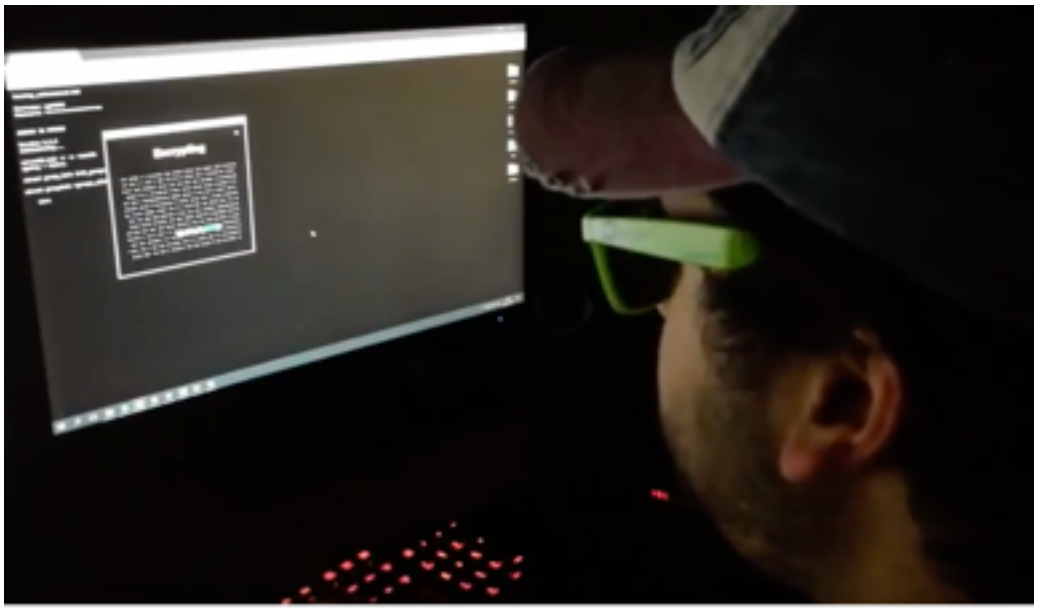
J2E++: Interceptors, MOMs

Sébastien Mosser et Anne Marie Dery

pictures: sxc.hu



Intercepting Messages

Man in the middle
(without the hoody)


1. PRINCIPES

Intercepting messages

Une implémentation du concept d'

Aspect Oriented Programming
(Xerox Parc, Gregor Kickzales)

AspectJ extension de Java.
IBM en 2001 propose HyperJ.

La programmation orientée aspect a vu le jours dans les années 2001 dans la mouvance de la séparation des préoccupations dans le code et pour augmenter la modularité des langages à objets

Aspect Oriented Programming

Permet

- d'augmenter la modularité
- de séparer des préoccupations orthogonales au code

Toujours le même objectif :

Ne pas polluer la logique métier par des comportements relatifs à des préoccupations fonctionnelles différentes :

traces, sécurité, persistance….

L'objectif reste toujours de séparer le code métier de tout code qui est ajouté pour gérer des préoccupations fonctionnelles indispensables mais indépendantes du cœur du métier telles que : l'ajout de trace pour mieux comprendre le code (code qui peut être enlevé à la livraison), la gestion de la persistance (code qui peut être changé – sauvegarde dans un fichier, dans une base de données, le niveau de sécurité qui peut être adapté et modifié.

A.O.P. Comment ?

En spécifiant les comportements à ajouter à un code existant.
advice code « à insérer » dans le code sans le modifier

En spécifiant le code à modifier
pointcut pour désigner où « insérer » un advice

Exemple une **advice** prenant en charge une préoccupation : *« log » peut être « insérée » au code existant à chaque appel d'accesseur en écriture (set*)*

Ce qui est préconisé est d'écrire le code des préoccupations séparément dans des « advices » et de désigner l'endroit où injectr le cide dans le code métier existant. Ainsi le même « advice » peut être injecté à différents endroits dans le code métier. On ne l'écrit qu'une fois et il alors plus facile à maintenir, modifier ou enlever. Pensez à toutes les parties de code que vous auriez à modifier pour enlever les traces sinon.

