



Lightweight J2EE

Die unerträgliche Leichtigkeit des Seins

Verfasser: Josef Adersberger, Dipl. Inf. (FH)
Datum: 15.3.2005

© 2005 by QAware

■ Dienstleistungen

• QAssess

- Qualitätssicherung von Architektur und Code (konstruktiv, analytisch)
- Komplett toolgestützte Methodik mit zeitlich festgelegten Deliveries
- Hotspots, Ist-Situation, Soll-Situation, Roadmap

• QArchitecture

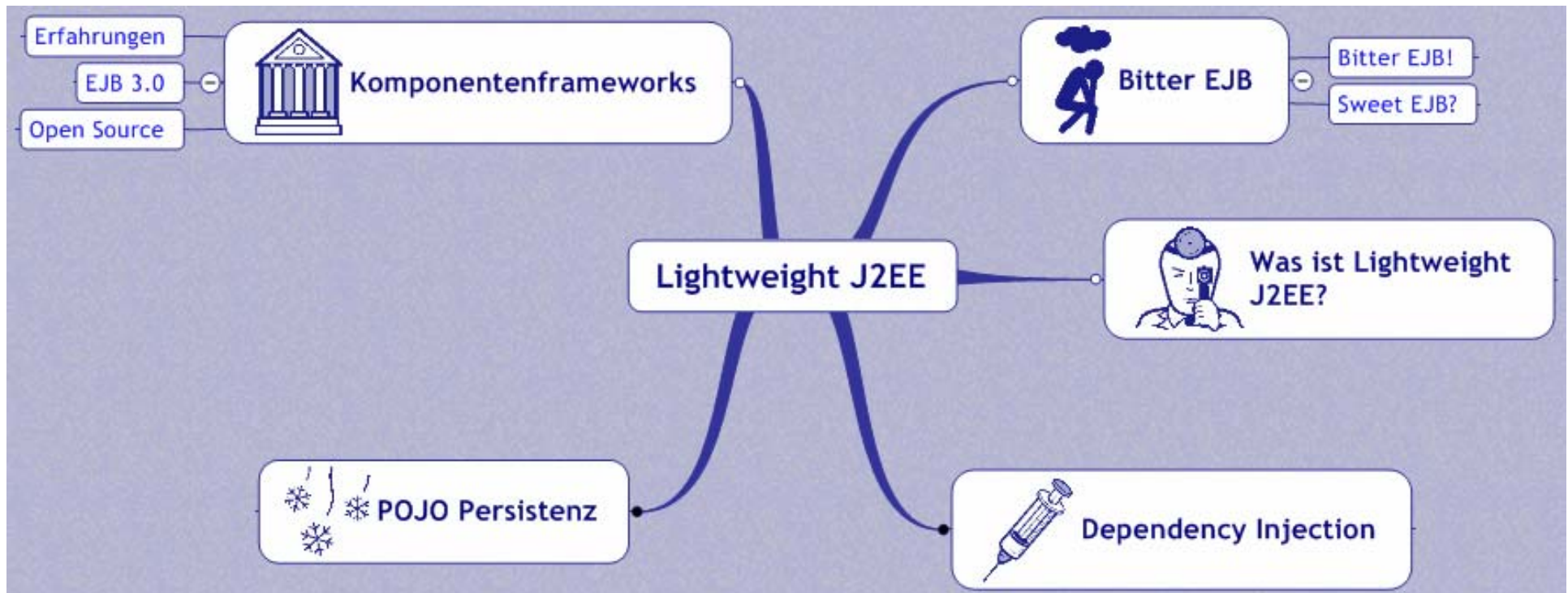
- Architekturentwicklung, Architekturberatung, Architekturgutachten
- Konzepte und Frameworks für *Lightweight J2EE* und Regelengines
- Migrationsberatung, Migrationsdurchführung (sanfte Migrationen)

