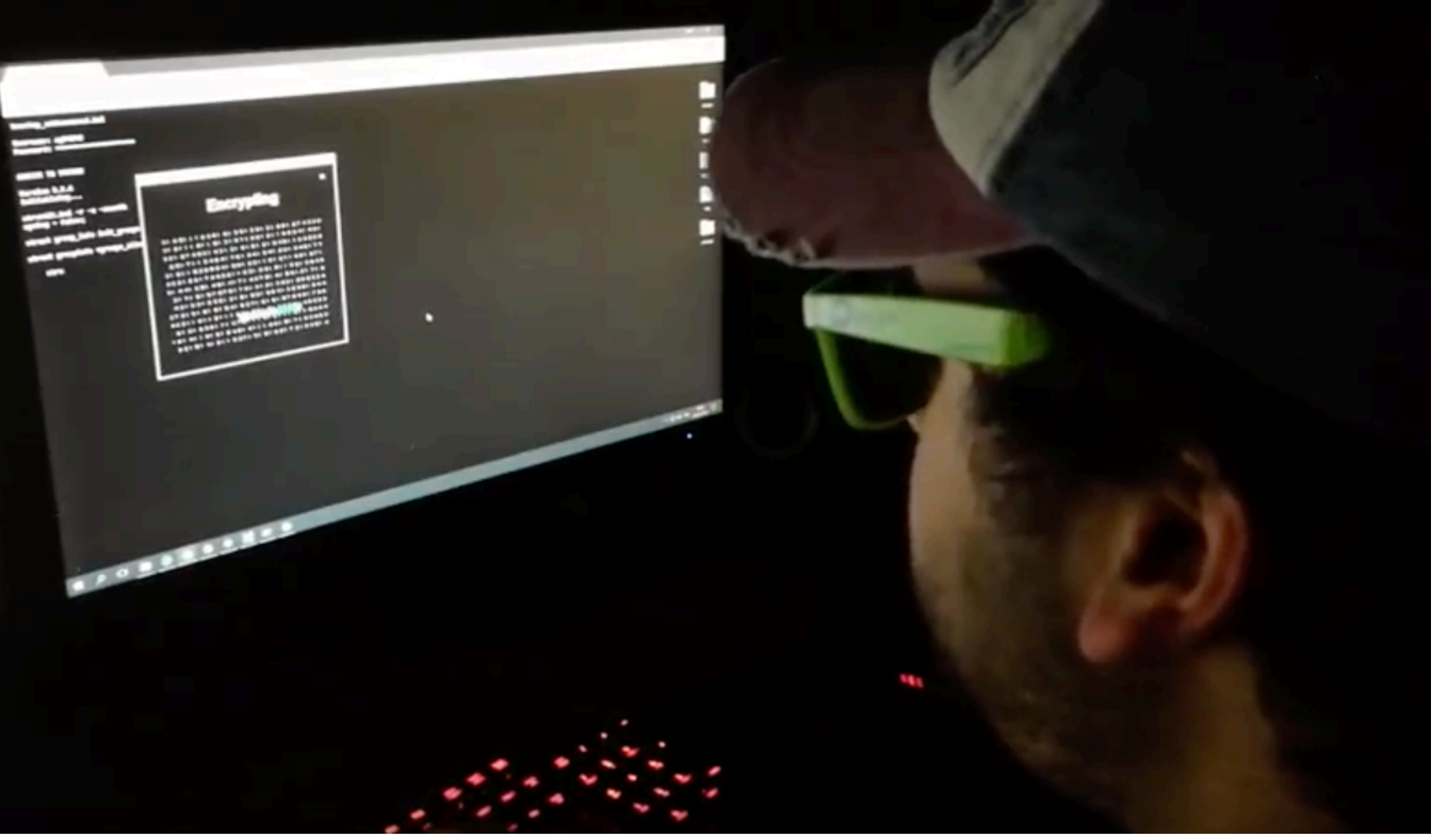
J2E++: Interceptors, MOMs

Sébastien Mosser et Anne Marie Dery

pictures: sxc.hu



Intercepting Messages

Man in the middle
(without the hoody)

Intercepting messages

Aspect Oriented Programming
(Xerox Parc, Gregor Kickzales)

AspectJ extension de Java.
IBM en 2001 propose HyperJ.

