
SOA

(modélisation avec uml et bpmn)

Table des matières

I - Cadre de la modélisation SOA.....	4
1. Rappels fondamentaux (paradigme SOA).....	4
2. Quelques typologies de services à assembler.....	7
3. Cadre pour la modélisation (SOA) d'entreprise.....	8
Ce cadre (très classique) servira de point de repère pour les activités de la modélisation "SOA".....	8
4. Positionnement des niveaux de services.....	9
II - Modélisation des services élémentaires.....	10
1. Vision conceptuelle des services élémentaires.....	10
2. Cadre classique pour services élémentaires.....	13
.....	13
3. Modélisation des services élémentaires.....	14
4. Spécifications fines des services (détails).....	17

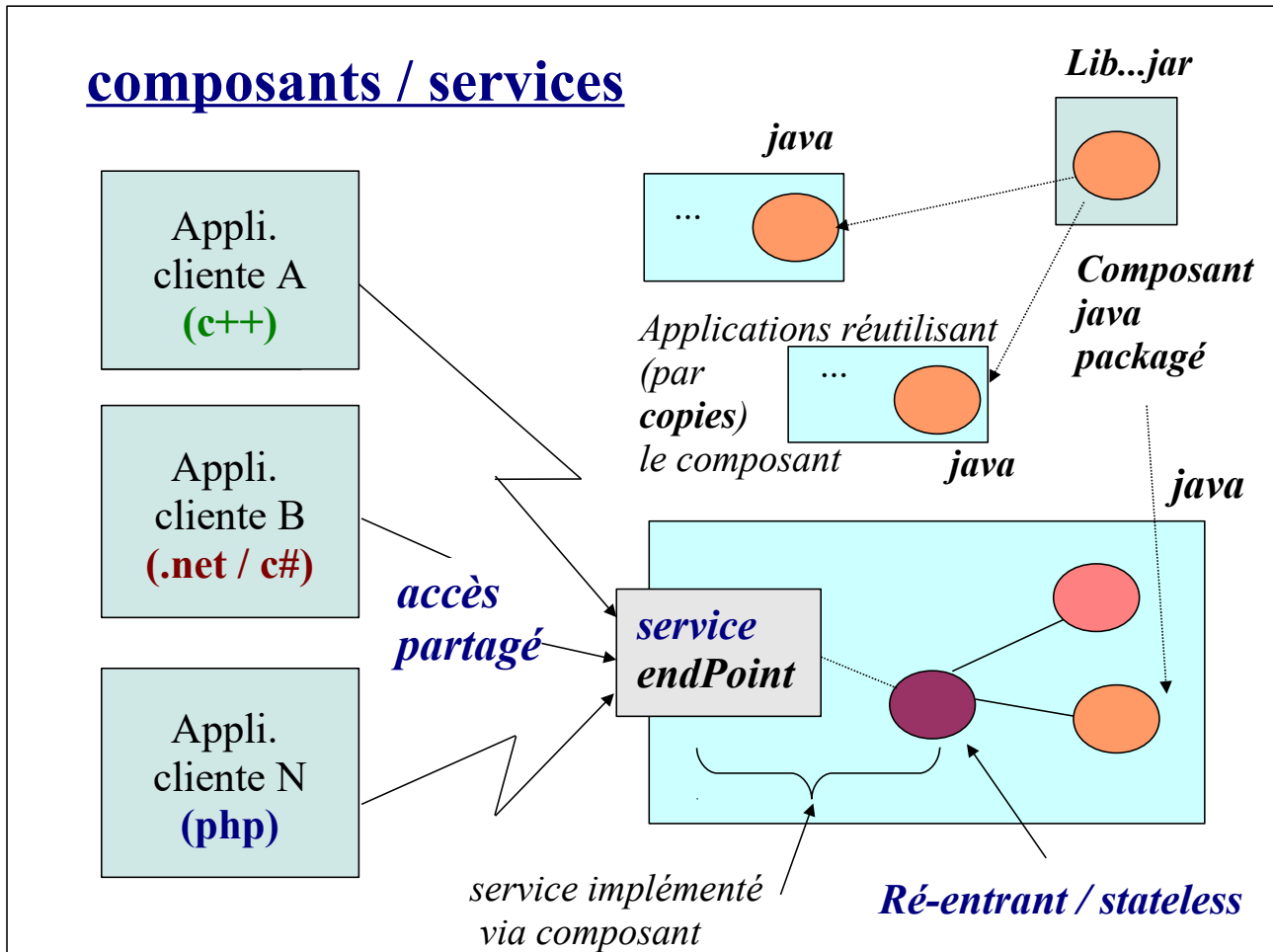
III - Transpositions vers contrats WSDL ou YAML.....	21
1. Transpositions "UML" en contrats WSDL.....	21
2. Contrats "YAML" pour Web-services "REST"	25
IV - Services fonctionnels.....	27
1. Services fonctionnels (définition de la catégorie).....	27
2. Données "pivots" (échangées).....	27
3. Modélisation des services fonctionnels.....	28
4. Intégration de l'existant (adaptateurs).....	28
V - Modélisation métier (processus , bpmn, ...).....	30
1. Structure cible d'une modélisation SOA.....	30
2. Diagrammes sémantiques (pré-étude facultative).....	30
3. Business Modeling et "Business Uses Cases"	32
4. Cas d'utilisation d'une partie du SI (soa).....	33
5. Modélisation des processus métiers.....	35
6. BPMN (essentiel).....	35
VI - Problématiques SOA (intégrité , MDM, ...).....	49
1. Modélisation et problématiques SOA.....	49
2. Gouvernance des données.....	50
3. Transactions longues et compensations.....	55
4. Intégrité référentielle & MDM.....	56
VII - Repères d'urbanisation.....	58
1. Notions essentielles d'urbanisation.....	58
2. Urbanisation fonctionnelle.....	60
3. Urbanisation applicative.....	62
4. Prise en compte de l'urbanisation dans la modélisation SOA.....	62
5. Modéliser les cycles de vie des objets de référence.....	63
VIII - Architecture logicielle (SOA).....	67
1. Dérivation du fonctionnel en "logiciels SOA"	67
2. Dérivation du modèle logique sous la forme de composants SOA.....	68
3. Prise en compte des aspects techniques.....	70
IX - Structuration des données Pivots.....	72
1. Données "pivots" (échangées).....	72

2. Recherche d'abstractions, spécialisations/variantes.....	74
.....	75
3. Hiérarchie de domaines / références communes.....	76
X - Packaging, déploiement composants SOA.....	77
1. Modélisation de "composants SOA"	77
2. Packaging composants "soa" (itf , impl , delegate).....	79
3. Gestion des versions (évolutions à prévoir).....	81
4. documentation.....	82
5. pilotage , suivi et éventuelles corrections/anticipations	83
XI - Processus long, orchestration asynchrone.....	84
1. Dualité "services avec orchestration de sous-services" et "processus"	84
2. "processus métier vers processus exécutable"	85
XII - SOA réaliste selon moyens.....	87
1. SOA bien dimensionné (selon moyens et besoins).....	87
2. Cycle de vie d'un projet SOA.....	89
3. Activités de modélisation SOA (enchaînement).....	92
XIII - Annexe – Eléments clefs – UML.....	94
1. Vue d'ensemble sur UML.....	94
2. Diagramme des cas d'utilisations.....	95
3. Scénarios et descriptions détaillés (U.C.).....	99
4. Diagramme d'activités.....	101
5. Distinction "micro activité / macro activité" selon granularité.....	107
6. Diagramme de classes (notations , ..).....	108
7. Diagramme d'états et de transitions (StateChart).....	119
8. Diagramme de séquences (UML).....	122
XIV - Annexe – Praxème.....	125
1. Méthode Praxème.....	125
XV - Annexe – Etude de cas , TP.....	128
1. Présentation de l'étude de cas.....	128
2. TP (modélisation SOA).....	128

I - Cadre de la modélisation SOA

1. Rappels fondamentaux (paradigme SOA)

1.1. Les services sont potentiellement partagés



Rappel fondamental :

Une fois installé/publié (en production) , un service est souvent partagé :

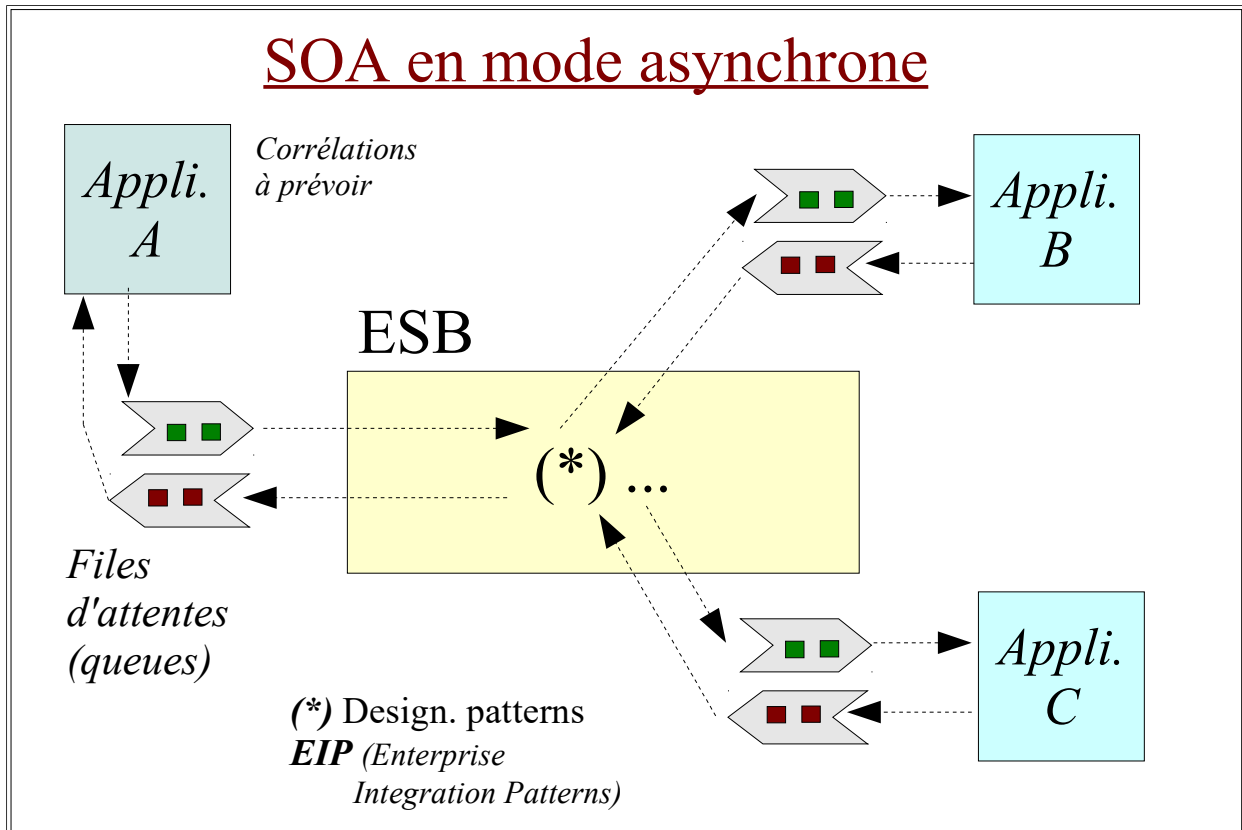
- plusieurs applications "clientes" peuvent effectuer des appels distants.
- les appels distants sont généralement "ré-entrant" et "stateless" .

→ ceci permet de partager :

- des données de références
- des traitements communs
- des (sous-)processus métiers
-

Cependant, il faut commencer par modéliser ce qui doit être partagé (dénominateur commun entre différents besoins , ...).

1.2. Penser à SOA en mode asynchrone



Bien qu'un service élémentaire soit souvent "synchrone" (appels de méthodes distantes , ...), **certains aspects avancés de l'architecture SOA se situent dans un contexte asynchrone :**

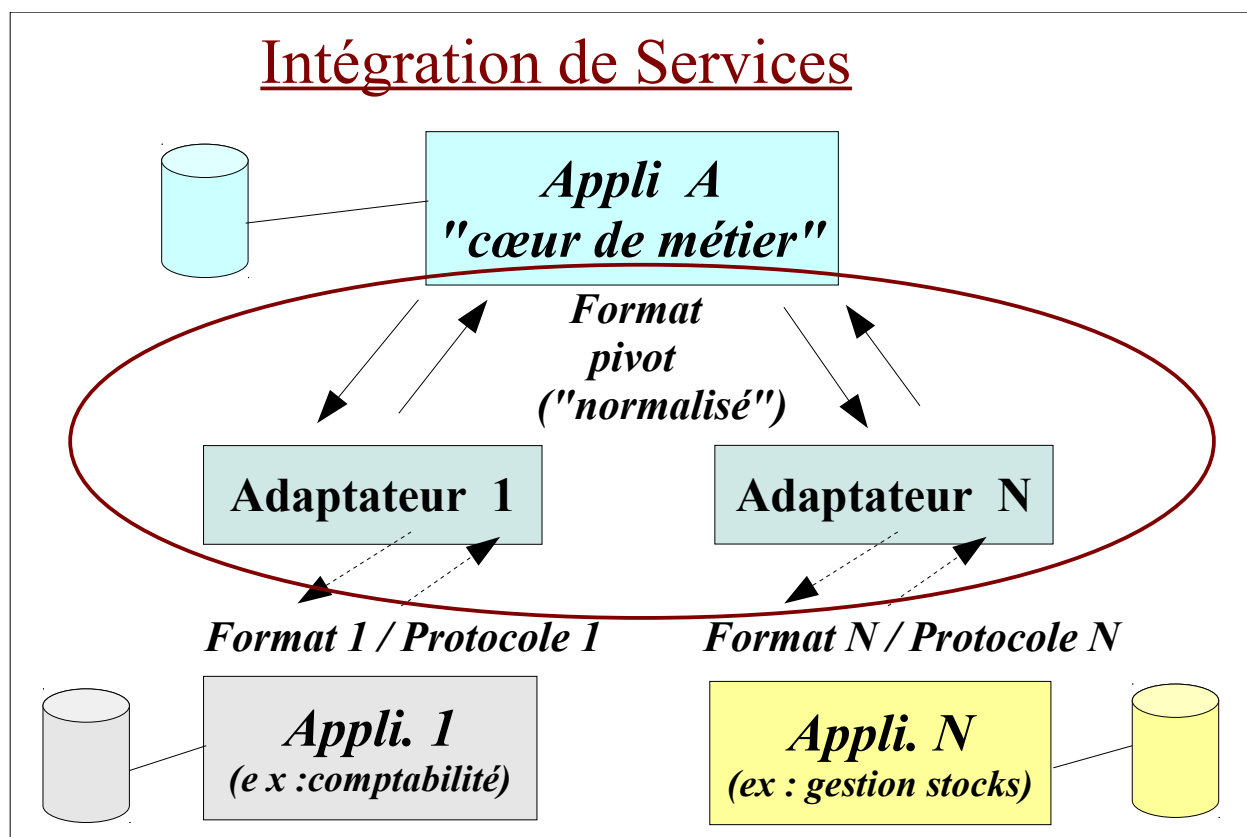
- échanges de messages (requêtes , réponses, documents , notifications , ...)
- processus métiers longs
- ...

1.3. Rappel important: SOA et Intégration

L'architecture SOA est souvent utilisée pour **intégrer** dans un **SI spécifique au cœur de métier** des **applications existantes (non développées en interne mais achetées)** qui **rendent des services transverses ou génériques** (ex: Compta , gestion de Stock, ...).

Pour faire communiquer entre elles des applications provenant de différents éditeurs , on a besoin de tout un tas d'éléments **intermédiaires** fournis par l'**infrastructure SOA** (*ESB* , *Adaptateurs*, ...).

Dans le cas d'une application "clef en main" achetée auprès d'un éditeur de logiciels, on ne maîtrise absolument pas le format exact des services offerts (liste des méthodes , structures des données en entrées et en sorties des appels). Via des paramétrages de transformations "ad-hoc" effectuées au niveau d'un ESB, il sera (en général) tout de même possible d'invoquer cette application depuis d'autres applications internes au SI de l'entreprise.



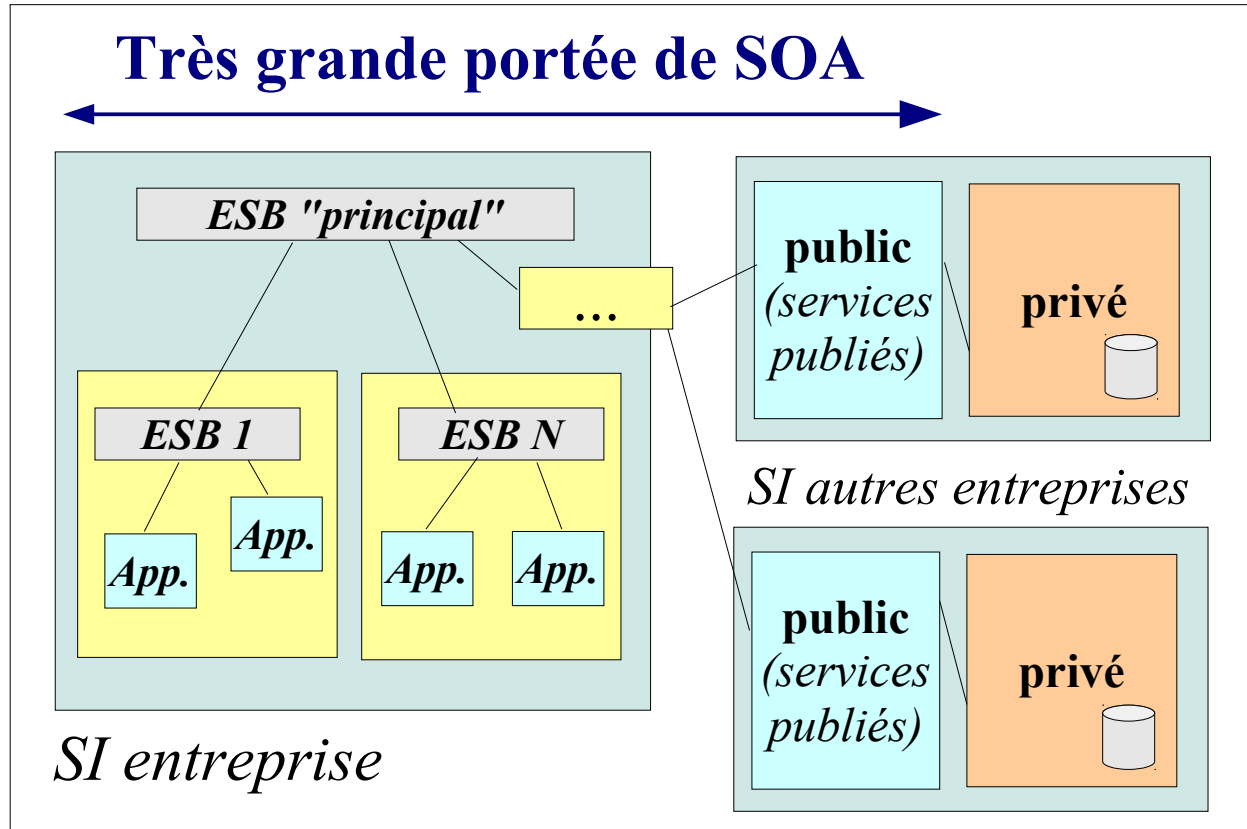
Une norme SOA du monde Java s'appelle JBI (Java Business **I**ntegration)

1.4. Très grande portée de SOA

SOA c'est un peu "programmer/configurer le SI à grande échelle".

Un document d'IBM de 2005 mentionnait déjà "**SOA = programming in the large**".

La portée générale d'une modélisation SOA est une partie du SI de l'entreprise (et pas une seule application)



Vue la **grande portée de SOA**, l'aspect **modélisation** est **très important**.

(petite portée → petites conséquences, **grande portée** → **grandes conséquences**)

==> Une **démarche prudente** s'impose absolument !

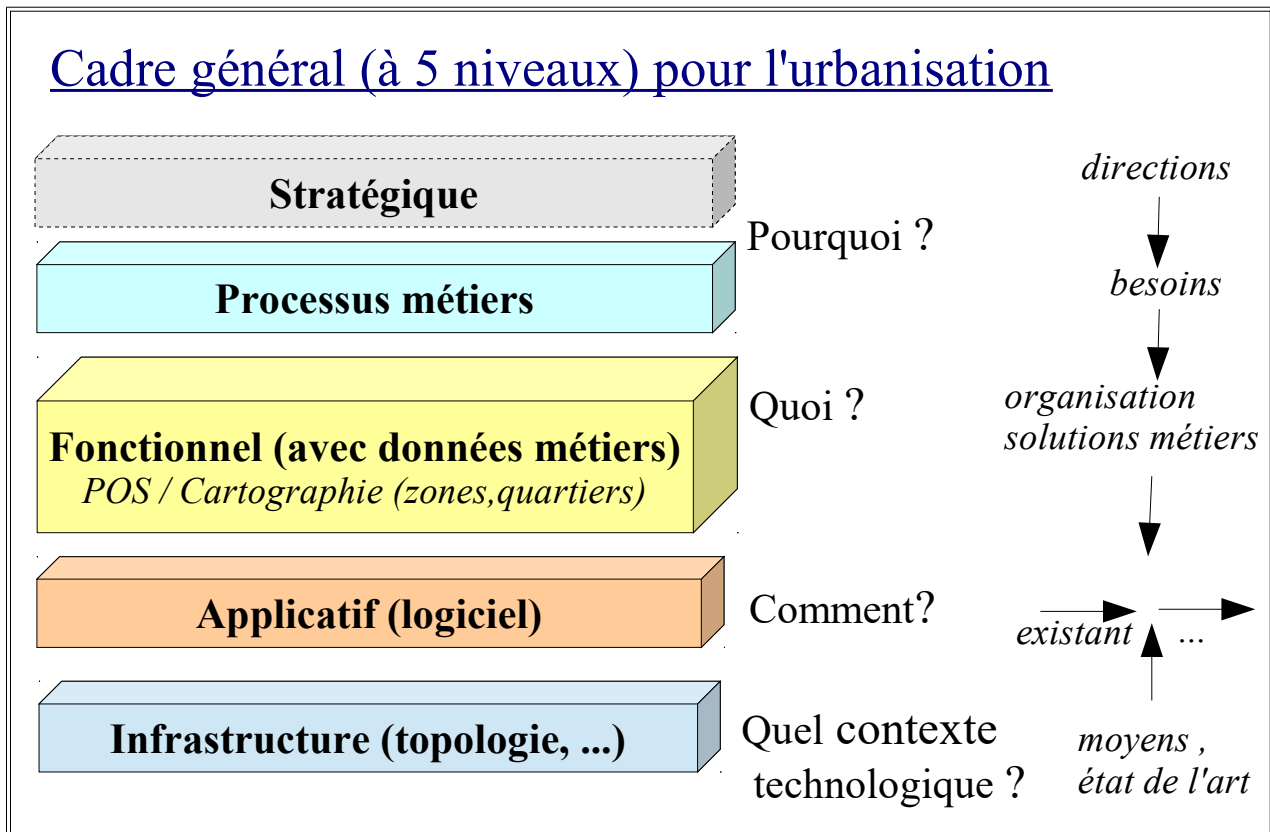
(1. prototype technique expérimental, 2. petit système, 3. système étendu)

2. Quelques typologies de services à assembler

Types de services	Principales caractéristiques
Service purement "métier"	Lié à un domaine métier précis (ex : comptabilité, facturation, ...) et idéalement réutilisable
Service "organisationnel"	Service spécifique à une organisation/entreprise, s'appuie souvent sur un ou plusieurs services "métier"
Service "technique/utilitaire"	Service utilitaire (plutôt technique). Ex: impression, mailing, ...

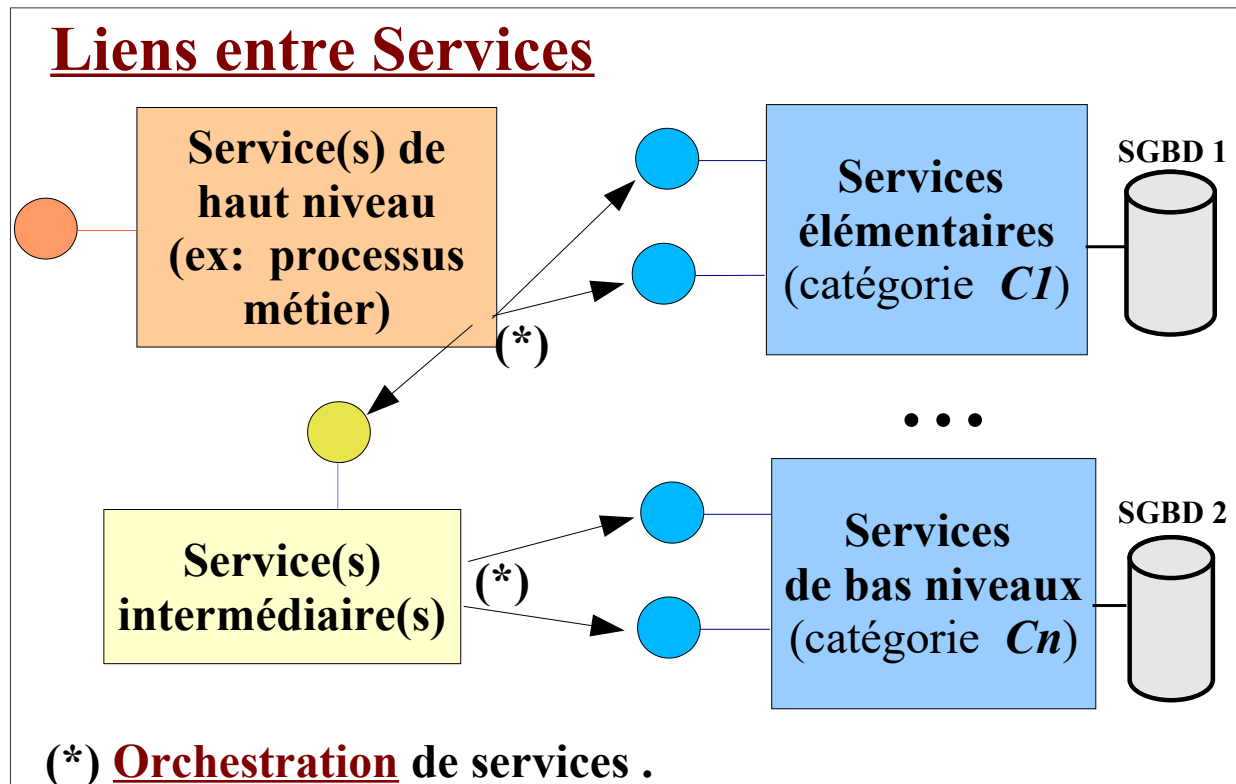
A également généralement distinguer : **"cœur de métier"** et **"support"**

3. Cadre pour la modélisation (SOA) d'entreprise



Ce cadre (très classique) servira de point de repère pour les activités de la modélisation "SOA".

4. Positionnement des niveaux de services



Services élémentaires (de bas niveaux) :

- Accès (potentiel) à des données , des règles métiers , ...
- Souvent liés à un seul secteur (ou une "catégorie") métier
- Quelquefois techniques (Authentification / Habilitation , Impression, ...)
- Services métiers distants (avec transactions courtes) [ex : Spring , EJB]
- ...

Services intermédiaires techniques :

- Adaptateurs (format, ...) , ...

Services intermédiaires fonctionnels:

- recombinaisons fonctionnelles (composition, assemblage , délégation ,)
- orchestrations simples (en mode synchrone) , ...

Services d'orchestrations évoluées (de hauts niveaux) :

- Processus métier (avec orchestration) → technologies **BPEL** ou **activiti** ou **jBpm**
modélisation **BPMN** , **BPMN2**
- processus potentiellement longs , opérations asynchrones,

=> Le plan de ce cours suivra une progression logique du simple/élémentaire vers le complexe/élaboré.

II - Modélisation des services élémentaires

1. Vision conceptuelle des services élémentaires

Un **service** est un *élément concret du Système d'Information rendant des services fonctionnels qui peuvent être utilisés par d'autres parties (applications) du SI.*

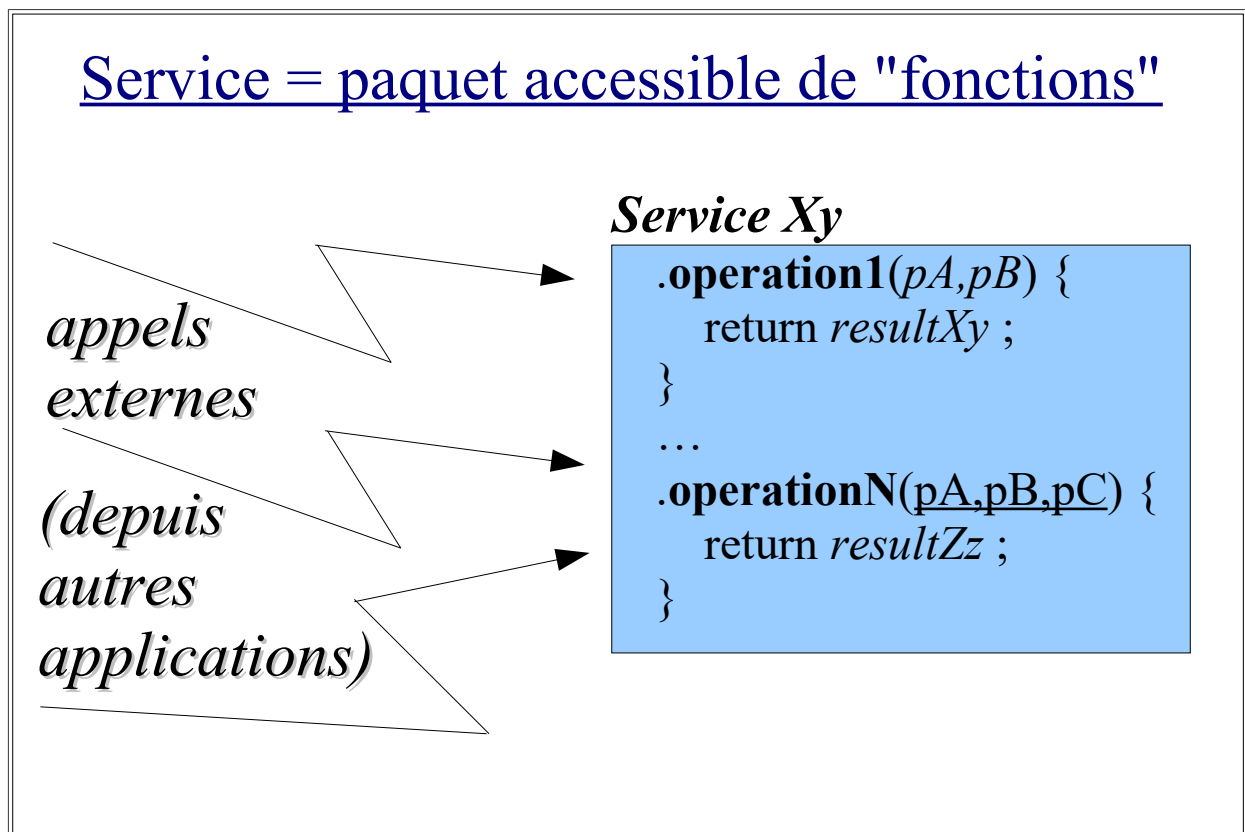
Les traitements effectués par un service sont généralement pris en charge par un composant interne d'une application. Cependant *les fonctionnalités d'un service sont exposées à l'extérieur de l'application hôte d'une manière assez universelle pour que presque toutes les autres applications puissent y accéder.*

Un **service web** est un service qui est invoqué avec une des technologies standardisées du web (HTTP , XML , JSON , ...) . On distingue essentiellement les "*Web-Services SOAP*" (toujours XML) et les "*Web-Services REST*" (toujours HTTP).

Dans l'architecture SOA , certains services sont basés sur d'autres technologies (anciennes, propriétaires, spécifiques, ...) et sont tout de même interopérables via des adaptateurs .

Un service a la particularité de pouvoir être invoqué depuis n'importe quel langage de programmation (ex: un service développé en "C++" peut être appelé depuis une application java). La principale fonctionnalité recherchée/visée dans l'architecture SOA est l'**interopérabilité** .

1.1. Vision "RPC (Remote Procedure Call)"



Chaque opération à son "contrat fonctionnel"

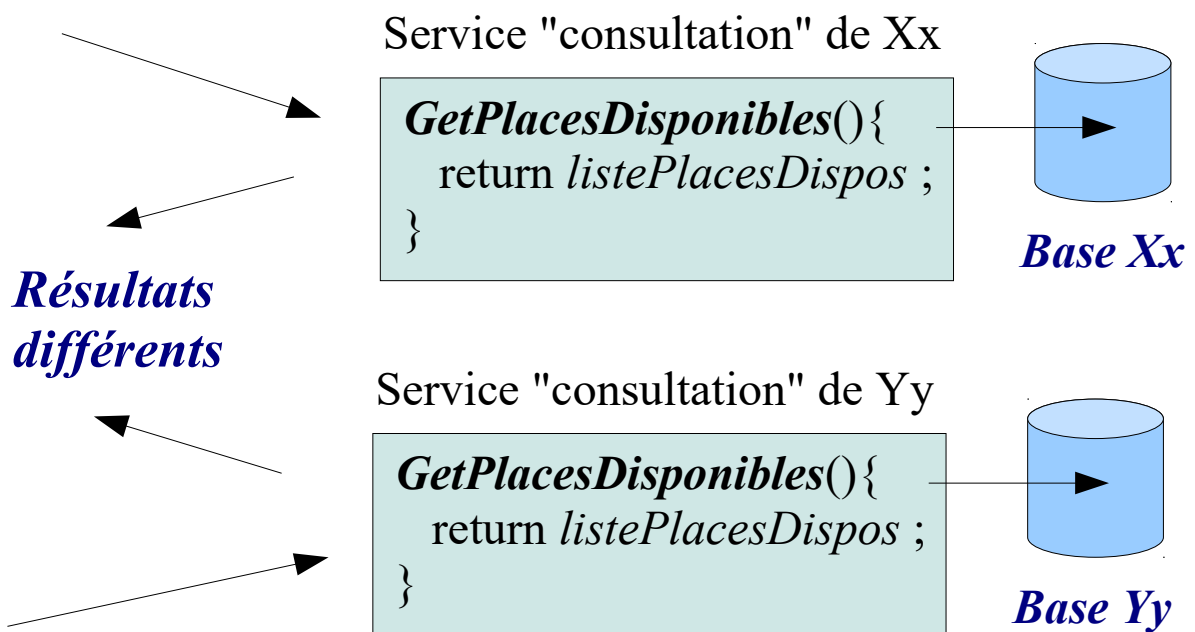
Requête (*avec paramètres
et types significatifs*) →

```
.operationXy(pA,pB) {  
    return resultXy ;  
}
```

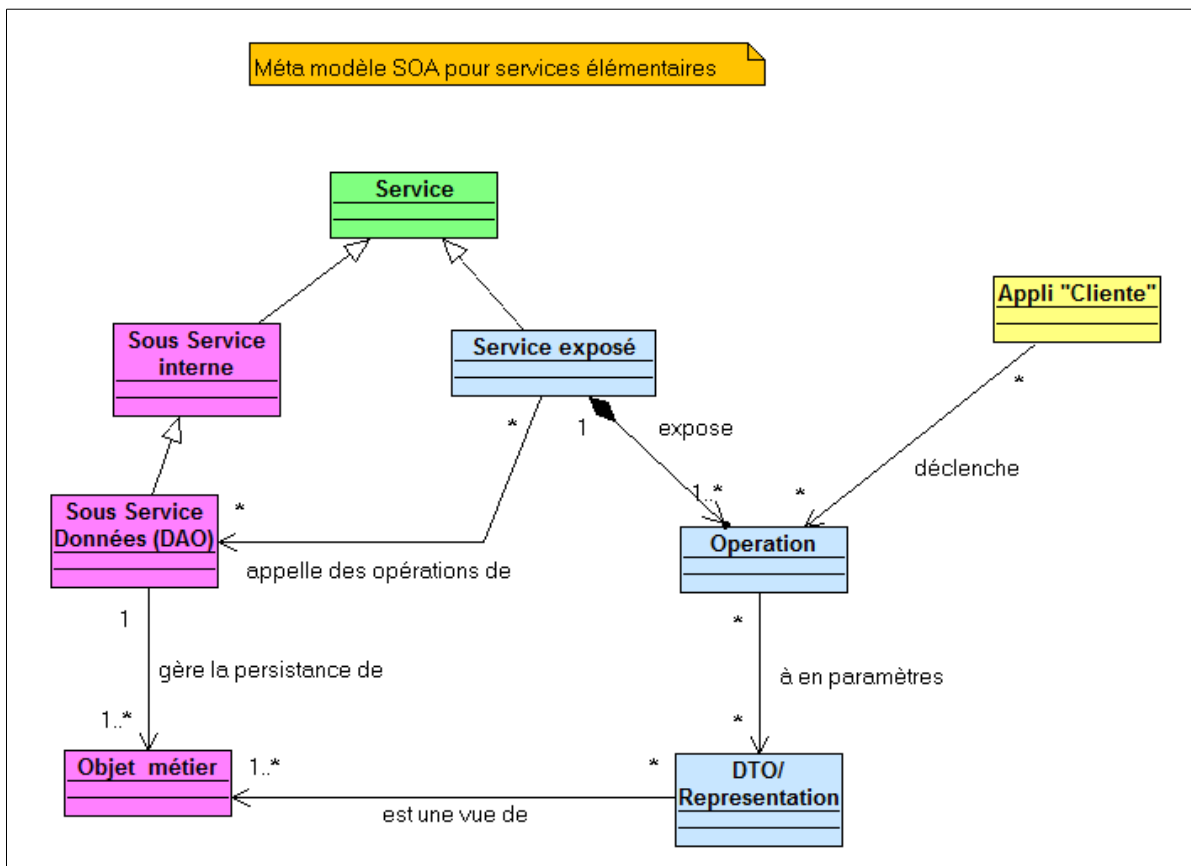
← **Réponse** (*ou exception*)
avec type et sémantique précis

Exemple : l'opération "*addition(a , b)*" attend en entrée les 2 opérandes à additionner et retourne le résultat de l'addition [*a , b et le résultat sont des nombres réels*].

2 instances de services avec même(s) opération(s)



1.2. Méta-modèle pour services élémentaires :



NB: Les paramètres des opérations appelées sur un service sont généralement des objets de données (nommés "**DTO : Data Transfert Objects**" ou "**Représentations**" ou ...).

[ex : une "représentation" d'une demande de devis , une "représentation" d'une facture ,]
Ces représentations sont généralement produites via des conversions opérées à partir d'entités "métier" (stockées dans des bases de données).

1.3. Vision "Message" (document , requête / réponse , ...)

Certaines technologies (Web-Services "REST" , communications par messages asynchrones ,) sont plutôt associés à une **logique d'échange de messages** .

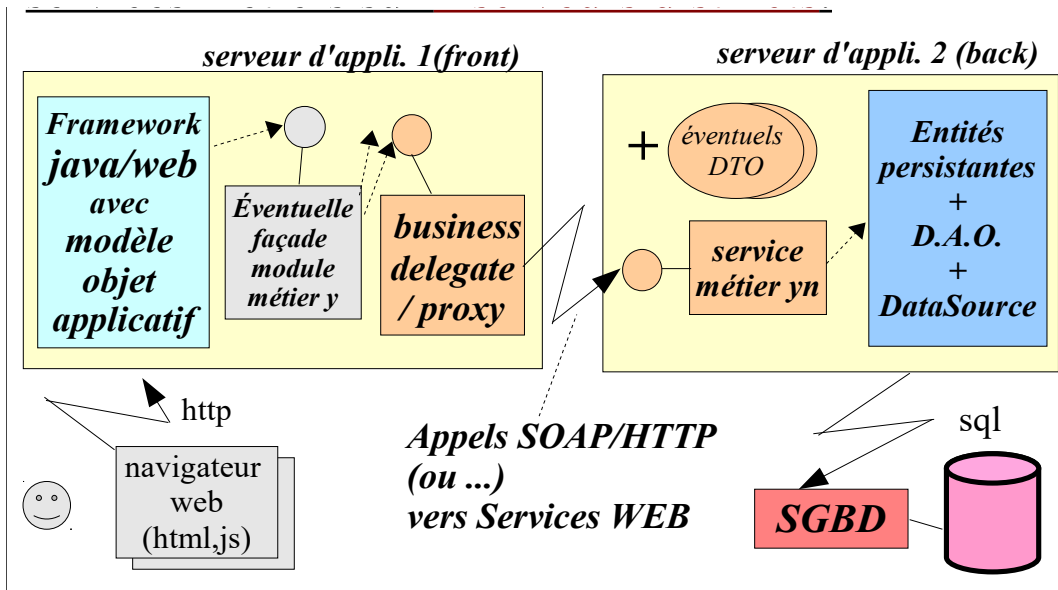
La modélisation devra donc (en partie) s'attacher à **décrire finement les structures des messages échangés** (ex : via **diagramme de classes UML**) .

1.4. Aspects asynchrones

La modélisation des services élémentaires asynchrones peut souvent s'effectuer via des jeux d'interfaces "one-way" :

- une **interface** pour les **appels sortants** (sans retour immédiat)
- une **interface "callback"** pour les **appels entrants** (notifications des réponses en différé).

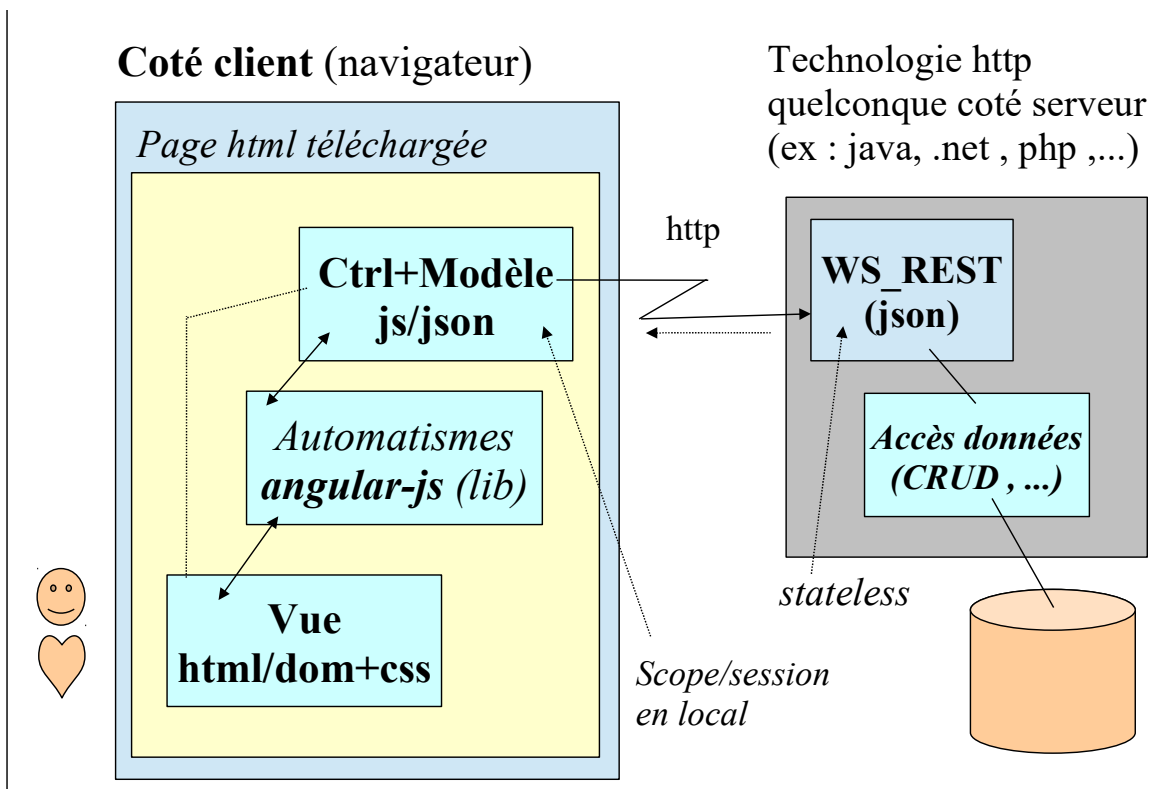
2. Cadre classique pour services élémentaires



Le schéma ci-dessus montre une architecture logicielle classique (ici basée sur Java/JEE) où les communications inter-applications sont centrées autour d'appels de méthodes distantes (sur des Web-Services "SOAP").

L'application qui fournit le service est souvent prise en charge par des technologies traditionnelles (Java/JEE, .net, ...) prenant en charge les transactions courtes et l'accès aux SGBDR.

Pour ne pas limiter l'illustration à des technologies purement "RPC", voici un autre exemple (ou type) de service élémentaire :

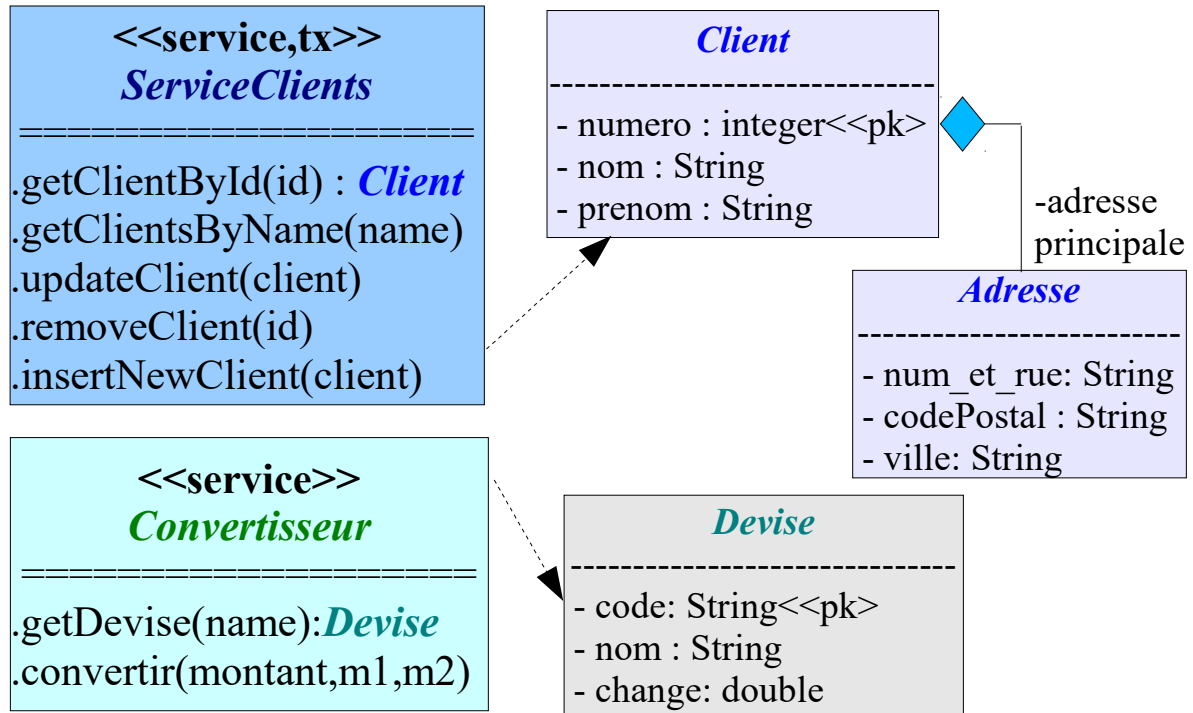


Message de réponse = représentation (UML → JSON) d'une ressource téléchargée.

3. Modélisation des services élémentaires

3.1. Modélisation essentielle visée

Modélisation UML essentielle d'un Webservice (exemples)



Les **services élémentaires** sont essentiellement modélisés comme des **classes (ou interfaces) UML**. Un service (en vision RPC) est avant tout une **liste d'opérations (méthodes distantes)**.