Terminologie

Cross-cutting concerns : « fonctionnalités secondaires / préoccupations orthogonales » qui peuvent être partagées par plusieurs classes

Advice : code additionnel gérant un *cross-cutting concern* à appliquer au code métier existant.

Pointcut : le point d'exécution où le cross-cutting concern doit être appliqué, l'advice correspondant ajouté.

Aspect : advices et pointcuts.

Weaver : tissage du des advices dans le code pour obtenir le code final

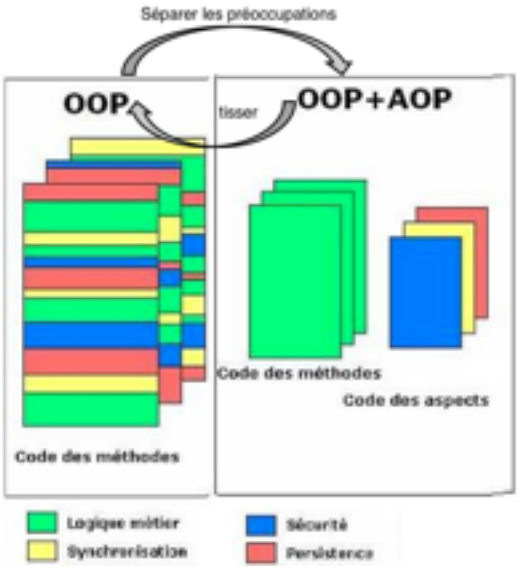
Terminologie : exemple du log

Cross-cutting concerns : log / trace

Advice : le code du log que l'on veut appliquer (afficher la valeur avant l'exécution et la valeur après)

Pointcut : avant l'exécution et après l'exécution des accesseurs en écriture

Principes



Ce schéma illustre le tissage d'un code objet OOP + AOP donne le code objet que vous avez l'habitude d'écrire. Un code objet dans lequel les classes ne g§rnt que le métier est rare même si l'utilisation de design patterns ou de bons principes de programmation peut faciliter la séparation des préoccupations.

pictures: sxc.hu

AOP : Intercepteurs et EJB

Intercepteurs
⇒ support pour développer des préoccupations orthogonales (cross-cutting features) et
diminuer la duplication de code au niveau du code métier.

Permettent d'intercepter
- des opérations,
- des services…

Pour insérer des pré ou post actions au code existant

Mise en œuvre

Pour placer des pointcuts :

- Utiliser l'annotation **@Interceptors** au bon niveau d'interception (opération, service…) ou
- Définir dans un fichier l'expression régulière à appliquer pour retrouver les beans à intercepter

Pour insérer des advices

Utiliser l'annotation **@AroundInvoke**
associée à l'opération **proceed** sur le contexte
proceed appelle le corps de l'opération interceptée
Selon sa place, on a une pré action, une post action ou les 2

@interceptors permet de désigner les pointcuts et
@AroundInvoke + proceed à écrire l'advice

Exemples : TCF

1. Intégrer des statistiques au code métier
2. Vérifier les données reçues
3. Tracer des exécutions

Les 3 «exemples permettent d'illustrer 3 façons différentes d'intercepter le code métier

Statistiques dans TCF

1. Pour des besoins statistiques

compter le nombre de cartes

A chaque exécution de l'opération **validate** sur une ***Cart***, un compteur doit être incrementé si l'opération s'est bien passée

Statistiques : Annotation et post action

```
public class CartCounter implements Serializable {

        @EJB private Database memory;

        @AroundInvoke
        public Object intercept(InvocationContext ctx) throws Exception {
                Object result = ctx.proceed();  // do what you're supposed to do
                memory.incrementCarts();
                System.out.println("  #Cart processed: " + memory.howManyCarts());
                return result;
        }

}
```

```
@Override
@Interceptors({CartCounter.class})
public String validate(Customer c) throws PaymentException
        { return cashier.payOrder(c, contents(c)); }
```

