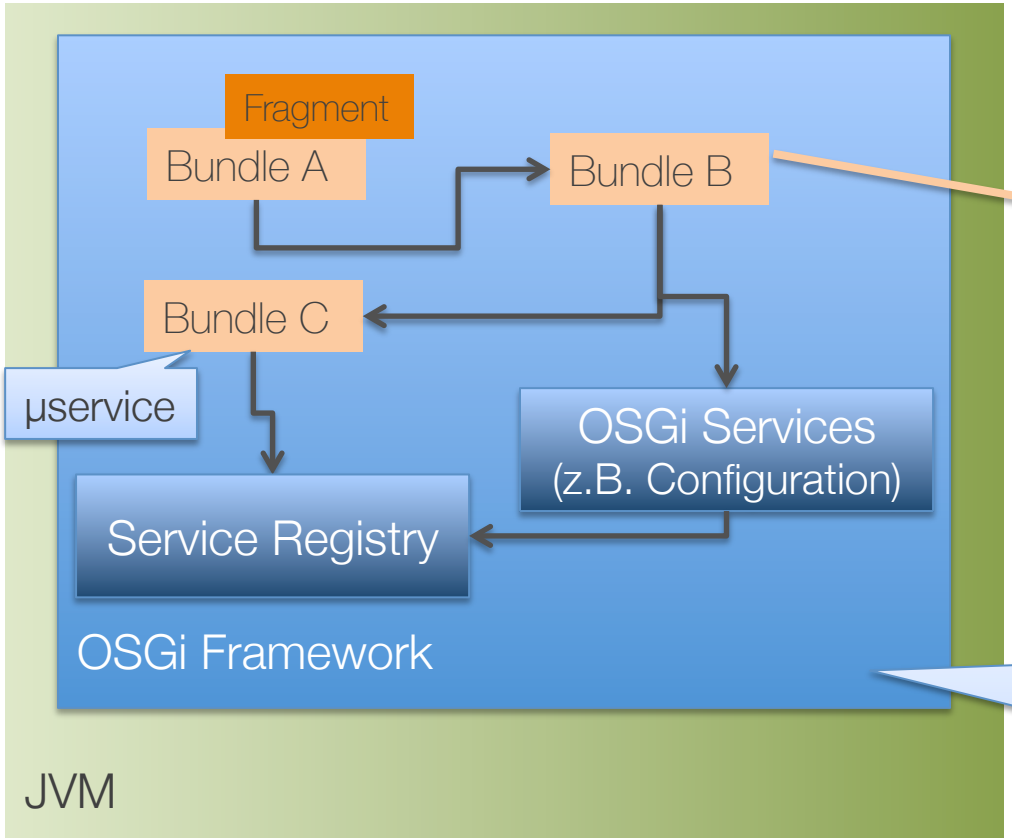
Programmiermodell

Laufzeitmodell

Module im Java-Kontext



Open Services Gateway initiative - OSGi



Jar File mit `META-INF/MANIFEST.MF`

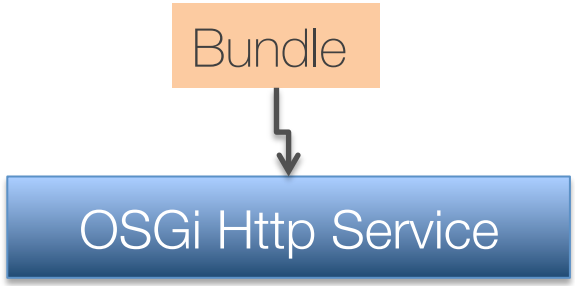
```
Bundle-Name: JFS
Bundle-SymbolicName: jfs
Bundle-Description: A Java Forum bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0
Bundle-Activator: jfs.Activator
Export-Package: jfs.helloworld
Import-Package: jfs.other;version="1.0“
```

Neuere OSGi Versionen:
Declarative Services Annotationen
`@Component, @Activate, @Reference`

- Kommerziell, z.B. ProSyst
- Open Source, z.B. Equinox, Apache Felix, Knopflerfish