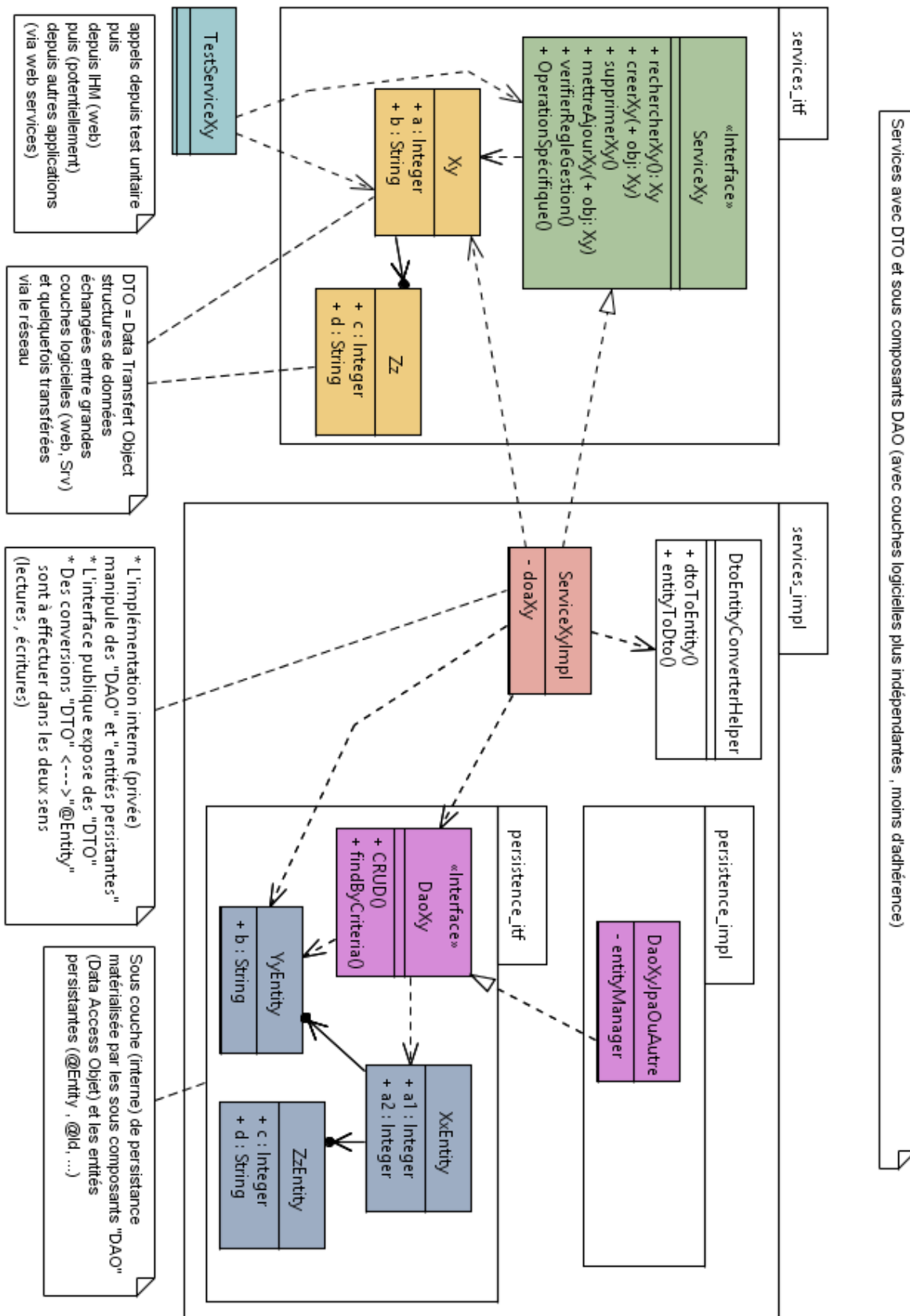
Pour que la **modélisation** du service soit **précise** il faut **indiquer** tous les **types** de données (pour les **paramètres d'entrées** et les **valeurs de retour**).

Dans de très nombreux cas les types des paramètres sont des structures objets complètes (*ici "Client", "Adresse" et "Devise"*) qui se modélisent comme d'autres classes UML.

Autre précision quelquefois importante: **"package englobant UML"** → namespace UML

3.2. Dto exposés par les services

Dans la cadre de la modélisation SOA , les structures de données exposées (en entrées et en sortie) par les méthodes d'un service sont à considérer comme des données d'échanges *(représentation externes quelquefois différentes du format d'implémentation interne)* .



3.3. Démarche méthodologique classique (autour d'UML)

Pour qu'un **service élémentaire** soit vraiment **utile** , il faut qu'il **rende le bon service** !

La démarche méthodologique classique permettant d'analyser les services utiles consiste à :

1. ré-exprimer certains besoins fonctionnels via un diagramme de "uses-cases" UML
2. accrocher un rapide scénario à chaque use-case .
3. illustrer éventuellement certains use-cases via des diagrammes d'activités (fin du processus = objectif visé au niveau du cas d'utilisation).
4. rédiger un glossaire (dictionnaire des données) comportant une description des entités du domaine. Effectuer éventuellement un regroupement en packages.
5. transposer le glossaire en un (ou plusieurs) diagramme(s) de classes UML.
6. ajouter des classes de "services" pour gérer chaque entité fondamentale .
7. transposer les scénarios (rattachés aux uses-cases) en diagrammes de séquences et (par cohérence) faire émerger de nouvelles méthodes dans les classes de services.
8. laisser reposer quelques temps cette première modélisation.
9. se relire avec un esprit critique et peaufiner / restructurer si besoin certains points

NB : cette séquence sera (assez rapidement) étudiée en TP.

La première annexe du cours permet (si nécessaire) d'étudier certaines syntaxes UML .

4. Spécifications fines des services (détails)

La plupart des points qui seront détaillés ci-après peuvent souvent être spécifiés de manière générique (à appliquer systématiquement au sein d'un certain contexte / projet) .

4.1. pré-conditions, post-conditions

Quelques opérations importantes ne peuvent être effectuées que si certaines conditions importantes sont vérifiées (au début/avant ou après le traitement principal).

Ces règles de gestions sont souvent matérialisées par des sous méthodes de vérifications qui sont

- quelquefois déclenchées automatiquement à partir de la méthode principale (pré/post)
- éventuellement appelées explicitement avant (pour s'assurer que l'opération est déclenchable) ou bien après (pour vérifier que l'opération a bien été effectuée et enregistrée)

Lorsqu'une méthode de vérification n'arrive pas à vérifier une règle de gestion, elle doit idéalement remonter une exception précisant la raison de l'échec .

4.2. exceptions ou signaux

En **mode synchrone** une opération invoquée sur un service web retourne normalement une **exception** en cas de déroulement anormal du traitement effectué ou en cas de paramètres erronés en entrée. Une exception "java" (ou C# ".net") est transformée en "**Fault**" **SOAP/XML** .

En **mode asynchrone** , il faut généralement prévoir au moins 2 *sortes de notifications* :

- une **notification positive** du genre "demande bien reçue , en cours de traitement"
- une **notification négative** du genre "impossible de traiter la demande (avec raisons)" .

[l'instruction "peek" de BPEL est par exemple capable de récupérer plusieurs sortes de notifications en retour des invocations asynchrones]

Une modélisation complète/fine peut donc spécifier les exceptions susceptibles d'intervenir.

On pourra éventuellement s'appuyer sur des **exceptions génériques** (pour le SI ou bien par domaine) de façon à simplifier/alléger la modélisation .

L'important (au niveau de la modélisation) est de bien spécifier :

- le besoin de remonter (quasi systématiquement) des exceptions (ou notifications asynchrones) en cas de problème (du côté serveur / implémentation)
- le besoin de tenir compte des exceptions ou notifications (du côté client) en les transformant en messages d'erreurs dans l'IHM ou bien en mentionnant des routes spéciales/alternatives dans les processus métiers.

4.3. Définition des types complexes associés

Une opération appelée sur un service n'est parfaitement intelligible (sans ambiguïté) que si les paramètres d'entrée et de retour sont parfaitement définis.

La plupart des technologies et langages de programmation n'autorisent qu'une seule valeur de retour. Celle ci doit donc assez souvent être d'un type "complexe / décomposé".

Tous ces types complexes pourront :

- être modélisés finement dans des diagrammes de classes **UML**
- être implémentés comme des objets (**java** , **c++** , **c#** , ...) en mémoire
- être échangés sous formes de structure **XML** (ou **JSON** ou) à travers les réseaux informatiques .

Dans le cas d'un format d'échange XML , ces types complexes seront spécifiés au sein de schéma **XSD** via la balise `<xsd:ComplexType>` .

Quelques exemples :

effectuerLivraison(demande : **DemandeLivraison**) : **AccuseReception**

La structure de donnée **DemandeLivraison** peut être décomposée en :

- un sous objet "Adresse" (de livraison)
- une liste de sous objet "Produit" (à livrer)
- un numéro de commande associé à la livraison
- ...

La structure de donnée **AccuseReception** peut être décomposée en :

- un message "A.R."
- un rappel du numéro de commande (véhiculé dans la demande)
- une date de prise en compte
- ...

getPlacesDisponibles(criteres : **CritereRecherche**) : **ListePlacesDisponibles**

La structure de donnée **CritereRecherche** peut être décomposée en :

- une quantité demandée
- une plage de prix
- des préférences
- ...

La structure de donnée **ListePlacesDisponibles** peut être décomposée en :

- une liste de "**PlaceDisponible**"
- ...

La (sous) structure de donnée **PlaceDisponible** peut elle même être décomposée en

- une référence
- un prix
- des caractéristiques
- ...

4.4. gestion de variantes et contextualisation

En règle générale, les **types de données complexes** liés aux opérations des services sont des **vues** (et non pas directement des entités persistantes).

Il est donc envisageable qu'un même service puisse **fournir différentes vues/représentations (variantes) d'une même chose** en fonction:

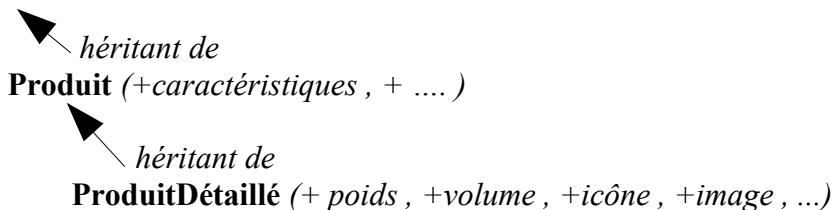
- d'appels vers différentes *opérations ayant explicitement des noms évoquant des variantes*
- d'un *contexte fonctionnel* passé explicitement *en argument* des opérations appelées
- d'un *contexte technique (ou fonctionnel)* prise en charge par des *intercepteurs* et dont l'analyse même à certaine *décoration automatique* (via par exemples des mécanismes AOP).

Comme **bases structurelles des variantes** on peut penser à :

- l'**héritage** (lorsque c'est technique possible)
- un ou plusieurs **sous élément(s) "extension" optionnel(s)**.
- ...

Exemple1 (via héritage) :

EssentielProduit (*ref, designation, prix*)



Exemple2 (via extensions imbriquées) :

Produit (*ref, designation, prix ,
caractéristique éventuellement vide , liste_extensions_produit*)
 et avec **ExtensionProduit** = **ExtensionGenerique**(*nom , valeur*)
 ou bien **ExtensionProduit** (*poids, volume , icône , image ,*) ou ...

Remarque technique:

Les technologies "web services" les plus perfectionnées (ex : JAX-WS de java , équivalent .net, ...) sont capables de gérer une sorte de **polymorphisme**. Elles peuvent donc retourner et transférer à travers le réseau alternativement plusieurs variantes d'une même information demandée.

En java, ce paramétrage peut s'effectuer via `@XmlElement { XmlElement("TypeVariante1") , XmlElement("TypeVariante2") }` et encodé en `<xsd:choice>` coté XDS/WSDL ou bien (ce qui encore mieux) via `@XmlSeeAlso ("SubType1", "SubType2")` .

4.5. Contrat d'utilisation, spécification de la qualité de service

On peut également spécifier et mettre en œuvre certains contrats d'utilisation (techniques et/ou fonctionnels) :

- service accessible que si *abonnement* préalable du client (num_compte ou)
- service plus ou moins réactif selon *priorité* attachée au client

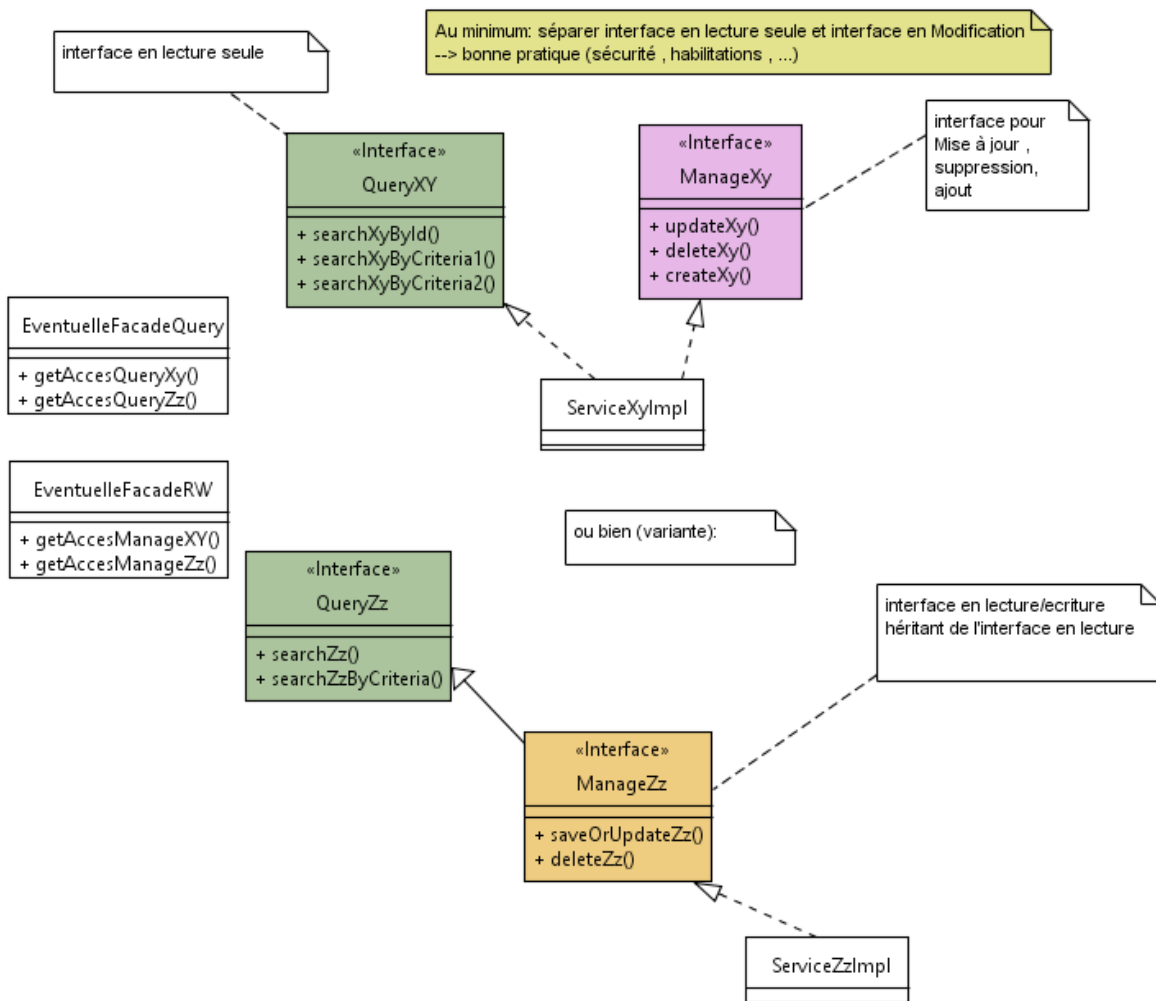
D'autre part, les spécifications fonctionnelles peuvent exiger certaines qualité de service :

- *cryptage* / *confidentialité* requis(e), *authentification* requise,
- *temps de réponse maximum supportable*, ...

La plupart de ces critères correspondent à des "*aspects*" complémentaires à la structure proprement orientée objet des services et se modélisent assez bien comme des stéréotypes à généralement placer sur l'ensemble d'un service (ou éventuellement sur une ou plusieurs méthodes précises) :

- *<<confidential>>*
- *<<auth>>*
- *<<member_only>>*
- *<<1s_max>>*, *<<2s_max>>*, *<<5s_max>>*, *<<10s_max>>*, *<<15s_max>>*, *<<20s_max>>*

4.6. Séparation des interfaces en "lecture" et en "écriture" :



III - Transpositions vers contrats WSDL ou YAML

1. Transpositions "UML" en contrats WSDL

1.1. Construire des schémas XML interoperables

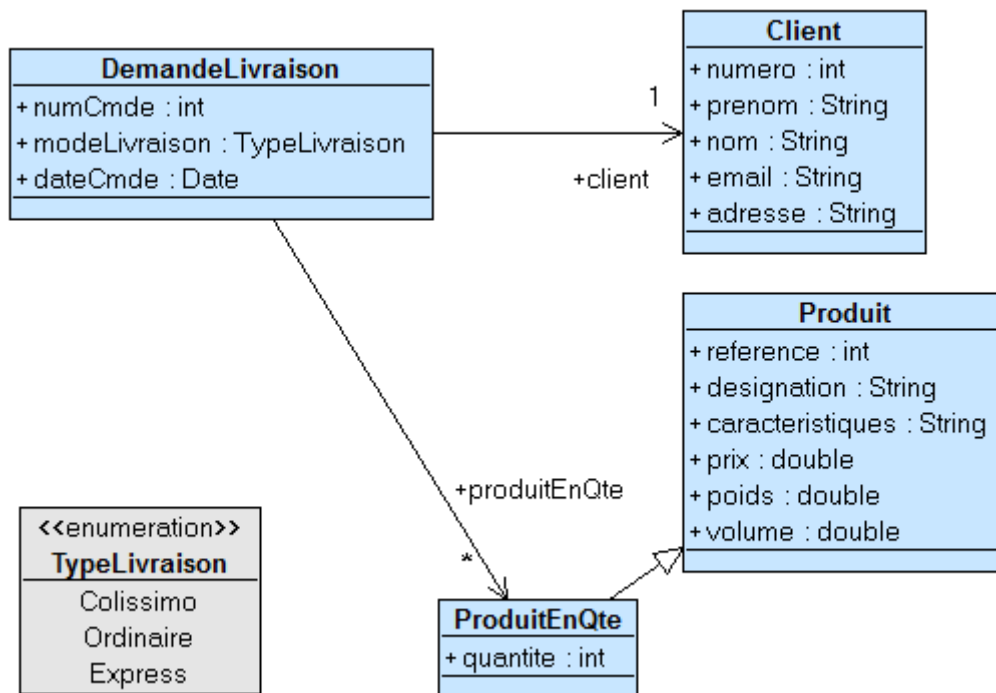
La plupart des types objets complexes modélisés en UML peuvent être transposés (de façon assez systématique) en schéma XSD (utiles pour les descriptions WSDL des services WEB) .

- **Nom de classe UML** ---> **nom de complexType XSD**
- **Nom de package UML** ---> à **transposer** en **targetNamespace XML/XSD** selon une certaine convention.

Exemple : si le package fonctionnel UML est "comptabilite.journal" alors le namespace xml peut être un de ceux-ci:

- <http://journal.comptabilite.xy.com/>
- urn://comptabilite:journal
- <http://www.mycompany.com/si/comptabilite/journal>
- ...

Exemple de structure complexe XSD (avec correspondance UML) :



produit.xsd

```

<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:nfprod="http://produit.nf.org" targetNamespace="http://produit.nf.org"
  elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xsd:element name="produit" type="nfprod:Produit"/>
  <xsd:element name="produitEnQte" type="nfprod:ProduitEnQte"/>

  <xsd:complexType name="Produit">
    <xsd:sequence>
  
```

```

        <xsd:element name="reference" type="xsd:int"/>
        <xsd:element name="designation" type="xsd:string"/>
        <xsd:element name="caracteristiques" type="xsd:string"/>
        <xsd:element name="prix" type="xsd:double"/>
        <xsd:element name="poids" type="xsd:double"/> <!-- en kg -->
        <xsd:element name="volume" type="xsd:double"/> <!-- en l -->
    </xsd:sequence>
</xsd:complexType>

<!-- produitEnQte(avec quantite) heritant de produit -->
<xsd:complexType name="ProduitEnQte"> <xsd:complexContent>
    <xsd:extension base="nfprod:Produit">
        <xsd:sequence>
            <xsd:element name="quantite" type="xsd:int"/>
        </xsd:sequence>
    </xsd:extension> </xsd:complexContent>
</xsd:complexType>

</xsd:schema>

```

demandeLivraison.xsd

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:nflivr="http://livraison.nf.org" xmlns:nfprod="http://produit.nf.org"
    xmlns:nfcli="http://client.nf.org" targetNamespace="http://livraison.nf.org"
    elementFormDefault="qualified" attributeFormDefault="unqualified">
    <xsd:annotation>
        <xsd:documentation>demande de livraison</xsd:documentation>
    </xsd:annotation>
    <!-- importation du "sous-schéma" décrivant la structure d'un produit -->
    <xsd:import namespace="http://produit.nf.org" schemaLocation="produit.xsd" />
    <!-- importation du "sous-schéma" décrivant la structure d'un client -->
    <xsd:import namespace="http://client.nf.org" schemaLocation="client.xsd" />
    <xsd:element name="demandeLivraison" type="nflivr:DemandeLivraisonType"/>

    <xsd:simpleType name="TypeLivraison">
        <xsd:restriction base="xsd:string">
            <xsd:enumeration value="Colissimo"/>
            <xsd:enumeration value="Ordinaire"/>
            <xsd:enumeration value="Express"/> <!-- and so on ... -->
        </xsd:restriction>
    </xsd:simpleType>

    <xsd:complexType name="DemandeLivraisonType">
        <xsd:sequence>
            <xsd:element name="numeroCmde" type="xsd:int"/>
            <xsd:element name="modeLivraison" type="nflivr:TypeLivraison" />
            <xsd:element name="dateCmde" type="xsd:date"/>
            <xsd:element ref="nfcli:client" />
            <xsd:element ref="nfprod:produitEnQte"
                minOccurs="0" maxOccurs="unbounded" />
        </xsd:sequence>
    </xsd:complexType>

```

```

    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>

```

exemple de contenu xml ayant une structure valide vis à vis du schéma ci dessus :

```

<?xml version="1.0" encoding="UTF-8"?>
<demandeLivraison xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://livraison.nf.org demandeLivraison.xsd"
  xmlns="http://livraison.nf.org" >
  <numeroCmde>123</numeroCmde>
  <modeLivraison>Colissimo</modeLivraison>
  <dateCmde>2010-01-20</dateCmde>
  <client xmlns="http://client.nf.org">
    <numero>1</numero>
    <prenom>alain</prenom>
    <nom>Therieur</nom>
    <email>xxx@yyy.fr</email>
    <adresse>12 rue Elle, 75000 Par ici</adresse>
  </client>
  <!-- debut listeProduits-->
    <produitEnQte xmlns="http://produit.nf.org">
      <reference>1</reference>
      <designation>cahier</designation>
      <caracteristiques> petits carreaux , 100 pages </caracteristiques>
      <prix>12.5</prix>
      <poids>0.120</poids>
      <volume>0.0239</volume>
      <quantite>2</quantite>
    </produitEnQte>
    <produitEnQte xmlns="http://produit.nf.org">
      <reference>2</reference>
      <designation>trousse</designation>
      <caracteristiques> petite trousse rouge </caracteristiques>
      <prix>4.5</prix>
      <poids>0.20</poids>
      <volume>0.0439</volume>
      <quantite>4</quantite>
    </produitEnQte>
  <!-- fin listeProduits -->
</demandeLivraison>

```

Ce genre de structure XML peut former la base des échanges d'informations entre les services web (synchrones ou asynchrones) :

- XSD importé dans WSDL
- XSD → pour générer code de parsing JAXB2 → pour charger par code java des documents xml en mémoire .

1.2. Construire des descriptions WSDL abstraites et stables

Un fichier WSDL (décrivant un service web) importe des sous structures XSD (vues précédemment et liées aux objets complexes passés en paramètres ou bien en valeur de retour) .

La partie fondamentale d'un fichier WSDL est le "**PortType**" abstrait correspondant à la notion d'**interface de service** .

Toute la structure d'un "**PortType WSDL**" est directement dérivée de la structure d'un **service métier** (décrit comme une *classe* ou comme une *interface* en UML) .

Autrement dit , la liste des **opérations** du service UML est transposée en une liste d'opérations du PortType WSDL avec les mêmes paramétrages (types de paramètres en entrée et en retour ,exceptions éventuelles).

Ceci dit, cette transposition est sémantiquement claire mais syntaxiquement pas très lisible (du fait de la complexité XML de la norme WSDL) .

La partie haute d'un fichier **WSDL** (avec *import XSD* et *PortType*) forme la **partie abstraite** correspondant à un **contrat de service SOA** .

Cette partie est fondamentale puisqu'elle constitue la base des paramétrages des principaux ESB. Les fichiers **WSDL** peuvent être vus comme des **mini-normes** (internes ou partagées) et méritent d'être stockés dans certains **référentiels** (annuaire ou ...) .

1.3. Eventuel emploi de l'approche MDA

Ecrire directement (avec un clavier et un éditeur texte) des fichiers XSD et WSDL est un exercice très fastidieux (c'est long et syntaxiquement délicat/prise de tête) .

Même avec des éditeurs/assistants spécialisés , l'exercice est assez laborieux .

Il est fortement conseillé de trouver un moyen direct ou indirect pour générer ces fichiers (quitte à les retoucher partiellement si nécessaire).

Deux grandes solutions sont envisageables :

- générer des classes JAVA (ou C#) depuis UML pour indirectement générer du WSDL
- générer directement des fichiers XSD/WSDL depuis UML via un générateur MDA (ex: accéléo M2T)

Bien que ça demande un investissement en temps assez conséquent , la mise au point (par écriture de templates "accéléo_m2t") d'un générateur de code "java_ws + wsdl/xsd +" en fonction d'UML offre l'avantage de pouvoir régénérer rapidement tout un tas de structures cohérentes entre elles dès que la modélisation UML évolue (nouvelle version).

2. Contrats "YAML" pour Web-services "REST"

Les web-services "SOAP" sont depuis l'origine décrits par des fichiers WSDL (norme officielle du W3C).

A l'inverse la structure d'un web-service REST n'est pas (encore) associée à une norme officielle.

WAML (en XML) semble être une description abandonnée.

Par contre la syntaxe "YAML" (à base d'indentations) semble être adoptée par une grande communauté de développeurs.

Le vocabulaire "Api REST" est quelquefois employé pour nommer les spécifications structurales d'un web-service REST.

Il existe aujourd'hui au moins 3 outils/formalismes permettant de décrire la structure d'un web-service "REST" :

Swagger	Existe depuis plusieurs années Déjà beaucoup utilisé Bien outillé (générateurs)	Syntaxe YAML
RAML (<i>RESTful API Modeling Language</i>)	Plus récent (à partir de 2013) Peut être plus simple (mieux conçu) ?	Syntaxe YAML
API blueprint	?	?

Trois pistes pour transposer un modèle UML en YAML:

- Saisie manuelle de la description "YAML" depuis une lecture d'un diagramme UML.
- Génération directe MDA (avec interprétation de stéréotypes spécifiques) → pas facile
- Modèle UML (avec adaptateur REST) → **transposition en java (JAX-RS)** avec ou sans MDA → **génération d'une description YAML** (via outils/plugin de **swagger**)

Exemple de description YAML (en version "swagger") :

Partie 1	Partie 2 (suite)
<pre> swagger: "2.0" info: version: 1.0.0 title: Swagger Petstore license: name: MIT host: petstore.swagger.io basePath: /v1 schemes: - http consumes: - application/json produces: - application/json paths: /pets: get: summary: List all pets operationId: listPets tags: - pets parameters: - name: limit in: query description: How many items to return required: false type: integer format: int32 responses: 200: description: An paged array of pets headers: x-next: type: string description: link next responses schema: \$ref: Pets default: description: unexpected error schema: \$ref: Error post: summary: Create a pet operationId: createPets tags: - pets responses: 201: description: Null response default: description: unexpected error schema: \$ref: Error </pre>	<pre> /pets/{petId}: get: summary: Info for a specific pet operationId: showPetById tags: - pets parameters: - name: petId in: path required: true description: The id of pet to retrieve type: string responses: 200: description: Expected response schema: \$ref: Pets default: description: unexpected error schema: \$ref: Error definitions: Pet: required: - id - name properties: id: type: integer format: int64 name: type: string tag: type: string Pets: type: array items: \$ref: Pet Error: required: - code - message properties: code: type: integer format: int32 message: type: string </pre>

IV - Services fonctionnels

1. Services fonctionnels (définition de la catégorie)

Un **service** (*purement*) **fonctionnel** est un **service intermédiaire** (non élémentaire) qui s'appuie à son tour sur d'autres services .

Autrement dit : "un service peut en cacher un autre " . Ce qui est très fréquemment le cas dans une architecture SOA.

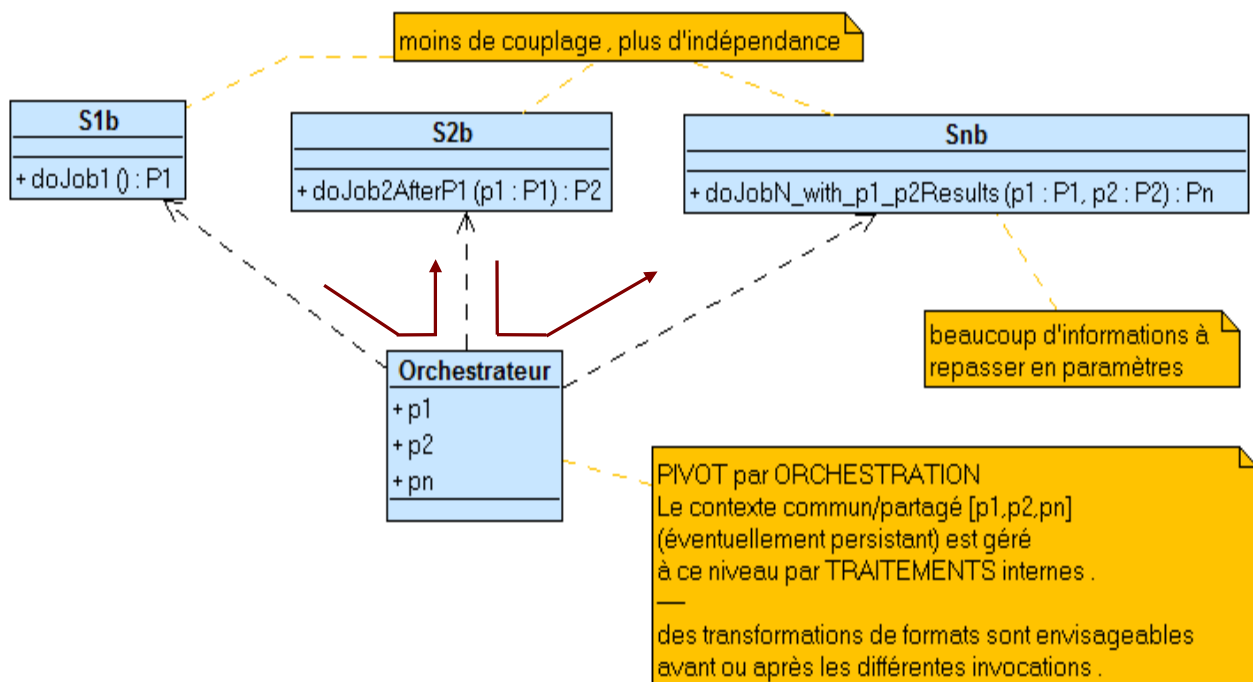
Contrairement aux services évolués d'orchestrations (souvent avec des parties asynchrones) , **un service fonctionnel reste simple** (*traitements synchrones et orchestration élémentaire*) .

Un service fonctionnel doit pouvoir être programmé et mis en œuvre avec des technologies simples (ex : classe java) . Pas besoin de bpmn / bpe1 .

Un service fonctionnel effectue essentiellement des **recombinaisons fonctionnelles** (*composition, assemblage , délégation , appels conditionnés ,*)

2. Données "pivots" (échangées)

Données "pivot"



Récupérées comme résultat d'un appel sur un premier service et ré-envoyées en entrée d'un autre service, certaines données pivotent (telles quelles ou retransformées) au niveau de l'orchestrateur.

...

3. Modélisation des services fonctionnels

La modélisation d'un service fonctionnel comporte **trois aspects complémentaires importants** :

- **structure du service rendu**
(même modélisation que pour un service élémentaire)
- **dépendances vis à vis d'autres services**
(via "dependency UML" au sein d'un diagramme de classes ou ...)
- **logique / algorithme des sous-appels internes**
(via "**diagrammes de séquences UML**" ou bien "**diag d'activités UML**" ou bien ...)

4. Intégration de l'existant (adaptateurs)

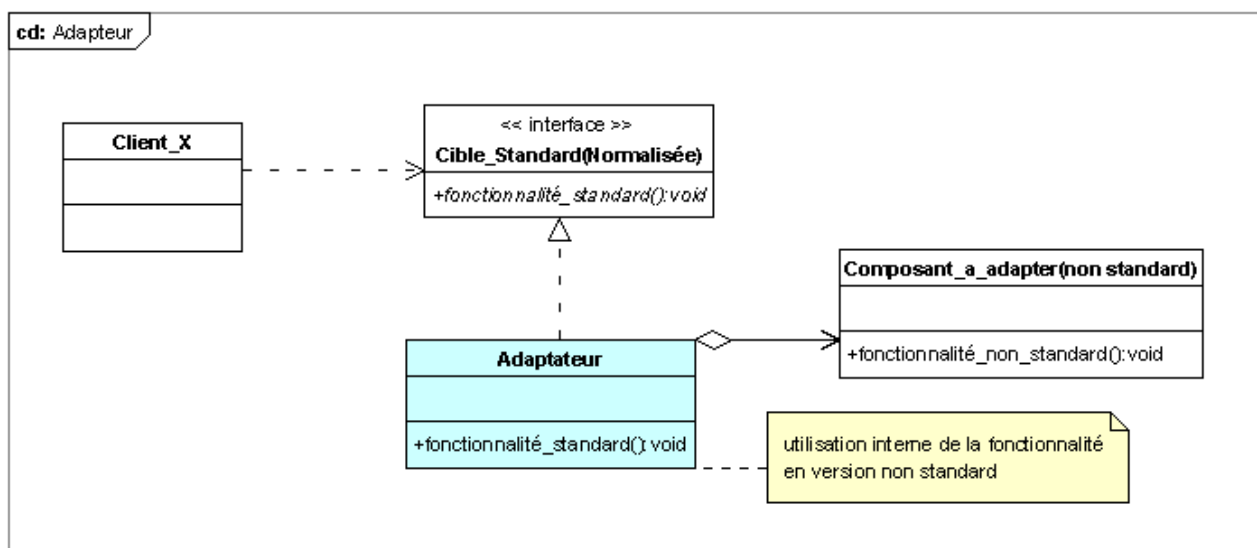
Via des transformations (généralement effectuées au niveau des E.A.I. / E.S.B.) , il est possible d'invoquer un service dont l'interface fonctionnelle ne coïncide pas tout à fait avec la structure (normalisée ou pas) de l'appelant .

4.1. Design pattern adaptateur

Problème courant: un composant "Ca" doit utiliser "Cx" mais l'interface de "Cx" ne convient pas .
D'autre part, les composants "Ca" et "Cx" ont des interfaces figées que l'on ne peut pas modifier.

Solution ==> introduire un composant intermédiaire Ci qui va :

- implémenter l'interface attendue par l'appelant "Ca"
- re-déléguer en interne à Cx la plupart des appels/fonctionnalités



Concrètement l'application SOA du design pattern "*adaptateur*" se concrétise généralement par un **service intermédiaire** effectuant *des transformations dans les 2 sens* :

- requête au format "appelant" à transformer au format "appelé"

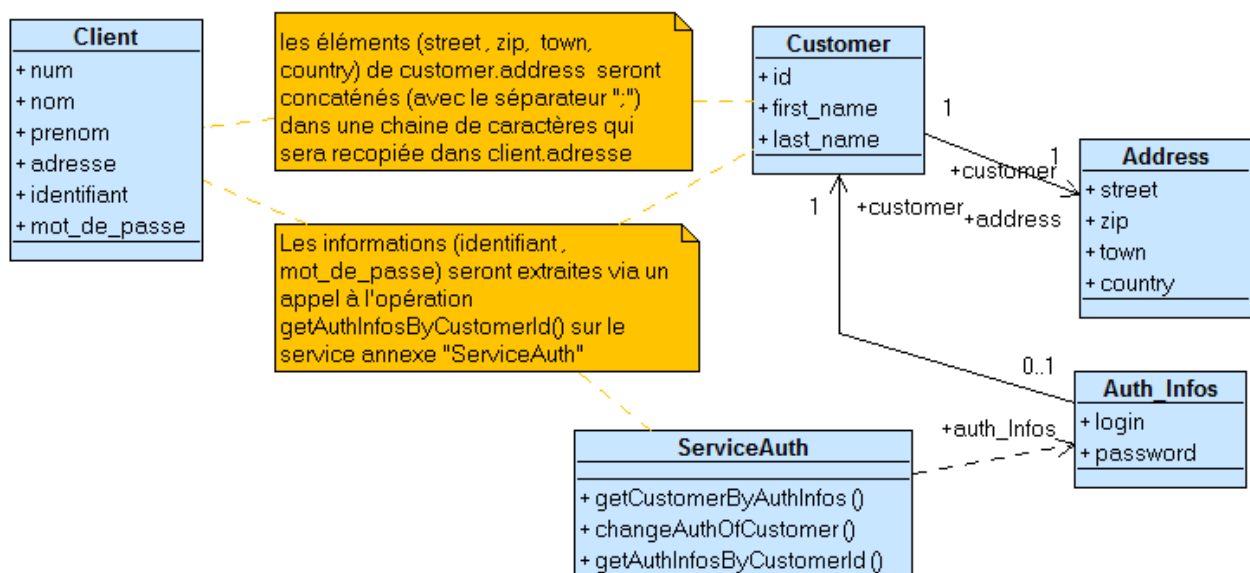
- réponse au format de l'appelé à transformer au format de l'appelant

Ceci peut être *technologiquement* effectué via des *transformations XSLT* ou du *code java spécifique* ou bien encore par des *mappings spécifiques à l'ESB* .

Ce qui est important , c'est de spécifier les mappings qui ne sont pas triviaux (autres que "nom" transformé en "name" ou "zip" transformé en "code_postal") .

4.2. Spécification des adaptations à réaliser

La solution la plus simple (et la plus intuitive/compréhensible) pour spécifier les adaptations/transformations à réaliser consiste à juxtaposer dans un même diagramme de classes les 2 structures à convertir et de placer au milieu un ou plusieurs commentaire(s) expliquant les transformations non triviales à effectuer :



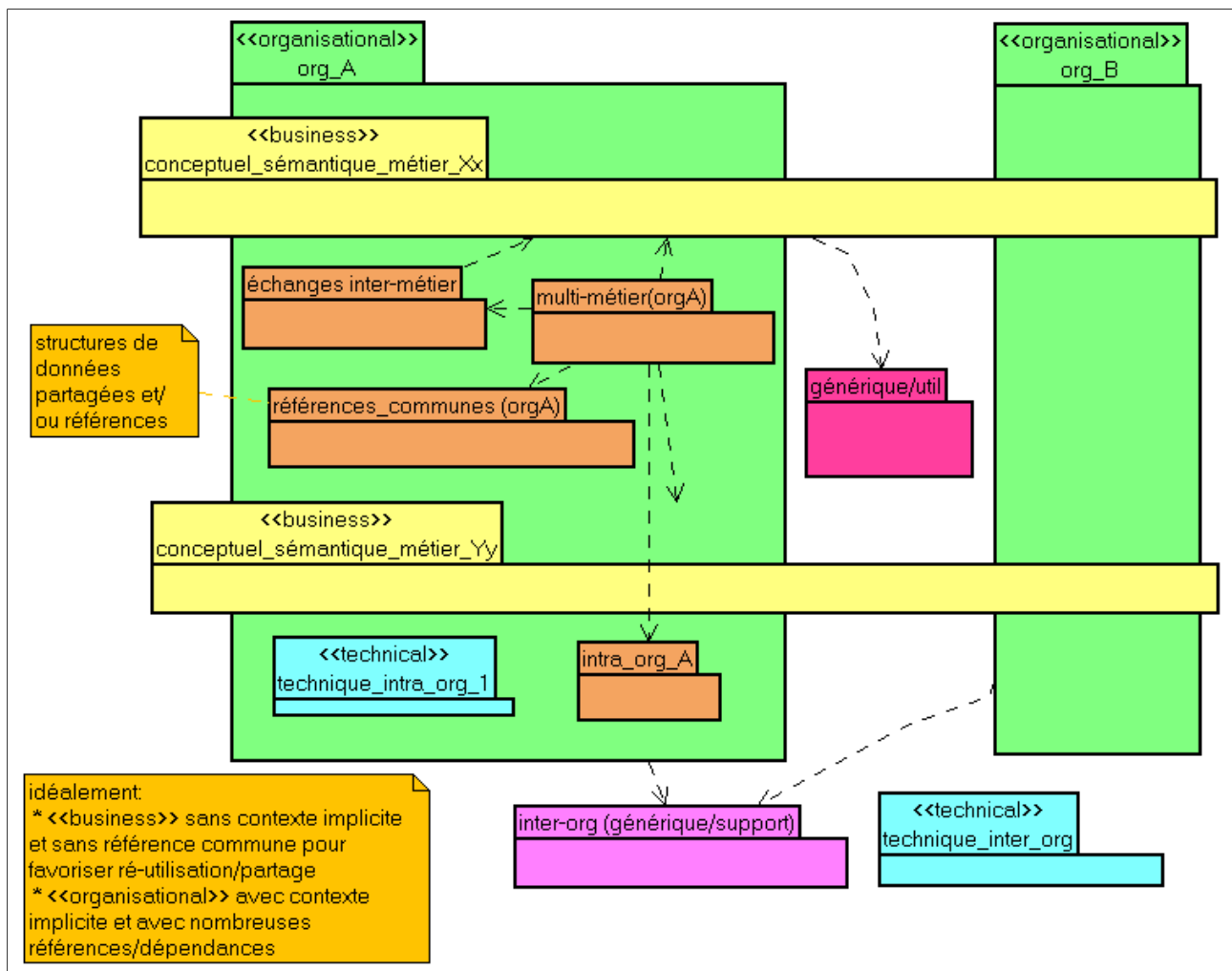
V - Modélisation métier (processus , bpmn, ...)

1. Structure cible d'une modélisation SOA

1.1. Distinguer le "pur fonctionnel/métier" de l'organisationnel

Distinguer au besoin :

- domaines purement métiers (comptabilité , R&D , marketing , ...) avec services métiers élémentaires
- domaines organisationnels (départements , centres de gestion ,) avec contextes
- domaines purement techniques (facilement réutilisables , sans contexte , ...)



...

2. Diagrammes sémantiques (pré-étude facultative)

La modularité est recherchée (au niveau des modules métiers à organiser par "secteurs")

En plus de l'essentielle modélisation des processus métiers, une bonne modélisation métier (orientée

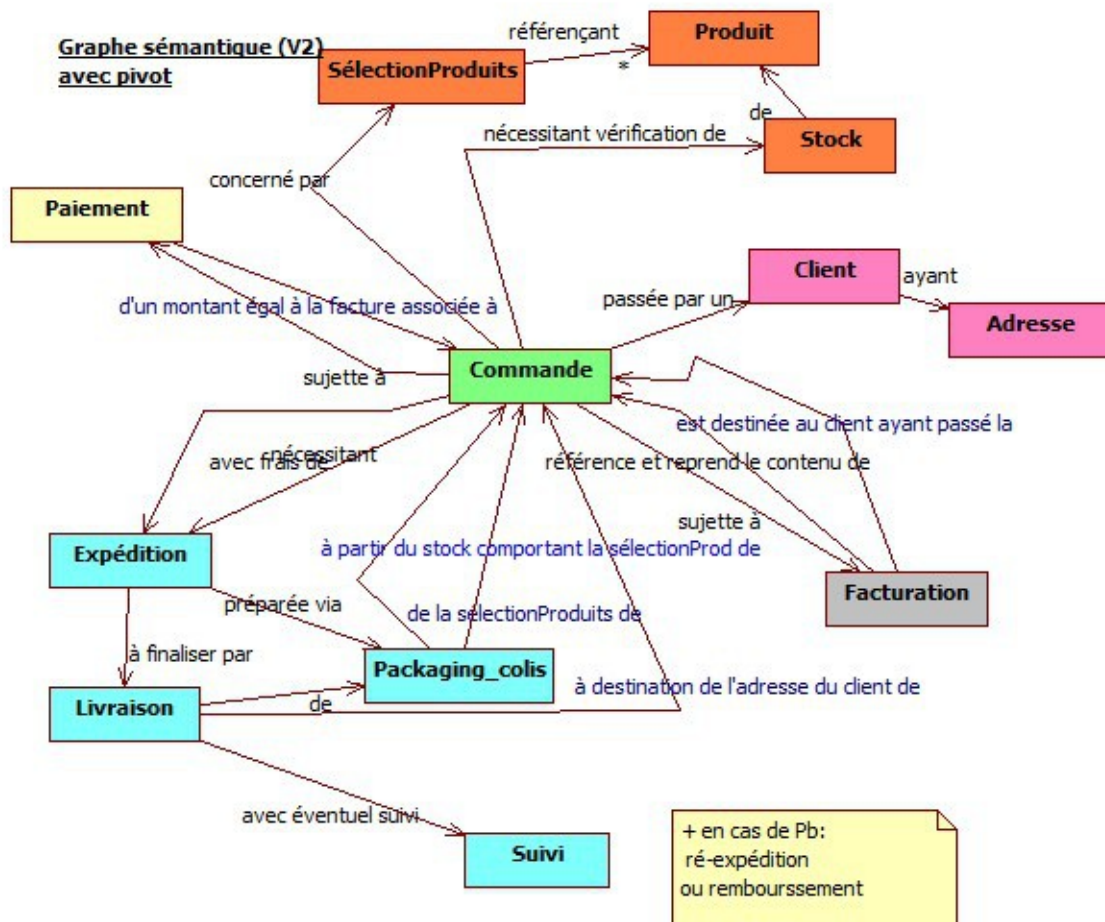
objet avec UML) doit idéalement se préoccuper d'un bon découpage en "secteurs métiers" .

Sachant qu'un **module/secteur métier** est assez souvent réalisé comme "**un paquet homogène de services métiers centrés autour d'une entité métier fondamentale**" , la réflexion à mener (au niveau du découpage) porte avant tout sur le "pourquoi ?/ quoi ?" que sur le "comment ?" .

Autrement dit , il ne faut pas toujours s'arrêter sur un découpage superficiel (de niveau "traitements" ou uniquement organisationnel) mais **il faut idéalement effectuer une analyse métier en profondeur pour faire émerger des blocs fonctionnels cohérents** .

Un "diagramme de classes UML" simplifié (ne montrant pas les attributs ni les méthodes) mais simplement les noms des entités et leurs relations/associations peut s'avérer être très efficace pour réfléchir sur les concepts et les leurs inter-dépendances . On parle assez souvent en terme de **graphe sémantique** pour désigner ce type de diagramme.

Exemple:

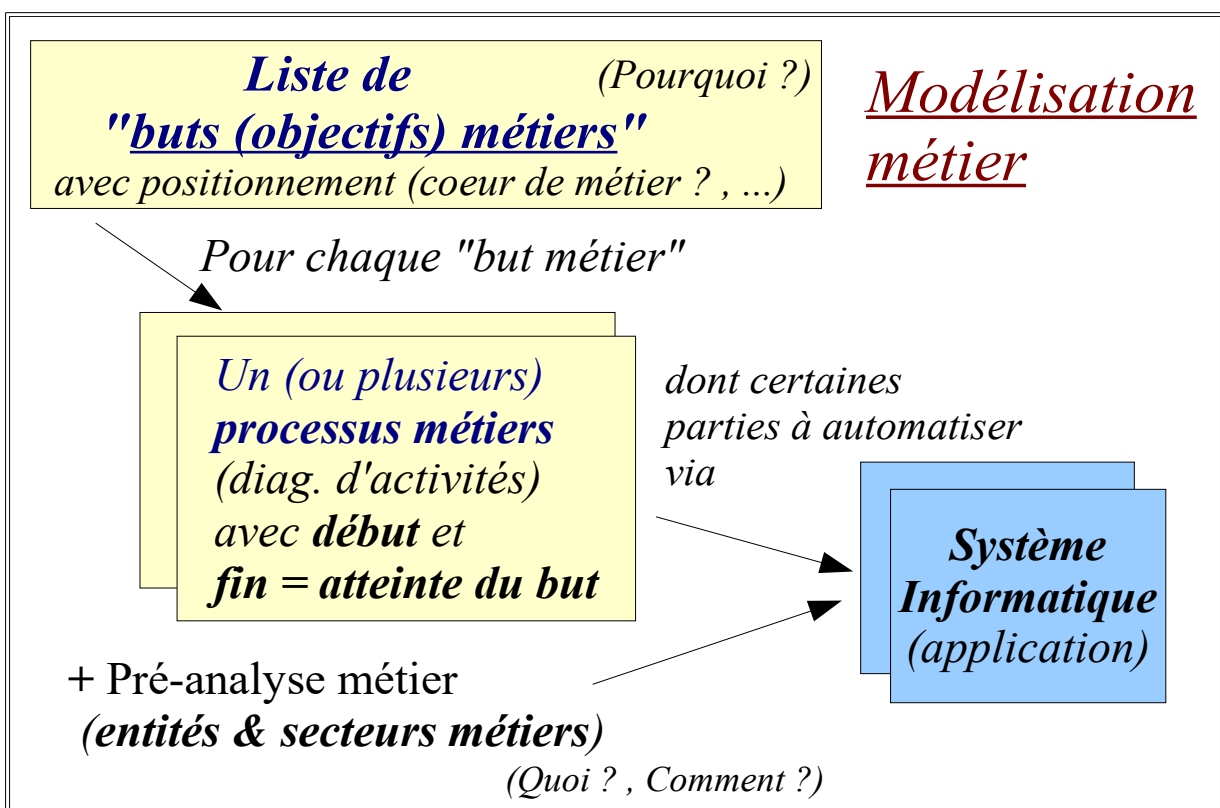


3. Business Modeling et "Business Uses Cases"

3.1. (objectifs métiers)

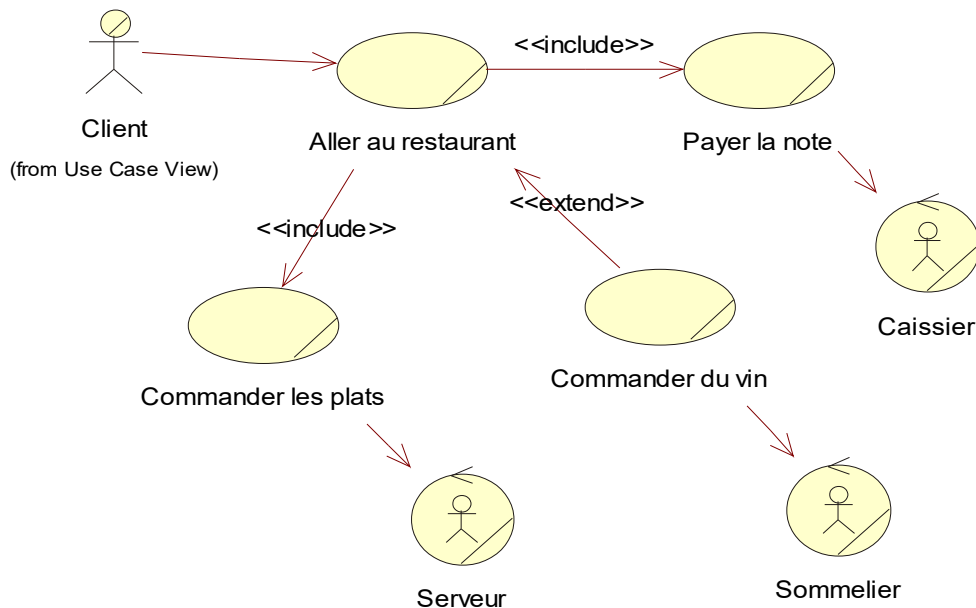
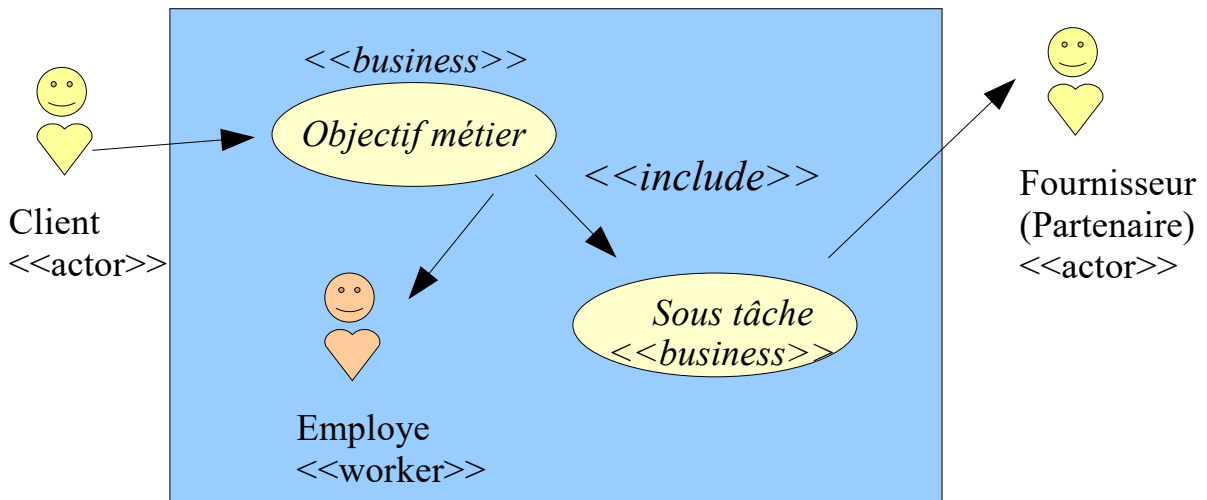
Modélisation métier (*business modeling*)

- Expression du "**pourquoi ?**" (*quelle(s) utilité(s) ? , quels **objectifs** ? , ...*)
- **Contexte très large** (entreprise + partenaires , ...) *dépassant les frontières d'un seul système informatique*
- Segmentation (*découpage/regroupement*)
--> **secteurs métiers** , packages métiers , ...
- **Processus métiers** (*diagrammes d'activités ,*)
avec activités informatisées ou non .



3.2. Business Uses Cases

Portée définie par le diagramme des "**business use case**"



Un diagramme de "*business uses cases*" ou "*cas d'utilisations métiers*" est une extension pour UML (provenant de RUP) et qui vise à **montrer les fonctionnalités d'un service ou département d'une l'entreprise** plutôt que les fonctionnalités d'un système informatique précis.

En plus de la notion d'acteur UML (implicitement externe), le stéréotype **<<worker>>** (ici associé aux caissier, sommelier et serveur) désigne **une personne interne (ex: employé)**.

4. Cas d'utilisation d'une partie du SI (soa)

Dans le cadre d'une modélisation UML , les cas d'utilisations ("métiers" ou "applicatifs") sont intelligibles dans le cadre d'un contexte bien précis (SI , sous système , application xy).

Les "processus métiers" (diagrammes d'activités UML ou bien diagramme BPMN) pourront ensuite être considérés comme :

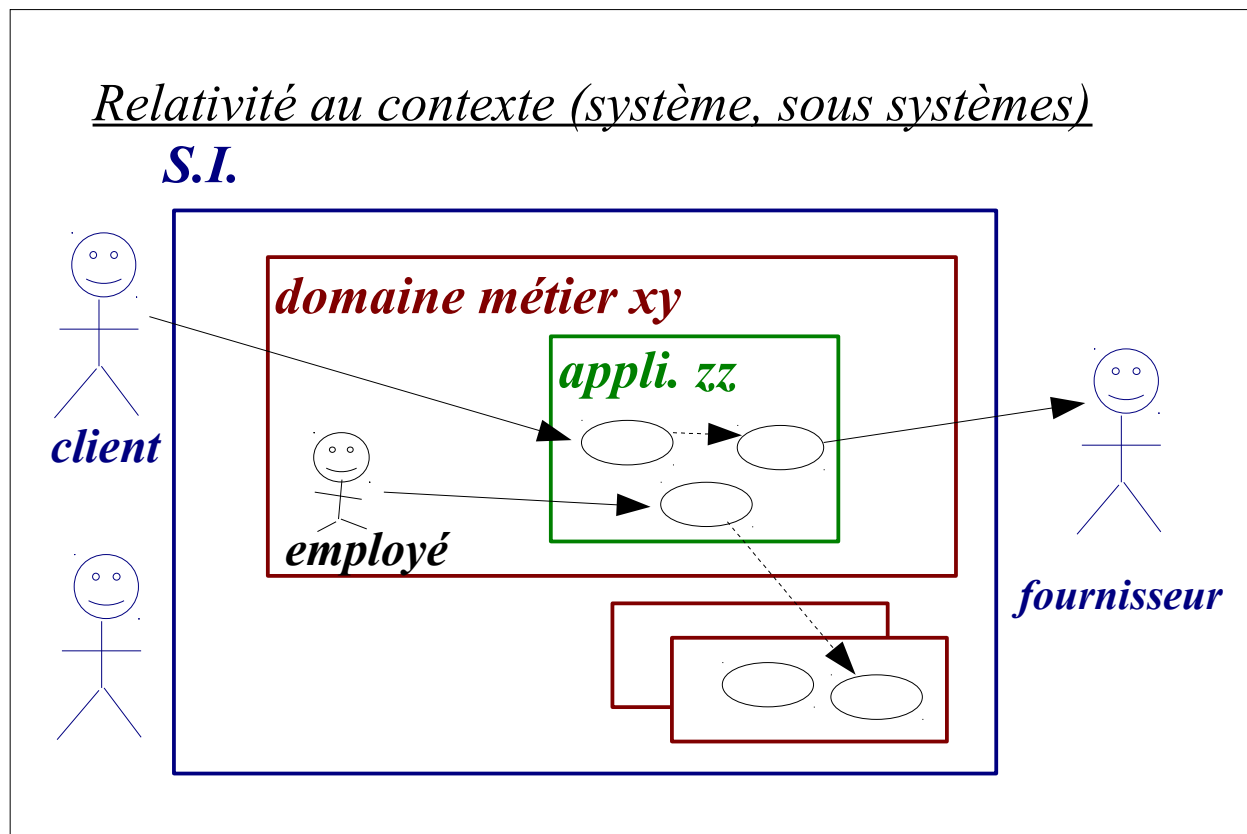
- des illustrations des scénarios des uses cases
- des macro-procédés fonctionnels permettant d'atteindre l'objectif rattaché au cas d'utilisation
- ...

Selon le cadre et la portée exacte de la modélisation SOA, on pourra être amené à :

- modéliser UC et processus au niveau macroscopique "S.I. Complet"
- modéliser UC et processus au niveau d'un domaine/secteur métier précis
- modéliser UC et processus au niveau d'une application précise

Si on se place dans un cycle itératif et incrémental , on a généralement à la fois besoin de :

- parfaire la modélisation macroscopique de l'ensemble du S.I.
- modéliser précisément (ou moins) un secteur métier du S.I.
- étudier les fonctionnalités et/ou l'intégration d'une application



Dans le schéma ci-dessus :

- l'**employé** est un **acteur extérieur** vis à vis de l'application "zz" et est un **"worker" interne** par rapport au SI et par rapport au domaine métier "xy" .
- Les "Client" et "Fournisseurs" sont des **acteurs extérieurs** pour tous les niveaux.

L'incidence du contexte par rapport à la modélisation est à évaluer au cas par cas .

Il est conseillé de rappeler le contexte courant dans les diagrammes pour éviter toute ambiguïté.

5. Modélisation des processus métiers

D'un point de vue méthodologie de modélisation, les processus métiers (diagrammes d'activités) peuvent être vus comme des illustrations des scénarios des cas d'utilisation du SI.

D'un point de vue fonctionnel , les "**processus métier**" constituent *le cœur dynamique de la modélisation métier/soa* .

Ces "processus métier" encapsulent :

- la *logique métier* principale (*stratégie / procédé*) pour atteindre un objectif métier
- l'*identification des éventuels partenaires* (fournisseurs, services externes , ...)
- une représentation des *données échangées* dans le cadre du "*workflow*" .
- la prise en compte des *alternatives* (selon les *événements* susceptibles d'intervenir)
- la prise en compte de certaines *exceptions* (avec *actions de compensation*).
-

Quelques généralités sur les diagrammes d'activités représentant les processus métier :

- il doit y avoir au minimum un **début** , une **fin** et un *chemin passant* au milieu.
- Il ne doit normalement pas y avoir d'impasse bloquante (il faut éventuellement prévoir des alternatives ou des annulations explicites)
- ...

5.1. via diagrammes d'activité UML ou via "BPMN"

Le diagramme d'activités du formalisme UML est en théorie assez approprié pour modéliser des "processus métier". En pratique, sa mise en œuvre est plus ou moins simple selon l'ergonomie de l'outil UML utilisé.

UML offre certains avantages (par rapport aux diagrammes d'activités de BPMN) :

- toute la modélisation (Uses Cases , diag de classes , activités , ...) peut se faire avec un même et seul outil. Il est quelquefois possible d'établir les relations entre les différents diagrammes (pour naviguer de l'un à l'autre).
- au cas par cas selon outil UML

==> Etudier l'**annexe "UML"** pour approfondir la syntaxe .

BPMN offre certains avantages (par rapport aux diagrammes d'activités de UML2) :

- syntaxe plus claire pour les personnes du monde "fonctionnel" (non technique)
- outils/éditeurs (pour créer les diagrammes) plus simples à utiliser
- transposition BPMN (vers BPEL ou bpmn+java) plus aisée
-

6. BPMN (essentiel)

6.1. Présentation de BPMN

BPMN signifie ***B**usiness **P**rocess **M**odel and **N**otation*

- Il s'agit d'un formalisme de modélisation spécifiquement adapté à la modélisation fine des processus métiers (sous l'angle des activités) et prévu pour être transposé en BPEL ou jBpm.
- Un diagramme **BPMN** ressemble beaucoup à un diagramme d'activité UML . Les notions exprimées sont à peu près les mêmes.

Les différences entre UML et BPMN sont les suivantes:

- la syntaxe des diagrammes d'activités UML est dérivée des diagrammes d'états UML et est plutôt orientée "technique de conception" que "processus métier"
- à l'inverse la syntaxe des diagrammes BPMN est plus homogène et plus parlante pour les personnes qui travaillent habituellement sur les processus métiers.
- UML étant très généraliste et avant tout associé à la programmation orientée objet, il n'y a pas beaucoup de générateurs de code qui utilisent un diagramme d'activité UML
- à l'inverse quelques éditeurs BPMN sont associés à un générateur de code BPEL et la version 2 de Bpmn peut être accompagnée d'extensions "java / jbpm ou activiti)" pour le rendre exécutable. Les éléments fin d'une modélisation BPMN ont été pensés dans ce sens.

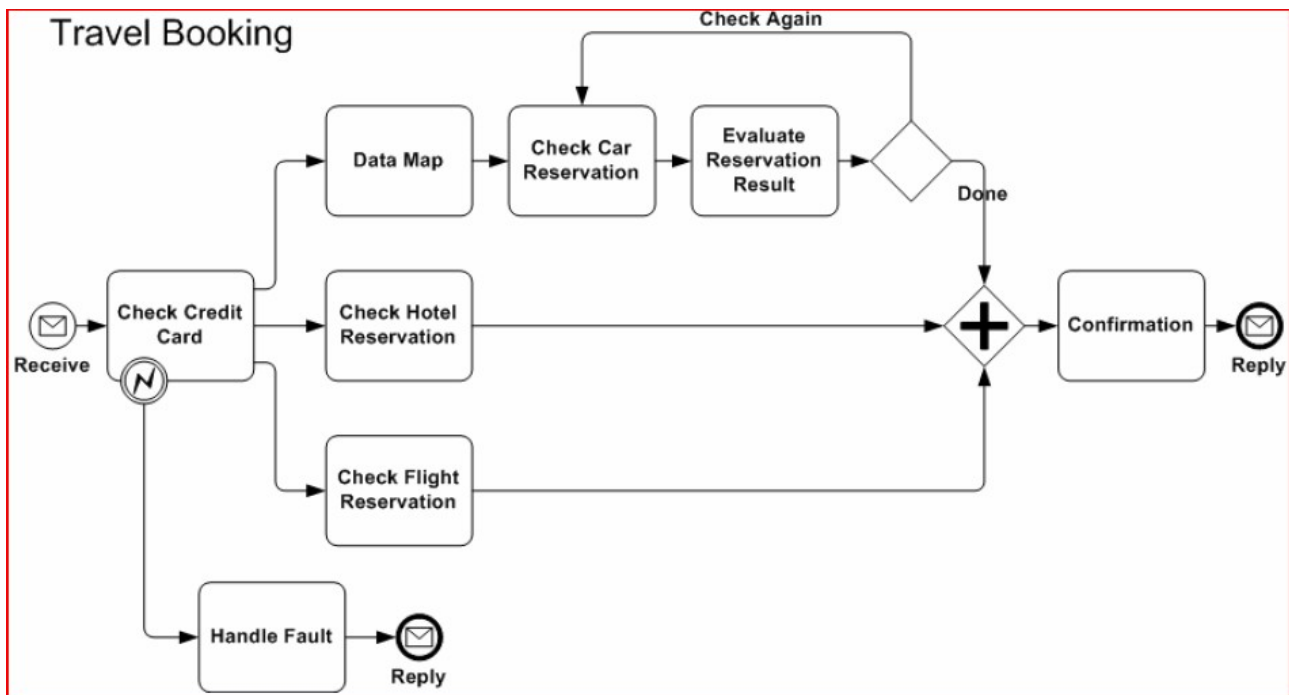
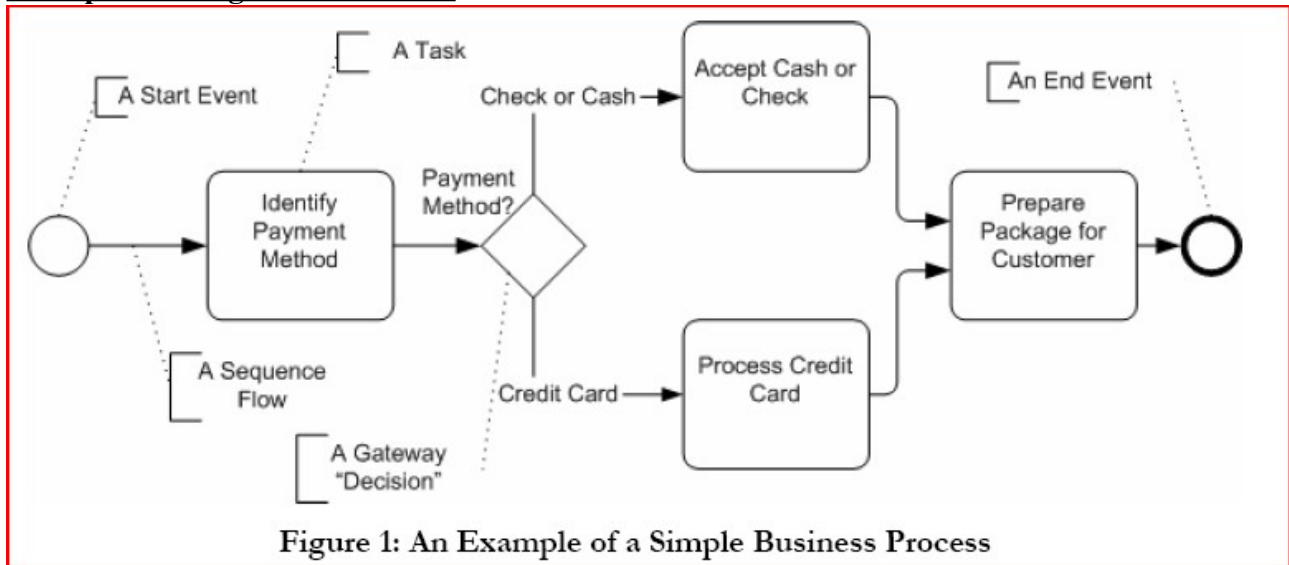
Outils concrets pour la modélisation BPMN :

- ***Intalio BPMN*** .
- ***Bizagi Process Modeler*** (version gratuite)
- ***Editeur BPMN2 intégré à l'IDE eclipse (drools / jbpm5)***.
- ***Yaoqiang bpmn(2) editor*** (très bien : versions gratuites et payantes).

NB : Certains anciens outils BPMN 1.x (tels que **Bizagi**) étaient capables d'effectuer des imports/exports au format **XPDL** . **XPDL** signifie "***X**ml **P**rocess **D**efinition **L**anguage*" : c'est une sorte de sérialisation Xml de BPMN.

Des outils BPMN récents (supportant **BPMN 2**) peuvent quelquefois directement utiliser xml comme format natif (ex : **Jboss drools/jbpm5**, **activiti bpmn** , **yaoqiang bpmn editor**) .

Exemples de diagrammes BPMN:



6.2. Principales notations BPMN :

Flow objects :

Event (événement) :

- **Start** (début)
- **Intermediate** (intermédiaire)
- **End** (fin)






Activity (activité au sens large) :



- **Task** (tâche)
- **Sub-Process** (sous processus)



Gateway (décision / contrôle du flow)




Le losange (gateway) peut comporter alternativement plusieurs types/marqueurs :

<p>Exclusif (selon état des variables/données internes au processus)</p>  <p>ou bien</p>  <p>(selon outil)</p> <p>parallel</p> 	<p>Le mode exclusif désigne une seule route de sortie possible (selon condition/décision)</p> <p>Les conditions sont portées par les "sequenceFlow" de sorties .</p> <p>parallèlement (=fork <i>au début</i>) , aucune condition ne sera analysée.</p> <p>(sémantique "= join" <i>à la fin</i>) → suite que si toutes les branches concurrentes/parallèles sont finies/terminées .</p> <p>Le mode inclusif désigne plusieurs (au moins 2) routes de sortie possibles (par exemple: une</p>
---	--


<p>inclusif</p> 	<p>branche principale/obligatoire et d'autres branches facultatives) qui se rejoignent généralement par la suite .</p> <p>Comportement au début (fork selon conditions portées par les "sequenceFlow")</p> <p>Comportement à la fin (join en attendant que la fin des exécutions commencées)</p>
<p>eventBased</p> 	<p>(exclusivement) selon (premier) événement qui sera ultérieurement reçu.</p> <p>Les événements qui suivent un "eventBasedGateway" ne peuvent être que de type "intermediateCatchEvent" (souvent Timer ou Message)</p>

NB: Lorsque le processus effectue plusieurs **activités concurrentes (en //)** , la technologie interprétant Bpmn gère alors plusieurs "**jeton d'exécutions**" pour suivre et synchroniser les différents **états d'avancement** .

Connecting objects (connexions):

- **sequence flow** (séquence ordonnée) 
- **message flow** (entre 2 participants) 
- **association** (entre tâche et données) 

Swimlane (couloir/partition d'activités)

- **pool** (*un par participant : Processus ou Partenaire ou ...*) 
- **lane** (*sous partition*) 

Data Objects :

- **Data** (*document xml ,objet en mémoire , ...*)
- **Data Store** (*extraction/persistance en base,...*)

6.3. Structure BPMN2

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions xmlns="http://www.omg.org/spec/BPMN/20100524/MODEL"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema" xmlns:activiti="http://activiti.org/bpmn"
xmlns:bpmndi="http://www.omg.org/spec/BPMN/20100524/DI"
xmlns:omgdc="http://www.omg.org/spec/DD/20100524/DC"
xmlns:omgdi="http://www.omg.org/spec/DD/20100524/DI"
typeLanguage="http://www.w3.org/2001/XMLSchema"
expressionLanguage="http://www.w3.org/1999/XPath"
targetNamespace="http://www.activiti.org/test">
  <message id="asyncResponseEvent" name="asyncResponseEvent"></message>
  <!-- logique (importante) du processus : -->
  <process id="myProcess" name="My Process" isExecutable="true">
    <startEvent id="startevent1" name="Start"></startEvent>
    <scriptTask id="scripttask1" name="SendAsyncRequest" activiti:async="true"
      scriptFormat="javascript" activiti:autoStoreVariables="false"><script>.... </script>
    </scriptTask>
    <sequenceFlow id="flow1" sourceRef="startevent1" targetRef="scripttask1"></sequenceFlow>
    <eventBasedGateway id="eventgateway1" name="Event Gateway"></eventBasedGateway>
    <intermediateCatchEvent id="timerintermediatecatchevent1" name="TimerCatchEvent">
      <timerEventDefinition>
        <timeDuration>PT1M</timeDuration>
      </timerEventDefinition>
    </intermediateCatchEvent>
    <intermediateCatchEvent id="...." name="MessageCatchEvent">
      <messageEventDefinition messageRef="asyncResponseEvent"></messageEventDefinition>
    </intermediateCatchEvent>

    <endEvent id="endevent1" name="End"> </endEvent>
  </process>

  <!-- coordonnées graphiques (non importantes) du processus : -->
  <bpmndi:BPMNDiagram id="BPMNDiagram_myProcess">
    <bpmndi:BPMNPlane bpmnElement="myProcess" id="BPMNPlane_MyProcess">
      <bpmndi:BPMNShape bpmnElement="startevent1" id="BPMNShape_startevent1">
        <omgdc:Bounds height="35.0" width="35.0" x="60.0" y="101.0"></omgdc:Bounds>
      </bpmndi:BPMNShape>
      ...
      <bpmndi:BPMNEdge bpmnElement="flow1" id="BPMNEdge_flow1">
        <omgdi:waypoint x="95.0" y="118.0"></omgdi:waypoint>
        <omgdi:waypoint x="140.0" y="151.0"></omgdi:waypoint>
      </bpmndi:BPMNEdge>
    </bpmndi:BPMNPlane>
  </bpmndi:BPMNDiagram>
</definitions>

