# Programmazione Java Avanzata

## Hibernate (Parte 3)

*Ing. Giuseppe D'Aquì*

# Testi Consigliati

- Beginning Hibernate 2nd edition (Apress)
  - Sul sito è possibile scaricare, tra gli “extra”, il codice sorgente e il capitolo 3
- Hibernate Getting Started
  - http://docs.jboss.org/hibernate/core/3.6/quickstart/

# Criteria API: cos'è

- Criteria API permette di costruire, tramite codice, un oggetto query che contiene al suo interno i criteri per la selezione e proiezione
- È una sorta di strutturazione ad oggetti di una query SQL
- Un oggetto di tipo Criteria viene creato a partire dalla session, può essere manipolato e infine si possono chiamare i suoi metodi che eseguono la query

# Criteria API: cos'è

- Criteria crit = session.createCriteria(User.class);
- List results = crit.list();

# Criteria API: pro e contro

- Pro: Le espressioni possono essere create in Java, semplificando la creazione di sub-query e introducendo la possibilità di controllare il tipo degli oggetti al momento della compilazione
- Contro: Le espressioni possono essere comprese solo da uno sviluppatore, non possono essere ottimizzate da un DBA

# Restrictions

- Le Restrictions sono i vincoli che i nostri oggetti devono rispettare per essere selezionati dalla query
- (equivalgono al WHERE)
- Ci sono un certo numero di Restriction definiti come metodi statici della classe Restrictions
- Vengono aggiunti ai Criteria tramite il metodo add()

# Restrictions

- Criteria crit = session.createCriteria(User.class);
- crit.add(Restrictions.eq(“username”, “paolo”));
- List results = crit.list();

# Restrictions

- Altri metodi:
  - ne: not equal
  - like: come LIKE SQL
  - ilike: come like, ma case-insensitive
  - isNull/isNotNull:
  - gt/lt/ge/le: greater than/less than/greater-equal/less-equal
  - or: permette di specificare due criteri opzionali

# Nota Bene

- Il metodo add() restituisce un oggetto di tipo Criteria, e questo permette di concatenare i vincoli:
- List results = session.createCriteria(User.class)
  - .add(Restriction.eq(“username”, “paolo”))
  - .add(Restriction.like(“email”, “paolo@%”))
  - .list();
- Tutto su una sola riga

# Pagination

- Criteria, come HQL, permette di definire il primo risultato restituito e il numero massimo di risultati da restituire all'utente
- I metodi sono:
  - setFirstResult(int)
  - setMaxResults(int)

# Unique Result

- Il metodo list() di Criteria ci restituisce l'insieme dei risultati
- Quando la query ha una sola tupla come risultato, usando uniqueResult() ci verrà restituito un oggetto e non una lista
- Se però i risultati sono più di uno, dobbiamo usare setmaxResult(1) altrimenti verrà lanciata un'eccezione

# Sorting

- L'ordinamento si effettua tramite il metodo addOrder() di Criteria
- L'ordinamento vero e proprio è definito da metodi statici della classe Order [asc() o desc()]
- Criteria crit = session.createCriteria(User.class);
- crit.add(Restrictions.eq(“username”, “paolo”));
- crit.addOrder(Order.desc(“age”));
- List results = crit.list();

# Join

- Le associazioni (join) vengono seguite creando degli oggetti Criteria da Criteria già esistenti, tramite il metodo createCriteria()
- Criteria crit = session.createCriteria(User.class);
- Criteria crit2 = crit.createCriteria(“roles”);
- crit.add(Restrictions.eq(“username”, “paolo”));
- List results = crit.list();

- I risultati si possono ottenere da entrambi i Criteria

# Proiezioni

- Per effettuare le proiezioni si costruisce un oggetto ProjectionList al quale aggiungeremo tutte le definizione dei campi da selezionare
- ProjectionList pl = Projections.projectionList();
- pl.add(Projections.property(“username”));
- pl.add(Projections.property(“password”));
- crit.setProjection(pl);

# Aggregazioni

- Le aggregazioni si effettuano tramite i metodi di Projections:
  - avg(String)
  - count(String)
  - countDistinct(String)
  - max(String)
  - min(String)
  - sum(String)

# Aggregazioni

- ProjectionList pl = Projections.projectionList();
- pl.add(Projections.countDistinct(“username”));
- crit.setProjection(pl);

- I criteri di aggregazione si definiscono tramite il metodo groupProperty(), sempre di Projections
- ProjectionList pl = Projections.projectionList();
- pl.add(Projections.avg(“age”));
- pl.add(Projections.groupProperty(“city”));
- crit.setProjection(pl);

# Query by Example

- Query By Example (QBE) ci permette di generare i nostri Criteria a partire da un oggetto già esistente
- Per esempio, vorremmo caricare tutti gli oggetti “simili” ad un oggetto che già abbiamo, senza specificare la query
- Per creare un “esempio di ricerca” si usa il metodo statico create() della classe Example

# Query by Example

- Criteria crit = session.createCriteria(User.class);
- User user = new User();
- user.setUsername(“paolo”);
- crit.add(Example.create(user));
- List results = crit.list();

# Programmazione Java Avanzata

## Spring

*Ing. Giuseppe D'Aquì*

# Testi Consigliati

- Beginning Spring 2  (Apress)
  - Sul sito è possibile scaricare, tra gli “extra”, il codice sorgente e il capitolo 1
- Spring Reference Docs
  - http://static.springsource.org/spring/docs/2.5.x/refe
-

# Le dipendenze

- Generalmente, scrivendo codice, le classi accumulano dipendenze
- Alcune classi arrivano ad usare dozzine di altre classi
- E' il principio base della separazione di responsabilità, per cui si tende ad avere
  - Molte classi
  - Ognuna specializzata su una particolare responsabilità

