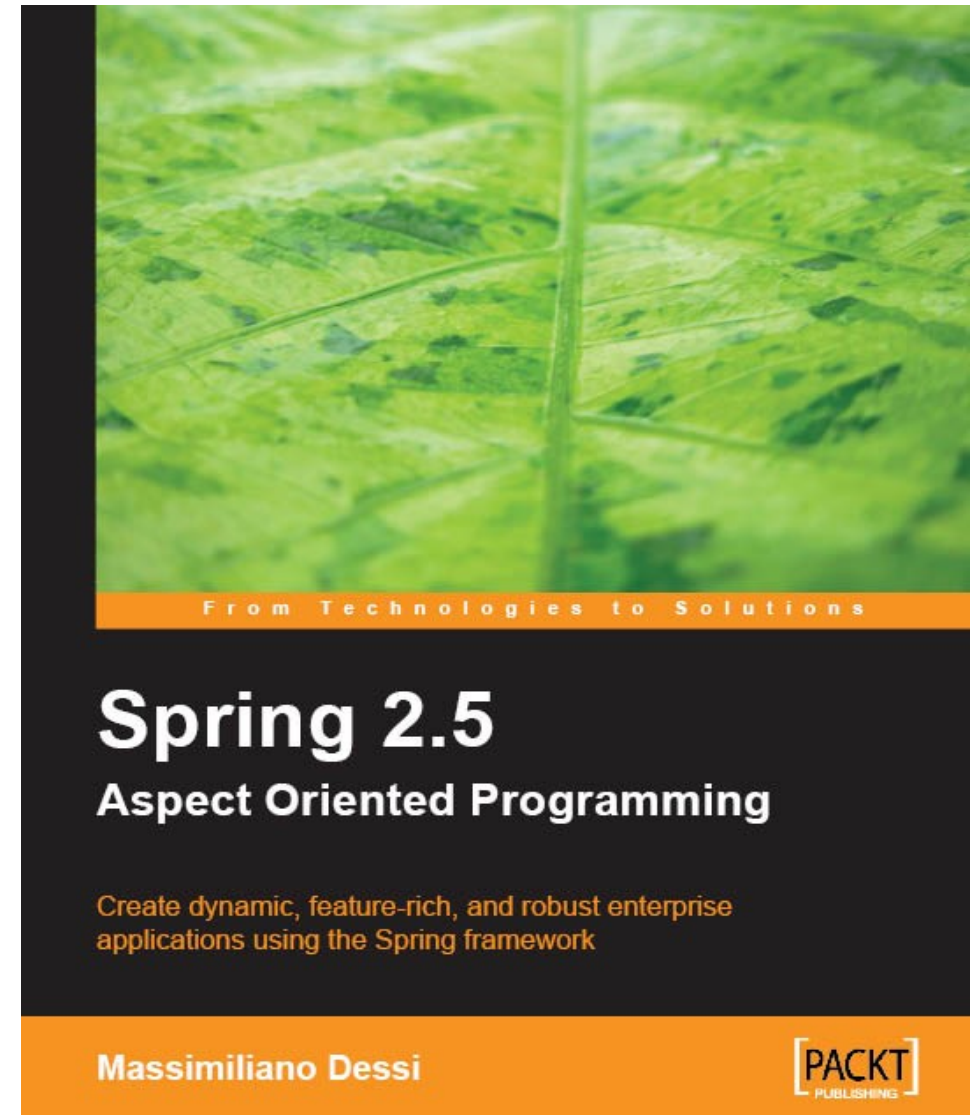


Spring @Aspect e @Controller

Come utilizzare Spring
AOP e Spring MVC per
scrivere migliori webapp
con meno codice e meno
XML.





Author

Java Architect 2008 (pro-netics)

Co-fondatore e consigliere

JugSardegna Onlus 2003



Fondatore e coordinatore:

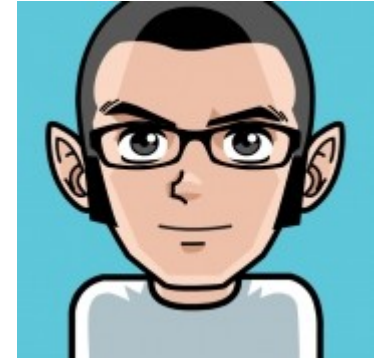
SpringFramework Italian User Group 2006



Jetspeed Italian User Group 2003



Groovy Italian User Group 2007



(pro-netics)

Massimiliano Dessì - desmax74@yahoo.it
Milano - 30/01/2009



L' Aspect Oriented Programming supporta l'Object Oriented Programming nella fase di implementazione, nei punti in cui mostra dei punti di debolezza.

L'AOP è complementare all' OOP nella fase implementativa.

L' AOP deve essere utilizzato con
giudizio e cognizione di causa.

Code scattering, quando una funzionalità è implementata in moduli diversi.

Si presenta in due forme:

- Blocchi di codice duplicato (es. identica implementazione di una interfaccia in classi diverse)
- Blocchi di codice complementari della stessa funzionalità, posti in differenti moduli.

(es in una ACL, un modulo esegue l'autenticazione e un altro l'autorizzazione)

Code tangling, un modulo ha troppi compiti contemporanei.

```
public ModelAndView list(HttpServletRequest req, HttpServletResponse res)
    throws Exception {

    log(req); //logging

    if(req.isUserInRole("admin")) { // authorization

        List users ;
        try { //exception handling
            String username = req.getRemoteUser();
            users =cache.get(Integer.valueOf(conf.getValue("numberOfUsers")),
                username); //cache with authorization
        } catch (Exception e) {
            users = userManager getUsers();
        }

        return new ModelAndView("usersTemplate", "users", users);

    }else{
        return new ModelAndView("notAllowed");
    }
}
```

