Spring, a J2EE extension framework

JUGS presentation

by Philipp H. Oser

30.08.2005

TECHNOLOGY CONSULTING INNOVATION

ELCA

Agenda

Spring: a J2EE extension framework

Introduction

More spring

Experience and benefits

ELCA in Brief

- **Foundation**
  - 1968

Technology Awards

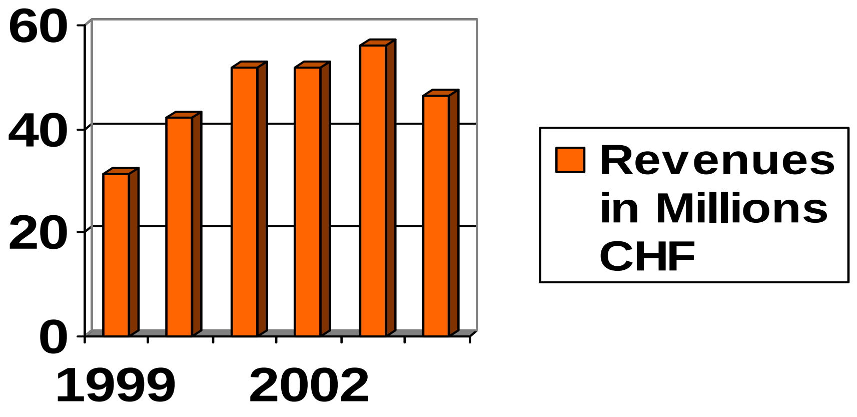
Workforce

Over 300 highly qualified employees

Revenue Progression

For 16 years ELCA has recorded positive results

Largest independent software developer & system integrator in Switzerland by now



Locations

Lausanne (Headquarters), Zurich, Geneva, Bern, London, Paris, Ho Chi Minh City

Typical Benefits of J2EE Extension Frameworks

Pre-built pieces	▪Architecture and guidelines ▪Reusable components ▪Development environment
Application Solidity & Homogeneity	▪Proven components ▪Best practices and patterns in code and guidelines ▪Standardized use of technology
Abstraction of platform	▪Platform simplification ▪Application agility through extension mechanism ▪Protection of platform error and change ▪Product and vendor independence

Commoditization of comprehensive Java EE frameworks

- Definition: *Comprehensive* Java EE framework :=
  A J2EE extension framework covering many domains of the Enterprise (not just one domain such as Persistence, Web UIs, or Remoting)

- Many comprehensive Java EE simplification & extension frameworks exist since the the early days of the J2EE
  - Mostly in-house/ proprietary
  - Some were successes in smaller contexts (e.g. CS Java Application Platform; SBB Framework; ELCA LEAF, used in 20 projects)
- Few became mainstream (exception: frameworks for a smaller domain, e.g. struts, hibernate, Xdoclet)

- More recently, *open* Java EE framework emerge (e.g. Spring or Keel)
  => customization
  => harder to justify new proprietary frameworks

Spring framework

- A popular lightweight dependency injection container
- Open source project (Apache license)
- http://www.springframework.org/
- Strengths
  - A lot of momentum around (used a lot, books, new developments around)
  - Based on JavaBeans
  - Significant improvement over pure Java EE development: standard container resources, code templates, ...
  - Integrated with many existing Java technologies: Struts, JSF, Hibernate, JDO, Toplink, Ibatis, JDBC, JMS, RMI, Soap, Velocity, Quartz, ...
  - Good documentation: free and not free
  - Mature and robust: fundamental parts go back to year 2000

- Other candidates: Pico container, HiveMind, Keel (seem less popular)

JavaBeans (the essence)

- The standard „component model“ of Java (JDK abstraction)
  - Uses normal Java classes
  - Components can have: properties, methods, events
- Uses naming conventions, no particular interfaces
  - E.g., read-write property `startDate` of type `Date` requires 2 methods:

```
Date getStartDate();
void setStartDate(Date date);
```

  - Other Java methods are *bean methods*
  - Events
- Most Java classes are JavaBeans
- Sample:

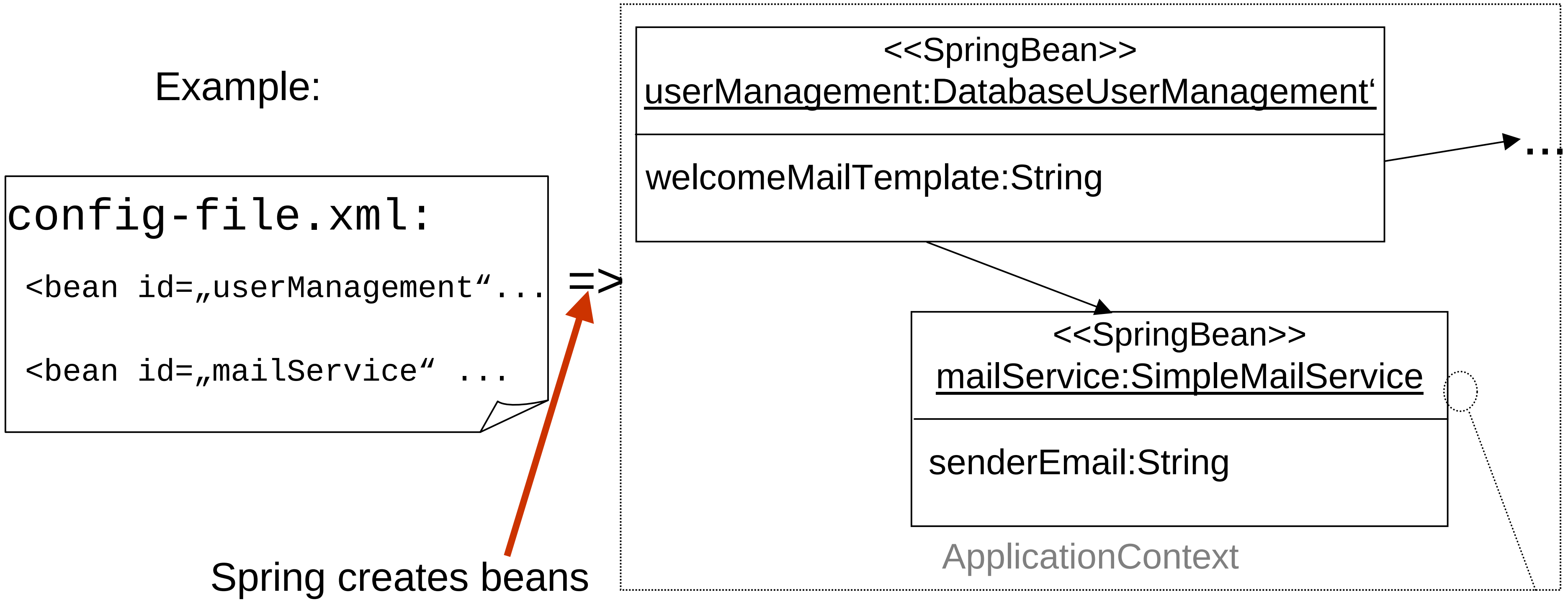
```
public class Person {
   private String name = „Titi“;

    void setName(String name) {
       this.name = name;
    }
}
```

