AOP in the Enterprise

Wim Vanderperren

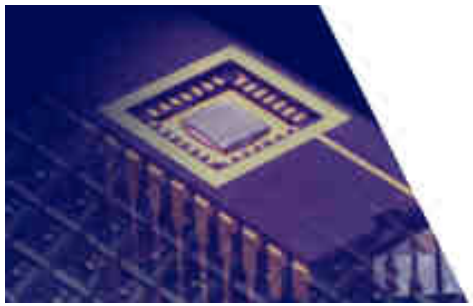
Overview

- CBSD
- EJB
- Spring (DI)
- Spring Multi-Tier Architecture
- Spring AOP

Software Components

- " A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties "

Formulated at the 1996 ECOOP conference
(Szyperski and Pfister, 1997)



Component Based Development

- The ultimate re-use scenario: application development as black-box assembly of off-the-shelf components
- The use of components is a law of nature in any maturing engineering discipline, the cost of component building can be spread out over multiple applications
- The concept of component software brings the middle path between custom-made and standard software: each component is a standardised product with all the advantages this brings while the process of component assembly allows for significant customisations
- Software components are binary units of independent production, acquisition, and deployment
- Software differs from other engineering discipline as it is the blue print that is delivered rather than the realisations of it

Interfaces and Explicit Context Dependencies

- The interface of a component defines its access points, clients (usually other components) access services provided by a component through these access points
  - **Syntax:** specification of the *provided* interfaces using some standard Interface Description Language
  - **Semantics:** (formal) specification of the functionality of each service provided (e.g. pre- and post-conditions)
  - **Synchronisation:** (formal) specification of expected or imposed ordering, grouping and mutual exclusion of services provided
  - **Quality of Service:** specification of guaranteed response times, upperbounds for resource consumption (CPU-time, memory, etc.), failure rates, mean time between failure, etc.
- The component must specify what the deployment environment must provide for the component to be able to function properly
  - Required interfaces of other components.
  - Since there are multiple component world emerging, components must also mention the world they are prepared for (I.e. platform, implementation language, component model, component and library versions etc.)

Component "Weight"

In CBSD components have to be loosely coupled

Fat Components

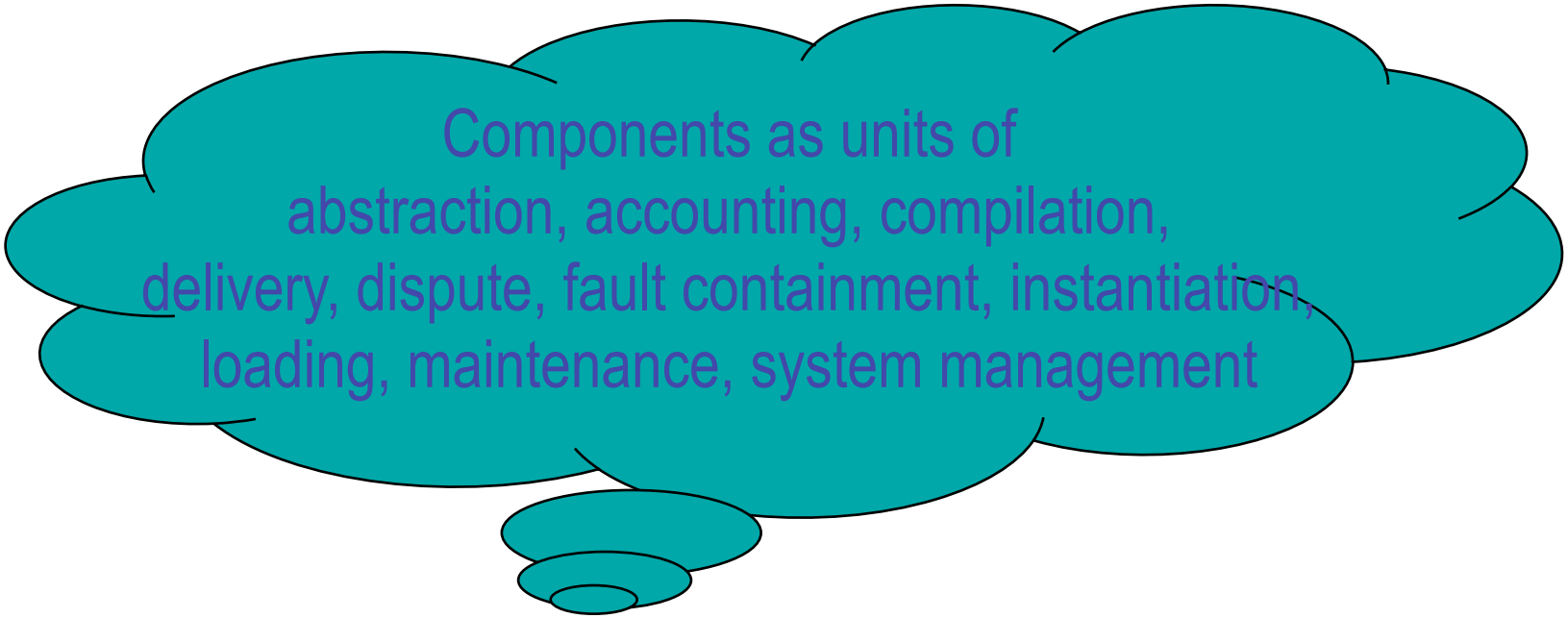
- The component is self-contained and can function under weak environmental guarantees
- The context dependencies are reduced making the component more robust over time
- But a component with everything bundled in is not a component anymore

Lean Components

- Other components are (re)-used to achieve the component's services
- The context dependencies increase making the component more vulnerable in case of context evolution
- Re-use is maximized, use is compromised