```


6.4. Types précis d'événements BPMN2 :

Catching event (en réception):

Le processus est bloqué tant qu'il n'a pas reçu l'événement attendu.

L'icône interne est "non colorié" (laissé à blanc)

Throwing event (en envoi):

Le processus déclenche (envoi/soulève) un événement

L'icône interne est "colorié" (rempli de noir)

Définitions d'événements (Bpmn) :




La "définition d'un événement bpmn" englobe :










- une sémantique (fonctionnelle) de l'événement (nom logique de l'événement , description , ...)
- un type d'événement (ex : Timer , Message , Signal , ...)
- d'éventuels paramétrages précis (ex : "timeDuration" pour un Timer)



Sémantiques des types (de définition) d'événements de BPMN :

Signal	Signal (avec un nom logique) qui peut être simultanément traité/reçu par plusieurs processus . Un signal envoyé par un processus est potentiellement diffusé vers plusieurs processus destinataires par la technologie prenant en charge l'exécution BPMN . Un signal n'a pas de paramètres .
Message	Message avec un " nom logique " et un " payload " (paquets de paramètres / arguments) . Un message "bpmn" est envoyé vers un seul destinataire .
Error	Exception métier / business (un peut comme message négatif)
Timer	À date ou heure fixe ou bien après période écoulée ou bien de façon cyclique .
Rule	règle métier vérifiée (extension pour technologie avec moteur de règles telle que "drools" ou ...)
...	...

Principaux types d'événements (bpmn) :

Start	None 	Selon contexte de déclenchement externe
	Signal 	A la réception d'un certain signal <i>(un déclenchement de signal peut quelquefois servir à démarrer plusieurs processus au même moment)</i>
	Message 	A la réception d'un message (avec d'éventuels paramètres/arguments) <i>(Un processus peut éventuellement avoir plusieurs "MessageStartEvent" si</i>

	<p>Timer </p> <p>Error </p> <p>...</p>	<p>plusieurs façons de démarrer).</p> <p>A une heure précise (éventuellement de manière périodique) ou ...</p> <p>Pour un démarrage de sous-processus en cas d'exception métier .</p>
Intermediate Catching	<p>Signal </p> <p>Message </p> <p>Timer </p> <p>...</p>	<p>Attente d'une réception de signal</p> <p>Attente d'une réception de message</p> <p>Attente d'une période écoulée ou ...</p>
Intermediate Throwing	<p>None </p> <p>Signal </p> <p>Compensation </p> <p>...</p>	<p>Juste pour indiquer (localement) qu'une tâche est finie (utile si techniquement associé à "listener d'événement" activi ou jbpn → stats (KPI , BAM))</p> <p>déclenchement/envoi d'un signal</p> <p>pour déclencher une demande de compensation/annulation (à rattraper via "compensation boundary catching event " ailleurs)</p>
End	<p>None </p>	<p>sans "résultat" en retour</p>

<p>Error </p>	<p>fin (souvent au sein d'un sous-processus) en envoyant un événement d'erreur devant être rattrapé par un "intermediate boundary error event" du niveau englobant</p>
<p>Cancel </p>	<p>Retourné (en mode transactionnel) par un sous-processus de façon à ce que le processus parent puisse le rattraper via un "cancel boundary event" lui même associé à un déclenchement de compensations.</p>
<p>... selon outils/technologies "Message" , "Signal" , "..." au niveau du "endEvent" ou bien au niveau d'une "Task" préalable .</p>	

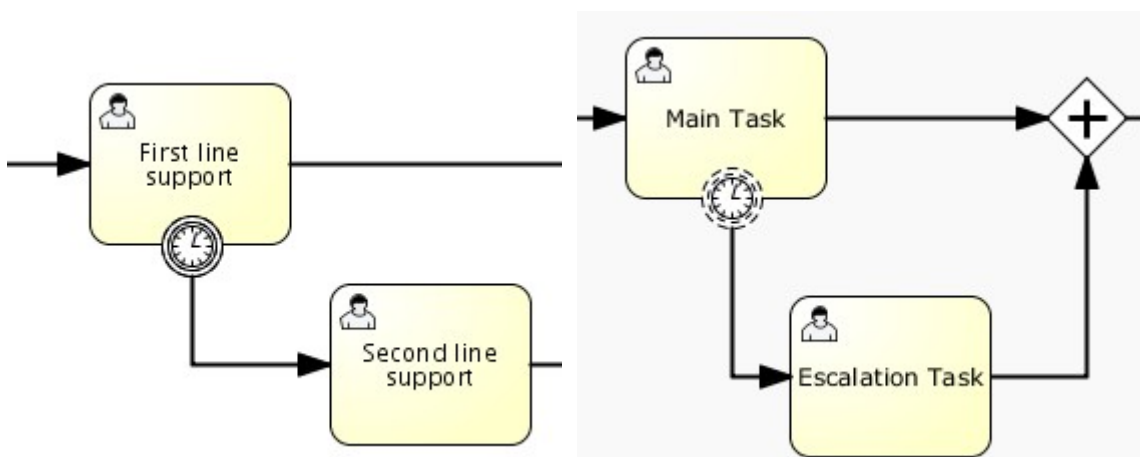
Boundary Events :

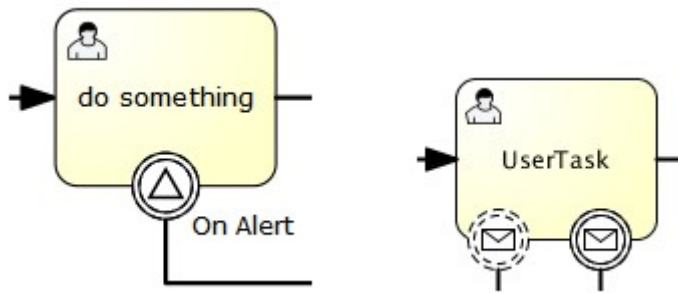
Les "**Boundary events**" sont une sorte de "**catching events**" qui sont **attachés à une activité**.

Lorsque l'activité est en cours d'exécution, l'événement (en écoute) est potentiel déclenché .

Lorsque l'événement est attrapé , l'activité est interrompue et la séquence qui suit l'événement déclenché est alors exécutée .

Nouveauté de "BPMN2" : Lorsque les 2 cercles périphériques d'un "boundary event" sont en **pointillés** , l'activité n'est pas interrompue et on ne fait qu'effectuer des **opérations supplémentaires ("escalation")**. En xml , `cancelActivity="false"` si pointillés.











Tous les "boundary events" sont définis de la même manière :

```
<boundaryEvent id="myBoundaryEvent" attachedToRef="theActivity">
  <XXXEventDefinition/>
</boundaryEvent>
```

Type de "Bondary Events" :

Timer 	Au bout du "timeout" , activité interrompue (ou pas) et suite après "evt"
Error 	Erreur remontée par l'activité (sous-processus , call-activity , ...)
Signal 	Signal reçu (émis depuis un endroit quelconque) au moment de l'exécution de l'activité
Message 	Message reçu au moment de l'exécution de l'activité (en mode interruption ou pas)
Cancel 	Issue potentielle d'une activité (en mode "transactionnal")
Compensation 	Pour attacher un "gestionnaire de compensation" à une activité
...	

Détails du certains types (de définition) d'événements :

Paramétrage d'un **timerEventDefinition** :

timeDate (ex : 2011-03-11T12:13:14 YYYY-MM-DD T(ime) hh:mm:ss)

ou bien







timeDuration (ex : P10D pour une Période de "10 Day")

ou bien

timeCycle (ex : R3/PT10H/... Repeating 3 times / ...)

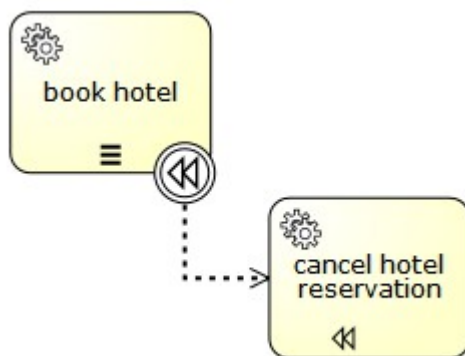
Les valeurs des paramétrages correspondent à la norme **ISO 8601**

6.5. Principaux types de Tâches "BPMN"

ServiceTask 	Tâche automatique (via appel de service web ou via classe java ou ...)
ScriptTask 	Tache automatique (via script "javascript" ou "groovy" ou ...)
UserTask 	Tâche effectué par un utilisateur (souvent un employé appartenant à un certain groupe) avec généralement une console (web ou ...) : Les actions effectuées par l'utilisateur dépendent des valeurs du processus (selon tâches précédentes) et les valeurs saisies par l'utilisateur seront récupérées par le processus et influenceront les tâches ultérieures. Asynchronisme automatique (attente tant que tâche pas complètement effectuée)
ReceiveTask 	Réception d'un message (attente bloquante) Alternative fréquente : "intermediate Message Catching event"
ManualTask 	Référence à une tâche entièrement manuelle (sans interaction avec le processus), pas gérée (ignorée) par le moteur d'exécution bpmn (ex : activiti).
BusinssRuleTask 	Applications de règles métiers (souvent en s'appuyant sur "drools")
Autres (spécifiques selon technologies / extensions)	EmailTask , WsTask , EsbXyTask , ShellTask , ...

6.6. Gestionnaire de compensation :

Activité spéciale prévue pour compenser (via action inverse ou palliative) une activité (déjà effectuée) suite à un problème détecté tardivement .



6.7. Sous-processus BPMN

Un sous processus est une activité (non atomique) qui comporte d'autres activités , "gateway" , événements , Le sous-processus est lui même une partie d'un processus plus grand.

Un "sous-processus" BPMN ordinaire est entièrement défini dans le processus parent et a donc une sémantique de "embedded sub-process".

Deux grands intérêts :

- **modélisation hiérarchique** (avec +/- pour visualiser/cacher les détails)
- **sous-processus = nouvelle portée pour certains événements** (certains événements soulevés par les éléments du sous-processus sont rattrapés via des **"boundary event"** du sous-processus) .

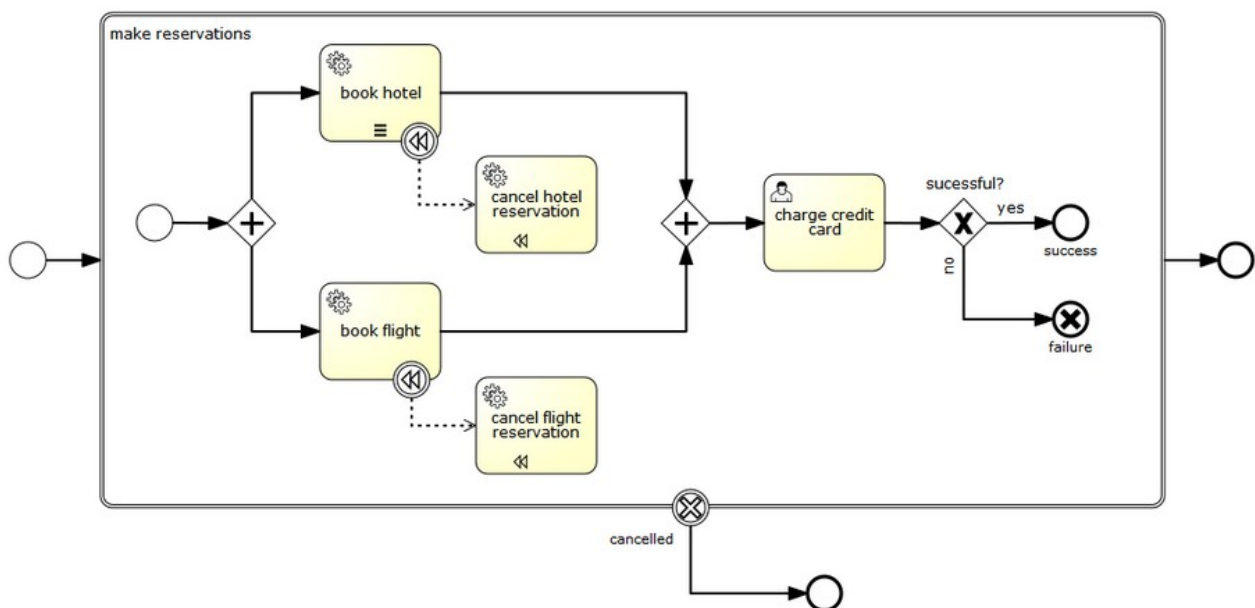
Contraintes (BPMN1 et certains moteurs BPMN2) :

- Un sous-processus doit comporter un seul sous "(none) startEvent"
- Un sous-processus doit comporter au moins un "endEvent"

Structure Xml :

```
<subProcess id="subProcess">
  <startEvent id="subProcessStart" />
  ... other Sub-Process elements ...
  <endEvent id="subProcessEnd" />
</subProcess>
```

Transactional (sub-process) :



délimité par <transaction> ... </transaction> en xml , double contour en notation graphique.

Un processus transactionnel comporte tout un tas de sous-activités qui seront toutes "réussies" ou toutes "annulées / éventuellement compensées en interne" .

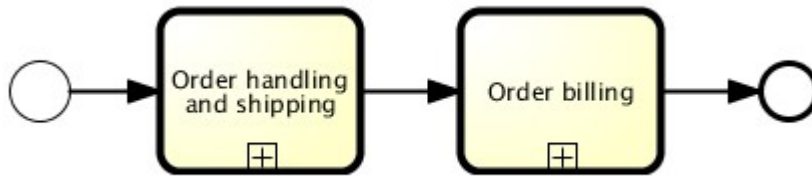
L'issue globale de la transaction sera "success" (suite par défaut) ou bien "cancel" (vu comme un "cancel boundary event") . Dans certains cas rares , une erreur interne technique (hazard) sera gérée comme un troisième type d'issue (sans compensation mais souvent "log à étudier/gérer") .

Attention : "transactional" (au sens transaction longue) est assez récent (et pas encore parfaitement interprété/géré partout) .

Call activity (au sens "call external subProcess") :

Le (sous) processus appelé est alors défini (de façon externe) dans un autre fichier "bpmn" ce qui permet une plus grande ré-utilisabilité .

Le code du (sous-)processus appelé est censé être chargé (ou pré-chargé) dynamiquement .



Certains (rares) outils permettent une visualisation développée (en ouvrant/analysant le sous fichier ".bpmn")

```
<callActivity id="callCheckCreditProcess" name="Check credit"
calledElement="checkCreditProcess" />
```

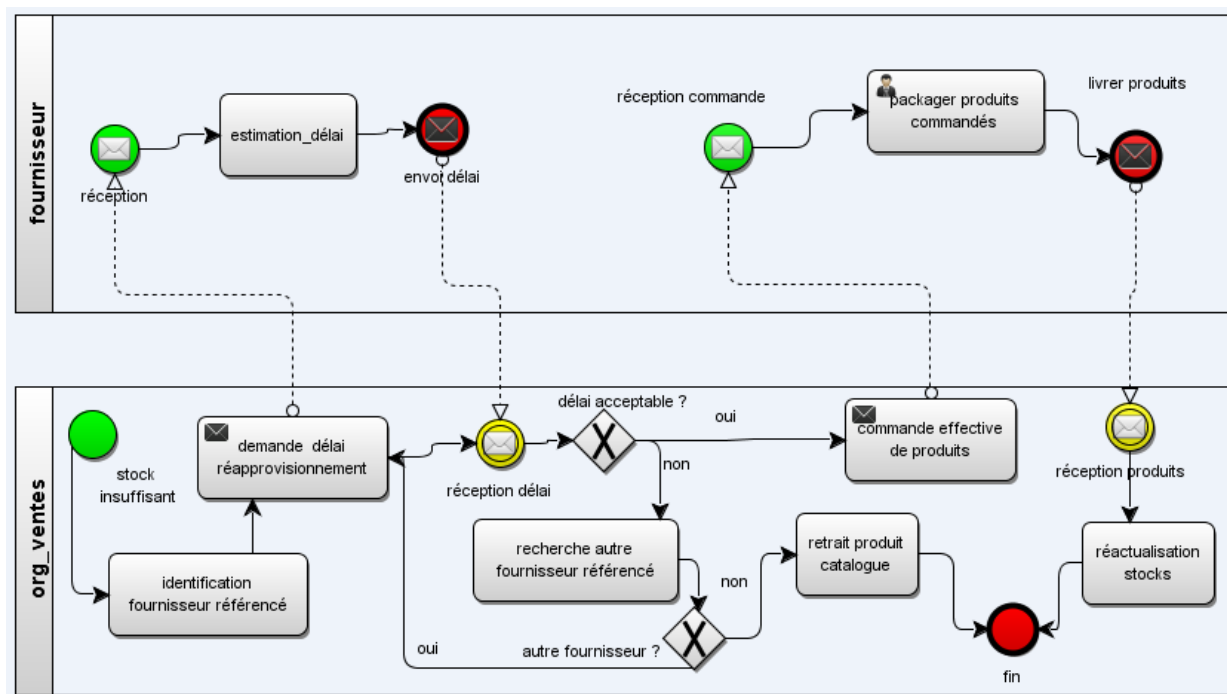
Le (sous-) processus appelé est **identifié par son nom logique** (identifiant interne au fichier ".bpmn") .

Possibilité de passer des contenus de variables lors de l'appel d'un sous-processus (via certaines extensions telles que "activiti" , ...) .

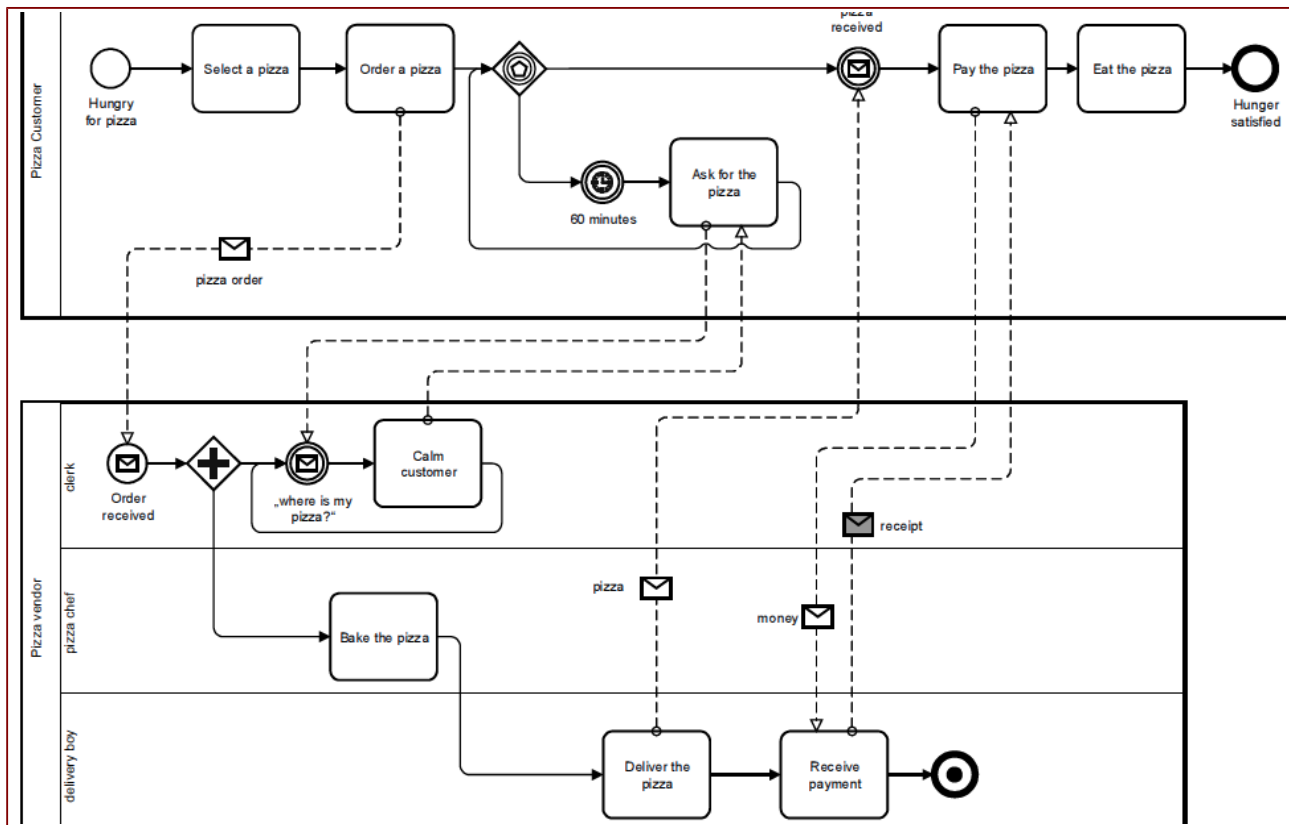
6.8. Exemples de diagrammes BPMN

Exemple "approvisionnement fournisseurs" (édité avec "yaoqiang-bpmn-editor") :

Exemple de diagramme BPMN



Exemple BPMN "pizza" (tiré de la norme officielle de l'OMG) :



VI - Problématiques SOA (intégrité , MDM, ...)

1. Modélisation et problématiques SOA

Eléments importants d'une bonne modélisation SOA:

- Services (à avant tout voir de manière fonctionnelle)
- Structure de données échangées (entrées,sorties,exceptions/faults) .
Le vocabulaire "ValueObject" / "Data Transfert Object / View" est ici assez approprié.
- Catégorie/secteur métier englobant tel ou tel service.
- Dépendances fonctionnelles entre les services (et avec structures de données produites/consommées indirectement concernées).
- Processus métiers avec logiques/règles métiers.
- Quelques éléments techniques fondamentaux (volume, perf , synchrone ou asynchrone,)
-

1.1. Orchestration ou bien chorégraphie ?

- Le terme d'**orchestration de service** correspond à une *logique d'interaction de type "maître / subordonnés (promptement coopératifs ou pas)"*. **Un service de haut niveau pilote des services de bas niveaux** en invoquant des opérations dans un ordre bien établi et/ou selon une logique conditionnelle .
- Le terme de **chorégraphie** (SOA/services) correspond en plus à une **interaction plus étroite/collaborative** ou chaque membre **agit de façon plus spontanée (idéalement pas trop désynchronisée)**.

==> Techniquement parlant, dans la plupart des cas , l'orchestration suffit (c'est plus simple à mettre en œuvre et plus fiable). La plupart des technologies existantes (bpel , ...) s'occupent de l'orchestration (et pas de la chorégraphie).

==> Fonctionnellement, on a souvent besoin de modéliser des chorégraphies (avec plusieurs couloirs) pour bien montrer une collaboration de principe entre plusieurs parties prenantes (ex : "client" , "logistique" , "fournisseur") qui partagent un même but (ou sous but) et qui sont quelquefois mutuellement engagées au sein d'une logique de partenariat (quelquefois contractuelle).

1.2. UML et/ou BPMN ?

UML est assez universel et très approprié pour la modélisation orientée objet (des services et des structures de données utilisées/échangées par les services) .