• QApplication

- Produkt: Generic Rule-based Data Capture Engine

■ Pool an innovativen Know-How

- Lightweight J2EE (Spring, Hibernate, HiveMind, JDO 2.0, EJB 3.0)
- Presentation-Layer (Swing, Flex, Laszlo, JSF, Struts, UTC/SA, WebWork)
- Software-Entwicklungs-Umgebungen, Software-Entwicklungs-Prozesse



- **Integriertes Verteilungsmodell auf Objektebene**
 - Entitäten sind keine verteilten Objekte! Businesslogik höchstens auf Ebene der fachlichen Komponenten.
 - Falsche Granularität der EJB-Komponenten
 - Fowler:
 - „*first law of distributed objects: don't distribute*“
 - „*sucks like an inverted hurricane*“

- **Monolithischer Behälter**
 - Verhalten nur per XML konfigurierbar (*Deployment Descriptor Hell*)
 - in Teilen nicht standardisiert
 - Fest verankerte Dienstleistungen (Persistenz, Sicherheit, Tx)
 - CMT ist gut
 - CMS deckt Sicherheitsanforderungen nur teilweise ab (was ist mit deskriptiver Sicherheit auf Mengenebene?)
 - CMP war zu keiner Zeit konkurrenzfähig
 - Lange Startup Zeit
 - Objekte laufen nur im Container (schlechte Testbarkeit)

▪ **Komplexes Programmiermodell**

- **Intrusiv:** zu viele Vorgaben (Schnittstellen, Basisklassen, Namensregeln)
- **Nicht intuitiv:** gegen das gewohnte Java Objekt-Protokoll (*isIdentical()*, *Referenzen*)
- Auch einfache Dinge sind aufwändig

▪ **Viele Fehlermöglichkeiten**

- Umfassendes technisches Know-How nötig
- Seiteneffekte durch Verteilung, Pooling und Feature-Race der Appserver Hersteller

▪ **Veraltete Design-Entscheidungen**

- Glue-Code Generator statt Dynamic Proxies
- Objekt-Pooling

▪ **Ungute Abhängigkeiten in Anwendungslogik**

- Geprüfte technische Ausnahmen
- Technisch motivierte Vorgaben und Einschränkungen

▪ **Bruce Tate:**

- I couldn't teach the stuff
- I couldn't test the stuff

- Für viele Projekte ist EJB technischer Overkill
 - keine verteilte Verarbeitung
 - keine verteilten Transaktionen

- Steigende Anzahl an fehlerhafter Software (Computerwoche)
 - Es ist einfach, schon beim System-Design Fehler zu machen
 - Je früher Fehler gemacht werden, desto teurer ist ihre Behebung

- Framework-Layer / Generatoren / Komponentenmodelle oberhalb der EJB Technologie
 - *BMW*: Entwicklung von CA 2.0 (MDA + Komponentenmodell + Framework)
 - *Open-Source*: Spring, HiveMind, Hibernate

- Sun J2EE Blueprint Patterns als Workarounds
 - + die dazugehörigen Bücher, Artikel, Vorträge und Beratungsleistungen

- Wachsende Zahl der Kritiker
 - Literatur:
 - Rod Johnson: „J2EE without EJB“
 - Bruce Tate „Better, Faster Lighter Java“, „Bitter EJB“
 - „... *and a lot of very capable developers understand it and still don't like it*“
(Rod Johnson, Javapolis 2004)

- EJB 3.0 !?

- Warum hat sich EJB durchgesetzt?
 - Einziger nativer Standard auf seinem Gebiet in Java
 - Gute Unterstützung von hochverteilten Systemen
 - Ausgereifter Sicherheits- und Transaktionsmechanismus
 - Robuste und standardisierte Infrastruktur mit Clustering und Session Failover
 - All-In-One Lösung (kanalisiertes Know-How)
 - Exzellente Tools verfügbar