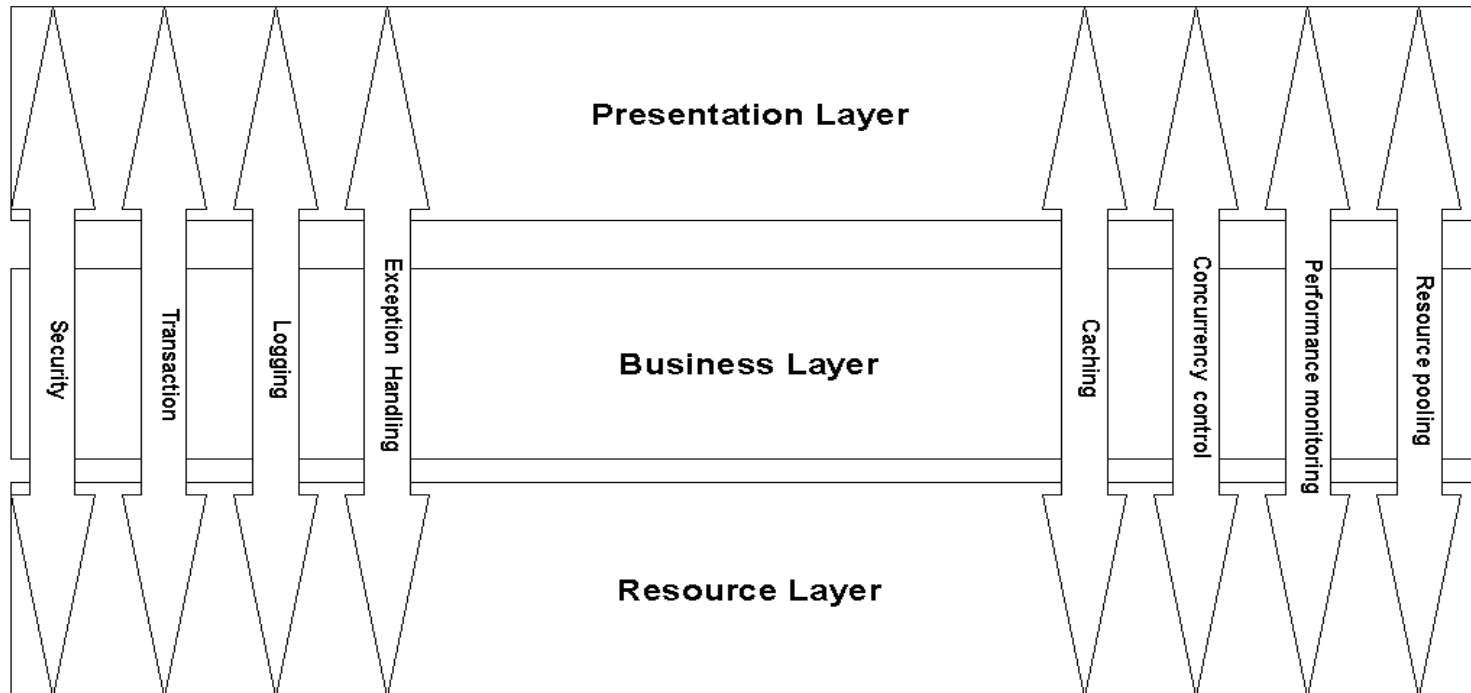
- Evoluzione difficile, il codice è legato a diverse funzionalità
- Cattiva qualità, quale era la funzione del metodo ?
- Codice non riusabile, è adatto solo a questo scenario
- Tracciabilità, chi fa cosa ?
- Improduttività, dove è il punto dove intervenire ?

Eppure è scritto con un linguaggio Object Oriented...

L'implementazione è il vero problema, non il linguaggio.

Origine del problema

Le classi si ritrovano a dover implementare funzionalità trasversali.



Il problema non è più una classe che deve avere un solo compito/scopo/responsabilità/funzionalità, ma come quel compito viene richiamato per essere utilizzato.

L' AOP fornisce i costrutti e gli strumenti per modularizzare queste funzionalità che attraversano trasversalmente l'applicazione (crosscutting concerns).

L' AOP permette quindi di centralizzare realmente il codice duplicato consentendo di applicarlo secondo regole prestabilite nei punti desiderati, durante il flusso di esecuzione.

Aspect: Corrisponde alla classe nell' OOP, contiene la funzionalità trasversale (crosscutting concern).

Joinpoint: Punto di esecuzione del codice
(es l'invocazione di un costruttore, esecuzione di un metodo o la gestione di una Exception)

Advice: L'azione da compiere nel joinpoint.

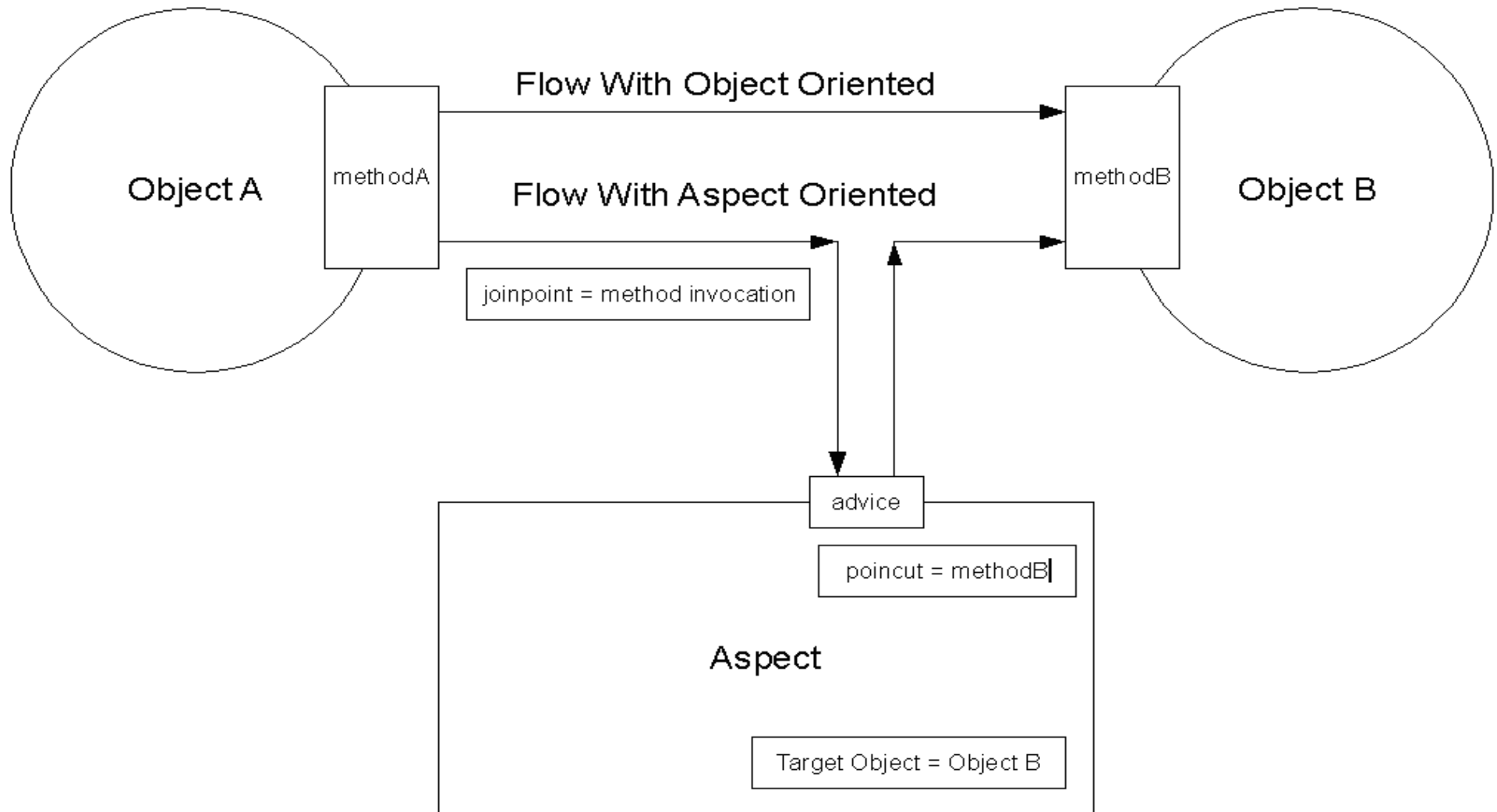
Pointcut: Contiene l'espressione che permette di individuare il joinpoint al quale vogliamo applicare l'advice

Introduction: Permette l'aggiunta di interfacce e relativa implementazione a runtime, a oggetti già definiti.

