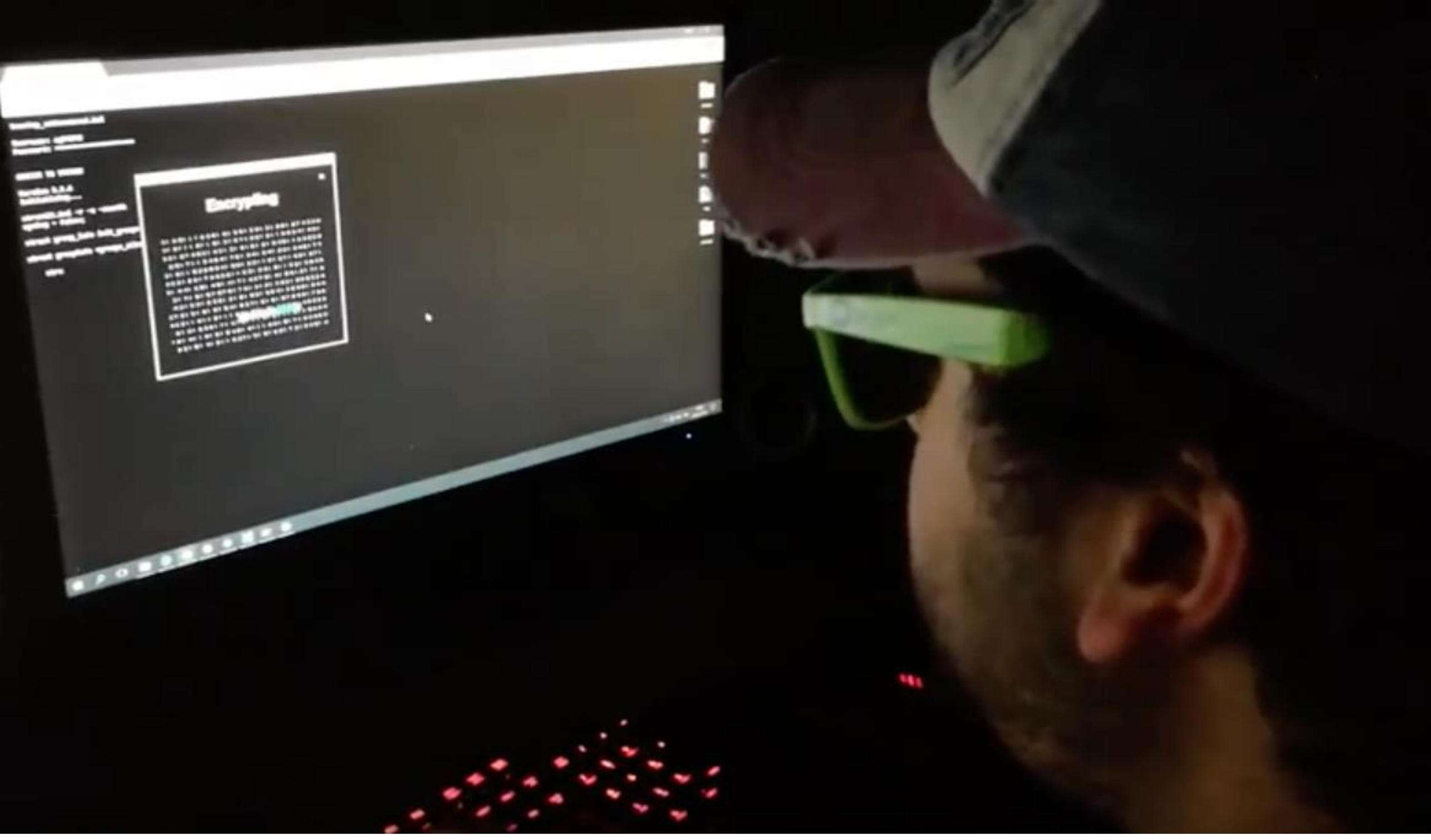
# J2E++: Interceptors, MOMs

Sébastien Mosser et Anne Marie Dery

pictures: sxc.hu




# Intercepting Messages

Man in the middle
(without the hoody)

# Histoire : A.O.P.

Aspect Oriented Programming
(Xerox Parc, Gregor Kickzales)

Proposent AspectJ comme extension de Java.
IBM en 2001  propose HyperJ.

# A.O.P. Pourquoi ?

Augmenter la modularité
Séparation des préoccupations orthogonales au code

Ne pas polluer la logique  métier par des comportements relatifs à des préoccupations telles que les traces, la sécurité….

# A.O.P. Comment ?

En ajoutant des comportements à un code existant ***advice*** sans modifier le code,
En spécifiant séparément quel code est modifié
Par un la spécification de ***pointcut***.

Par exemple une préoccupation « log » peut être « ajoutée » au code pour chaque appel d'accesseur en écriture (set*)

# Terminologie

*Cross-cutting concerns* : « fonctionnalités secondaires » partagées par plusieurs classes

*Advice* : code additionnel à appliquer  au code métier existant.

*Pointcut :* le point d’exécution où le cross-cutting concern doit être appliqué.

*Aspect :* les advices et les pointcuts.