# Le dipendenze (2)

- Le classi che usiamo possono essere semplici, come String o Integer
- Oppure più complesse, ad esempio i loro oggetti possono aver necessità di essere creati ed inizializzati in un certo modo
- Si forma una rete di dipendenze in cui, per creare un oggetto, dobbiamo crearne altri, e così via

# Le prime soluzioni

- Una soluzione potrebbe essere l'uso dei pattern di tipo *Creational* (Builder, Abstract Factory, etc)
- Usandoli possiamo nascondere l'implementazione dell'inizializzazione di oggetti complessi

# Sapere quando fermarsi

- La soluzione proposta però è diventate sempre più insostenibile al crescere della complessità delle applicazioni
- Es.
  - 1) Per costruire l'oggetto B, usiamo un Factory Method per costruire l'oggetto A
  - 2) per costruire C che dipende da B, usiamo un Factory Method
  - 3) per costruire D che dipende da C, usiamo un Factory Method
  - ….

# Sapere quando fermarsi (2)

- 
  - AFactory af=AFactory.getInstance();
  - BFactory bf=BFactory.getInstance();
  - A a = af.createNewA();
  - B b = bf.createNewB(a);
  - CFactory cf=CFactory.getInstance();
  - C c = cf.createNewC(b);
  - DFactory df=DFactory.getInstance();
  - D d = df.createNewD(c);
  - // infinite loop?
  - (Bonus: cercare “hammer factory factory” su Google)

# Glue-code

- Il codice che stiamo considerando non ha alcuna responsabilità se non “tenere insieme” tutti gli oggetti
- Oggetti che abbiamo progettato per essere il più possibile indipendenti tra loro
- Prende spesso il nome di Glue Code (“codice colla”) per questo motivo
- La logica di questo Glue Code va incapsulata all'interno di classi che non “sporcano” le classi originali, anche perché il glue code aggiunge dipendenze “definitive”

# Inversion of Control

- Ci viene in aiuto il pattern Inversion of Control (IoC)
- È un pattern astratto che afferma che la logica di esecuzione deve avvenire in senso opposto a come avverrebbe in un codice procedurale
  - In un codice procedurale abbiamo il controllo completo
  - Con l'Inversion of Control deleghiamo il controllo (per esempio della costruzione delle dipendenze) al sistema (Hollywood Principle, “Don't call us, we'll call you”)

# Spring

- Spring è un framework che implementa l'Inversion of Control tramite il pattern Dependency Injection
- Definendo la configurazione delle dipendenze, Spring IoC si occupa di creare tutti gli oggetti da cui un oggetto complesso può dipendere

# Dependency Injection

- La Dependency Injection funziona predisponendo dei metodi “setter” (oppure parametri del costruttore) che Spring si occuperà di chiamare
  - Es. se B dipende da A (possiede una variabile membro “a” di tipo A) basta predisporre un
    - b.setA(A a){ this.a = a; }
- Spring, per costruire B, si occuperà dapprima di costruire A e poi di “iniettarlo” in B
- In pratica il nostro glue code viene eliminato dal codice “utile” delle classi

# Componenti di Spring

- Spring in effetti non si occupa solo di IoC
- Altri componenti utili della parte “core”:
  - AOP: Aspect Oriented Programming
  - Transazioni
  - Accesso ai dati (wrapper per JDBC, Hibernate, etc)
- Inoltre ci sono altri progetti collegati:
  - Spring MVC / Spring Web Flow
  - Spring Integration
  - Spring Security, ecc

# Componenti di Spring (2)

- In generale i framework Spring-based sono progettati per essere poco invasivi
  - Le nostre classi che implementano la logica di dominio non devono dipendere da classi di framework esterni
  - Si possono usare anche “parti” del framework e non tutto intero

# Dependency Injection con Spring

- In linea di principio, per la Dependency Injection non è necessario usare un framework esterno, possiamo farlo tramite codice
  - Ogni volta che riempiamo un parametro di costruttore o usiamo un setter, in teoria stiamo iniettando dipendenze
- In pratica però ci può convenire se le dipendenze sono espresse fuori dal codice, utilizzando file di configurazione o annotazioni
- Spring è in grado di leggere la configurazione e poi, utilizzando Java Reflection oppure manipolando direttamente il bytecode, inserire le dipendenze mancanti

# Dependency Injection con Spring

- Il piccolo svantaggio è che la costruzione degli oggetti via Spring non viene più controllata dal compilatore
- Es. se costruiamo una dipendenza specificando un oggetto di tipo non valido, non ci saranno errori in fase di compilazione ma solo a runtime
- Un altro svantaggio è che il debug potrebbe essere più complicato

# BeanFactory

- BeanFactory è il container alla base di Spring
- Come dice il nome si occupa di costruire bean
  - Uno Spring Bean può essere un qualsiasi oggetto Java, non è necessario che sia un Java Bean
- Quindi BeanFactory si occupa di costruire (e restituirci) oggetti di ogni tipo, basta che siano configurati in Spring
- Non ci interessa sapere come fa a raggiungere lo scopo

# Configurazione XML

- Spring utilizza file XML per la configurazione dei bean
- Possiamo mettere tutti i bean dell'applicazione in un solo file
- Oppure possiamo definire più file XML, più BeanFactory, una per ogni settore dell'applicazione

# Esempio Configurazione XML

- <beans … >
  - <bean id=”oggettoA” class=”it.unirc.pja.esempio.OggettoA” />
  - <bean id=”oggettoB” class=”it.unirc.pja.esempio.OggettoB”>
    - <constructor-arg ref=”oggettoA” />
  - </bean>
- </beans>

# Costruire un bean

- Una volta che i bean sono configurati, possiamo costruirli con due passi:
  - Otteniamo un oggetto BeanFactory a partire dalla configurazione (BeanFactory bf)
  - Usiamo bf.getBean(String), passandogli il nome del bean configurato
    - Es. bf.getBean(“oggettoB”);