BPMN (*Business Process Modeling Notation*) est très approprié pour modéliser les processus métiers (mieux que les diagrammes d'activités d'UML).

UML et BPMN se complémentent plutôt bien .

1.3. en Mode SaaS (Software as a Service) ?

SaaS (*Software as a Service*) est **étroitement lié** à l'architecture SOA .

Au lieu que chaque entreprise développe elle même (et sans assez de moyen) ses propres services, il vaut mieux louer des services offerts par des éditeurs expérimentés dans tel ou tel domaine fonctionnel.

Le mode "Saas" apporte quelques nouvelles problématiques :

- compte client / authentications/habilitations sécurisées
- éventuel partage (partiel) de données "publiques" entre plusieurs clients
- base de données mutualisée (avec séparation claire de la propriété des différents "clients") ou
-

1.4. Performances et sécurité (en mode SOA)

Un des problèmes très souvent rencontrés dans une architecture SOA est un **mauvais temps de réponse** (mauvaises performances).

Ceci est du au fait que chaque **intermédiaire** (*technique et/ou fonctionnel*) introduit potentiellement (selon l'architecture exacte) de nouveaux besoins en termes de :

- transfert de données (multiples traversées du réseaux ?)
- traitements CPU (transformations de formats de données / de protocoles)
- reformulations/réinterprétations , encodage/décodage (ex : java <-->xml)
- éventuels cryptages/décryptages (selon sécurité)

Il est clair que l'on ne peut pas avoir le beurre et l'argent du beurre. Autrement dit , la flexibilité/agilité a un prix.

Cependant , toute mise en œuvre d'une architecture SOA devrait idéalement être accompagnée d'une étude annexe d'optimisation des performances (au cas par cas) .

Le second problème récurrent de SOA est la **sécurité** :

par défaut un service web peut être appelé par n'importe quel client qui connaît son URL .

...

2. Gouvernance des données

2.1. Problématique d'échanges/communications

SOA mène idéalement à une architecture décentralisé/départementalisée où différents systèmes collaborent via un couplage faible (pas de liens très étroits, relative indépendance des systèmes).

Bien que certains éléments techniques de l'infrastructure SOA (transformations des formats de données via xslt ou ... , conversion de protocoles ,) puissent aider à rendre techniquement indépendants les différents systèmes devant communiquer entre eux, il reste toujours des dépendances assez fortes d'un point de vue fonctionnel.

Par exemple, on peut transformer un "client" en "customer", une chaîne de caractères "adresse" en éléments décomposés (rue, code_postal , ville) . On peut aussi récupérer un user_name manquant via un service intermédiaire d'extraction du user_name en fonction d'un user_id mais on ne peut pas transformer un numéro_de_client et numéro_de_produit.

Autre point à prendre en considération : qui dépend de qui (d'un point de vue système)? Quels sont les systèmes "cœur de métier" dont on maîtrise l'évolution. Quels sont les systèmes annexes (bien souvent achetés) et pourtant fondamentaux/indispensables dont on ne maîtrise pas tout à fait l'évolution (en fonction des choix et de la pérennité des éditeurs).

Ce vaste sujet (lié à l'urbanisation) est très important au niveau de la modélisation SOA.

2.2. Format de données pivots et gouvernance des données

Une **bonne approche pragmatique** souvent appelée "**gouvernance des données**" consiste à garantir une bonne agilité du SI en appliquant les dispositions suivantes:

- *applications périphériques complètement indépendantes*
- **éléments collaboratifs principaux (cœur de métier) avec données "pivot" dont la structure est en partie déterminée par gouvernance (prise en compte de multiples besoins).**

Soit appli1

---> tx1(requête_format1)

-----> requête_format_pivot

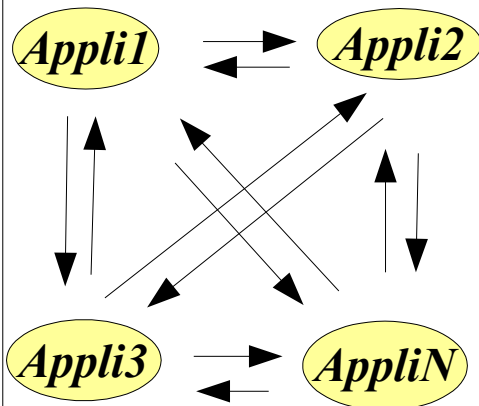
(format normalisé par organisation)

--->tx2(vers_format_appli2)

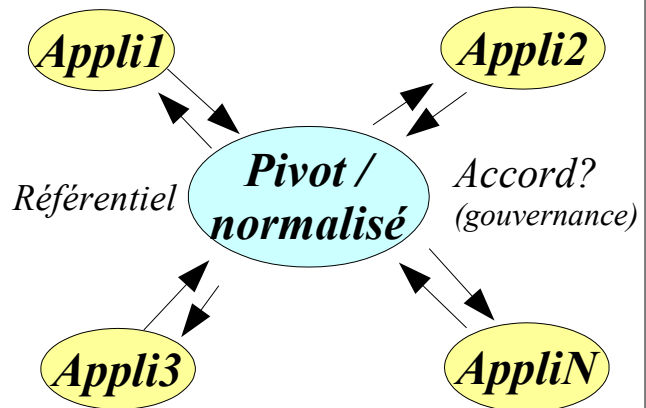
---> requête_format2 reçu par appli2

où tx1(...) et tx2(...) sont deux **transformations** (éventuellement) nécessaires.

SOA : format normalisé / pivot

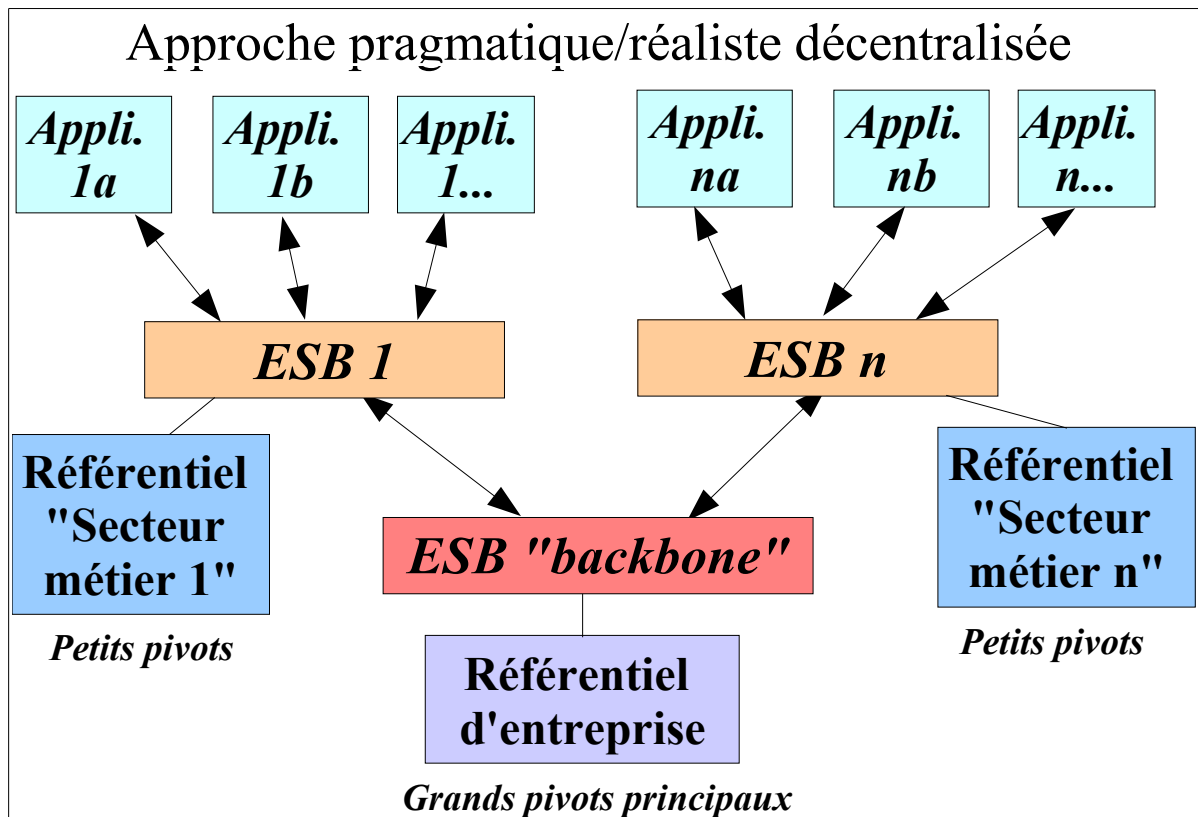


Si transformations
des formats de données
au cas par cas :
--> besoins en $O(n^2)$!

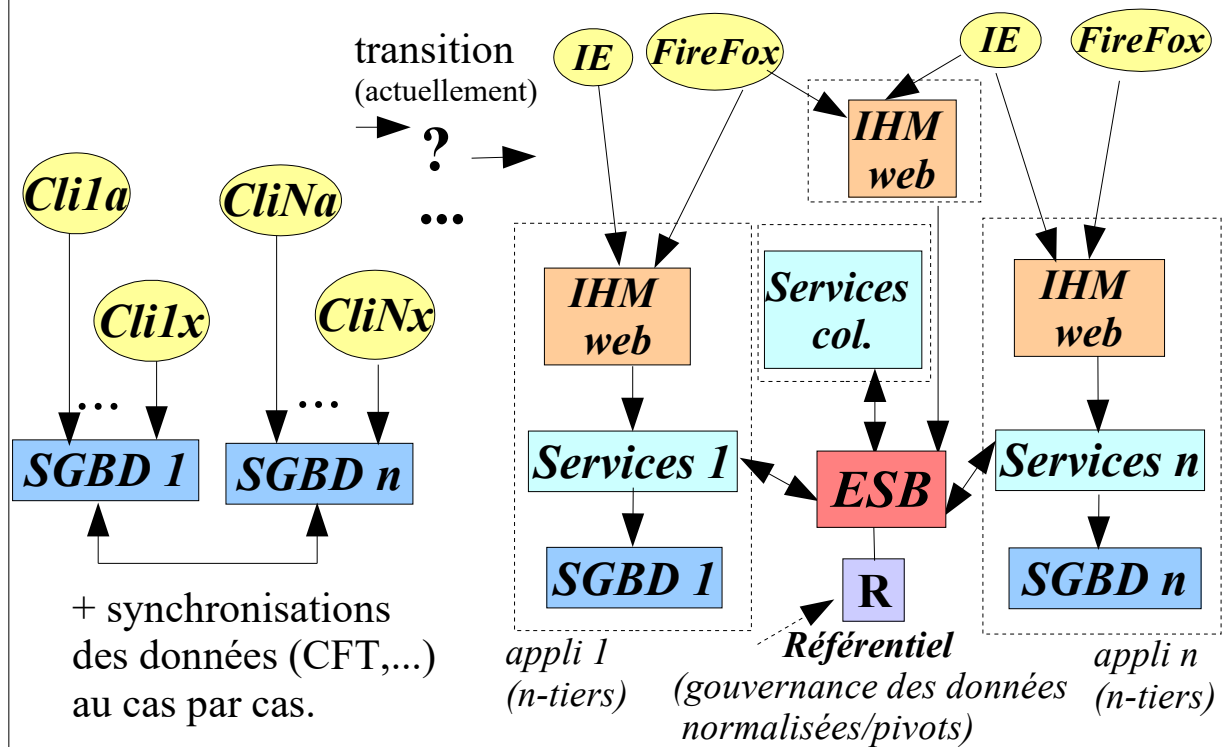


Si transformations avec format
de données intermédiaire
normalisé (pivot) :
--> besoins en $O(n)$!

Formats "pivot" et "sous pivot"
dans architecture idéalement décentralisée

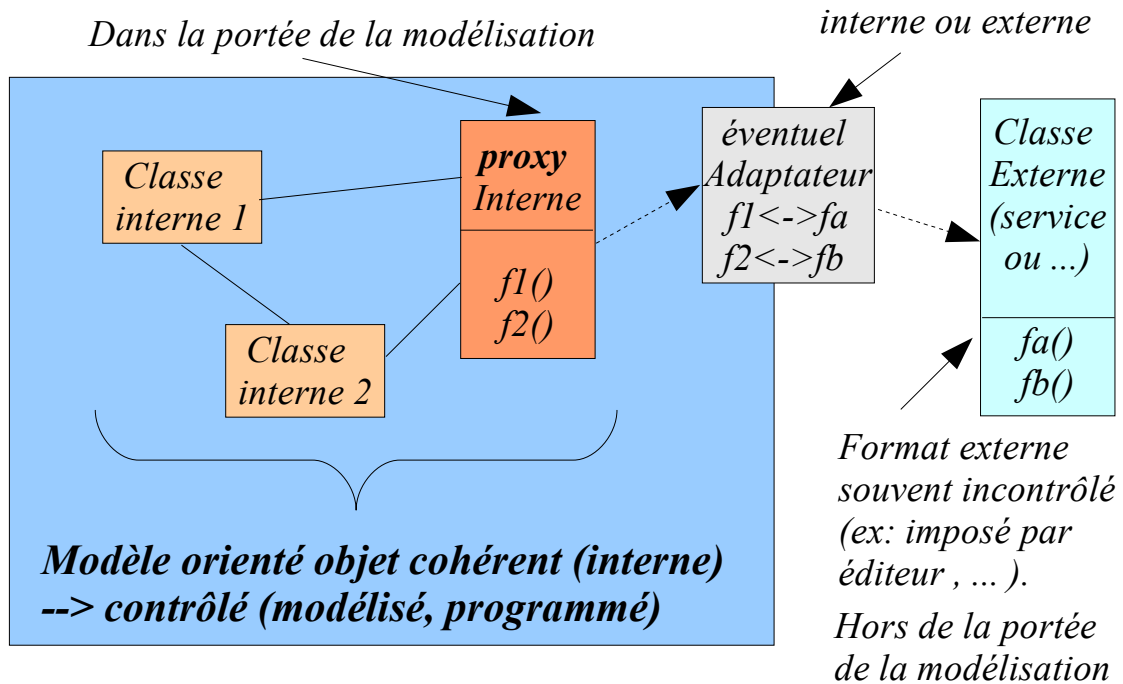


du client/serveur vers SOA



2.3. Proxy métier/fonctionnel

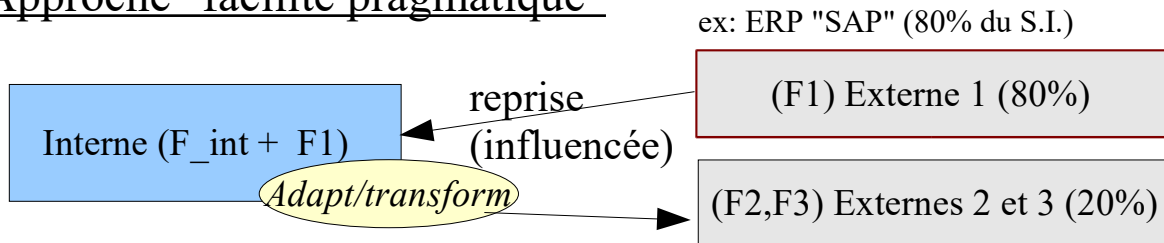
Notion fondamentale de "proxy métier"



2.4. Approches "conformisme" et "gouvernance"

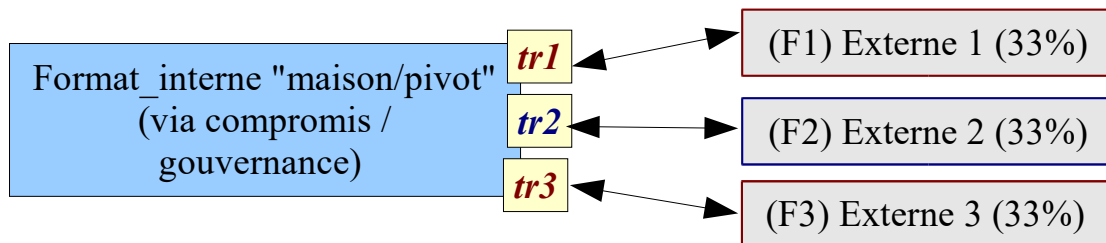
Choix du format interne (*) quelquefois appelé "format pivot" en SOA

Approche "facilité pragmatique"



--> Rapide , efficace (mais on est devenu très dépendant du format F1)

Approche "gouvernance des données"



--> A définir , maintenir (mais on y gagne en indépendance)

L'approche "gouvernance des données" (théoriquement conseillée) consiste à **définir certains formats fonctionnels (structure de données) en confrontant les besoins (en communications / échanges) de différentes parties du SI** (applications , sous systèmes , ...) .

Pour cela , des **réunions** entre "parties prenantes" sont souvent nécessaires. Pour éviter les réunions inutiles où personne ne se met d'accord à la fin, on pourra s'organiser avec une matrice "RACI" .

R : Responsable (**r**esponsable , **r**éalisateur)

A : Accoutable , approuver (**a**pprobateur avec droit de vote)

C : Consulted (**c**onsulté – donne son avis)

I : Informed (**i**nformé)

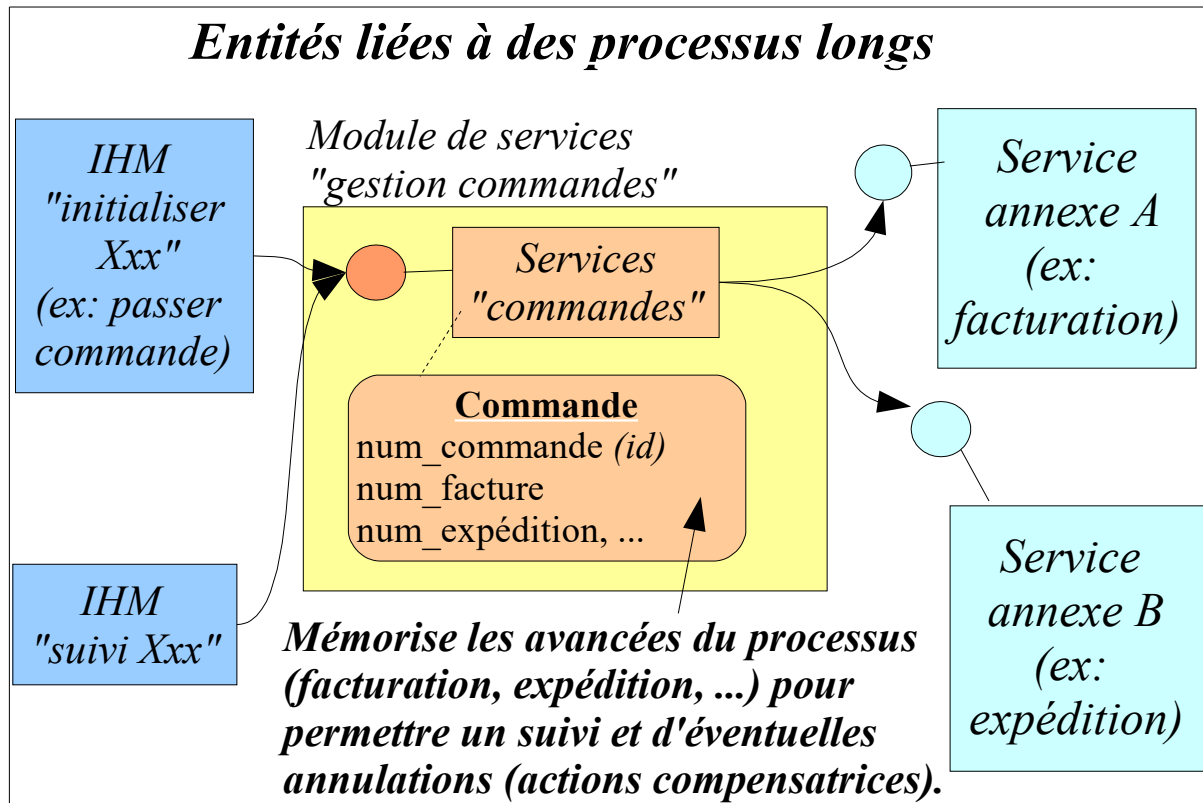
Task Description	Sponsor	Business Owner	Business Program Mgr	Process Manager
Identify missing or incomplete policies		R	A	R
Establish Policies as necessary and ensure adoption globally		A	R	R
Completion of necessary Policies		R	A	R
Document Policies as appropriate		R	R	A
Approve Policies	A	C	I	I

3. Transactions longues et compensations

De nombreux échanges SOA sont effectués en mode asynchrone lorsque certains traitements ou sous processus (ex: livraison) sont longs.

Les éventuelles transactions associées, longues elles aussi, ne peuvent pas se permettre d'exiger un verrouillage ou une isolation temporaire des données mises en jeu sur une longue période.

En cas d'échec d'une transaction longue , il faut prévoir des mécanismes de compensation (action inverse annulant l'action d'origine : exemple= remboursement si livraison non effectuée).



Autre exemple classique : entité "dossier" avec num_dossier,

4. Intégrité référentielle & MDM

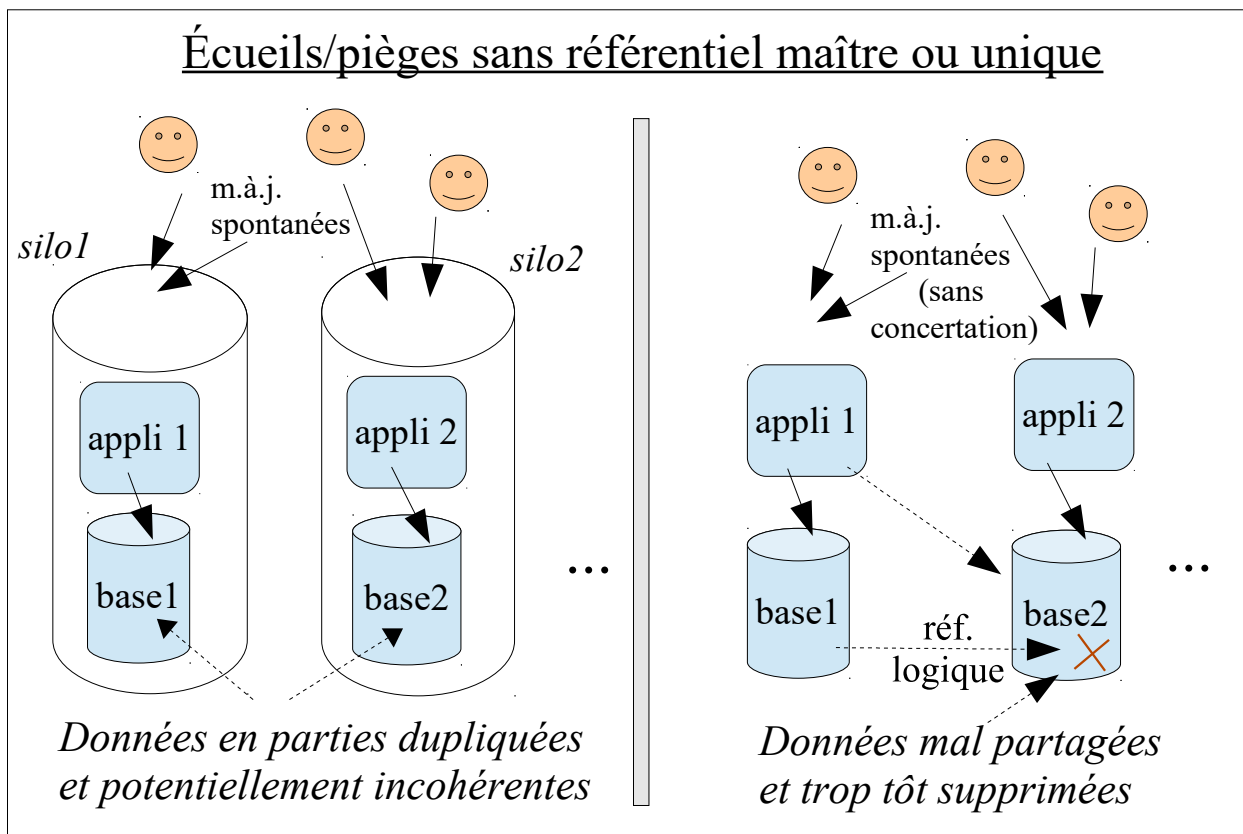
4.1. Stabilités des éléments référencés

Dans un système à contraintes d'intégrités centralisés (telle qu'une base de données unique prise en charge par un SGBDR), il est souvent impossible de supprimer involontairement/accidentellement une entité qui est encore référencée par une autre.

A l'inverse dans un système plutôt décentralisé comme internet ou SOA , une entité peut être supprimée dans un système (application) sans que les autres systèmes soient au courant (ex: URL devenue invalide, service utilisant un "vieil" id pour référencer une entité qui existait et n'existe plus.

==> Les spécifications (cahier des charges , modélisations,) devraient idéalement spécifier clairement certains éléments tels que la durée de validité d'une référence (et/ou des régulières vérifications ,).

sans MDM :



4.2. MDM (Master Data Management)

Pour adresser le problème de l'intégrité référentielle au sein de systèmes décentralisés , on se réfère souvent au concept de "**MDM**" (**Master Data Management**) .

MDM est quelquefois dénommé *GDR* (*Gestion de Données Référentielles*) en français .

MDM (ou GDR) est surtout utilisé pour gérer les données fondamentales (*de références*) d'une grande entreprise :

- les données « **clients/fournisseurs** »
- les données « **produits** »

- les données « **financières** ».

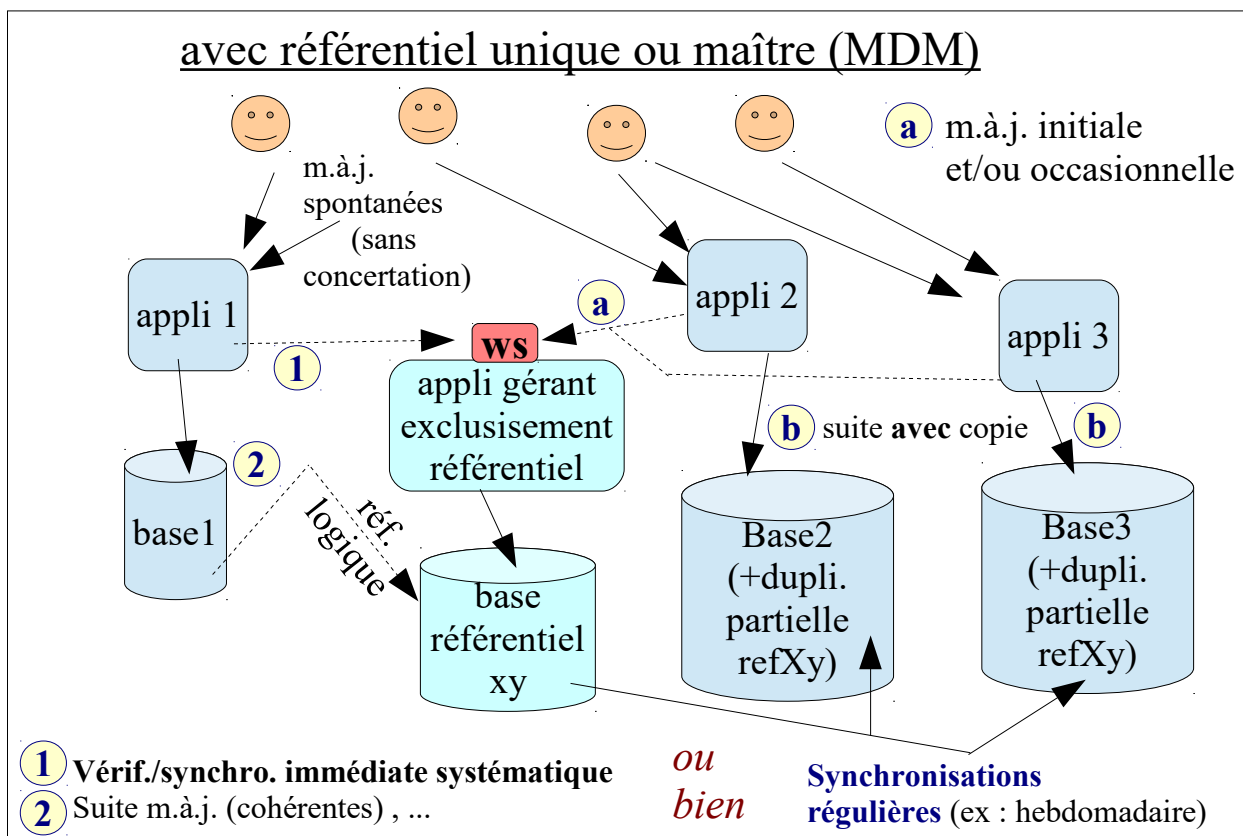
Besoin de distinguer (ne pas confondre):

- données maîtres de référence (partagées , avec différentes vues , stratégiques, sous système propriétaire souvent difficile à identifier)
- données procédurales (fruit d'une procédure , d'un processus métier , factuel , date précise, ...) souvent liées directement ou indirectement à la zone d'urbanisation "gisement de données"

et **bien modéliser les cycles de vie** (lorsque c'est possible) des données de référence.

avec MDM/GDR :

- Les données référentielles sont encodées et maintenues en un seul endroit, ce qui diminue le coût opérationnel lié à la maintenance et à l'encodage.
- Le système est le maître des données. Il les contrôle en sélectionnant quelles données il transmet à quel système.
- Le système contient une seule version active (il peut exister plusieurs versions inactives, tant passées que futures ; c'est même recommandé pour une meilleure flexibilité de la solution). Il est donc le garant de la seule version de la vérité et en cas de litige, sa version tient lieu de version officielle.



Un mode "pull" ou bien "push-pull" est envisageable au niveau des synchronisations entre les données de référence du référentiel "maître" et celles des référentiels/bases secondaires .

Fonctionnalités classiques de l'application gérant un référentiel :

- gestion d'un cycle de vie des données de référence (à travers WS de contrôle ou ...)
- ...

VII - Repères d'urbanisation

1. Notions essentielles d'urbanisation

1.1. Urbanisation du S.I. (présentation)

L'objectif principal consiste à *faire évoluer les différentes parties du SI de façon convergente* (en respectant les mêmes grandes lignes directrices) et à *faire cohabiter les parties existantes* au sein d'une *structure globalement découpée en zones/quartiers/blocs*.

Ce découpage en **modules** et sous modules **autonomes** vise à :

- garantir une certaine liberté d'implémentation
- clarifier les zones d'échanges (communications) entre les différentes parties du SI.

Plus particulièrement, l'urbanisation vise :

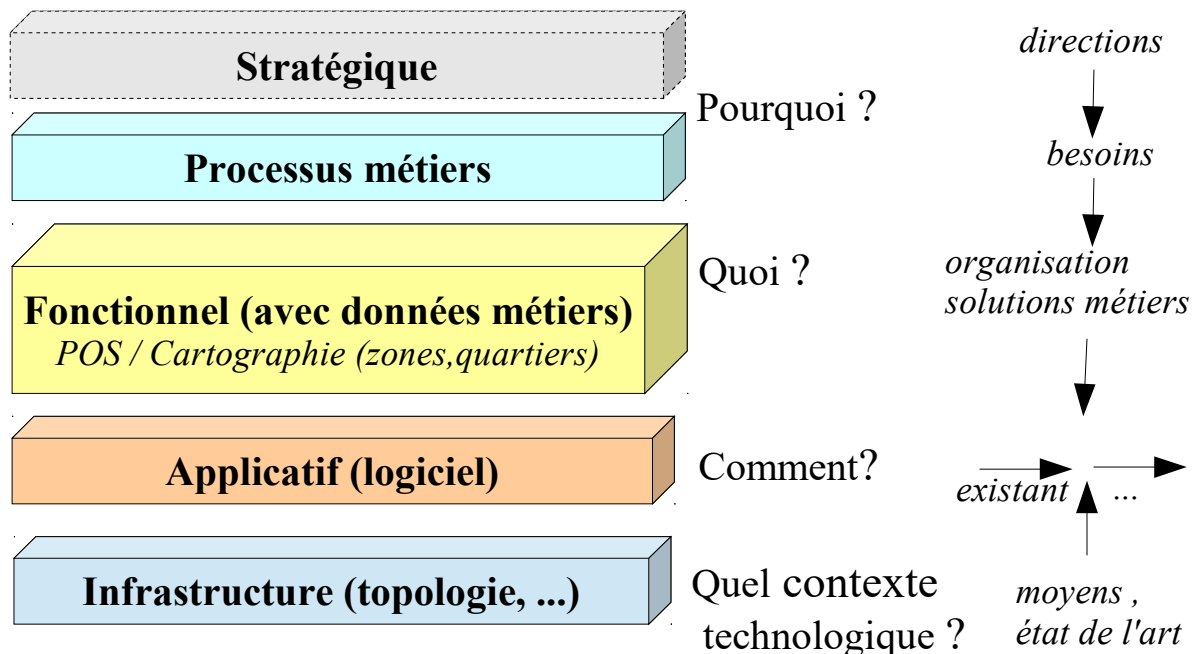
- à renforcer la capacité à construire et à intégrer des sous-systèmes d'origines diverses,
- à renforcer la capacité à faire interagir les sous-systèmes du SI et les faire interagir avec d'autres SI (interopérabilité),
- à renforcer la capacité à pouvoir remplacer certains de ces sous-systèmes (interchangeabilité).

et de manière générale pour le SI à :

- favoriser son évolutivité, sa pérennité et son indépendance,
- renforcer sa capacité à intégrer des solutions hétérogènes (progiciels, éléments de différentes plate-formes, etc.).

1.2. Cadre général / classique à 5 niveaux

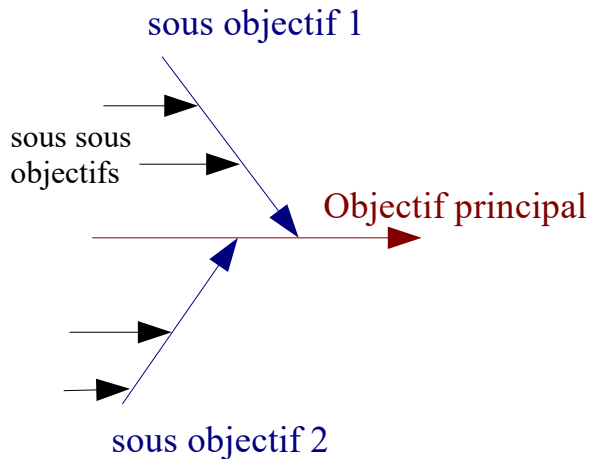
Cadre général (à 5 niveaux) pour l'urbanisation



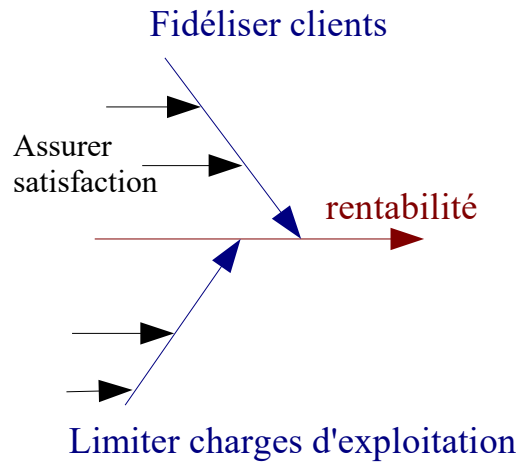
1.3. Niveaux "stratégiques" et "processus"

Expression des directions stratégiques

Diagramme d' Ishikawa



exemple



Processus métiers

**Objectifs métiers
(et sous objectifs)**

à atteindre en développant des

**Processus métiers
(et sous processus)**

*éventuellement
schématisés via*

**Business
Uses Cases**

généralement modélisés via des

**Diagrammes d'activités
(et sous diagrammes)**

UML

*ou
bien*

**BPMN
(Business Process
Model and Notation)**

2. Urbanisation fonctionnelle

Objectif principal : **Bien ranger , bien s'organiser** et surtout "**éviter les doublons**".

2.1. Zones , quartiers , blocs

L'urbanisation consiste à découper le SI en modules autonomes, de taille de plus en plus petite :

- les **zones**,
- les **quartiers** (et les **îlots** si nécessaire),
- les **blocs** (blocs fonctionnels).

Exemple :

- zone *production bancaire*
 - quartier *gestion des crédits*
 - îlot *gestion des crédits immobiliers*
 - bloc fonctionnel *gestion d'un impayé*

Entre chaque module (zone, quartier, îlot, bloc) se dessinent des **zones d'échange d'informations** qui permettent de *découpler* les différents modules pour qu'ils puissent évoluer séparément tout en conservant leur capacité à interagir avec le reste du système.

Éléments de cartographie fonctionnel du SI

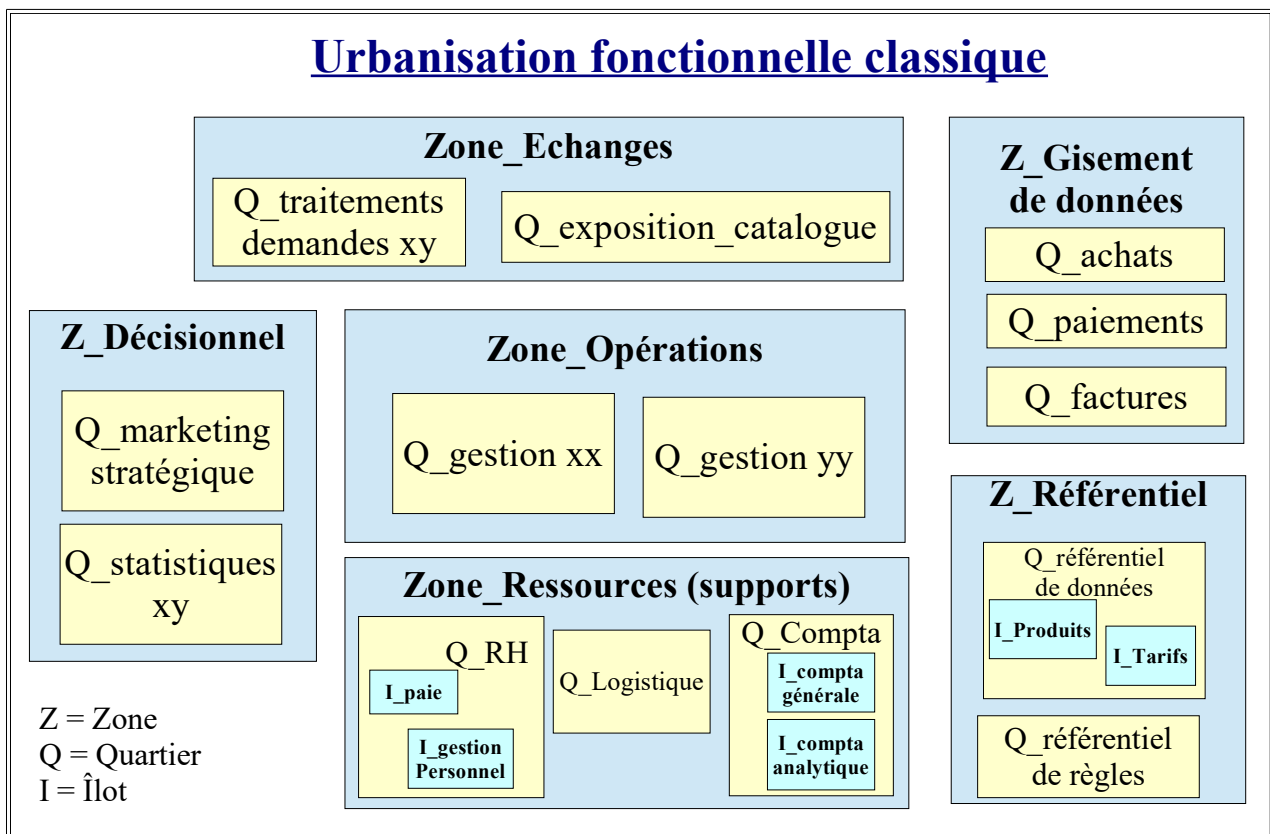
Domaines (types de Zones)	<<échanges>>, <<référentiel et gisement>> , <<opérationnel>> , <<ressources/support>>, <<décisionnel>>
Zones (fonctionnelles)	Regroupement homogène (métier , organisation/ entreprise , ...)
Quartiers (fonctionnels)	Secteur métier (ou para-métier)
Îlots (fonctionnels)	Souvent via une application (ou une partie d'un ERP , ...)

2.2. Les principaux types de zones

Zones	Caractéristiques
-------	------------------

<i>échanges</i> avec l' <i>extérieur</i> du SI	<u>acquisition</u> /émission de/vers les <u>partenaires</u> : <u>clients</u> , <u>fournisseurs</u> , etc. (prise du S.I.) Zone incluant les "IHM" exposées ,
<i>activités opérationnelles</i> (<i>cœur de métier</i>)	Zone(s) "cœur de métier" (incluant par exemple des quartiers de types gestion des opérations bancaires, gestion des opérations commerciales, gestion des opérations logistiques internes, etc.)
gestion des <u>données de référence</u> communes à l'ensemble du SI	les <u>référentiels</u> de <u>données structurées</u> (<u>données clients</u> , <u>catalogue</u> de <u>produits</u> et <u>services</u> , etc.) (référentiel de données stables)
gestion des <i>gisements de données</i> (également à voir comme un référentiel commun)	ensemble des <i>informations produites quotidiennement</i> , communes à l'ensemble du SI (données de production, etc.) (référentiel de données qui évoluent rapidement)
<i>Ressources (activités de support)</i>	<u>comptabilité</u> , <u>ressources humaines</u> , etc.
aide à la <u>décision</u> et le <i>pilotage</i>	<u>informatique décisionnelle</u> .

Urbanisation fonctionnelle classique



à évidemment adapté en fonction :

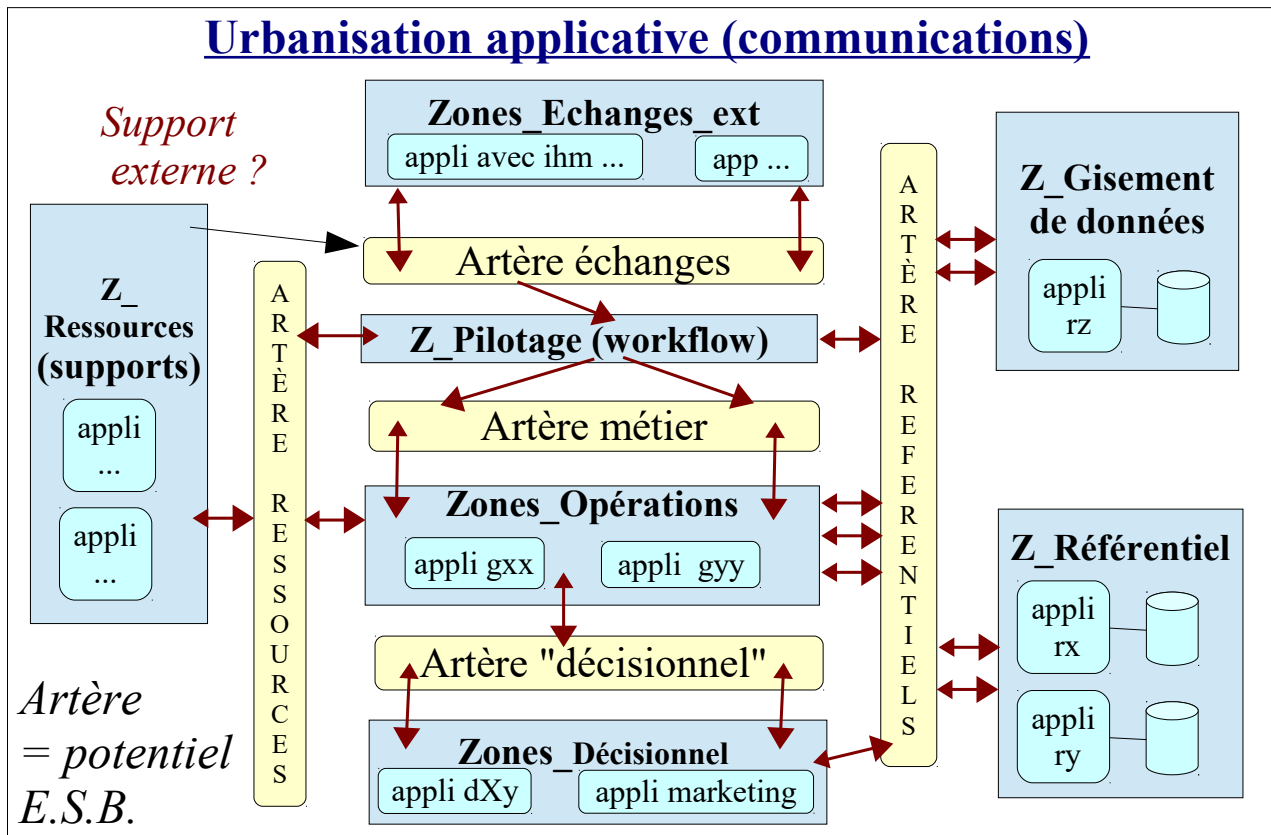
- du cœur de métier (banque, assurance , automobile,)
- de la taille de l'entreprise (TPE , PME, GE)
- de l'existant
- des choix stratégiques
- ...

Si plusieurs "cœurs de métier" coexistent , on aura alors plusieurs "zones opérationnelles" .

Certaines zones sont censées être uniques (ex : référentiels , échanges_extérieurs, décisionnel , ...)

3. Urbanisation applicative

Un exemple (discutable) parmi plein d'autres variantes / alternatives :



La Zone de pilotage (plutôt organisationnelle et en partie technique) à pour rôle de s'occuper

- des orchestrations de services métiers
- des workflows à base de communications asynchrones
- ...

Ce diagramme (à grande échelle) ne montre que les principales artères de communication. Il peut aussi exister des canaux d'échanges entre les quartiers (au cas par cas).

4. Prise en compte de l'urbanisation dans la modélisation SOA

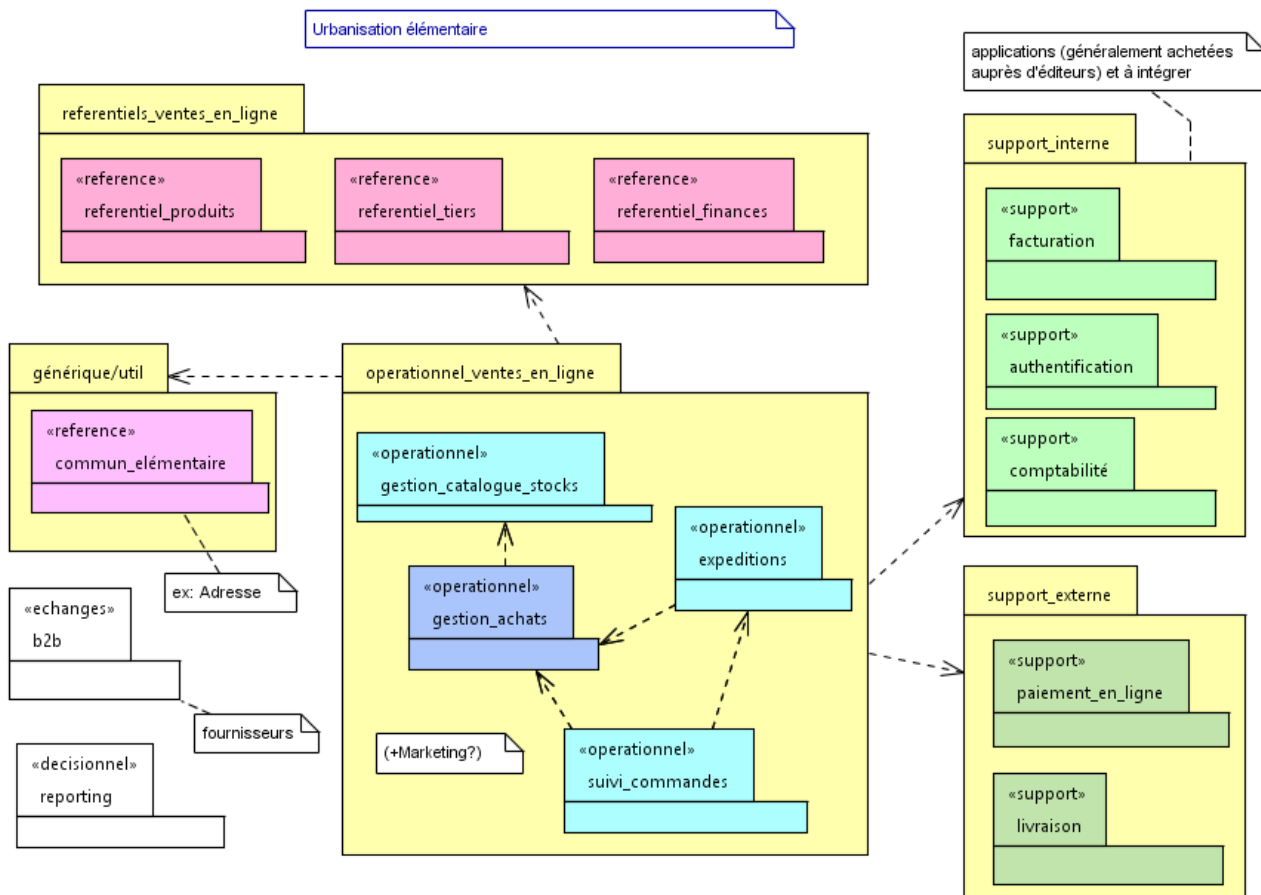
4.1. Modélisation SOA à grande échelle (vue d'ensemble)

Un modèle UML "grandes lignes" (sans détails) peut servir à fixer quelques repères

d'urbanisation et ainsi **dresser une vue d'ensemble sur le contexte organisationnel** d'une entreprise ou d'une institution.