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….

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*)*

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

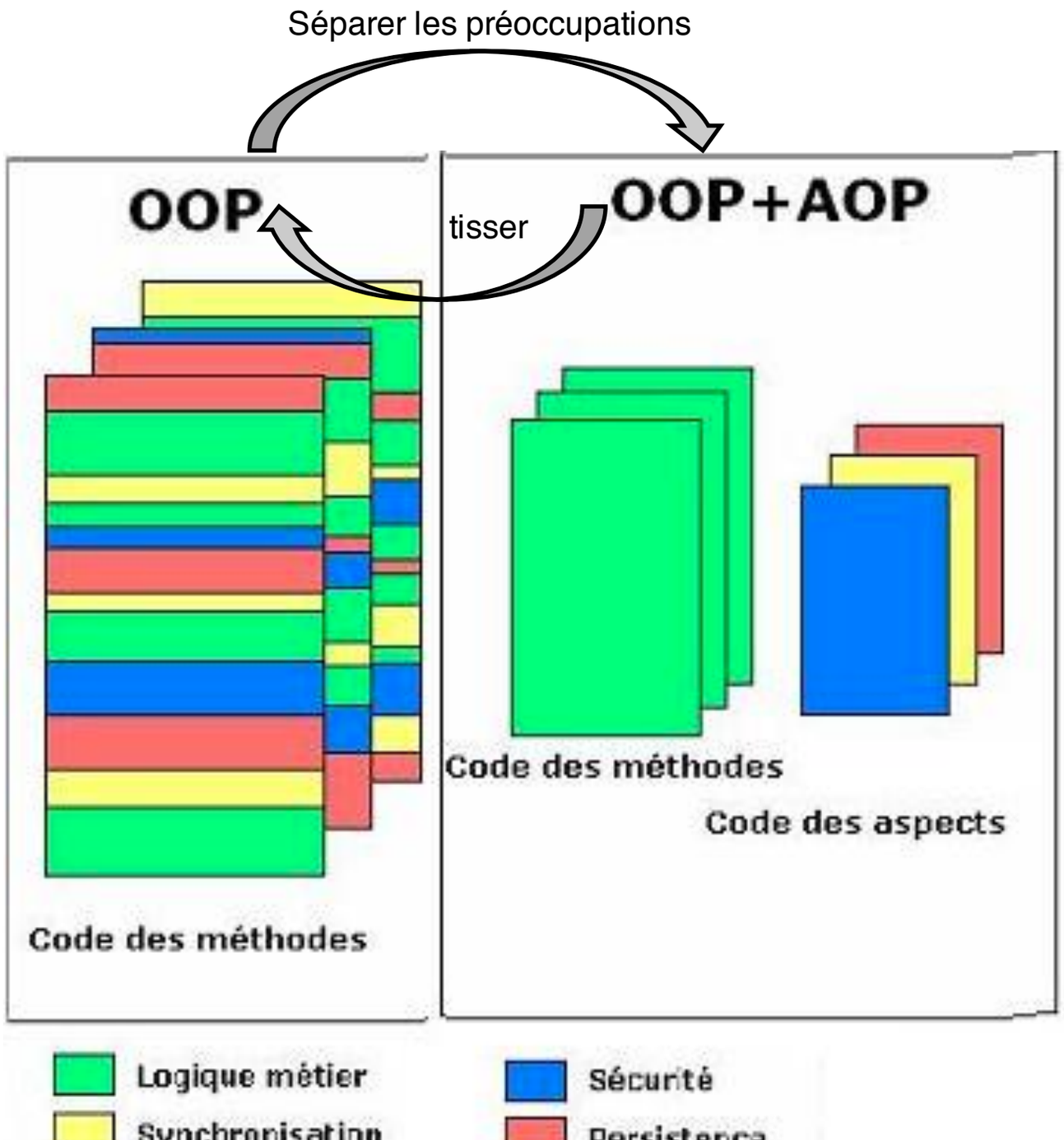
Terminologie : exemple du log

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


Séparer les préoccupations
OOP
tisser
OOP+AOP
Code des méthodes
Code des méthodes
Code des aspects
Logique métier
Synchronisation
Sécurité
Persistence