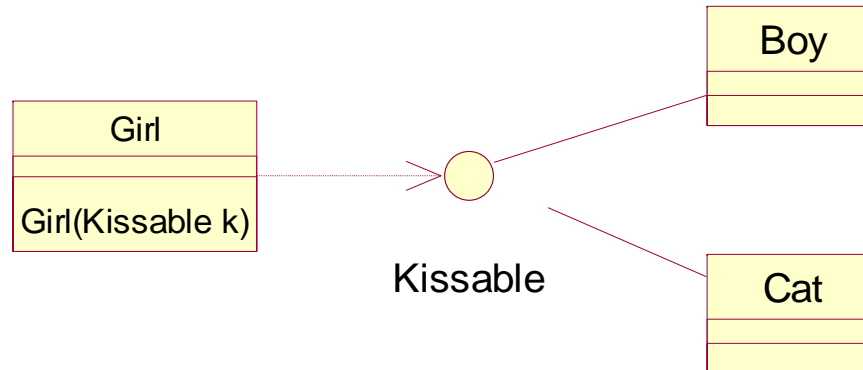
- ➔ Relevant für Groß-Projekte in Konzernen
- ➔ Over-Engineering bei kleineren Dimensionen

- **Zentrale Qualitätsattribute: Einfachheit, Testbarkeit**
 - Einfache API (Single Way, Simple Way, Single Point)
 - Anwendungslogik außerhalb des Containers testbar
 - Berücksichtigung des Pareto Verhältnisses
 - Funktionalität optimiert auf alltägliche Arbeit. Erweiterbarkeit für den Einzelfall
 - Configuration on Exception:
sinnvolle Default-Werte und sinnvolles Default-Verhalten

- **J2EE Umgebung wird entkoppelt**
 - Servlet-Umgebung (z.B. Bea WLX Premium) für Hochverfügbarkeit mit Clustering und Failover
 - J2SE Umgebung für Entwicklung und Unit-Test
 - Deskriptive Umstellung der Umgebung

- **Neue Art von Komponentenframeworks**
 - **POJO –Domänenmodell** und Geschäftslogik; Kein starres Rahmenwerk
 - Komponenten sind aus POJOs und POJIs zusammengesetzt
 - EJB Container wird ersetzt durch **Dependency-Injection Container**
 - Variable **Infrastrukturdienste** (Persistenz, Sicherheit, Transaktionskontrolle)
 - Bereitstellung durch einfache und allgemeine API (z.B. Spring Templates)
 - Bereitstellung auch in Aspekten (AOP)
 - Framework Plug-Ins für div. OS-Frameworks
 - Möglichkeit der **aspektorientierten Programmierung**
 - Verteilung ist kein integraler Bestandteil. Nutzer bestimmt eine evtl. Verteilung.
 - (Anwendungslogik ist zustandslos)
 - Heterogene Systeme (mit EJB) weiterhin möglich. Komfortable Kapselung von EJBs.

- Beispiel: „Hollywood“



```
//Anfrage nach einem „Girl“
Kissable kissable = new Boy();
Girl girl = new Girl(kissable);
girl.kissWhatYouFind();
```

Abhängigkeit wird dem Objekt von Außen übergeben
(*Dependency Injection*)

```
//Girl bekommt ein wenig funktionales Kissable
//zu Testzwecken untergeschoben (jMock)
Mock mock = new Mock(Kissable.class);
mock.expects(atLeastOnce()).method("kiss");
Girl girl = new Girl((Kissable) mock.proxy());
girl.kissWhatYouFind();
```

Mock-Objekte für Abhängigkeiten

▪ Vorteile

- Das Objekt hat keinerlei Möglichkeit, auf die gewählte Implementierung Einfluss zu nehmen. Es kennt wirklich nur die Schnittstelle.
- Explizite Abhängigkeiten
- Das Objekt benötigt keine Hilfsklassen und besitzt keinen *Wiring-Code*
- Das Objekt ist nicht abhängig vom Konfigurationscode

