Walmart

Mike Barlotta

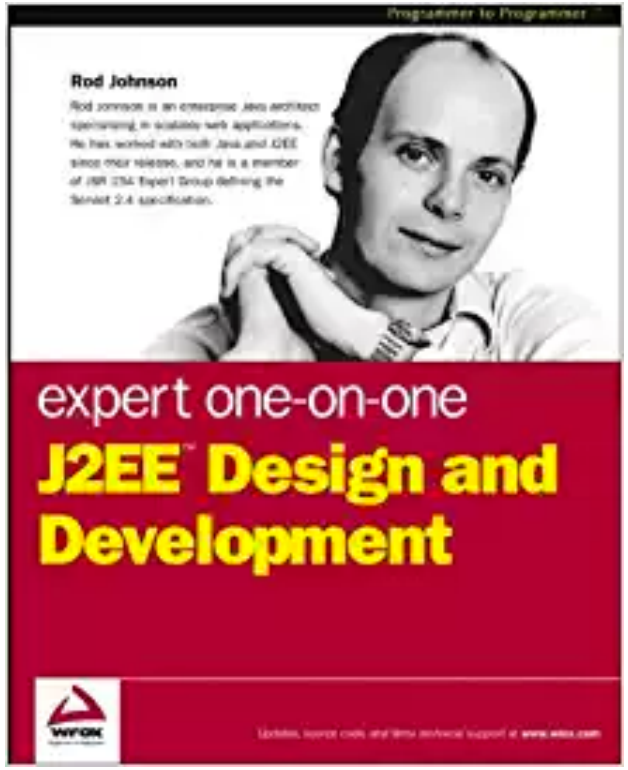
Spring Framework Part 1: the Basics

Java Guild Talks on Spring & DI

- Part 1: Basics
  - What is DI?
  - What does DI solve?
- Part 2: Spring DI
  - How it works
    - How to define dependencies
    - Application Context(s)
    - Bean Factory
- Part 3: Spring Bean Factory
  - How does it do that

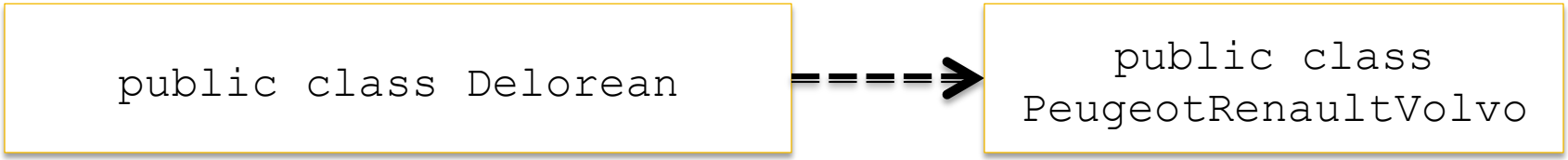
What is Spring?

- Dependency Injection framework
  - introduced in 2002 in Rod Johnson's book
  - open sourced in 2003
  - Spring 1.0 available in 2004
- Ecosystem of Capabilities
  - Spring MVC
  - Spring Data
  - Spring Batch
  - Spring Integration
  - Spring Cloud



What is Dependency Injection

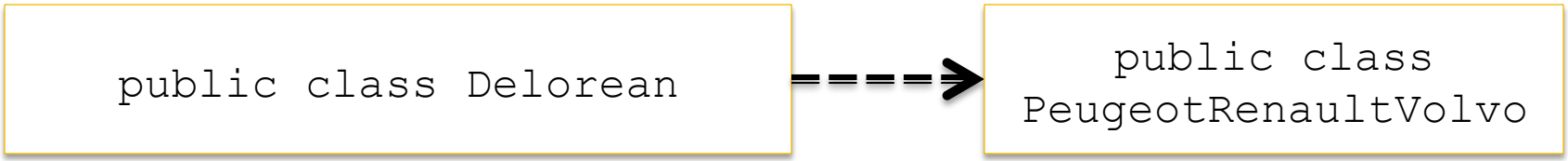
Starting with simple set of Classes




```
public class Delorean {
    private PeugeotRenaultVolvo prv;

    public Delorean() {
       prv = new PeugeotRenaultVolvo();
    }
```

What is wrong with this?

Simple Dependency Injection

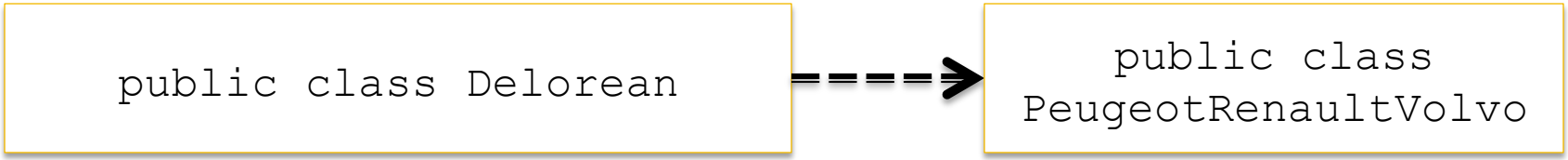



```
public class Delorean {
    private PeugeotRenaultVolvo prv;

    public Delorean(PeugeotRenaultVolvo prv){
       this.prv = prv;
    }
```

What is wrong with this?

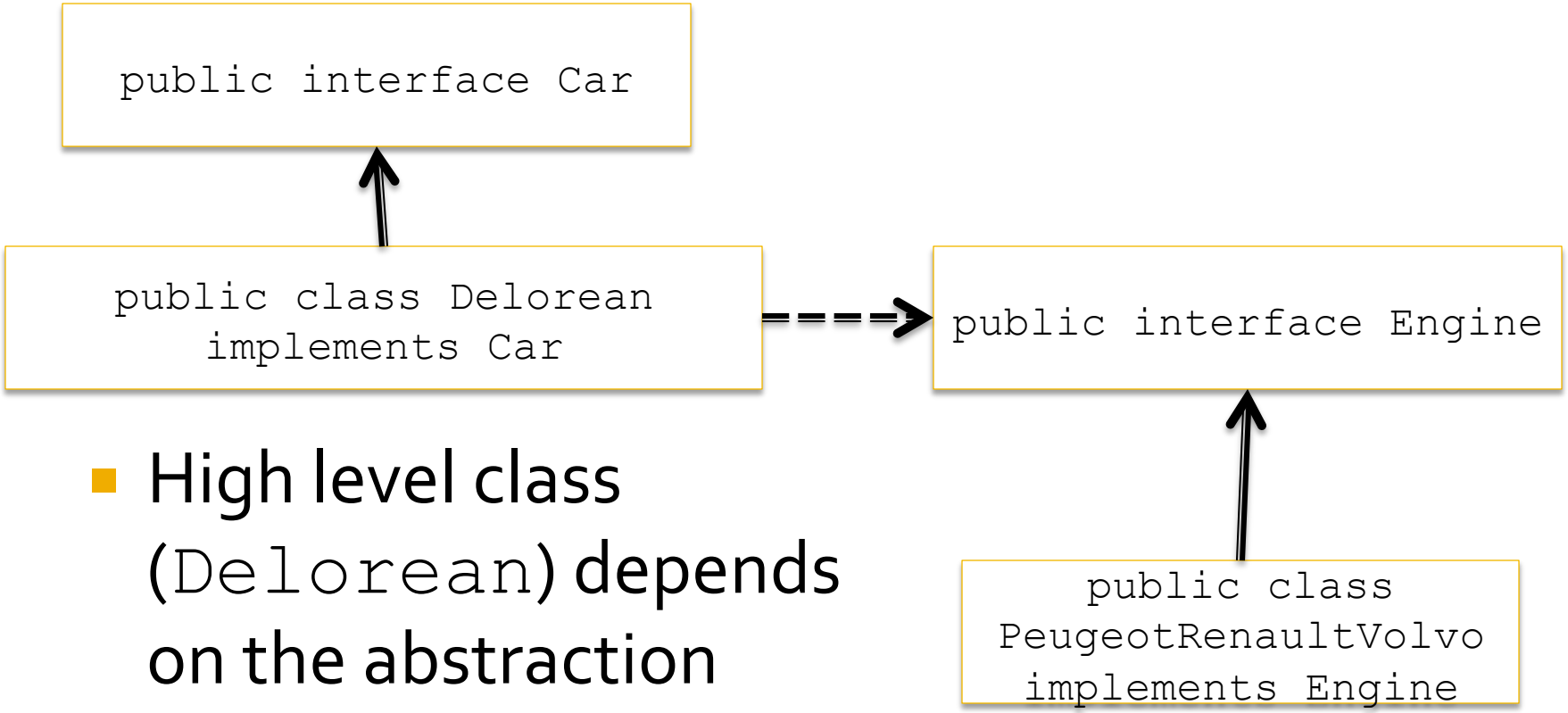
Dependency Injection is not DIP

Delorean can use dependency injection but violate DIP



- High level class (`Delorean`) depends on the lower level class (`PRV`) instead of an abstraction.
- Anyone that depends on `Delorean` would also be depending directly on a lower level class