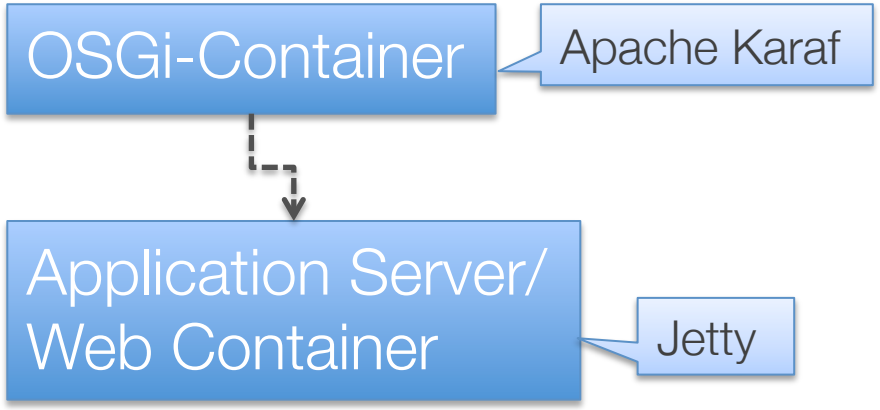
OSGi und Web

Web Application Bundle (WAB)



OSGi-fähiger Application Server

WebSphere, GlassFish, Weblogic, Wildfly



Quo vadis OSGi?

2000: OSGi R1
2001: OSGi R2
2003: OSGI R3

Primär für eingebettete Systeme (Telematik, Home Automation, ...)

2003: eclipse

2005: OSGi R4

2012: OSGi R5

2015: OSGi enRoute

Internet of Things

„OSGi nicht beim Mainstream Java-Entwickler angekommen“

Programmiermodell schränkt ein
- Package-Orientierung
- Service Modell

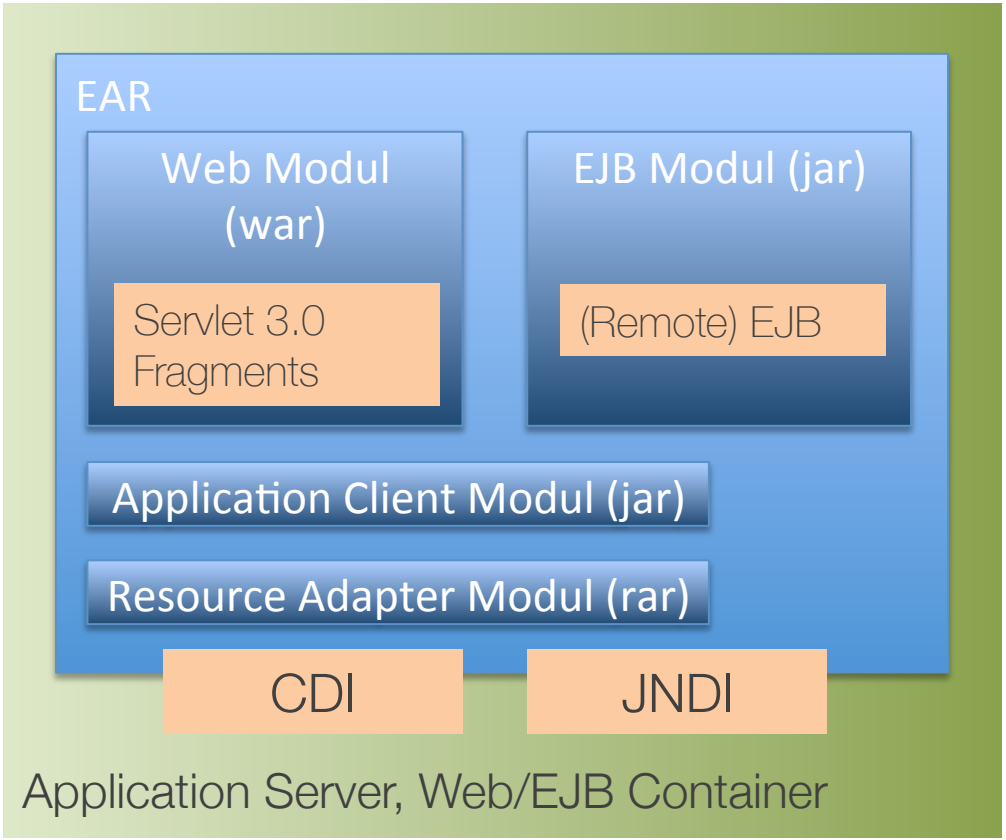
Runtime / Container mit eigenem Life Cycle und Deployment-Modell

