
conception avec UML

Table des matières

| | |
|---|-----------|
| I - Présentation du cours (objectifs)..... | 4 |
| 1. Conception avec UML..... | 4 |
| 2. <i>Rappels fondamentaux</i> :..... | 5 |
| II - Conception générique / frameworks..... | 8 |
| 1. Rôles de la conception..... | 8 |
| 2. Activités de la conception..... | 9 |
| 3. Interfaces (diag. de classes)..... | 11 |
| 4. Conception générique..... | 13 |
| 5. Présentation du concept de Design Pattern..... | 14 |
| 6. Importance du contexte et anti-patterns..... | 16 |
| 7. Principaux "design patterns"..... | 17 |
| 8. Différences entre "Design Pattern" et "Framework"..... | 19 |
| 9. Éléments de conception générique..... | 20 |
| III - Projection (avec ou sans M.D.A.)..... | 25 |
| 1. Projection du fonctionnel dans technologies..... | 25 |
| 2. Objectifs de la conception préliminaire..... | 26 |

| | |
|--|-----------|
| 3. Utilisation de MDA en conception..... | 27 |
| IV - Conception modulaire (composants)..... | 29 |
| 2. Description des modules (interfaces,façades)..... | 29 |
| 3. Modélisation des dépendances inter-modules (diagramme de composants UML)..... | 32 |
| V - Eléments de conception détaillée..... | 34 |
| 1. les conceptions détaillées..... | 34 |
| 2. Conception détaillée des services métiers et de la persistance..... | 34 |
| 3. Conception détaillée de l'IHM (couche présentation)..... | 36 |
| 4. Quelques notations détaillées et/ou pointues..... | 39 |
| 5. Templates/Generics (notations avancées, ...)..... | 40 |
| 6. Diagramme de structure composite..... | 41 |
| 7. Aperçu sur diagramme de timing UML2..... | 42 |
| VI - Implémentation , tests , itérations..... | 43 |
| VII - Annexe – UML et mapping objet-relationnel..... | 46 |
| 1. Cas de figures (down-top , top-down , ...)..... | 46 |
| 2. Correspondances essentielles "objet-relationnel" | 46 |
| 3. Correspondances "objet-relationnel" avancées..... | 49 |
| 4. Stéréotypes UML pour O.R.M..... | 50 |
| VIII - Annexes – Design Principles..... | 52 |
| 1. Présentation des " <i>design principles</i> "..... | 52 |
| 2. Gestion des évolutions et dépendances..... | 53 |
| 3. Organisation d'une application en modules..... | 57 |
| 4. Gestion de la stabilité de l'application..... | 58 |
| IX - Annexe - Patterns "GOF" et "JEE"..... | 61 |
| 1. Liste des principaux "design patterns" du GOF..... | 61 |
| 2. Design Patterns fondamentaux du GOF..... | 63 |
| 3. Injection de dépendances..... | 75 |
| X - Annexe - Infrastructure Eclipse EMF / UML..... | 79 |
| 1. Infrastructure pour UML, MDA, | 79 |
| 2. XML..... | 79 |

| | |
|---|-----------|
| XI - Annexe - M.D.A. (principes)..... | 82 |
| 1. MDA..... | 82 |
| XII - Annexe – Outils / éditeurs UML..... | 85 |
| 1. Quelques outils UML (Editeurs , AGL)..... | 85 |
| XIII - Annexe - Mise en oeuvre MDA via Accéléo..... | 87 |
| Présentation d'accéléo3_M2T (plugin eclipse)..... | 87 |
| 1. Point d'entrée de la génération de code MDA : un modèle logique UML stéréotypé..... | 87 |
| 2. Versions d'accéléo..... | 88 |
| 3. Installation du plugin accéléo3 (M2T) sous eclipse..... | 88 |
| 4. Création d'un projet "accéléo" sous eclipse..... | 88 |
| 5. Exemple de "template" (modèle de code à générer)..... | 89 |
| 6. lancement unitaire d'une génération de code..... | 90 |
| 7. lancement d'une génération de code via script ant..... | 90 |
| 8. Module , sous modules et importation..... | 91 |
| 9. Templates , sous templates et applications..... | 91 |
| 10. Quelques éléments de syntaxe..... | 93 |
| 11. Générateurs sophistiqués..... | 94 |
| XIV - Essentiel outil "Papyrus UML" | 99 |
| 1. Utilisation de Papyrus_UML (éditeur UML2 eclipse)..... | 99 |
| 2. Génération de documentation (gendoc2)..... | 111 |

I - Présentation du cours (objectifs)

1. Conception avec UML

Le cours "**Conception avec UML**" se focalise essentiellement sur
*ce qu'il est nécessaire de bien modéliser en vue d'implémenter/générer
une application informatique concrète.*

La connaissance (supposée déjà acquise) des concepts objets est indispensable pour une bonne compréhension des modèles de conception.

Une expérience de la programmation Java/JEE ou C++ ou C# ou typescript ou "php orienté objet" ou "python" (ou équivalent) est conseillée (bien que non indispensable)

Les phases amonts de la modélisation UML (Expression des besoins avec les cas d'utilisations , analyse fonctionnelle) sont supposées déjà connues/abordées et ne seront que brièvement rappelées en début de formation.