AOP : Intercepteurs et EJB

https://www.jmdoudoux.fr/java/dej/chap-ejb3.htm#ejb3-11

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

Exemples : TCF

1. Intégrer des statistiques au code métier
   pré et/ou post action de quelles opérations ?

2. Vérifier les données reçues
   pré et/ou post action de quelles opérations ?

3. Tracer des exécutions
   pré et/ou post action de quelles opérations ?

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)); }
```

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;
```

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);
		}
	}

}
```

Tracer l’exécution : expression régulière

Toutes les opérations du système

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd
<ejb-jar>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-class>fr.unice.polytech.isa.tcf.interceptors.Logger</interceptor-class>
        </interceptor-binding>
    </assembly-descriptor>
</ejb-jar>
```

```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;
	}

}
```

Statistique

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();
	}
}
```

Vérification

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);
		}
	}

}
```

Trace

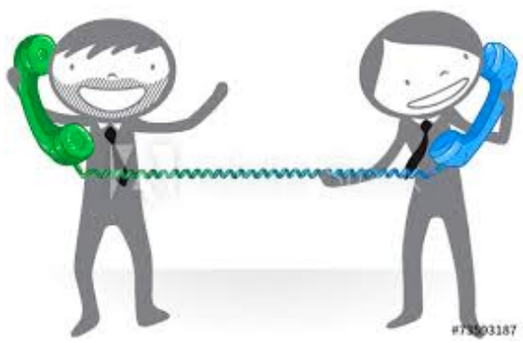
Pre et Post action

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



MOM : Pourquoi ?

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

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

Message-Oriented Middleware (MOM) : intermédiaire entre l’expéditeur du message et le destinataire