Hohe Einstiegshürde, wenig vorhandene Komponenten und Beispiele

Überschneidungen mit Java EE

In der Praxis vor allem im Web-Bereich eher selten eingesetzt

Java EE Artefakte als Module



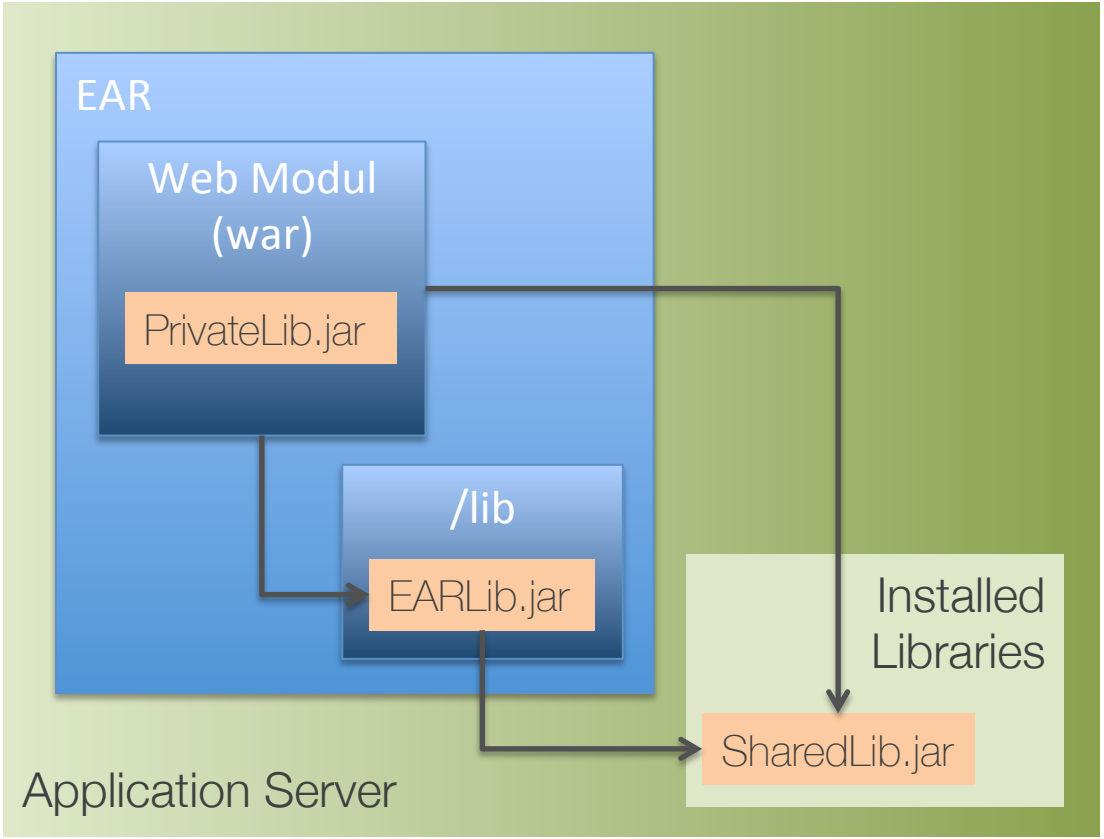
Java EE Module sehr grob-granular

Weitere Bausteine uneinheitlich, oft Container-abhängig

Fest vorgegebenes, unflexibles Programmiermodell

Wiederverwendbare Bausteine in der Praxis oft nicht realistisch

Java EE Libraries als Module



Ausnahme:
WildFly / JBoss Modules

Keine saubere Modulabgrenzung

Versions- und Classloader-Konflikte

Nur pragmatische Notlösung in der Praxis

Modularisierung im JCP

1999: JSR 8

Open Services Gateway Specification
Status: Withdrawn → OSGi

2005: JSR 277

Java Module System
Format und Repository für Menge von Java Code und Ressourcen
Status: Dormant

2006: JSR 291

Dynamic Component Support for Java SE
Übernahme Teile der OSGi Spec in Mainstream Java
Status: Final seit 2007

2006: JSR 294

Improved Modularity Support in the Java Programming Language
`superpackages` Status: Dormant

2008: Project Jigsaw
...
2014: JSR 376

Java Platform Module System
„Low-Level“ Modulsystem für Modularisierung des JDK
Status: ~~Java 7~~ → ~~Java 8~~ → Java 9 (!?)