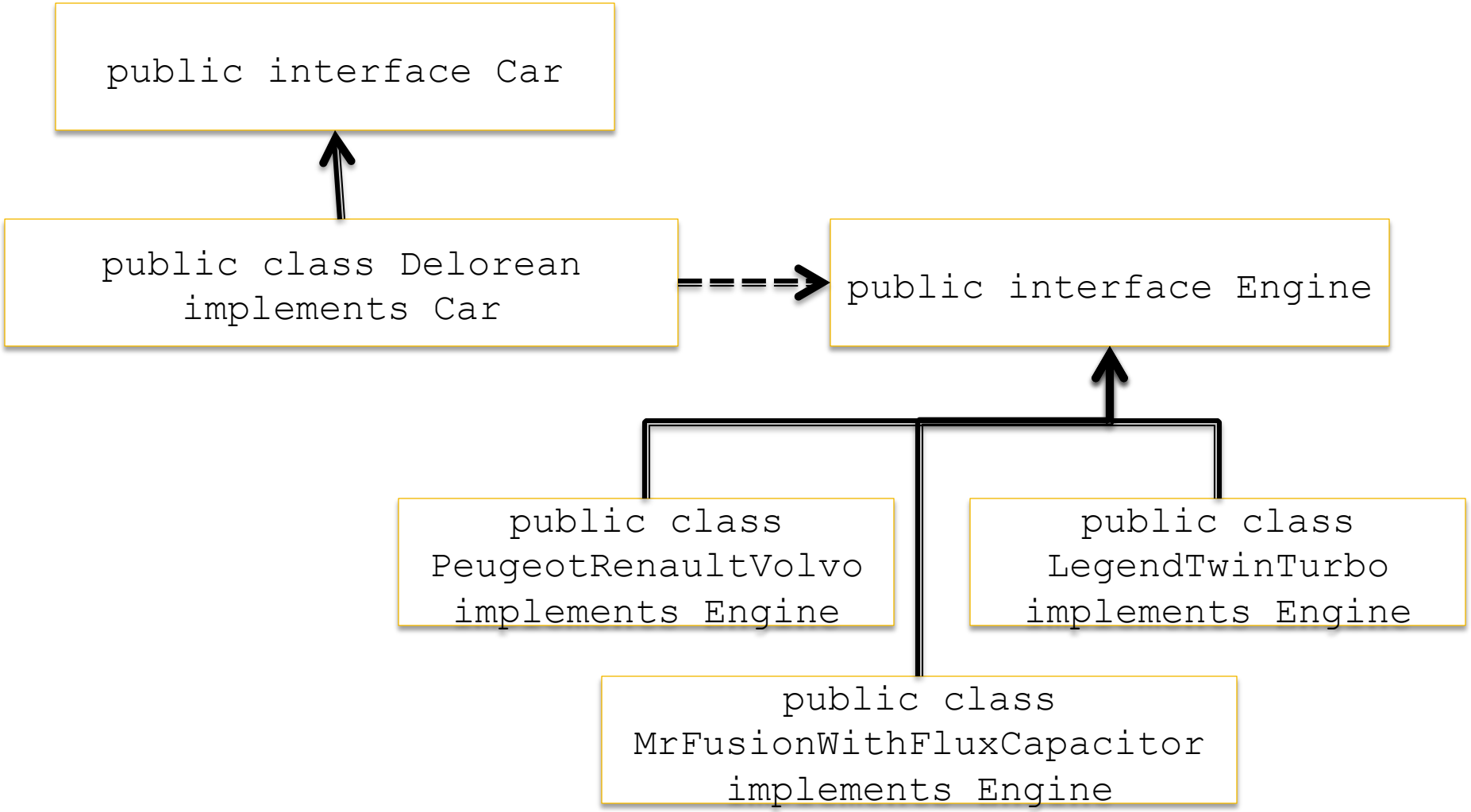
Design that can be used for DIP



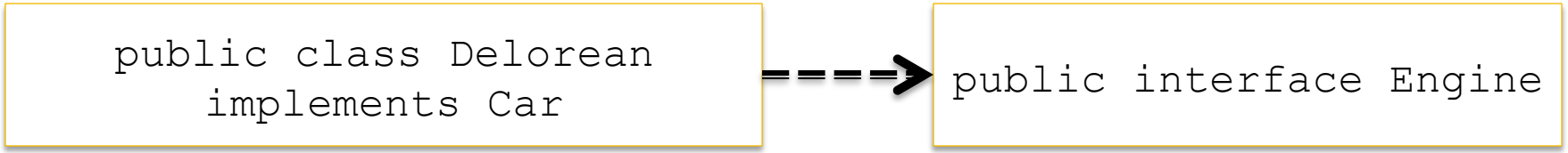
- High level class (`Delorean`) depends on the abstraction (`Engine`)

Which Engine?

DMC adds more Engine options



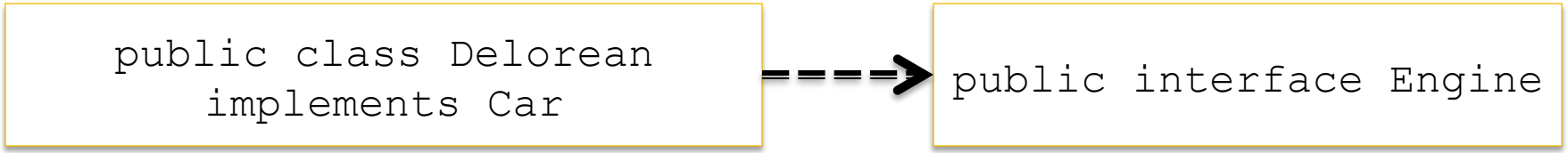
That new car smell




```
public class Delorean implements Car {
    private Engine engine;
    public Delorean() {
        engine = new PeugeotRenaultVolvo();
    }
```

Abstraction without DIP

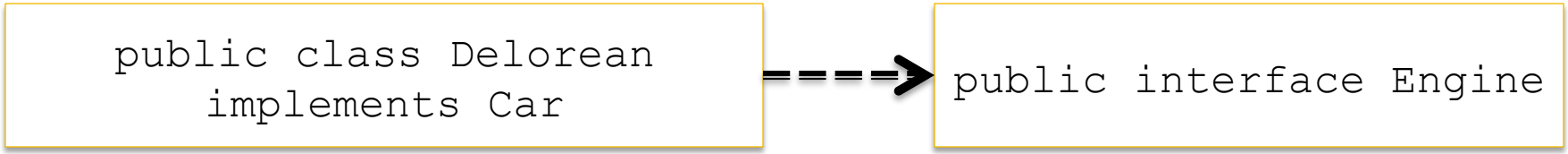
Factory Ordered




```
public class Delorean implements Car {
    private Engine engine;
    public Delorean() {
        engine =
        EngineFactory.getEngine("FLUX");
    }
```

DIP without DI

Fuel Injected




```
public class Delorean implements Car {
    private Engine engine;
    public Delorean(Engine engine) {
        this.engine = engine;
    }
```

DIP and DI

What decides which Engine to build and provides an instance?