Spring (the essence)

Spring sets up JavaBeans

- *Instantiation* of JavaBeans
- *Configuration* of JavaBeans (via Dependency Injection/ IoC)
  - Wiring between JavaBeans
  - Setting parameters on JavaBeans

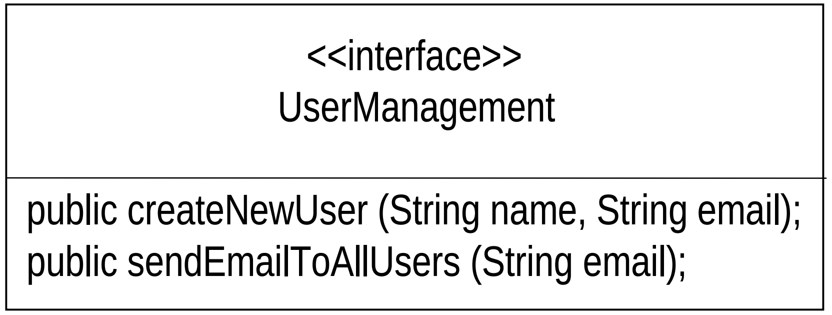
Example:


config-file.xml:
<bean id=„userManagement“...
<bean id=„mailService“ ...
=>
<<SpringBean>>
userManagement:DatabaseUserManagement‘
welcomeMailTemplate:String
...
<<SpringBean>>
mailService:SimpleMailService
senderEmail:String
ApplicationContext

Spring creates beans
in an ApplicationContext

Spring Demo

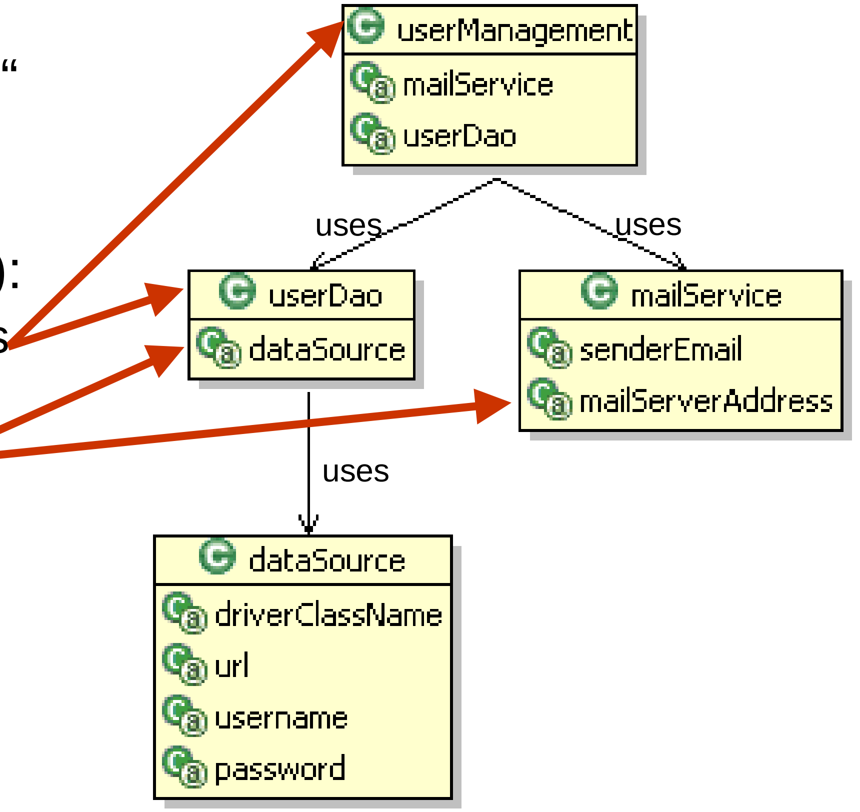
- Demo with a little user management component



<<interface>>
UserManagement

public createNewUser (String name, String email);
public sendEmailToAllUsers (String email);

- We setup the following „components“ (=spring beans) to implement the user management (picture auto generated with eclipse spring plugin):
  - Each box is a *spring bean*, spring id is shown
  - *Properties & wiring* of each spring bean are shown underneath



Demo: File overview

Java files (they contain no references to spring!)

- Interfaces
  - UserManagement.java
  - MailService.java
  - UserDao.java

- Classes implementing these interfaces
  - DatabaseUserManagement.java
  - SimpleMailService.java
  - DatabaseUserDao.java

- Helper class
  - UserDto.java

Configuration file (uses spring DTD)

- config2.xml

Demo in beanshell

- Launch beanshell (bsh) with the required jars/ classes in classpath
- Steps in the shell:
  - `Menu File->recapture System in/out/err`
  - `show();`

  - `import org.springframework.context.support.*;`
  - `ac = new FileSystemXmlApplicationContext("config2.xml");`

    Sets the graph of components up (does it lazily by default)

  -
  - `b.sendMail("I","hello");`
  - `b.getSenderEmail();`

Demo in beanshell (2)

- Steps in the shell:
  - `print(ac.getBeanDefinitionNames());`

  - `u = ac.getBean("userManagement");`
  - `u = ac.getBean("userManagement");`

    2 times the same instance (= singleton per JVM)
  - `u.sendEmailToAllUsers("hello");`

  - `u.createNewUser("John", "John@demos.org");`
  - `u.setWelcomeMail("Shorter Email {0} {1} ");`
  - `u.createNewUser("John2", "John@demos.org");`