Quelques éléments de conception seront développés dans les détails:

- **Design Patterns** utiles pour bien structurer une application n-tiers (**Façade** , Fabrique/**Injection de dépendances** , ...)
- **Conception générique** (framework , interfaces/contrats,)
- **MDA** (Model Driven Architecture) & génération de code paramétrable.
- **Projection (fonctionnel dans technologie)**
et éléments essentiels du Processus 2TUP (en Y) .
- Bonne utilisation des **stéréotypes** .
-

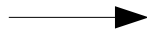
Formation préalable conseillée :

Expression des besoins et analyse avec UML

2. Rappels fondamentaux :

Pourquoi ?

(Quels objectifs ?
Quelles utilités ?)



Modélisation métier

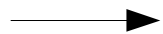
(business modeling)

+ expression des besoins

(C.I.M. : Computation Independant Model)

Quoi ?

(Quelles entités ?
Quelles structures ?
Quels services ?)

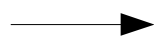


Analyse

(P.I.M. : Platform Independant Model)

Comment ?

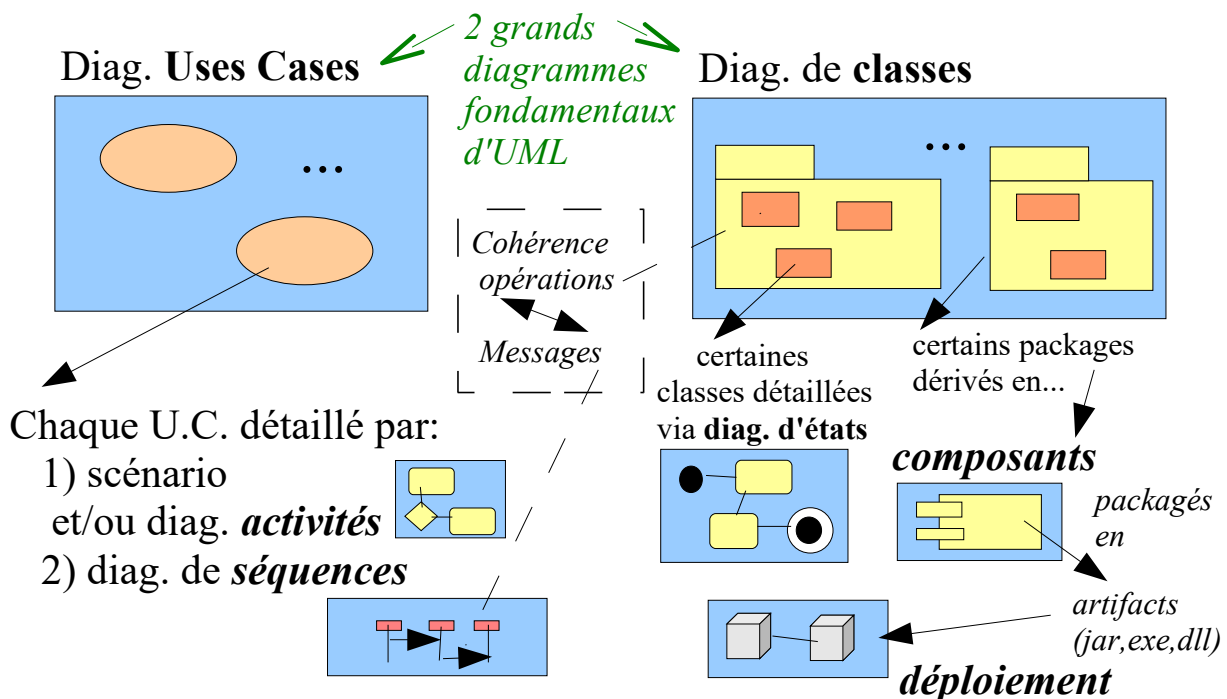
(avec quelles
Technologies ?)