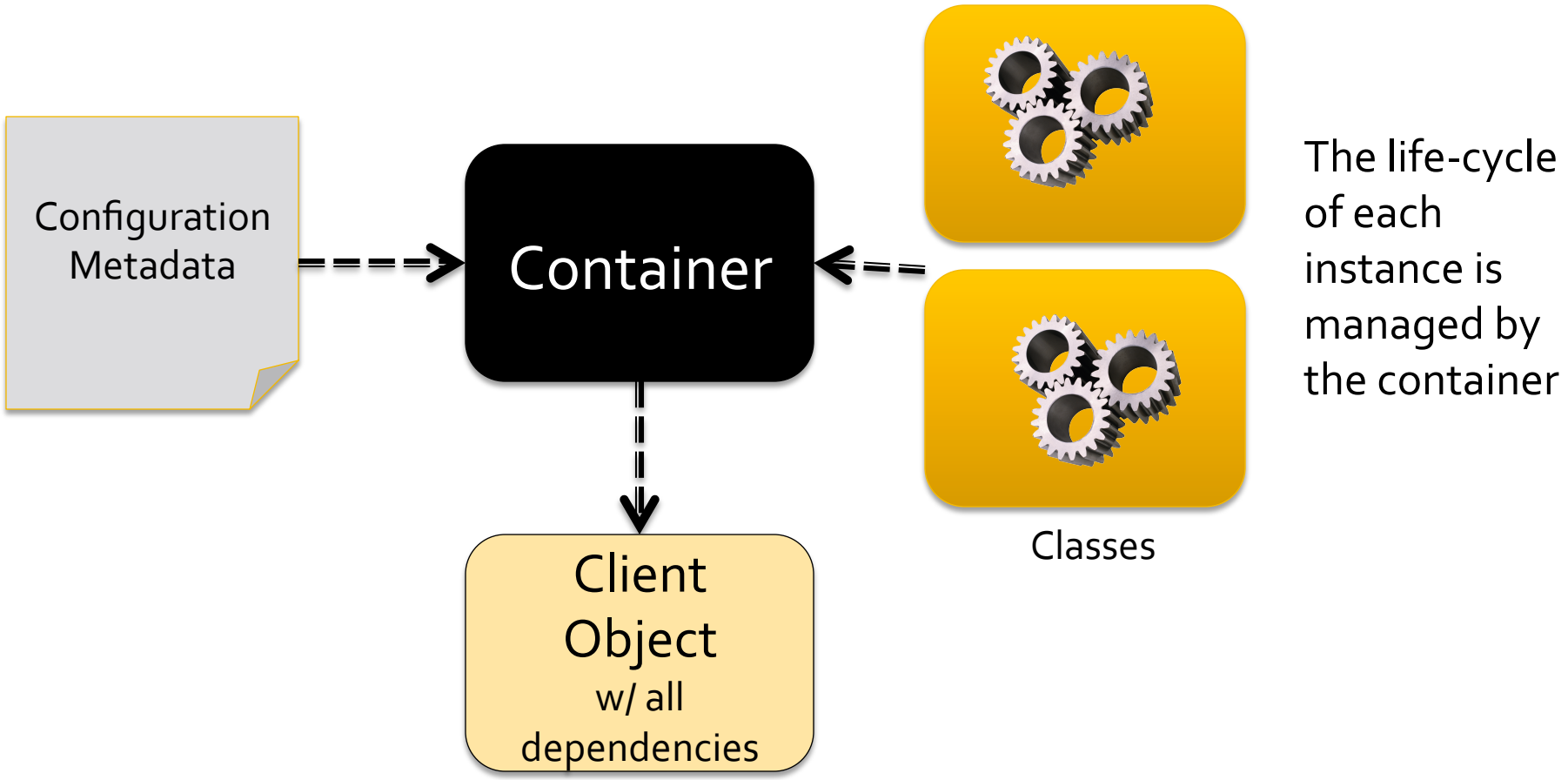
What is Dependency Injection

- A pattern or principle used to manage object dependencies
  - A client object has all of its dependencies given to it from outside of itself (i.e. injected)
  - When used with DIP it avoids tightly coupling the client object with the objects it uses
- Inversion of Control
  - Instead of the client object having control of what type of dependency it uses, it is inverted.
  - ***Someone else*** has control over creating the dependency and which type is to be used.

Dependency Injection Framework

- Decides which type of dependency the client object will **use**
  - which Engine is injected into our Delorean
- Decides how to **create** the dependent object
  - creates objects
  - handles the dependencies of dependent objects
- Decides the **lifecycle** of the dependent object
  - will the client object get a singleton or a new instance of the dependency
- Provides way to configure dependencies

Dependency Injection Framework



Types of Dependency Injection

- **Constructor**
  - Client object is created w/ all dependencies or not at all (assuming null checking)
  - Can test without the DI container
  - Can create immutable object
  - Lots of Dependencies = lots of parameters

```
private Engine engine;

public Delorean(Engine engine) {…

public Delorean(Engine e, DriveTrain dt, Tires t…
```

Types of Dependency Injection

- **Setter**
  - Can test without the DI container
  - Requires object to be mutable
  - No long list of parameters in constructor
  - Requires coupling to container to enforce all dependencies are provided

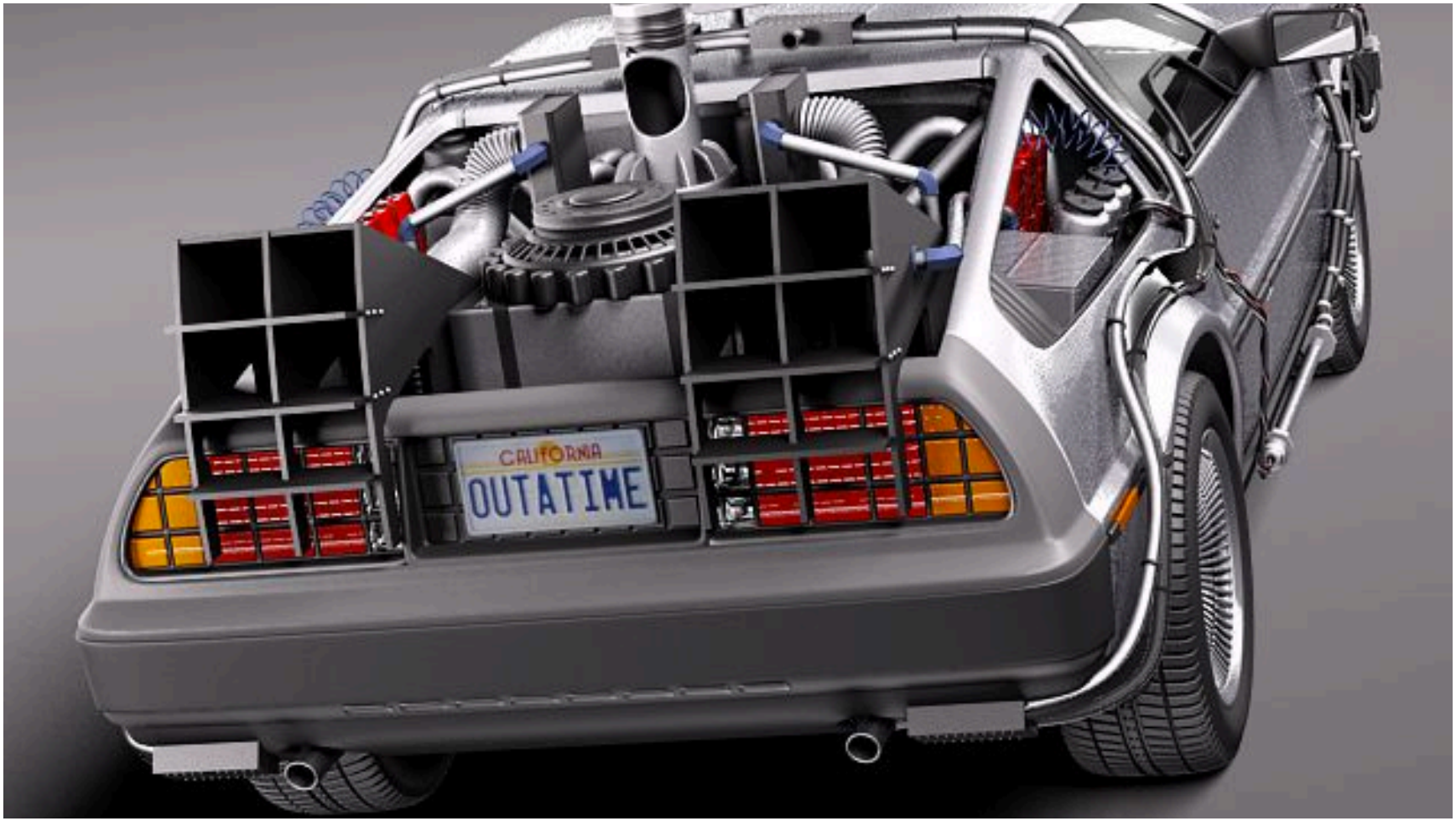
```
private Engine engine;

@Required
public void setEngine(Engine e) {
    engine = e;
}
```