Scale and Granularity

- A component's size may vary from a single class or function to a complete subsystem
- Most of the aspects relevant to granularity seem to demand fairly coarse grained partitionings

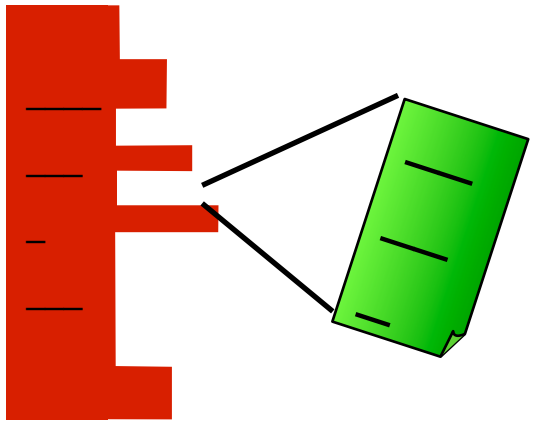


Component Composition or Wiring (1)

- Most class libraries and frameworks are not components in the strict sense, they are delivered in source form and implementation inheritance is the common re-use mechanism which is typically white-box reuse

Component Composition or Wiring (1)

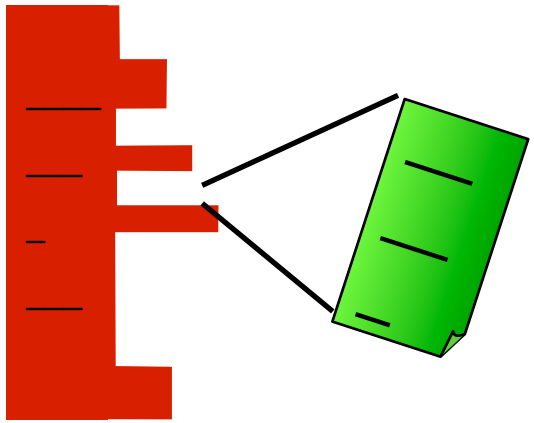
- Most class libraries and frameworks are not components in the strict sense, they are delivered in source form and implementation inheritance is the common re-use mechanism which is typically white-box reuse



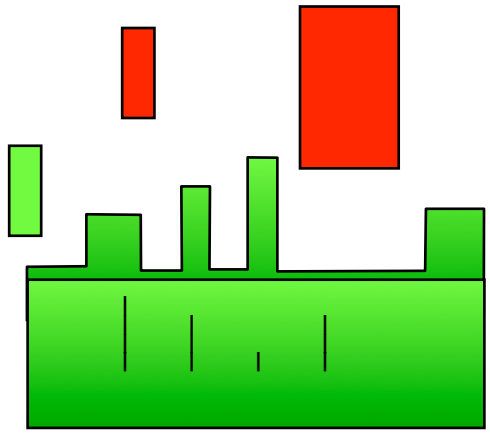
Library approach

Component Composition or Wiring (1)

- Most class libraries and frameworks are not components in the strict sense, they are delivered in source form and implementation inheritance is the common re-use mechanism which is typically white-box reuse



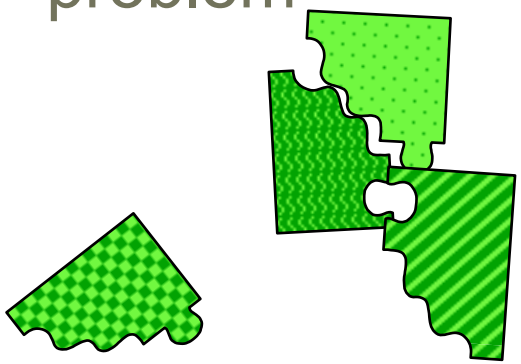
Library approach



Framework approach

Component Composition or Wiring (2)

- Software components are third party configurable: blackbox reuse with plug-and-play composition is aimed for
- In practice, lots of glue code needs to be written to make components work together
- Distribution and heterogeneaty aggreviates the problem



Utopic approach: plug and play

In practice: glue code

Middleware = Component Infrastructure Technology

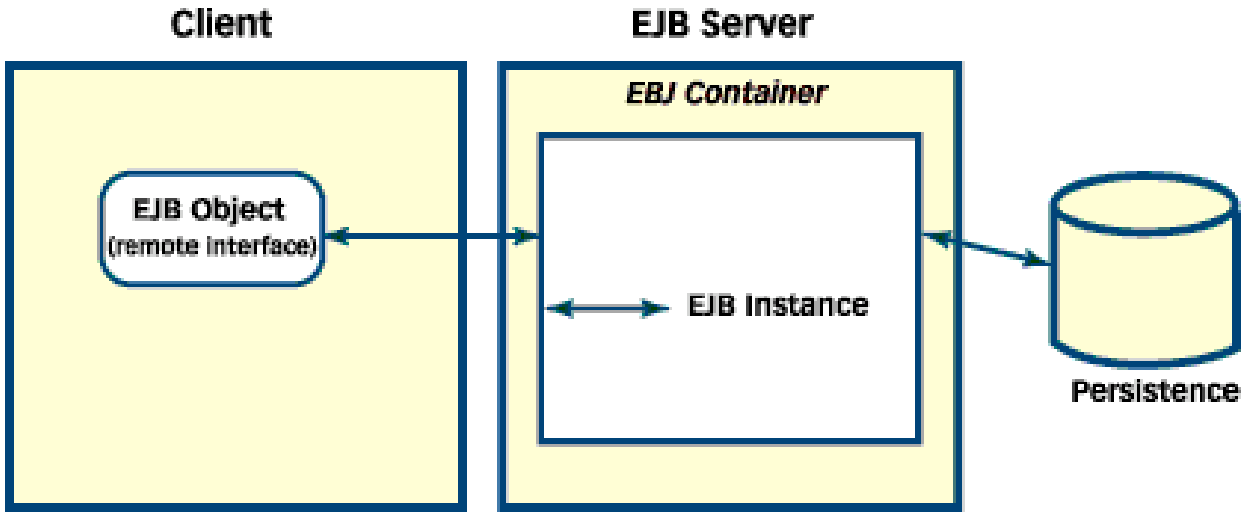
- Support the interoperation of components over different platforms
- Find out what components are currently connected to the infrastructure
- Make reference to other components via some meaningful naming scheme
- Guarantee once-only delivery of messages between components
- Manage transactions consisting of multiple interactions among components
- Allow secure communication between components