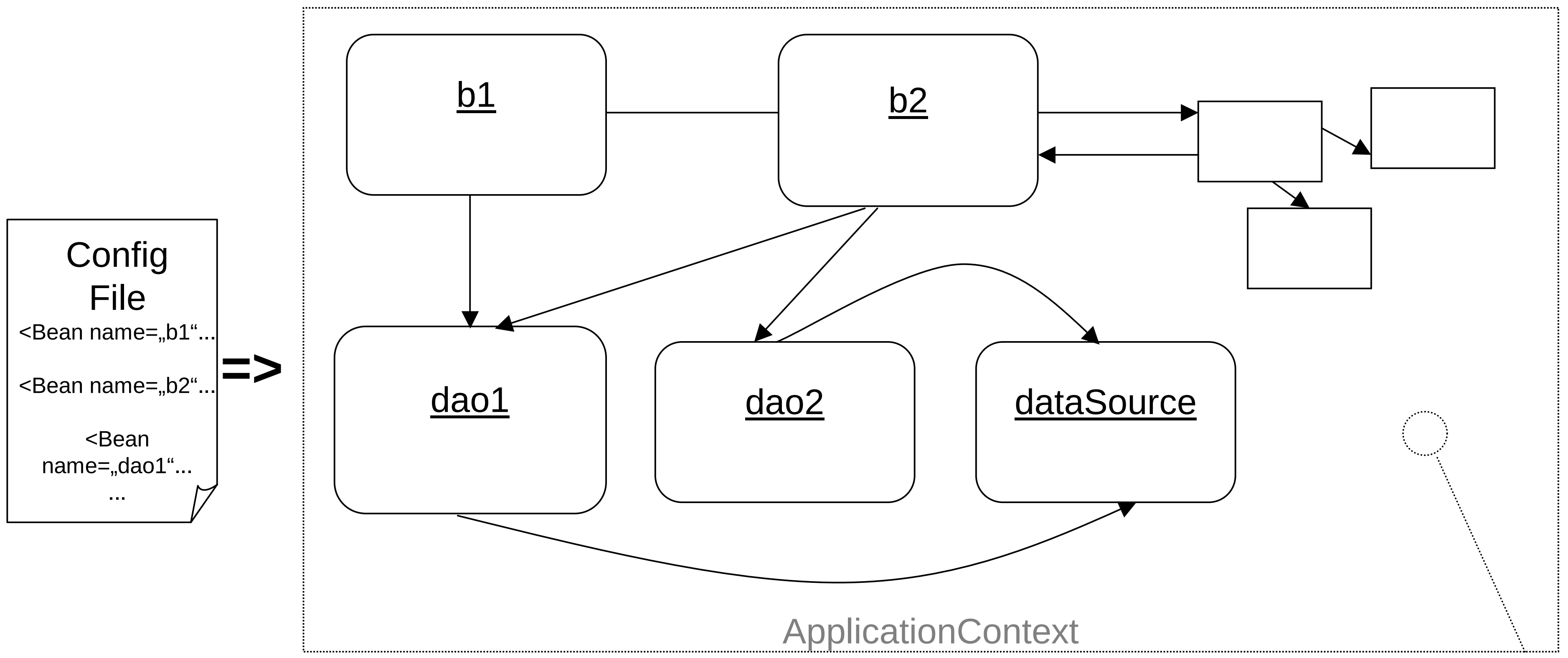
Spring: a J2EE extension framework

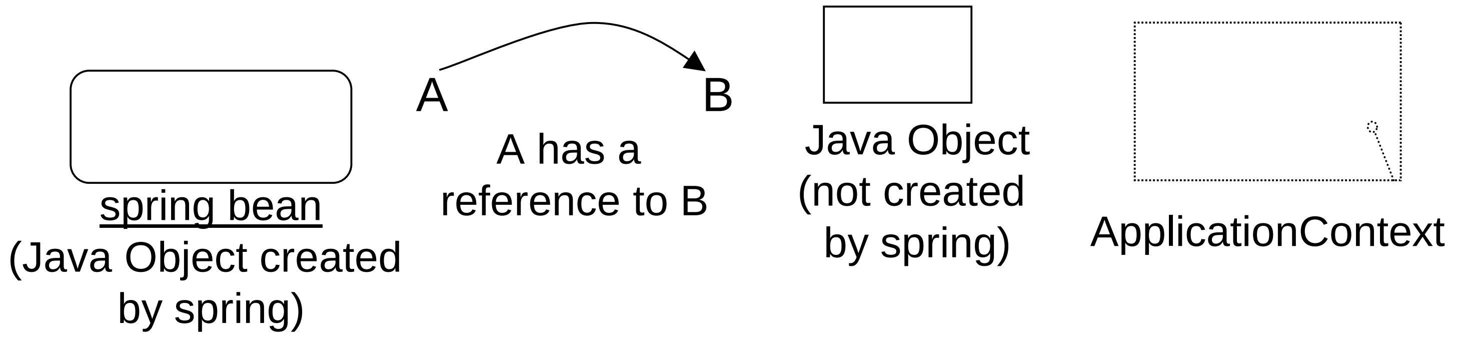
Introduction

More spring

Experience and benefits

Basic Spring: recapitulation





What java objects should be set up via Spring?

- We still create objects outside of spring!
- When to create objects with spring? When to create objects in plain java?

- Reasons why we would want to create an object via spring are:
  - It needs some configuration values and the configuration values may change over time
  - It needs to get a reference to some other objects/services or resources, such as a dataSource, a transactionManager
  - One may want ot use another implementation after compilation (i.e. we would like to create it via configuration)
  - We would like to add method interceptors to it (see following slides)

- Reasons why we not want to create an object via spring:
  - For simple tests (where config-indirection may be overkill)
  - For simple objects (e.g. Map, String, ...)

Too easy to be true?