Project Jigsaw in Java 9

JEP 201: Modular Source Code
Modulares Ordner-Schema und Build System für JDK

Closed/Delivered

JEP 220: Modular Run-Time Images
Restrukturierung JDK und JRE Images bzgl. Modulen
Neues URI Schema für Artefakte in Images

Integrated

Minimale Annahmen zum Modulsystem: OSGi-ähnlich, module-graph (globale XML-Datei)

JEP 200: The Modular JDK
Auftrennung des JDK in Module, Kombinierbarkeit zur Compile-, Build-, Installations- und Laufzeit

Candidate

JSR 376: Java Platform Module System
Bisher nur „Goals & Requirements“, Referenzen auf frühere „explorative Phase“ von Jigsaw

Modularisierung über Frameworks

Struktur / wiederverwendbare Bausteine werden von Framework bestimmt, z.B.

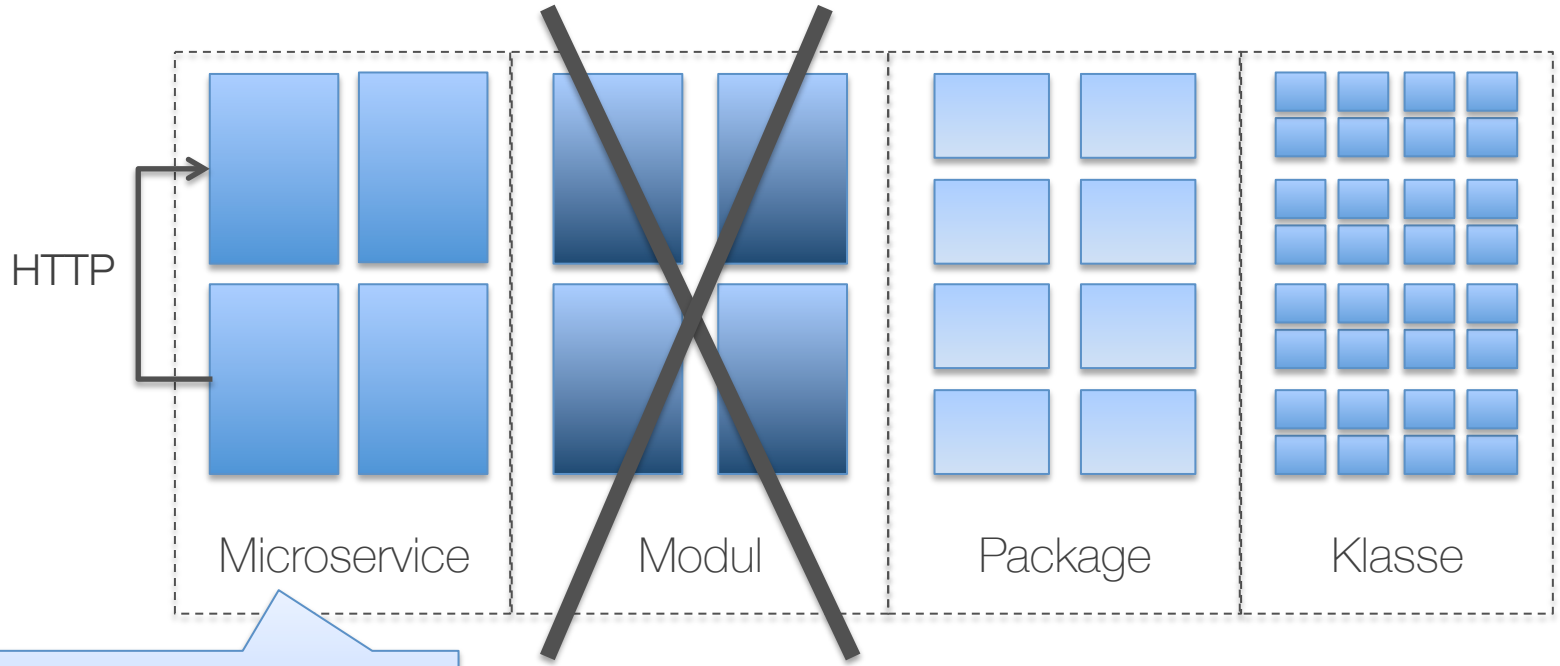
Spring Beans / Container
Spring MVC / Web Flow
Spring AOP
Spring Dynamic Modules (OSGi)

Wisdom Framework

Teilweise OSGi Unterbau

Starke Bindung an konkrete Technologie

Microservices



Feingranularer Service
Eigener Prozess

Microservice = Modul?