▪ Kraft durch generischen DI-Container

- Verbindung zwischen Schnittstelle und Implementierung in Metadaten (z.B. XML) definiert
- Benötigte Abhängigkeiten (zu Schnittstellen) werden per Default durch Reflection erkannt (Auto-Wiring). Gefahr bei komponentenübergreifenden Abhängigkeiten!
- Dynamische Komponentenbildung durch Steuerung der Dependency Injection (Komponentenmodell liegt in den Metadaten; Prüfung des Modells übernimmt der Container)

▪ Testbarkeit

- *DI-Container* kann durch statischen Code ersetzt werden. (z.B. zur Platzierung von Mock Objekten bei Tests)

▪ Alternative: Dependency Injection per Methode (Setter) oder Variable

- *POJO = Plain Old Java Object, POJI = Plain Old Java Interface*
 - Fowler für uneingeschränkte Java Klassen (evtl. Java Beans)

- **Entitäten unterliegen nur wenigen Einschränkungen**
 - Vererbung und Polymorphismus möglich
 - Assoziation, Aggregation, Komposition möglich
 - Datentypen möglich

- **Neue Auffassung von Entitäten**
 - Direkte Persistenz des Domänenmodells
 - Können auch außerhalb einer Transaktion verwendet werden (Detached)

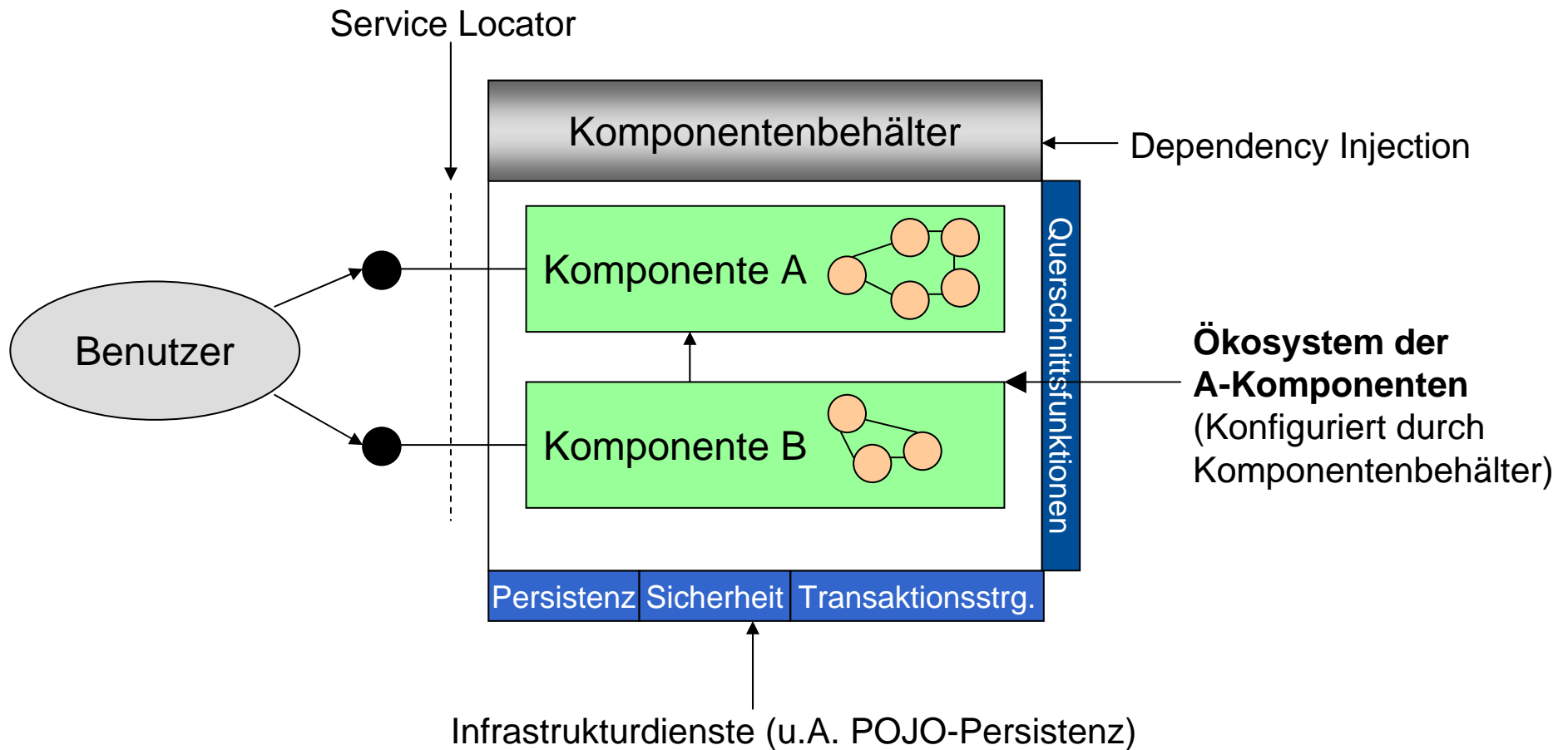
- **Muster: Data Access Object (DAO)**
 - Verwaltet Entitäten (Anlegen, Ändern, Löschen, Abfragen)
 - Kapselung der Daten-Beschaffungs-Technik
 - Fremdsystem
 - Webservice
 - OR-Mapper
 - Plain Old JDBC
 - Viel flexibler als EJB Home Interface