J2EE (now Java EE) middleware for the Java world

- Industry standard for developing portable, robust, scalable, multi-user, and secure server-side Java applications
- Builds on the **Enterprise Java Beans** component model
- EJB is designed to make application creation easy, I.e free programmers from details of managing transactions, thread, load balancing, etc.
- Allows to combine components from different vendors, to combine with non-Java applications and interoperates with Corba

EJB basics (1)

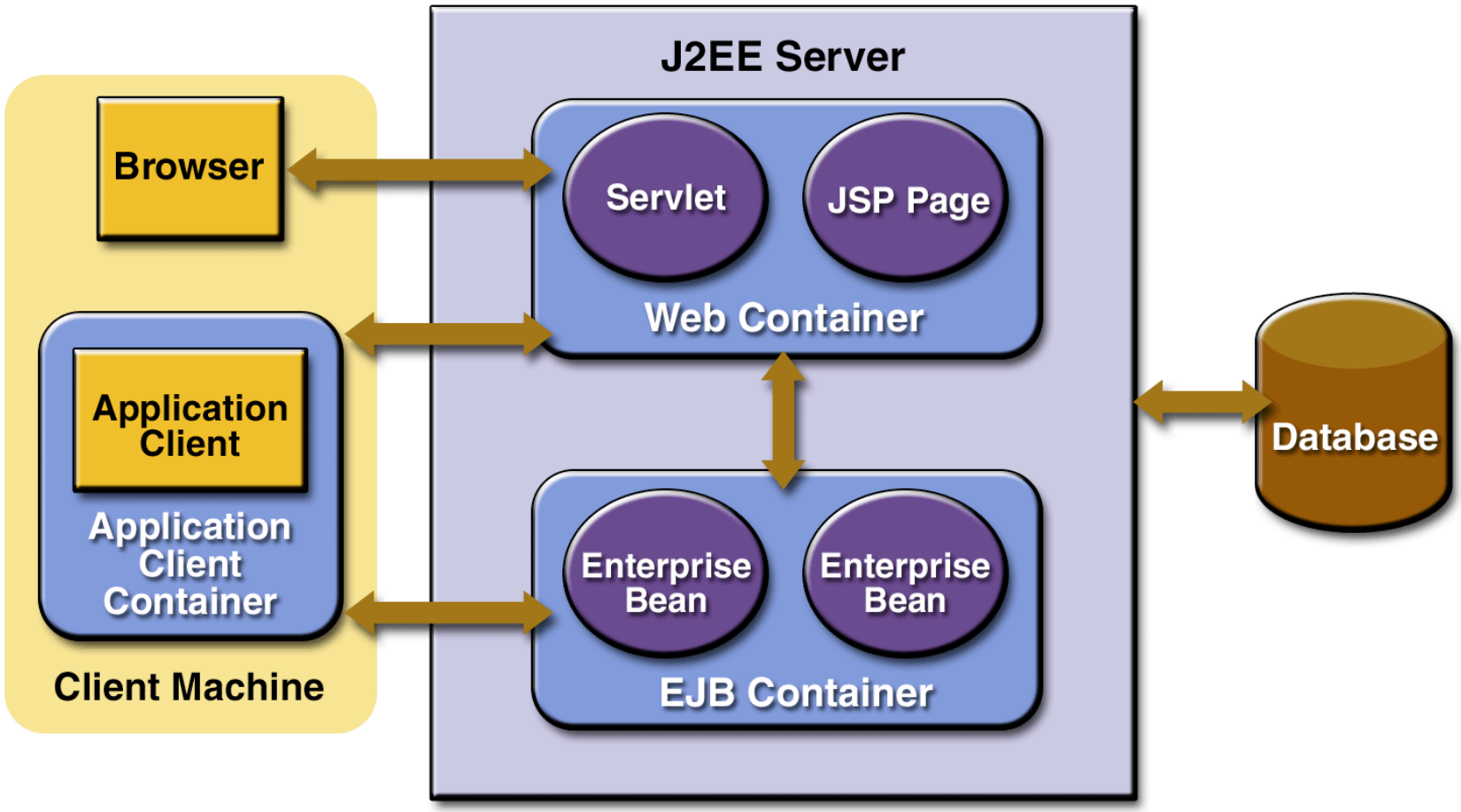
- **EJB component:** A Java class written by a developer, implements business logic, lives in a EJB container that runs on a EJB server
- **EJB container:** Resides on the server and provides services such as transaction and resource management, versioning, scalability, mobility, persistence…
- **EJB object and the remote interface:** An EJB object resides on the client and remotely executes the the EJB components's methods (proxy). *(The EJB object is created by code generation tools that come with the EJB container).*



EJB basics (2)