Microservice setzt zugehörige Architektur voraus

Herausforderungen:
- Verwaltung von Abhängigkeiten
- Sicherstellung von Schnittstellenversionen

Wiederverwendbarkeit und Kombinierbarkeit nur als kompletter Service

Microservice erfüllt einige der elementaren Anforderungen an Module im Java-Kontext nicht

Theorie und Praxis

Alle betrachteten Modularisierungsansätze sind unvollständig oder haben klare Schwächen

In der Praxis sind deshalb pragmatische Lösungen gefragt

- Java / Java EE Bordmittel und gängige Tools mit Patterns und Best Practices
- Nutzung einzelner Elemente und Prinzipien der beschriebenen Ansätze wo sinnvoll

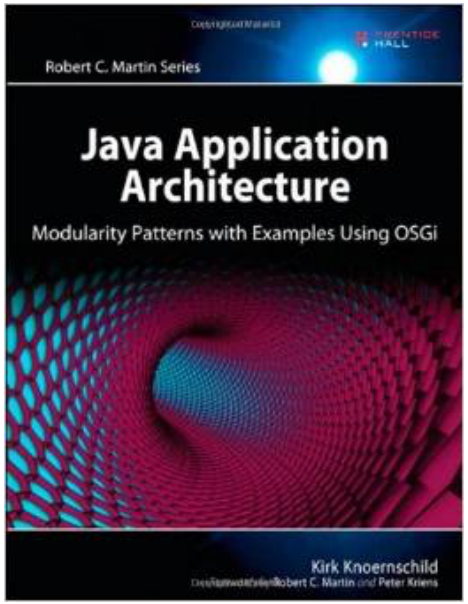
Modularity Maturity Model *

Level	Bezeichnung	Beschreibung
1	Ad Hoc	Keine Modularisierung, globaler Classpath
2	Module	Formale Modul-Identitäten entkoppelt von Artefakt
3	Modularität	Formale Schnittstellen zwischen Modulen
4	Loose Coupling	Trennung Interface und Implementierung: Services, semantische Versionierung von Abhängigkeiten
5	Befugnisübergabe	Zentrale Modul-Repositories („Baukasten“), optional mit Collaboration- und Governance-Funktionen
6	Dynamik	Dynamischer Lebenszyklus von Modulen, Betriebs-unterstützung für Hinzufügen, Entfernen, Ersetzen
7	Peter Kriens	

** Graham Charters, IBM, OSGi Community Event 2011*

Patterns und Best Practices

Entscheidend: Einstellung und Wissen der Entwickler



Erweiterte SOLID Principles

Modularity Patterns

Technologie-unabhängig,
Beispiele mit OSGi

Modularity Patterns

Base Patterns
Manage Relationships
Module Reuse
Cohesive Modules

Usability Patterns
Published Interface
External Configuration
Default Implementation
Module Facade

Dependency Patterns
Acyclic Relationships
Levelize Modules
Physical Layers
Container Independence
Independent Deployment

Extensibility Patterns
Abstract Modules
Implementation Factory
Separate Abstractions

Utility Patterns
Colocate Exceptions
Levelize Build
Test Module

Modularity Patterns Beispiele

Physical Layers
Module relationships should not violate the conceptual layers.

External Configuration
Modules should be externally configurable.

Separate Abstractions
Place abstractions and the classes that implement them in separate modules.

Anti-Patterns

Über-generalisierte Schnittschnelle

Falsche / unnötige Abstraktion

NeverReusedReusableModule

Zu feingranularer Modulschnitt

Zu große Module (kleine Monolithen)

Modul-Zuschnitt und Abhängigkeiten

Domänenmodell

Architekturschicht

Kunden-verwaltung

Zahlungs-abwicklung

Einkaufs-steuerung

...