The **Messaging** paradigm

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

MOM : Principe

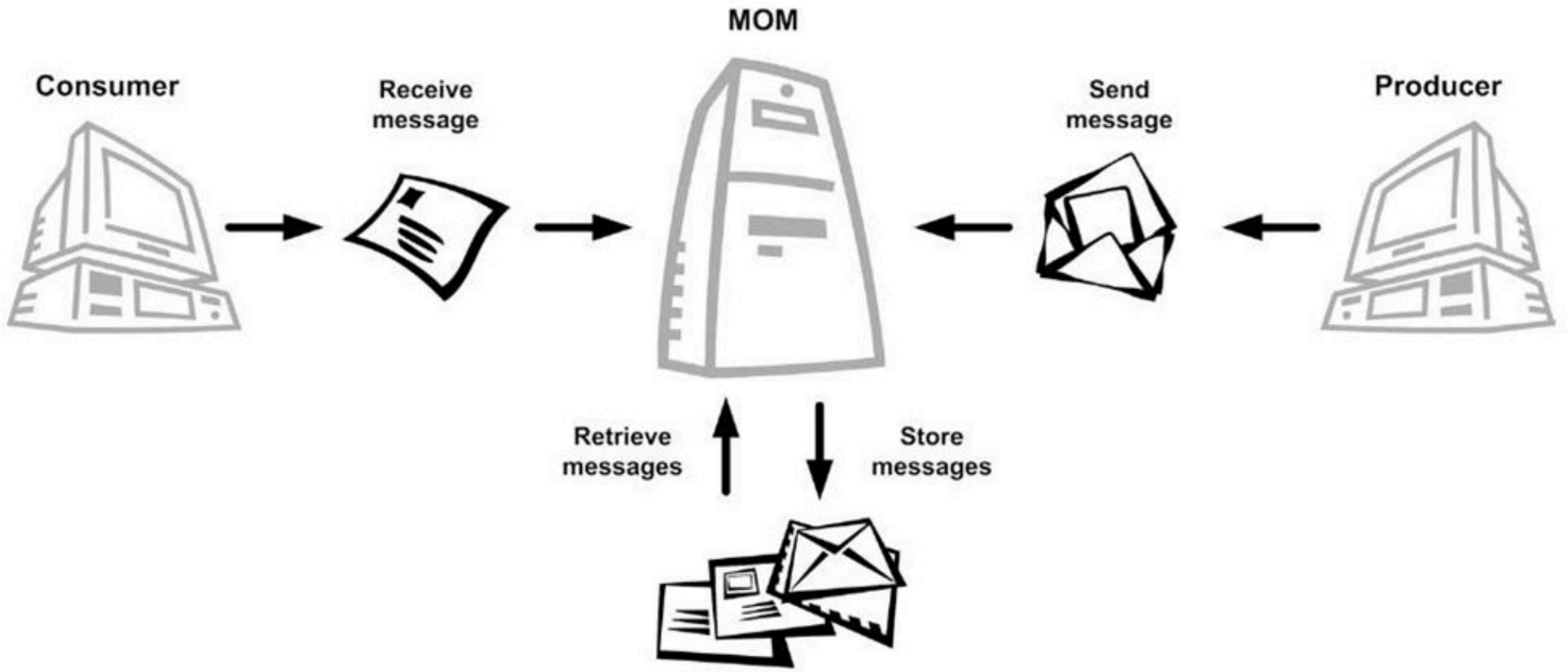
Envoi du message =>

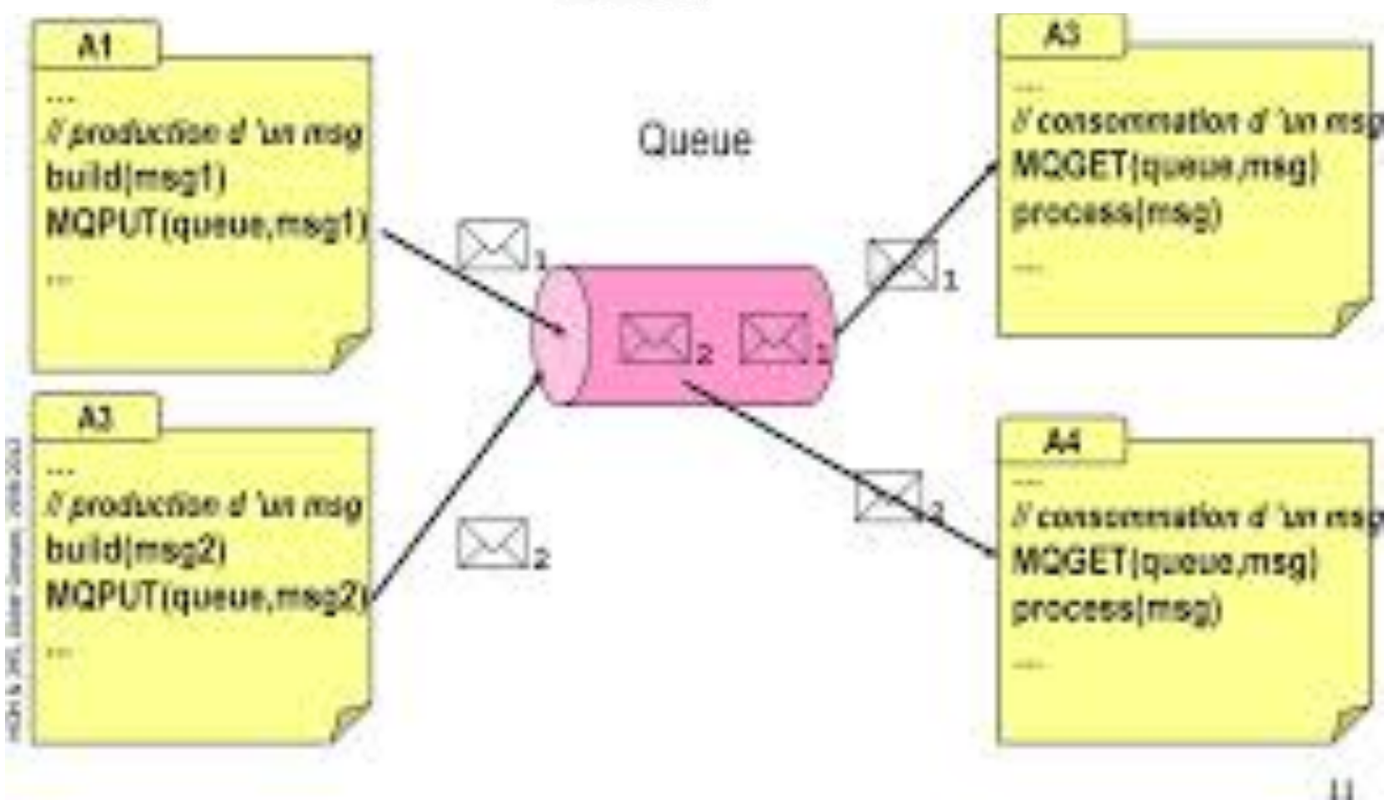
1. stockage du message dans un lieu (Destination) spécifié par l'expéditeur (Producer)