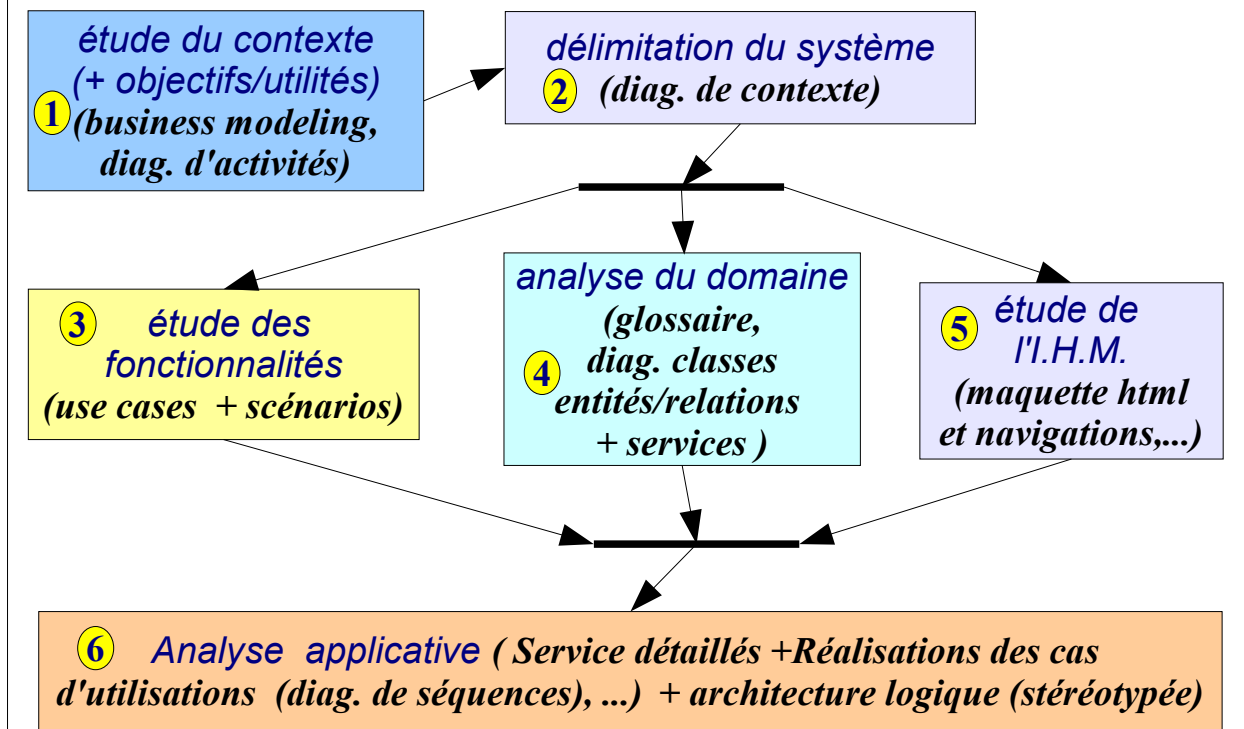
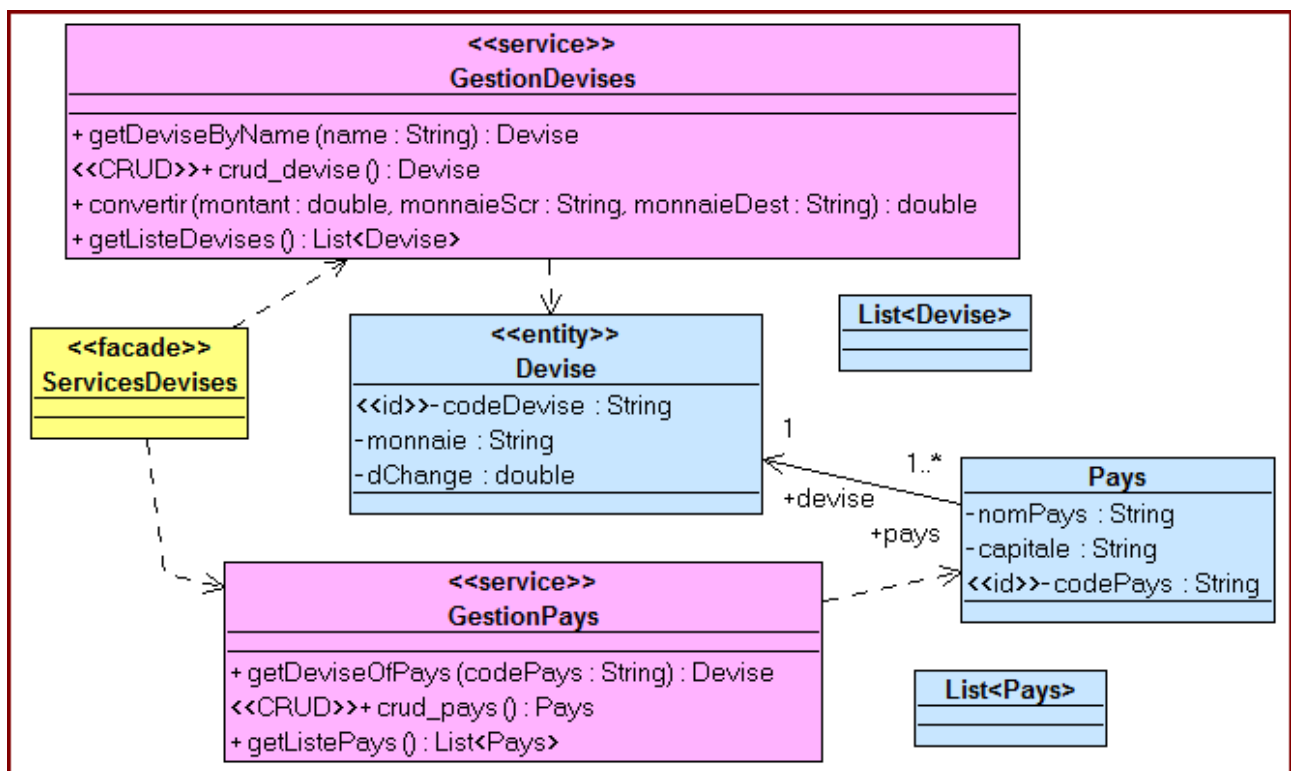