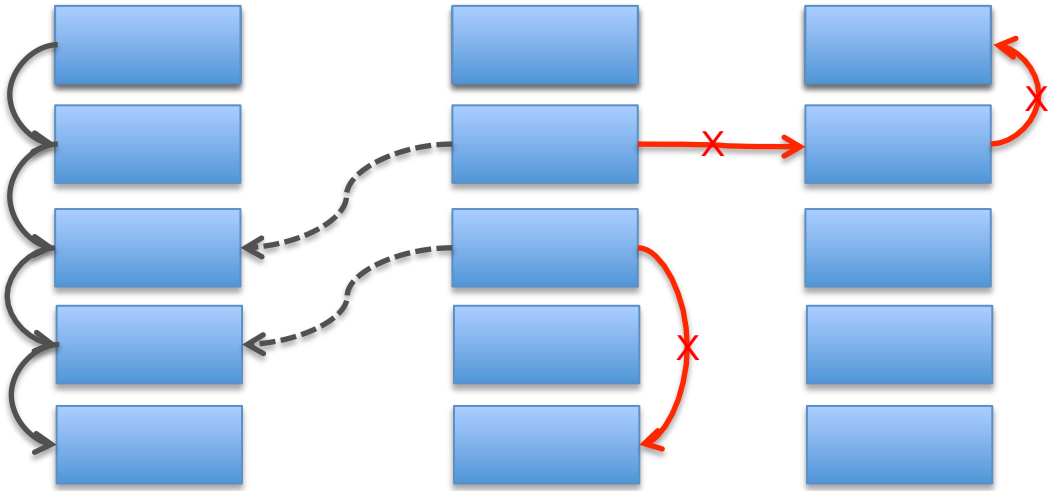
Präsentation

Steuerung

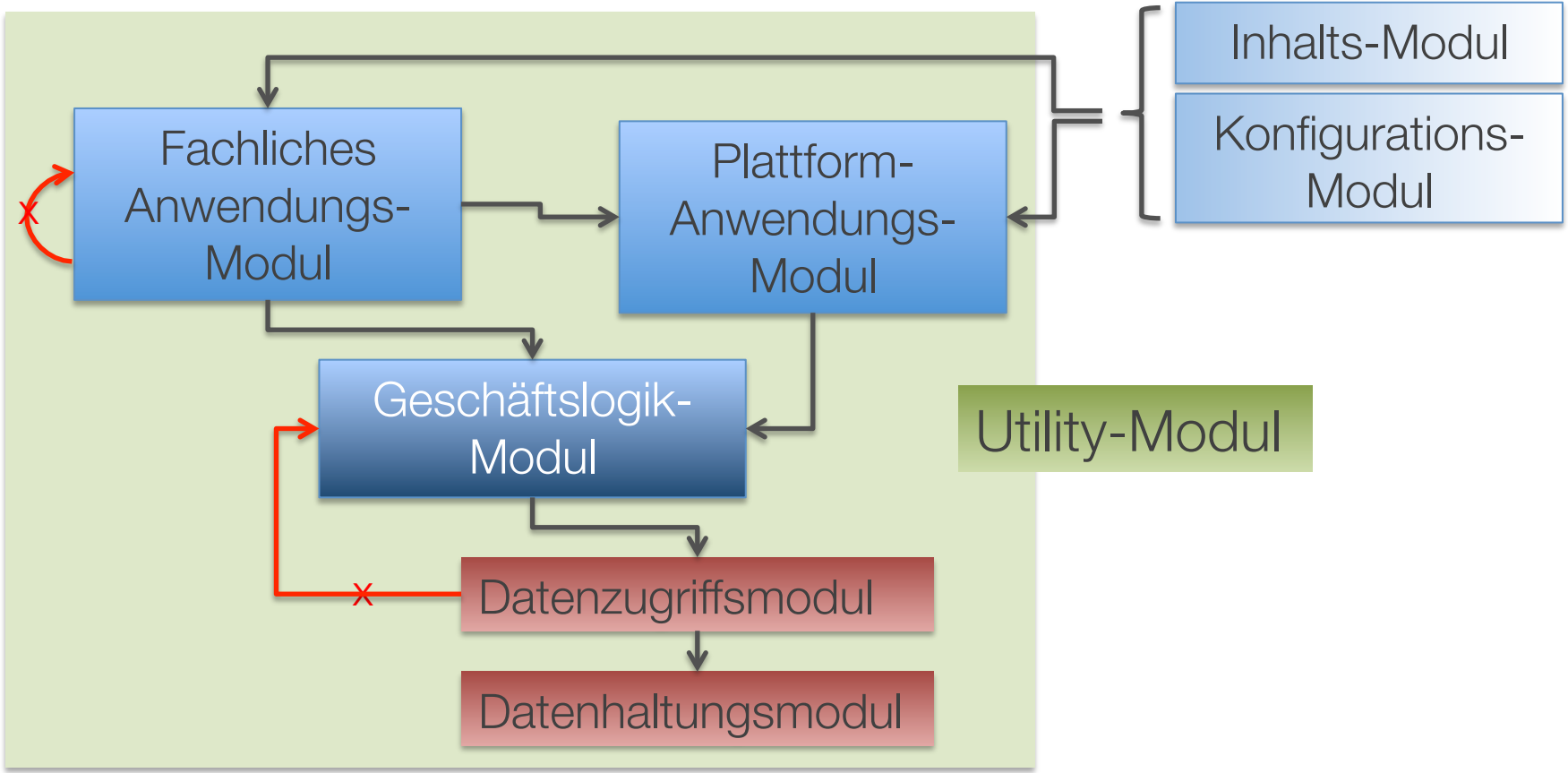
Geschäftslogik

Datenzugriff

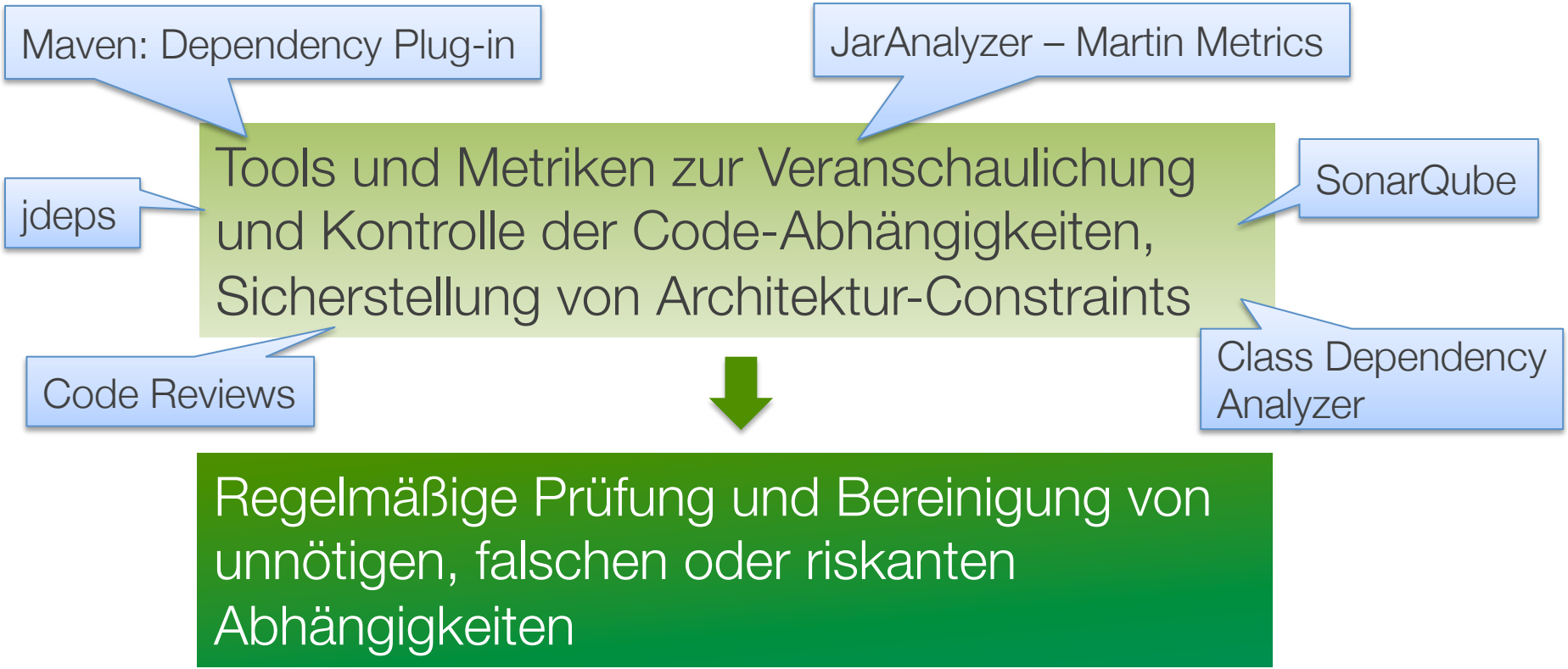
Datenhaltung



Modul-Typen und Abhängigkeiten



Überprüfung Abhängigkeiten



Fazit