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



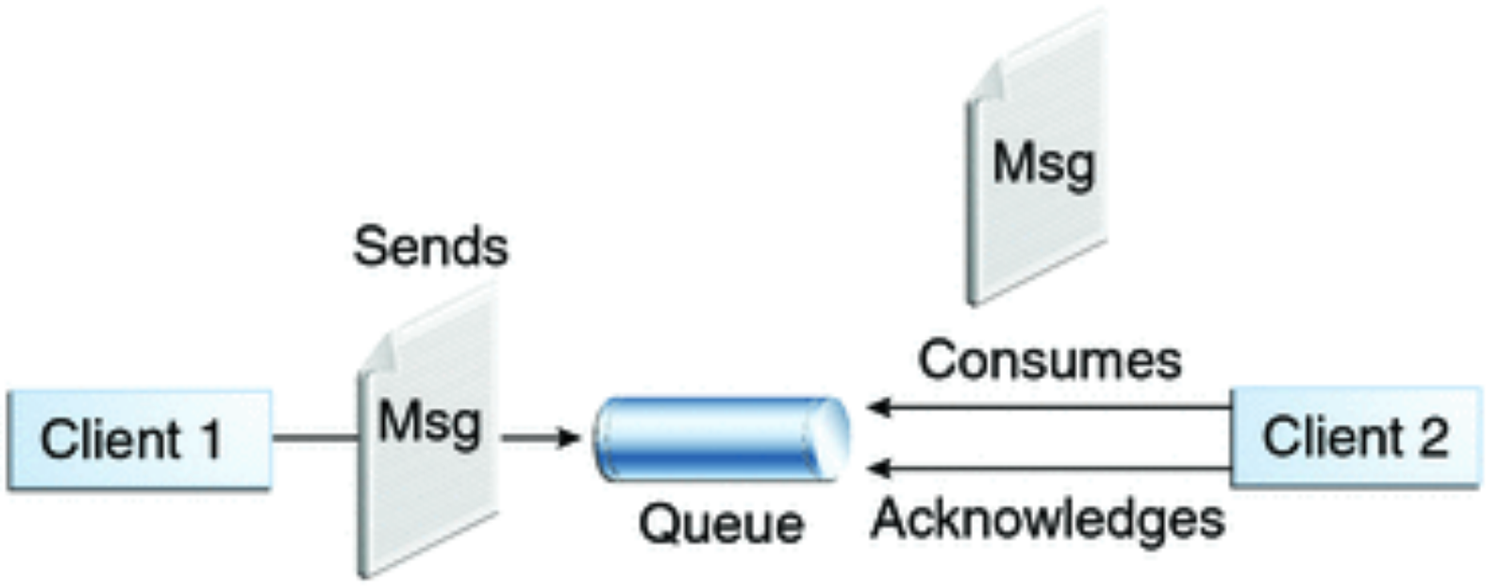


Point à Point : Principe

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.