# Prototype e Singleton

- Per comportamento di default, i bean sono Singleton, cioè Spring ne costruisce una sola istanza e restituisce sempre quella
  - Ovvero, ogni volta che chiamiamo bf.getBean(“oggettoB”) non otteniamo un oggetto B “nuovo”, ma un riferimento al primo creato
- Se invece vogliamo che Spring crei un nuovo oggetto ogni volta che chiamiamo getBean(), dobbiamo impostarlo in modalità “prototype” con l'attributo del tag <bean>:
  - singleton=”false”

# Inizializzazione

- Quando un bean viene creato, può essere necessario eseguire una sua funzione interna che si occupa di inizializzazione
- Per questo c'è l'attributo del tag <bean>:
  - init-method=”nomeDelMetodo”
- Ad esempio se vogliamo che, dopo la creazione dell'oggetto, venga chiamata la funzione setup(), scriveremo:
  - init-method=”setup”

# Distruzione

- In Java non ci sono i distruttori, ma alcuni oggetti potrebbero aver necessità di funzioni similari
- Funzioni che si occupano di rilasciare risorse esterne (es. connessioni al DB, ecc)
- Per questo c'è l'attributo del tag <bean>:
  - destroy-method=”nomeDelMetodo”
- Ad esempio se vogliamo che, quando l'oggetto deve essere distrutto, venga chiamata la funzione destroy(), scriveremo:
  - destroy-method=”destroy”

# Simple Property Injection

- Conosciuta anche come “Injection tramite setter”
- Per iniettare nel nostro bean una proprietà (variabile membro) di tipo predefinito (int, String, e altri oggetti considerati semplici), basterà:
  - Definire il setter corrispondente
  - Aggiungere il tag <property> al nostro <bean>

# Simple Property Injection

- Es. abbiamo una proprietà “int maxVoto;” della classe Esame
  - Definiamo setMaxVoto(int maxVoto){...}
  - Scriviamo il bean:
    - <bean id=”esame” class=”it.unirc.pja.Esame”>
      - <property name=”maxVoto” value=”30” />
    - </bean>

# Reference Property Injection

- Possiamo anche iniettare, tramite setter, oggetti non basilari che abbiamo definito come bean
- Come negli esempi precedenti con OggettoA e OggettoB:
  - <bean id=”oggettoB” class=”it.unirc.pja.esempio.OggettoB”>
    - <property name=”a” ref=”oggettoA” />
  - </bean>
- L'attributo “ref” è un riferimento ad un altro bean

# Collection Property Injection

- Gli oggetti da iniettare possono anche essere Collection, come List, Map, Set
- È buona prassi utilizzare Java Generics nei setter, per evitare ambiguità
- Es.
  - public void setMap(Map<String, List<String>> map){...}

# Collection Property Injection (List)

- Le List si definiscono in questo modo:
  - <property name=”lista”>
    - <list>
      - <value>elemento1</value>
      - <value>elemento2</value>
      - <value>elemento3</value>
    - </list>
  - </property>

# Collection Property Injection (Set)

- I Set si definiscono in questo modo:
  - <property name=”insieme”>
    - <set>
      - <value>elemento1</value>
      - <value>elemento2</value>
      - <value>elemento3</value>
    - </set>
  - </property>

# Collection Property Injection (Map)

- Le Map si definiscono in questo modo:
  - <property name=”dizionario”>
    - <map>
      - <entry key=”chiave1”>
        - <value>elemento1</value>
      - </entry>
      - <entry key=”chiave2”>
        - <value>elemento2</value>
      - </entry>
    - </map>
  - </property>

# Collection Property Injection (Vari)

- Le Map<String, List<String>> si definiscono in questo modo:
  - <property name=”sinonimi”>
    - <map>
      - <entry key=”parola1”>
        - <list>
          - <value>sinonimo1</value>
          - <value>sinonimo2</value>
        - </list>
      - </entry>
    - </map>
  - </property>

# Null Injection

- Può capitare di dover inizializzare un bean con proprietà vuote oppure null
  - <!-- questo chiama setEmail(“”); (stringa vuota) -->
  - <property name=”email”>
    - <value />
  - </property>
  - 
  - <!-- questo chiama setEmail(null); →
  - <property name=”email”>
    - <null />
  - </property>

# Constructor Injection

- I bean possono non avere il costruttore di default (senza argomenti)
  - Infatti non devono essere Java Beans
- In tal caso per costruirli dobbiamo usare il tag <constructor-arg> al posto di <property>
- <bean name=”oggettoB” class=”it.unirc.pja.esempio.OggettoB”>
  - <constructor-arg ref=”oggettoA” />
- </bean>
- Questo chiama il costruttore di OggettoB passandogli un OggettoA

# Ambiguità nel Constructor Inj.

- I parametri del costruttore non hanno nome
- Finché c'è in solo parametro non ci sono problemi
- Quando ce n'è più di uno dobbiamo specificare l'ordine dei parametri
- <bean name=”oggettoB” class=”it.unirc.pja.esempio.OggettoB”>
  - <constructor-arg index=”0” ref=”oggettoA” />
  - <constructor-arg index=”1” value=”Ciao mondo” />
- </bean>
- <!-- questo chiama new OggettoB(new OggettoA(), “Ciao mondo”);

# Inner Bean Injection

- Un Inner Bean è un bean che non ha un nome ma è definito all'interno di altri bean
- Si può usare quando un bean ha senso solo nella costruzione di un altro bean e non ha vita indipendente
- <bean name=”oggettoB” class=”it.unirc.pja.esempio.OggettoB”>
  - <property name=”a”>
    - <bean class=”it.unirc.pja.esempio.OggettoA” />
  - </property>
- </bean>