- **Two types of Enterprise JavaBeans**
  - **Session Beans:**
    - Associated with a single client
    - Typically not persistent, will not survive server crashes
  - **Entity Beans:**
    - Represent information persistently stored in a database
    - Associated with database transactions
- **The home interface**
  - Each EJB component has a home interface that defines methods for creating, destroying and (in case of entity beans) locating EJB instances
  - The EJB container is responsible for the life-cycle of server-side objects, e.g. a client request a container to create an instance of a particular EJB component and the container installs an instance and returns an EJB object to manipulate the instance
  - The Java Naming and Directory Interface (JNDI) is used by clients to locate the home interface for the class of beans it wants to use

J2EE global architecture



EJB Example

- Main bean implementation
  - Cluttered with lifecycle methods
  - Where is the business logic?
  - Not usable outside container
    - Not easily testable

```
public abstract class Counter implements EntityBean{
    private EntityContext context = null;
     public abstract Long getID( );
     public abstract void setID(Long id);
     public abstract int getCount( );
     public abstract void setCount(int count);

   public Object ejbCreate(Long id, int count);
      throws CreateException {
      setId(id);
      setCount(count);
      return null;
     }
     public void ejbPostCreate(Long id, int count)
      throws CreateException { }
     public void setEntityContext(EntityContext c) {
      context = c;
     }
     public void unsetEntityContext( ) {
      context = null;
     }
     public void ejbRemove( ) throws RemoveException { }
     public void ejbActivate( ) { }
     public void ejbPassivate( ) { }
     public void ejbStore( ) { }
     public void ejbLoad( ) { }
[3]     public void increment( ) {
      int i=getCount( );
      i++;
      setCount(i);
    }
     public void clear( ) {
      setCount(0);
     }
}
```

EJB Example (2)

- Local Interface

```
public interface CounterLocal extends EJBLocalObject {
  public abstract Long getID( );
  public abstract void setID(Long);
  public abstract int getCount( );
  public abstract void setCount(int count);

}
```

- Home Interface

```
public interface CounterLocalHome extends EJBLocalHome {
        public Collection findAll( ) throws FinderExceptionpublic;
        CounterLocal findByPrimaryKey(Long id) throws FinderException;
        public CounterLocal create(Long id, int count)
                throws CreateException;
}
```

- Deployment descriptor
- DTO's
- ……

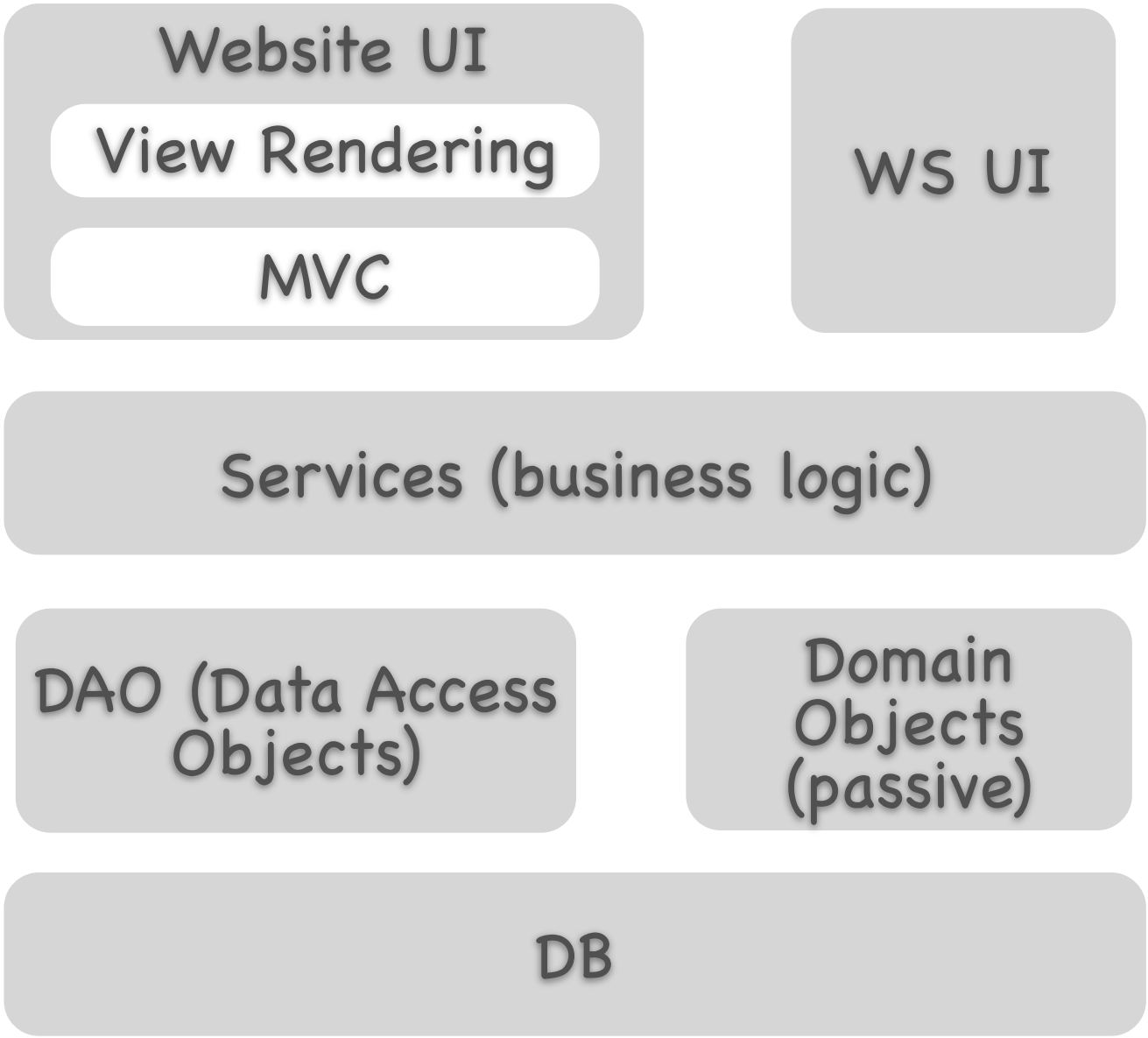
Do I reallly need to continue?