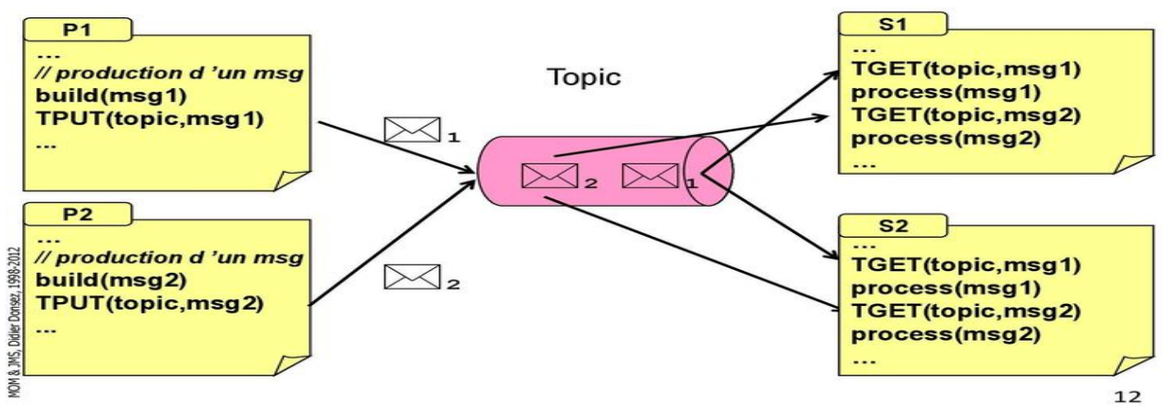
Ex: notification de maintenance

Modele : Publish-Subscribe

18/05/2014

17/05/2014

Modèle Publication-Souscription



MOM & JMS, Didier Donsez, 1998-2012

12

6

Connaissez vous ?

---

https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html

JMS : Java Message Sending

JMS et JEE

JMS API dans la plateforme Java EE :

Les clients peuvent recevoir des messages JMS en asynchrones.

Les Message-Driven Beans, (MDB) sont des composants capables de consommer des messages asynchrones.
Un JMS provider peut implémenter la gestion de messages des MDB.

L’envoi et la réception de messages peut être utilisée pour gérer les transactions distribuées…

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

https://www.jmdoudoux.fr/java/dej/chap-ejb3.htm#ejb3-11

Spécifité d'un MDB

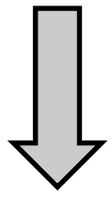
Un message-driven bean
1. n'implémente pas de business interface
2. est sans état.
3. les transactions peuvent être managée par le bean ou par le container.

Une classe **message-driven bean** :
- Doit être annotée avec l'annotation **@MessageDriven**
- Doit être *public*.
- Ne peut pas être définie ni *abstract* ni *final*.
- Doit contenir un constructeur par défaut
- Ne doit pas définir une méthode *finalize*.
- Doit implémenter l'interface MessageListener

(javax.jms.MessageListener interface dans le cas d'une implementation de L'API JMS)

Fonctionnement d'un MDB

`@MessageDriven`



Chaque composant 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)

Cycle de vie : les étapes

Crée des instances de MDB (EJB stateless)

1. Injecte des ressources, incluant le « message-driven context »
2. Place les instances dans un managed pool
3. Sort un bean inactif du pool lorsqu'un message arrive