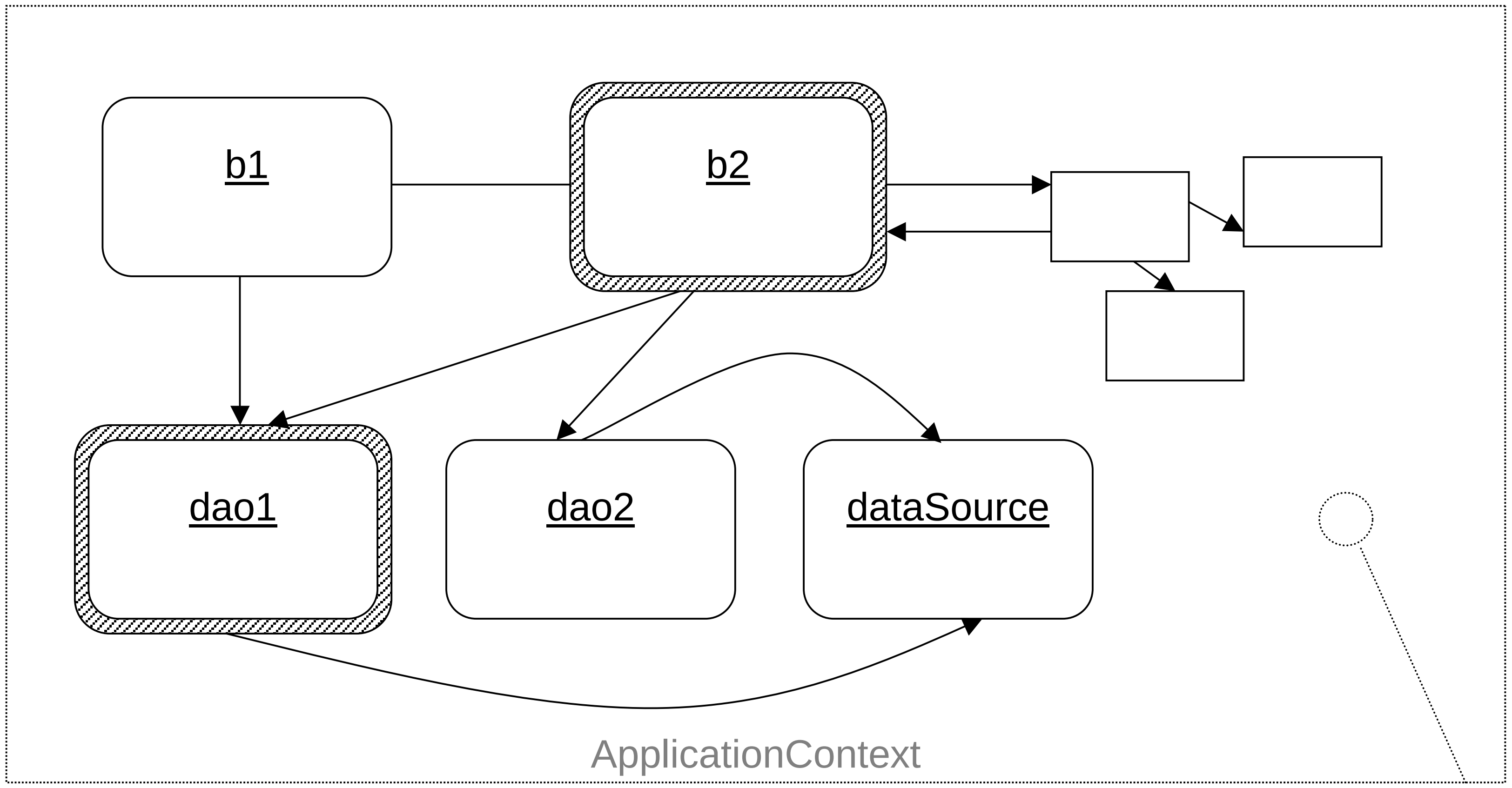
Current model: graph of explicitly setup singleton objects in 1 JVM

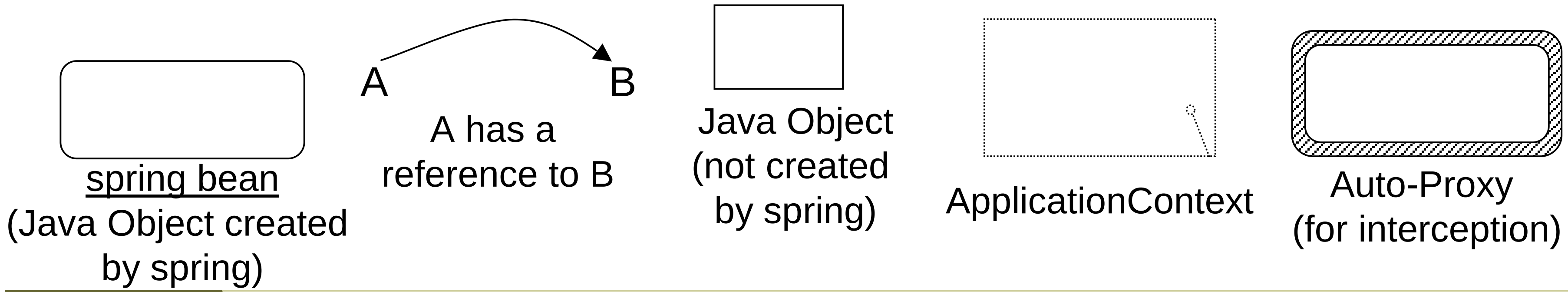
Sometimes this is not enough: e.g. 2 cases

- Java beans that are **NOT** singletons (there exists more than 1 instance per JVM)
  - Attribute of `<bean>` tag `singleton="false"`
- Java beans that are created indirectly
  - *Factory beans* (implement the FactoryBean interface)
  - E.g. JndiObjectFactoryBean needs a JNDI name, returns the object with that JNDI name

```
<bean id="dataSource"
    class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiName">
    <value>jdbc/jpetstore</value>
  </property>
</bean>
```

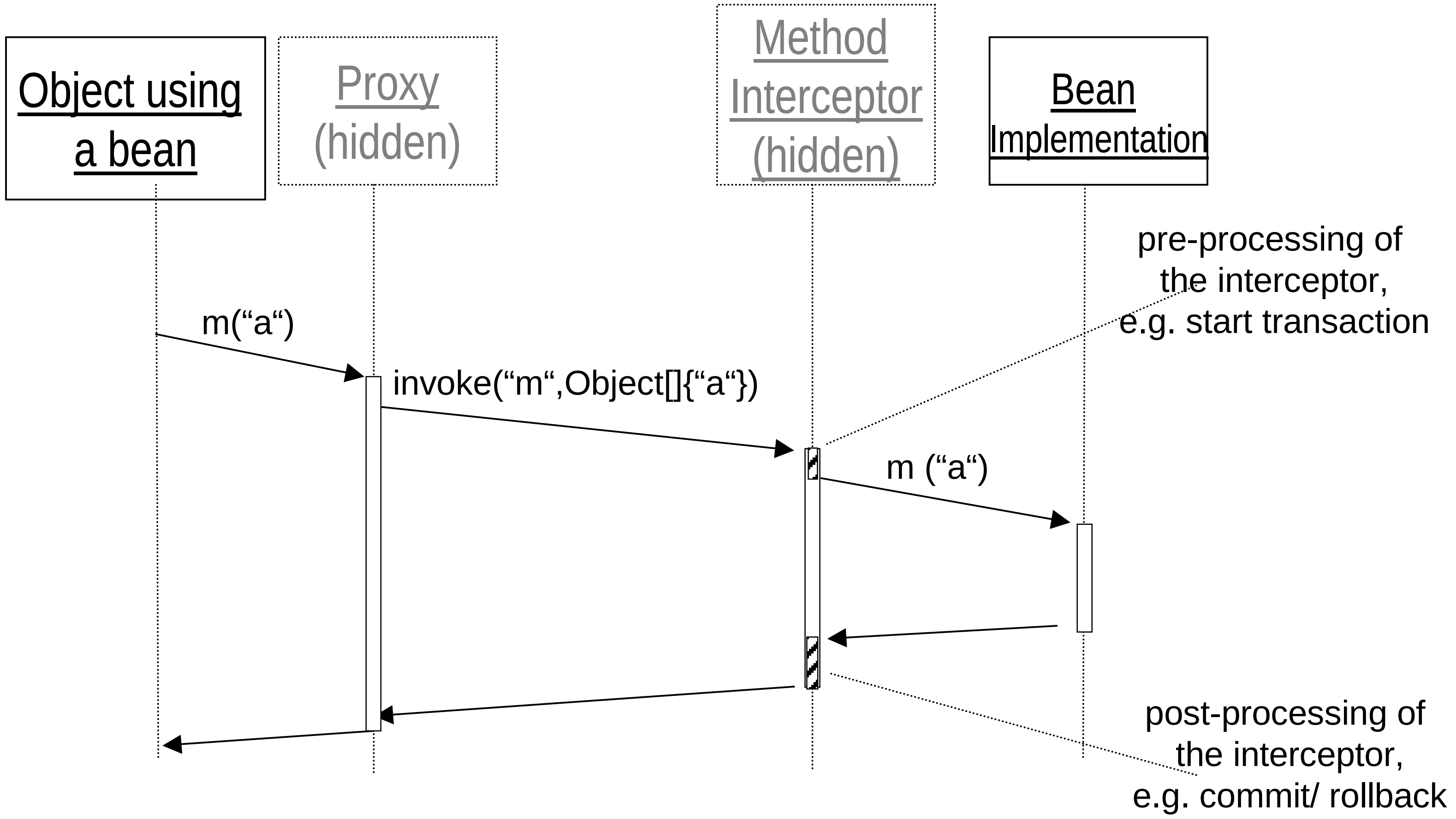
Bean proxies: adding an *indirection* to objects accesses