Target: Classe su cui agisce l'aspect con l'advice

Weaving: L'azione con cui vengono legati gli “attori” AOP e le classi target.

Come opera l' AOP



Per permettere un utilizzo semplificato dell' AOP, Spring utilizza come Joinpoint l'esecuzione dei metodi.

Questo significa che possiamo agire:

prima, dopo, attorno ad un metodo, alla sollevazione di una eccezione, dopo l'esecuzione qualsiasi sia l' esito (finally),
utilizziamo normali classi Java con annotazioni oppure con XML.
Chi lavora nell' ombra per permetterci questa semplicità ?

`java.lang.reflect.Proxy` oppure CGLIB

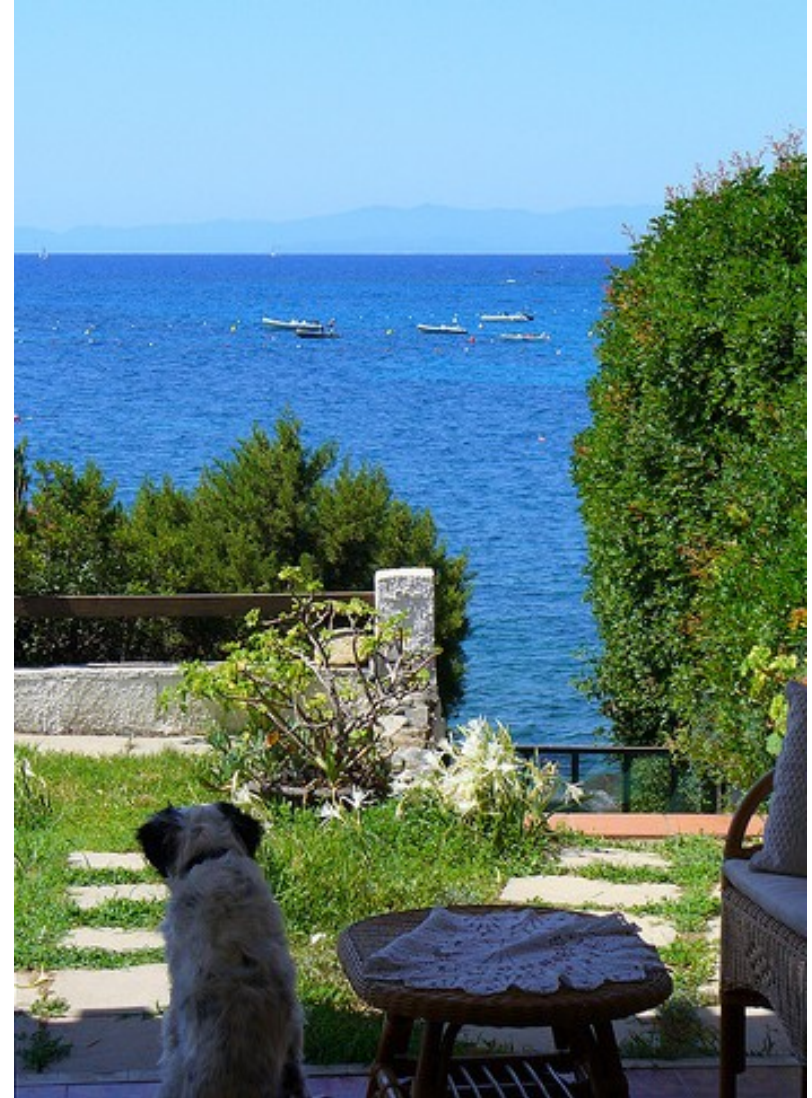
Spring AOP: take it easy

Dobbiamo solo preoccuparci di decidere:

- Cosa centralizzare/modularizzare ?
- Quando deve entrare in azione ?

Cosa posso ottenere:

- Supporto al Domain Driven Design
- Meno codice
- Codice gestibile e manutenibile
- Meno scuse per usare realmente l' OOP, non per spostare contenitori...



Spring AOP - Concorrenza

```
@Aspect() @Order(0)
public class ConcurrentAspect {

    private final ReadWriteLock lock =
        new ReentrantReadWriteLock();
    private final Lock rLock = lock.readLock();
    private final Lock wLock = lock.writeLock();

    @Pointcut("execution (* isVisible(..))")
    private void isVisible() {}

    @Pointcut("execution (* retainItem(..))")
    private void retainItem() {}

    @Pointcut("execution (* release(..))")
    private void release() {}

    @Pointcut("release() || retainItem()")
    private void releaseOrRetain() {}
}
```

Prima parte:

-Mi creo dei lock rientranti in lettura e scrittura

-Annoto la classe con @Aspect

-Definisco con l'annotazione @Pointcut le espressioni che mi indicano quando l'Aspect deve intervenire.

Spring AOP - Concorrenza

Seconda parte:

Definisco riutilizzando i nomi dei metodi sui quali ho annotato i `@Pointcut` la logica da eseguire con i lock

Con le annotazioni seguenti dichiaro quando voglio sia eseguita

`@Before`
`@After`
`@AfterReturning`
`@Around`
`@AfterThrowing`

```
@Before("isAvailable()")
public void setReadLock() {
    rLock.lock();
}

@After("isAvailable()")
public void releaseReadLock() {
    rLock.unlock();
}

@Before("releaseOrRetain()")
public void setWriteLock() {
    wLock.lock();
}

@After("releaseOrRetain()")
public void releaseWriteLock() {
    wLock.unlock();
}
}
```


Spring AOP – Advice Type

@Before

@After

@AfterReturning

@Around

@AfterThrowing

Gli advice non solo ci permettono di eseguire logica nei punti definiti dai pointcut, ma anche di ottenere informazioni sulla esecuzione (classe target, metodo chiamato, argomenti passati, valore restituito). Con alcuni tipi di advice (around) anche di avere il controllo sul flusso di esecuzione. La classe chiamante naturalmente non sa nulla di quello che avviene, vede sempre una semplice classe



Spring AOP – Pointcut

- **execution**
- **within**
- **this**
- **target**
- **args**
- **@target**
- **@args**
- **@within**
- **@annotation**
- **bean**

Nei pointcut precedenti abbiamo visto come espressione per definire il punto di applicazione fosse l' esecuzione di un metodo con un determinato nome.

Abbiamo a disposizione anche altri designatori di pointcut per avere il massimo controllo.



```
@ManagedResource("freshfruitstore:type=TimeExecutionManagedAspect")
@Aspect() @Order(2)
public class TimeExecutionManagedAspect {

    @ManagedAttribute
    public long getAverageCallTime() {
        return (this.callCount > 0
            ? this.accumulatedCallTime / this.callCount : 0);
    }

    @ManagedOperation
    public void resetCounters() {
        this.callCount = 0;
        this.accumulatedCallTime = 0;
    }

    @ManagedAttribute
    public long getAverageCallTime() {
        return (this.callCount > 0 ?
            this.accumulatedCallTime / this.callCount : 0);
    }

    ...
}
```

Oltre a poter intervenire nel flusso di esecuzione, posso anche gestire gli Aspect stessi con JMX, esponendo come attributi o operazioni, i normali metodi dell' Aspect che è sempre una classe java.