Conception

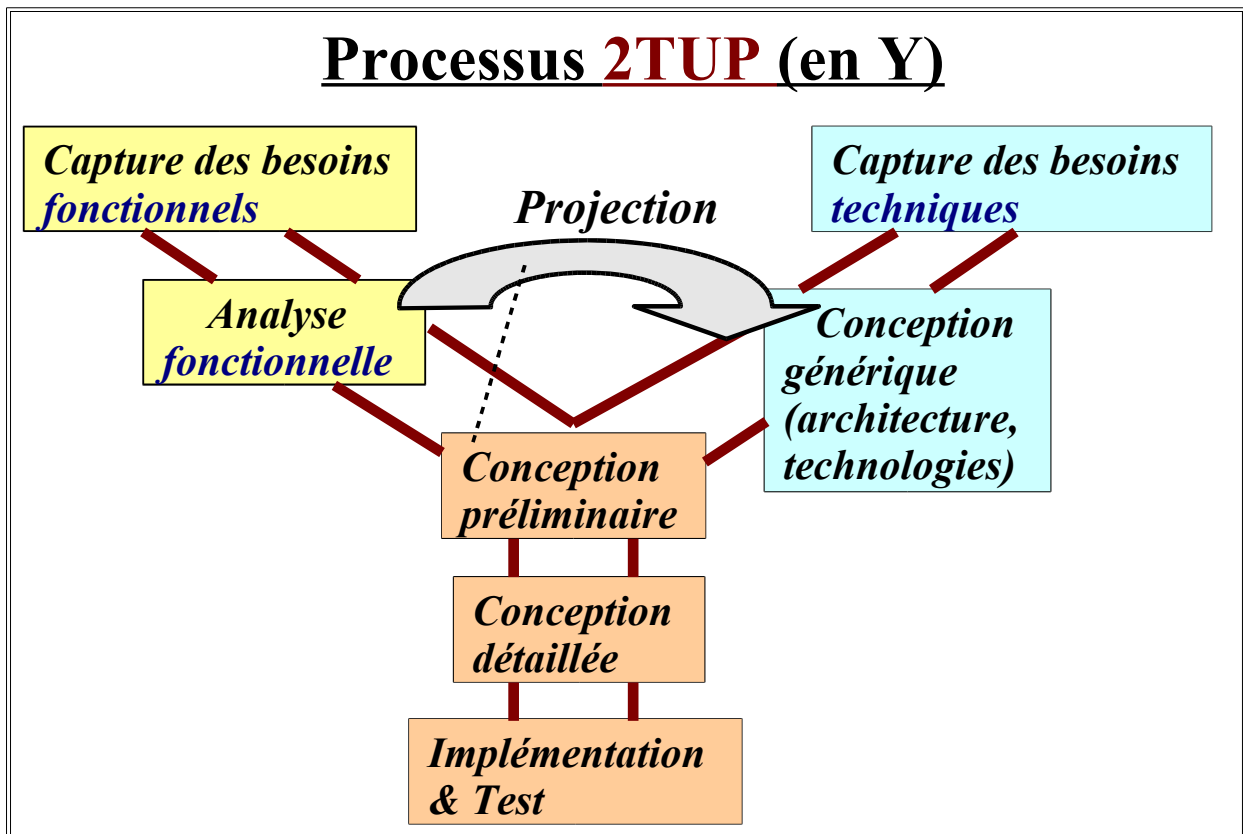
(P.S.M. : Platform Specific Model)

Principaux liens entre les diagrammes UML



Phases amonts (avant la conception) :**Enchaînement classique d'activités sur la partie fonctionnelle**Diagramme de classes (en fin d'analyse) :

Repère méthodologique souvent pertinent:



II - Conception générique / frameworks

1. Rôles de la conception

La **conception** a pour rôle de définir "**comment**" les choses doivent être **mise en oeuvre**:

- quelle **architecture** (client-serveur , n-tiers, SOA ,)
- quelles **technologies** (langages , frameworks ,)
- quelle **infrastructure** (serveurs à mettre en place ,)

avec tous les détails nécessaires .

Étymologiquement:

conception ==> **inventer** (concevoir) une solution

pragmatiquement:

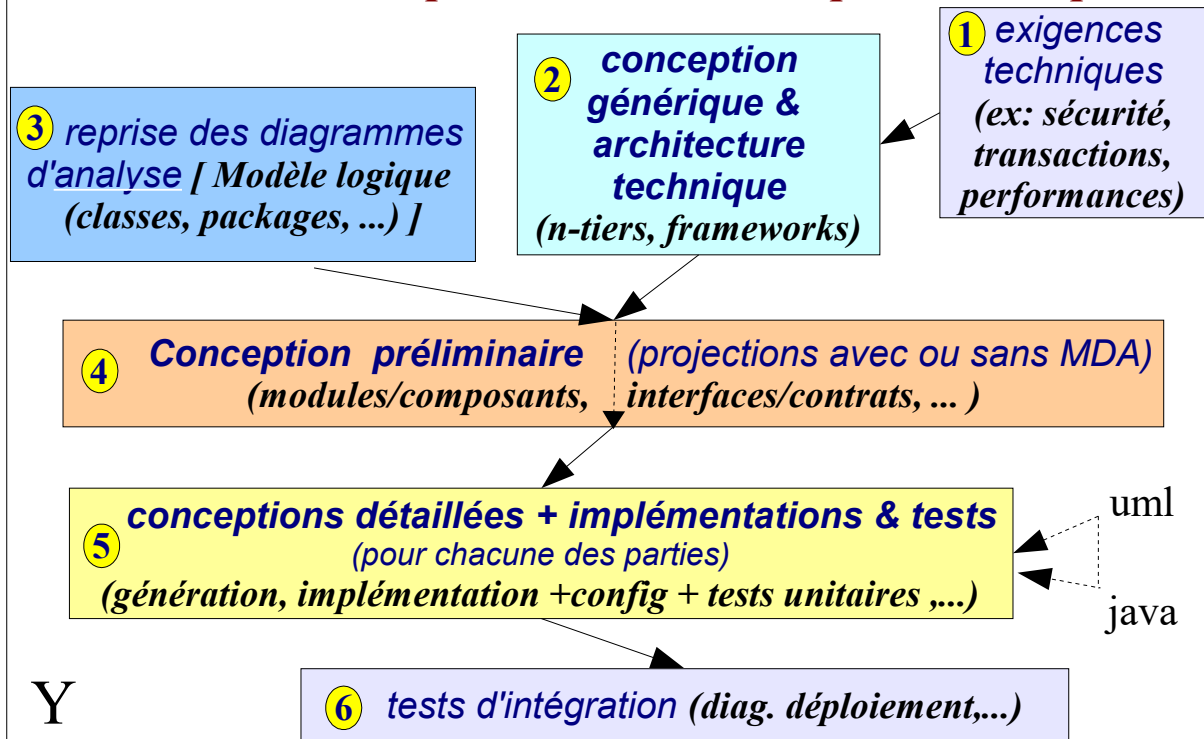
conception ==> très souvent **réutiliser/choisir une solution/technologie (framework)**
[*ne pas ré-inventer*]