# Constructor vs. setter

- Quale usare?
  - I parametri del costruttore rendono superflua la scrittura dei setter (a meno che i nostri oggetti non debbano essere Java Beans per altri motivi)
  - Se non ci sono i setter, le proprietà dell'oggetto sono immutabili dall'esterno
  - Se ci sono molte proprietà, il costruttore può diventare lungo ed illeggibile
  - I costruttori non sono molto adatti quando abbiamo l'ereditarietà

# Configurazione e XML Namespace

- La configurazione XML di Spring si basa su XML Schema, quindi ci sono svariate estensioni che si possono aggiungere ai tag base
- Ogni XML Schema aggiunto si associa ad un namespace (funzione simile alle Tag library di JSP)

# Namespace p:

- Il Property Name namespace (p:) ci permette di definire in modo più compatto la configurazione delle proprietà dei bean, definendole come attributi di <bean>
- La sintassi è
  - <bean … p:nomeProperty=”valore” />
- Esempio:
  - <bean … p:email=”abc@example.com” />
- Equivale a:
  - <bean ...><property name=”email” value=”abc@example.com ” /> </bean>

# Namespace p:

- Con il Property Name namespace si possono definire anche i riferimenti
- La sintassi è
  - <bean … p:nomeProperty-ref=”beanDiRiferimento” />
- Esempio:
  - <bean … p:a-ref=”oggettoA” />
- Equivale a:
  - <bean ...><property name=”a” ref=”oggettoA”></bean>

# Application Context

- ApplicationContext è un secondo tipo di container in Spring
- Un container con le stesse funzionalità di una BeanFactory, più altre (come la gestione dei messaggi di testo localizzabili, Resource Bundles)
- ApplicationContext implementa l'interface BeanFactory per cui si può usare indistintamente

# ApplicationContext (2)

- L'ApplicationContext può essere di vari tipi, a seconda della modalità di configurazione del file XML:
  - ClassPathXmlApplicationContext : specifica un file XML di configurazione che si trova nel classpath
  - FileSystemXmlApplicationContext : specifica un file XML di configurazione che si trova su un percorso preciso su disco

# ApplicationContext (3)

- ApplicationContext context = new ClassPathXmlApplicationContext(“context.xml”);
- context.getBean(“oggettoA”);
-
- ApplicationContext context = new FileSystemXmlApplicationContext(“c:/app/context.xml”);
- context.getBean(“oggettoA”);

# Spring: Accesso ai dati

# Accesso ai dati

- Abbiamo visto come, per mantenere disaccoppiata la logica di accesso ai dati dal resto del codice, si possano usare oggetti particolari chiamati DAO (Data Access Object)
- Tali oggetti concentrano al loro interno tutte le possibili chiamate per la persistenza dei dati, e sono quindi “punti di riferimento” per ogni classe che debba rendere persistenti degli oggetti

# Supporto ad Hibernate

- La SessionFactory di Hibernate rientra in questo modello: produce oggetti Session che svolgono tutte le funzioni di persistenza
- Sappiamo già che nel nostro codice possiamo creare una SessionFactory a partire da un file di configurazione
- Potremmo sfruttare l'iniezione di dipendenze di Spring per inserire la SessionFactory direttamente negli oggetti che ne fanno uso, senza doverla configurare ogni volta (copiando e incollando il codice)

# Configurazione Hibernate

- <bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
  - <property name="dataSource" ref="dataSource"/>
  - <property name="annotatedClasses"><list>
    - <value>test.package.Foo</value>
    - <value>test.package.Bar</value>
    - </list></property>
  - <property name="hibernateProperties"><value>
    - hibernate.dialect=org.hibernate.dialect.MySQL5Dialect
  - </value></property>
- </bean>

# Configurazione Hibernate (2)

- <bean id="dataSource"
  - class="org.springframework.jdbc.datasource.DriverManagerDataSource"
  - destroy-method="close"
  - p:driverClassName="org.gjt.mm.mysql.Driver"
  - p:url="jdbc:mysql://localhost/pja"
  - p:username="user"
  - p:password="password"/>

# SessionFactory Injection

- Una volta configurato Hibernate dall'interno di Spring, possiamo iniettare la SessionFactory all'interno di qualsiasi nostro oggetto
- Definendo il nostro oggetto come Spring Bean, un setSessionFactory() e un riferimento nella configurazione del bean
- <bean id=”userManager” class=”it.unirc.pja.esempio.UserManager”
  - p:sessionFactory-ref=”sessionFactory” />

# Spring-Hibernate Transaction

- Abbiamo semplificato l'utilizzo di Hibernate tramite Spring: adesso non dobbiamo più inizializzare manuamente una SessionFactory ogni volta che ci serve
- C'è però spazio per altri miglioramenti
- In particolare, ogni volta che apriamo una sessione dobbiamo aprire e chiudere una transazione
- Le transazioni sono funzionalità tipicamente gestite con l'Aspect Oriented Programming

# Spring-Hibernate Transaction (2)

- Spring ci mette a disposizione un particolare bean detto TransactionManager (txmanager)
- Questo bean viene richiamato con funzioni AOP tramite la java Annotation @Transactional
- Se annotiamo un metodo con @Transactional, Spring si occuperà di aprire la transazione all'inizio e chiuderla alla fine