...

```
@Around("within(it.mypackage.service.*Impl)")
public Object invoke(ProceedingJoinPoint joinPoint)
    throws Throwable {

    if (this.isTimeExecutionEnabled) {
        Stopwatch sw = new Stopwatch(joinPoint.toString());
        sw.start("invoke");
        try {
            return joinPoint.proceed();
        } finally {
            sw.stop();
            synchronized (this) {
                this.accumulatedCallTime += sw.getTotalTimeMillis();
            }
            logger.info(sw.prettyPrint());
        }
    } else {
        return joinPoint.proceed();
    }
}
```

...

Posso quindi con JMX
cambiare il
comportamento
dell' Aspect a runtime

Spring AOP – Introductions

Una introduction mi permette di decorare un oggetto a runtime, aggiungendogli interfacce e relativa implementazione.

Questo permette sia di evitare la duplicazione di una implementazione, sia di simulare l'ereditarietà multipla che java non ha.



```
@Aspect
public class ParallelepipedIntroduction {

    @DeclareParents(value = "org.springaop.chapter.four.introduction.Box",
        defaultImpl = Titanium.class)
    public Matter matter;

    @DeclareParents(value = "org.springaop.chapter.four.introduction.Box",
        defaultImpl = Cube.class)
    public GeometricForm geometricForm;
}
```

Spring AOP – @Aspect

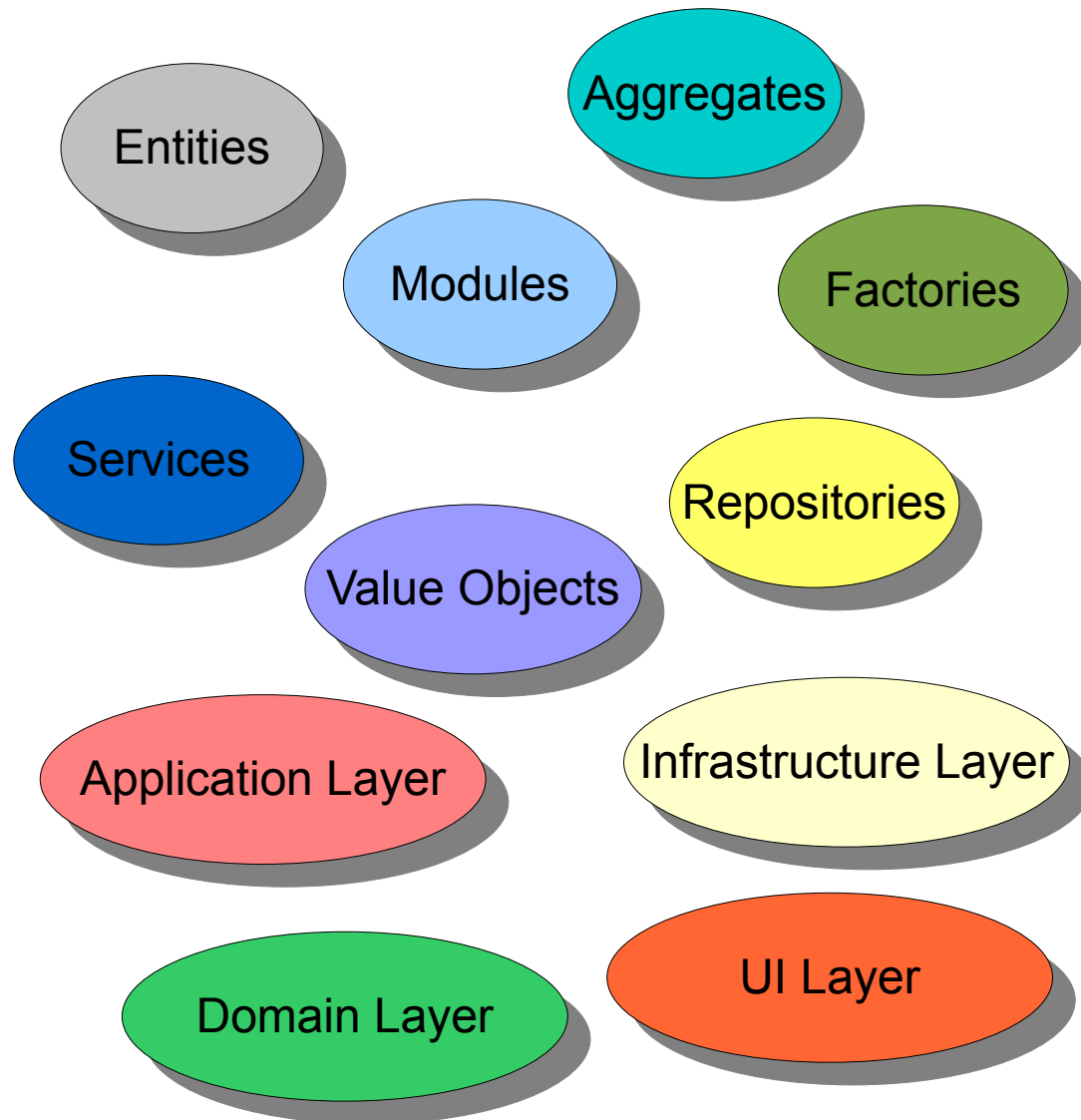
Le annotazioni viste finora compresa la sintassi dei pointcut sono fornite da AspectJ

```
import org.aspectj.lang.annotation.*
```

ma sono assolutamente dentro il “contesto” di Spring e su dei bean di Spring.

Vediamo ora invece cosa possiamo utilizzare di Spring (IoC) fuori dal “contesto” di Spring tramite AspectJ.

Dovremo scendere a qualche compromesso...



Domain-Driven Design è un modo di pensare le applicazioni. Suggestendo di concentrare l'attenzione sul dominio del problema, invitando a ragionare ad oggetti e non in base a contenitori di dati da movimentare...

```
public interface Customer extends NamedEntity {  
    public Address getAddress();  
    public ContactInformation getContact();  
    public void modifyContactInformation(ContactInformation contact);  
    public void modifyAddress(Address address);  
    public Boolean saveCustomer();  
    public Boolean createOrder();  
    public Boolean saveOrder();  
    public Order getOrder();  
    public List<Order> getOrders();  
}
```

Nelle entità è concentrata
la logica di business, non
nei servizi che eseguono
“procedure”...

In questo modo i nostri
oggetti hanno dati e
comportamenti, come
dovrebbe essere nelle
classi ad oggetti

SpringAOP +DDD +AspectJ

```
@Configurable(dependencyCheck = true, autowire=Autowire.BY_TYPE)
public class CustomerImpl implements Customer, Serializable {

    @Autowired
    public void setCustomerRepository(@Qualifier("customerRepository") CustomerRepository customerRepo) {
        this.customerRepository = customerRepository;
    }

    @Autowired
    public void setOrderRepository(@Qualifier("orderRepository") OrderRepository orderRepo) {
        this.orderRepository = orderRepo;
    }

    public Boolean createOrder() {
        Boolean result = false;
        if (order == null) {
            order = new OrderImpl.Builder(Constants.ID_NEW, new Date(), id
                .toString()).build();
            result = true;
        }
        return result;
    }

    public Boolean saveCustomer() {
        return customerRepository.saveCustomer(this);
    }

    ...
}
```


Con `@Configurable` abbiamo dichiarando che alla classe verranno iniettate delle dipendenze benchè non sia uno spring bean.