*Weaver* : tissage du des advices dans le code

# Terminologie : exemple

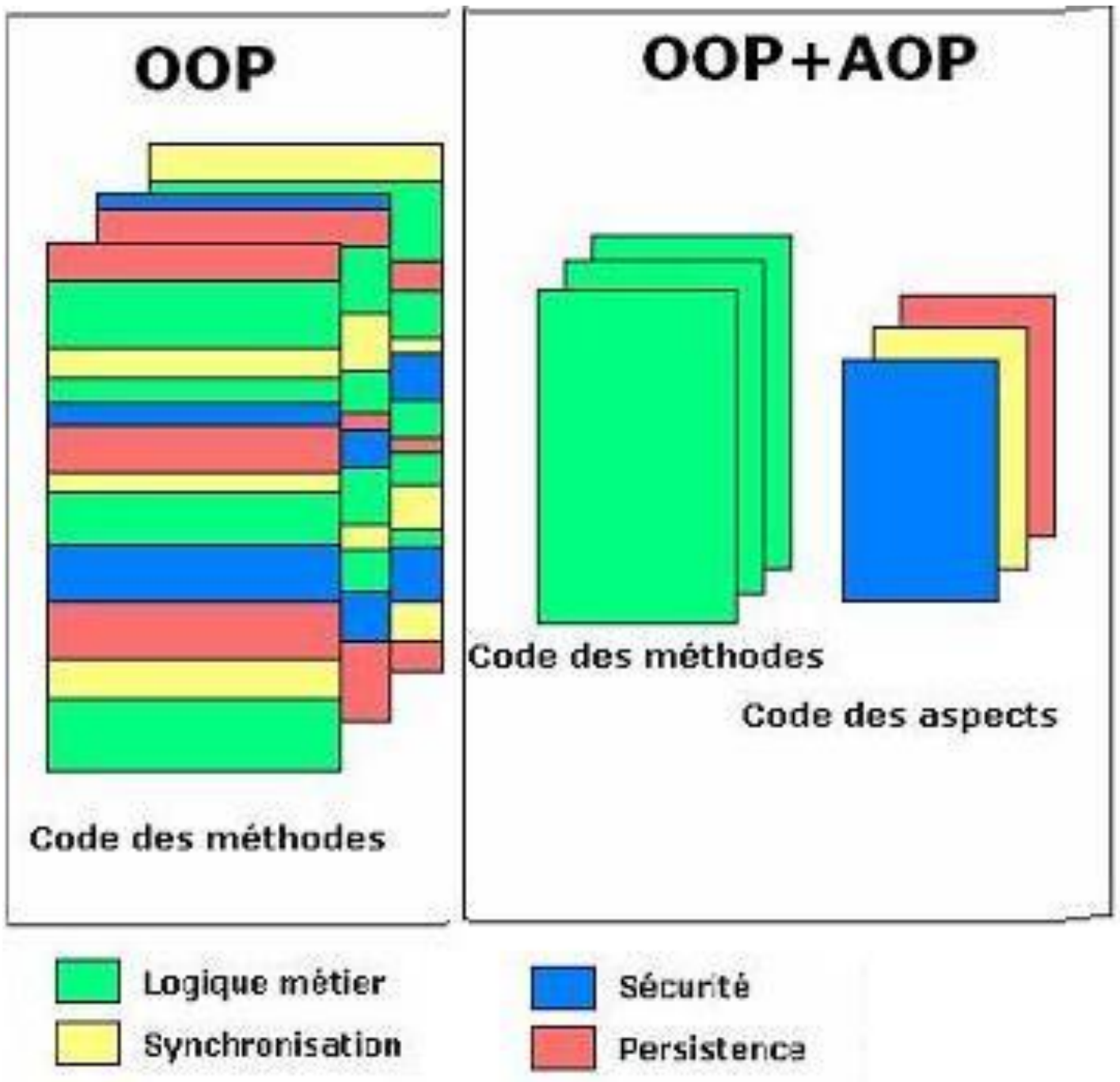
Cross-cutting concerns : log / trace

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

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

Aspect : Aspect Log

# Principes


OOP
OOP+AOP
Code des méthodes
Code des méthodes
Code des aspects
Logique métier
Synchronisation
Sécurité
Persistence

# AOP : Intercepteurs et EJB

Les intercepteurs et serveur d'applications
$\Rightarrow$ Gestion des messages échangés entre composants

Intercepteurs orientés métiers
$\Rightarrow$ support pour développer des préoccupations orthogonales (cross-cutting features) et diminuer la duplication de code.

# Annotations, mise en œuvre

**@AroundInvoke**
Associé 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

Pour intercepter placer
**@Interceptors** au bon niveau d'interception (opération, service…)
Ou définir dans un fichier l'expression régulière à appliquer pour retrouver les beans à intercepter

# Cas TCF

Pour des besoins statistiques :
1 compter le nombre de cartes.
2. Seulement si effectif

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

# Statistiques

```
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)); }
```

# Détection de faute

Gérer les pré-conditions d'une opération

Ex:

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

# Fault detection

```
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;
```

# Exemple du Logger

*Tracer* toutes les opérations invoquées dans le système.

```
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);
		}
	}

}
```

# Déclaration pour Logger

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
⇒ pour chaque qui matche

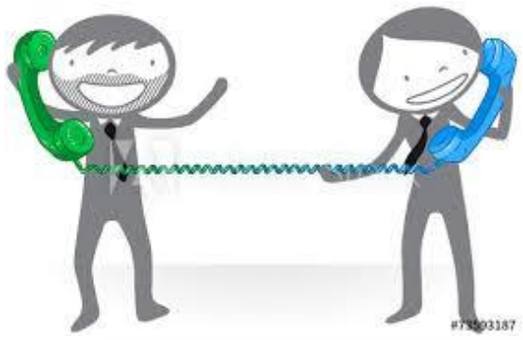
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise JavaBeans2.0//EN' 'http://java.sun.com/dtd/ejb-jar_2_0.dtd
<ejb-jar>
    <assembly-descriptor>
        <interceptor-binding>
            <ejb-name>*</ejb-name>
            <interceptor-class>fr.unice.polytech.isa.tcf.interceptors.Logger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