Types of Dependency Injection

- **Field / Annotation**
  - Reduces amount of code needed
    - Default constructor, no setters
  - There must be unique bean that matches dependency or must qualify which type to use
  - Configuration tends to be spread throughout project
  - Need to test w/ DI container (or use Reflection)

```
@Inject
@Named("turbo")
private Engine engine;
```

There's more but first – let's go back in time



Server Side Java (1995 – 2000)

- Proprietary Application Servers
  - complex & no compatibility
  - lots of consolidation

Java Web Server (Jeeves)

atg

netdynamics

iPlanet™
e-commerce solutions
A Sun | Netscape Alliance

SYBASE
Jaguar CTS

SilverStream

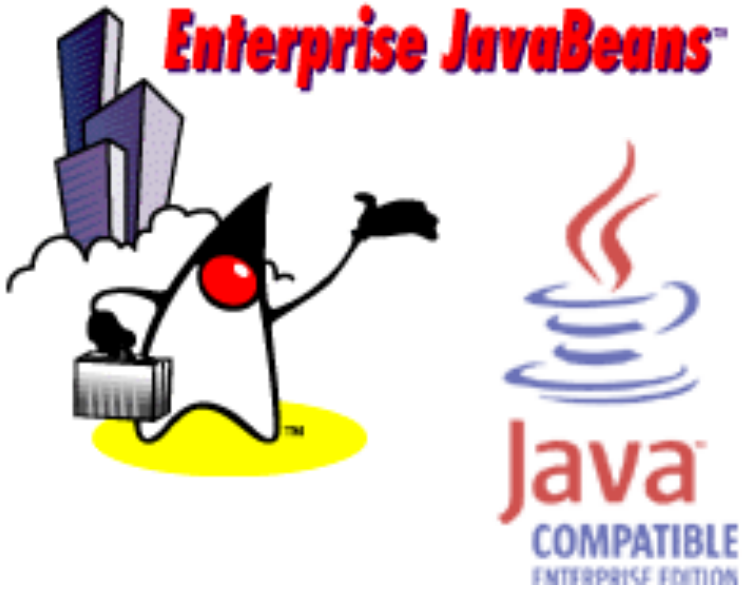
WebLogic

IBM

ORACLE

J2EE standardizes server-side Java

- Java Enterprise Edition
  - J2EE 1.2 in 1999
- Among the Specs
  - Servlets 2.2
  - EJB 1.1



- Vendors support & implement specs
  - Sun tests and certifies application servers

Java Web Server donated to Apache

Sun and Apache team up to deliver servlet and JSP code

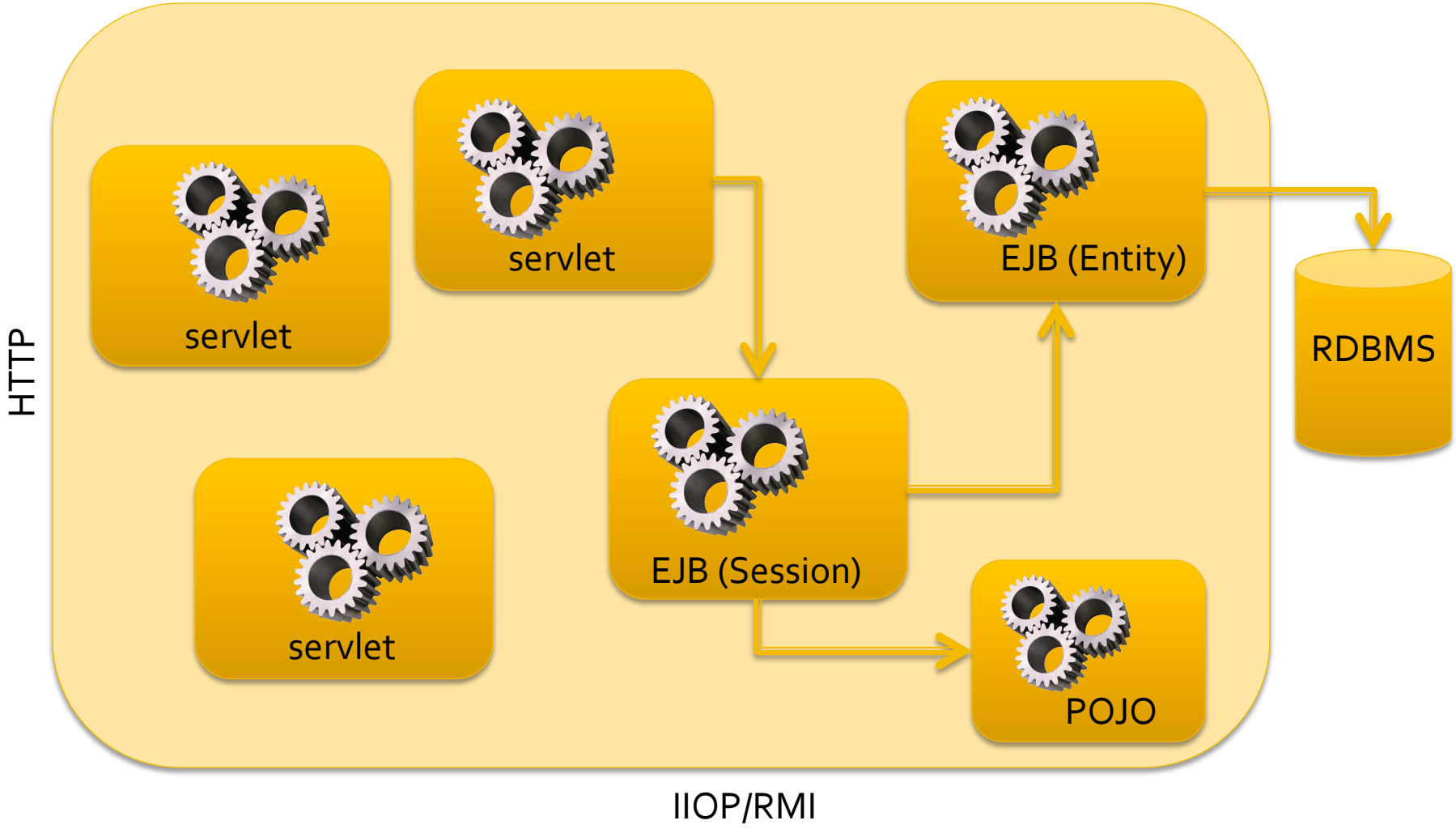
Servlet guru gives you the inside scoop on the newly announced

By Jason Hunter

JavaWorld | JUN 1, 1999 1:00 AM PT

- Java Web Server (released in 1997) is the initial Servlet container
  - before JSRs and the 1st official Servlet spec which was Servlet 2.1
- Donated in 1999 and released as Tomcat 3
  - Became the reference implementation for Servlet spec

J2EE Applications



What you get

- Remote accessible objects (IIOP/RMI or JMS)
- Three types of EJB objects
  - Entity Bean = ORM maps to persisted noun
  - Session Bean = business logic (services)
  - Message-Driven Bean = asynchronous logic
- Container managed life cycle of objects & transactions
  - Create, Destroy, Pooling, Number instances