NB:

- Au début des années 1990, il n'y avait pas encore beaucoup de framework parfaitement au point et il fallait à l'époque beaucoup inventer .
- Au début des années 2000, quelques api/frameworks bien structurés (ex : .net/C# , Java/JEE) dominaient le marché et il fallait alors comprendre et intégrer (ne pas ré-inventer)
- A l'aube des années 2020, l'architecture micro-services offre plein de variantes dans les possibilités de mise en œuvre et il faut donc bien argumenter/spécifier les choix d'architecture et avoir quelquefois recours à une double modélisation :
 - cohérence à grande échelle (urbanisation, communications entre applications, ...)
 - modèle orienté objet détaillé à petite échelle (ex : api-rest) .

2. Activités de la conception

Enchaînement classique d'activités sur la partie conception



2.1. conception générique et architecture technique

indispensable et utile !!! (travail d'un "architecte technique")

- Modéliser une bonne fois pour toutes les éléments récurrents.
- Structurer les grandes lignes de l'architecture technique en s'appuyant sur des **frameworks** (prédéfinis ou "maisons") et en tenant compte de l'**état de l'art** : technologies et infrastructures du moment (exemple : api-rest et conteneurs "docker").

2.2. Conception préliminaire

souvent utile !!! (au moins à "dégrossir" par un concepteur expérimenté)

- **Projeter** le résultat de l'analyse (fonctionnel / métier) dans l'architecture technique (logique ou physique) choisie.
- **Identifier les différents modules** qui seront ultérieurement modélisés et développés séparément .
- **Bien spécifier les interfaces (contrats) entre les différents modules**

2.3. éventuelles conceptions détaillées

*à faire que si certains détails techniques doivent être formalisés (ex : sur gros projet)
---> attention : très gros travail (chronophage , pas toujours rentable, dépendances vis à vis d'outils ou de technologies manquant de pérennité --> perte de temps potentielle)*

- conceptions détaillées (séparées , en //) de chacun des modules identifiés par la conception préliminaire.
- **Modélisation UML ==> codage/implémentation**
ou bien
codage direct (avec modèle générique dans la tête) ==> reverse engineering (doc UML)

2.4. implémentation & tests

indispensable et utile !!! (travail du "développeur")

- Tests unitaires via JUnit ou ... (validation de chaque module)

2.5. tests d'intégration

- Test d'intégration global (collaboration efficace entre les différents modules ?)

Important:

Entre petit et gros (projet/équipe) , la principale différence tient en un besoin plus ou moins important d'homogénéité:

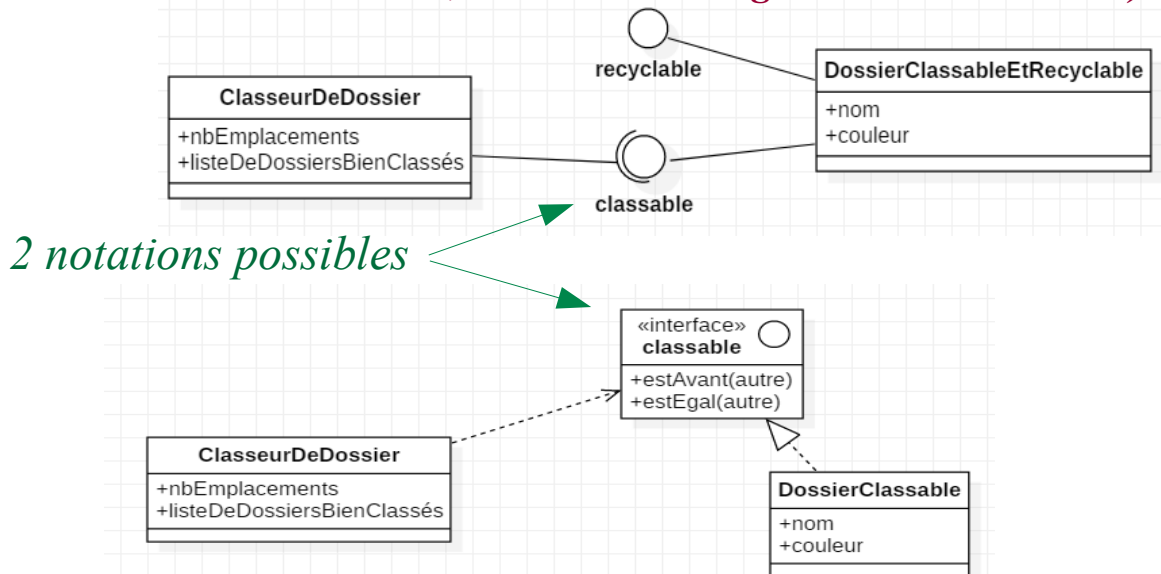
Style libre pas gênant si développement tout seul et s'il n'y pas trop de répétition.