Dans **contexte_XY.UML** , on pourra par exemple modéliser certaines "**Zones**" et "**Quartiers**" d'urbanisation comme des **packages (et sous packages) UML** .

Exemple simple (améliorable) :



Rappel : La modélisation SOA est en partie "transverse" et doit (en partie) être abordée "à grande échelle" .

Pour la maîtrise d'ouvrage → "plan d'urbanisation = selon organisation interne (à faire évoluer) "

Pour la maîtrise d'oeuvre → "plan (partiel) d'urbanisation = cadre imposé" (ex : telle nouvelle application "Xy" devra être intégrée dans le quartier "Q8" et devra communiquer avec les services du quartier "Q9").

4.2. Modélisation SOA (détaillée) à petite échelle

Besoin de bien structurer (avec packages et sous packages UML) :

- Les services rendus par l'application courante
- Les services invoqués (fournis par d'autres applications)
(idéalement : "import de sous-modèle UML" si l'éditeur UML en est capable)

5. Modéliser les cycles de vie des objets de référence

Pour pouvoir gérer l'intégrité référentielle dans un système décentralisé/éclaté en de multiples applications , il faut absolument définir précisément les cycles de vie des entités fondamentales.

5.1. Utilisation adéquate d'un diagrammes d'états UML

Après avoir identifié (et modéliser sous forme de classes UML) les entités de références , il est fortement conseillé d'élaborer quelques **diagrammes d'états UML** .

Dans ce diagramme d'états ,

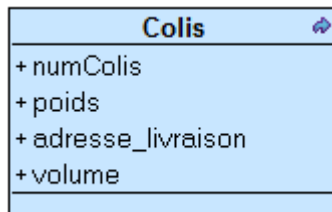
- l'état initial correspondra souvent à la création de la nouvelle entité (ex : nouveau client , nouveau produit)
- l'état final correspondra souvent à la suppression définitive de l'entité en base (delete SQL).
- les états intermédiaires et les transitions associées permettront de bien spécifier l'avancement contrôlé du cycle de vie.

Points importants :

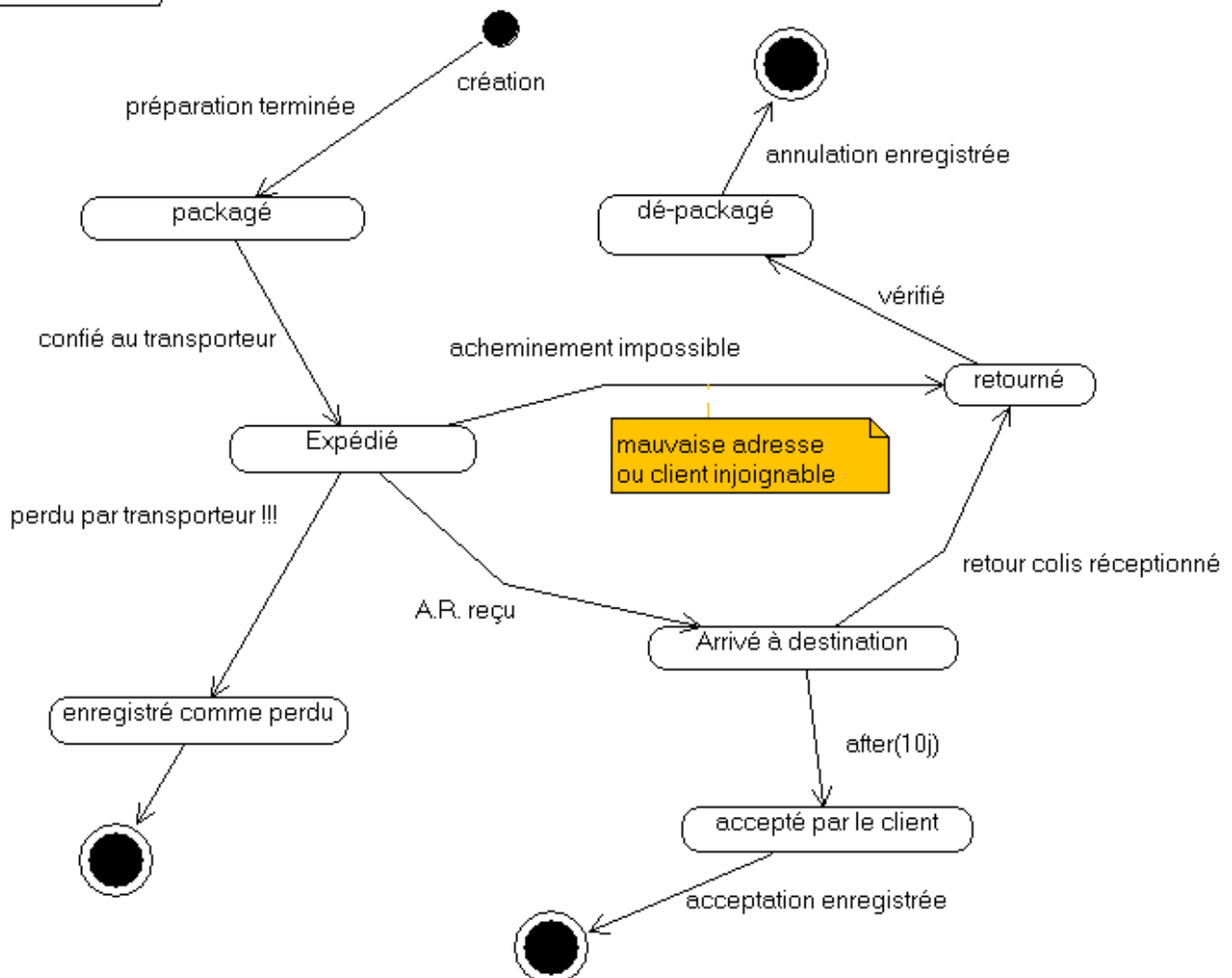
Souvent besoin d'une suppression douce et progressive :

- 1) désactivation (ou suppression) logique en mémorisant la date de désactivation
- 2) attente d'une période écoulée (ex : 2 mois ou 1 an) pour être certain que l'entité ne soit plus référencée par d'autres éléments actifs.
- 3) éventuel archivage
- 4) suppression définitive

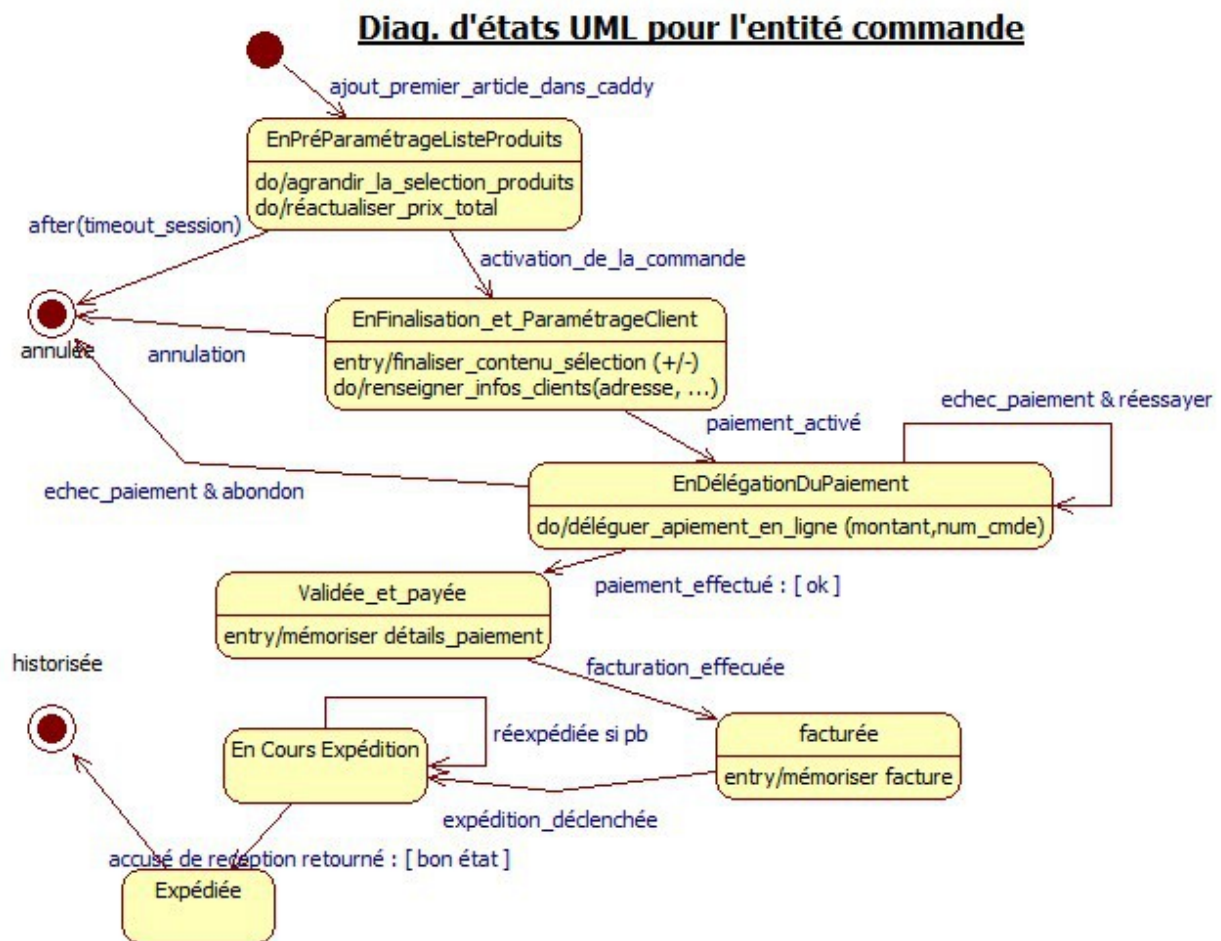
5.2. Exemple 1 (cycle de vie d'un colis)



StateMachine



5.3. Exemple 2 (pour la syntaxe) :

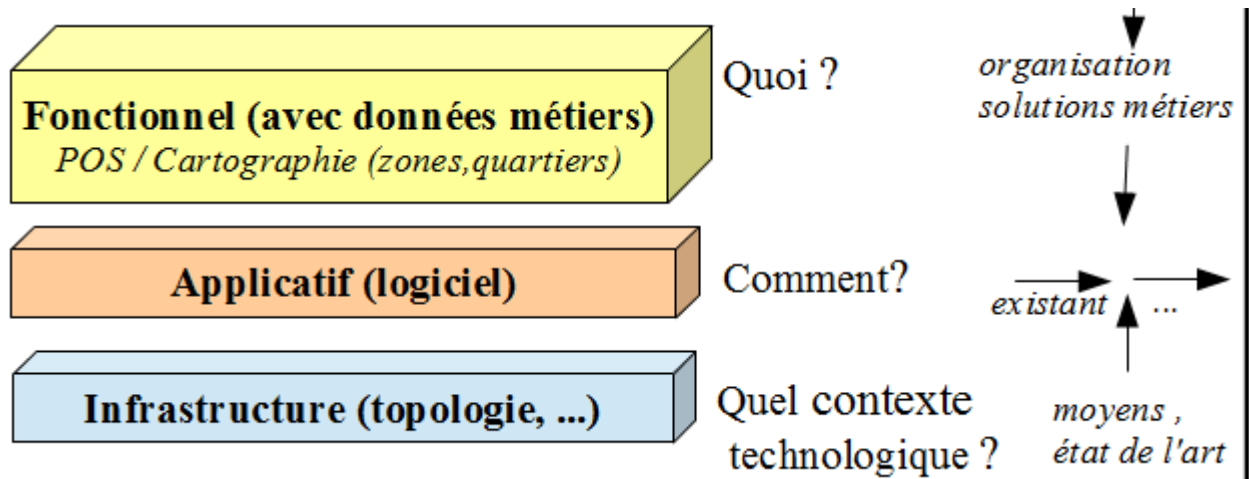


VIII - Architecture logicielle (SOA)

1. Dérivation du fonctionnel en "logiciels SOA"

1.1. Au cas par cas

Rappel du cadre (de cette activité méthodologique) :

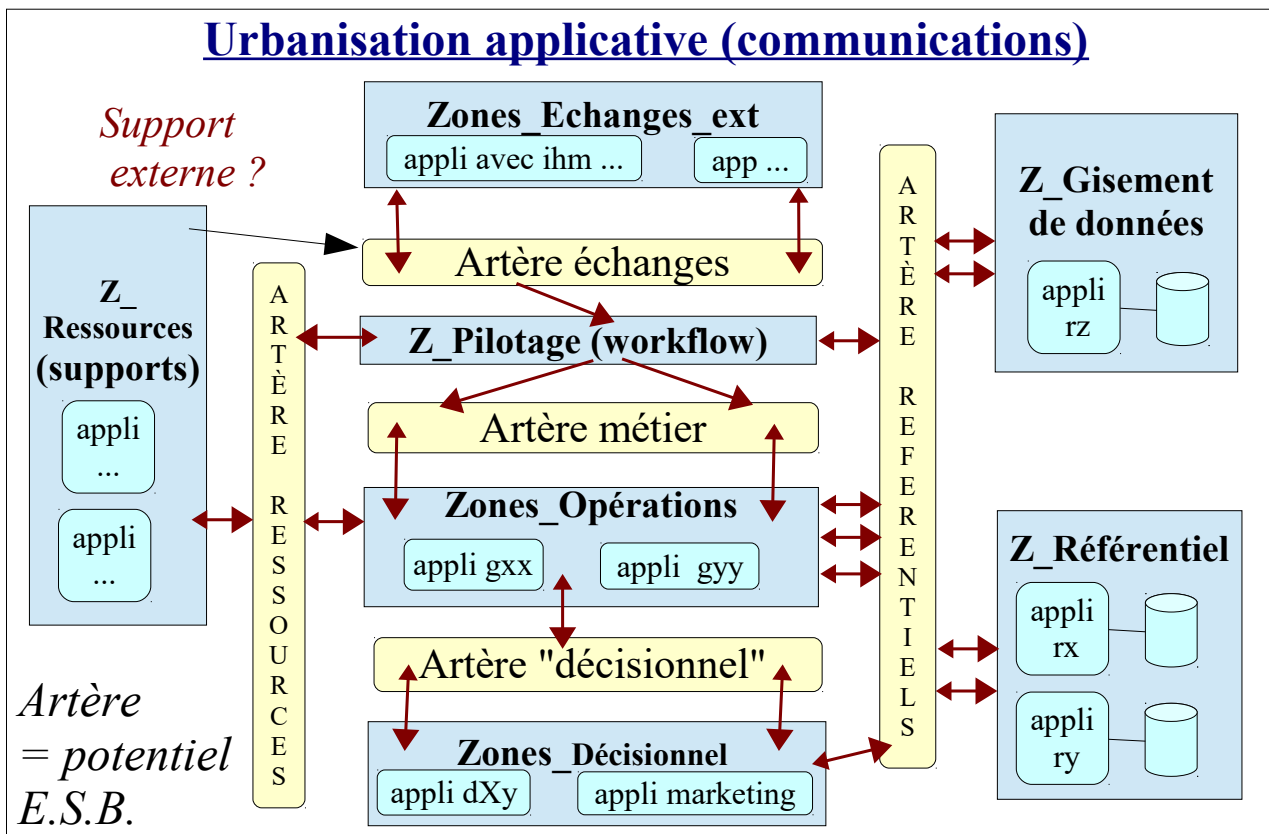


Considérations préliminaires :

Comme le montre le diagramme ci dessus, **l'architecture logicielle est à la croisée des chemins**. Sa définition doit tenir compte d'un grand nombre de contraintes (existant , organisationnel , état de l'art , moyens financiers et techniques , ...)

→ il n'y a donc pas une seule solution figée mais **de multiples solutions (au cas par cas)** .

1.2. Pistes classiques



- 1) élaborer une variante (éventuellement améliorée) du diagramme ci-dessus
- 2) définir les natures (potentiellement hybrides) des artères de communications :
 - canaux réseaux
 - DMZ
 - ESB purement technique (à traverser)
 - ESB hébergeant du fonctionnel (ex : orchestration /pilotage)
 - ...
- 3) formaliser les règles d'échange (sens unique , ...)
- 4) répartir les applications dans ce contexte
- 5) passer petit à petit aux détails
- 6) ...

2. Dérivation du modèle logique sous la forme de composants SOA

Bien qu'il existe de multiples possibilités technologiques pour mettre en œuvre une architecture SOA, on peut tout de même dégager quelques règles assez générales au niveau de la projection du fonctionnel dans la technique :

- Tout service métier synchrone, proche des données persistantes et/ou devant être associé à des transactions courtes sera en général transposé sous forme de service java ("EJB3 stateless" ou "Service Spring" avec un point d'accès SOAP) [*java ou équivalent "c#/.net" ou ...*]

- Tout service métier de coordination pure (ne gérant pas directement la persistance et n'ayant pas impérativement besoin de transaction courte) pourra être soit transposé en java (ou c#) , soit transposé en BPEL ou bpmn. Le plus simple est souvent le mieux (facile à maintenir).
- Tout processus métier (non trivial / avec logique conditionnée et/ou aspect asynchrone et/ou opérations de compensations associées à des transactions longues) pourra être transposé en BPEL (ou éventuellement en Bpmn + java (jbpm , activiti , ...)) .
- Tout service asynchrone pourra être transposé en EJB3 MDB (Message Driven Bean) ou en équivalent Spring (ou autre) et pourra s'appuyer sur une technologie "files d'attentes" (ex : JMS).
- Tout service à caractéristiques hybrides (partiellement transactionnels , partiellement pas ,) aura plutôt intérêt à être décomposé en un service principal et un sous service (par exemple transactionnel).

Les structures de données échangées (documents , paramètres des méthodes distantes, ...) seront généralement retranscrites dans divers formats (java , xml, ...) et leurs structures seront spécifiées sous forme de schémas XSD (quelquefois stockés dans des référentiels).

En résumé , les principaux critères influant le choix des technologies sont :

- besoin ou pas de persistance directe (accès à une base de données) <<persistence>>
- besoin ou pas de transaction courte <<tx>>
- synchrone ou asynchrone ? (<<async>>)
- Logique métier évoluée (processus non linéaire , opérations de compensations , ...)

3. Prise en compte des aspects techniques

3.1. performances

Sur le point des performances, il y a certains aspects à surveiller de près :

- plus on décompose un système informatique en services et sous services de technologies différentes et plus on a potentiellement à effectuer des "traversées du réseau"
- plus on effectue de transformations intermédiaires (protocoles/formats/....), plus on consomme de traitement CPU (encodage/décodage).

Autrement dit, lorsque c'est possible (sur du neuf), il vaut mieux utiliser des technologies uniformes (java/jbi, ws "soap") pour éviter trop de traversées réseaux (ou de re-traitements).

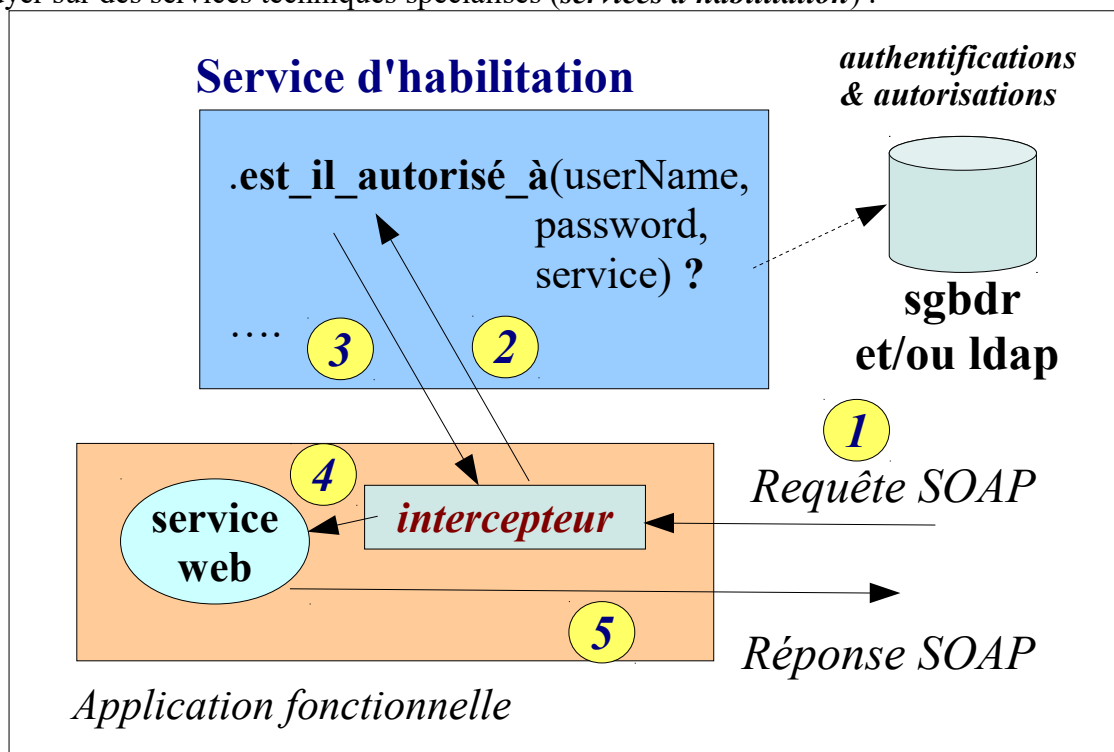
Et d'autre part, un effort d'urbanisation (consensuelle et pro-active) s'avère généralement rentable sur le long terme. Autrement dit, quelques lignes directrices communes à l'ensemble du SI (avec des implémentations programmées et éventuellement différées) est un minimum à mettre en œuvre pour éviter une trop grande divergence entre les différentes parties du S.I. et pour diminuer le nombre de retraductions qui pourraient être évitées.

En bref : il faut gérer au cas pas cas (dans l'urgence) les disparités du présent mais il faut également viser/programmer une convergence des futures versions.

L'optimisation des performances n'est pas "que technique" ; elle passe également (en partie) par des réorganisations structurelles analysées comme nécessaires.

3.2. Sécurité (service d'habilitation)

Pour vérifier la validité d'une authentification (username/password), on pourra éventuellement s'appuyer sur des services techniques spécialisés (*services d'habilitation*).



3.3. Pour ne pas traverser les ESB pour rien ...

Puisque les grandes lignes de l'urbanisation logicielle nous incite fortement à passer par des canaux / artères de communication pour invoquer des services situés dans d'autres zones , autant que cette indirection soit techniquement utile.

Un ESB pourra par exemple :

- effectuer des mesures de performances (temps de réponses , débits , ...) → KPI , SAM
- intercepter les requêtes et effectuer des contrôles de sécurité (authentification , ..)

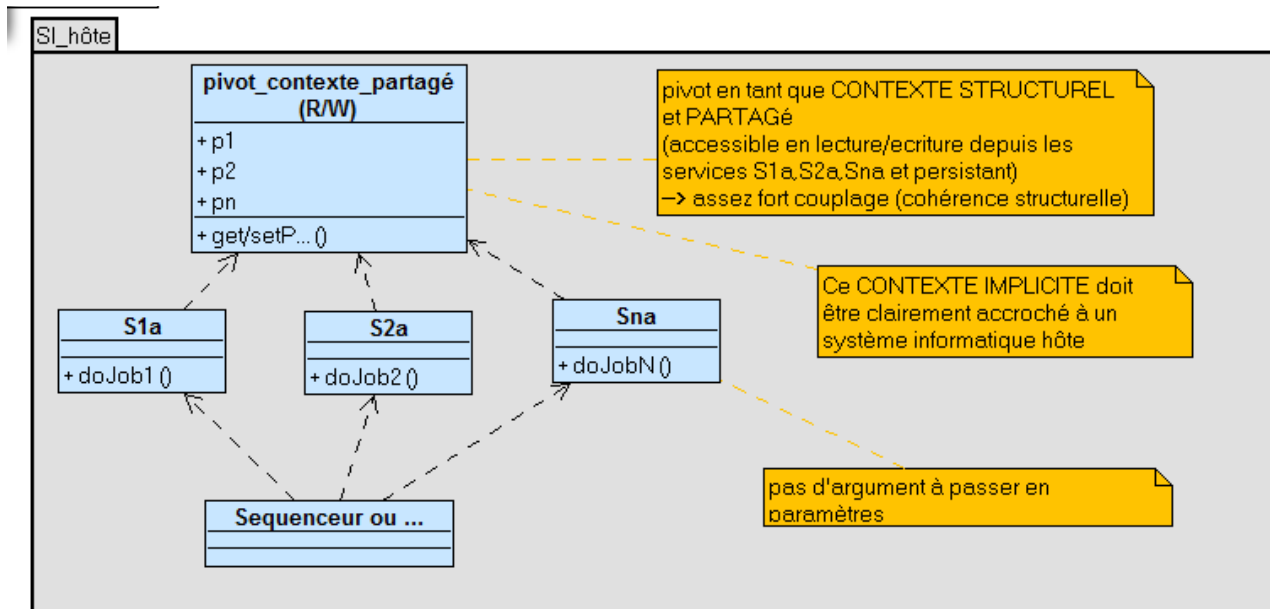
...

IX - Structuration des données Pivots

1. Données "pivots" (échangées)

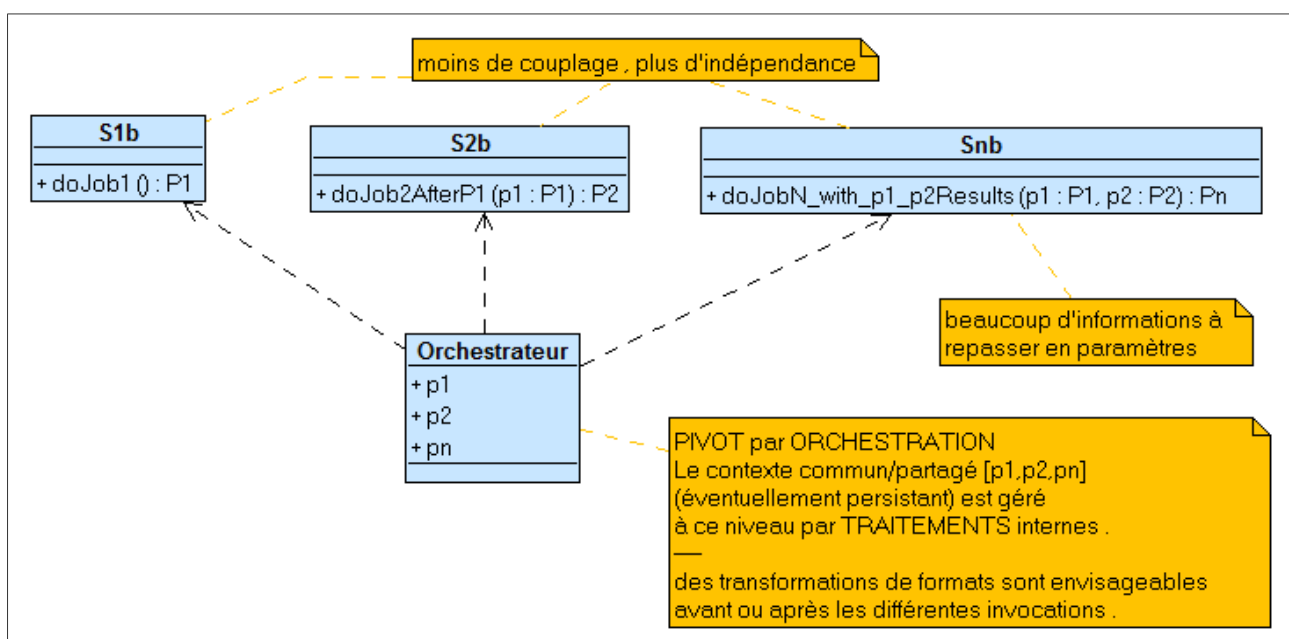
1.1. Bien choisir les types de pivots

Pivot en tant que contexte structurel partagé :



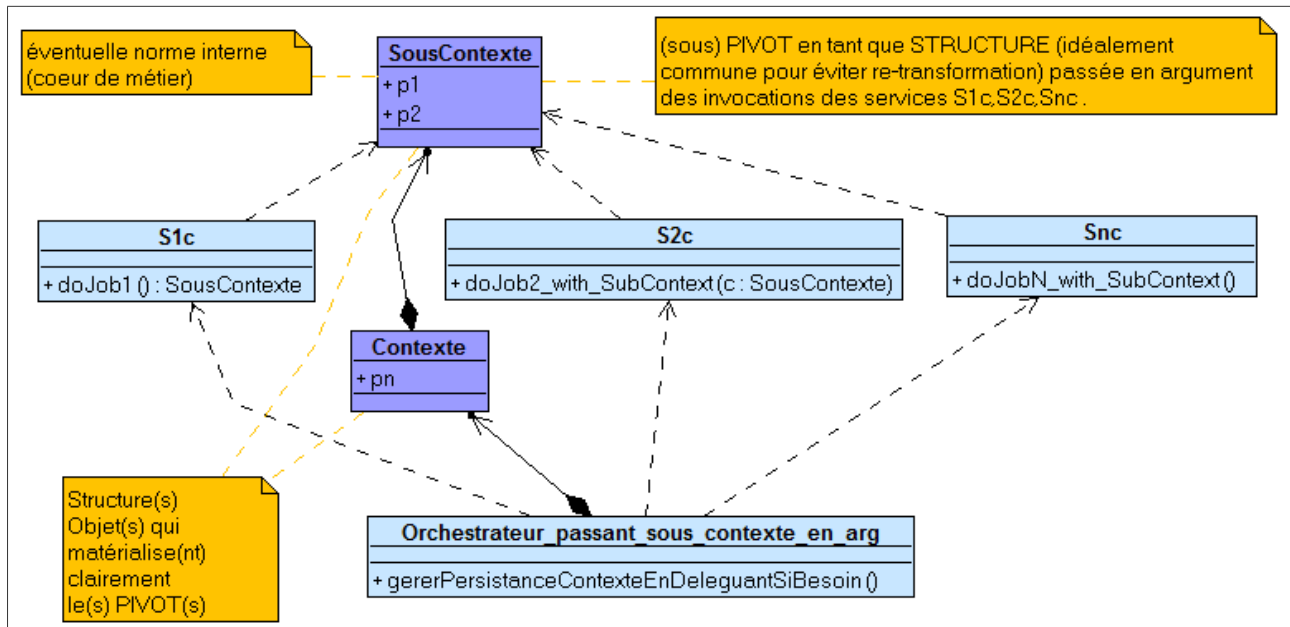
→ envisageable que si services clairement cloisonnés à un sous système informatique précis .

Pivot par orchestration :



→ adapté si les services à orchestrer sont placés dans des sous systèmes informatiques différents .

Pivot en tant que structure de données (éventuellement partagée) passée en argument :



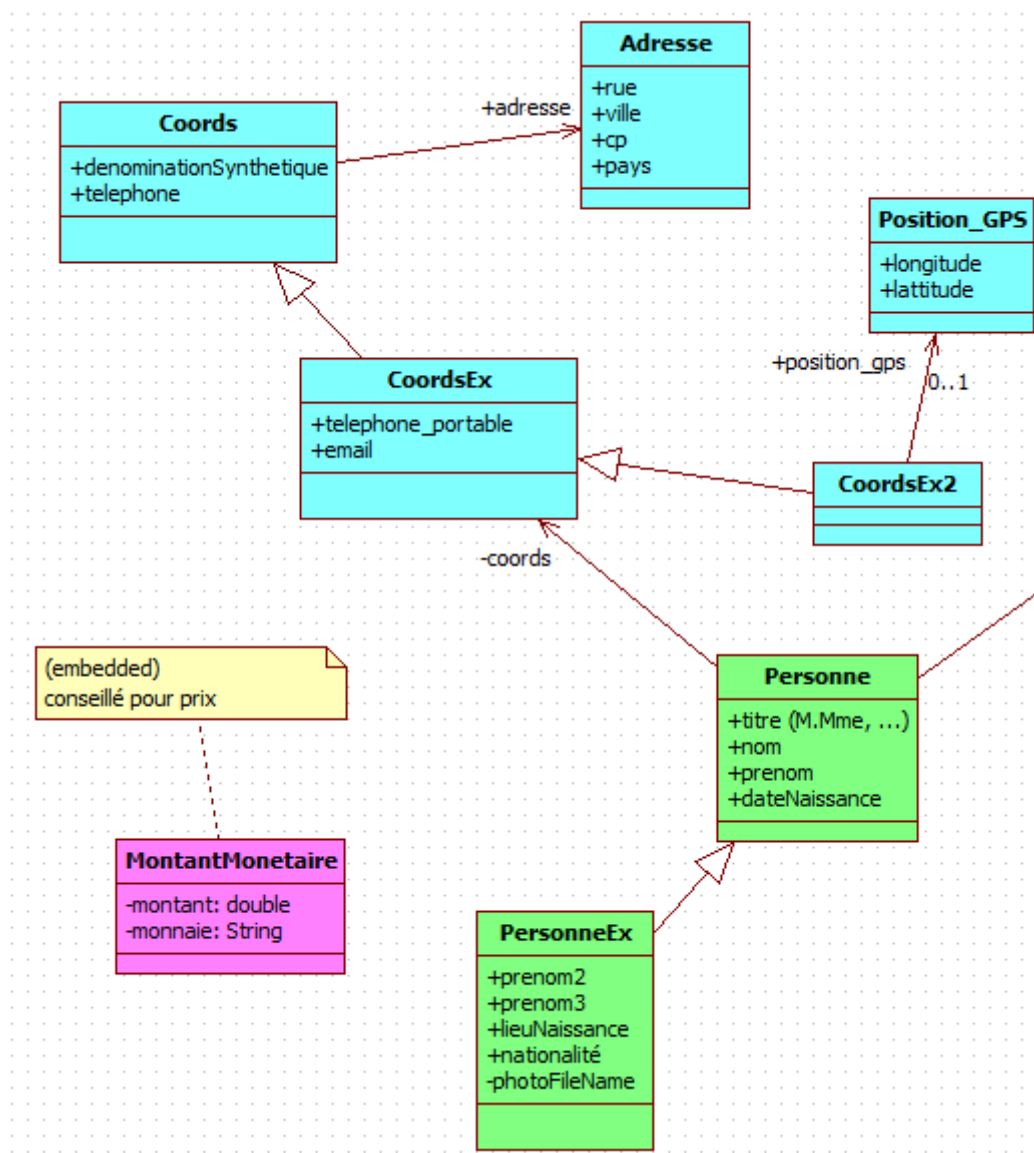
2. Recherche d'abstractions, spécialisations/variantes

De façon à bâtir des composantes stables du S.I. (fondation/socle réutilisable) , on a tout intérêt à

- ne pas se limiter aux besoins spécifiques du projet du moment
- **effectuer une analyse approfondie des invariants** (*bases des relations d'héritages*)
- **analyser les besoins en variantes** (*avec une éventuelle petite anticipation*) .

Ceci est à fond dans l'esprit de la méthode Praxème qui appelle "machine logique" l'ensemble des abstractions fondamentales .

Quelques exemples (à discuter/adapter au cas par cas) :



Quelques autres exemples (à discuter , à adapter au cas par cas) :

Considérer une "**Prestation**" comme un cas particulier de "**Produit**" (au sens large)

- avantage : on peut commander de la même façon un produit ou une prestation .
- inconvénient : le sous entendu "Prestation héritant de Produit" peut engendrer quelques ambiguïtés (si ce n'est pas bien documenté).
- variante : "**Prestation**" et "**Produit**" comme deux sous classes de "**Offre**"

Considérer "**PersonnePhysique**" et "**PersonneMorale**" comme héritant tout deux de "**Personne**"

- avantage: ce qui est commun au deux est rangé dans "Personne" , ce qui est spécifique est placé dans "PersonnePhysique" ou "PersonneMorale".
- inconvénient: "PersonnePhysique" est souvent "trop précis / pas naturel" au niveau des "vues"
- variante : "**Personne**" (physique) et "**Entreprise**" comme deux sous classes de "**Acteur**"

La composition est souvent plus appropriée que l'héritage :

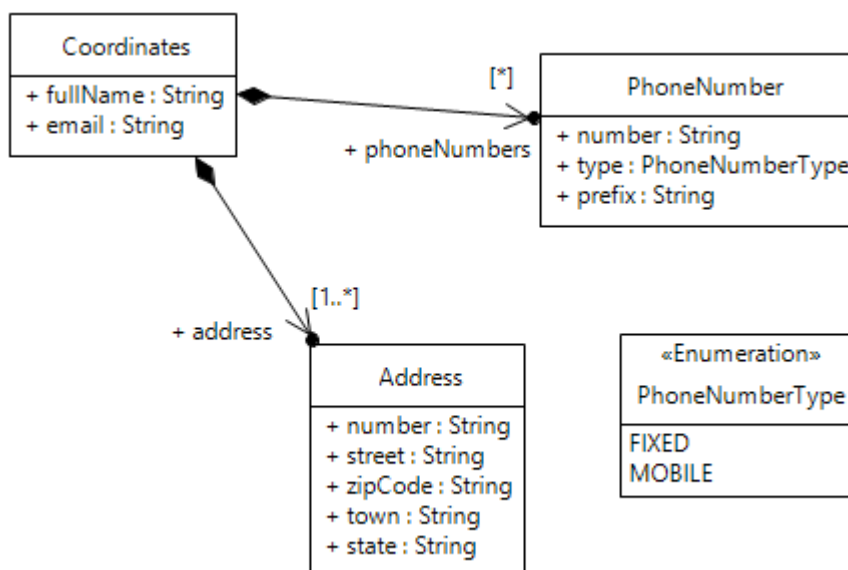
Structurer des variantes en se basant sur des relations d'héritages est naturel (et efficace) dans un monde purement orienté objet .

Cependant, les **données "pivots"** sont avant tout des données d'échanges qui **ont souvent besoin d'être transformées , assemblées , désassemblées , ...**

De ce point de vue , **la composition est alors plus pratique et efficace à gérer que l'héritage :**

On peut souvent récupérer et recopier d'un seul bloc un sous objet pour le retransmettre ailleurs.

Une sous-structure commune (lorsqu'elle est envisageable) permet ainsi d'économiser de nombreuses lignes de code java ou bpel et les performances sont également améliorées.



3. Hiérarchie de domaines / références communes

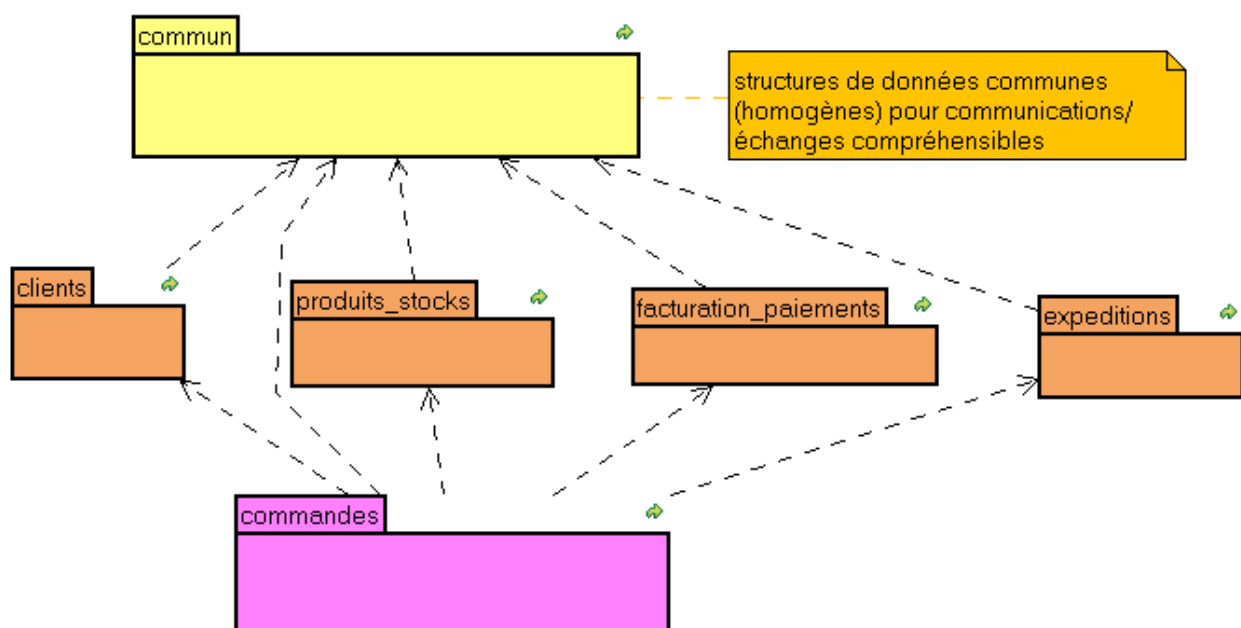
Idee clef (souvent pertinente) :

Une application de gestion de facture à besoin de gérer très finement une facture avec tous ses détails. Les autres applications qui sont concernées par des factures ont généralement besoins de manipuler qu'une vue simplifiée d'une facture .

Une structuration élégante consiste à définir dans un *package "commun"* une *série de vues simplifiées des principales entités "métier" manipulées dans l'ensemble du SI* .

Dans le package "facturation" , on pourra définir une entité précise "*facture_détaillée*" (qui pourra éventuellement hériter de la vue "facture" du package "commun").

Au final , le package "commun" comporte une grande liste de structures de données communes à une grande partie du S.I. (ce qui facilitera les échanges/communications) .



Attention : Ces structures (idéalement "communes" au sens "plus petit dénominateur commun") ne sont à considérer que comme des structures de données d'échange. Il s'agira souvent de "vues" ou d'adaptations .

3.1. Liens avec l'urbanisation et avec l'existant

L'urbanisation traditionnelle nomme "**données de références**" les données communes à l'ensemble du SI (référentiel de produits , clients , ...) .

X - Packaging, déploiement composants SOA

1. Modélisation de "composants SOA"

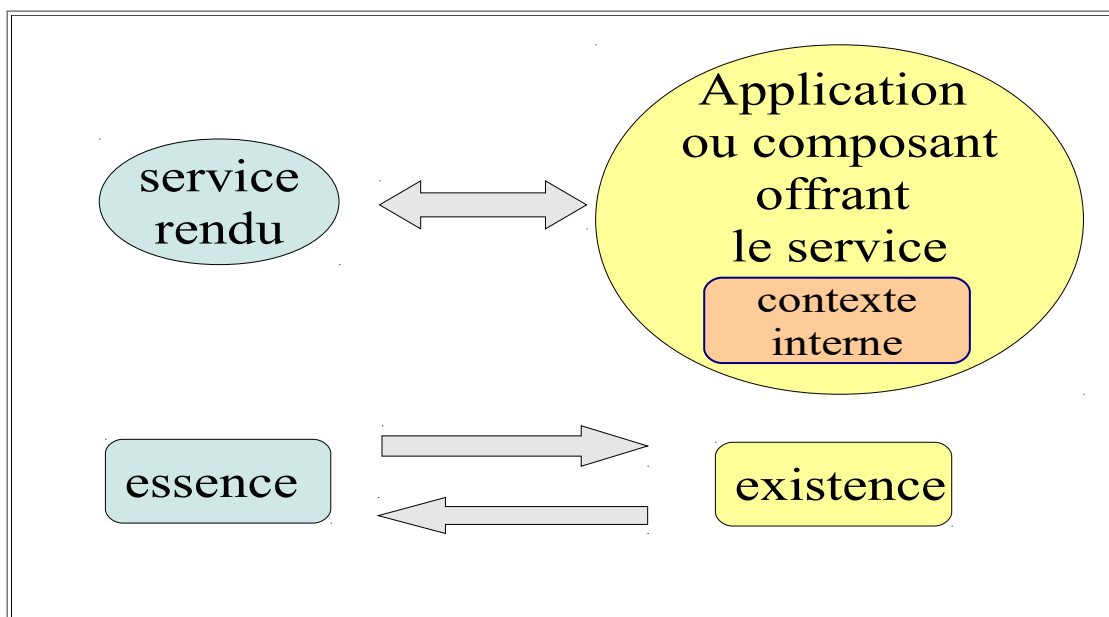
1.1. Bien modéliser (d'une façon ou d'une autre) les contextes

A part quelques cas triviaux (ex: service de calcul élémentaire) , la plupart des services sont intelligibles (sans ambiguïté) que si l'on précise un minimum certains contextes :

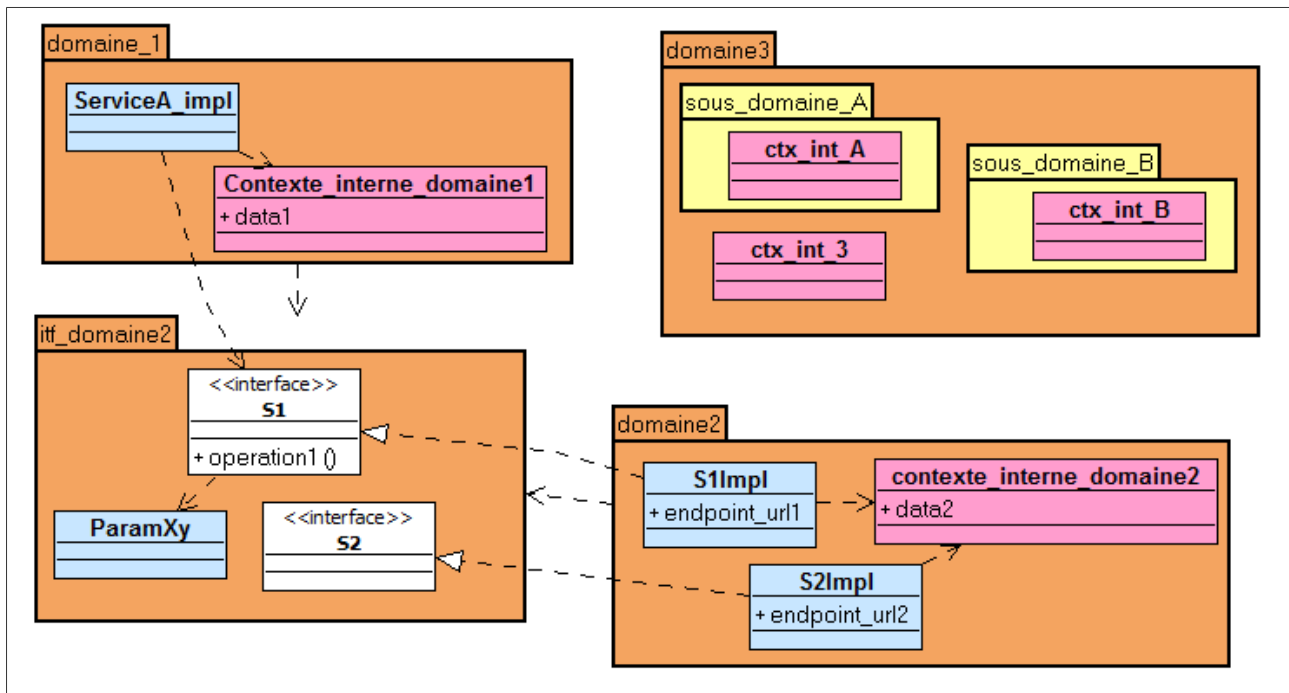
- contexte de persistance (base de données de l'application offrant le service)
- contexte d'exécution (mémorisation de l'état d'avancement d'un processus métier : processus long en partie asynchrone)
- contexte implicite/explicite , privé/public , opaque ,

1.2. Penser à une structuration logique (future implémentation) à base de composants offrant des services

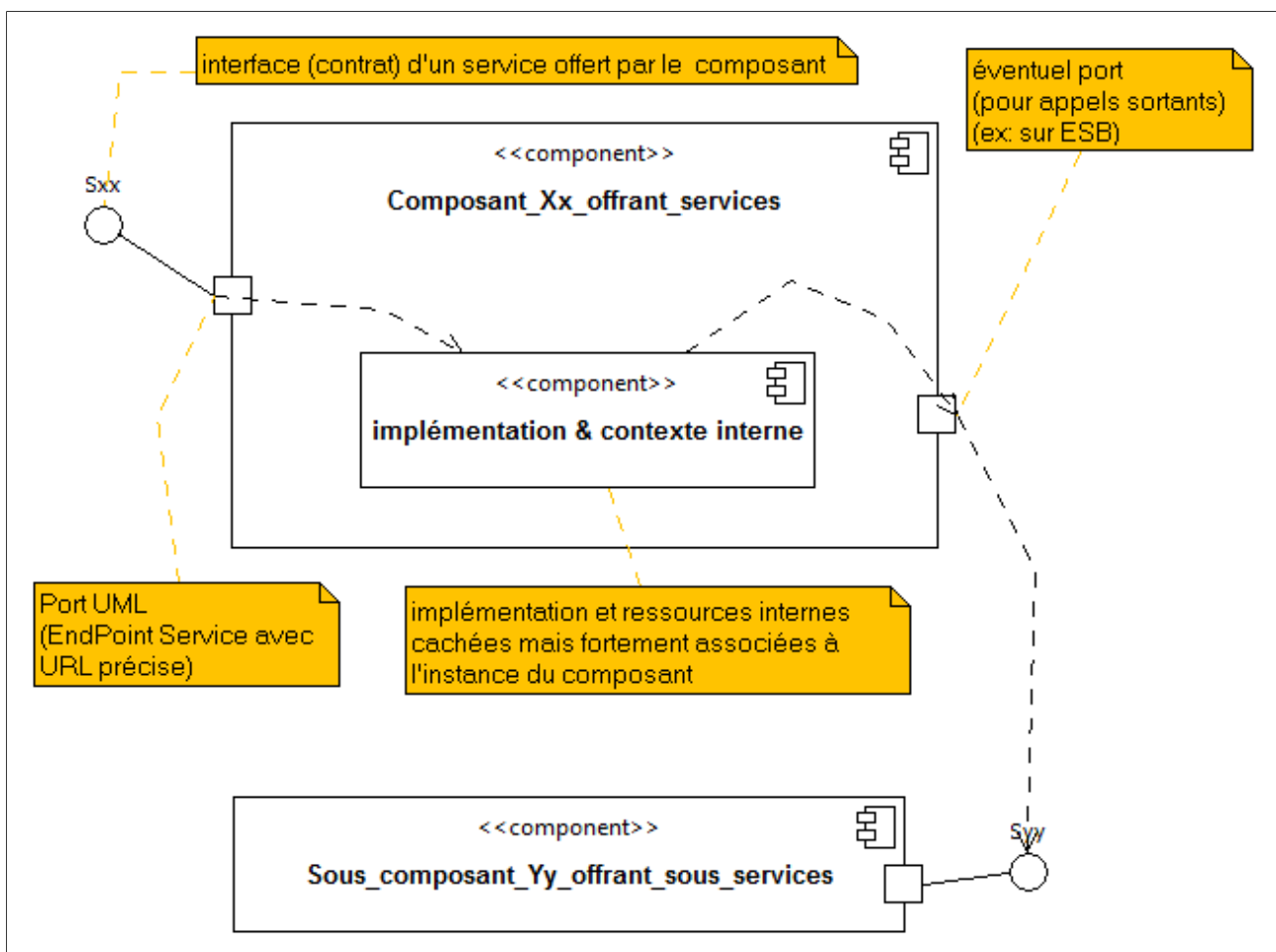
Beaucoup de contextes peuvent être concrètement pris en charge par des composants informatiques (de petites ou grandes tailles) .