L' `applicationContext` di Spring conosce la classe semplicemente come un bean prototype.

Per avere questa funzionalità abbiamo necessità di comunicare alla jvm il jar di spring da usare per il Load Time Weaving:

`-javaagent:<path>spring-agent.jar`

oppure configurare tomcat in questo modo per farlo al nostro posto:

```
<Loader
loaderClass="org.springframework.instrument.classloading.tomcat.TomcatInstrumentableClassLoader"
useSystemClassLoaderAsParent="false"/>
```

Spring AOP + AspectJ Weaver

Utilizzare l'AspectJ Weaver comporta un approccio diverso rispetto alla semplicità di SpringAOP vista sinora.

Annotare delle classi come `@Aspect` e farli creare da Spring, significa rimanere comunque dentro l' IoC Container, così come definire pointcut su esecuzioni di metodi da parte di bean di Spring.

Con il LTW stiamo dicendo a Spring di operare fuori dal suo “contesto”, per iniettare le dipendenze su oggetti non creati dall' IoC.

Per utilizzare il LTW definiamo su un file per AspectJ le classi su cui deve operare e quali Aspect deve creare, Aspects che non sono bean creati da Spring...

100% AspectJ

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
<aspectj>
<weaver options="-showWeaveInfo
-XmessageHandlerClass:org.springframework.aop.aspectj.AspectJWeaverMessageHandler">
  <!-- only weave classes in our application-specific packages -->
  <include within="it.freshfruits.domain.entity.*"/>
  <include within="it.freshfruits.domain.factory.*"/>
  <include within="it.freshfruits.domain.service.*"/>
  <include within="it.freshfruits.domain.vo.*"/>
  <include within="it.freshfruits.application.repository.*"/>
  <exclude within="it.freshfruits.aspect.*"/>
</weaver>
<aspects>
  <aspect name="it.freshfruits.aspect.ConcurrentAspect" />
  <aspect name="it.freshfruits.aspect.LogManagedAspect" />
  <aspect name="it.freshfruits.aspect.TimeExecutionManagedAspect" />
</aspects>
</aspectj>
```

Spring AOP + AspectJ Weaver

Una volta che abbiamo le dipendenze in una entità del dominio, sia con l'iniezione delle dipendenze con il LTW, o tramite passaggio delle dipendenze come argomenti del costruttore ad opera di una factory, vediamo il risultato sulla UI.

```
@Controller("customerController")
public class CustomerController {

    @RequestMapping("/customer.create.page")
    public ModelAndView create(HttpServletRequest req) {
        return new ModelAndView("customer/create", "result", UiUtils.getCustomer(req).createOrder());
    }

    @RequestMapping("/customer.order.page")
    public ModelAndView order(HttpServletRequest req) {
        return new ModelAndView("customer/order", "order", UiUtils.getOrder(req));
    }

    @RequestMapping("/customer.items.page")
    public ModelAndView items(HttpServletRequest req) {
        return new ModelAndView("customer/items", "items", UiUtils.getOrder(req).getOrderItems());
    }

    ...
}
```

I controller risultanti saranno completamente stateless e senza dipendenze, e con una semplice chiamata sulla entità.

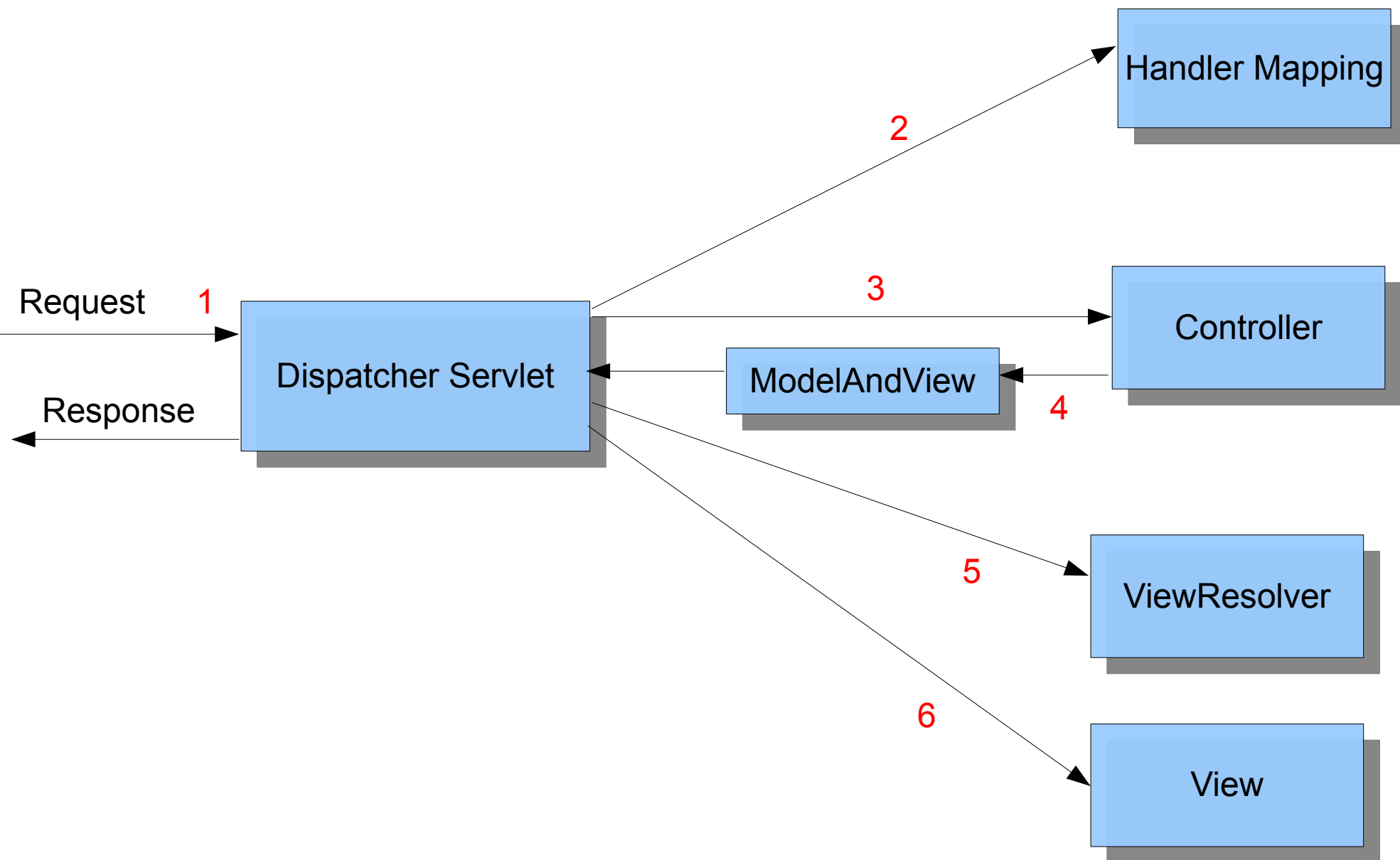
Handler Interceptor che pone nella HttpServletRequest l'entità customer utilizzata dal Customer Controller