- **Umgang mit Komponentengrenzen**
 - Wie werden Beziehungen zwischen Entitäten über Komponentengrenzen aufgelöst?
 - Vorschlag: per Id-Verweis auflösen
 - Problem dabei: Kaskadierung von Operationen (eigentlich aber fachlich)

- **Umgang mit Schichtengrenzen**
 - Schichten verschwimmen, wenn selbe Entität in mehreren Schichten verwendet wird
 - Vorschlag:
 - mehrere Modelle (Persistenz-Modell, Domänenmodell, Value Objects)
 - Transformation der Modelle an den Schichtengrenzen (Boundaries)
 - Beispiel:
 - Persistenz-Modell (Mittler zwischen OO und DB, volle DB Flexibilität, evtl. Kompromisse für DB)
 - Business-Entitäten-Modell + Transformator des Persistenz-Modells (fachliche Repräsentation der Domäne inkl. fachlicher Methoden, volles OO-Geschütz)
 - Evtl. Standard: *Service Data Objects* (SDO, JSR-235)

- Vorteile von Hibernate:
 - Bietet POJO-Persistenz
 - Starker Einfluss auf kommenden EJB 3.0 Standard
 - Frei und kostenlos verfügbar (www.hibernate.org)
 - Exzellenter Support durch Community und breit verstreuten Erfahrungen. Fester Entwicklerstamm durch JBoss Inc.
 - Gute Toolunterstützung
 - Gute Plugins für Eclipse und IntelliJ
 - Tools für Roundtrip (Modell, Code, Mapping-Files, DDLs)
 - Hochgradig anpassbar (z.B. bei unternehmensweiten Datentypen)
 - Robuste und performante Implementierung
 - Einfache Verwendung
 - Transparentes Dirty Checking (Änderungsüberprüfung)
 - Operationen nahe am gewohnten Insert, Update, Delete
 - Kaskadierung von Operationen kann gesteuert werden
 - Einfache Konfiguration
 - Eng in J2EE Server integrierbar
 - Standalone Betrieb ohne Einbuße von Features möglich

Überblick: Komponentenframework



- CLEAR – *Component-oriented Lightweight Enterprise Architecture* [QAware, DialogData, BMW Group]
- Spring Framework
- HiveMind [Apache Jakarta]
- PicoContainer / NanoContainer [Codehaus]
- EJB (3.0)

- **Kein echtes Komponentenmodell (wie EJB auch)**
 - Komponentendefinition gem. Komponentenmodell = Metadaten (Java kennt keine Komponenten)
 - Spring und HiveMind Deskriptoren sind wenig komponentenorientiert
 - Höchstens Modul-Konzepte
 - Vermischung von Anwendungslogik und Technik in den Deskriptoren