spring bean
(Java Object created by spring)
A
B
A has a reference to B
Java Object (not created by spring)
ApplicationContext
Auto-Proxy (for interception)

Method Interceptors to make this indirection handy

- Method Interceptor: „contains what should be done in the indirected method call“


Object using a bean
Proxy (hidden)
Method Interceptor (hidden)
Bean Implementation
pre-processing of the interceptor, e.g. start transaction
m(“a“)
invoke(“m“,Object[]{“a“})
m (“a“)
post-processing of the interceptor, e.g. commit/ rollback

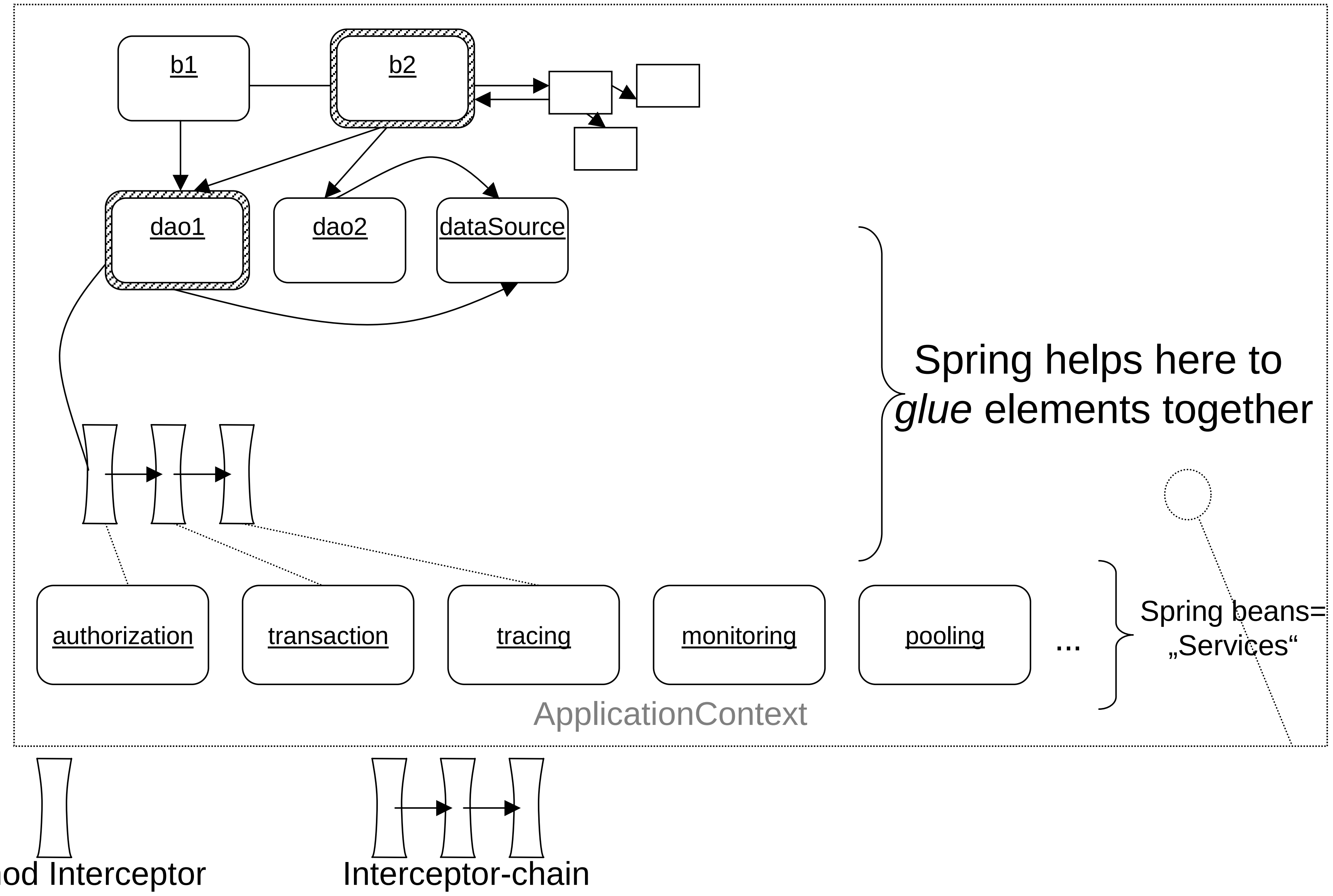
- Proxy and interceptor is not typically visible for the user of the spring bean
- A *chain* of interceptors is possible
- Original spring bean remains! `this.myMethod()` is not intercepted!

Setting up an interceptor (sample)

```
<bean id="jdkBeanNameProxyCreator"
     class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
      <value>userManagement,*Dao</value>
    </property>
    <property name="interceptorNames">
      <list>
        <value>authorizationInterceptor</value>
        <value>transactionInterceptor</value>
      </list>
    </property>
</bean>
```

Remark: this is not a concrete spring bean, but sets up the method interceptors on a group of beans.

Method Interceptors to change behavior of spring beans



How to *glue* functionality to spring beans?