Style libre (exotique) très gênant si développement en équipe ou si diversification de style dans les différentes parties devant être ultérieurement assemblées.

3. Interfaces (diag. de classes)

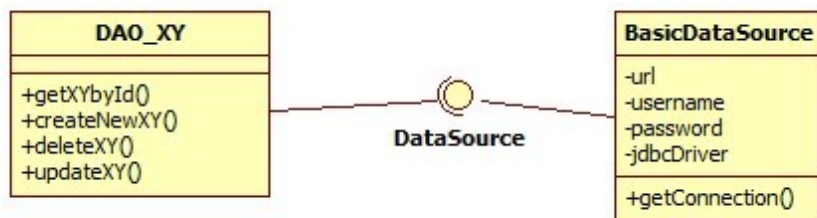
Interface

Une interface correspond à une collection d'opérations sans code (classe spéciale sans aucun attribut, seulement des signatures de méthodes).

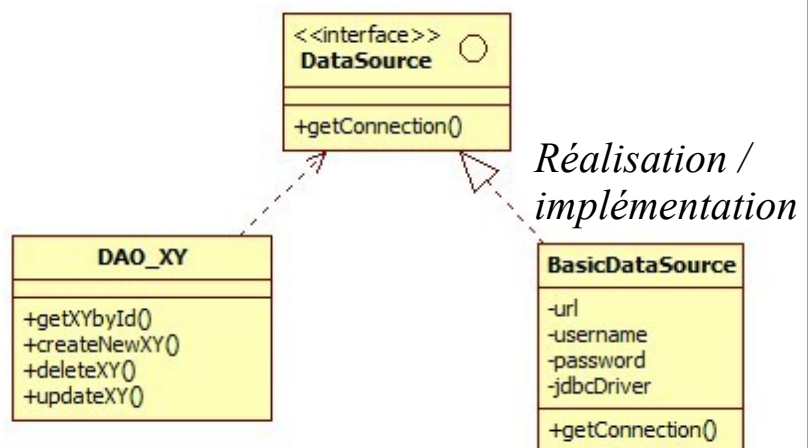


Tout comme une classe abstraite, une **interface** correspond à un **type de données** (permettant de déclarer des références).

*Notation
Compacte
(dépendance)*

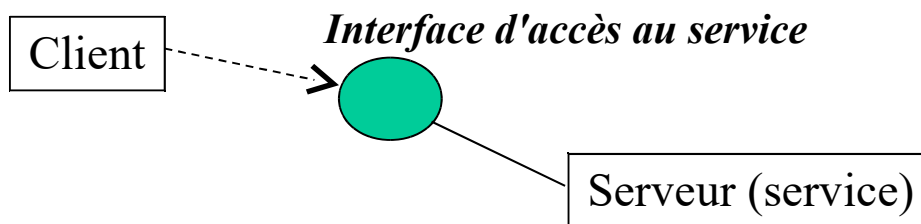


*Notation
développée
(avec détails)*



Interface (contrat)

- Une **interface** peut être considérée comme un **contrat** car **chaque** classe qui choisira d'implémenter (réaliser) l'interface sera obligée de programmer à son niveau toutes les opérations décrites dans l'interface.
- Chacune de ces opérations devra être convenablement codée de façon à rendre le **service effectif** qu'un client est en droit d'attendre lorsqu'il appelle une des méthodes de l'interface.



NB : Désignant un paquet de fonctionnalités invocables, certaines interfaces ont des noms finissant par "able" :

- Imprimable, Printable
- Serializable
- Affichable
- Configurable
- ...

4. Conception générique

4.1. Infrastructure technique générique (framework , ...)

De façon à :