- **Was ist ein echtes Komponentenmodell?**
 - Expliziter Export und Import von Schnittstellen (Außensicht)
 - Versteckt die Implementierung hinter Schnittstellen. Besitzen ein abgeschirmtes Innenleben. (Innensicht)
 - Einheit der Wiederverwendung und Unabhängigkeit.
 - Sie macht nur minimale Annahmen über ihre Umgebung
 - Kann isoliert entwickelt und getestet werden
 - Wesentliche Einheit des Entwurfs, der Implementierung und der Planung (neben den Schnittstellen)
 - Kann zusammen mit anderen Komponenten zu größeren Strukturen vereinigt werden (Service-Verbund)

- **Wachsen anorganisch**
 - Besonders Spring kann mittlerweile mehr als EJB
 - Aber: es können auch nur Teile verwendet werden
 - Wust an Bibliotheken und Frameworks, die mitgeliefert werden (vgl. Linux Distribution)

- **Wenig Architektur-Prinzipien für Großprojekte**
 - Literatur beschränkt sich auf Basisthemen *Dependency Injection* und Infrastrukturdienste (POJO Persistenz, Transaktionskontrolle, ...)

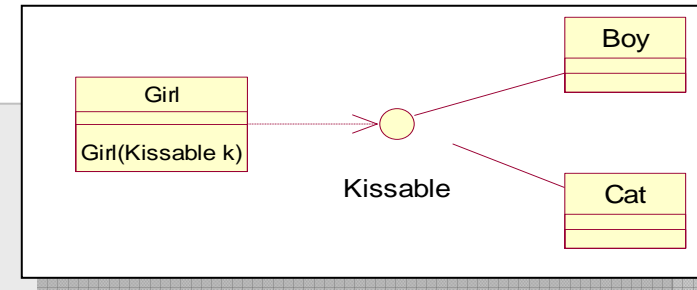
- Kann an Unternehmens-Architektur angepasst werden
 - Abstraktes Modell, das gemäß der Standardarchitektur implementiert werden kann (eigene Deskriptoren und Regeln für den Aufbau von Abhängigkeiten)
 - Eigener Design-Checker, der Verstöße gegen Architekturvorgaben meldet

- Strikte Trennung von Anwendungslogik und Technik
 - Eigene, abstrakte Schnittstellen für Infrastrukturdienste „Persistenz“, „Transaktionssteuerung“, „Konfiguration“, „Sicherheit“ und „Logging“
 - Raum- und Umgebungskonzept
 - Test/Integration/Produktion; J2EE und J2SE
 - Konfigurationswechsel durch eine Änderung eines Parameters.
 - Definiert in Groovy-Skripten

- Unterstützung von echten Software-Komponenten
 - Kontrolliertes Dependency Injection (Auto-Wiring der Innensicht, Deklarative Außensicht)
 - Integriertes Mock-Konzept für Isolation von Komponenten

```
<model >
  <component name=„hol l ywood“ >
    <exports i nterface=„Gi rl “ />
    <i mports i nterface=„Ki ssabl e“
      component=„ki ssabl es“ versi on=„1.0“ />
    <defi ni ti ons>
      <servi ce i nterface=„Gi rl “ i mpl =„Gi rl Impl “ />
    </defi ni ti ons>
  </component>

  <component name=„ki ssabl e“ versi on=„1.0“ >
    <exports i nterface=„Ki ssabl e“ />
    <defi ni ti ons>
      <servi ce i nterface=„Ki ssabl e“ i mpl =„Boy“ />
      <!--<servi ce i nterface=„Ki ssabl e“ dummy=„true“ />-->
    </defi ni ti ons>
  </component>
</model >
```