- Explicitly setup in configuration
  - Setup an interceptor-chain on a set of beans

- Via Code annotations (Attributes/ Metadata/ Annotations)
  - JDK 1.5 metadata/ annotations
  - Javadoc tags (pre JDK 1.5)

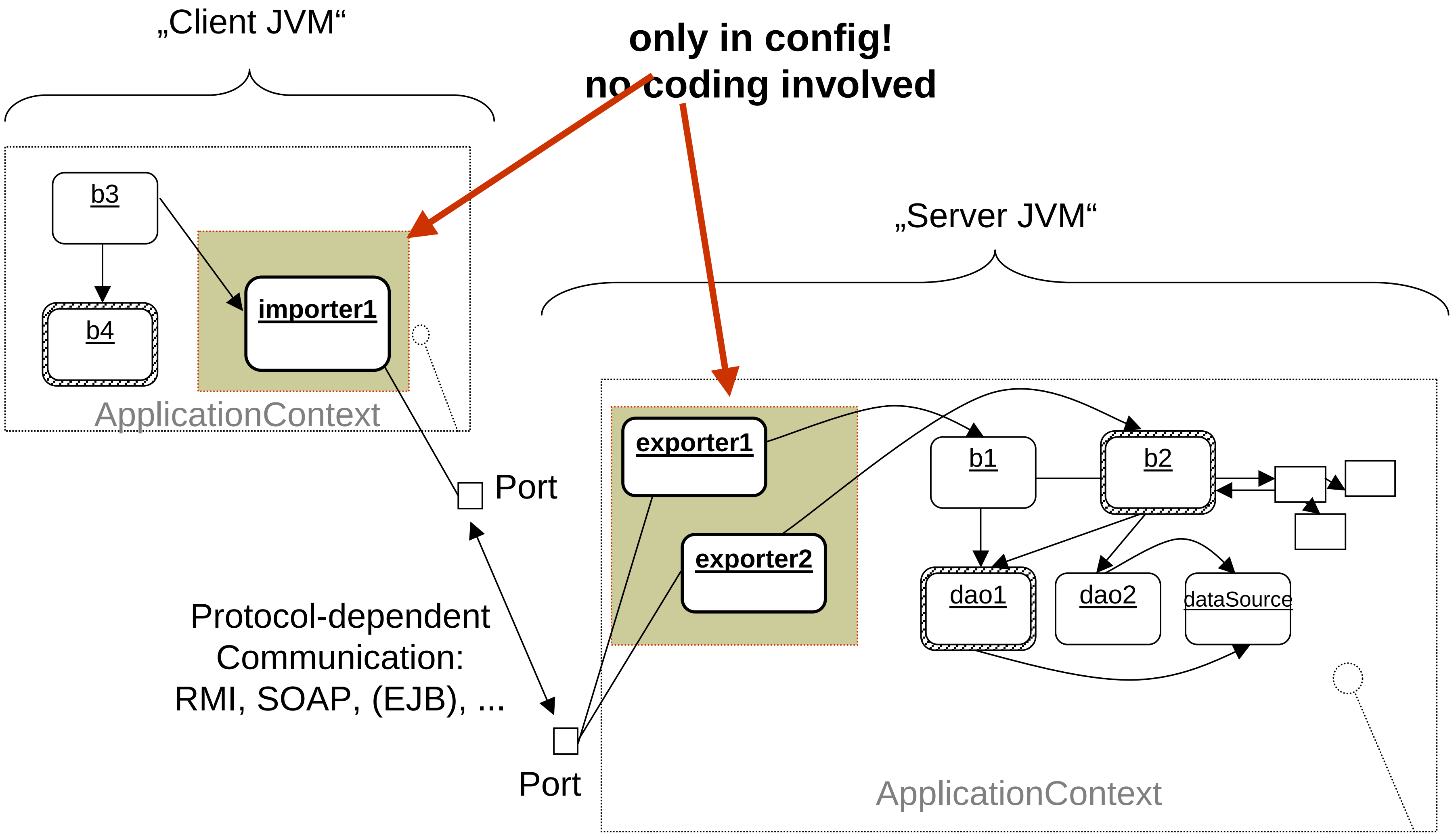
```
/**
* Normal javadoc comments...
* @@org.springframework.transaction.interceptor.RuleBasedTransactionAttribute()
* @@org.springframework.transaction.interceptor.RollbackRuleAttribute
  (Exception.class)
*/
public void echoException(Exception ex) throws Exception {
```

- Via API
  - Explicit calls to functionality
  - Setup an interceptor via code

Benefits of Method Interceptors

- We use method interceptors in the Java EE (in various implementations) for more than 5 years with a lot of success
  - For transactions, logging, exception handling, synchronization, performance measurement, caching, ...
- Main benefits
  - **Separation of concerns**: business concerns in normal code, cross-cutting technical concerns in interceptors
  - **Flexibly adaptable**: one chooses only the interceptors one needs
  - **Easy to reuse functionality**: an interceptor imposes almost nothing on the code that can use it
  - **Simple to understand and use**
- Remark: IMHO method interceptors are *the* essential feature of the more complex AOP of Spring (I will not go into details)
  - Spring AOP offers additionally: Mixins, AspectJ integration (for advanced needs)

Remoting


„Client JVM“
only in config!
no coding involved
b3
b4
importer1
ApplicationContext
„Server JVM“
Port
exporter1
exporter2
b1
b2
dao1
dao2
dataSource
Protocol-dependent
Communication:
RMI, SOAP, (EJB), ...
Port
ApplicationContext