Identifiez le code de l'advice et le pointcut dans cet exemple. Posez vous la question de si ce type d'interception pourrait vous permettre d'ajouter des statistiques à moindre coût de développement dans votre système

Vérification de valeurs dans TCF

Gérer les pré-conditions d'une opération

Eviter de pouvoir ajouter ou retirer à une carte donnée un item avec une valeur négative ou un montant nul de cookies

⇒ Intercepter l'invocation des opérations du service CartWebService et vérifier les paramètres

Vérification de valeurs : Annotation et pré action

```
public class ItemVerifier {

        @AroundInvoke
        public Object intercept(InvocationContext ctx) throws Exception {

                Item it = (Item) ctx.getParameters()[1];
                if (it.getQuantity() <= 0) {
                        throw new RuntimeException("Inconsistent quantity!");
                }

                return ctx.proceed();
        }

}
```

```
@WebMethod
@Interceptors({ItemVerifier.class})
void addItemToCustomerCart(@WebParam(name = "customer_name") String customerName,
                                                @WebParam(name = "item") Item it)
        throws UnknownCustomerException;
```

Identifiez le code de l'advice et le pointcut dans cet exemple. Posez vous la question de si ce type d'interception pourrait vous permettre d'ajouter des des pré conditions à moindre coût de développement dans votre système

Tracer l'exécution dans TCF

Tracer toutes les opérations invoquées dans le système.

Ne pas annoter toutes les opérations à la main
=> Remplir **ejb-jar.xml** (dans le répertoire **resources**),
définir l'expression régulière associée à cet intercepteur.
⇒ Le container va mettre en place l'intercepteur

Tracer l'exécution : pre et post actions

```
public class Logger implements Serializable {

	@AroundInvoke
	public Object methodLogger(InvocationContext ctx) throws Exception {
		String id = ctx.getTarget().getClass().getSimpleName() + "::" + ctx.getMethod().getName();
		System.out.println("*** Logger intercepts " + id);
		try {
			return ctx.proceed();
		} finally {
			System.out.println("*** End of interception for " + id);
		}
	}

}
```

Identifiez le code de l'advice et le pointcut dans cet exemple. Posez vous la question de si ce type d'interception pourrait vous permettre d'ajouter traces s à moindre coût de développement dans votre système. Cela pourrait il être utile pour mettre et enlever des traces utiles dans votre prochaine démonstration ?

Tracer l'exécution : expression régulière

Peut être faudrait il une autre expression irrégulière dans votre cas ?

```java
public class CartCounter implements Serializable {

	@EJB private Database memory;

	@AroundInvoke
	public Object intercept(InvocationContext ctx) throws Exception {
		Object result = ctx.proceed();  // do what you're supposed to do
		memory.IncrementCarts();
		System.out.println("  #Cart processed: " + memory.howManyCarts());
		return result;
	}

}
```

Post action

```java
public class ItemVerifier {

	@AroundInvoke
	public Object intercept(InvocationContext ctx) throws Exception {

		Item it = (Item) ctx.getParameters()[1];
		if (it.getQuantity() <= 0) {
			throw new RuntimeException("Inconsistent quantity!");
		}

		return ctx.proceed();
	}
```

Pre action

```java
public class Logger implements Serializable {

	@AroundInvoke
	public Object methodLogger(InvocationContext ctx) throws Exception {
		String id = ctx.getTarget().getClass().getSimpleName() + "::" + ctx.getMethod().getName();
		System.out.println("**** Logger intercepts " + id);
		try {
			return ctx.proceed();
		} finally {
			System.out.println("**** End of interception for " + id);
		}
	}

}
```

Post et pre actions

Références

- https://www.eclipse.org/aspectj/doc/next/progguide/starting-aspectj.html

- http://vitiy.info/c11-functional-decomposition-easy-way-to-do-aop/

Message-oriented Middleware

EJB Messages


1. PRINCIPES

MOM : Pourquoi ?