Writing an EJB prior to JEE 6

```
import javax.ejb.*;
public interface HelloHome extends EJBHome | EJBLocalHome {
      public Hello create() throws CreateException, …
```

```
import javax.ejb.*;
public interface Hello extends EJBObject | EJBLocalObject {
      public String hi(String name) throws EJBException;
}
```

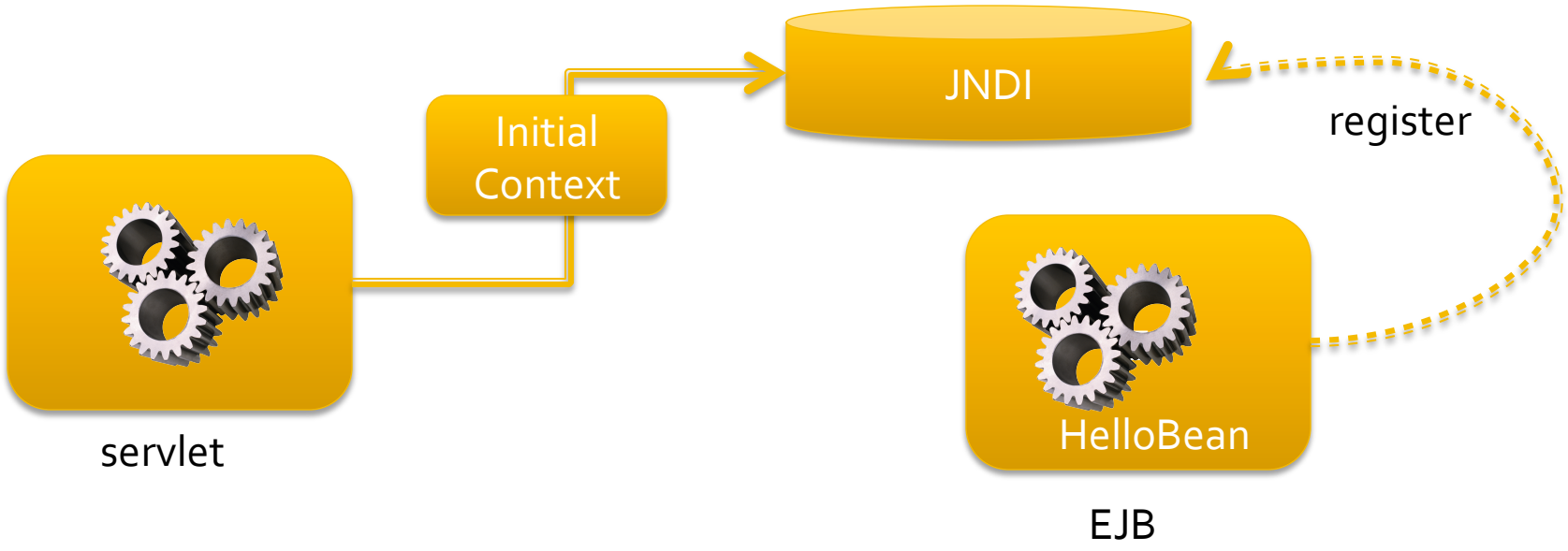
```
import javax.ejb.*;
public class HelloBean implements SessionBean {

      public String hi(String name) throws EJBException {
            return "Hello " + name;
      }

      // EJB life cycle methods go here
```

Did you notice…

- The actual EJB `HelloBean` does not implement either the **`EJBHome`** or **`EJBObject`** interfaces.
- The implementations are generated by the container. The **`EJBHome`** is registered in JNDI. All of this is done during deployment.
- The EJB must implement either **`SessionBean`** or **`EnityBean`**

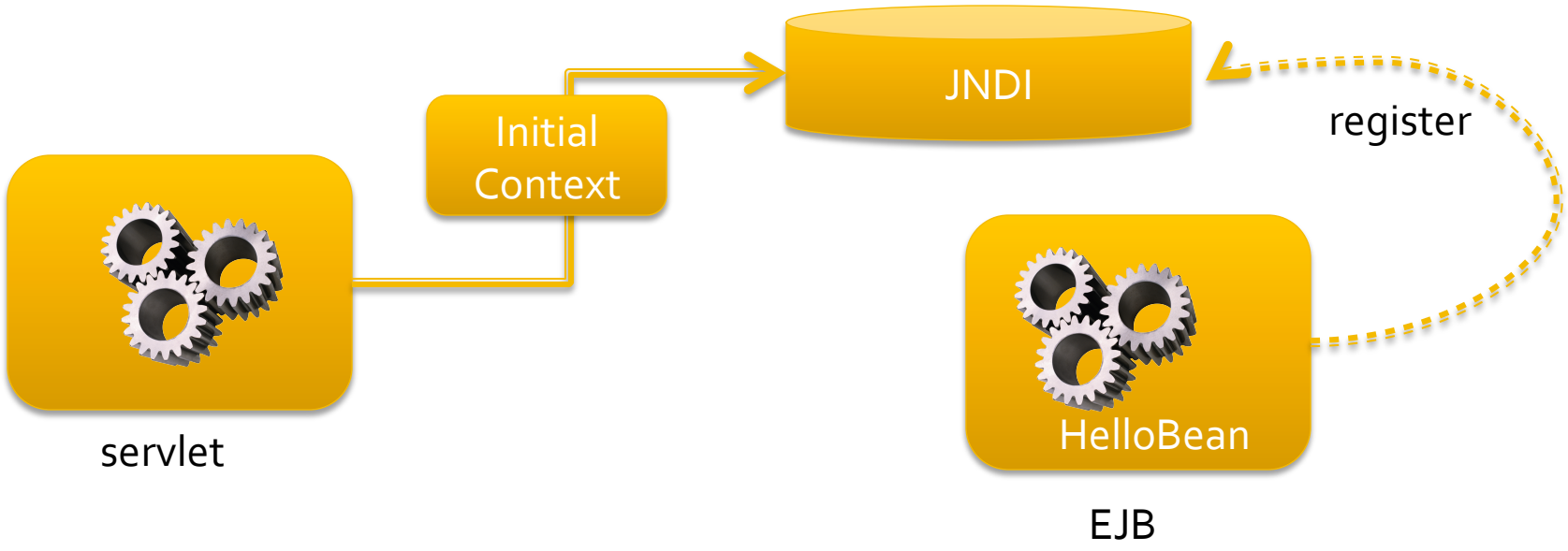
Calling the EJB: access JNDI of JEE container

```
InitialContext ctx = new InitialContext(env);
```



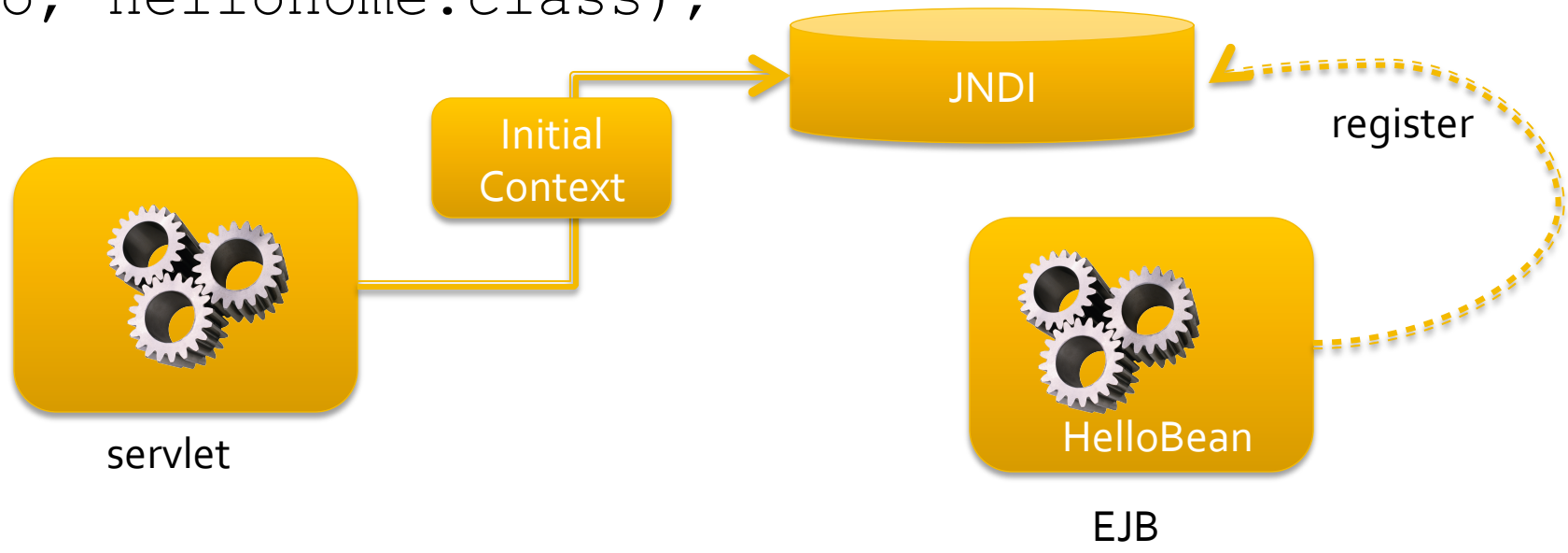
Calling the EJB: look up the EJB (local)

```
String jndiName = "java:comp/env/ejb/home";
HelloHome h = (HelloHome)ctx.lookup(jndiName);
```