# 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 ?

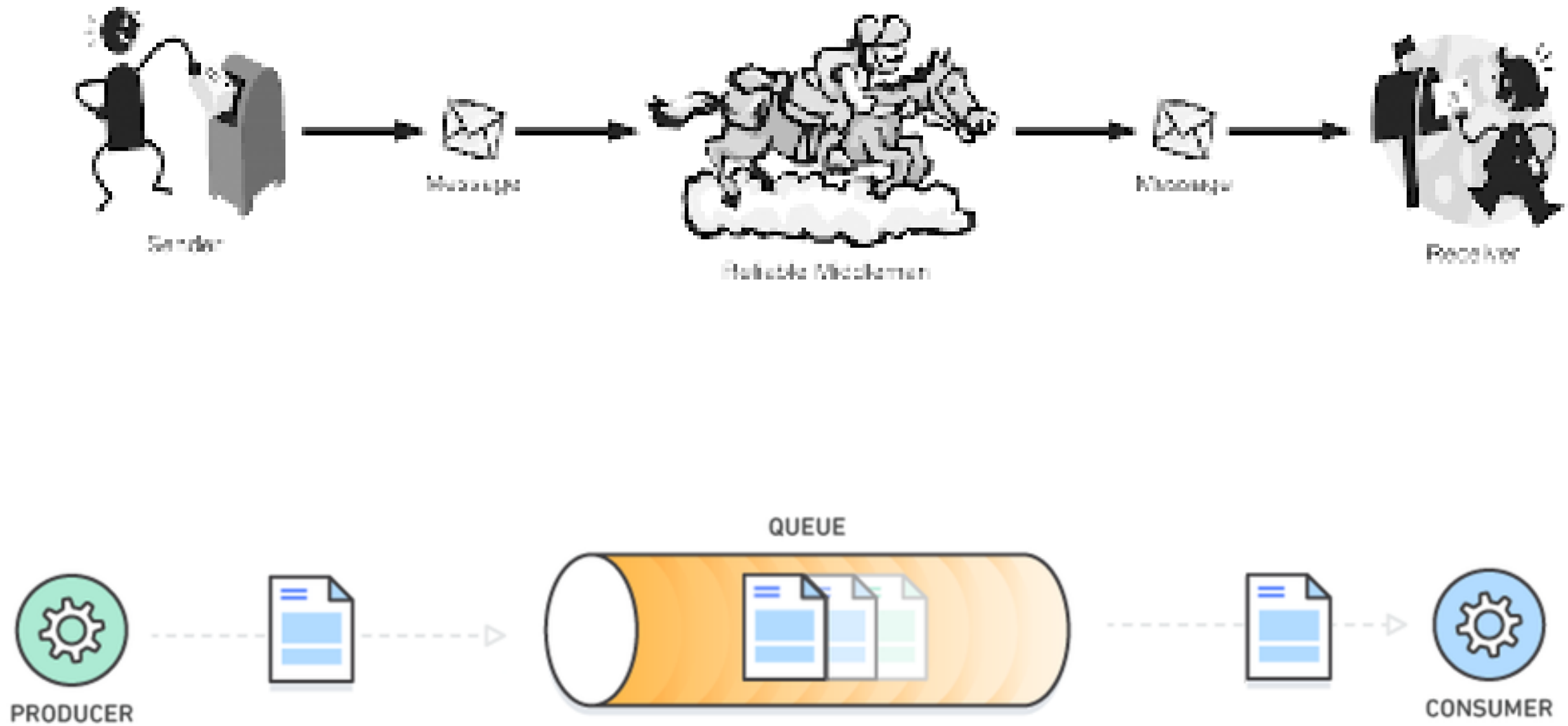
Communication asynchrone entre composants.
L’appelant et l’appelé n’ont pas besoin d’être présents pour que la communication réussisse