```
public class CustomerInterceptor extends HandlerInterceptorAdapter {  
  
    @Override  
    public boolean preHandle(HttpServletRequest req, HttpServletResponse res, Object handler) throws  
Exception {  
        req.setAttribute(Constants.CUSTOMER, customerFactory.getCurrentCustomer());  
        return true;  
    }  
  
    @Autowired  
    private CustomerFactory customerFactory;  
}
```

OOP + AOP + DDD
=
Codice a oggetti
Codice pulito
Codice elegante



Spring MVC




```
@Controller("fruitController") @RequestMapping("/fruit.edit.admin")
public class FruitController {

    @RequestMapping(method = RequestMethod.POST)
    public String processSubmit(@ModelAttribute("fruit") FruitMap fruit, BindingResult result,
        SessionStatus status) {

        validator.validate(fruit, result);
        if (result.hasErrors()) {
            return "userForm";
        } else {
            fruit.save();
            status.setComplete();
            return "redirect:role.list.admin";
        }
    }

    @InitBinder()
    public void initBinder(WebDataBinder binder) throws Exception {
        binder.registerCustomEditor(String.class, new StringTrimmerEditor(false));
    }

    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(@RequestParam(required = false, value = "id") Integer id, ModelMap model) {
        model.addAttribute(Constants.FRUIT, id == null ? new FruitMap() : fruitRepository.getFruitType(id));
        return "role/form";
    }

    @Autowired @Qualifier("fruitRepository")
    private FruitTypeRepository fruitRepository;
    @Autowired @Qualifier("fruitValidator")
    FruitValidator validator;
}
```

```
@Controller("customerController")
public class CustomerController {

    @RequestMapping("/customer.create.page")
    public ModelAndView create(HttpServletRequest request req) {
        return new ModelAndView("customer/create", "result", UiUtils.getCustomer(req).createOrder());
    }

    @RequestMapping("/customer.save.page")
    public ModelAndView save(HttpServletRequest request req) {
        return new ModelAndView("customer/save", "result", UiUtils.getCustomer(req).saveOrder());
    }

    @RequestMapping("/customer.show.page")
    public ModelAndView show(HttpServletRequest request req) {
        return new ModelAndView("customer/show", "customer", UiUtils.getCustomer(req));
    }

    @RequestMapping("/customer.order.page")
    public ModelAndView order(HttpServletRequest request req) {
        return new ModelAndView("customer/order", "order", UiUtils.getOrder(req));
    }

    @RequestMapping("/customer.items.page")
    public ModelAndView items(HttpServletRequest request req) {
        return new ModelAndView("customer/items", "items", UiUtils.getOrder(req).getOrderItems());
    }

    @RequestMapping("/customer.remove.page")
    public ModelAndView remove(@RequestParam("id") String id, HttpServletRequest request req) throws Exception {
        Order order = UiUtils.getOrder(req);
        return order.removeOrderItem(order.getId().toString(), id) ? new ModelAndView("customer/items",
            "items", order.getOrderItems()) : new ModelAndView("customer/remove", "result", false);
    }
}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:p="http://www.springframework.org/schema/p"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan base-package="it.freshfruits.ui"/>
<context:annotation-config/>

<bean name="urlMapping" class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="customerInterceptor"/>
        </list>
    </property>
</bean>

<bean name="customerInterceptor" class="it.freshfruits.ui.interceptor.CustomerInterceptor"/>

</beans>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:p="http://www.springframework.org/schema/p" xmlns:sec="http://www.springframework.org/schema/security"
xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/security http://www.springframework.org/schema/security/spring-security-2.0.4.xsd">

    <sec:http>
        <sec:intercept-url pattern="/log*.jsp" filters="none" />
        <sec:intercept-url pattern="/*.page" access="ROLE_USER" />
        <sec:intercept-url pattern="/*.admin" access="ROLE_ADMIN" />
        <sec:form-login login-page="/login.jsp" default-target-url="/" login-processing-url="/j_security_check"
            authentication-failure-url="/loginError.jsp" />
        <sec:logout logout-url="/logout.jsp" invalidate-session="true" logout-success-url="/login.jsp" />
        <sec:remember-me />
        <sec:intercept-url pattern="*.htm" access="ROLE_USER,ROLE_ANONYMOUS" />
        <sec:intercept-url pattern="*.page" access="ROLE_USER,ROLE_ADMIN" />
        <sec:intercept-url pattern="*.edit" access="ROLE_USER,ROLE_ADMIN" />
        <sec:intercept-url pattern="*.admin" access="ROLE_ADMIN" />
    </sec:http>

    <sec:authentication-provider user-service-ref="sffsUserDetailsService"><sec:password-encoder hash="sha" /></sec:authentication-provider>

    <bean id="accessDecisionManager" class="org.springframework.security.vote.AffirmativeBased">
        <property name="decisionVoters">
            <list>
                <bean class="org.springframework.security.vote.RoleVoter" />
                <bean class="org.springframework.security.vote.AuthenticatedVoter" />
            </list>
        </property>
    </bean>

    <bean id="sffsUserDetailsService" class="it.freshfruits.security.AuthenticationJdbcDaoImpl"
        p:rolePrefix="ROLE_" p:dataSource-ref="dataSource"
        p:usersByUsernameQuery="SELECT username, password, enabled FROM authentication WHERE username = ?"
        p:authoritiesByUsernameQuery="SELECT username, authority FROM roles WHERE username = ?" />