# Spring-Hibernate Transaction (3)

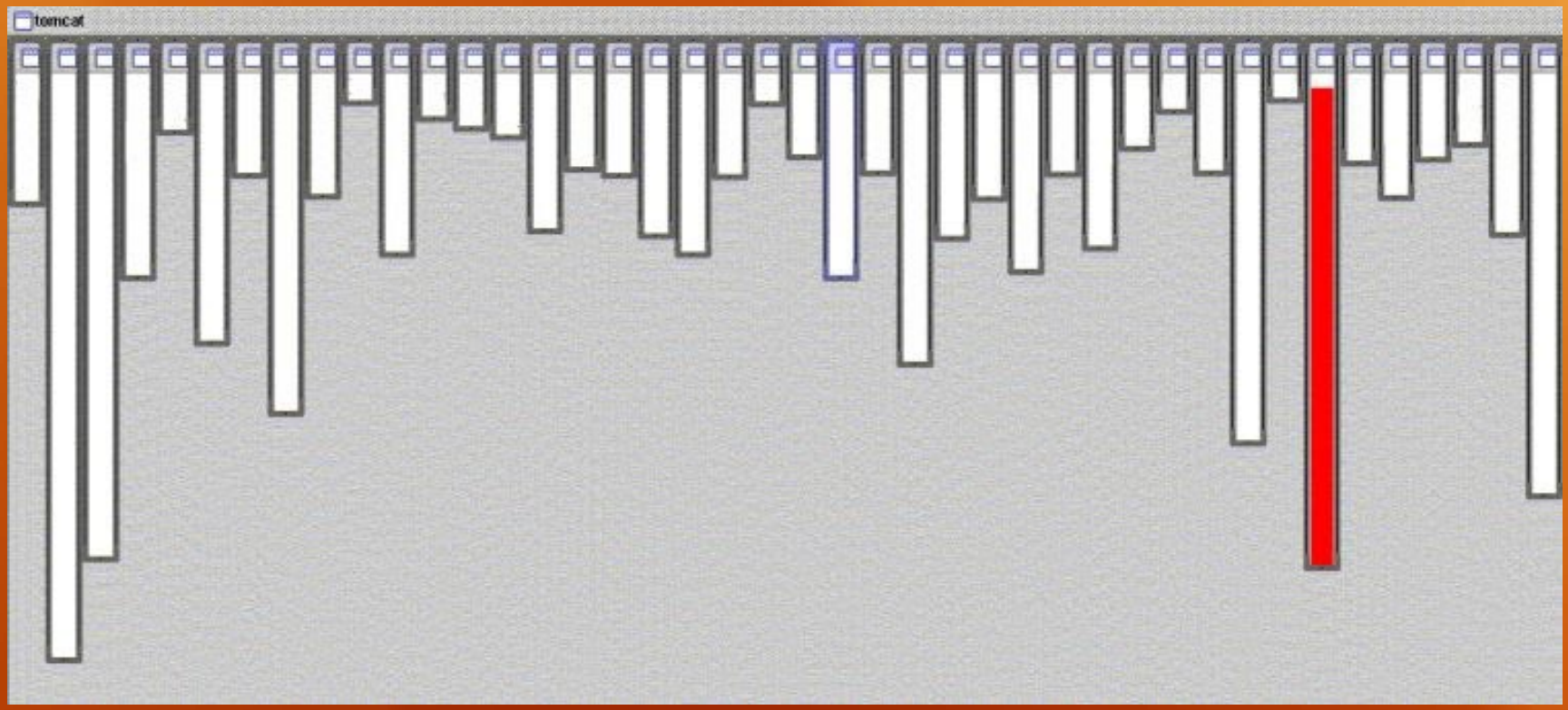
- Il txmanager si può configurare in modo da usare JDBC, Hibernate o altri ORM
- Ecco la configurazione per Hibernate:
- <bean id="txManager" class="org.springframework.orm.hibernate3.HibernateTransactionManager">
  - <property name="sessionFactory" ref="sessionFactory"/>
- </bean>
- <tx:annotation-driven transaction-manager="txManager"/>