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

# The **Messaging** paradigm




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

# MOM : Principe

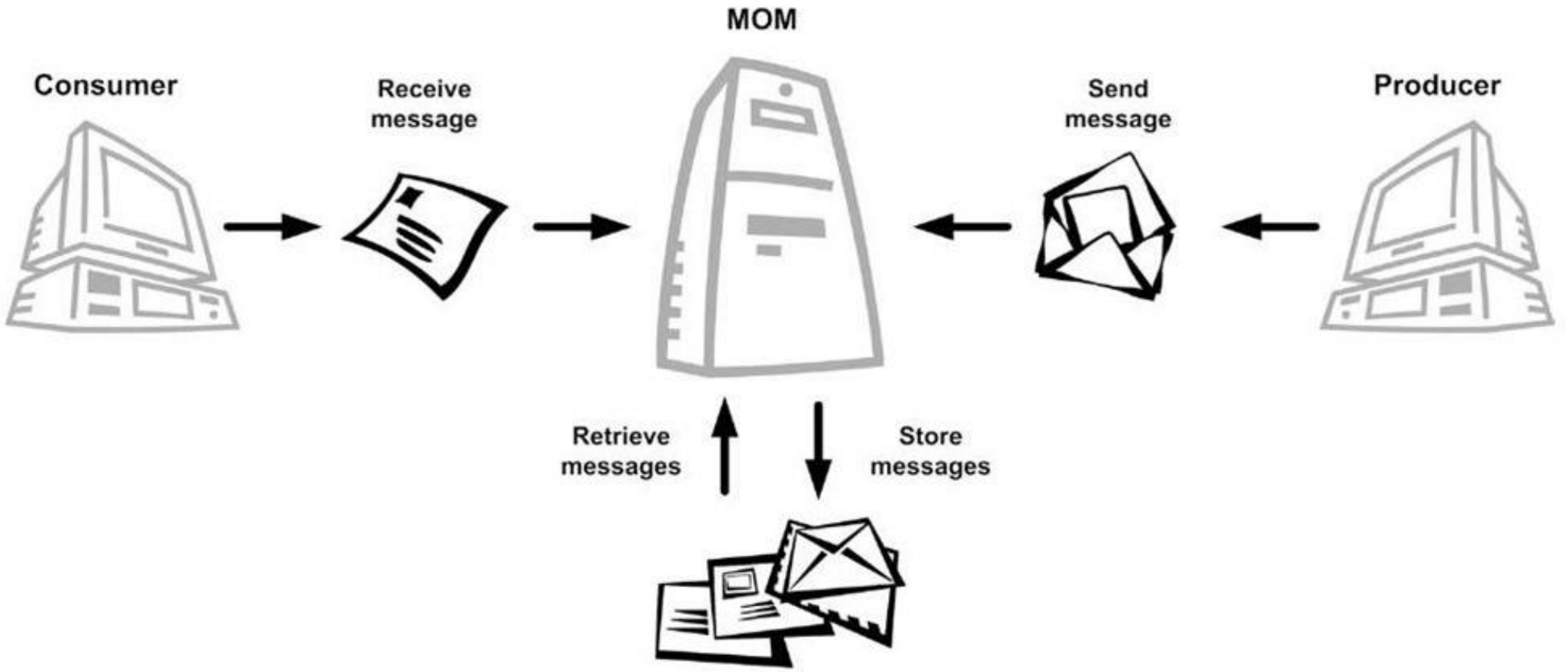
Envoi du message  =>

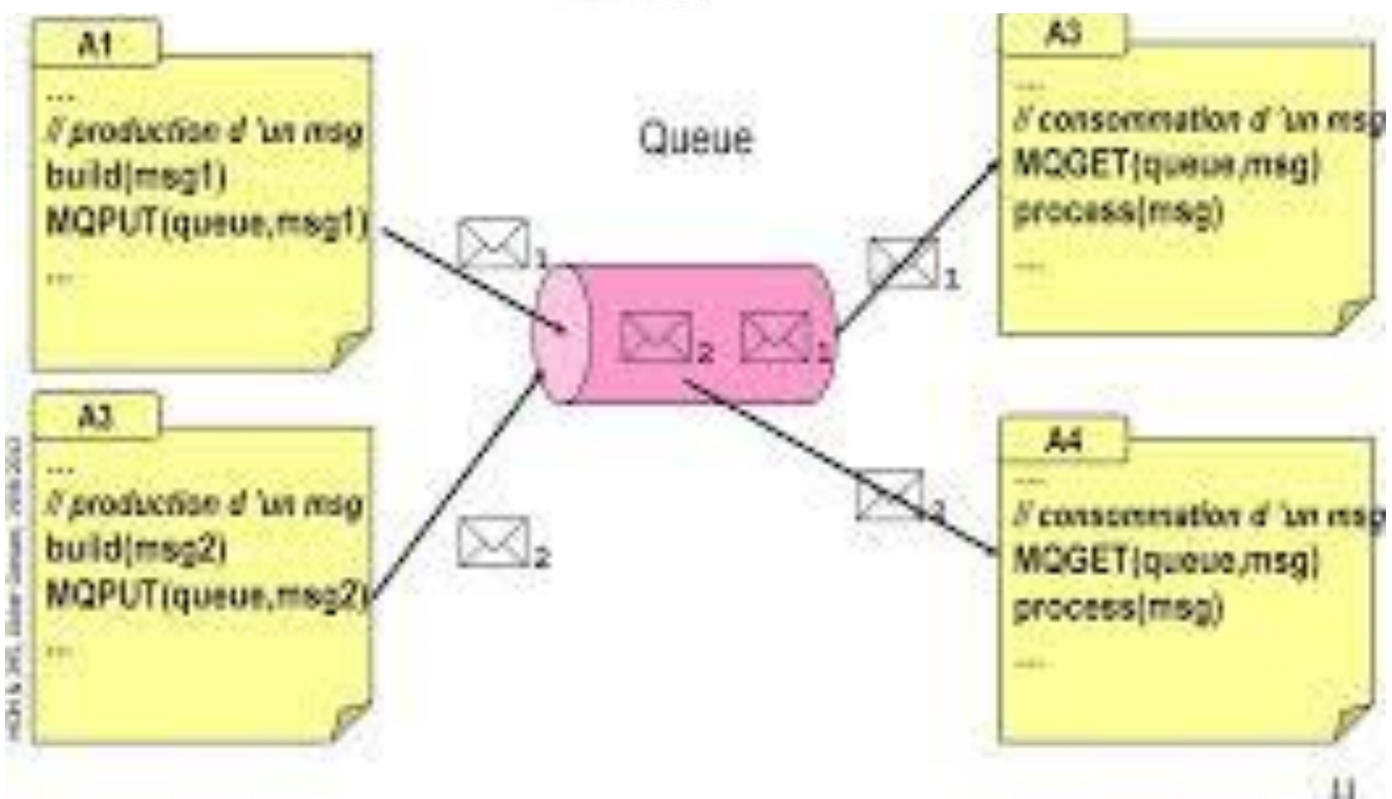
1. stockage du message dans un lieu (Destination) spécifié par l'expéditeur (Producer)