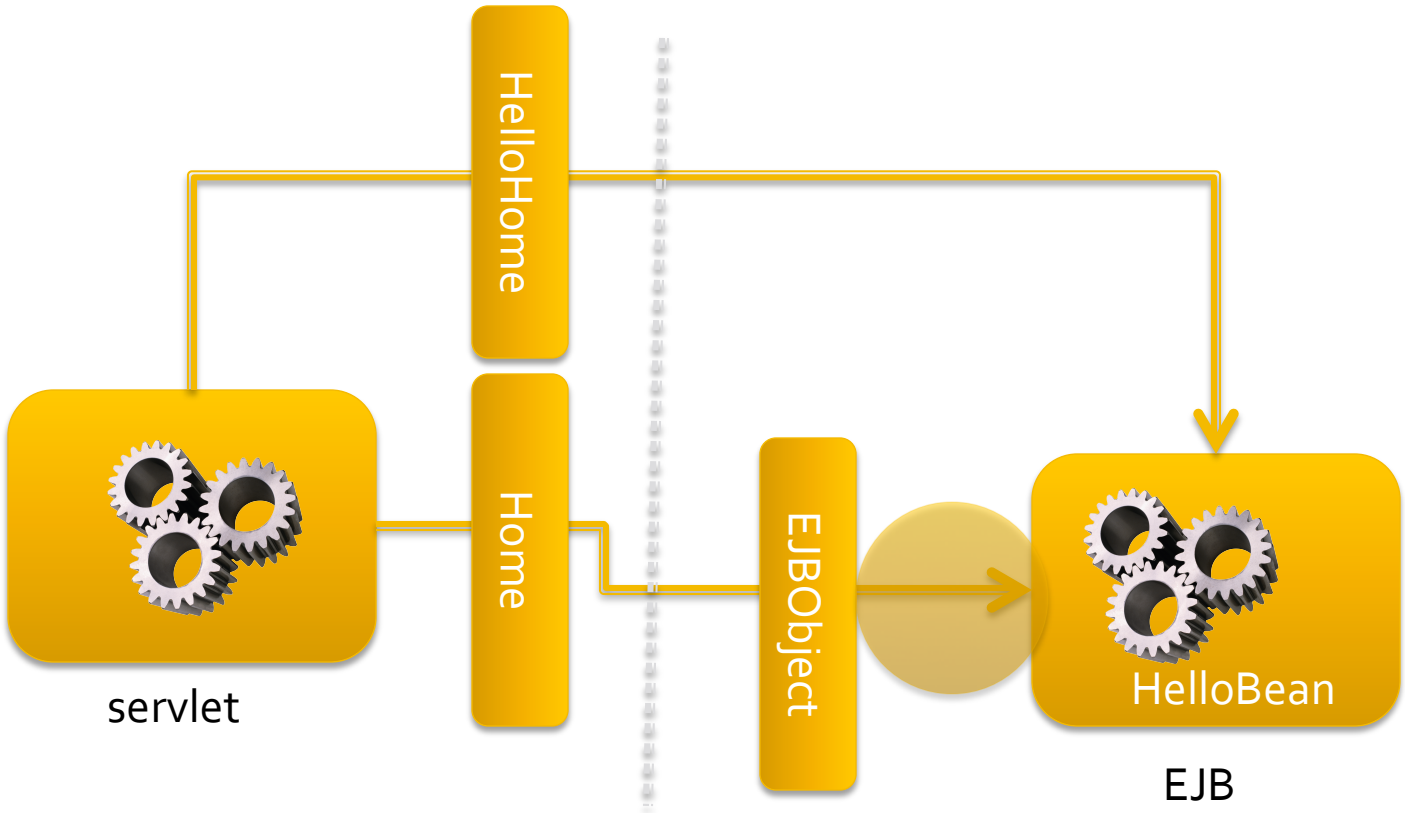
Calling the EJB: look up the EJB (remote)

```
String jndiName = "java:comp/env/ejb/home";
Object ho = ctx.lookup(jndiName);
HelloHome h = (HelloHome)
javax.rmi.PortableRemoteObject.narrow
(ho, HelloHome.class);
```



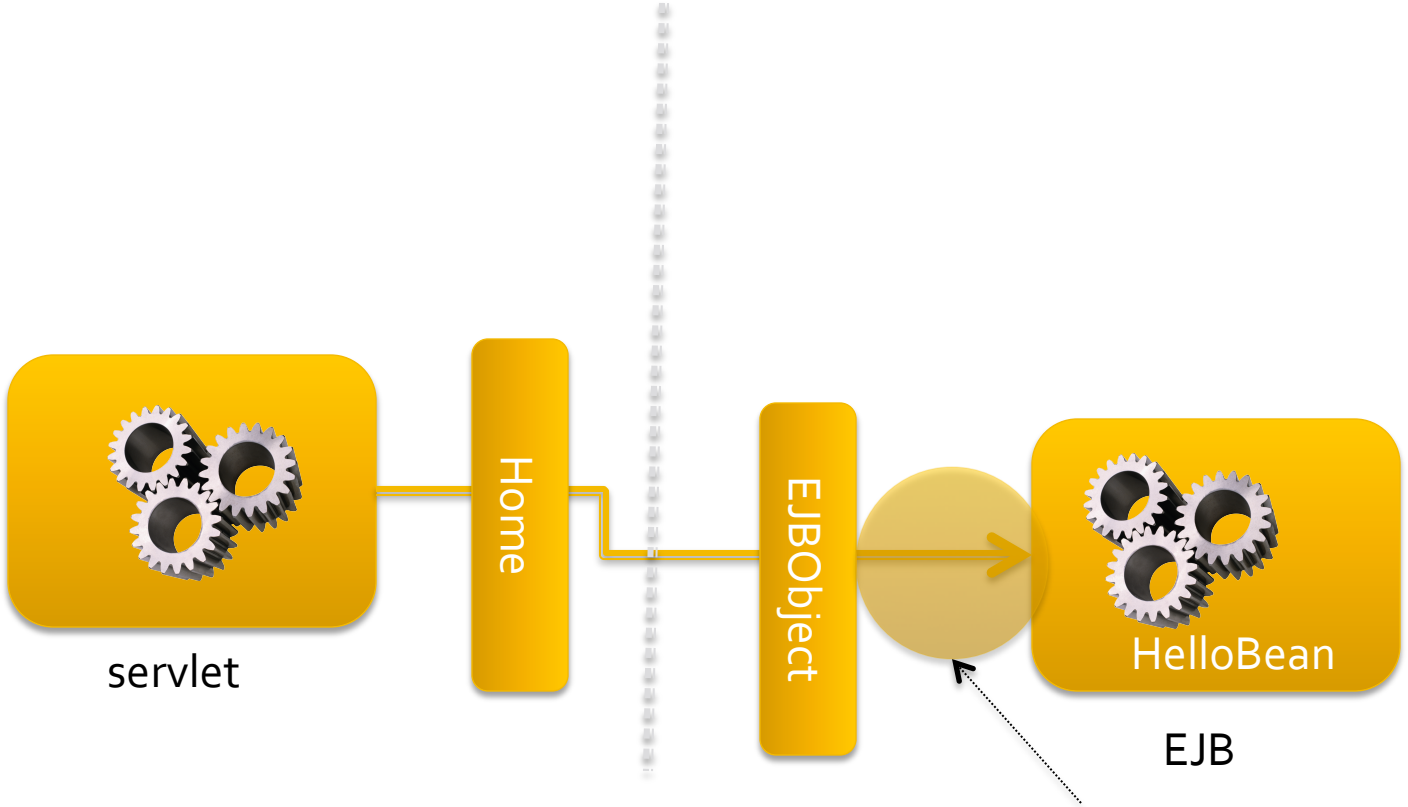
Calling the EJB: get an Object reference thru Home

```
Hello ejb = h.create();
```