Templates

Templates simplify usage of integrated technologies

„Hide all the nasty detail you don‘t want to be bothered about“

Supports common case, proven exception & resource handling

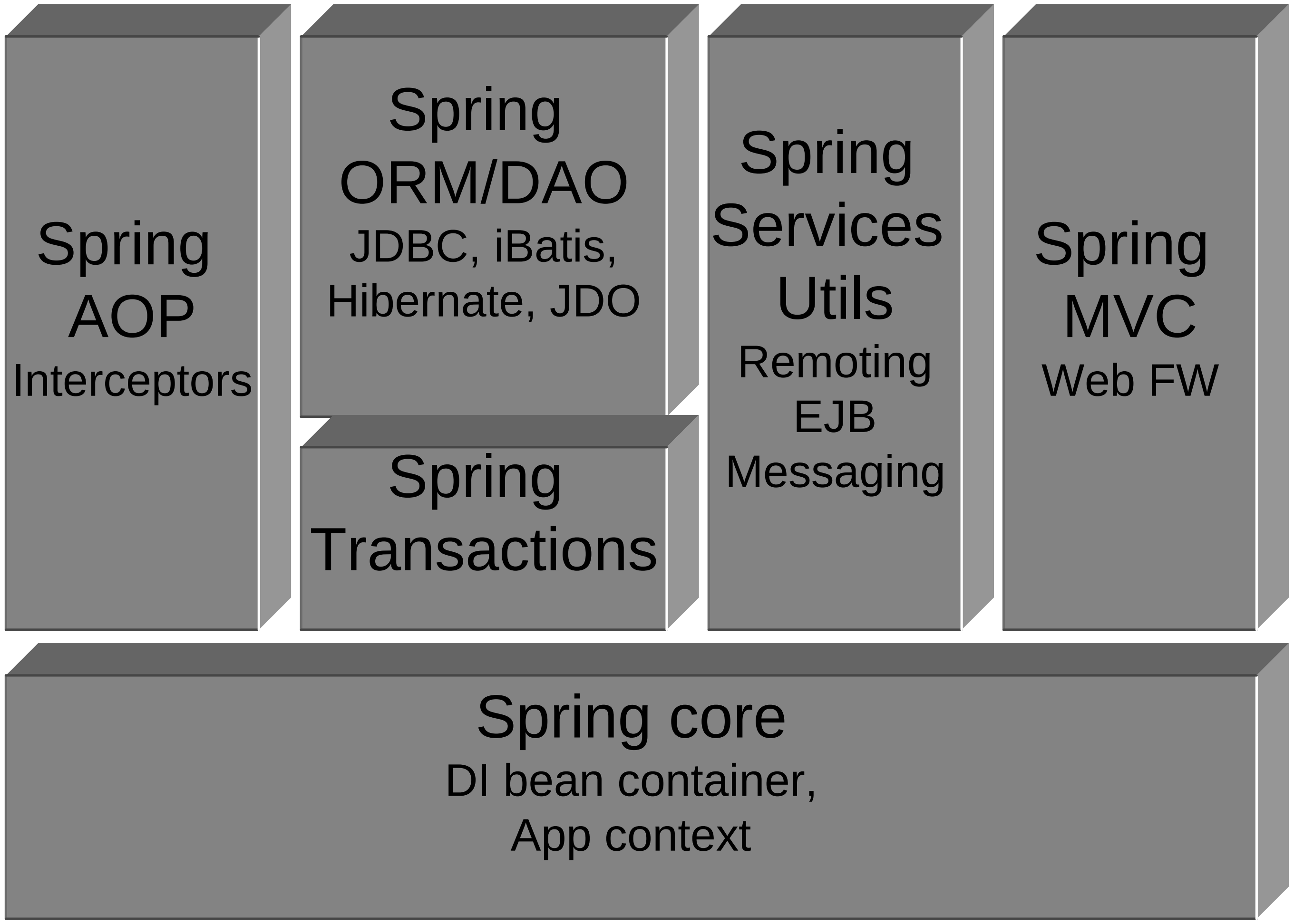
Spring provides templates for
- JDBC, JMS, Hibernate, ibatis, JDO, JMX, ...

Example: JDBC template

```
JdbcTemplate jdbc = new JdbcTemplate(dataSource);

jdbc.update("update EMPLOYEE set FIRST_NAME=? where LAST_NAME=?",
            new String[] {"Rick", "Hightower"});

int maxSalary = jdbc.queryForInt(
  "select max(Salary) from EMPLOYEE");

String name = (String)jdbc.queryForObject(
   "select FIRST_NAME from EMPLOYEE where LAST_NAME=‘Hightower'",
   String.class);
```

- saves up to 50% of „boring“ code: resource & error handling, mappings

How is Spring structured?



The different parts are well decoupled and can be used independently!

Practices often found with spring and related technology

Separate interfaces from implementation, program to interfaces

- Plug implementations via config into interfaces

Work really object-oriented, work with POJOs

- Avoid non-oo component models
- Try to avoid „fake“ objects such as DTOs, SLSB Home interfaces

Promote architectural choice

- Facilitate deployments in different contexts
- Allow substitution of layers with others (e.g. for tests)

Avoid distribution: only distribute if absolutely necessary

- There is no remote component model in the core of spring

Remarks on terminology

Lightweight container

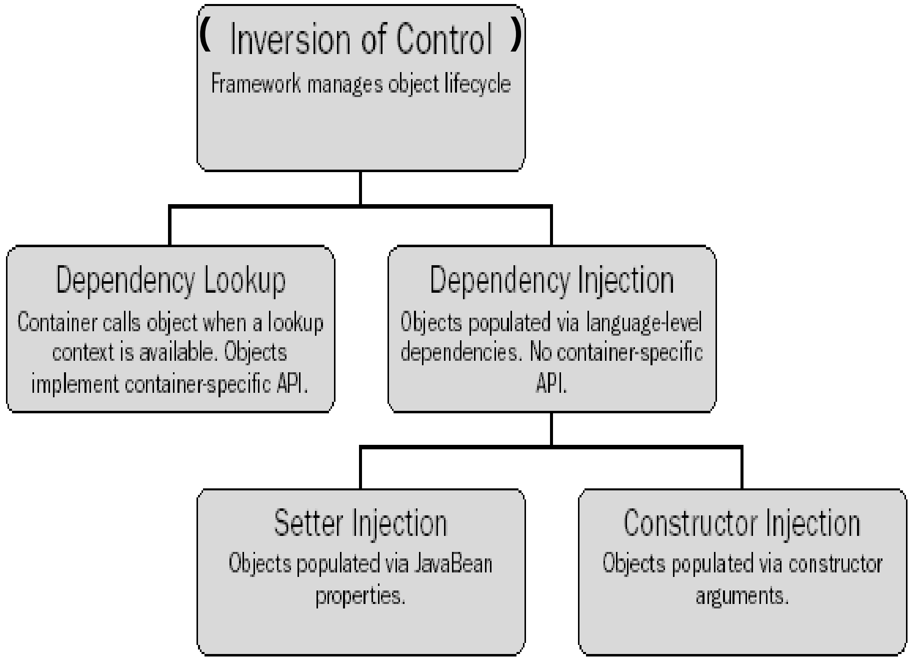
- As light as possible (code size, constraints, memory usage, ...)
- As opposed to a *heavy* container such as EJB

Dependency Injection

- The dependencies (other spring beans and parameters) are *pushed* into the beans (as opposed to the beans going to look for the dependencies)

Dependency Injection vs. IoC

- Some people speak of *Inversion of Control* (IoC)
- For Martin Fowler (and me) inversion of control is more general than Dependency Injection or Dependency Lookup, so we propose not to use it


(Inversion of Control)
Framework manages object lifecycle
Dependency Lookup
Container calls object when a lookup context is available. Objects implement container-specific API.
Dependency Injection
Objects populated via language-level dependencies. No container-specific API.
Setter Injection
Objects populated via JavaBean properties.
Constructor Injection
Objects populated via constructor arguments.