Rein technisch ist die Herausforderung der Modularisierung für Java Web nicht zu lösen

- Kein allgemein verwendbares Modulsystem
- Technologie kann nicht alle Constraints sicherstellen

Patterns und Best Practices für bestehende Lösungen und Tools helfen

Weitere Entwicklung von OSGi und Java Modul System (JSR 376) bleibt spannend

Vielen Dank!

?!?

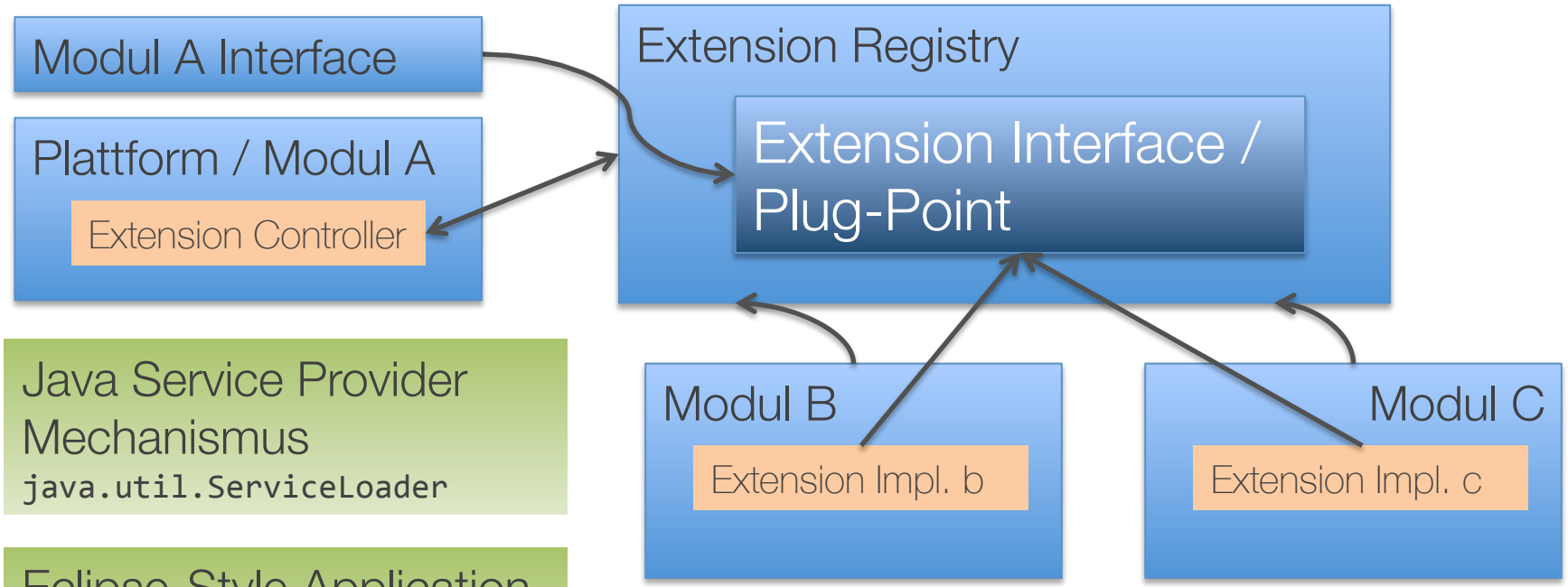
Jan Paul Buchwald

http://www.j-pb.com

@jpspec

Backup

Extension-Modell



Java Service Provider Mechanismus

`java.util.ServiceLoader`

Eclipse-Style Application Extension Registry in WebSphere

Problematisch: Classloading!