Calling the EJB: call the method

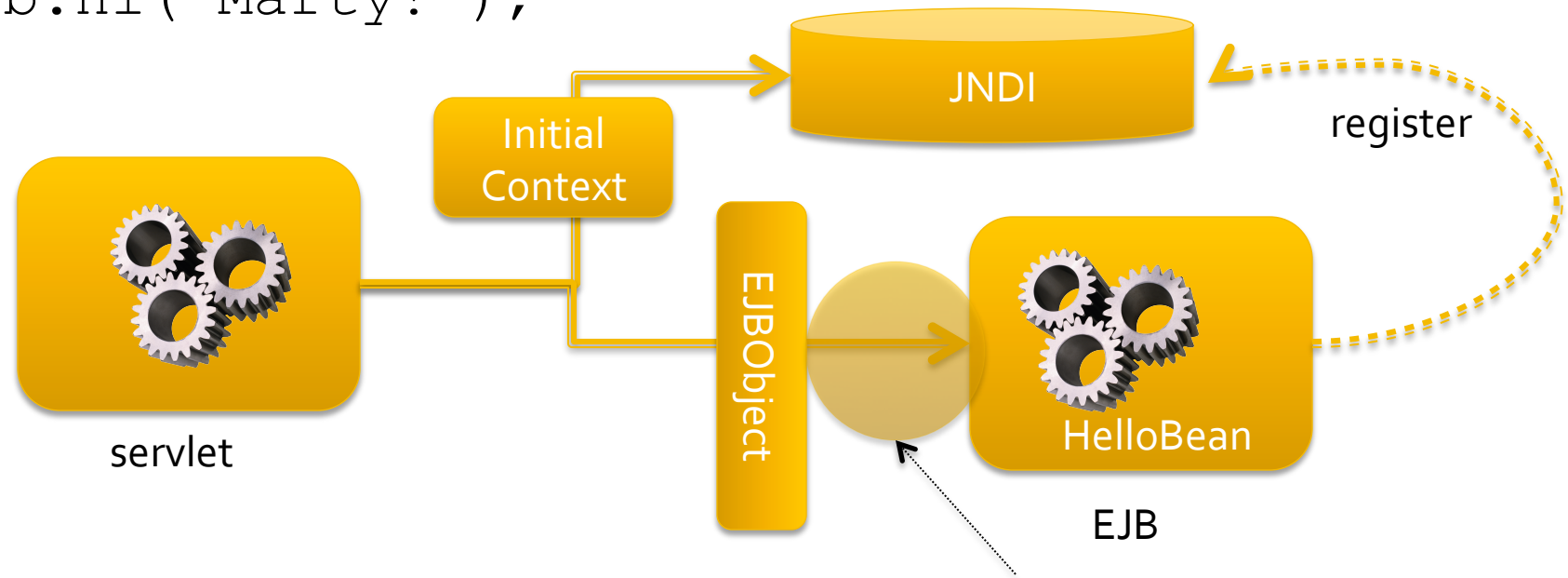
```
ejb.hi("Marty!");
```



Method interceptor: where container does work to manage life cycle & transactions

Calling the EJB

```
InitialContext ctx = new InitialContext(env);
String jndiName = "java:comp/env/ejb/home";
HelloHome h = (HelloHome)ctx.lookup(jndiName);
Hello ejb = h.create();
ejb.hi("Marty!");
```



Method interceptor: where container does work to manage life cycle & transactions

Did you notice…

- The client object must ask for its dependency (`HelloBean`) from the container (JNDI)
- The client object has to know ahead of time if the call to the object is remote or local.
- The client has fair bit of work to do just to call a local EJB method

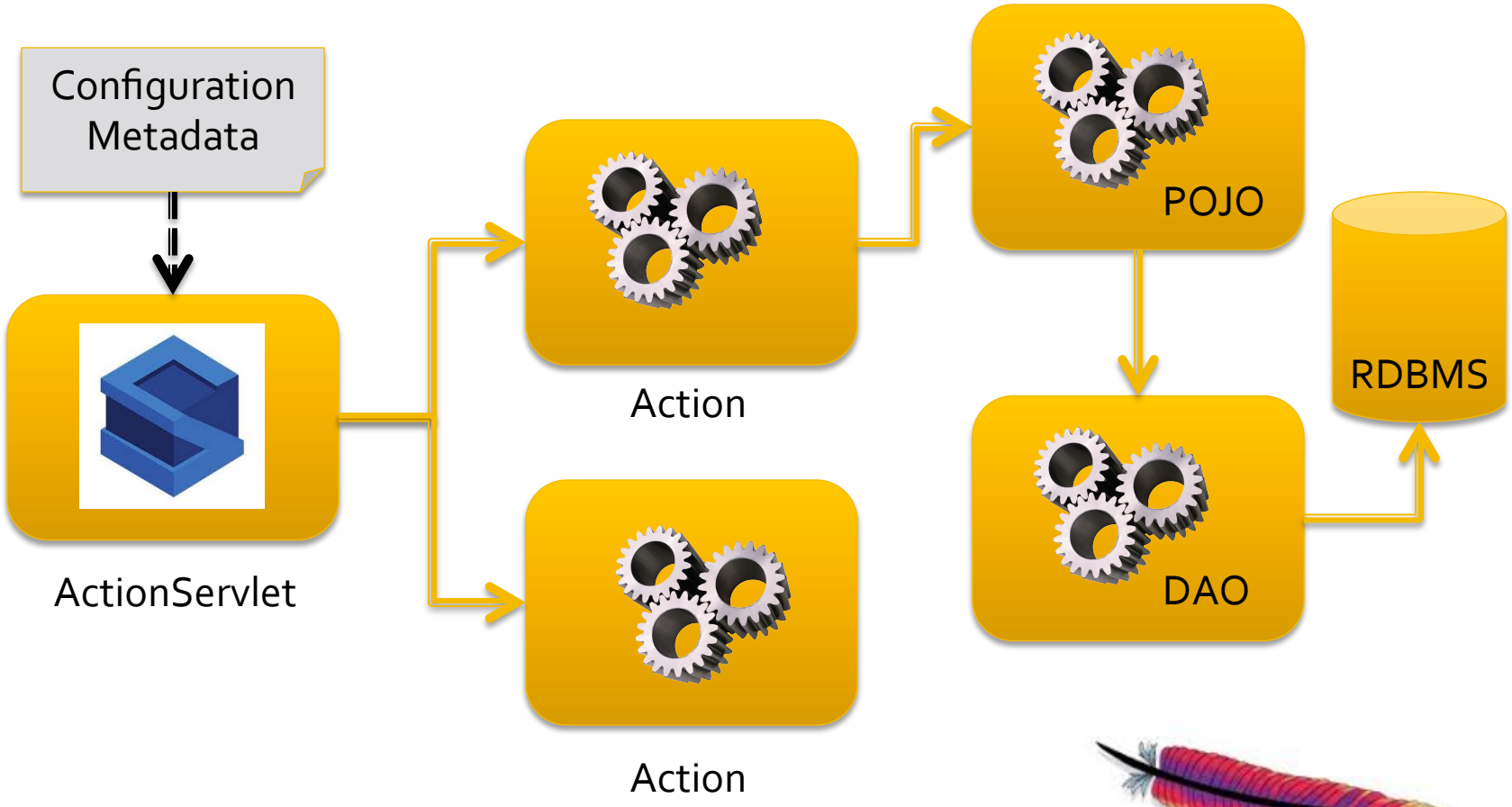
J2EE development was not fun

Typical day in the life…

Code is *dependent* on the container to run so…

1. Write EJB implementation
2. Write client to call EJB
3. Package application in an EAR file
4. Deploy EAR to JEE container
5. Test
6. Find Bug & start over

Most adopted Servlets & Struts



Apache

Struts donated to Apache in 2000


BACK
TO THE FUTURE

What ever happened to all those application servers from the 90's?