Spring: a J2EE extension framework

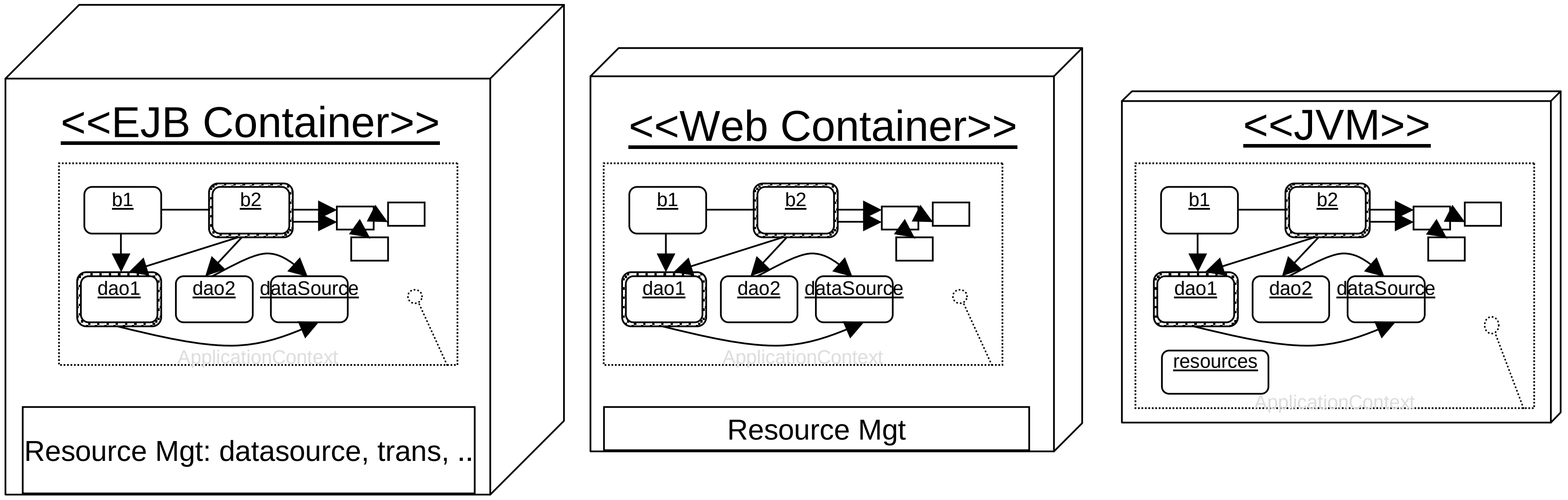
Introduction

More spring

Experience and benefits

Different deployments of an ApplicationContext

- A Spring application can be deployed in many different ways:



- What changes is the *plugging* of the resources: Datasources, Transaction Manager, Classloaders, ...
- Spring prepares applications for this: resource access is factored out of application

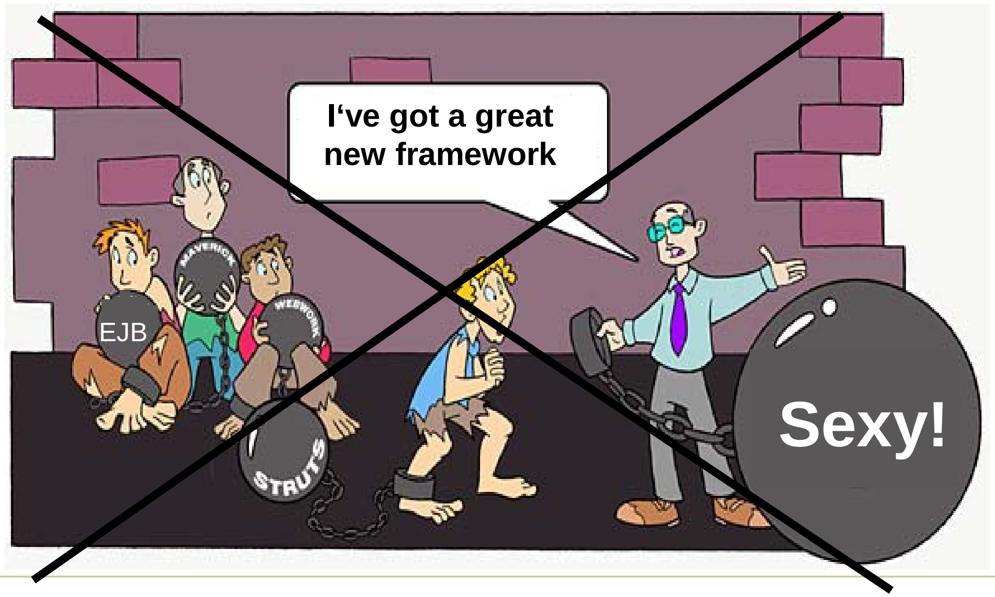
Benefits of Lightweight Dependency Injection Containers

Code lighter to manipulate (development, maintenance)

- One works with standard POJOs, no heavy container
  - Faster round-trips
  - Less concepts to handle

Minimizing constraints on your code

- Code does not depend on container (less lock-in)
  - Your code remains easier usable in other contexts, e.g. for tests or to embed
- Use other code without changes or integration
  - Typical Java code is directly usable



Picture source: theserverside.com

Spring experiences at ELCA

We have a framework team collecting Spring competences and providing punctual extensions to spring

For example:

- A light build system based on Ant
  - Module abstraction:
    module = code + config + transitive dependencies (modules + jars)
  - Plugins to extend it (Junit, Javadoc, Website, Emma, ...)
- Configuration improvements
- More flexible remoting
  - POJOs as EJBs, POJOs as SOAP servants
  - Implicit context passing
- Guidelines & Demos

- We plan to publish these punctual extensions as open source under the EL4J project

Spring experiences at ELCA (2)

Project experience with Spring/ EL4J

- ~10 projects, several of them already successfully deployed

Concrete experiences

- Solid backbone for configuration and plugging of applications
  - E.g. one fat-client application is now refactored for the web
- Spring typically solves issues „as one would wish they were solved“
- Practically no bugs in spring, excellent error messages
- Saves sometimes up to 50% of “boring” & error-prone code
- Very well integrated with essential technologies: Hibernate, ibatis, Struts, RMI, J2EE APIs
- Spring MVC (web-framework): more flexible than struts
- Sometimes spring can become complex (particularly AOP)
- Very well accepted by developers, short learning curve, fun to code!

=> Overall: very happy with Spring

References

Spring

- http://www.springframework.org/
- Article
  - http://www.theserverside.com/articles/article.tss?l=SpringFramework
  - http://www-128.ibm.com/developerworks/opensource/library/os-lightweight4/
- Books
  - Java Development with the Spring Framework, Wrox, Rod Johnson, et al.
  - Pro Spring, apress, Rod Harrop et al.
- Reference projects
  - Large list: CERN, many US banks, Telco providers, ...
  - http://www.springframework.com/users.html

Dependency injection

- http://martinfowler.com/articles/injection.html

EL4J

- http://www.elca.ch

ELCA

Thank you for your attention

For further information please contact:

Philipp H. Oser	Christian Gasser
Manager	CTO
+41 44 456 32 11	+41 21 613 21 11
Philipp.Oser@nospam.elca.ch	christian.gasser@nospam.elca.ch

TECHNOLOGY CONSULTING INNOVATION

ELCA