- **Management der Integration von OS-Bibliotheken**
 - Integrationsmöglichkeit
 - Wie integriert sich das Framework in die bestehende Architektur?
 - Konfigurationsmanagement
 - Was machen bei neuer Version?
 - Wie passen die unterschiedlichen Versionen zusammen?
 - Wie wird mit eigenen Anpassungen umgegangen?
 - Know-How, Community Support, Dokumentation

- **Anpassung der Software-Entwicklungs-Umgebung**
 - Spring IDE (Eclipse)
 - Hibernate Plugins (Eclipse, IntelliJ, ant, Maven)
 - Umfassende Lösung fehlt

- **Management Argumente:**
 - sanfte Migration zu EJB 3.0
 - sanfte Migration von EJB 1.x, 2.x

- **Wenig bis keine Verzögerungen durch Technology-Traps**
 - Kleine Hibernate Probleme: eigener Numserver, BLOBs, Enums
- **Konsequente Objektorientierung und Komponentenbildung**
 - Nachweisbar durch Metriken und Abhängigkeits-Visualisierung
- **Durchgängige Schichtentrennung**
 - Teil-Teams: Präsentation, Businesslogik und Persistenz
 - Wenig Synchronisationspunkte nötig
- **Steile Lernkurve**
 - Einführung innerhalb von einem Arbeitstag
- **Hohe Performance**
- **Gesteigerte Testbarkeit**
 - C1-Testüberdeckung von über 80% in der Anwendungslogik
 - Testsuite mit 54 Testcases in 11 Sekunden durchlaufen
- **Einziges Problem:**
 - Nerviges Deployment der Präsentationssicht auf den J2EE Server ☺

- **Kleine bis mittlere Projekte**
 - Keine transaktionale Legacy-System Anbindung
 - Kein Sitzungszustand in Anwendungslogik

- **Beispiele**
 - Abteilungsanwendungen
 - Anwendungen mit komplexen Objektmodell
 - Batch-Anwendungen
 - Reporting-Anwendungen
 - Komponentenmodell für Rich-Client Anwendungen

- **Flexibilität für Großprojekte vorhanden**
 - Zugriff auf EJB Komponenten

- **Grundlage**
 - Early Draft 8/2004
 - Konferenz-Präsentationen

- **Integraler Baustein der J2EE 5.0**
 - Ziel zur finalen Spezifikation: 2006
 - Grundsatz „Simplicity and Ease of Development“

- **Konzepte**
 - Inversion of Control mit Dependency Injection
 - Metadaten (JSR-175 Annotationen)
 - Configuration on Exception
 - POJOs
 - Session Beans (können remote sein)
 - Entity Beans (sind immer lokal)

- **Offene Punkte**
 - Persistenzlayer
 - in extra JSR? Konvergenz mit JDO 2.0
 - Generell abstrahiert und austauschbar ?
 - Abwärtskompatibilität (New Style – Old Style)

■ Buchtipp

- Rod Johnson: *J2EE without EJB*
- Bruce Tate: *Better, Faster, Lighter Java*
- Gavin King, Christian Bauer: *Hibernate in Action*
- J. Siedersleben: *Moderne Softwarearchitektur – Quasar*

■ Quellen im Internet

- *PicoContainer* (<http://www.picocontainer.org>)
- *HiveMind* (<http://jakarta.apache.org/hivemind>)
- *Spring* (<http://www.springframework.org>)
- *EJB 3.0* (<http://www.jcp.org/en/jsr/detail?id=220>)

■ Organisation

- Arbeitskreis der JUGM
- Treffen im 2-monatlichen Turnus

■ Ziele

- Sammlung von Best Practices und Patterns
 - Generator
 - Publikation
- Migrationsprobleme EJB → Hibernate
- ...

■ Interessenten

- Bitte melden... (Josef.Adersberger@QAware.de)

Taken together, we have a roadmap to a better place where the code we write solves the problems we face instead of the problems brought on by our tools.

By keeping to the path, remembering our principles, and using good old-fashioned common sense, we can beat back the bloat and write better, faster, lighter Java.

Bruce Tate, 2004, "Better Faster Lighter Java"