Spring to the rescue

Spring

- A Layered Java Application framework
- Plain POJO beans instead of EJB
- Dependency injection instead of lookup
- Convention over configuration
- Abstraction layers for external APIs
- Compatible with a large range of application servers

Typical Spring Architecture



POJO Bean??

- Plain Old Java Object
- But with a couple of naming conventions:
  - A setter for property *prop* is named *setProp*
  - A getter for property *prop* is named *getProp*
- Example:

```
public class MyComponent {
  private String name;

  public String getName() { //the getter
    return name; }

  public void setName(String name) { //the setter
    this.name=name; }
}
```

Dependency Injection?

- References and properties are injected by the container
- Container follows the composition specified in a Spring Beans Configuration file (XML)

Dependency Injection? (2)

```
//Plain POJO Component
public class MyService {
  private IMailService mailService;
  public void register() {
      ....
      mailService.sendMail(address,"registration successful");
  }
}

//Spring configuration:
<beans>
  <bean id="mailServiceC" class="org.vub.mytool.MyMail">
    <property name="host" value="smtp.vub.ac.be"/>
  </bean>
  <bean id="myService" class="org.vub.mytool.MyService">
    <property name="mailService" ref="mailServiceC"/>
  </bean>
</beans>
```

Autowiring Dependencies

- The Spring container can figure out depencies automagically.
  - By name
  - By type
  - or both

```
<beans>
  <bean id="mailServiceC" class="org.vub.mytool.MyMail">
    <property name="host" value="smtp.vub.ac.be"/>
  </bean>
  <bean id="myService" class="org.vub.mytool.MyService"
      autowire="autodetect">
  </bean>
</beans>
```

Instantiation Strategy

- Spring instantiates beans lazily on first use
  - to avoid: lazy-init=false
- Bean instantiation will cause instantiation of all dependencies
- In case there is a hidden dependency use depens-on

```
<beans>
  <bean id="mailServiceC" class="org.vub.mytool.MyMail"
      depends-on="myDatabase">
    <property name="host" value="smtp.vub.ac.be"/>
  </bean>
  <bean id="myService" class="org.vub.mytool.MyService"
      lazy-init="autodetect" lazy-init="false">
  </bean>
</beans>
```

Starting a Spring App

- Via a Spring-based or Spring Compatible GUI framework
- Or manually, by creating the appropriate application context (Spring container):
  - ClassPathXmlApplicationContext
  - FileSystemXmlApplicationContext
  - WebApplicationContext (only via servlets)
  - ....

```
ApplicationContext ctx =
  new ClassPathXmlApplicationContext("conf/appContext.xml");
ctx.getBean("myBeanName");
```

Spring Resources

- Spring Resource abstraction is recognized automatically in configuration.
  - Find in classpath:
    - classpath:com/myapp/config.xml
  - Find in file system:
    - file:/data/config.xml
  - Find on the internet:
    - http://myserver/logo.png
  - Depends on application context:
    - /data/config.xml

Spring Beans Scoping

- Per default every Spring Bean is a singleton
- Other scopes:
  - prototype: new instance on each request
- Only relevant in web context:
  - request: new instance for every http request
  - session: new instance for every http session
- E.g.:

```
<bean id="loginAction" class="com.foo.LoginAction"
    scope="request"/>
```

Spring Beans Scoping (2)

- Watch out when injecting differently scoped beans
  - e.g. injecting a request scoped bean into a singleton scoped bean
- Per default, injection happens once, so only one instance is injected
- Typically you want the injected bean to be updated to reflect the current context
  - e.g. the bean associated with the current http context
  - Solution: proxies

Spring Beans Scoping (3)

```
<!-- a HTTP Session-scoped bean exposed as a proxy -->
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session">

        <!-- this next element effects the proxying of the surrounding bean -->
        <aop:scoped-proxy/>
</bean>

<!-- a singleton-scoped bean injected with a proxy to the above bean -->
<bean id="userService" class="com.foo.SimpleUserService">

     <!-- a reference to the proxied 'userPreferences' bean -->
      <property name="userPreferences" ref="userPreferences"/>
</bean>
```

Other Spring config stuff

- Extensive XML language to e.g. define lists, collections etc...
- Constructor injection as alternative to setter injection
  - But beware of cycles!
- Inner anonymous beans
- Automatic unresolved dependency checking