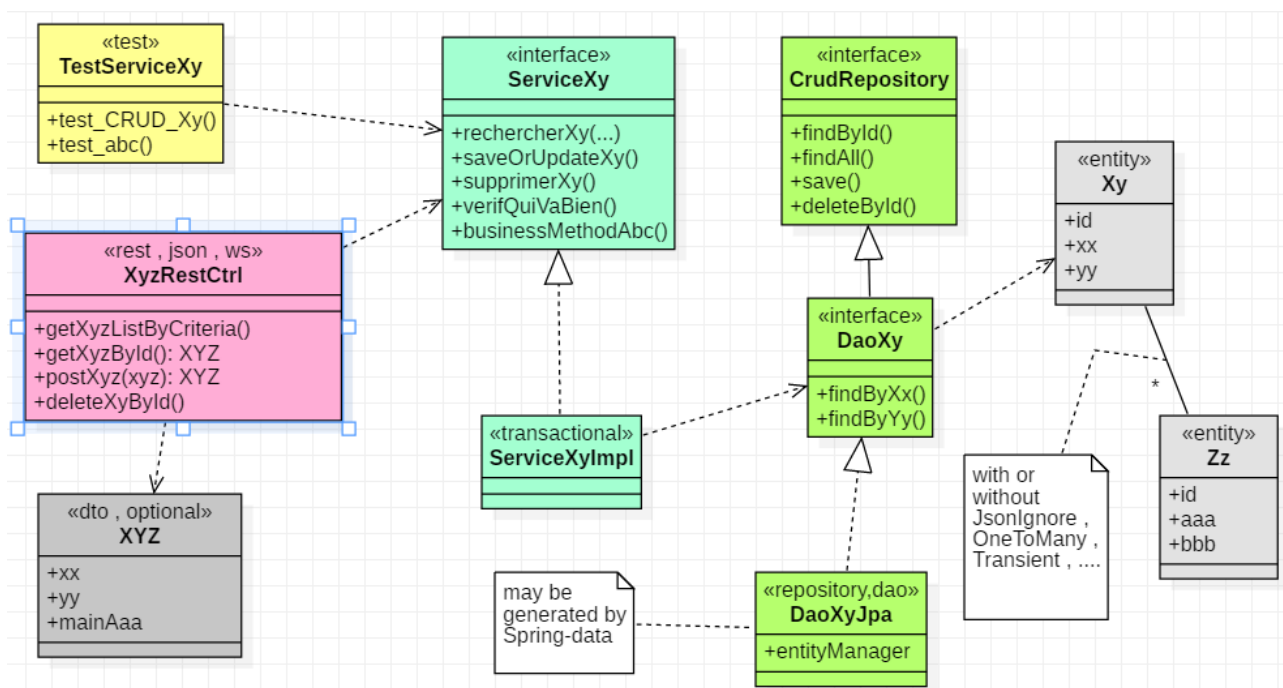
- répondre aux principaux besoins techniques préalablement identifiés .
- bien dissocier l'interface d'une couche de son implémentation ,

on a généralement besoin de s'appuyer sur différents frameworks:

- frameworks techniques prédéfinis (JEE, MVC2/Struts/JSF2 , AngularJs, ...)
- frameworks spécifiques (à bâtir de toutes pièces en s'inspirant de divers "Design Pattern")

Des diagrammes UML de classes ou de collaboration peuvent être à ce niveau utiles pour décrire la structure et les principes de fonctionnement d'un framework (que ce dernier soit prédéfini ou pas).

Exemple: architecture technique (d'une api REST) basée sur une des technologies des années 2018_2019 (Spring-boot / Spring-mvc / Spring-Data) et devant prendre en charge les exigences techniques d'une application de gestion (transactions, ...).



4.2. Intérêts de la conception générique

Les principaux intérêts de la conception générique sont les suivants:

- ne modéliser qu'une seule fois les éléments récurrents
- dégager les invariants et bâtir ou réutiliser des solutions génériques permettant de gagner beaucoup de temps sur le développement.
- modéliser un template pour une éventuelle génération de code automatique (ex: accéléo / MDA).

5. Présentation du concept de Design Pattern

Notion de "Design Pattern"

Un **Design Pattern** [DP] est un **modèle de conception réutilisable** (dont on peut s'inspirer de multiples fois sans pour autant toujours coder les choses de la même façon).

C'est concrètement un document d'une dizaine de pages comprenant:

Description d'un problème récurrent (contexte)

Modélisation (ex: *UML*) d'une solution de conception générique

Description des avantages/inconvénients

Exemples de code , Variantes

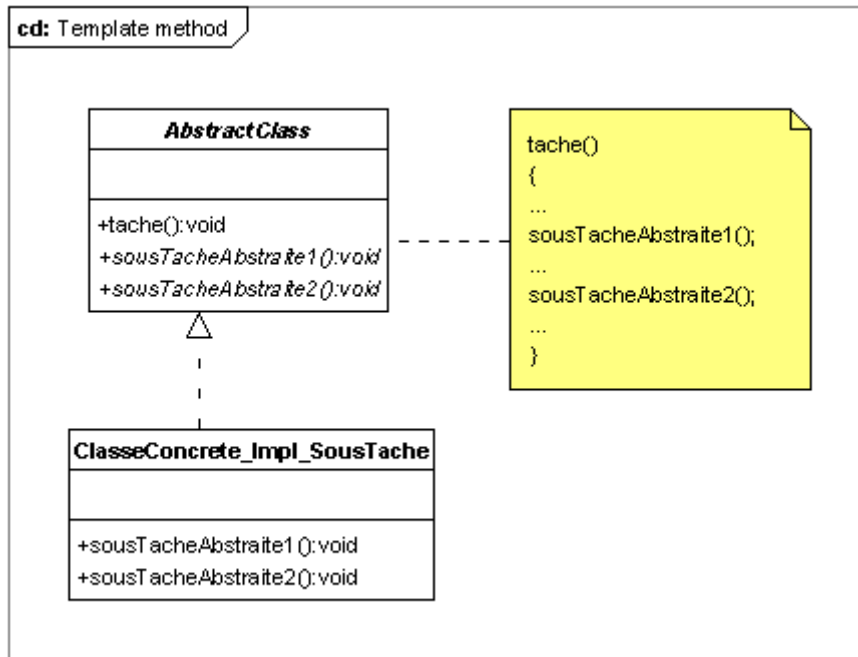
NB:

L'inventeur du concept de "Design Pattern" (**Christopher Alexander**) ne travaillait pas dans le domaine de l'informatique.
Selon lui :

<< Chaque modèle décrit un problème qui se manifeste constamment dans notre environnement, et donc décrit le coeur de la solution de ce problème, d'une façon telle que l'on peut réutiliser cette solution des millions de fois, sans jamais le faire deux fois de la même manière >> .