# Spring AOP

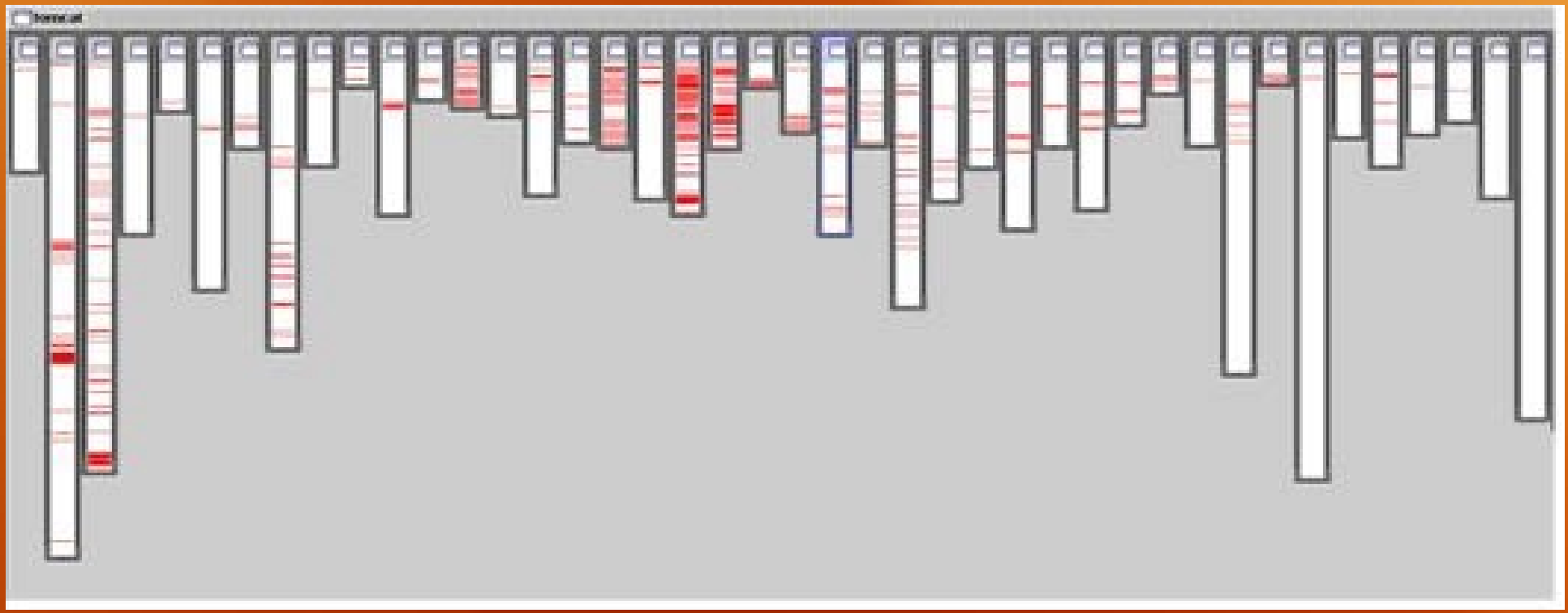
# AOP: Perchè

- L'Aspect Oriented Programming nasce per implementare comportamenti che non si adattano molto bene al modello ad oggetti
- Problemi che sono cross-cutting concerns, comuni a più oggetti che non sono in relazione tra loro
  - Esempi classici: Logging, autenticazione, transazioni...

# Limiti dell'OOP (Lettura XML)



# Limiti dell'OOP (Logging)



# Code Scattering

- Il Code Scattering è la distribuzione del codice che riguarda uno stesso “aspetto”, su più moduli
- Può essere di due tipi:
  - Codice identico, copiato e incollato
  - Codice complementare, ogni modulo esegue una parte dell'aspetto

# Code Tangling

- Il Code Tangling si ha, invece, quando un certo modulo ha più di una responsabilità contemporaneamente
  - Per esempio, deve gestire la sua responsabilità base e poi le eccezioni, il logging, l'autenticazione, ecc

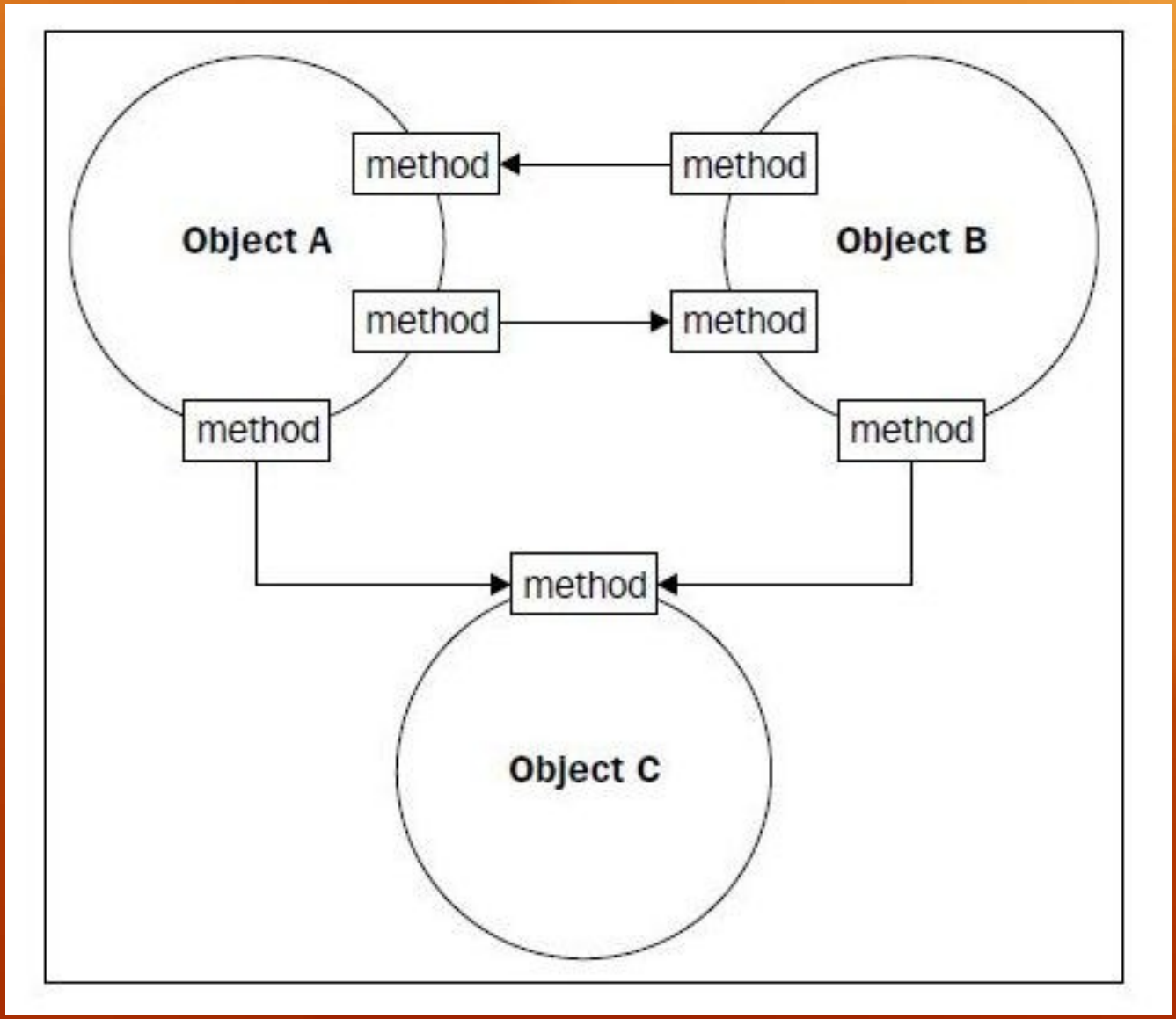
# Effetti negativi con l'OOP

- Evoluzione difficoltosa: i moduli non sono disaccoppiati tra loro
- Bassa Qualità: Se si verifica un problema, non è immediato capire qual è la responsabilità di un certo modulo
- Codice non riusabile: se l'implementazione contiene varie responsabilità, non sarà riusabile in un altro scenario
- Produttività: il codice che riguarda un aspetto è difficile da trovare e da modificare