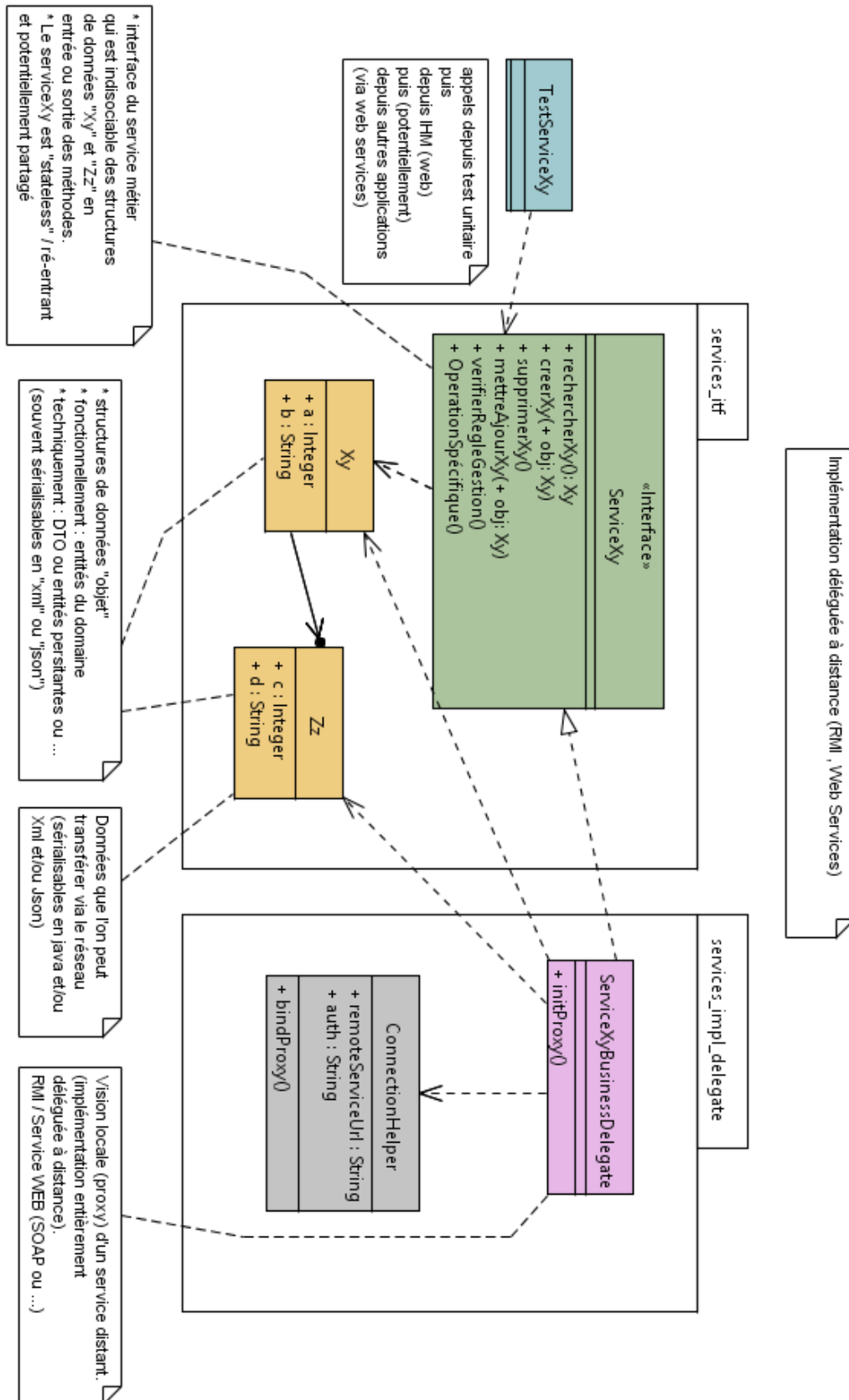
En UML, cette notion de contexte peut se modéliser via des diagrammes de classes (avec packages)

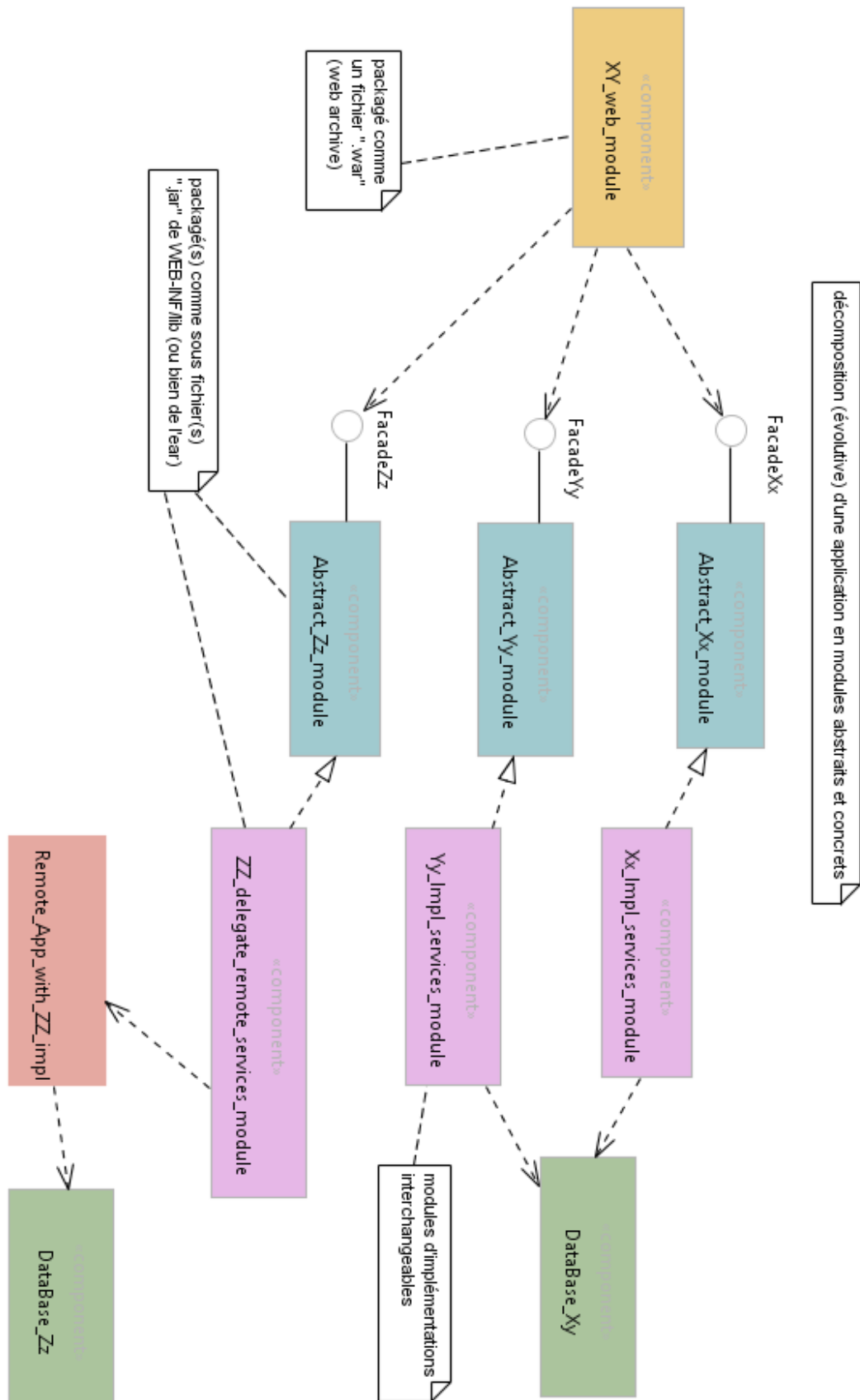


ou bien via des diagrammes de composants .../...



2. Packaging composants "soa" (itf , impl , delegate)





3. Gestion des versions (évolutions à prévoir)

3.1. Versions de niveau "fonctionnel/logique/abstrait"

Changements de niveau "XSD", "WSDL", évolution des contrats /interfaces.

Ces évolutions ont de très grandes conséquences et peuvent remettre en cause certaines communications entre applications.

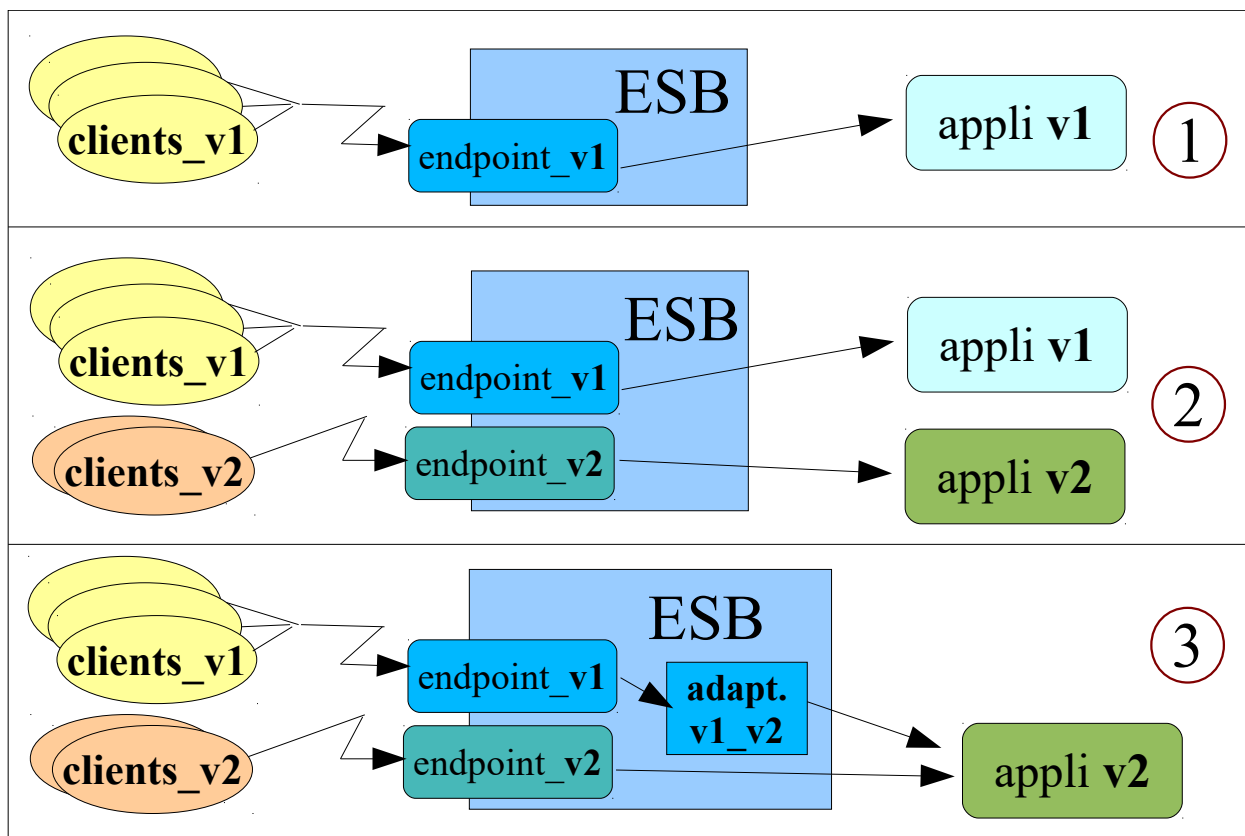
==> Il sera quelquefois nécessaire de faire cohabiter 2 (ou 3 ou ...) versions différentes d'un même service de façon à ce tous les appels soient possibles :

- ceux émanant des applications qui invoquent une ancienne version
- ceux qui sont émis par des applications invoquant une nouvelle version

Astuce (via ESB) :

Pour éviter de faire fonctionner (et maintenir) deux versions en parallèles , on peut souvent :

- faire fonctionner que la nouvelle version (en générale plus riche que l'ancienne)
- mettre en place sur un ESB un point d'accès vers un adaptateur qui retranscrit des appels à l'ancien format vers des appels au format enrichi de la nouvelle version.



3.2. Versions de niveau "pragmatique/physique/implémentation"

Simple changements d'implémentations (autre techno , autre URL , ...).

Il suffira en général de retirer de l'ESB l'ancienne implémentation puis d'enregistrer la nouvelle implémentation .

Les applications clientes qui invoquent un service en passant par l'ESB auront alors des requêtes qui

seront automatiquement re-routées vers la nouvelle implémentation .

3.3. changement d'implémentations avec basculement idéalement en douceur , ...

Lors d'un changement d'implémentation (enregistré sur l'ESB) , il faudra tout de même prévoir un basculement en douceur :

- test (hors prod) du basculement pour détecter éventuels problèmes
- prévoir une période calme (nuit , week-end , ...)
pour retirer l'ancienne version et placer la nouvelle
avec une potentielle petite interruption de service qu'il s'agit de minimiser .
- selon la nature du service (stateless & traitements courts , processus long) on aura (ou pas) besoin de se préoccuper des sessions ou processus en cours .

3.4. restructuration(s) cohérente(s) : nouveaux services , nouvelles dépendances ,

Certaines restructurations touchent en profondeur l'architecture du SI :

- toute une nouvelle chaîne de dépendances est à mettre en place
- certains anciens services pourront (à terme) être arrêtés s'ils ne sont plus utilisés.

Ce type d'évolution (au cas par cas) est à étudier en profondeur (en analysant les répercussions de chaque dépendance).

L'utilisation d'ESB (avec URL stables) est recommandé pour effectuer une transition douce mais ça ne règle pas tous les problèmes.

Certains gros changement d'architecture seront quelquefois pris en charge en mettant en place une période transitoire où deux systèmes fonctionneront en parallèle :

1. l'ancien système est en marche (avec de nombreuses sessions en cours)
2. on démarre en parallèle le nouveau système (qui commence alors à être utilisé)
3. on envoie un message de basculement à tout le monde
"arrêter au plus vite les sessions en cours et utilisez la nouvelle URL"
et on attend quelques heures (ou plus) de façon à ce que toutes les anciennes sessions aient le temps de se terminer
4. on arrête l'ancien système

4. documentation

Etant donné que l'architecture SOA repose sur un ensemble de dépendances vitales au fonctionnement du SI , on a absolument besoin de bien documenter les différents éléments de l'architecture.

Cette documentation doit être idéalement assez synthétique et facilement lisible/compréhensible.

Le contenu habituel d'une bonne documentation SOA (à versionner) est à peu près le suivant :

- spécifications fonctionnelles détaillées de chacun des services (ex : diag de classes UML)
- spécifications fonctionnelles des processus métiers (ex : BPMN et/ou diag d'activité UML)
- spécifications d'architecture (plan d'ensemble , liens entre ESB et applications , ...)

- spécification détaillées des contrats de services (référentiel de XSD/WSDL)
-

5. pilotage , suivi et éventuelles corrections/anticipations .

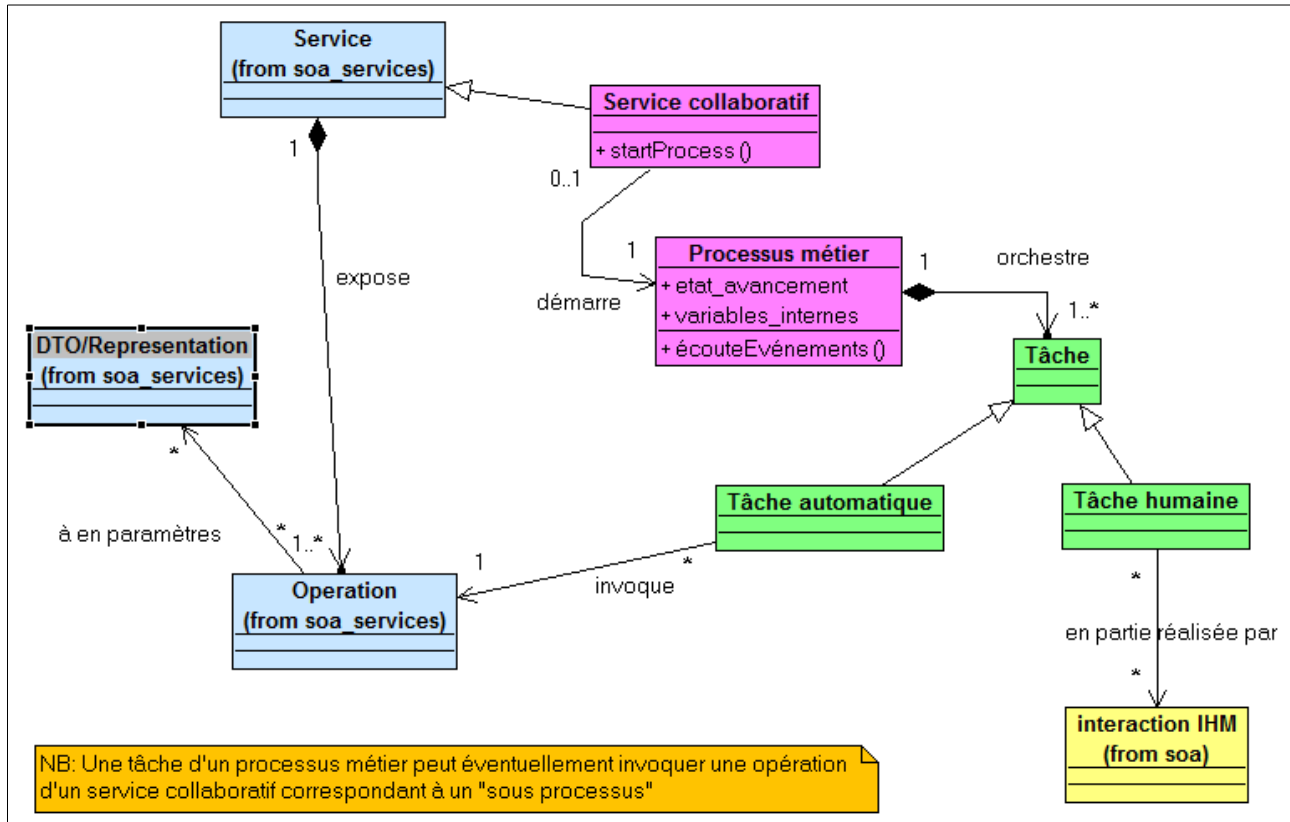
Via des intercepteurs et des logs générés par les ESB et/ou serveurs d'applications on peut mesurer l'activité "SOA" d'une partie du SI de façon à :

- localiser d'éventuels goulets d'étranglement
- anticiper les montées en charge
- ...

XI - Processus long, orchestration asynchrone

1. Dualité "services avec orchestration de sous-services" et "processus"

Méta-modèle pour service collaboratif (de haut niveau) :



NB : Un processus métier peut éventuellement :

1. être modélisé en UML (et/ou BPMN)
2. être implémenté en XML/BPEL (ou bien en Java/jBpm , ...)
3. être vu comme un service collaboratif (orchestrant différents services/tâches élémentaires)
4. être invoquée comme un service web
5. être (en partie) contrôlé par une IHM (web ou ...) invoquant ce service.

2. "processus métier vers processus exécutable"

2.1. Traçabilité souhaitée

Le principal intérêt des technologies d'orchestration (ex : BPEL , bpmn2 + java , ...) réside dans la traçabilité généralement rencontrée entre les éléments du modèle métier (processus bpmn abstrait) et les éléments du code exécutable .

Autrement dit , si la version N+1 évolue par "suppressions , modifications , ajouts partiels" au sein du modèle métier abstrait , on peut rapidement identifier les parties à mettre à jour au sein du code exécutable .

Sans technologie d'orchestration , on a souvent besoin d'éléments spécifiques éparpillés au 4 coins du code de l'application (colonne d'une table Xy , questionnaire événementiel Zz , ...) et la traçabilité n'est pas évidente.

2.2. Partie exécutable d'un processus

Un processus métier (abstrait) modélisé avec BPMN comporte souvent plusieurs couloirs ("pools") . Par exemple : "client" , "fournisseur" , "logistique" .

Ceci permet de bien montrer certaines collaborations attendues entre partenaires (chorégraphie) .

A l'inverse, **une technologie d'orchestration** (ex : BPEL ou bien bpmn2+java) **ne peut exécuter le code que d'un seul couloir ("pool")** à éventuellement marquer comme **"exécutable"** (selon la technologie) .

En fonction de la technologie qui interprète le processus (bpel , bpmn2, ...) , les éventuels autres pools seront :

- soit ignorés
- soit utilisés comme base pour la définition des partenariats
- soit "à enlever" pour éviter des erreurs
- ...

Un fichier "bpmn" (décrivant un processus métier abstrait) a donc généralement besoin d'être :

- versionné
- repris (par copie) et adapté à la technologie d'interprétation.

Bonnes pratiques :

- décomposer (dès la modélisation abstraite) un processus en sous-processus (via call-activity)
- environ 60 % des processus modélisés ne resteront que des modèles BPMN (faisant partie des spécifications de chorégraphie à lire , ré-interpréter)
- environ 40 % des processus (ou sous-processus) pourront (partiellement) être rendus exécutables (si la traçabilité est appropriée / souhaitée ou bien si certaines parties à piloter sont asynchrones) .

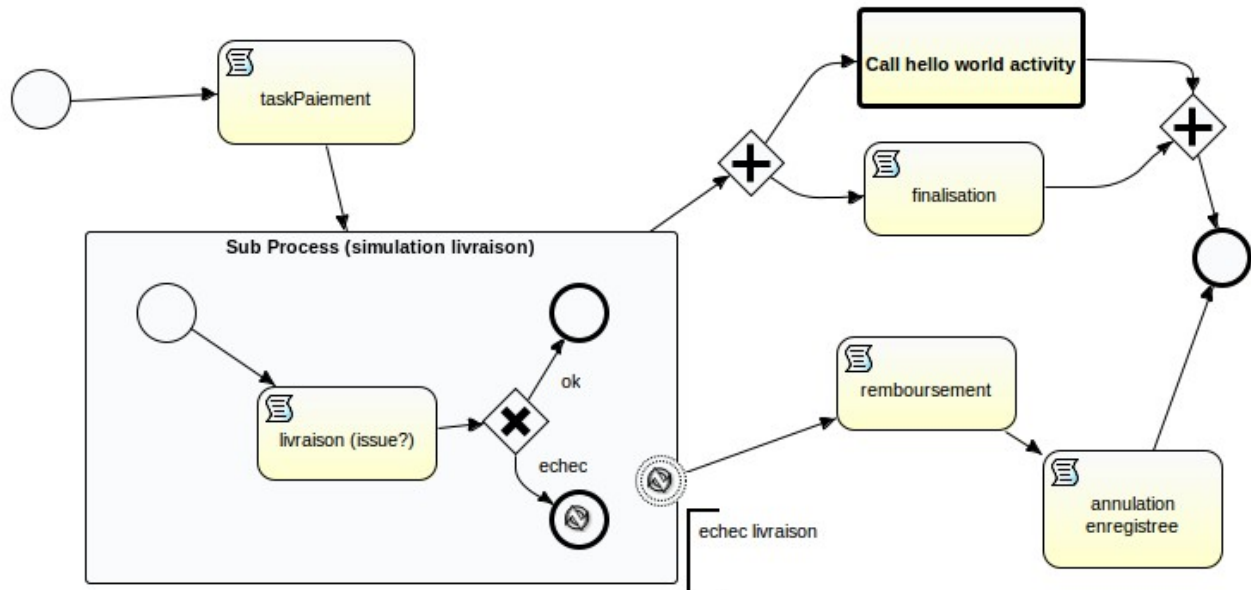
2.3. Fiabilisation des processus

Pour rester "maintenable", un processus exécutable doit idéalement rester "petit" et "simple" d'un point de vue métier.

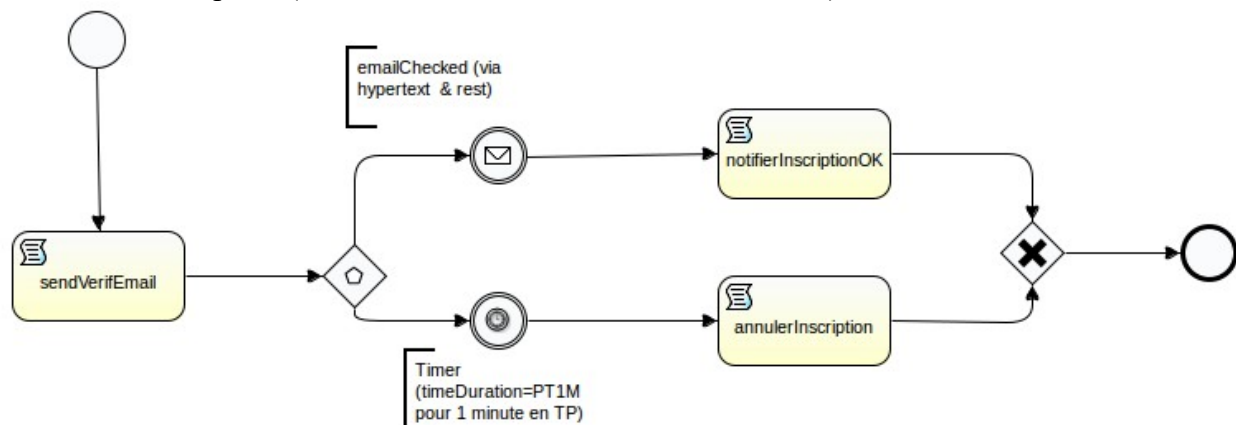
Sachant qu'en version "exécutable", il faut généralement ajouter des gestionnaires d'erreurs pour fiabiliser tous les cheminements.

2.4. Exemples simples :

Attente de l'issue d'un sous-processus de livraison et éventuel remboursement :



Processus d'inscription (avec attente d'une "validation de l'e-mail") :



XII - SOA réaliste selon moyens

1. SOA bien dimensionné (selon moyens et besoins)

1.1. Prix relativement élevé de l'infrastructure SOA

Mettre en œuvre une architecture "SOA" nécessite les différents éléments suivants (dont les coûts s'additionnent) :

- **Analyse / modélisation / organisation (UML , BPMN , urbanisation)** avec gestion des évolutions (versions cohérentes , ...)
- Achat/location d'infrastructures matérielles (ordinateurs , machines virtuelles)
- Achat/location de serveurs logiciels (Serveurs d'applications JEE , **ESB** , **Moteur BPM**, ...)
- Formation ou auto-apprentissage , administration / configurations.
- **Outils de modélisation/développement (avec assistants spécifiques "bpel" ou "bpmn")**
- Développement (java , xml , ...) , tests unitaires
- **Tests d'intégration**
- Maintenance
-

Par rapport à une architecture plus simple , le surcoût de l'infrastructure SOA se situe essentiellement au niveau des points mentionnés ci-dessus en caractères gras.
Certains "ESB commerciaux" coûtent environ 50000 euros !

Attention :

Une technologie qui automatise beaucoup (ex : ESB , moteur BPM , framework complexe,) permet de gagner en "nombre de lignes à coder" mais nécessite plus de "tests" pour être bien fiabilisée. D'autre part , la lente compréhension de certains mécanismes complexes peut rendre les paramétrages délicats à optimiser/retoucher.

1.2. R.O.I. selon bénéfices métiers/fonctionnels

L'infrastructure SOA (potentiellement lourde) trouvera un retour sur investissement qu'en fonction :

- des bénéfices métiers/fonctionnels (éviter les re-saisies , meilleur agilité / réactivité , ...)
- d'un bon choix et dimensionnement des éléments de l'architecture (ESB commerciaux ou bien "open source" ,)
- ...

Comment souvent, au fil des années,

- les technologies lourdes (trop complexes) finissent par être abandonnées (ex : JBI)
- de nouvelles technologies (plus simples , mieux intégrées, ...) font baisser les coûts.

Avec un petit budget (à maîtriser) :

- Web services "SOAP" et "REST"
- ESB "open source" avec paramétrages restants simples
- Technologie d'orchestration légère (ex : bpmn2 + activiti)
- Modélisation / urbanisation légère
- Etudier si possible des hébergements "cloud" .

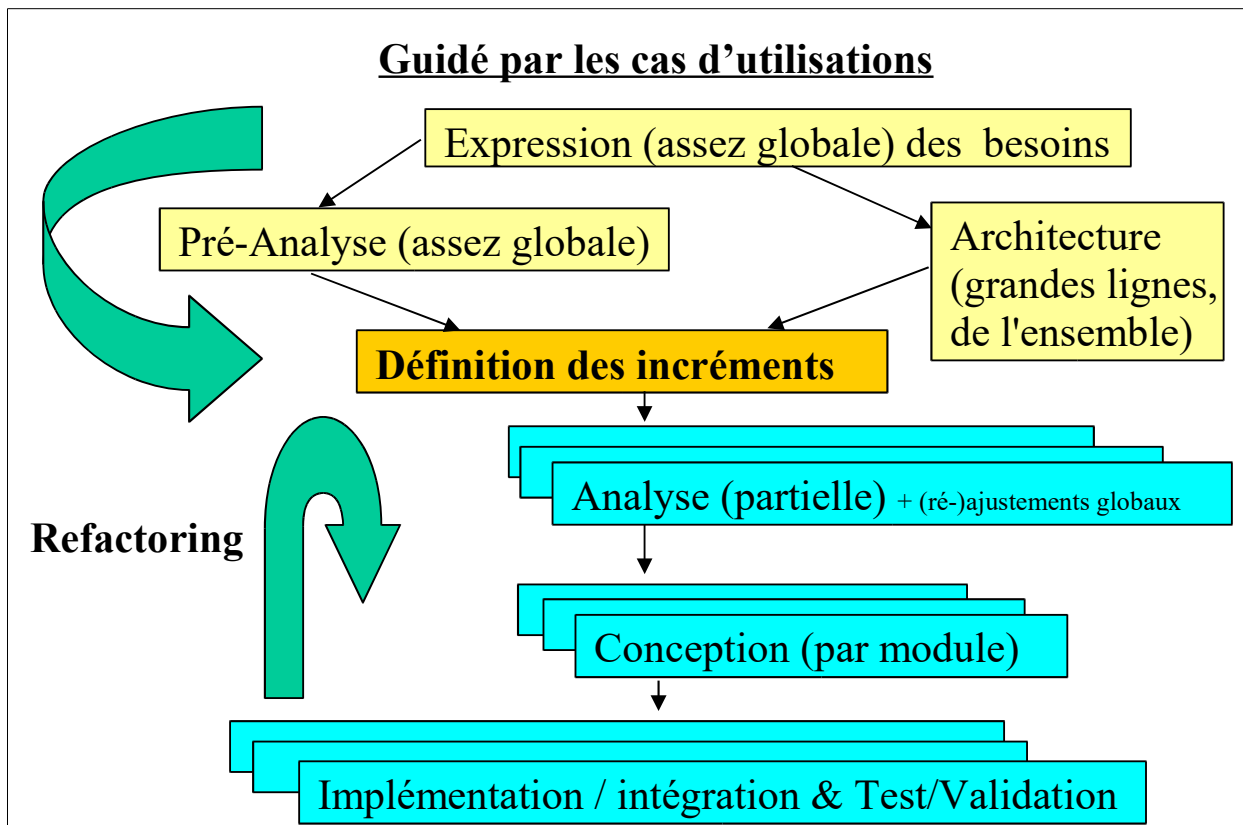
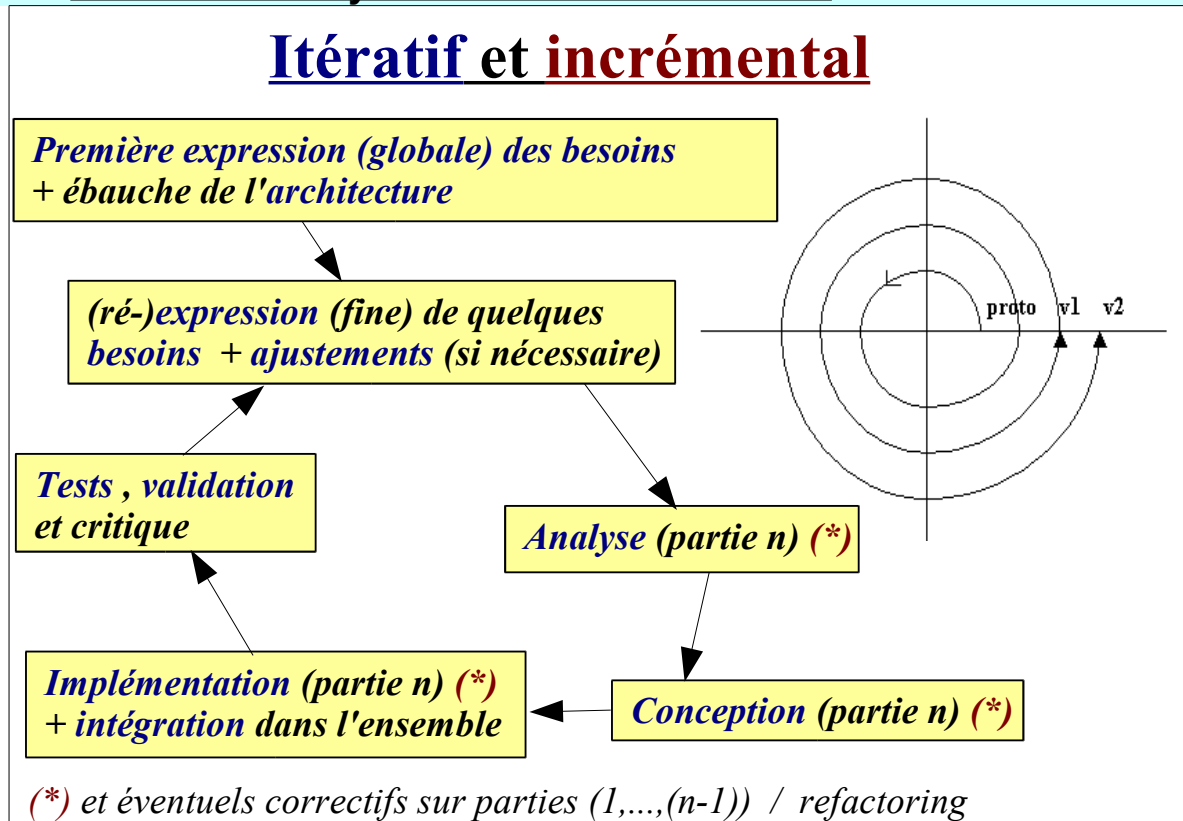
Pour entreprendre de grandes choses (objectifs ambitieux) :

- Technologies évoluées (fiables , pas trop limitées , ...) avec support
- Techniciens qui maîtrisent les technologies
- Budget conséquent (à priori pas pour petite entreprise)
- Cellule de modélisation / urbanisation
- Infrastructure suffisante (bien dimensionnée) pour ne pas rencontrer de problème de performance.
- Bonne organisation (gestion des changements , évolutions convergentes , tests , ...)

Le **cadre de capacité** de "Togaf" met par exemple en avant l'organisation nécessaire pour mener à bien un gros projet d'entreprise .

2. Cycle de vie d'un projet SOA

2.1. Besoins d'un cycle itératif et incrémental



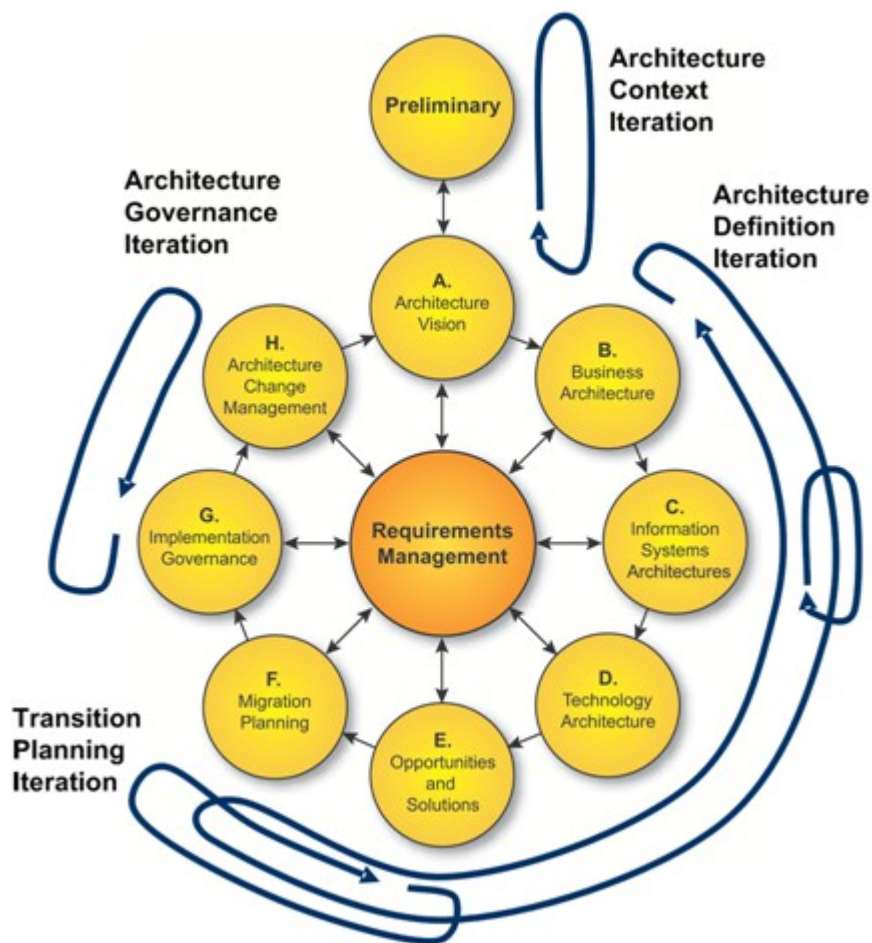
SOA étant lié à une **grande portée** (plusieurs partie du SI , extanet ,) , il est conseillé d'adopter une **démarche** à la fois **pragmatique** (en s'inspirant partiellement des méthodes agiles XP , SCRUM , ...) et à la fois **bien formalisée** (style UP).

Autrement dit des tests et des commentaires dans le code ne suffisent pas.

Il faut absolument définir(et versionner) :

- des **référentiels** de modèles UML et BPMN et de formats internes (XSD/WSDL,...)
- maintenir à jour des **documentations/spécifications précises** .

L'aspect "itératif" (à grande échelle) de SOA est clairement mise en avant dans le cycle "ADM" de "togaf" :



2.2. Besoins de transversalité

Contrairement à un projet applicatif à portée réduite , une bonne modélisation SOA (ayant des ambitions sur le moyen/long terme) doit être **transverse** .

Concrètement, ceci consiste à :

- recueillir les *besoins "soa" précis (échanges extérieurs , structures de données)* de chaque partie concernée du S.I.
- *faire ressortir des structures communes (via gouvernance , ...)*
- *identifier les besoins en "intégrité référentielle" (MDM, ...)*
- ...

2.3. exemple de cycle de vie adapté "soa"

Phase de mise au point (version 1) :

1. Effectuer une *pré-étude SOA (assez globale)*
2. *Identifier une sous partie non critique du SI* et mettre en place un *premier prototype* de l'*architecture SOA* (modèles , référentiels , ESB , services ,)
3. Effectuer (si besoin) les *réajustements nécessaires*

Phases d'intégration successives (versions "2,3,..., n") :

1. *Parfaire* si besoin l'*étude SOA globale* (en tenant compte des changements et des ajouts)
2. *Identifier une sous partie "à besoin SOA prioritaire" du SI*
3. *Intégrer* cette partie fonctionnelle dans l'architecture SOA existante en effectuant toutes les *restructurations* à priori nécessaires.
4. Effectuer (si besoin) les *réajustements* à posteriori nécessaires

3. Activités de modélisation SOA (enchaînement)

- 1) Faire (ou parfaire) un *"recueil textuel des besoins (transverses)"* et une *analyse sommaire de l'existant*.
- 2) Faire (ou parfaire) une *étude d'urbanisation* et éventuellement une *analyse sémantique* de façon à obtenir un *découpage en packages/modules (zones/secteurs/...) significatifs*.
- 3) Modéliser et/ou ajuster les *structures de données partagées "de référence"* (structure , cycle de vie)
- 4) Retranscrire/reformuler certaines "expression de besoins" et "objectifs métiers" en *"(business) Uses Cases"* dont les scénarios pourront être illustrés par des diagrammes d'activités (processus métier).
Modéliser (en UML ou BPMN) les principaux "processus métier" (et repérer les besoins en services)
- 5) Modéliser les *services métiers* (opérations, requêtes/réponses/exceptions, ...) et identifier/étudier les besoins en *"adaptateurs"* pour les *intégrations*.
- 6) Dériver les éléments de la modélisation en *"structures et contrats SOA" (XSD , WSDL, ...)* et alimenter les référentiels SOA (... ? , Annuaire interne ? , ...).
- 7) **Programmer** toutes les parties nécessaires (java , **web services** , **jbpm** ou **bpel** , **adaptateurs**, *intercepteurs (log, sécurité,...)* ,)
- 8) Tester et ajuster le tout , parfaire la documentation ,

...

ANNEXES

XIII - Annexe – Eléments clefs – UML

1. Vue d'ensemble sur UML

Les principaux diagrammes d'UML

Comportementaux (behavior)

fonctionnel

Diag. Uses cases

Diag. d'activités

Dynamique (interactions, ...)

Diag. d'états

Diag. de séquences

Diag. de collaboration / communication

Statique / structurel

logique

Diag. de classes

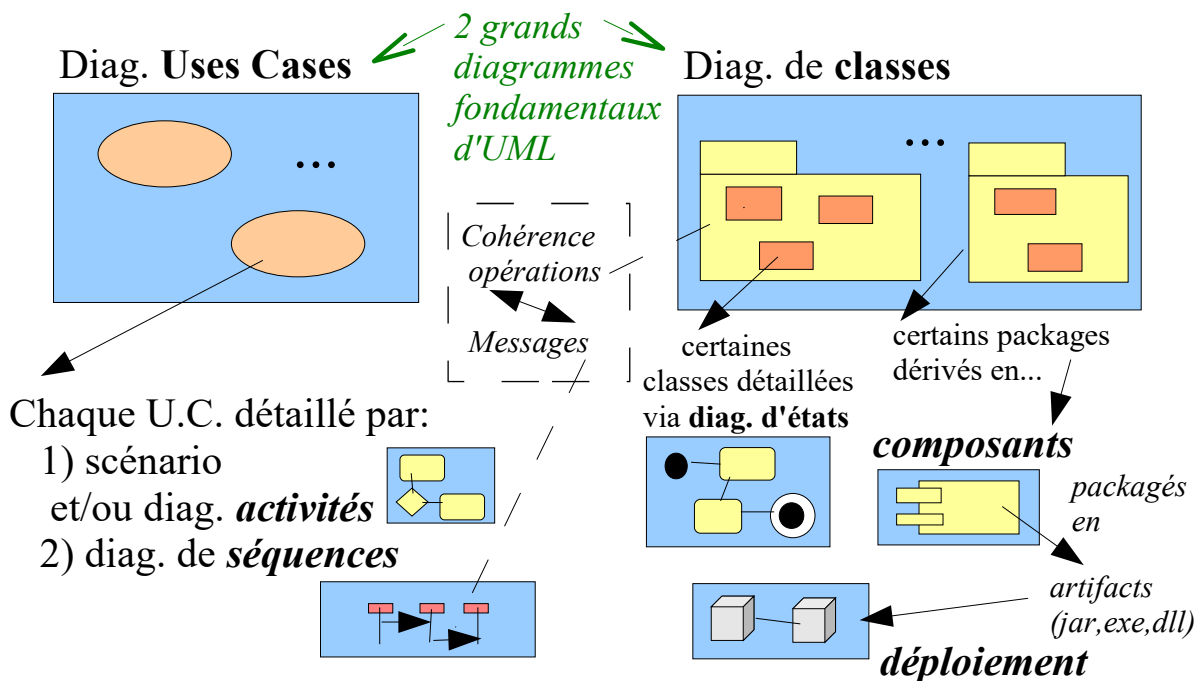
Diag. d'instances (rare)

Physique/ implémentation

Diag. de composants

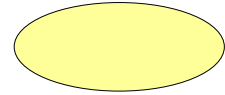
Diag. de déploiement

Principaux liens entre les diagrammes UML



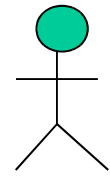
2. Diagramme des cas d'utilisations

Présentation des « Use Case »



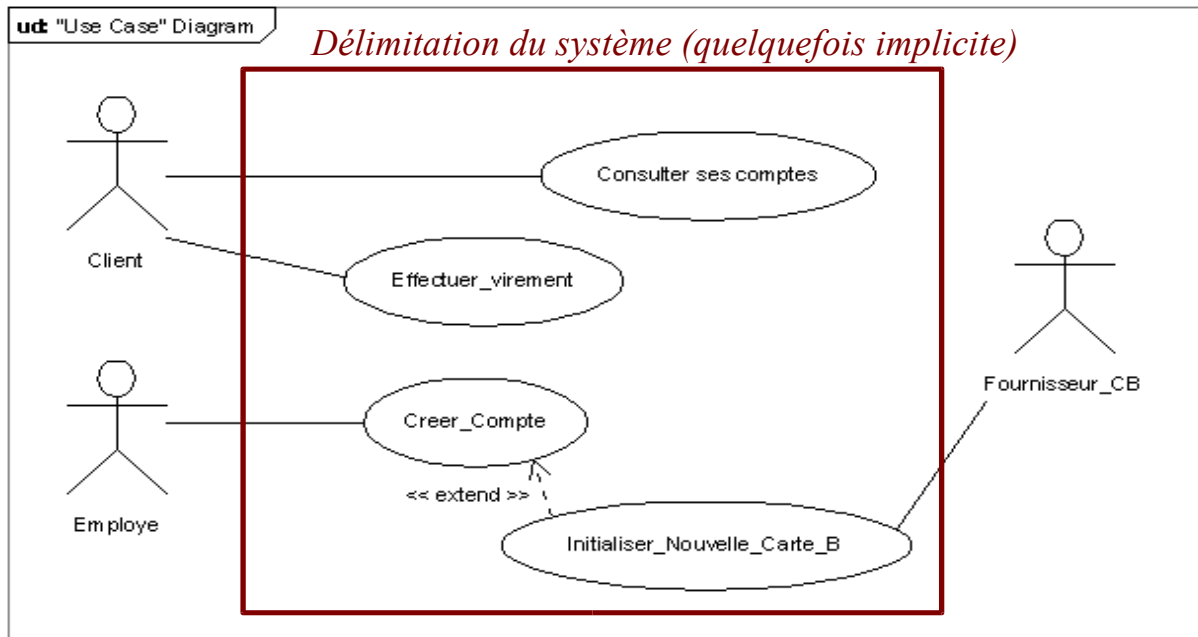
Faisant partie intégrante de UML, les "USE CASE" offrent un **formalisme** permettant de:

- **Délimiter** (implicitement) **le système** à concevoir.
- Préciser les **acteurs extérieurs** au système.
- Identifier et clarifier les **fonctionnalités** du futur système.