Structuring Spring

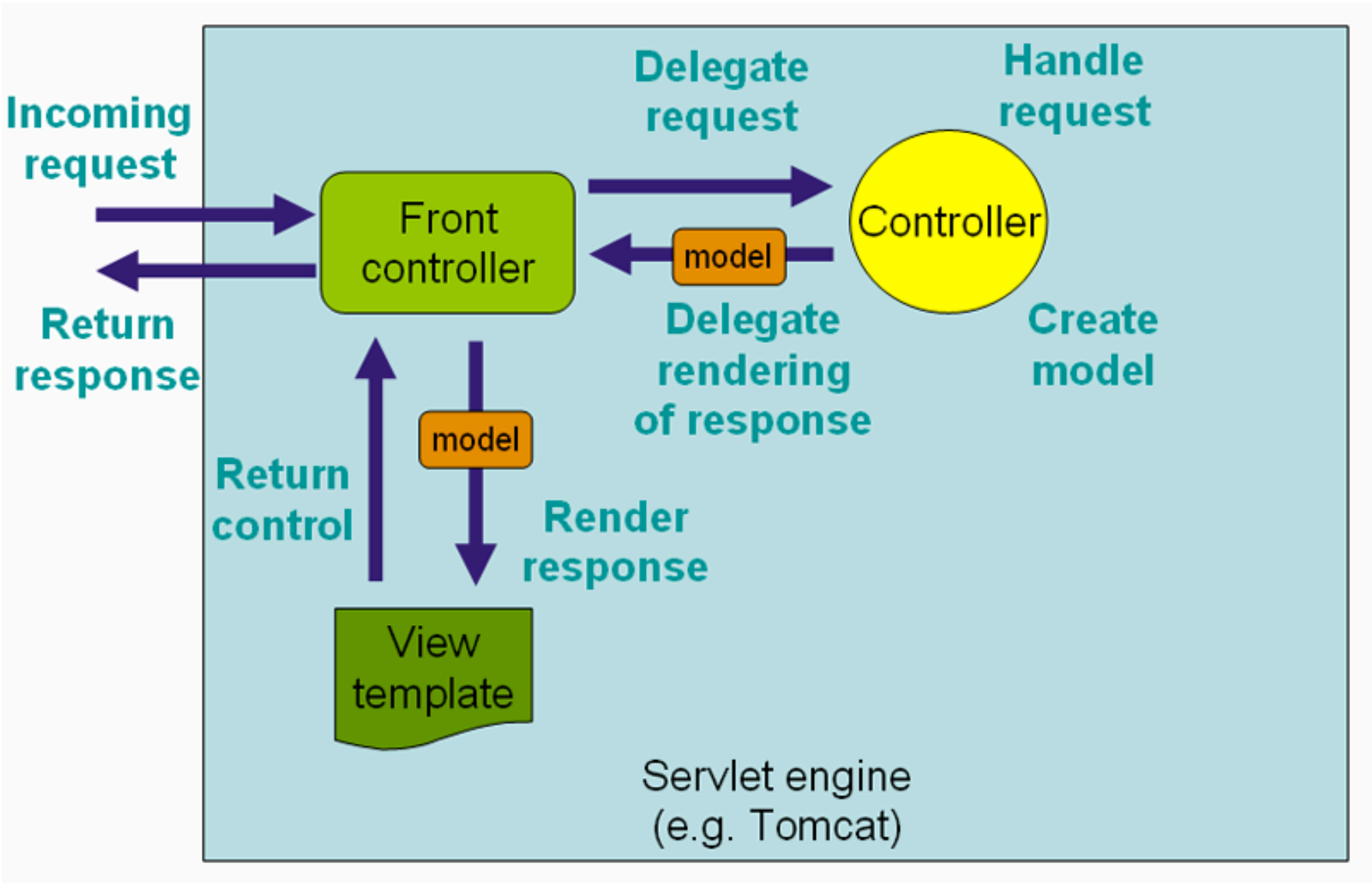
- Typical one config file per package
- Importing config files in deployment/test configs

```
<import resource="classpath:ssel/banking/dao/
banking.hsqldb.xml"/>

<import resource="classpath:ssel/banking/dao/banking.db.xml"/>

<import resource="classpath:ssel/banking/dao/jpa/
banking.dao.xml"/>
```

Spring MVC


Incoming request
Delegate request
Handle request
Front controller
Controller
model
Return response
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)

Spring MVC practical

- Controllers implement the Controller interface
- Model classes can be any java bean
- Views are typically implemented using JSP

Spring MVC: Welcome

```
class WelcomeController extends AbstractController {

    @Override
    protected ModelAndView handleRequestInternal(HttpServletRequest rq,
            HttpServletResponse rs) throws Exception {

            return new ModelAndView(getViewName());
     }

}
```

```
<bean name="/welcome.htm" class="ssel.banking.web.controller.WelcomeController">
</bean>
```

Spring MVC: Welcome (2)

- The view, in welcome.jsp:

```
<P>Welcome to JabobBank's online internet bank!
<P>You first have to login before you can access your accounts:
<P>
<a href="./login.htm">Login here </a>
<P>If you do not have a login yet, please contact our customer support service.
```

Spring MVC: login (1)

```
public class LoginController extends AbstractFormController {

    private IUserService userService;

    @Override
    protected Object formBackingObject(HttpServletRequest request)
            throws Exception {

        return new User();
    }


    @Override
    protected ModelAndView onSubmit(HttpServletRequest request,HttpServletResponse
          response, Object command, BindException errors) throws Exception {

        ModelAndView mv;

        if(!getUserService().authenticate((User)command))
            mv=new ModelAndView(getSuccessView());
        else mv = new ModelAndView(getFailureView());

        mv.add(user);  //idem as mv.add("user",user);
        return mv;
    }
     ....
}
```

Spring MVC: login (2)

```
<bean name="/login.htm" class="ssel.banking.web.controller.LoginController">
        <property name="successView" value="loginsuccess" />
        <property name="failureView" value="loginfailure" />
        <property name="userService" ref="userService" />
</bean>
```

- The view, in login.jsp:

```
<form:form>

<form:label path="email">Email Address:</form:label>
<form:input path="email" />
<form:errors path="email" />

<form:label path="password">Password:</form:label>
<form:password path="password" />
<form:errors path="password" />

<button type="submit">Login</button>

</form:form>
```

Spring and AOP

- Spring explicitly supports AspectJ AOP
- Aspects can be configured like normal Spring components (dependency injection)
- Supported syntax:
  - XML-based definition
  - AspectJ annotation-based development style
  - AspectJ language
  - Domain Specific Languages for e.g. Transaction Management
- Aspect library