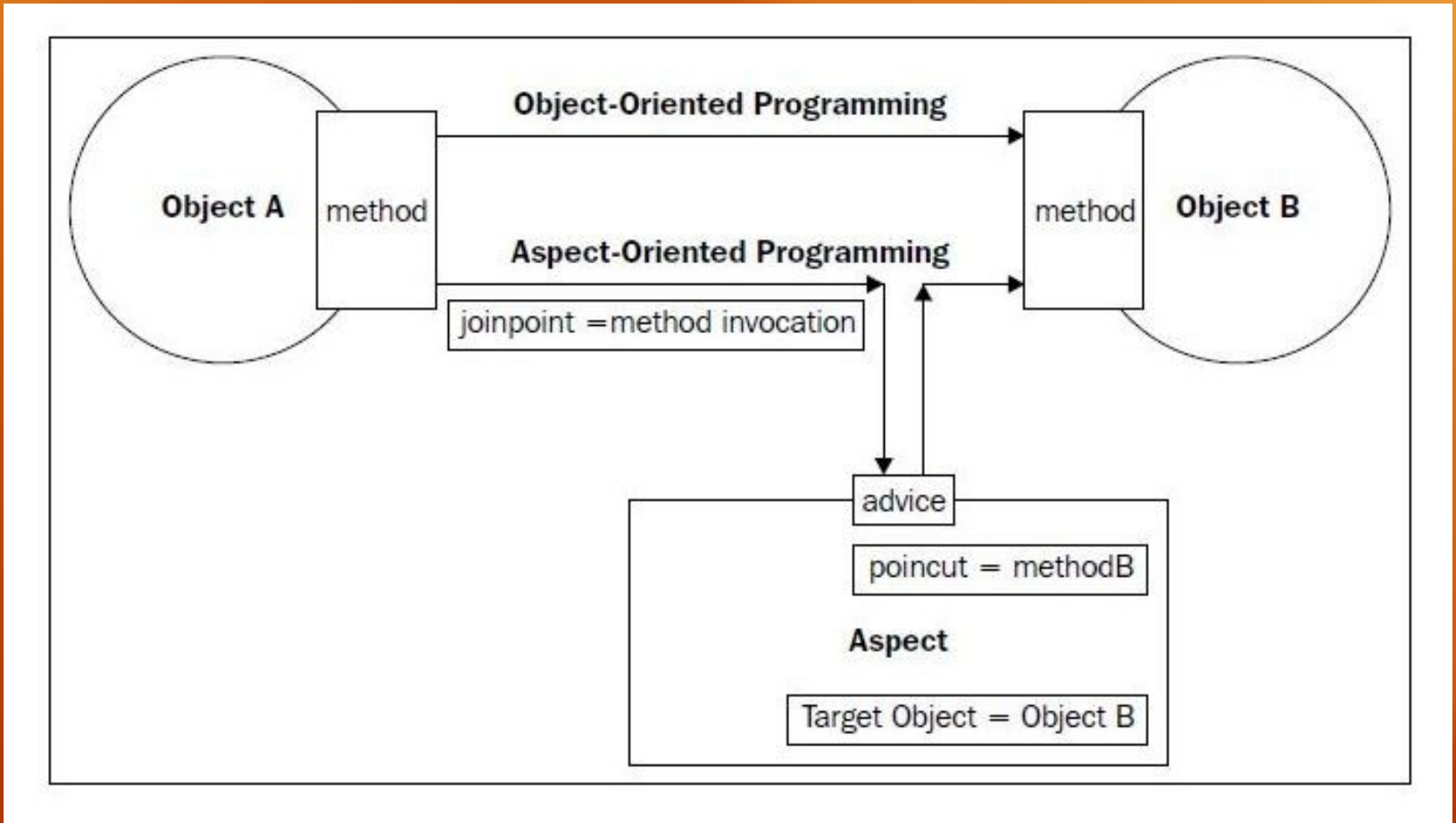
# Effetti positivi con l'AOP

- La modularizzazione dei cross-cutting concern
- Disaccoppiamento dei moduli
- Rimozione della dipendenza dei moduli funzionali dai moduli che gestiscono gli aspetti

# Funzioni OOP



# Funzioni AOP



# AOP: Concetti Base

- Aspect
- Joinpoint
- Advice
- Pointcut
- Target object
- Weaving

# AOP: Concetti Base

- **Aspect**
- Joinpoint
- Advice
- Pointcut
- Target object
- Weaving

# AOP: Aspect

- Un Aspect è l'equivalente di una classe
- Contiene al suo interno la definizione del cross-cutting concern
- Definisce il COSA

# AOP: Concetti Base

- Aspect
- **Joinpoint**
- Advice
- Pointcut
- Target object
- Weaving

# AOP: Joinpoint

- Il Joinpoint definisce il punto della nostra applicazione in cui va applicato l'aspetto
- Ovvero al verificarsi di quale evento vengono lanciate le funzionalità Spring AOP
- Definisce il QUANDO
- Nel caso di Spring un Joinpoint è l'esecuzione di un metodo

# AOP: Concetti Base

- Aspect
- Joinpoint
- **Advice**
- Pointcut
- Target object
- Weaving

# AOP: Advice

- Un Advice è l'implementazione della funzionalità che verrà applicata
- Definisce il COME
- Gli Advice possono essere di diverso tipo:
  - Before: la funzionalità è eseguita prima del metodo
  - After: la funzionalità è eseguita dopo il metodo
  - Around: la funzionalità è eseguita sia prima che dopo