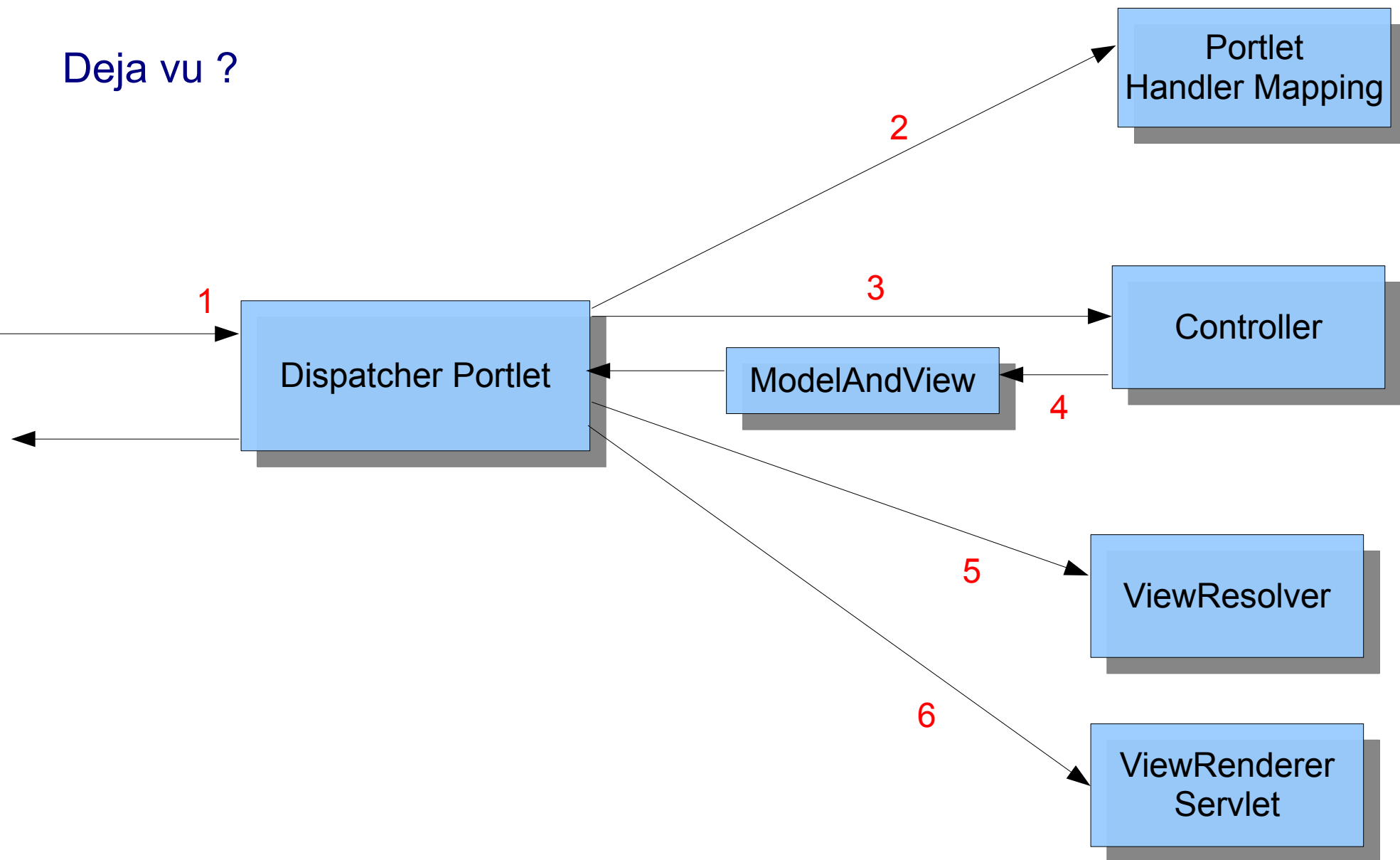
    <sec:global-method-security access-decision-manager-ref="accessDecisionManager">
        <sec:protect-pointcut expression="execution(* it.freshfruits.domain.entity.*(..))"
            access="ROLE_USER,ROLE_ADMIN" />
    </sec:global-method-security>

    ...
</beans>
```

```
...
<context-param>
  <param-name>webAppRootKey</param-name>
  <param-value>springFreshFruitsStore.root</param-value>
</context-param>
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/conf/spring/sffs-*.xml</param-value>
</context-param>
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
</filter>
<servlet>
  <servlet-name>sffs</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <load-on-startup>0</load-on-startup>
</servlet>
<servlet-mapping><servlet-name>sffs</servlet-name><url-pattern>*.page</url-pattern></servlet-mapping>
<servlet-mapping><servlet-name>sffs</servlet-name><url-pattern>*.edit</url-pattern></servlet-mapping>
<servlet-mapping><servlet-name>sffs</servlet-name><url-pattern>*.admin</url-pattern></servlet-mapping>
<servlet-mapping><servlet-name>sffs</servlet-name><url-pattern>*.htm</url-pattern></servlet-mapping>
...
```