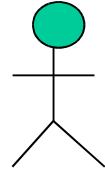
Nb: il s'agit d'une **vue externe** (point de vue de l'utilisateur).

Diagramme U.C. (Vue d'ensemble)



Acteurs

- Un **acteur** est une entité extérieure au système qui interagit d'une certaine façon avec le système en jouant un certain **rôle**.
- Un acteur ne correspond pas forcément à une catégorie de personnes physiques .
Un automate quelconque (Serveur, tâche de fond , ...) peut être considéré comme un acteur s'il est extérieur au système.
==> Deux grands types d'acteurs (éventuels stéréotypes) : **"Role_Utilisateur"** ,
"Système_Externe"



Acteurs primaire et secondaire

- Un cas d'utilisation peut être associé à 2 sortes d'acteurs:
 - L'unique acteur primaire (principal)** qui déclenche le cas d'utilisation. **C'est à lui que le service est rendu.**
 - Les éventuels **acteurs secondaires** qui **participent** au cas d'utilisation en apportant une aide *quelconque (ceux-ci sont sollicités par le système)*.



Cas d'utilisation

- Définition: *Un cas d'utilisation (use case) est une fonctionnalité remplie par le système et qui se manifeste par un ensemble de messages échangés entre le système et un ou plusieurs acteur(s).*

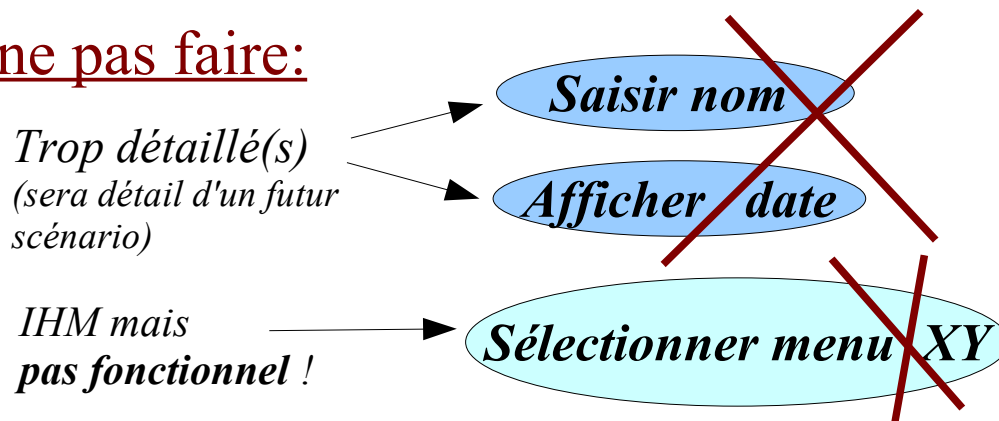
Notation:



A ne pas faire:

*Trop détaillé(s)
(sera détail d'un futur
scénario)*

*IHM mais
pas fonctionnel !*

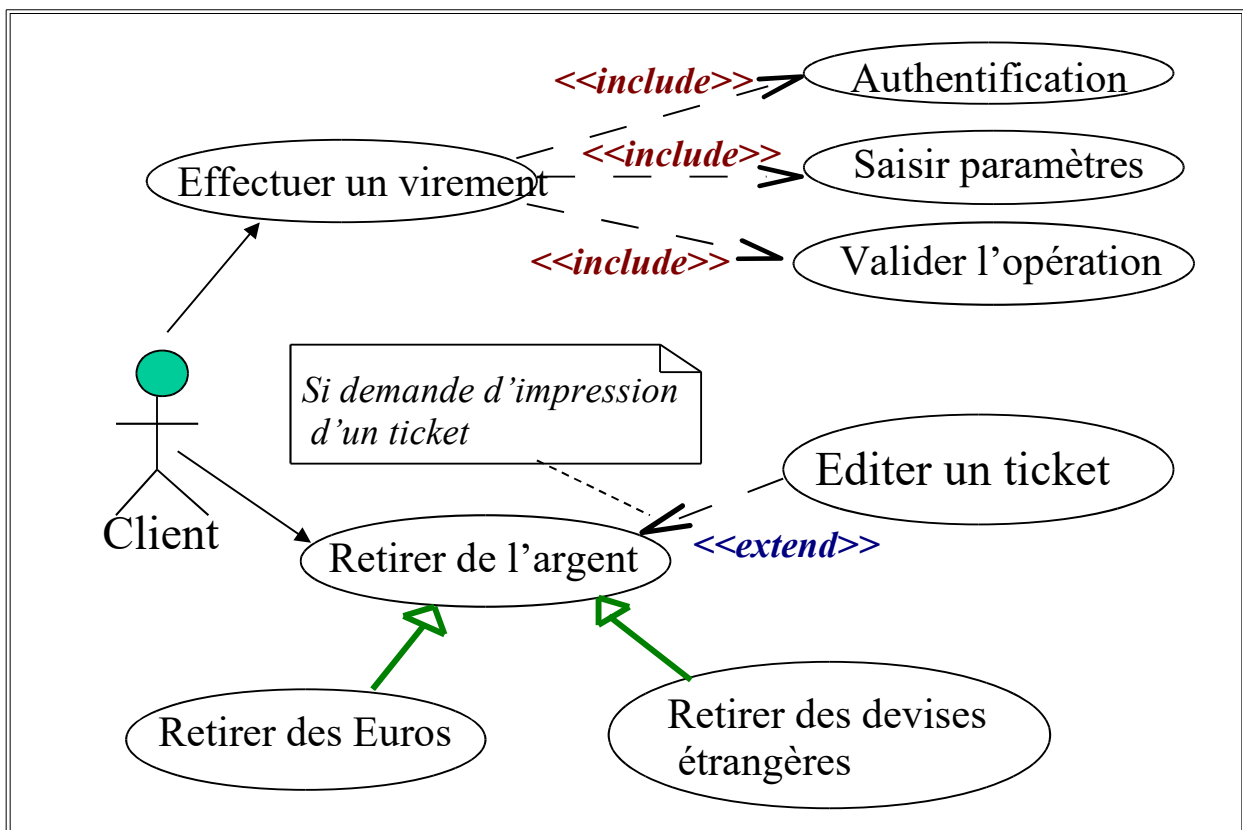


A prendre en compte:

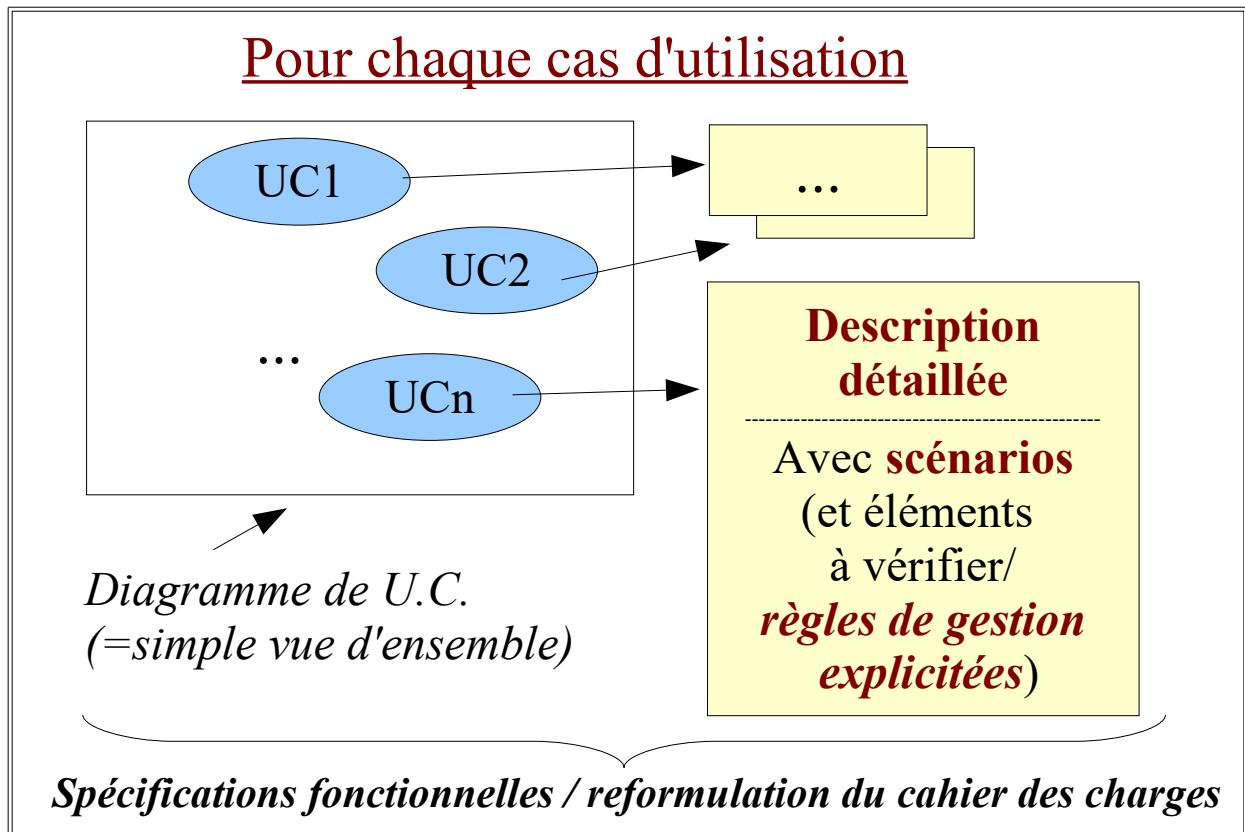
- * cas d'utilisation <--> session utilisateur
--> On peut donc relier entre eux les U.C. Effectués à peu près au même moment mais on doit séparer ce qui s'effectue à des instants éloignés (différentes sessions)
- * cas d'utilisation --> avec interaction avec l'extérieur .

Relations entre UC

- Le cas d'utilisation A **<<include>>** le (sous-) cas d'utilisation B si **B est une sous partie constante (systématique) de A.**
- Le cas d'utilisation (supplémentaire) C **<<extend>>** le cas d'utilisation A si **C est une partie additionnelle qui s'ajoute à A le cas échéant (facultativement).**
Une **note (commentaire) spéciale** appelée « **cas d'extension** » permet de préciser le cas où C étend A.
- La **relation d'héritage** (ou généralisation) classique: **D \rightarrow E** permet d'exprimer que le cas d'utilisation D est une **sorte (variante, déclinaison)** du cas d'utilisation E.



3. Scénarios et descriptions détaillées (U.C.)



Description textuelle (U.C.)

Bien que la norme UML n'impose rien à ce sujet, l'**usage** consiste à documenter chaque cas d'utilisation par un texte (word , html, ...) comportant les rubriques suivantes:

- **Titre** (et éventuelle numérotation)
- **Résumé** (description sommaire)
- Les **acteurs** (primaire, secondaire(s) , rôles décrits précisément)
- **Pré-condition(s)**
- **Description détaillée (scénario nominal)**
- **Exceptions** (scénario pour cas d'erreur)
- **Post-condition(s)**

Scénarios (U.C.)

Scénario nominal:

- 1) l'utilisateur place la carte dans le lecteur
- 2) l'utilisateur renseigne son code secret
- 3) le système authentifie et identifie l'utilisateur
- 4) l'utilisateur sélectionne le montant à retirer
- 5) le système déclenche la transaction (débit du compte de l'utilisateur)
- 6) le système rend la carte à l'utilisateur
- 7) le système distribue les billets et un éventuel ticket

Scénario Nominal

Tout se passe bien

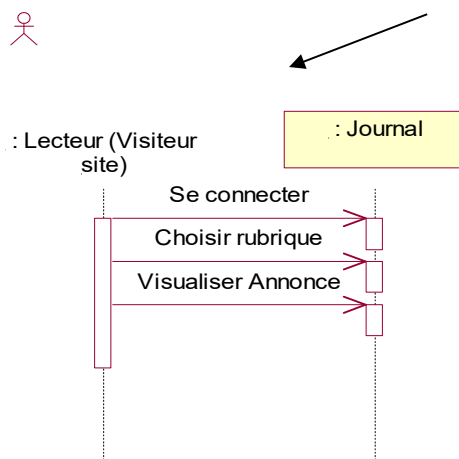
Cas/Déroulement le plus fréquent

Scénario d'exception "E1":

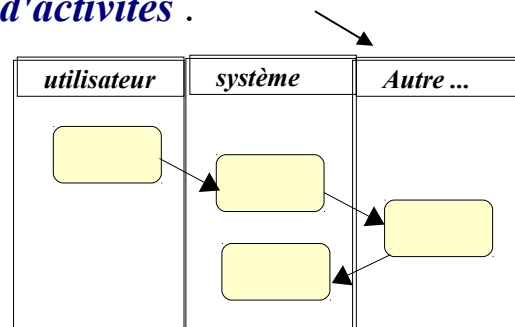
- si l'étape (3) échoue trois fois de suite alors
- avaler la carte
 - ne pas effectuer les étapes (4) à (7)
 - afficher un message explicatif

Illustration éventuelle (U.C.)

Un scénario peut éventuellement être graphiquement exprimé/illustré par un **diagramme de séquence UML**.



Un ou plusieurs scénario(s) peuvent également être exprimés à travers un **diagramme d'activités**.



Avec **décision(s)** :

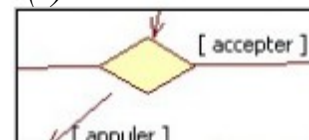


Tableau récapitulatif des U.C. (base pour la planification)

Use Case	Estimation charge (j/h)	Priorité	Autres caractéristiques
UC_1	6	++	techniquement difficile
UC_2	8	--	facile
UC_3	5	+	...
UC_n	6	-

NB: estimation charge = (modélisation + implémentation + tests + intégration)
 Priorité en partie selon <<include>> (+,++) et <<extend>> (-,--)

4. Diagramme d'activités

Diagramme d'activités

Un **diagramme d'activités** montre les **activités effectuées séquentiellement ou de façon concurrente** par un ou plusieurs éléments (acteur, personne, objet).

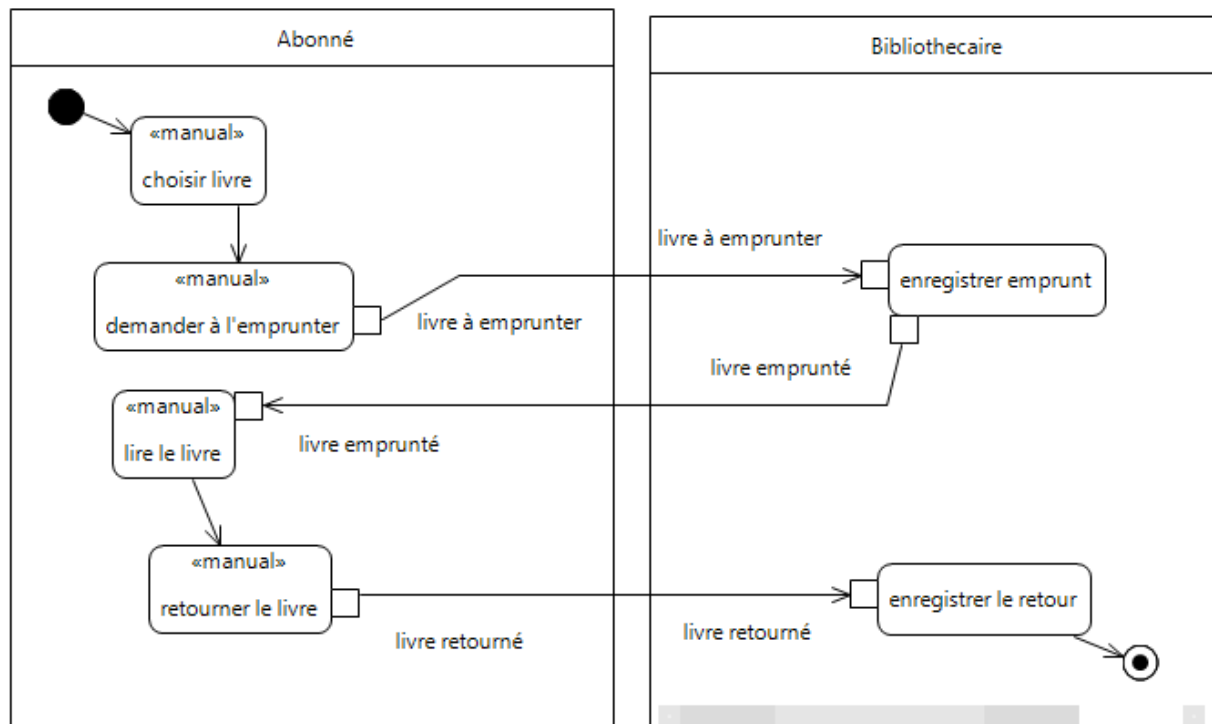
Principales utilisations:

- **Workflow** et **processus métier**.
- **Organigramme** (pour algorithme complexe).

4.1. couloirs d'activités (facultatifs)

Ces **couloirs** (sous forme de colonnes) sont quelquefois appelé "**partitions**" ou encore " *swimlane* / *lignes d'eau*" et permettent d'indiquer "**qui fait quoi**" :

emprunter des livres pour les lire



Au sein de l'exemple ci dessus le stéréotype `<<manual>>` (non normalisé) peut aider à préciser si une tâche/action est plutôt manuelle (ou informatisée sinon) et les petits rectangles blancs sur les bords des actions correspondent à des "output pin" et "input pin" associés à des "object flows".

4.2. Noeuds spéciaux et de contrôles



Initial/début (normalement unique)



fin complète (de toutes les branches du processus)

En général : une fin ordinaire (atteinte de l'objectif) et d'éventuelles "fin" de type "annulation".

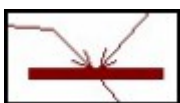


fin de flot/branche (ex : fin d'exécution d'un thread ou d'une tâche parallèle)

Barre de synchronisation avec une entrée et plusieurs sorties :



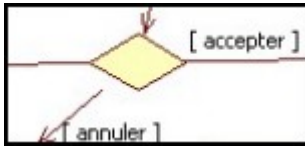
bifurcation (traitement en //)



union/synchronisation de type "et logique"

Pour modéliser un "ou" --> plusieurs transitions entrantes vers une même activité (sans barre de synchronisation).

Décision et *condition de garde* entre [] :



4.3. nœuds d'activités et variantes (actions, ...)

Un diagramme d'activités UML (activity group) est un graphe dont la plupart des nœuds sont des nœuds d'activités/actions (correspondants à des traitements ou des tâches).

UML2 distingue plusieurs types de nœuds d'activités (ou d'action) selon la granularité et la nature des traitements à modéliser.

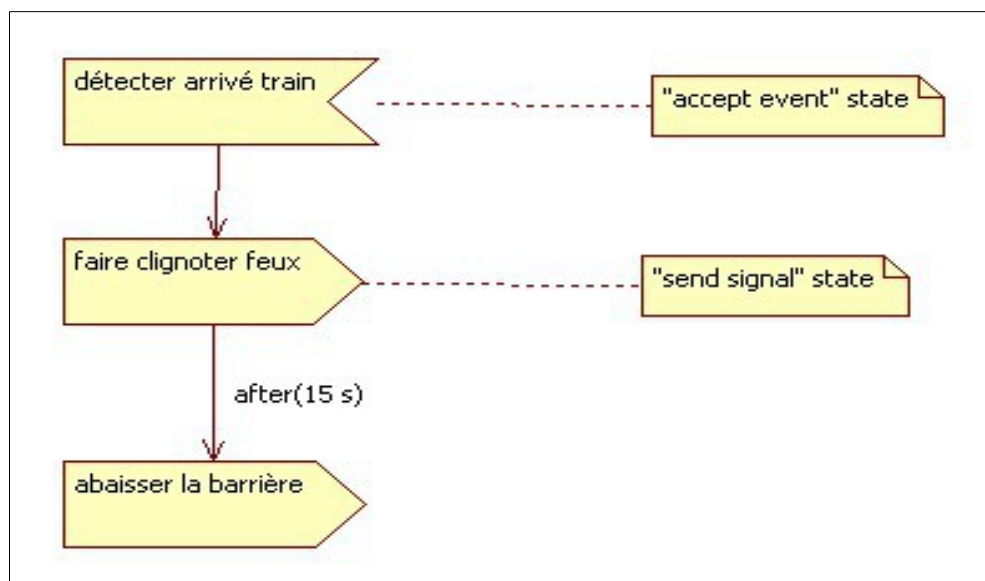
Types d'actions/activités souvent utiles en **expressions des besoins** :

Opaque Action	Action/activité quelconque (dont la nature pourra être ultérieurement précisée/affinée en conception)
Call behaviour	Appel global d'une autre sous activité (sans mentionner une opération précise). Souvent associé à un lien hypertexte vers autre diagramme.

Différents types très (trop) précis d'actions (en général pour la conception):

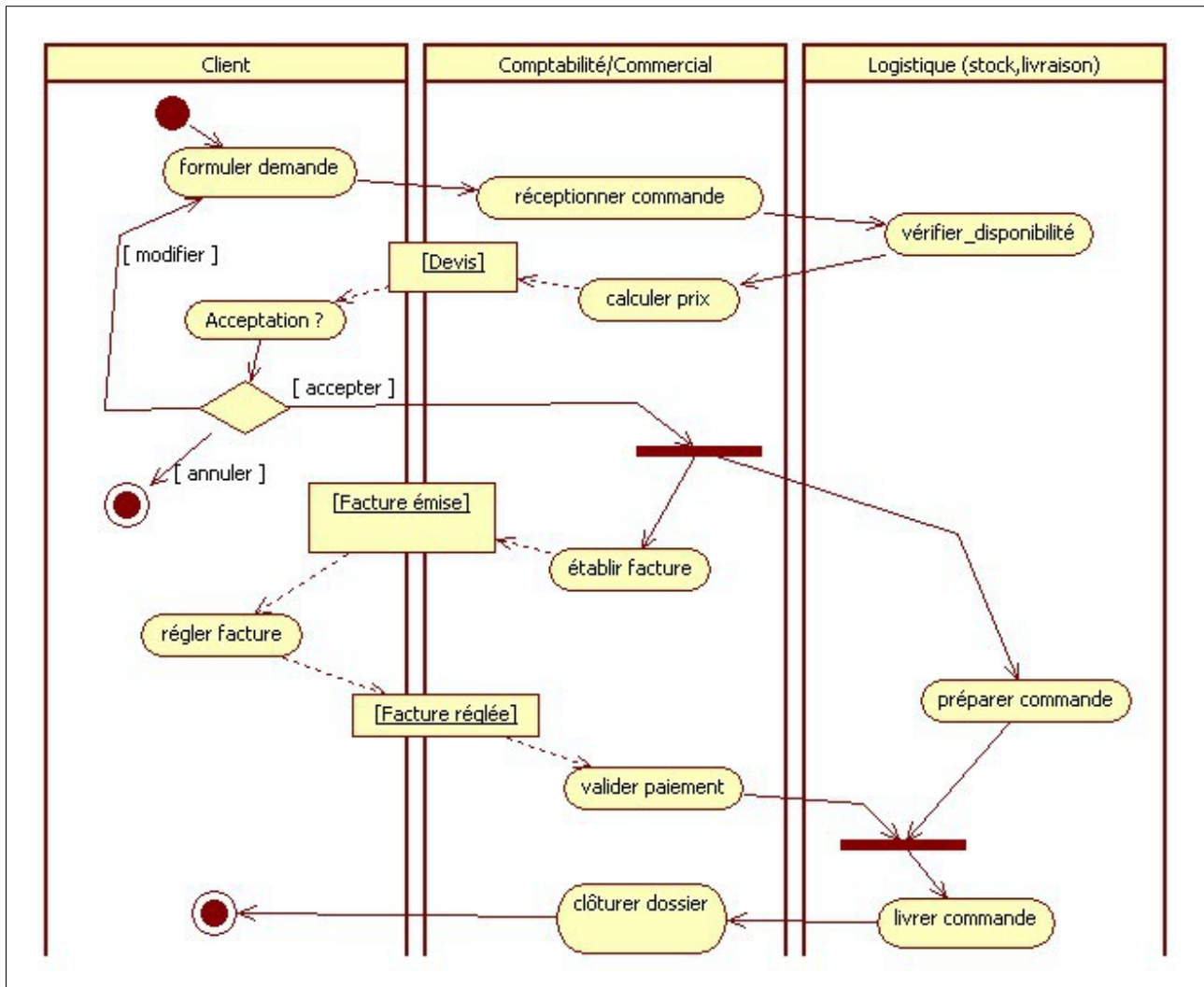
Call operation	Appel (en mode souvent synchrone ou rarement asynchrone) d'une méthode/opération avec passage possible de paramètre et récupération potentielle d'une valeur de retour
Send	Envoi d'un message ou d'un signal
Accept event	Attente (bloquante) d'un événement (souvent lié à une notification asynchrone)
Accept call	Variante de "accept_event" pour les appels entrants (avec réponse à donner ultérieurement via reply)
Reply	Répondre (lié à un accept_call)
créer/instancier	Créer un nouvel objet
destroy	Détruire un objet (ex: delete en c++)
Raise exception	Soulever (remonter) une exception

4.4. Notations pour actions particulières (UML2)



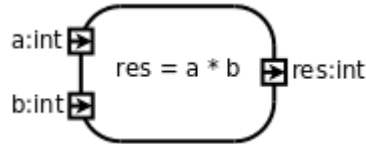
4.5. object flow (nœuds "objet") : UML 1 et 2

Un nœud "objet" correspond à un message (ou flot de données) construit par une activité préalable et qui sera souvent acheminé en entrée d'une autre activité (exemples: lettre , devis , facture , mail ,).

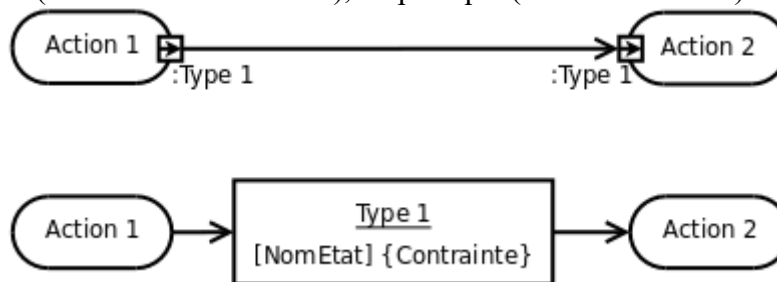


4.6. Pins et buffers (UML 2)

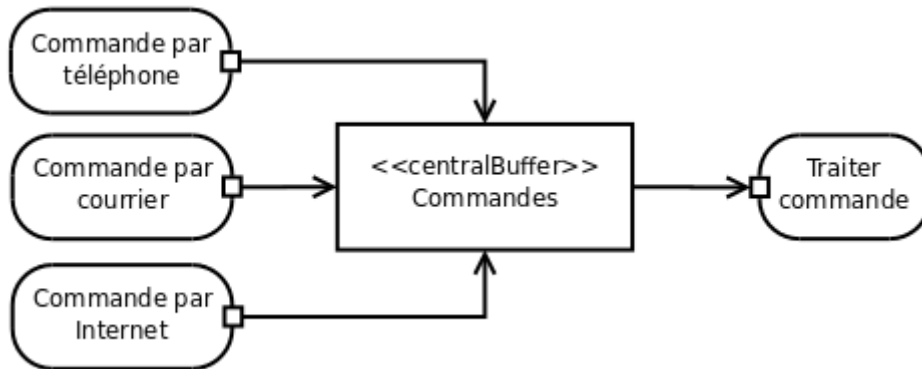
Depuis la version 2 d'UML, il faut placer des points ("pin") d'entrée ou de sortie sur une activité pour préciser un lien (éventuellement typé) avec les "object flow" entrant(s) ou sortant(s).
[avec une sémantique de passage de valeur(s) par copie(s)] .



2 notations possibles (en théorie avec UML2), en pratique (selon outil UML):



En UML2, un éventuel nœud intermédiaire de type <<centralBuffer>> peut être placé pour bufferiser des messages à acheminer ensuite ailleurs.

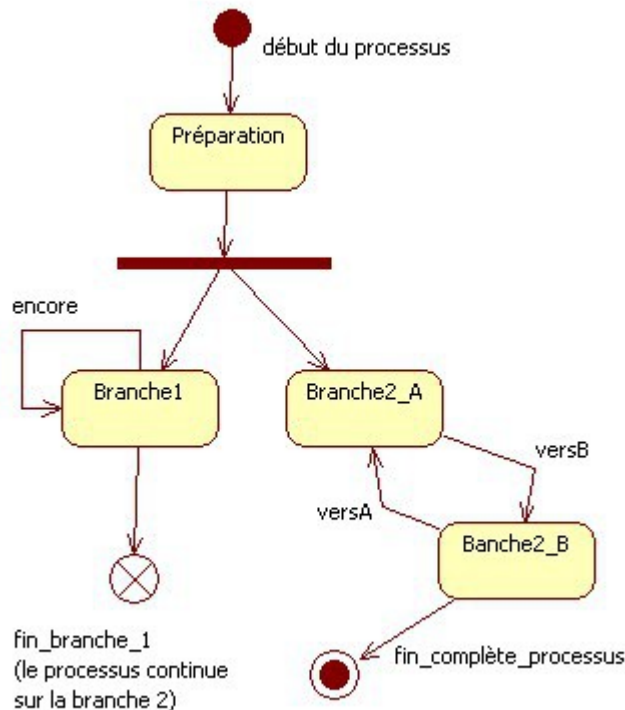


NB: Lorsque le buffer/tampon intermédiaire gère en outre la persistance des données (dans une base de données ou ...) , on utilise alors <<dataStore>> plutôt que <<centralBuffer>> .

4.7. Final flow

Final Flow = Arrêt du flux sur une branche (une autre branche peut éventuellement continuer)

Exemple :



5. Distinction "micro activité / macro activité" selon granularité

Un **diagramme d'activités UML** peut servir à modéliser des choses à des échelles (ou niveaux de granularité) assez variables:

- **macros activités :**
 - *durées* potentiellement *longues* (plusieurs jours)
 - avec souvent pleins d'intervenants (ex: client , logistique , fournisseurs ,)
 - *liées à des "buts/objectifs métiers" représentés via des "business uses cases"*
 - ...
- **micros activités (actions)**
 - *durées limitées* (session utilisateur , quelques minutes)
 - centrées sur un intervenant principal (acteur primaire)
 - *associées à des cas d'utilisations (uses cases)*
 - ...

6. Diagramme de classes (notations , ...)

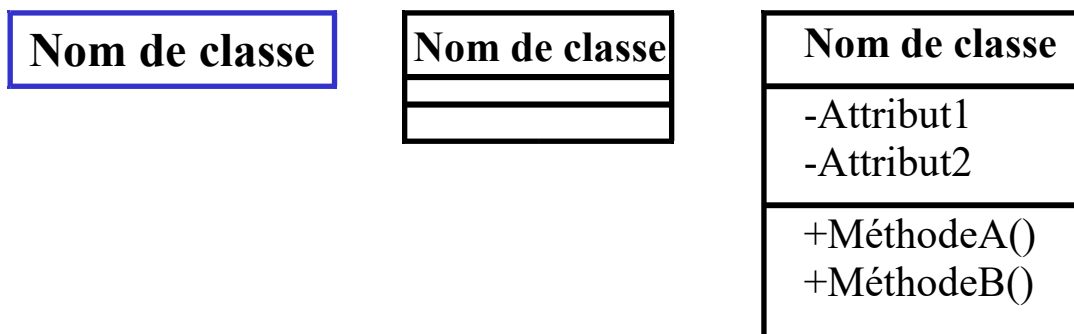
Diagramme de classes (*modèle structurel*)

- Le **modèle statique/structurel** (basé sur les diagrammes de classes et d'instances) permet de décrire la **structure interne du système** (entités existantes, relations entre les différentes parties, ...).
- Tout ce qui est décrit dans le diagramme de classes **doit être vrai tout le temps**, il faut raisonner en terme d'**invariant**.
- Le **diagramme de classes** est le plus important, il représente l'**essentiel de la modélisation objet** (c'est à partir de ce diagramme que l'essentiel du code sera plus tard généré).

Les classes (représentations UML)

Une **classe** représente un **ensemble d'objets** qui ont :

- une **même structure** (*attributs*)
- un **même comportement** (*opérations*)
- les **mêmes collaborations avec d'autres objets** (*relations*)

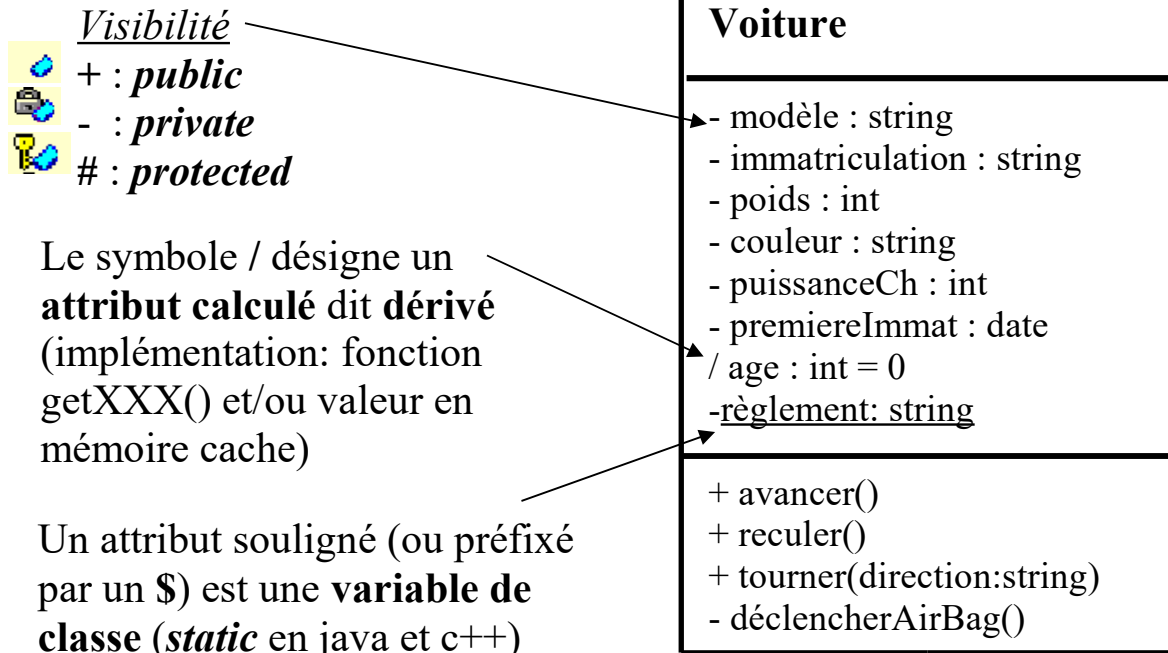


Attribut alias **Propriété/Property**

,

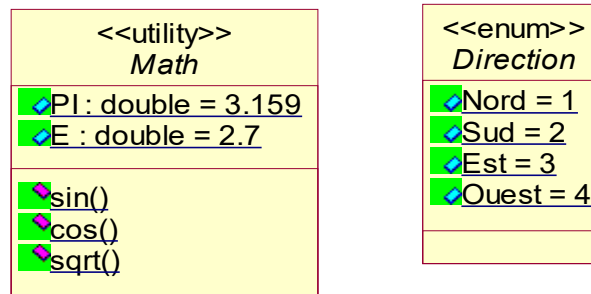
Méthode alias **Opération** .

Détails sur les éléments d'une classe



Classes spéciales (utilitaires, énumération, ...)

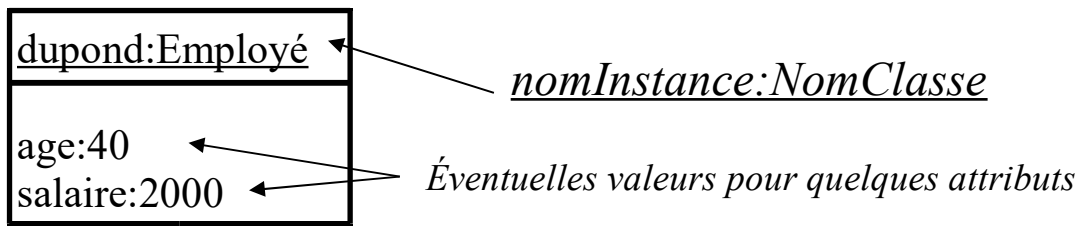
- Dans un monde matériel et rationnel ou tout est objet, il n'y a plus de place pour des fonctions globales (anarchiques).
Celles-ci doivent être rangées dans des classes utilitaires.
- Les constantes doivent elles aussi être placées dans des classes (ou interfaces) d'énumération.



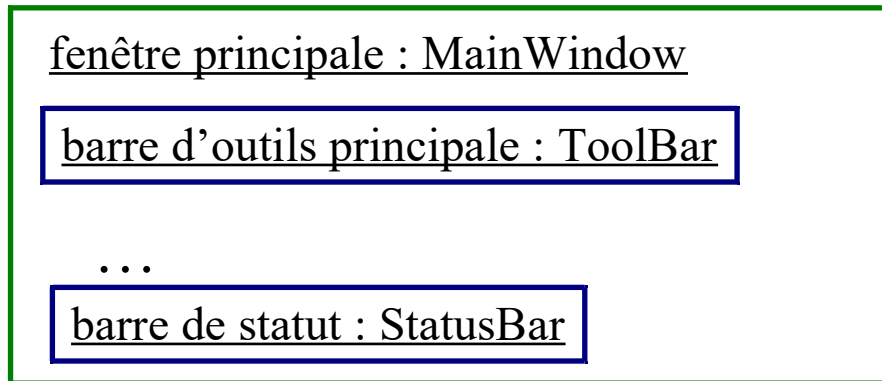
y = **Math.sqrt**(x);

obj.setDirection(**Direction.Nord**);

Eventuels (et rares) diagrammes d'instances



Instance composée (de sous objets):



Associations (relations)

Dans le cas le plus simple une **association** est **binaire** et est indiquée par un *trait reliant deux entités (classes)*. Cette association comporte généralement un nom (souvent un verbe à l'infinitif) qui doit clairement indiquer la signification de la relation. *Une association est par défaut bidirectionnelle.*



Dans le cas où le sens de lecture peut être ambigu, on peut l'indiquer via un triangle ou bien par le symbole `>`



Extrémités d'association

- Une **association** n'appartient pas à une classe mais à un package (celui qui englobe le diagramme de classe).
- On peut préciser des caractéristiques d'une association qui sont liées à une de ses **extrémités (association end)**:
 - *rôle* joué par un objet dans l'association
 - *multiplicité*
 - *navigabilité*
 - ...

Multiplicité UML (cardinalité)

1	exactement un
0..* ou *	plusieurs (éventuellement zéro)
0..1	zéro ou un
n (ex: 2)	exactement n (ex: 2)
1..*	un à plusieurs (au moins 1)

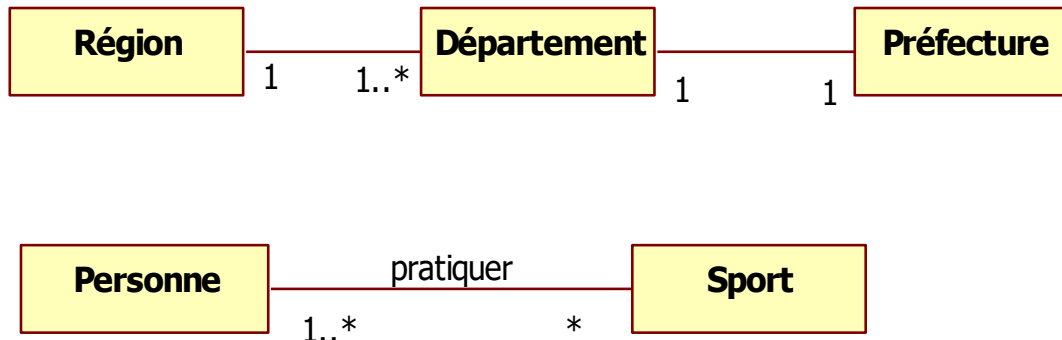
Les **multiplicités** permettent d'indiquer (pour chacune des classes) les nombres minimum et maximum d'instances mises en jeu dans une association.

Interprétation des multiplicités:

Livre	1	Comporte	1..*	Page
-------	---	----------	------	------

*1 livre comporte au moins une page
et
une page de livre se trouve dans un et un seul livre.*

Multiplicités (exemples)

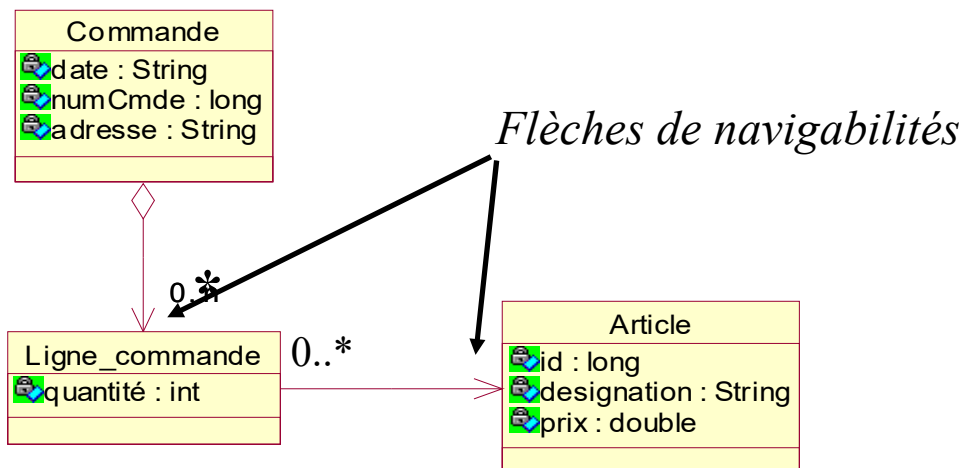


NB:

- Les multiplicités d'UML utilisent des notations inversées vis à des cardinalités de Merise.
- *Les multiplicités dépendent souvent du contexte* (système à modéliser).

Navigabilité

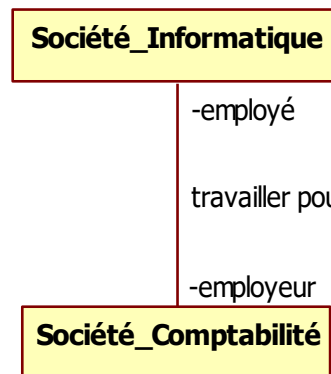
Une **flèche de navigabilité** permet de restreindre un accès par défaut bidirectionnel en un **accès unidirectionnel** *plus simple à mettre en œuvre et engendrant moins de dépendance*.



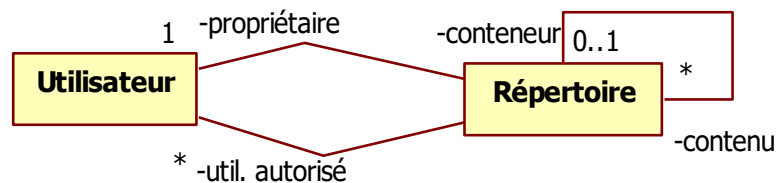
Un article n'a pas directement accès à une ligne de commande

Rôles

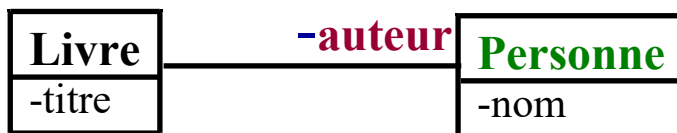
Les rôles (facultatifs) permettent d'indiquer le rôle joué par chaque entité dans le cadre d'une association.



Ils peuvent servir à lever certaines ambiguïtés:



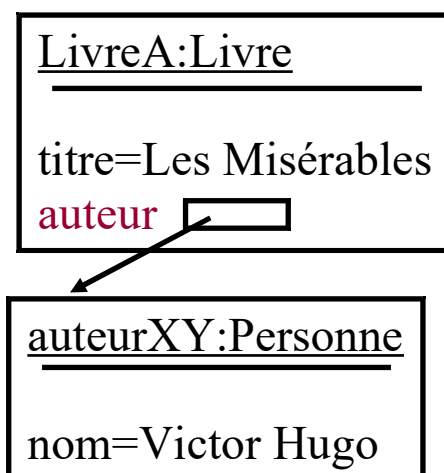
Rôles & implémentation



```

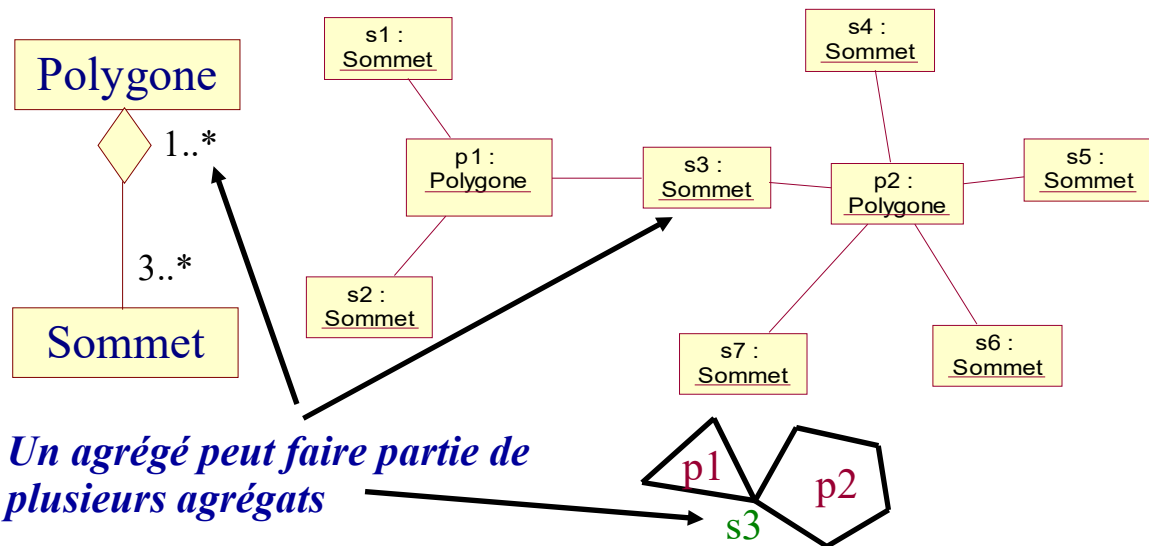
class Livre
{
    private String titre;
    private Personne auteur;
    ...
}
  
```

Les noms des rôles sont souvent utilisés pour nommer les références.