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



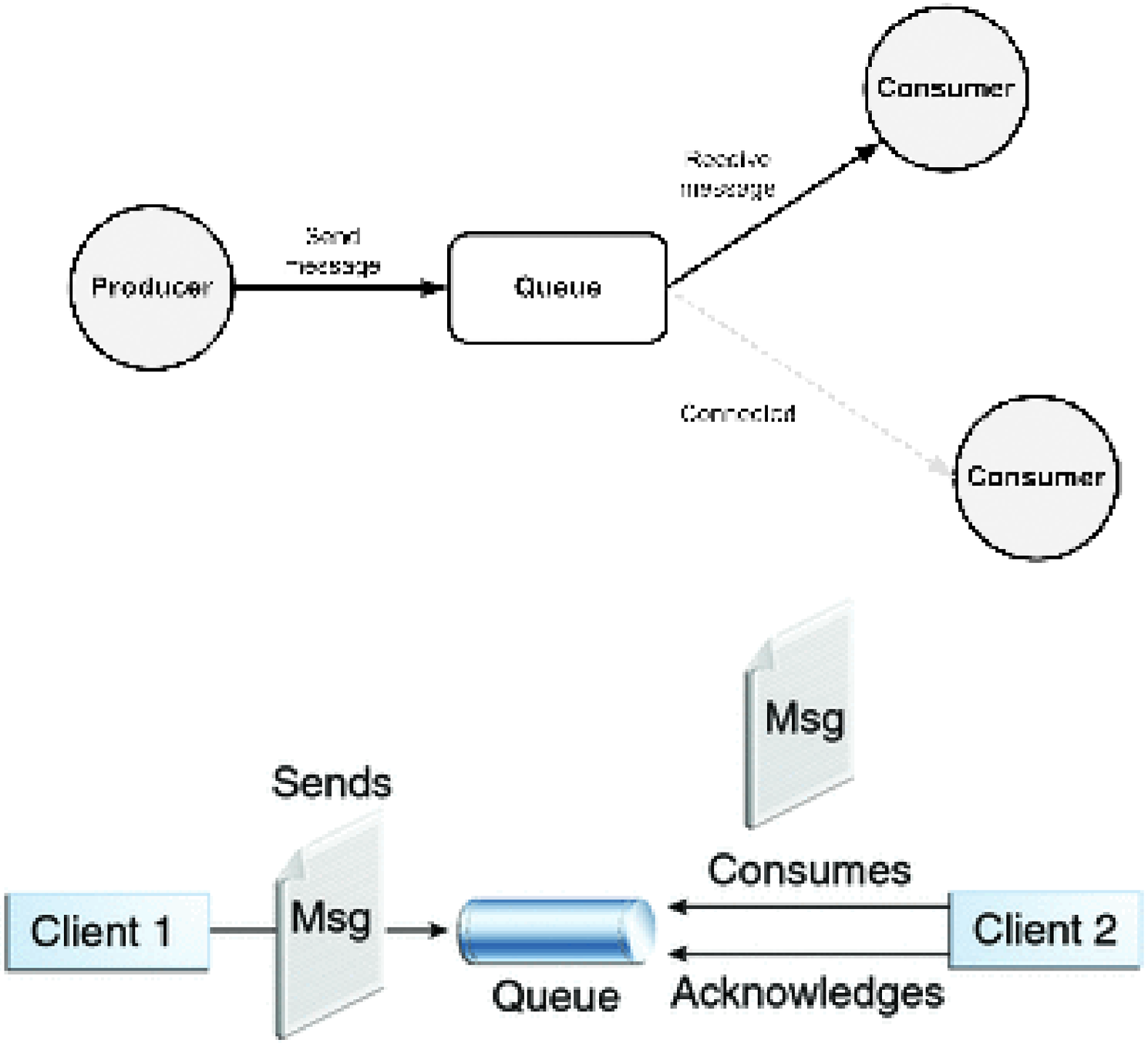



# Point à Point : Principe

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



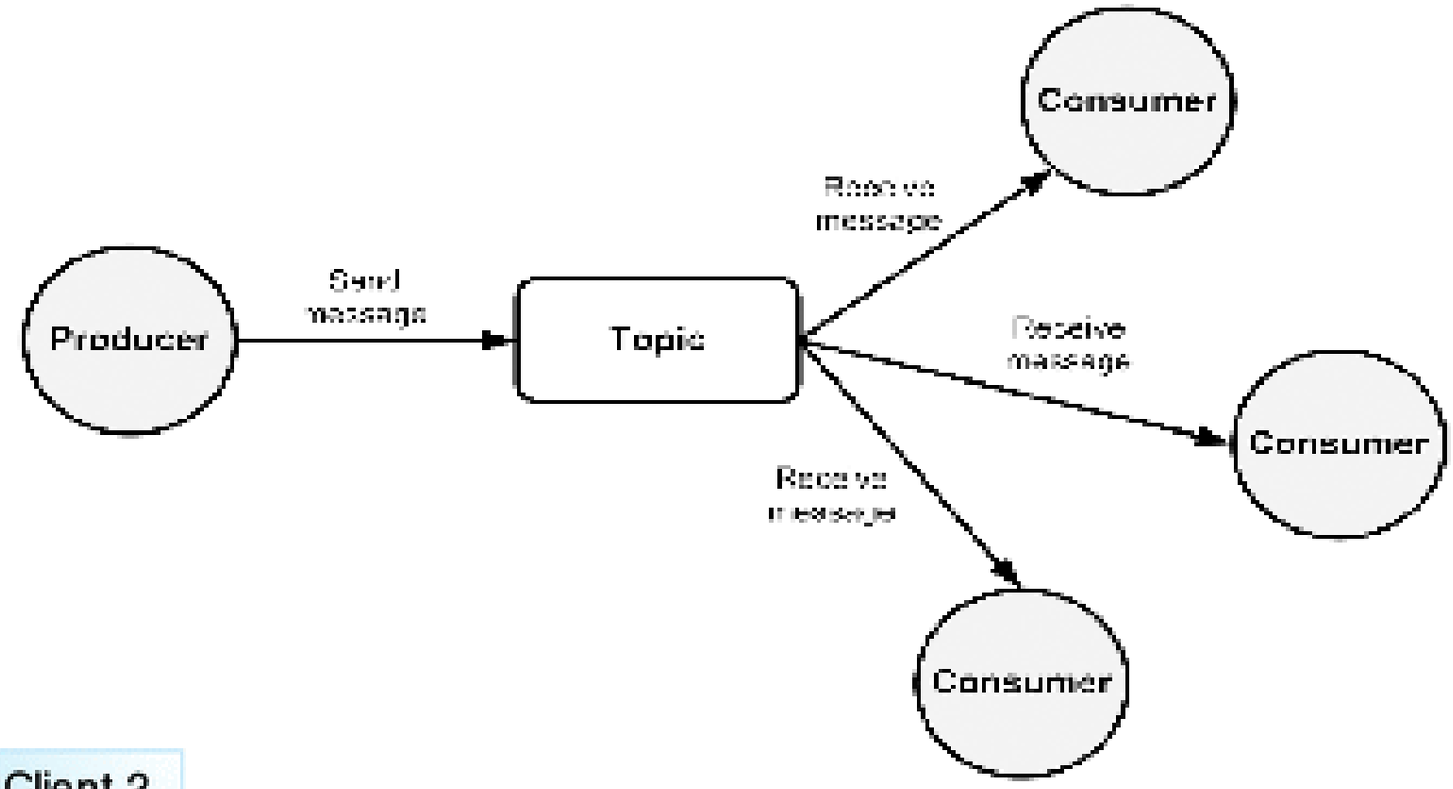
# Publish-Subscribe : principe

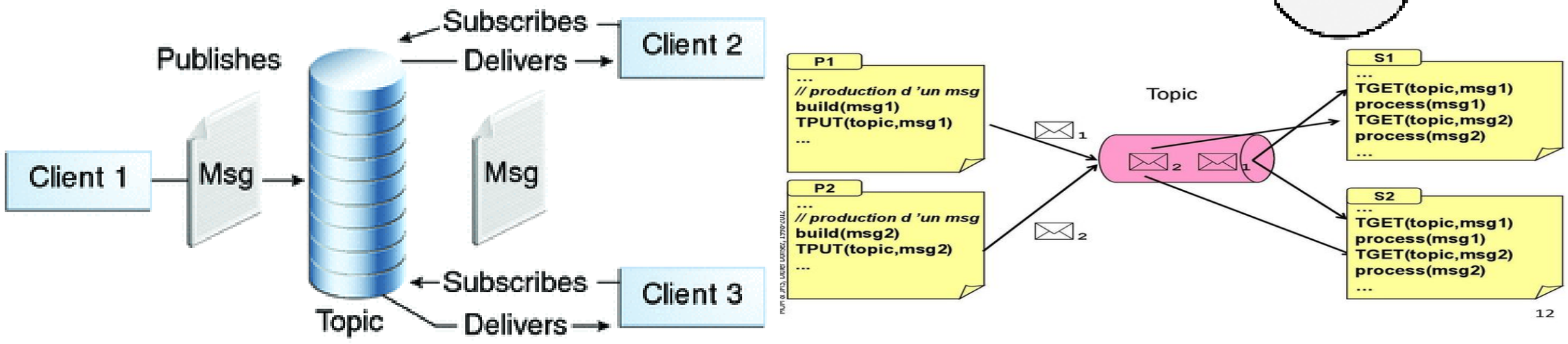
⇔ newsgroup Internet .
Un Producer produit  un message reçu par un nombre non déterminé de Consumers
La destinations du message est un Topic
Le consommateur est un subscriber.