```
<%@ include file="/WEB-INF/jsp/taglibs.jsp"%>
<tag:pager href="user.list.page"/>
<div>
  <div align="center"><span><spring:message code="ui.users"/></span></div>
  <c:if test="${msg ne ''}">${msg}</c:if>
  <table class="list">
    <thead><tr><th align="left"><spring:message code="ui.user.username"/></th></tr></thead>
    <tbody>
      <c:forEach var="user" items="${users}" varStatus="status">
        <c:choose>
          <c:when test="${status.count % 2 == 0}">
            <tr class="table-row-dispari"></c:when>
            <c:otherwise>
              <tr class="table-row-pari"></c:otherwise>
            </c:choose>
            <td align="left">${user.username}</td>
            <td align="center">
              <a href="user.roles.page?sid=${user.id}"><img border="0" title="<spring:message code="ui.action.view"/>"
                alt="<spring:message code="ui.action.view"/>" src="img/view.png"/></a>
            </td>
            <td align="center">
              <a href="user.detail.page?sid=${user.id}"><img border="0" title="<spring:message code="ui.action.edit"/>"
                alt="<spring:message code="ui.action.edit"/>" src="img/edit.png"/></a></td>
            </tr>
          </c:forEach>
        </tbody>
      </table>
    </div>
    <br/>
    <tag:pager href="user.list.page"/>
    <div align="center">
      <input type="image" src="img/add.png" title="<spring:message code="ui.action.user.new"/>" alt="<spring:message
code="ui.action.user.new"/>" onclick="location.href = 'user.edit.page'" value="<spring:message code="ui.action.user.new"/>"/>
    </div>
```

Deja vu ?



Spring riutilizza gli stessi concetti della parte MVC Servlet, traslando dove serve le richieste Portlet (action o render) in Servlet, permettendo il riuso dei controller.

Spring fornisce una `NativeWebRequest` che maschera il tipo di richiesta, questo approccio ci permette di scrivere controller che virtualmente possono non sapere dentro cosa stanno girando.

In pratica, i problemi li può creare il portal server...

Nonostante i problemi con il portal server, è comunque possibile riutilizzare la gran parte del codice scritto per una webapp normale.

Gestione Ruoli

http://localhost:8080/acl-web/role.list.page

Access Control List BETA

Home Ruoli Utenti logout

Ruoli
Elenco

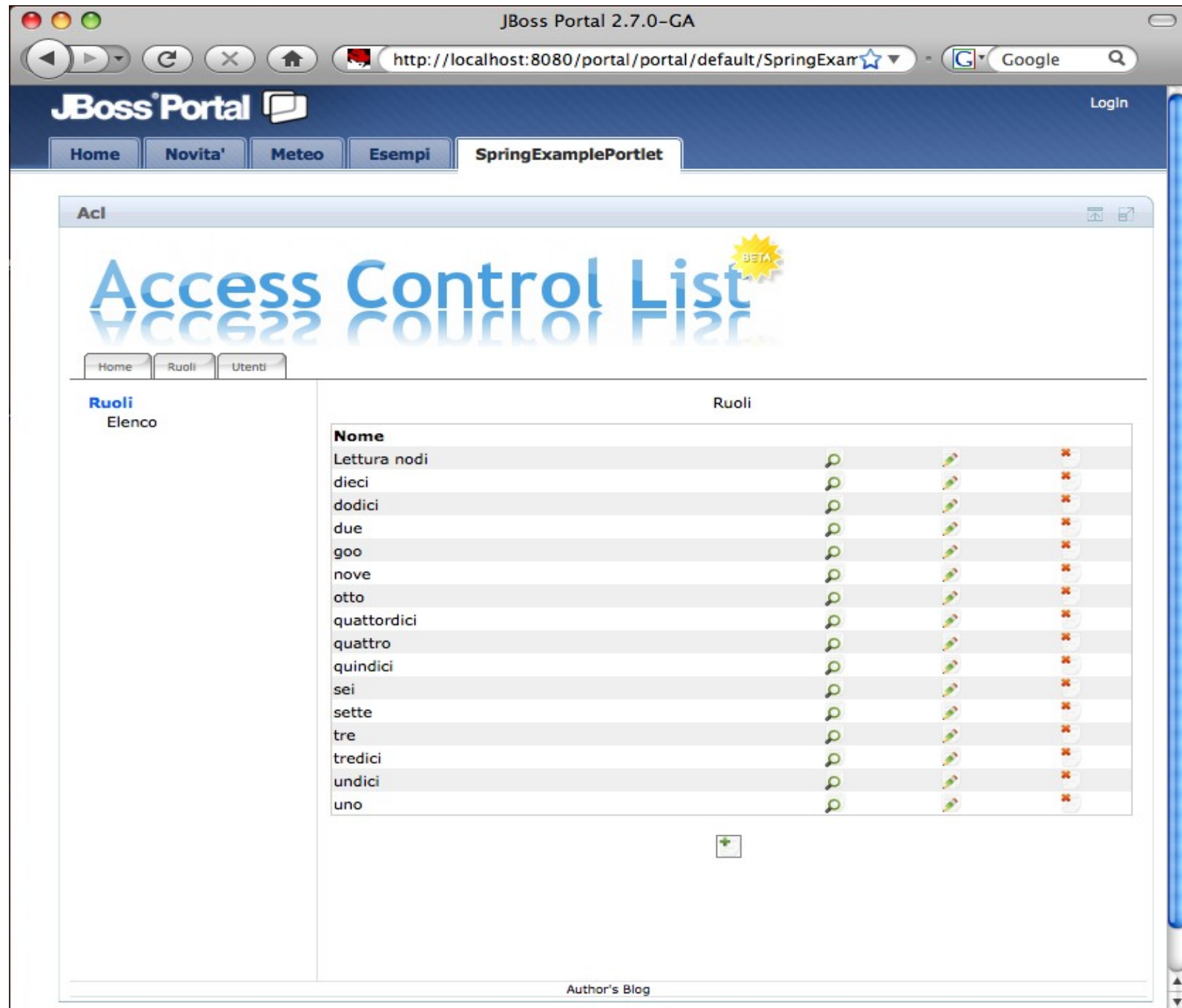
Ruoli

Nome			
Lettura nodi			
dieci			
dodici			
due			
goo			
nove			
otto			
quattordici			
quattro			
quindici			
sei			
sette			
tre			
tredici			
undici			
uno			

+

Author's Blog

Se abbiamo
disegnato bene la
nostra webapp
con SpringMVC,
non è troppo
complesso
portarlo su
SpringMVC Portlet



JBoss Portal 2.7.0-GA

http://localhost:8080/portal/portal/default/SpringExam

JBoss Portal

Home Novita' Meteo Esempi SpringExamplePortlet

Login

ACL

Access Control List **BETA**

Home Ruoli Utenti

Ruoli
Elenco

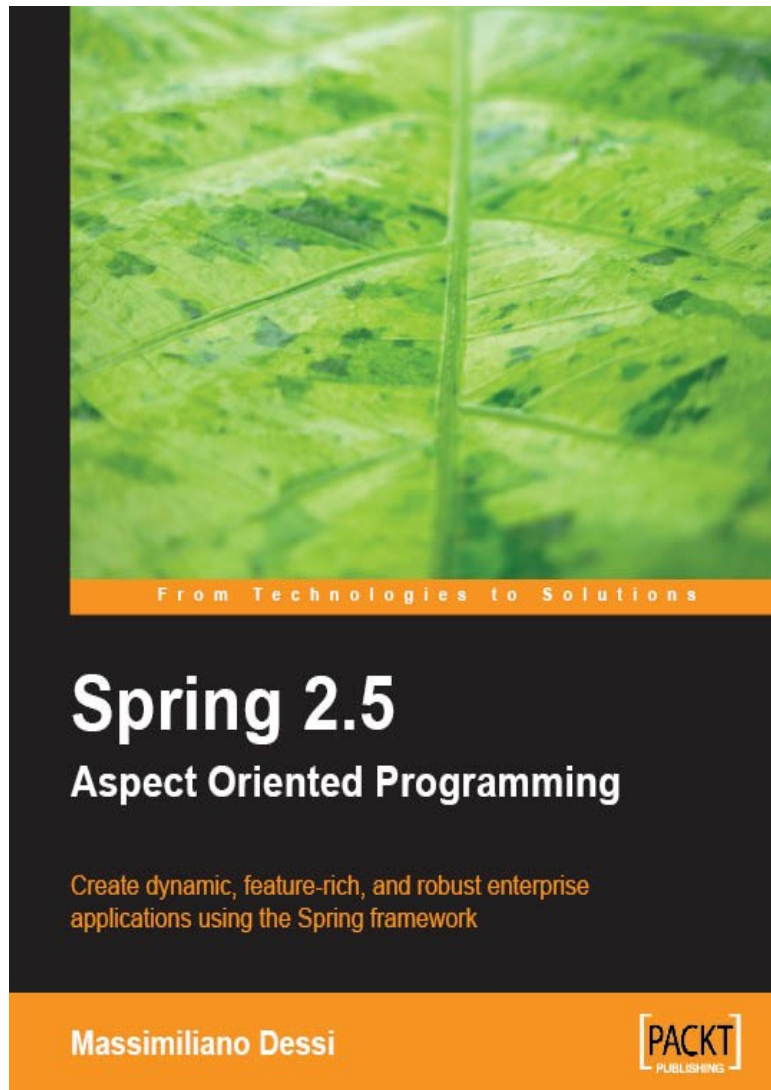
Nome			
Lettura nodi			
dieci			
dodici			
due			
goo			
nove			
otto			
quattordici			
quattro			
quindici			
sei			
sette			
tre			
tredici			
undici			
uno			

Author's Blog

Spring AOP Book

Maggiori
informazioni
a febbraio su :

<http://www.packtpub.com>



Un enorme grazie a Stefano Sanna per il supporto durante la scrittura del libro. Se è stato terminato è grazie al suo contributo che non è mai mancato.

Domande ?



Grazie per l'attenzione !

Massimiliano Dessì
desmax74 at yahoo.it

<http://jroller.com/desmax>
<http://www.linkedin.com/in/desmax74>
<http://wiki.java.net/bin/view/People/MassimilianoDessi>
<http://www.jugsardegna.org/vqwiki/jsp/Wiki?MassimilianoDessi>

Spring Framework Italian User Group
<http://it.groups.yahoo.com/group/SpringFramework-it>