Spring AOP Weavers

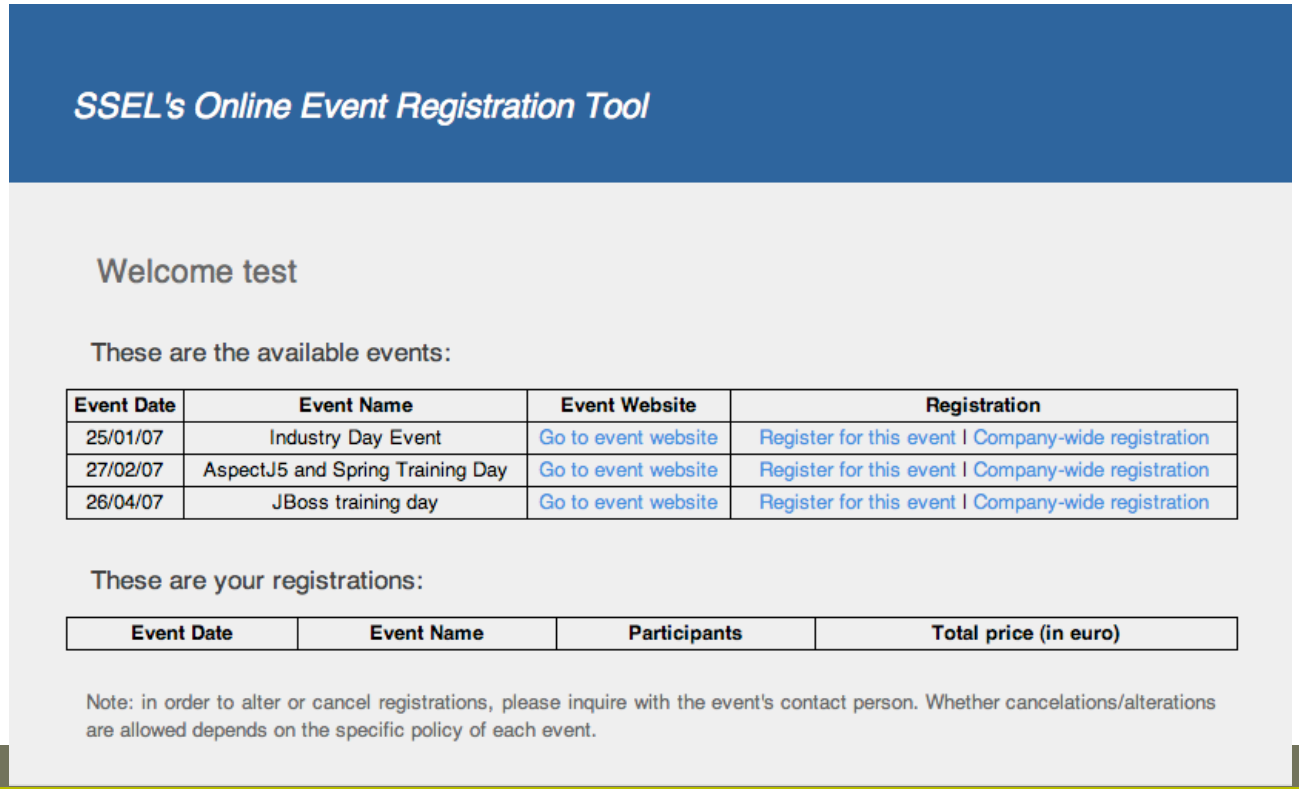
- AspectJ weaver or built-in Spring weaver
- Built-in Spring weaver:
  - No external tools
  - Weaving happens automagically
  - Proxy-based:
    - only weaving on configured beans
    - as such domain classes are typically excluded from weaving
  - Only supports execution pointcuts
    - No call, field set, field get etc...

Spring/AOP Syntax & Weavers

	AspectJ Language	AspectJ Annotation Style	XML Definition	DSL
Spring Weaver	No	Yes	Yes	Yes
AspectJ Weaver	Yes	Yes	No	No

Spring/AOP: case study

- Event Registration Tool
- Two aspects:
  - Discount Business Rule (Annotation-Style)
  - Dependency Injecting Domain Objects (Library Aspect)



SSEL's Online Event Registration Tool

Welcome test

These are the available events:

Event Date	Event Name	Event Website	Registration
25/01/07	Industry Day Event	Go to event website	Register for this event I Company-wide registration
27/02/07	AspectJ5 and Spring Training Day	Go to event website	Register for this event I Company-wide registration
26/04/07	JBoss training day	Go to event website	Register for this event I Company-wide registration

These are your registrations:

Event Date	Event Name	Participants	Total price (in euro)

Note: in order to alter or cancel registrations, please inquire with the event's contact person. Whether cancelations/alterations are allowed depends on the specific policy of each event.

Discount Business Rule

- Registration for two events allows a discount
- Not anticipated
- Crosscuts the service, domain and presentation layers

Discount Business Rule

```
//Injecting discount info into Domain Objects
@DeclareParents(value="regtool.model.AbstractRegistration",
      defaultImpl=DefaultDiscountImpl.class)
private IDiscount discountInterface;

//Intercepting price computation to add discount
@Around(value="execution(* regtool.service.RegToolService.computeCost(..))
     && args(reg,user,event)")
public Object computePrice(ProceedingJoinPoint thisJoinPoint, AbstractRegistration
reg, User user, Event event) {
 ..... //compute discount and add discount to reg domain object
}

//Inserting the discount in the view (Spring MVC Controllers)
@AfterReturning(
    value="execution(*
         org.springframework.web.servlet.mvc.AbstractFormController+.referenceData
                  (javax.servlet.http.HttpServletRequest, java.lang.Object,..))
         && within(regtool.web.controller.*)
         && args(request,registration,..)",
    returning="referenceData"
)
public void insertDiscount(HttpServletRequest request,
          AbstractRegistration registration, Map referenceData) {
   IDiscount discount = (IDiscount) registration;
   if(discount!=null&&discount.hasDiscount())
      referenceData.put("discount", discount);
}
```