Agrégation

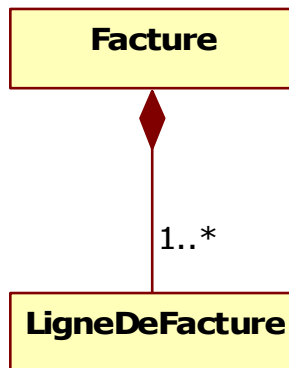
Un **agrégat** est composé de plusieurs sous objets (les agrégés). Cette relation particulière et très classique est symbolisée par un **losange** placé du coté de l'agrégat.



Agrégation (caractéristiques)

- Une **agrégation** est une association de type "**est une partie de**" qui vu dans le sens inverse peut être traduit par "**est composé de**".
- UML considère qu'une **agrégation est une association bidirectionnelle ordinaire** (le losange ne fait qu'ajouter une sémantique secondaire).
- Une agrégation (faible) ordinaire implique bien souvent que les sous objets soient **référéncés** par leur(s) agrégat(s).

Composition (agrégation forte)

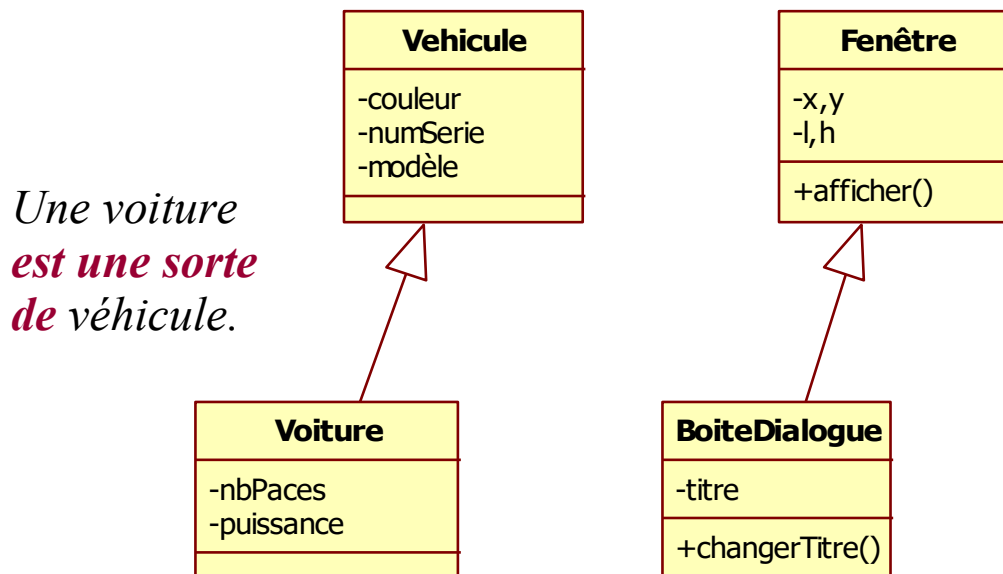


En général, le sous-objet n'existe que si le conteneur (l'agrégat) existe: **lorsque l'agrégat est détruit, les sous objets doivent également disparaître.** (*"cascade-delete" en base de données*).

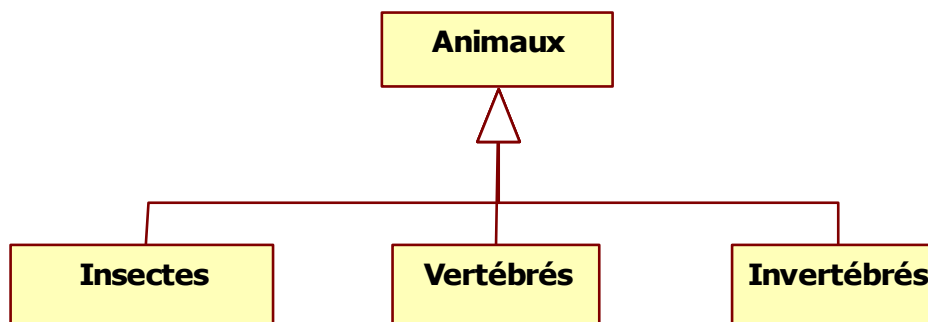
Dans une agrégation forte, un sous objet ne peut appartenir qu'à un seul conteneur.

Remarque: Le losange est quelquefois rempli de noir pour montrer que le sous objet est physiquement compris dans le conteneur (l'agrégat). On parle alors d'**agrégation forte** (véritable **composition**).

Généralisation (héritage)



Classification (généralisation)

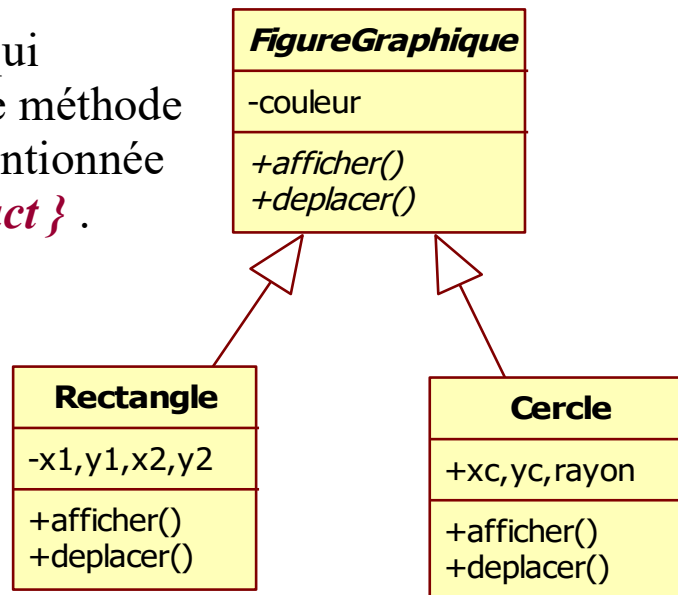


Les animaux peuvent être classées dans divers **groupes** (et sous groupes).

Classification selon caractéristiques discriminantes.

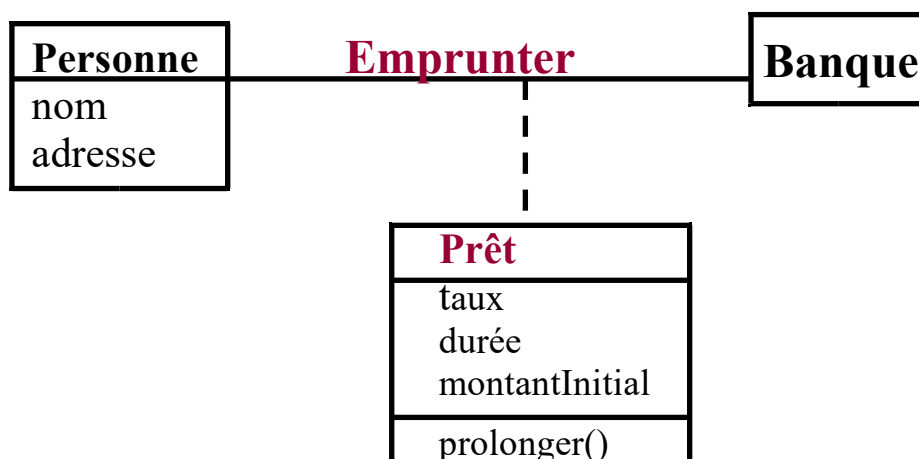
Classes abstraites et concrètes

Une *classe abstraite* (qui comporte au moins une méthode sans code) doit être mentionnée en *italique ou { abstract }*.

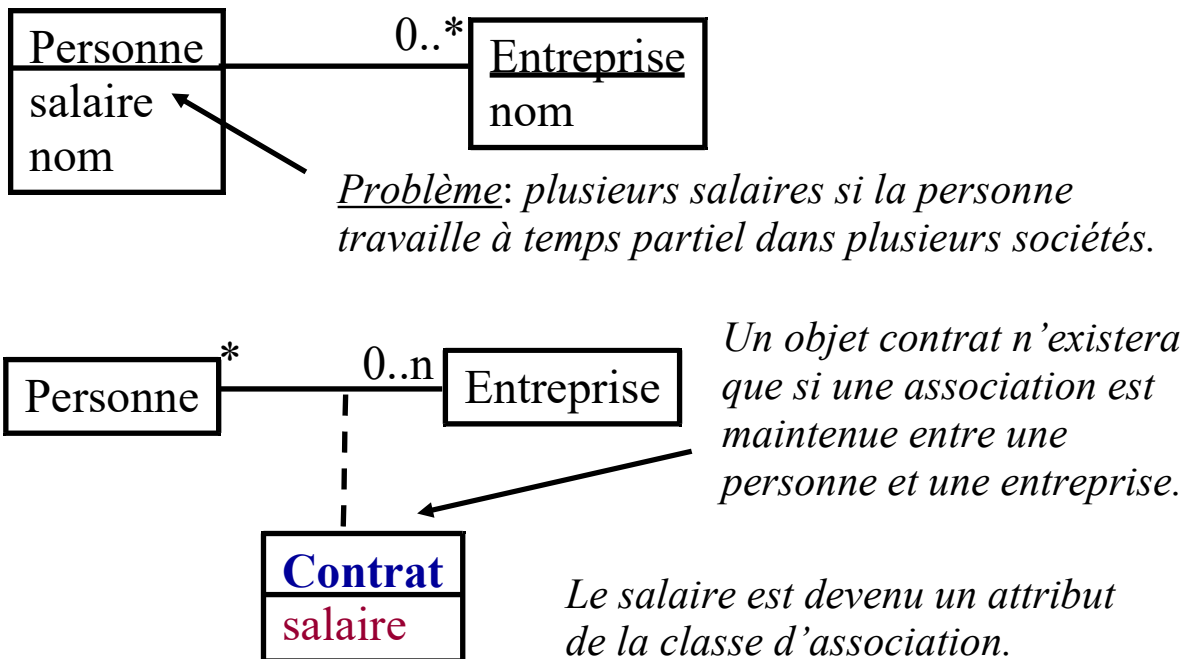


Classes d'association

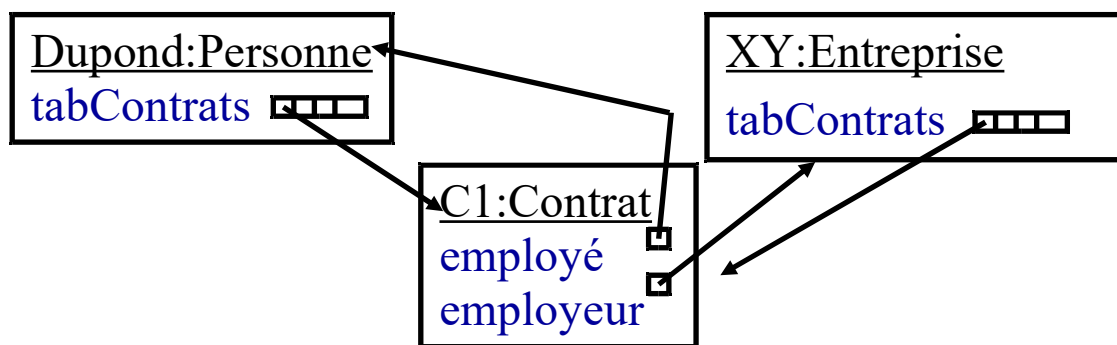
Lorsqu'une association comporte des données ou des opérations qui lui sont propre (non liés à seulement une des entités mises en relation), on a souvent recours à des classes d'association.



Classes d'association (exemples)



Implémentation des classes d'associations



Un objet contrat devient un intermédiaire permettant d'accéder à chacune des entités de l'association:

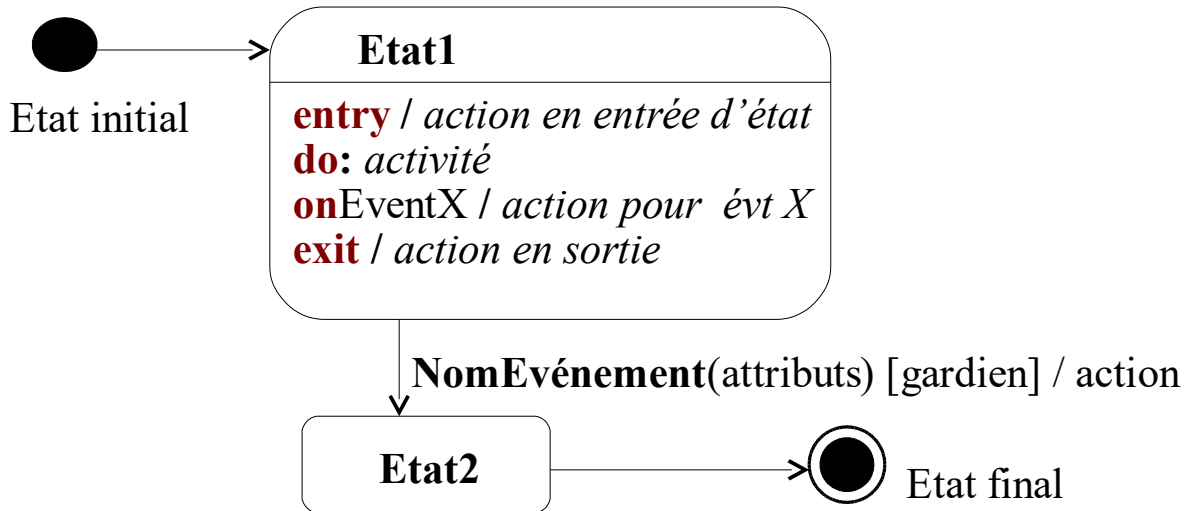
*Contrat1.**getEmploye()**; Contrat1.**getEmployeur()**;*

Package(s) et Namespaces

NB: Si A,B et C sont 3 packages imbriqués alors "A::B::C" est le namespace associé.

7. Diagramme d'états et de transitions (StateChart)

Principales notations (graphe d'états)



NB:

- Chaque **état** est représenté par un **rectangle aux coins arrondis**.
- Un état peut comporter certains détails (activités, actions,...) et peut éventuellement être décomposé en sous états (généralement renseignés dans un autre diagramme).
- Une **activité** dure un certain temps et peut éventuellement être interrompue.
- Une **action** est quant à elle immédiate (opération instantanée, jamais interrompue).

NB: Une transition d'un état vers lui même (self transition) implique une sortie et une nouvelle entrée dans celui-ci .

NB2 : Le diagramme précédent (de l'époque UML1) montre "entry" et "exit" en tant que détails d'un état. Avec UML2, il est préconisé de placer ces détails dans un sous diagramme (avec "entryPoint" et "exitPoint") comme on le verra au sein de quelques pages.

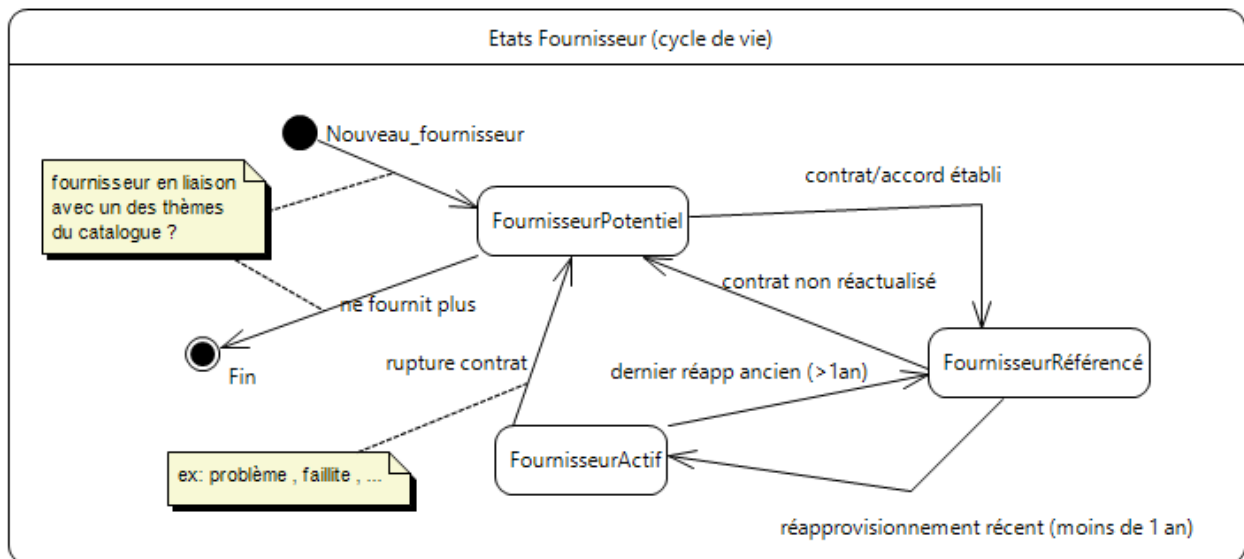
Quelques types (classiques) d'événements :

Types d'événements	exemples	sémantiques
Change Event	when (exp_bouleanne)	L'expression booléenne devient vraie (après changement)
Signal Event	<i>feu vert</i> , ...	Signal reçu (sans réponse à renvoyer)
Call Event	<i>demande_xy_reçue</i> , ...	Appel reçu
Time Event	after (temporisation)	Période de temps écoulée

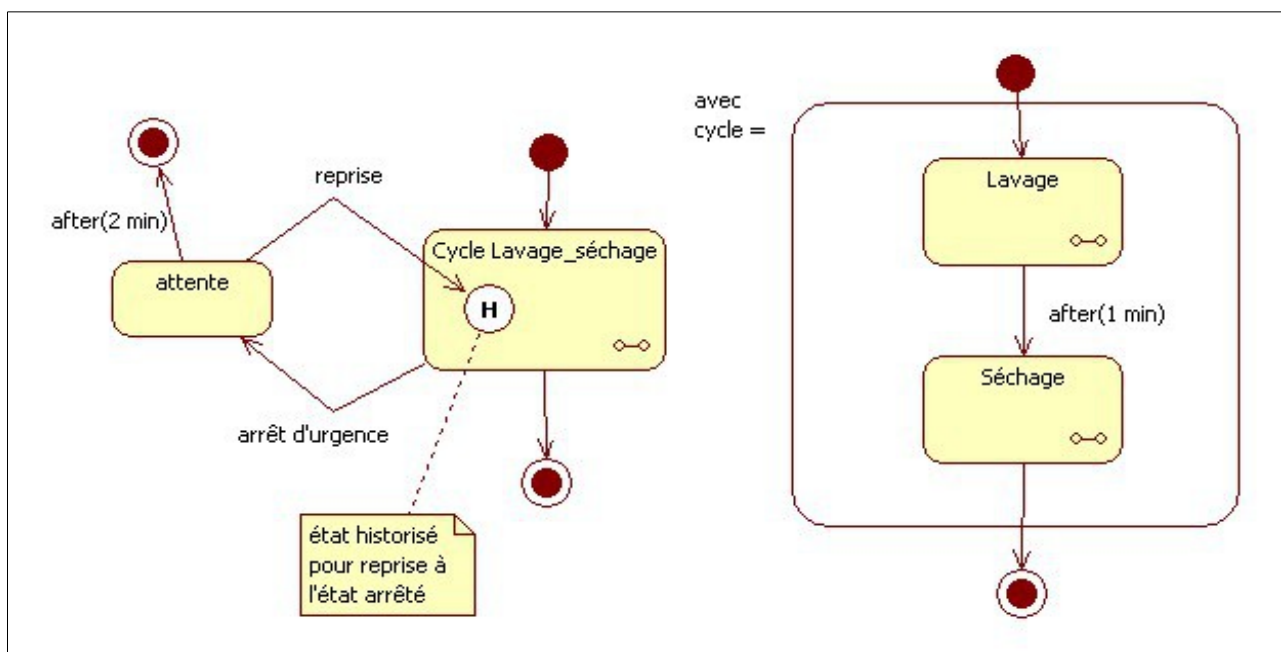
Exemple(s):

Lumière (avec minuterie) : *allumée* ----- **after(30s)**----> *éteinte*

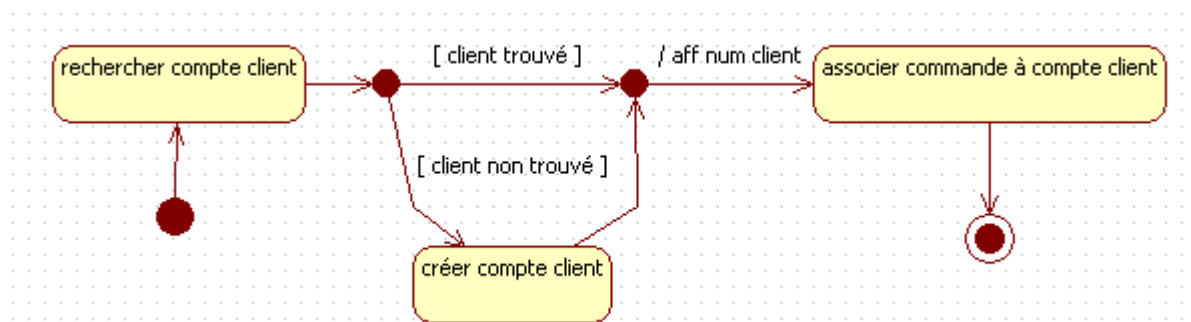
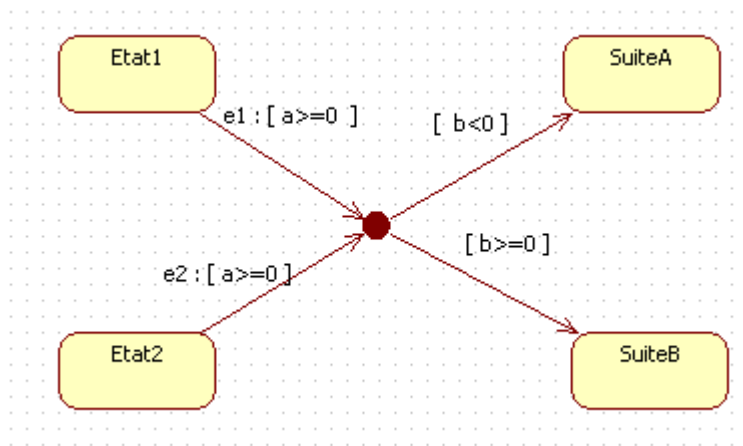
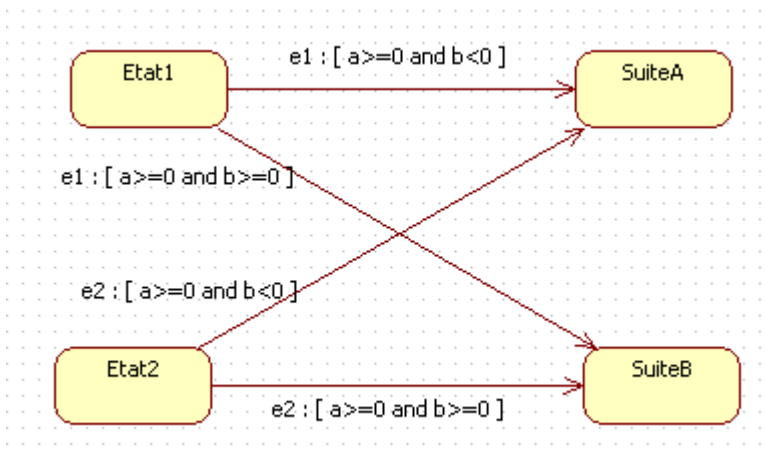
7.1. Exemple simple



7.2. Etats historisés (rares , pour informatique industrielle)



7.3. Point de jonction (depuis UML2)



avec des noms d'états à comprendre ici comme en *"train de"* sachant que les noms des états sont idéalement des *qualificatifs*.

7.4. Etat composite (super état)

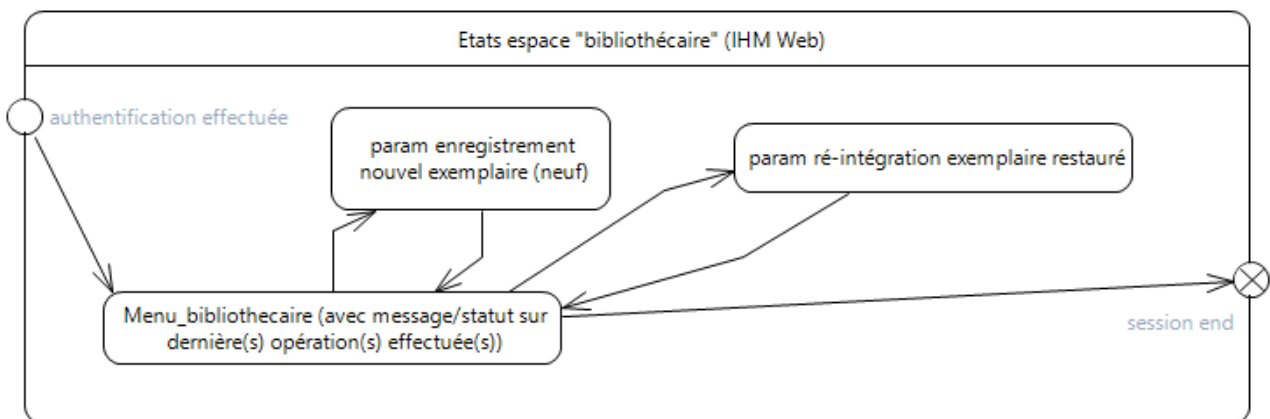
Notation abrégée d'un état composite (à adapter selon outil UML):

Recherche de logement



NB: Quelquefois considéré comme état ordinaire

Point de connexions (en entrée et en sortie) sur la frontière d'un état composite (dans sous diagramme) :



Au sein d'un autre diagramme (parent ou autre) , l'état composite "*Espace xxx*" sera noté comme un état presque ordinaire (simple rectangle) , on l'on pourra tout de même éventuellement placer des "*référence à des connections*" pour affiner les branchements des transitions (changements d'états).

7.5. Liens avec diagramme de classe.

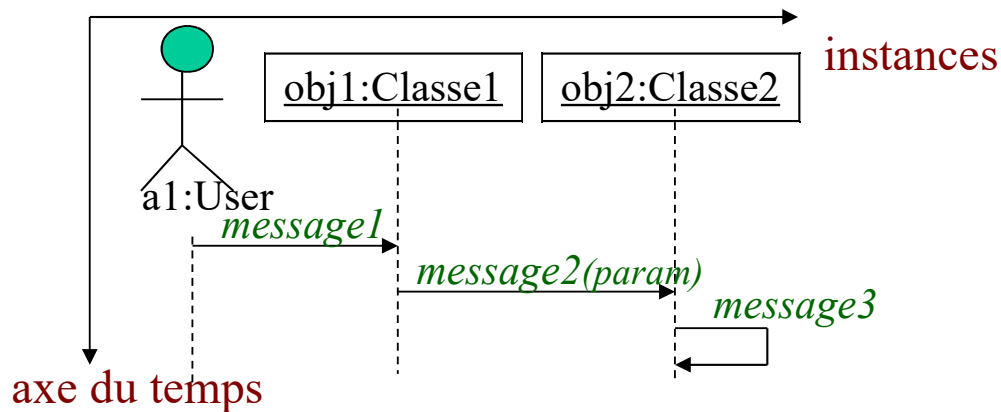
Un diagramme d'état (en tant qu'états de quelque chose) est souvent rattaché à une classe .

Un état peut souvent être codé par un ou plusieurs attributs complémentaires pouvant prendre plein de types/formes différent(e)s :

- "booléen"
- "énumération"
- "lien_vers_xy" existant ou pas ,
- "dateActionXy" nulle ou pas ,
- ...

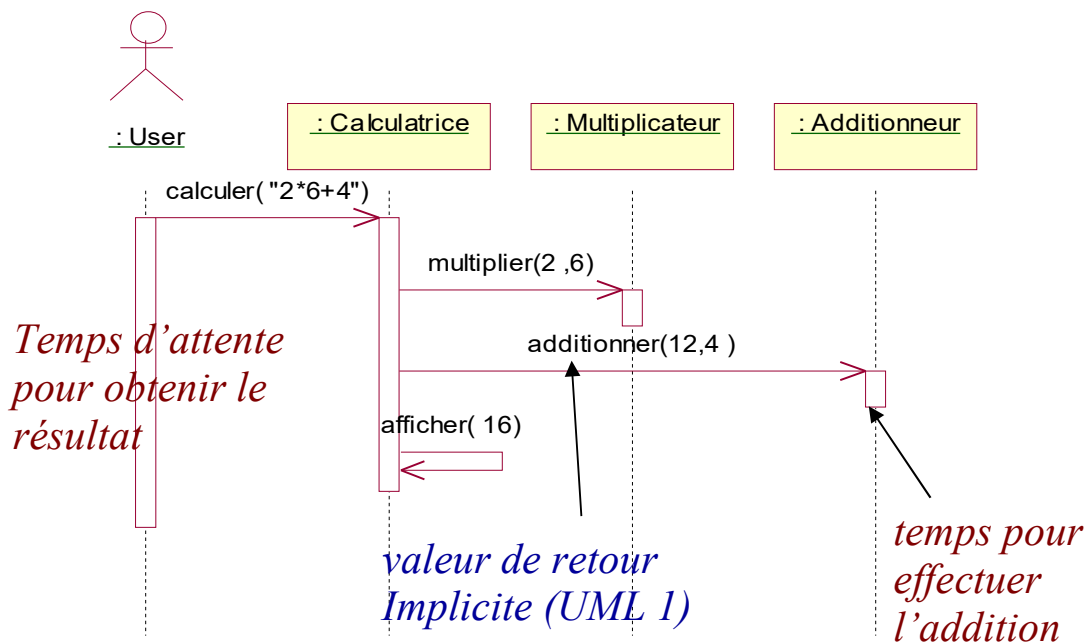
8. Diagramme de séquences (UML)

Diagramme de séquences

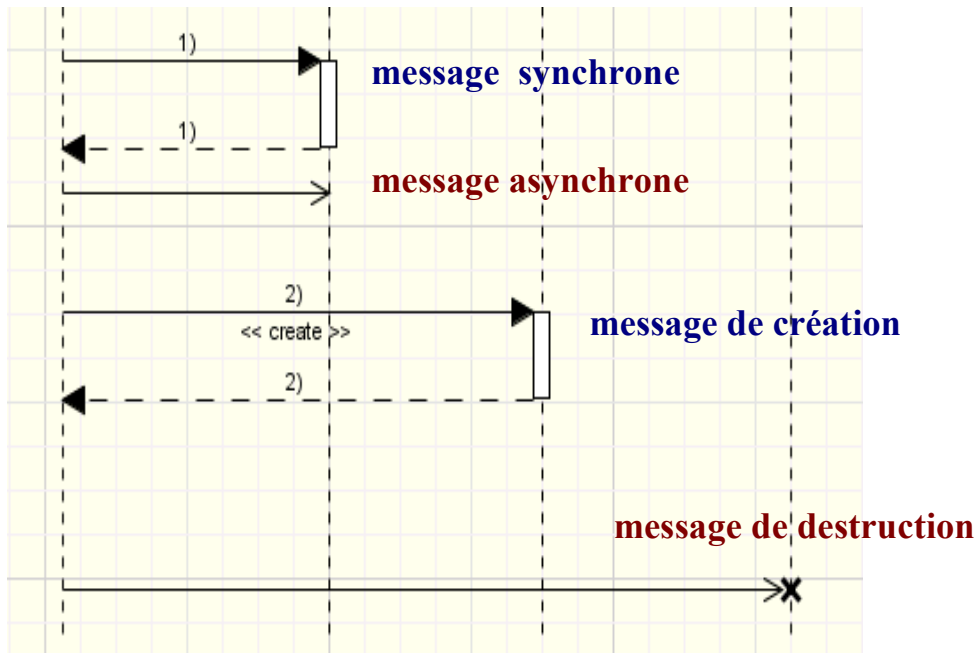


- Un **diagramme de séquence** montre un ensemble de messages échangés entre des objets pour remplir une fonctionnalité donnée ou pour obtenir un résultat.
- **Les interactions sont organisées dans le temps.**

Contrôle du focus (UML 1)

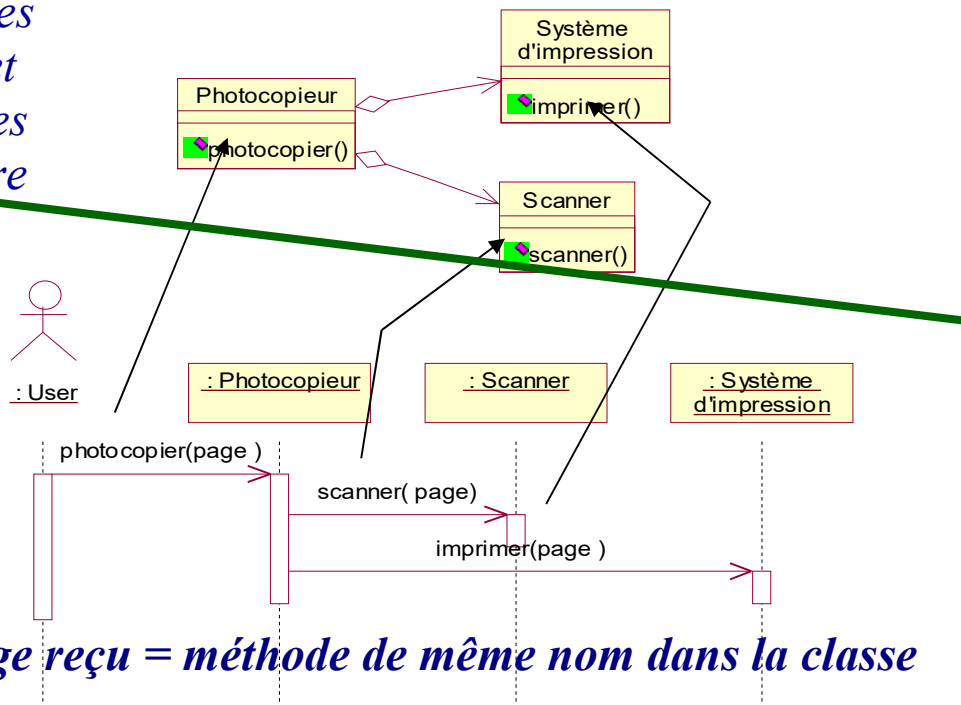


Différents **types** de messages (UML2)



Cohérence entre les digrammes

*Les modèles
statiques et
dynamiques
doivent être
cohérents*



XIV - Annexe – Praxème

1. Méthode Praxème

1.1. Présentation succincte de praxème

La méthodologie Praxème comporte quelques bonnes idées concernant l'approche SOA.

Traits saillants:

- distinguer les aspects logiques, pragmatiques, géographiques, physiques,
- modéliser d'abord les aspects structurels et logiques/fonctionnels des processus et services métiers (ex: diagrammes de classes UML + diagrammes d'états UML)
- modéliser seulement ensuite les activités

Autrement dit :

- ne surtout pas modéliser une série d'activités en s'appuyant sur une organisation pragmatique existante qui n'est pas remise en cause ou que l'on ose pas restructurer .
- Raisonner "fonctionnalités abstraites/logiques devant être apportées" plutôt que "comment relier ça exactement".

Auteurs de Praxème: Philippe DESFRAY , Dominique VAUQUIER .

Figure SLB-02_6. Le Référentiel méthodologique adopte une structure PRO³

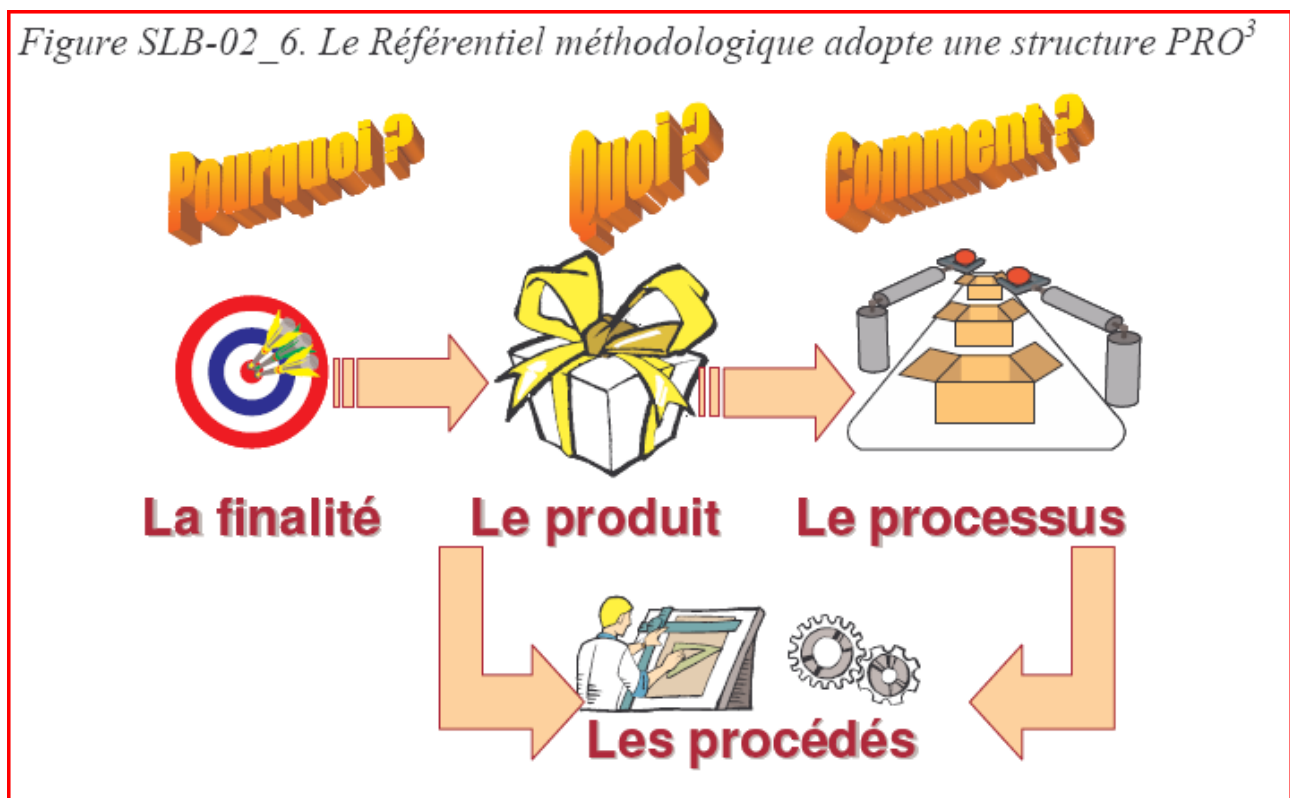


Figure PxM-02_3. Le schéma de la topologie du système

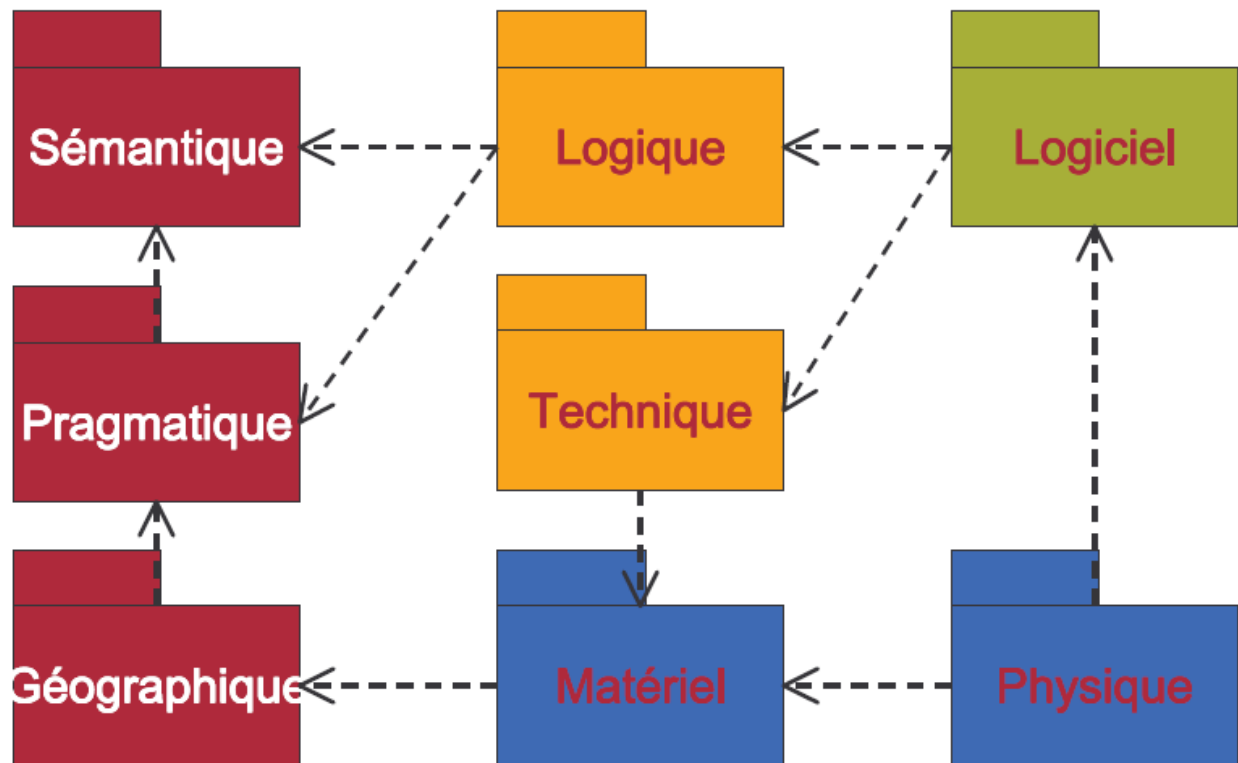
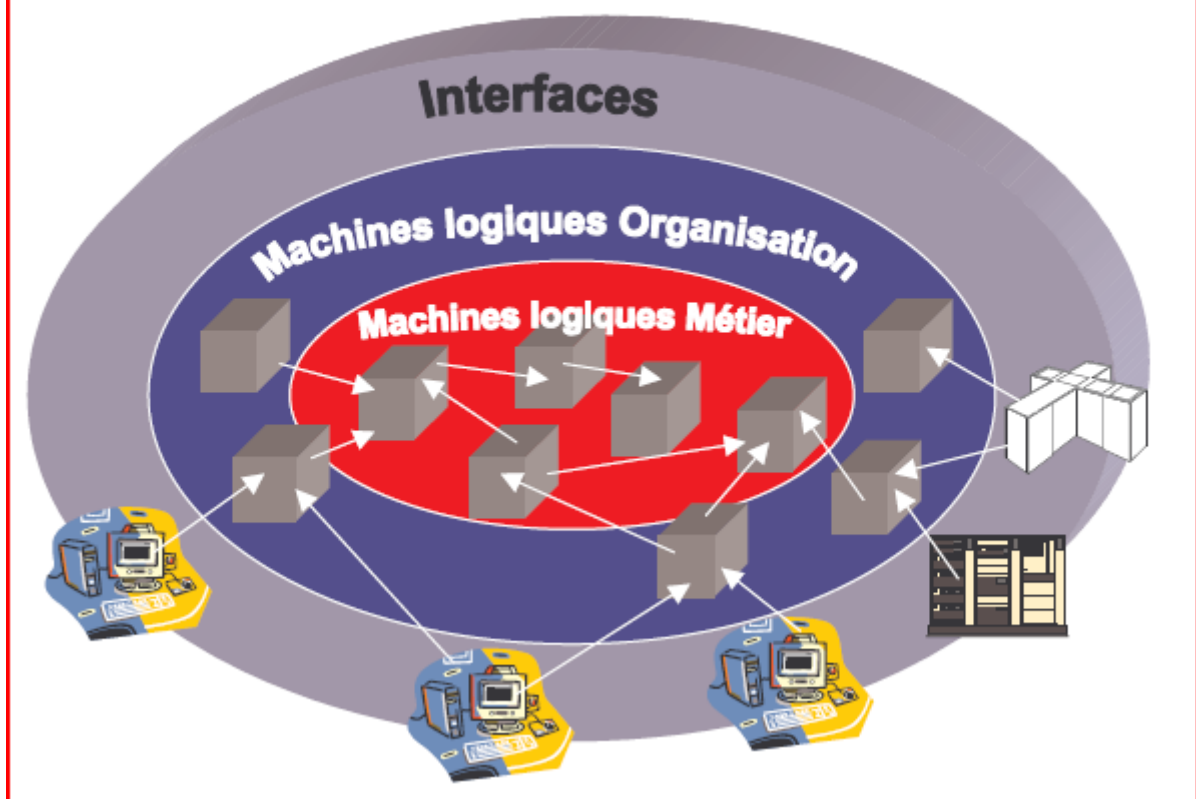


Figure PxM-02_11. La stratification du système d'information, déduite des règles de l'architecture logique



1.2. Terminologie "Praxème" et équivalence.

Nom des modèles de praxème :

<i>Terme "praxème"</i>	<i>Signification approchée</i>	<i>équivalence(s)</i>
M. sémantique (métier)	sémantique / purement métier (avec diag classes & états)	M. purement métier (abstrait, sémantique)
M. pragmatique (organisationnel)	fonctionnel / organisationnel (avec Uses Cases & activités)	M. adapté à l'organisation / entreprise (avec rôles précis)
M. logique	"logique complet" / invocations dépendances (avec services & réal. U.C. / diag. séquence)	M. analyse (proche PIM) [applicatif+métier]
autres modèles (nombreux et peu développés)	autres aspects (logiciel, technique, physique , ...)	M. de conception (proche PSM)

Métaphore urbanisation ---> **zone / quartier / bloc / ilot** (d'une ville)

Métaphore usine productive ---> **fabrique / atelier / machine** (d'une usine)

Noms des domaines (et sous domaines) logiques de praxème :

<i>Terme "praxème"</i>	<i>Signification approchée</i>	<i>équivalence(s)</i>
Fabrique logique	grand domaine	package fonctionnel/métier
Atelier logique	sous domaine (avec façade)	sous package f./m.
Machine logique	composant logique (avec interface) [granularité non élémentaire , cycle fonctionnement]	composant logique

States de "praxème" :

Strates "**métier**" , "**organisationnelle**" , "**ihm**" en tant que *couches logicielles*
ou plutôt "*parties/tiers*"

--> soit du n-tiers adapté "soa" en insistant sur

"service orga " ---invocation(s)---> "service purement métier"

XV - Annexe – Etude de cas , TP

1. Présentation de l'étude de cas

Sujet : Système Informatique (à grande échelle) d'un organisme spécialisé dans la **vente en ligne** (style "CSiscout" / "WebDistrib" / "PixMania" / "Amazon" ou autres).

--> éléments à prendre (un peu) en compte:

- gestion des stocks (produit commandé disponible ?)
- gestion des clients (adresses, ...)
- paiements, facturation
- expédition (packaging des colis , livraison , ...)

(Eventuel graphe sémantique / ébauche)

Elément central (tout tourne autour : **commande** de produits à livrer)

--> au moins un diagramme d'états et quelques "processus métiers / diag. D'activités".

Eléments externes:

- > Fournisseurs de produits (pour ré-approvisionner les stocks)
- Prestataire de transport pour les colis (ex: "La-Poste")

2. TP (modélisation SOA)

Modéliser partiellement un "*système de ventes en ligne*" basé sur une **architecture SOA**.

Cette modélisation pourra être considérée comme une pré-étude (débouchant habituellement sur un prototype).

- 1) Modélisation (partielle) d'un service élémentaire "GestionProduits" (avec données "Produit" dont le champ "quantitéEnStock" est à null (non déterminé , pas géré) ,...)
- 2) Transposition "UML → java et/ou WSDL"
- 3) Modélisation (rapide) d'un service élémentaire "GestionStock" (sur produits opaques)
- 4) Modélisation d'un service fonctionnel "ProduitsEnStock" s'appuyant sur les 2 services élémentaires précédents.
- 5) Modélisation métier (business Use-cases , processus "bpmn" avec chorégraphie)
- 6) mini étude d'urbanisation à compléter (remplir les noms manquants)
- 7) modéliser quelques données de "référence" (modéliser un cycle de vie avec un diag d'états)
- 8) étudier quelques structures d'échange (données communes , données pivots,)
- 9) étudier éventuellement quelques invocations de services (d'un package à un autre)
- 10) modéliser une orchestration de services avec BPMN (expédition/livraison de colis avec attente de l'accusé de réception)

recupérer sur clef USB une mini-solution (modélisation partielle)développée par le formateur