# AOP: Concetti Base

- Aspect
- Joinpoint
- Advice
- **Pointcut**
- Target object
- Weaving

# AOP: Pointcut

- Definisce il Joinpoint esatto (o l'insieme dei Joinpoint) in cui applicare l'aspetto
- Definisce il DOVE
- Il Pointcut, in Spring AOP, in genere è un'espressione regolare che identifica il metodo da tracciare

# AOP: Concetti Base

- Aspect
- Joinpoint
- Advice
- Pointcut
- **Target object**
- Weaving

# AOP: Target Object

- Il Target Object è il modulo, la classe a cui l'aspetto verrà applicato

# AOP: Concetti Base

- Aspect
- Joinpoint
- Advice
- Pointcut
- Target object
- **Weaving**

# AOP: Weaving

- Il weaving è il meccanismo con cui il framework applica gli aspetti ai target object
- In genere può avvenire tramite Java Reflection, oppure tramite manipolazione diretta del bytecode compilato

# Spring AOP

- Tirando le somme:
  - Si implementano gli Advice come funzioni
  - Si definiscono i Pointcut tramite espressioni regolari
- Il framework si occupa del resto

# Creare gli Advice

- Un Advice in Spring AOP è una classe che implementa una delle seguenti interface:
  - MethodBeforeAdvice
  - AfterReturningAdvice
  - MethodInterceptor
  - ThrowsAdvice

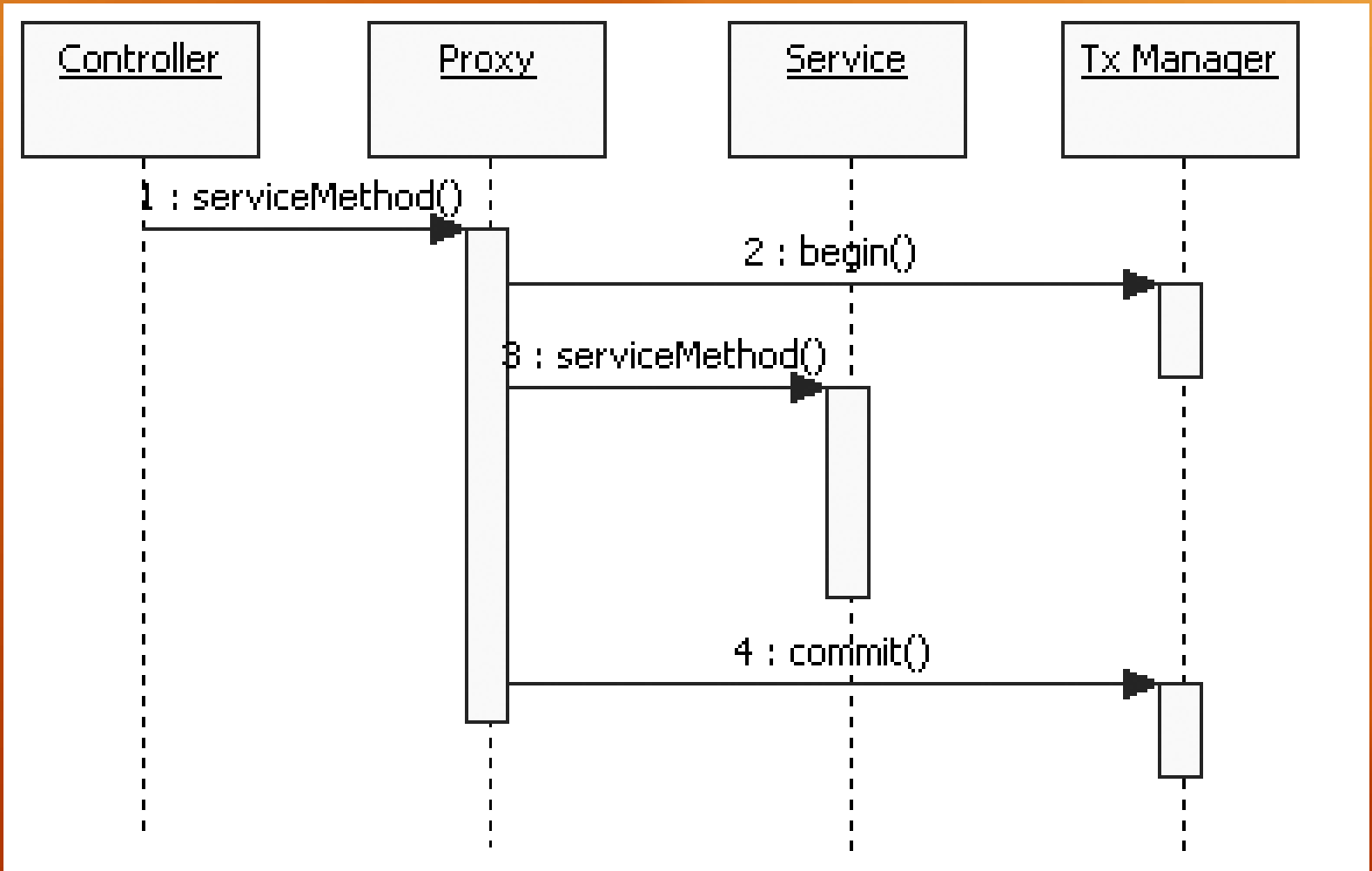
# Creare gli Advice

- Ognuna di quelle interface contiene dei metodi da implementare per realizzare l'advice

# Unire gli Advice ai Target: Proxy

- Per unire gli Advice ai Target Object si utilizza un pattern chiamato Proxy
- Il Proxy è un “wrapper”, si comporta in modo del tutto identico all'oggetto che sta mascherando
- Ma ha la possibilità di aggiungere funzionalità ad ogni funzione chiamata

# Unire gli Advice ai Target: Proxy



# Esempio programmatico

- Con ProxyFactory (vedi)