Choice: JEE 7 Containers

IBM WebSphere.

WildFly

ORACLE

FUSION MIDDLEWARE

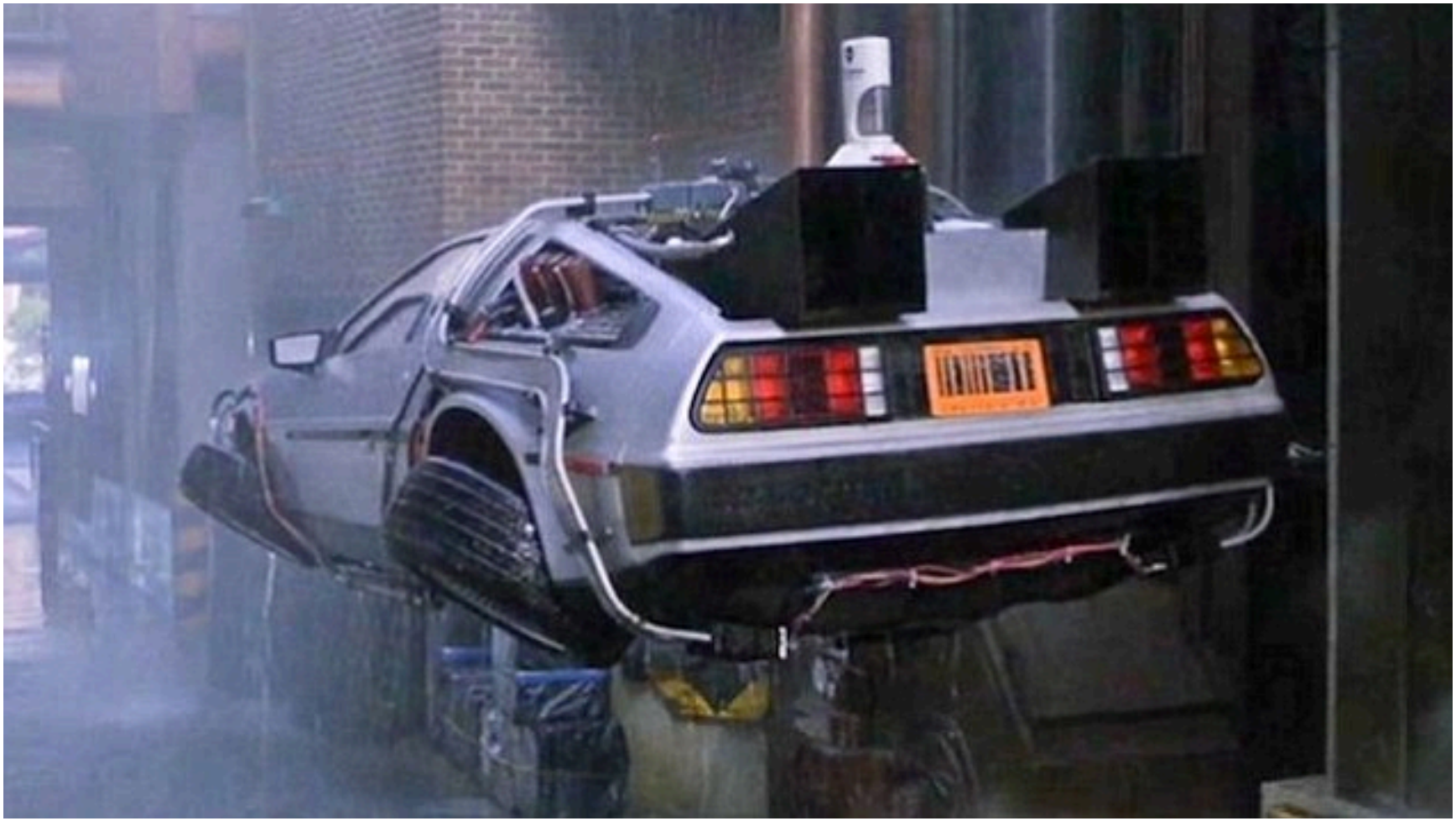
WEBLOGIC SERVER

Rod Johnson explains…

- some of the "J2EE Design Patterns" we've seen are more reactions to shortcomings in the technology … it's a hack rather than a pattern.
- I don't much like entity beans. There are very few situations in which I'd use them. … With AOP there's no need to use EJB to enjoy declarative transaction management.

http://www.theserverside.com/news/1364653/Expert-One-on-One-J2EE-Design-and-Development-Interview-with-Rod-Johnson

But where we're going we don't need EJB, JNDI, or JEE containers



Rebel Frameworks emerge

InfoWorld FROM IDG

INSIDER Sign In | Register

Java "rebel frameworks" touted

Burton Group: Open source projects can be alternative to or supplement J2EE technologies

Lightweight containers can be alternatives to J2EE or supplement the J2EE programming model, offering a simpler programming model by leveraging designs based on POJO (Plain Old Java Objects), IoC (Inversion of Control), and a lightweight form of aspect-oriented programming. Lightweight container examples include Spring, HiveMind, and PicoContainer.

The Mission

Spring's main aim is to make J2EE easier to use and promote good programming practice.

It does this by enabling a POJO-based programming model that is applicable in a wide range of environments.

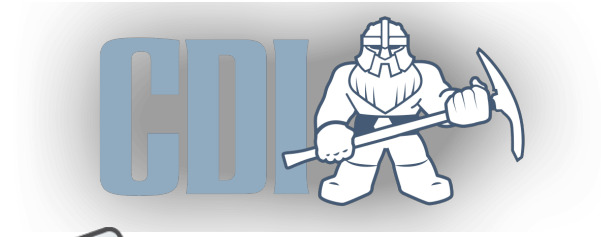
- Rod Johnson

http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework

Spring Cleaning: Cleaner Code

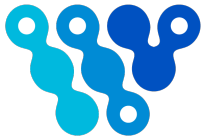
- Code is easy to write and test as POJO
- *Encourages* use of interfaces and DIP
- Minimal to no dependencies on Spring within objects
  - Compare that to EJB objects in J2EE
- Spring injects dependencies rather than having client objects ask for them
  - Compare that to using JNDI

Today: Lots of DI Options for Java



Guice

spring



OpenWebBeans

HiveMind

CAUCHO

PicoContainer

JSR 330 = DI for Java

- Defines standard set of annotations for DI
- Led by
  - Rod Johnson (Spring)
  - Bob Lee (Guice)
- Supported by
  - CDI spec
  - Spring IoC
  - Google Guice

JSR 330	Spring
Prototype by default	@Scope ("prototype")
@Singleton	@Scope ("singleton") *also the default*
@Inject	@Autowired
@Named	@Qualifier
@Named	@Component

Not to be confused w/ JSR-299 (and subsequent JSRs) which define CDI (injection and bean life cycle)

Context & Dependency Injection for JEE (CDI)

- Introduced as part of JEE 6 (2009)
  - JSR-299 (1.0)
  - JSR-346 (1.1, 1.2)
  - JSR-365 (2.0) *in review*

- Reference Implementation
  - Weld (Red Hat)
- Specification provides
  - Well-defined lifecycle for objects
  - Dependency Injection mechanism

Deeper look @ Spring