Utile pour le broadcast d’information.
Ex: notification de maintenance

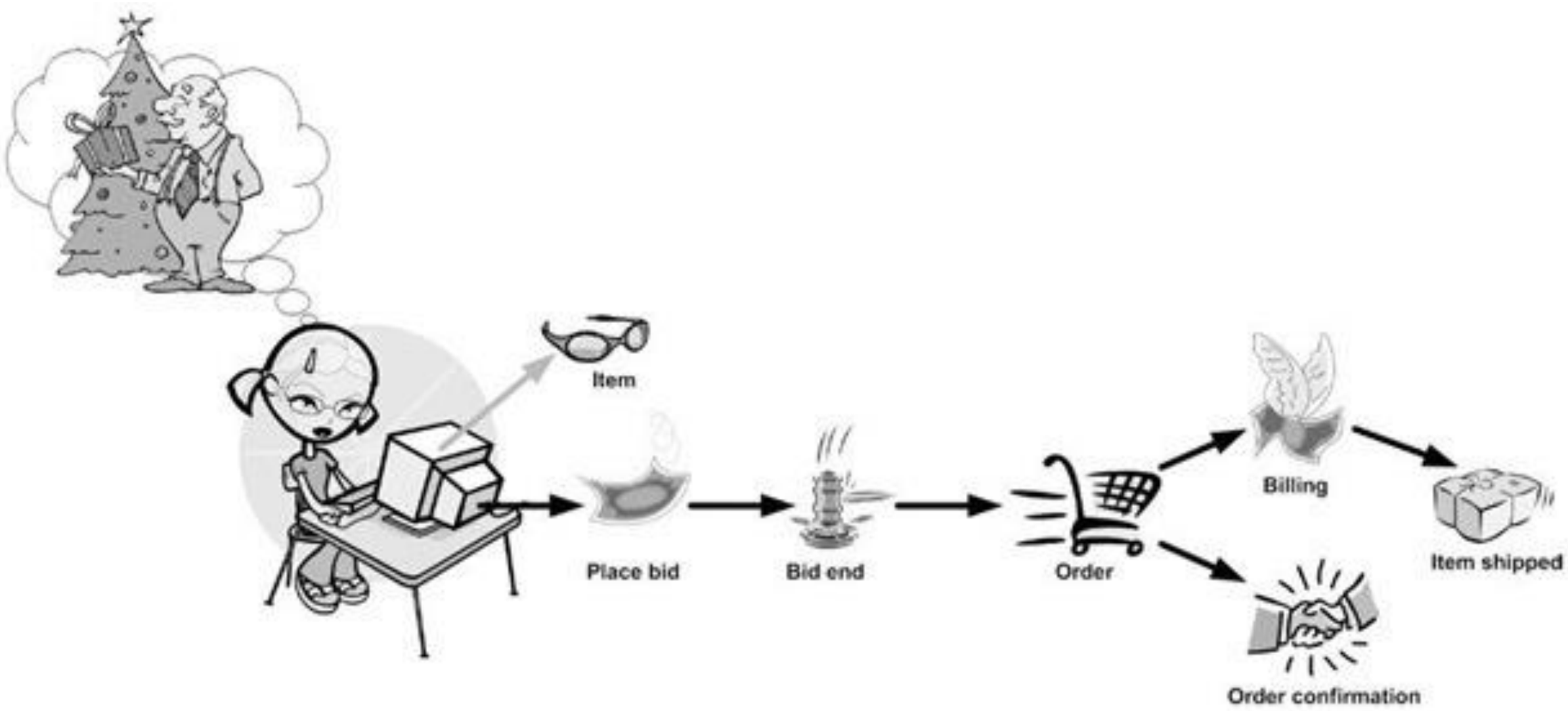
# Modele : Publish-Subscribe





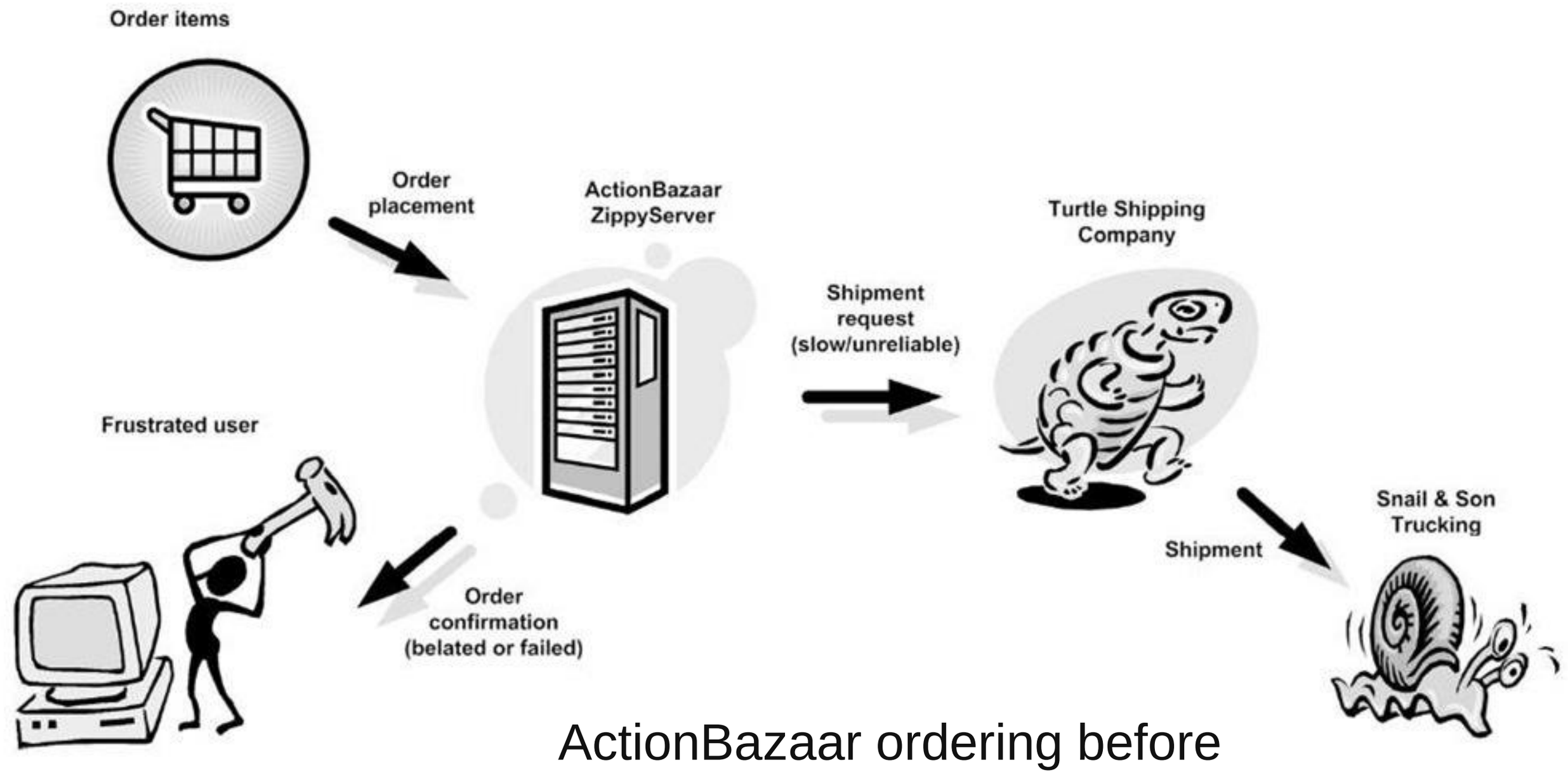



# The ActionBazaar example



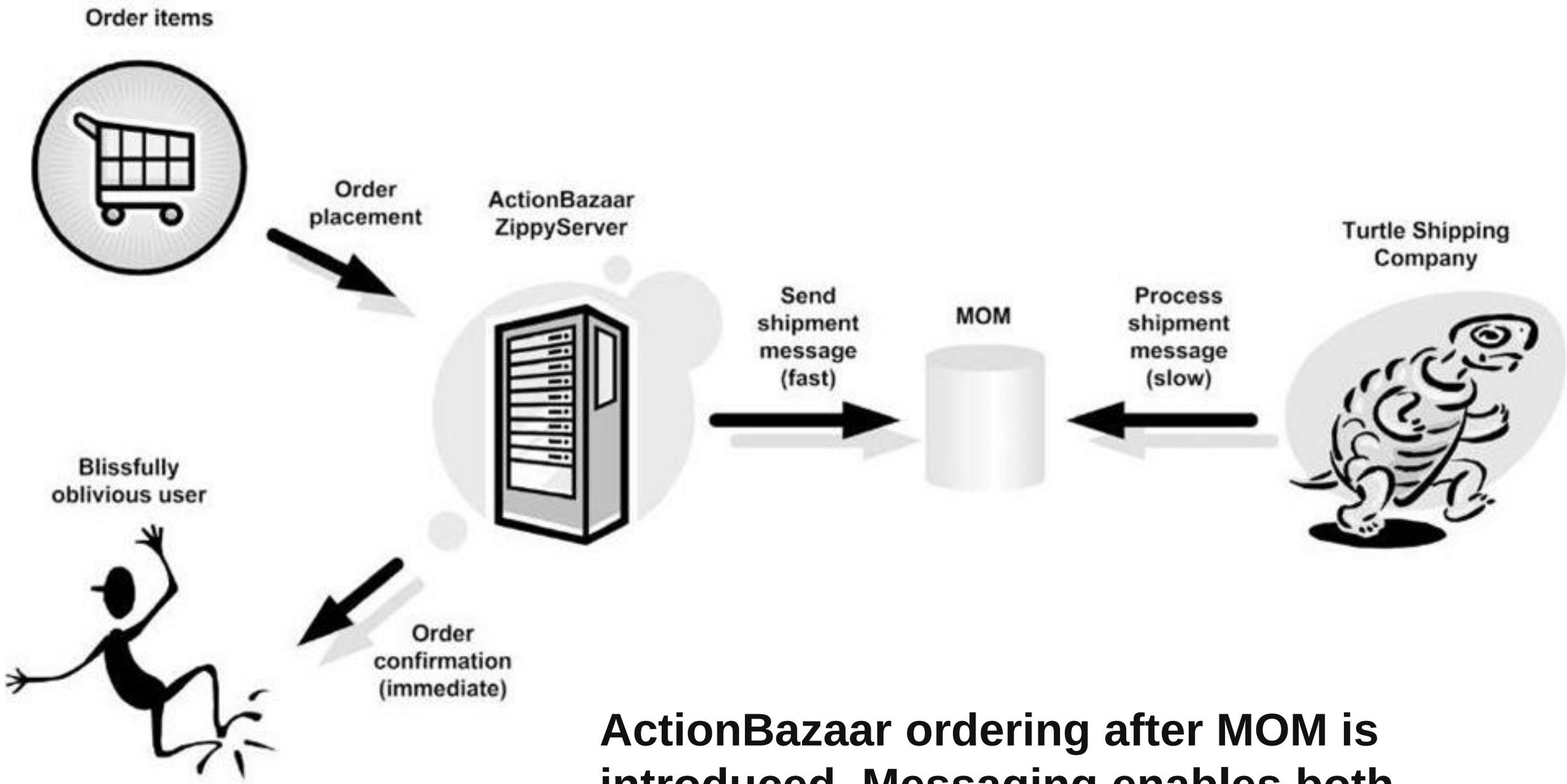

http://what-when-how.com/enterprise-javabeans-3/messaging-concepts-ejb-3

# The ActionBazaar example




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

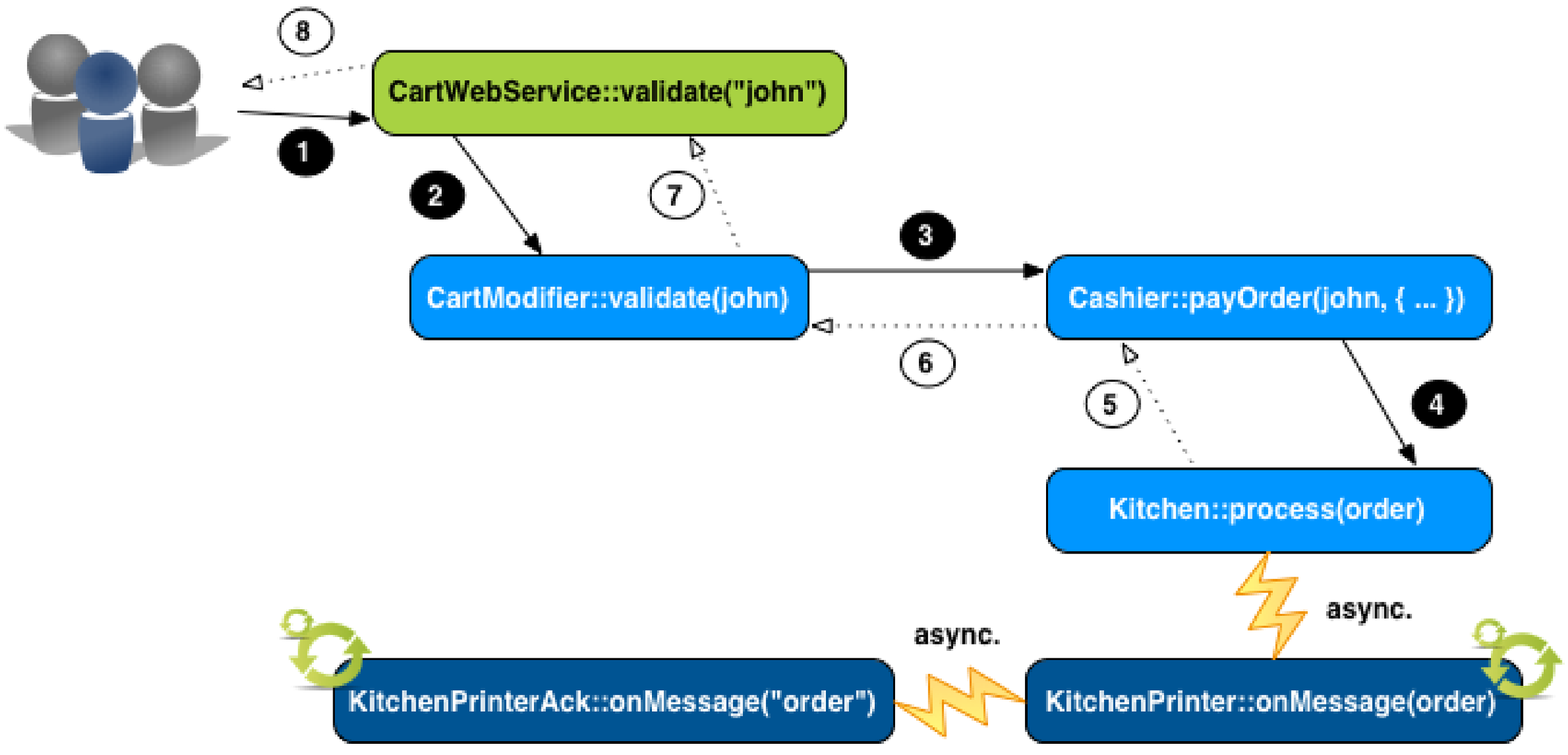
# Introducing messaging




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

# Example: The Cookie Factory

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