Communication asynchrone entre composants.
L'appelant et l'appelé n'ont pas besoin d'être présents pour que la communication réussisse

L'appelant n'a pas besoin d'attendre la fin de la communication pour continuer

Différent de l'invocation de méthode et de Java RMI.

Message-oriented middleware (MOM) : intermédiaire entre l'expéditeur du message et le destinataire

La communication synchrone est parfois une contrainte. La communication asynchrone permet de décorreler appelant et appelé. Vous connaissez déjà ce principe ;: pour développer des IHM, en réseau… Les MOM sont des middleware basés sur la communication par message

The **Messaging** paradigm

https://aws.amazon.com/fr/message-queue/

MOM : Principe

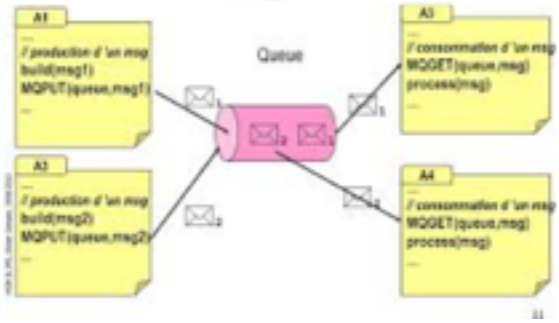
Envoi du message =>

1. stockage du message dans un lieu (Destination) spécifié par l'expéditeur (Producer)

2. un accusé de réception est tout de suite envoyé.

3. Tout composant (Consumer) intéressé par le message à cette destination peut récupérer le message envoyé

MOM: Message-oriented Middleware

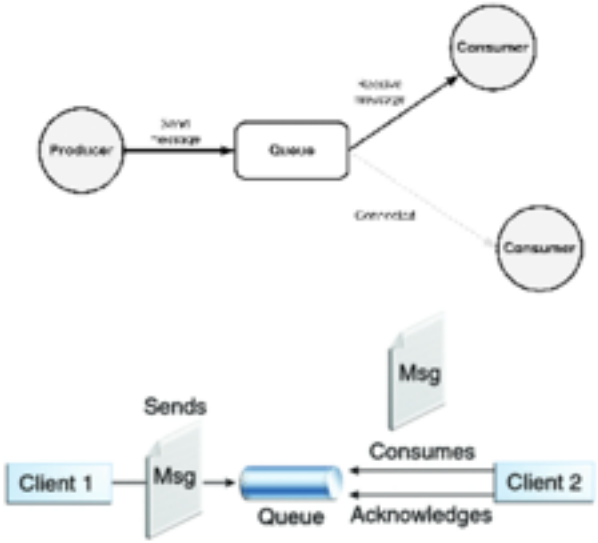


Point à Point : Principe

Un seul message d'un Producer vers un Consumer
Les destinations de messages sont appelées queues

Aucune garantie que les messages soient délivrés dans un ordre particulier

Model: Point-to-Point



Publish-Subscribe : principe

.

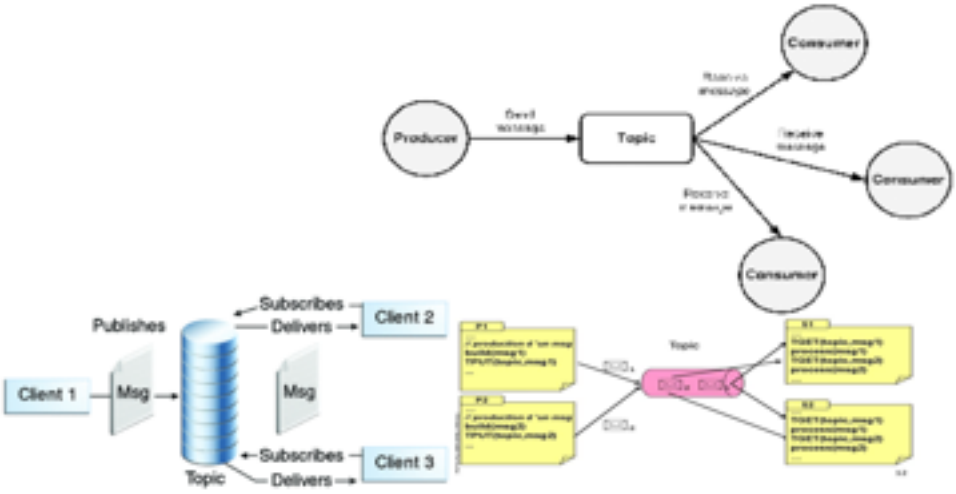
Un Producer produit un message reçu par un nombre non déterminé de Consumers