appel de la callback : PostConstruct

Cycle de vie : les étapes suite

4. Exécutes le message listener method : la méthode *onMessage*

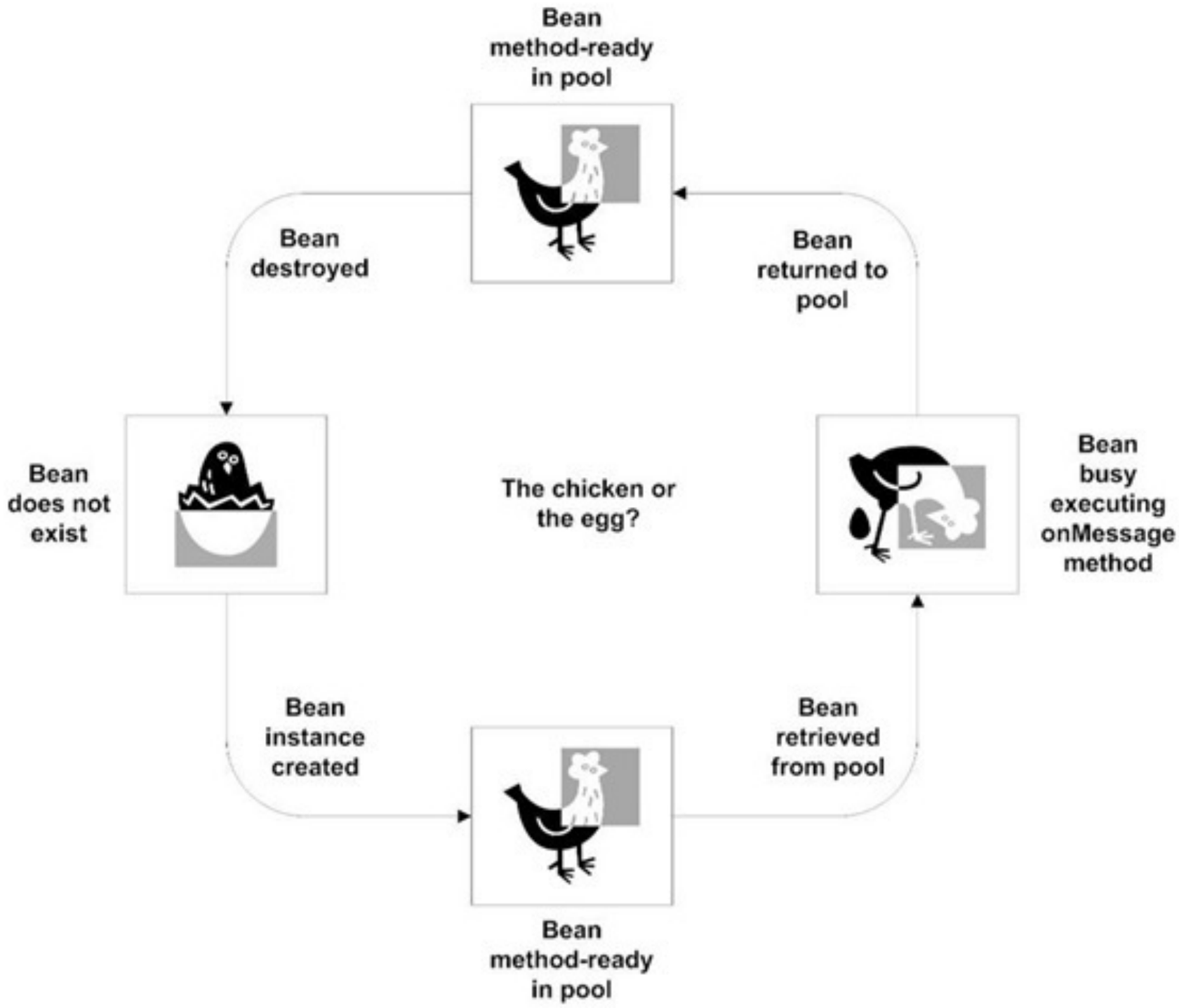
5. Lorsque la méthode onMessage termine son exécution, le bean inactif est replacé dans le pool « method-ready »

6. Au besoin, retire (ou détruit) les beans du pool

    5’ appel de la callback : PreDestroy

Message-driven bean lifecycle

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



Le rôle de la méthode onMessage

Implémenter l'interface MessageListener

Et donc le code de la méthode

```
public void onMessage(Message message)
```

onMessage method appelée par le contener quand un message est adressé au bean

Cette méthode gère la logique métier en fonction du message reçu

Le MDB parse le message et exécute la fonctionnalité métier

Messages

Un Message peut être :

un un message texte à parser de type TextMessage,

ou l'instance d'une classe de type ObjectMessage ….

Un ObjectMessage doit contenir un objet Serializable.