# Example: 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

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

Un ObjectMessage doit contenir un objet Serializable.

Ex: envoyer une instance de Order au lieu du texte à l’imprimante

# 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

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.
Tout ce qui est gestion de la queue est automatiquement géré par le container (*e.g.*, starting the queues, stopping it, reading messages, passivating elements)

# 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();
   }
}
```

# Annotations, mise en œuvre

```
@MessageDriven
```

**Interface** MessageListener {
    **public void** onMessage(Message message) {

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

// ...
```

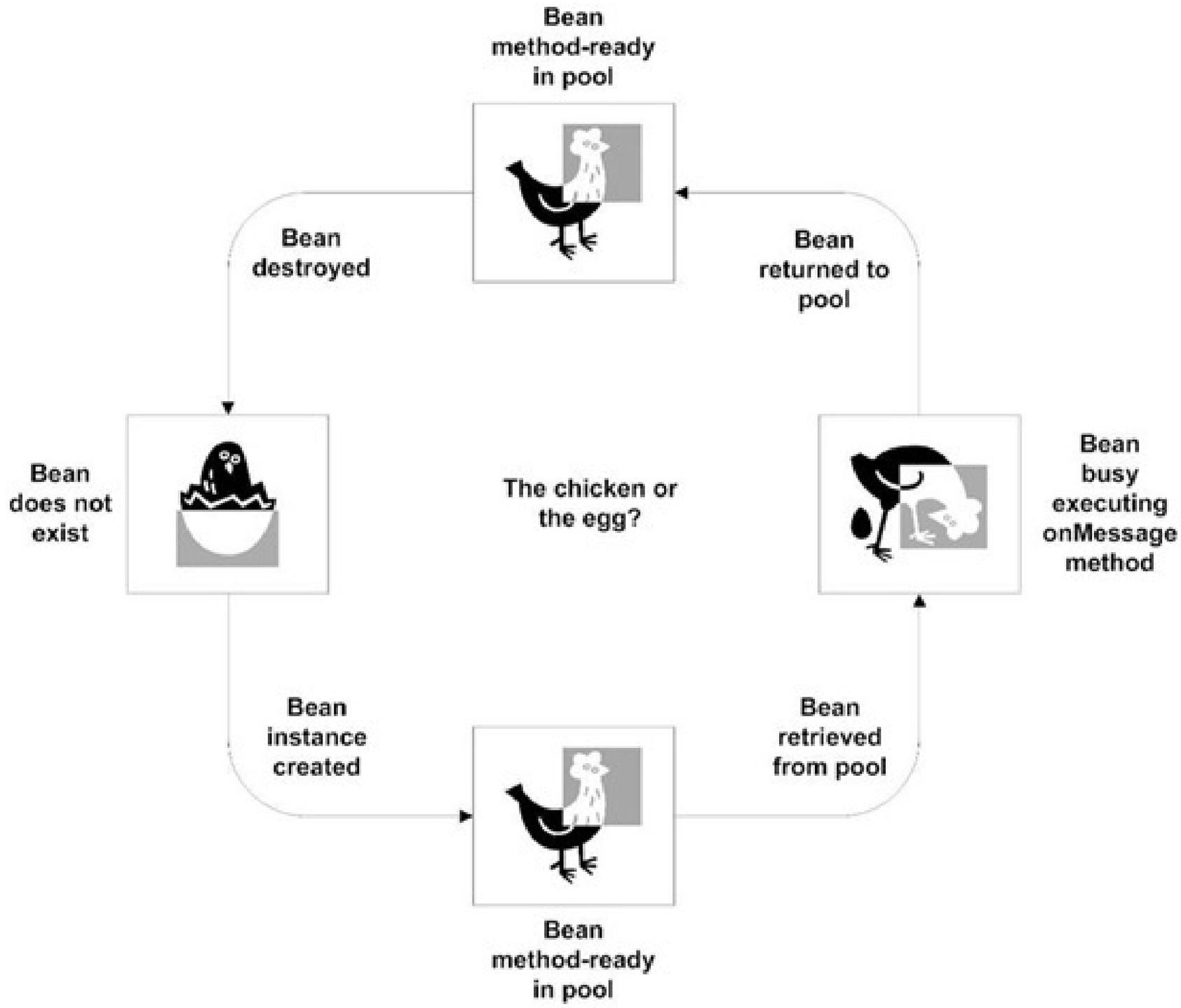
# Cycle de vie : les étapes

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 (the container may have to increase the pool size at this point)
- 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

# Message-driven bean lifecycle

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



# 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/