La destination du message est un Topic
Le consommateur est un subscriber.

Utile pour le broadcast d'information.

Ex: notification de maintenance

Modele : Publish-Subscribe



pictures: sxc.hu

Dans les EJB

Un nouveau type de Composants : les composants orientés messages (**M**essage **D**riven **B**ean)

Il y a en fait 3 types de composants : les composants Stateless , stateful et orientés message

Envoi d’un message à un MDB

Chaque Bean orienté message a une queue de messages associée dont le nom est donné

Pour invoquer un tel bean il faut envoyer un message à cette queue.

La queue est obtenue par l’injection d’une dépendance.

La queue est automatiquement gérée par le container (*e.g.*, starting the queues, stopping it, reading messages, passivating elements)

Message-driven bean lifecycle

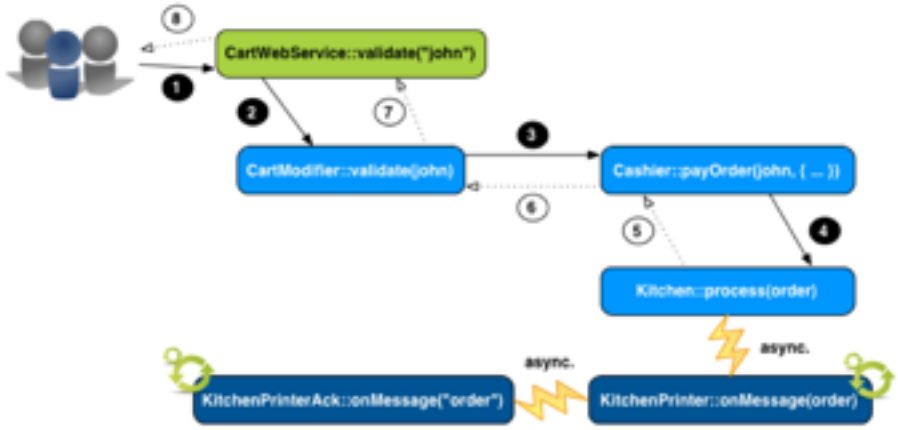
http://what-when-how.com/enterprise-javabeans-3/working-with-message-driven-beans-part-2-ejb-3/

Cycle de vie : les étapes

Creates MDB instances and sets them up
■ Injects resources, including the message-driven context
■ Places instances in a managed pool
■ **Pulls** an idle bean out of the pool ***when a message arrives***
■ Executes the message listener method; e.g., the onMessage method
■ When the onMessage method finishes executing, **pushes** the idle bean back into the "method-ready" pool
■ As needed, retires (or destroys) beans out of the pool

Exemple TCF

Envoi d'une commande à la cuisine équipée d'une imprimante physique qui imprime les commandes reçues de la caisse. Les opérations suivantes ne devraient pas attendre.



Dans votre système pouvez vous identifier les communications qui mériteraient à être asynchrone ? Existe-t-il un composant qui pourrait devenir un MDB ?

Handling objects

Un Message peut être :
- un TextMessage,
- un ObjectMessage ….

Un ObjectMessage doit contenir un objet Serializable.