Discount Business Rule

```
//Spring Aspect Configuration (DI)
<bean id="discountAspect"
        class="regtool.service.br.BulkDiscountAspect"
        autowire="autodetect" factory-method="aspectOf">
    <property name="discountPercentage" value="0.40"/>
    <property name="viableEvents">
        <list>
        <value>1</value>
        <value>2</value>
        </list>
    </property>
    <property name="message" value="discount when
        registering both the AspectJ and JBoss Training Days"
     />
</bean>

<aop:aspectj-autoproxy/>
```

Discount Business Rule

```
//enabling the Spring Weaver (listing aspects only needed in
//      mixed AspectJ weaver/Spring weaver config)
<aop:aspectj-autoproxy>
  <include name="discountAspect"/>
  <include name="myOtherAspect"/>
</aop:aspectj-autoproxy>
```

```
//disabling the Spring Weaver for Discount Aspect
<aop:aspectj-autoproxy>
  <include name="myOtherAspect"/>
</aop:aspectj-autoproxy>
```

Dependency Injection of Domain Objects

- Dependency injection only for Spring managed beans
- Domain objects are typically managed by the ORM framework
- Spring ships with "AnnotationBeanConfigurerAspect"
  - @Configurable Domain Objects have dependency injection
  - Aspect intercepts constructor of domain object

AnnotationBean-ConfigurerAspect

```
//creation of any object that we want to be configured by Spring
pointcut configuredObjectCreation(Object  newInstance,
                Configurable cAnnot)
      : initialization((@Configurable *).new(..)) &&this(newInstance) &&
        @this(cAnnot);

//ask Spring to configure the newly created instance
after(Object newInstance, Configurable cAnnot) returning
  : configuredObjectCreation(newInstance, cAnnot) {
    String beanName = getBeanName(newInstance, cAnnot);
    beanFactory.applyBeanPropertyValues(newInstance,beanName);
}


....
```

Dependency Injection of Domain Objects

```
//Domain object implementation
@Configurable("account")
public class Account {
   ...
}

//Spring configuration
<aop:spring-configured/> <!-- THIS IS A LIBRARY ASPECT -->

<bean id="account" class="com.xyz.myapp.domain.Account"
     scope="prototype">
 <property name="fundsTransferService" ref="transferBean"/>
  ...
</bean>
```

Dependency Injection of Domain Objects

```
//Domain object implementation with auto wiring
@Configurable(autowire=Autowire.BY_TYPE,dependencyCheck=true))
public class Account {
   ...
}
```

```
//Spring configuration with auto detection
<aop:spring-configured/>
```

Custom AspectJ Aspect Configuration

- AspectJ aspect configured by Spring
- But weaved by AspectJ

```
//Spring configuration

<bean id="authorizationAspect"
    class="ssel.banking.security.AuthorizationAspect"
    factory-method="aspectOf">

        <property name=.....

</bean>
```

Conclusion

- Spring can configure any AspectJ aspect
- Spring can only weave annotation-based style AspectJ aspects

Spring/AOP Syntax & Weavers

	AspectJ Language	AspectJ Annotation Style	XML Definition	DSL
Spring Weaver	No	Yes	Yes	Yes
AspectJ Weaver	Yes	Yes	No	No

XML-based aspects

- Advice in Java
- Conventions are the same as with AspectJ Annotation-based development style
- Aspect-specific info resides in Spring Configuration File
  - pointcuts
  - intertype declarations
  - precedence

XML-based aspects

- Security advice, a user has to be logged in.

```
public Object aroundControllerInvocation(ProceedingJoinPoint jp, HttpServletRequest
request) throws Throwable {

        if(getUserStorageService().getCurrentUser()!=null)
            return jp.proceed();

        String viewName = getViewName(request);

        for(String allowedView: allowedViews)
            if(viewName.endsWith(allowedView))
                return jp.proceed();

        return new ModelAndView(getRedirectView());

}
```

XML-based aspects

```
<bean id="authenticationAspect" class="ssel.banking.security.AuthenticationAspect">
   <property name="allowedViews">
   <list>
   <value>welcome</value>
   <value>login</value>
   </list>
   </property>
   <property name="redirectView" value="welcome"/>
   <property name="userStorageService" ref="userStorageService"/>
</bean>


<aop:config>
<aop:pointcut
   id="controllerInvocation"
   expression="execution(* org.springframework.web.servlet.mvc.Controller+
          .handleRequest(..)) and args(request,..)"
/>

<aop:aspect id="authenticationAspectAOP" ref="authenticationAspect">
     <aop:around  pointcut-ref="controllerInvocation"
          method="aroundControllerInvocation" arg-names="jp,request"/>
 </aop:aspect>

</aop:config>
```

Common Practices

- Define reusable pointcuts
- e.g. Knowledge of system layers:

```
@Aspect
public class SystemArchitecture {

  @Pointcut("within(com.xyz.someapp.web..*)")
  public static void inWebLayer() {}

  @Pointcut("within(com.xyz.someapp.service..*)")
  public static void inServiceLayer() {}

  .....
}

//Transactional Advice
@Around("execution(* *.*(..)) && SystemArchitecture.inServiceLayer()")
...
```

More Info

- Excellent documentation:
  - Reference Manual and JavaDoc:
    - **http://www.springframework.org/documentation**

- Further reading on lightweight JEE:
  - Expert One-on-One J2EE Development without EJB, Rod Johnson and Jurgen Hoeller, Wrox
  - Better, faster, lighter Java by Justin Gehtland and Bruce A. Tate, O'Reilly.