Le client d’un MDB

L’annotation @Resource

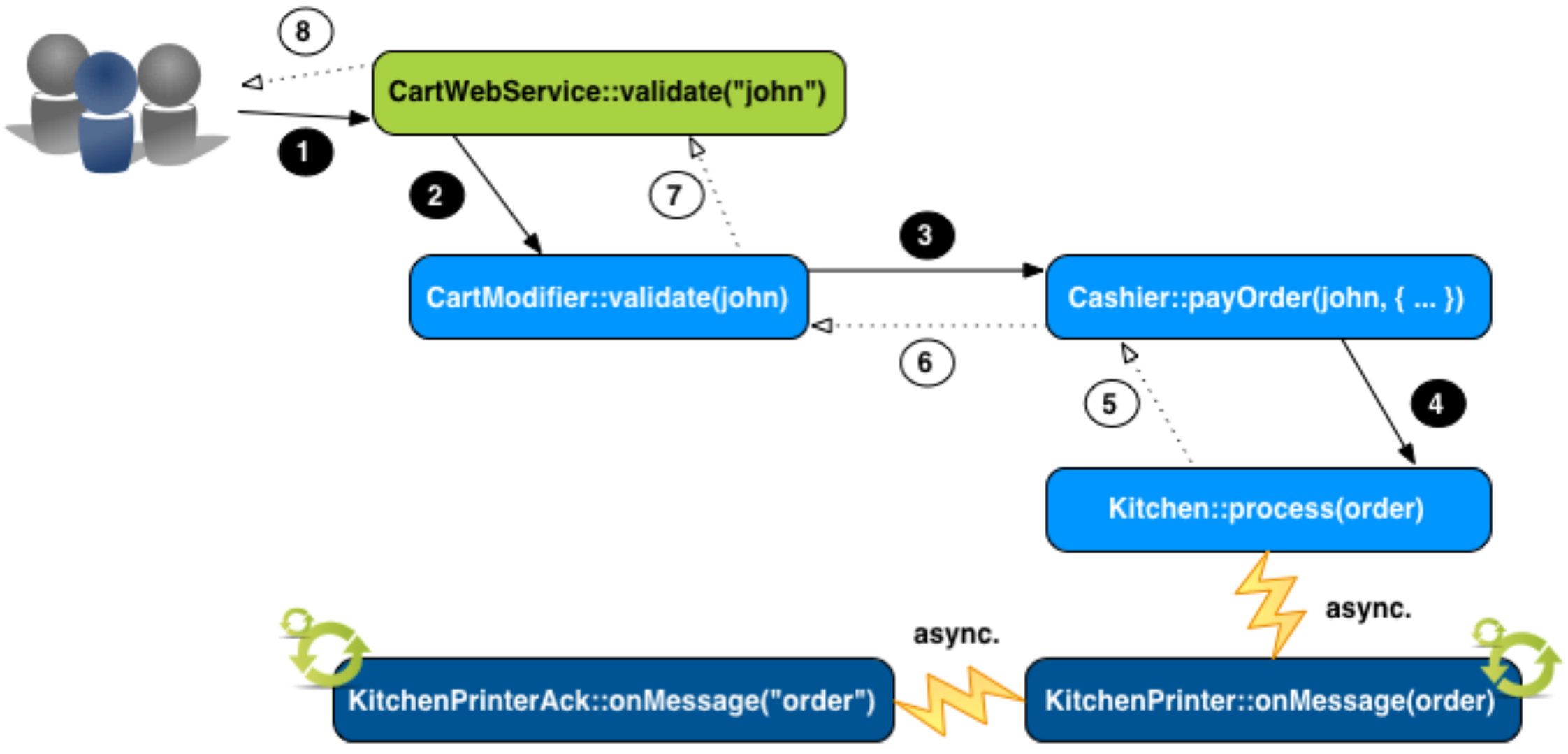
Le client doit envoyer les messages à la Queue que le Bean écoute

Le client injecte la factory de connexion et la ressource correspondant à la queue

avec l’annotation @Resource

TCF : envoyer une commande à une imprimante

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.



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

```
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 {
    ....

}
```

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();
   }
}
```

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/