Ex: envoyer un texte ou une instance de Order à l'imprimante

Exemple: Text-based receiver

Bean KitchenPrinterAck affiche les identifiants associés à l'ordre d'impression.

```
@MessageDriven
public class KitchenPrinterAck implements MessageListener {

    // ...

  public void onMessage(Message message) {
    try {
      String data = ((TextMessage) message).getText();
      System.out.println("\n\n****\n** ACK: " + data + "\n****\n");
    } catch (JMSException e) {
      throw new RuntimeException("Cannot read the received message!");
    }
  }
}
```

Handling objects : Order

```
public void onMessage(Message message) {
  try {
    Order data = (Order) ((ObjectMessage) message).getObject();
    handle(data);
  } catch (JMSException e) {
    throw new RuntimeException("Cannot print …");
  }
}

private void handle(Order o) throws IllegalStateException {
   ....

}
```

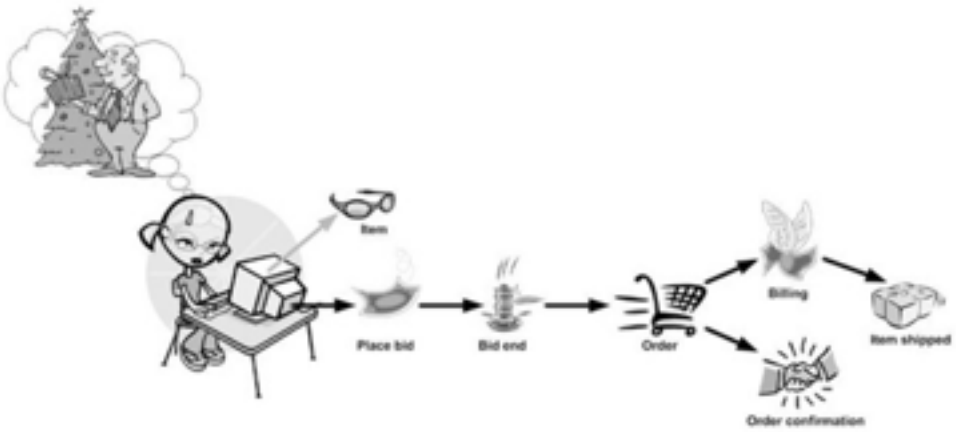
Mise en œuvre par annotations

```
@MessageDriven

Interface MessageListener {
   public void onMessage(Message message) {



@Resource private ConnectionFactory
connectionFactory;
@Resource(name = "KitchenPrinterAck")
private Queue acknowledgmentQueue; (); }

// ...
```

Sending a message to a MDB

```
@Resource private ConnectionFactory connectionFactory;
@Resource(name = "KitchenPrinterAck") private Queue q;

private void acknowledge(int orderId) throws JMSException {
  Connection connection = null; Session session = null;
  try {
    connection = connectionFactory.createConnection();
    connection.start();
    session =
      connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
    MessageProducer producer = session.createProducer(q);
    producer.setDeliveryMode(DeliveryMode.NON_PERSISTENT);
    producer.send(session.createTextMessage(orderId + ";PRINTED"));
  } finally {
    if (session != null) session.close();
    if (connection != null) connection.close();
  }
}
```

The ActionBazaar example



http://what-when-how.com/enterprise-javabeans-3/messaging-concepts-ejb-3

The ActionBazaar example

ActionBazaar ordering before MOM is introduced. Slow B2B processing is causing customer dissatisfaction.

Introducing messaging

ActionBazaar ordering after MOM is introduced. Messaging enables both fast customer response times and reliable processing.

Références

- Patterns Message https://www.enterpriseintegrationpatterns.com/patterns/messaging/toc.html
- Cours Françoise Baude https://www.i3s.unice.fr/~baude/AppRep/MOM.pdf
- Cours Didier Donsez https://docplayer.fr/997299-Message-oriented-middleware-mom-java-message-service-jms-didier-donsez.html
- TCF https://github.com/polytechnice-si/4A_ISA_TheCookieFactory/blob/develop/chapters/MessageDrivenBeans.md
- JMS
- https://docs.oracle.com/javaee/6/tutorial/doc/bncdx.html
- EJB : http://what-when-how.com/enterprise-javabeans-3/messaging-concepts-ejb-3/