5.1. Exemple: Patron/modèle de méthode (Template method)

Factoriser ce qui est commun , différencier le spécifique .



Exemple de code:

// code commun de la tache au niveau de la classe abstraite:

```

void dessinerAvecCouleur(String couleur) {
    choisirCouleur(couleur); // code commun (factorisé au niveau de la classe abstraite)
    dessiner(); // sous tache abstraite avec code différent pour ligne , rectangle , ...
}

```

6. Importance du contexte et anti-patterns

6.1. contexte d'un design pattern et objectif visé

Attention: un Design Pattern n'est généralement intéressant qu'au sein d'un contexte particulier:

- Certaines conditions doivent généralement être vérifiées (volume ,) .
- But à atteindre ?
-

L'utilisation inadéquate (non réfléchie ou systématique) d'un design pattern est quelquefois contre-productive ==> ceci constitue un des "anti-pattern" (choses à ne pas faire , modélisation d'un dysfonctionnement ou problème potentiel, ...).

6.2. patterns et anti-patterns

| | Design Pattern | Anti Pattern |
|-------------------------------|--|--|
| Pb Récurrent / Exemple | Objectif visé revenant souvent | Problème (défaut) apparaissant souvent |
| Modélisation / Généralisation | Modèle d'une chose à reproduire | Modèle d'une chose à éviter |
| ex de code / variantes | ex de code dont on peut s'inspirer + variantes | ex de mauvais code (ou mauvaise technique) à prohiber + alternatives conseillées |

Quelques exemples d'anti-patterns:

- **trop de "copier/coller" dans le code**
 ==> reproduction/prolifération de choses à faire évoluer en parallèle ou d'éventuels défauts
 ==> maintenance difficile
 ==> une factorisation est souvent possible (via une petite restructuration du code)
- **trop de "if" ou "switch/case" sans polymorphisme**
- **code "mort" (jamais utilisé)**
- ...

7. Principaux "design patterns"

7.1. Les grandes séries de "Design Patterns" orientés "objet"

Les grandes séries de "design patterns"

- **GRASP** [*General Responsibility Assignment Software Patterns*]
==> bonnes pratiques (peu formalisées mais grandes lignes directrices) , simples à comprendre et intuitifs ==> patterns adaptés dès la fin de l'analyse (non techniques)
- <<**Design Principles**>> (Robert MARTIN , Bertrand MEYER)
==> grands principes objets assez formalisés (règles à respecter) .
Essentiellement liée à la structure générale des modules , cette série de patterns est tout à fait adaptée à la conception préliminaire.
- **GOF** (Gang Of Four)
==> série assez technique de "design patterns" (proches du code)
==> adapté pour la conception (préliminaire et détaillée).
- ...

7.2. "Design Patterns" du GOF

Principaux "Design Pattern" (G.O.F.)

| | Rôle | | |
|-------------------|--------------------|--------------------|--------------------------|
| | Créateur | Structurel | Comportemental |
| Classe / Statique | Fabrication | Adaptateur (stat.) | Interprète |
| | | | Patron de Méthode |
| Objet / Dynamique | Fabrique Abstraite | Adaptateur (dyn.) | Chaîne de responsabilité |
| | Monteur | Composite | Commande |
| | Prototype | Décorateur | Itérateur |
| | Singleton | Façade | Médiateur |
| | | Poids Mouche | Memento |
| | | Pont | Observateur |
| | | Procuration | État |
| | | | Stratégie |
| | | | Visiteur |

G.O.F. ==> Gang Of For (le gang des 4 personnes qui ont lancé les "Design Patterns" dans le monde de la conception orientée objet) ==> Thèse & Livre :

Livre (de référence) fortement conseillé :

DESIGN PATTERNS

Catalogue de modèles de conception réutilisables [traduction française de Jean-Marie Lasvergères.] / Vuibert / Addison Wesley

Auteurs (les 4):

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

7.3. Autres ensembles de designs patterns

- GRASP
- Object Principles (Meyer & Martin)

8. Différences entre "Design Pattern" et "Framework"

Notion de "**Framework**"

Un "Design Pattern" est un modèle qui reste à programmer (en fonction du contexte).

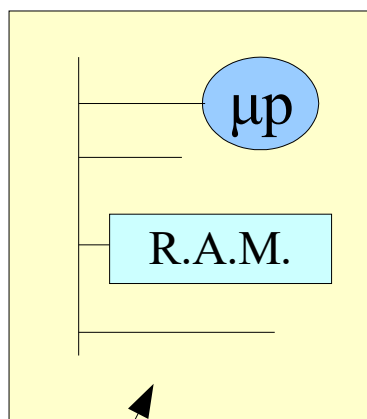
Un "**Framework**" est *en grande partie déjà programmé*: il s'agit d'un **schéma applicatif "pré-cablé"** prêt à recevoir des composants logiciels.

Analogie classique:

Carte mère avec circuit pré-cablé pour recevoir des composants matériels (CPU, Ram, carte PCI).

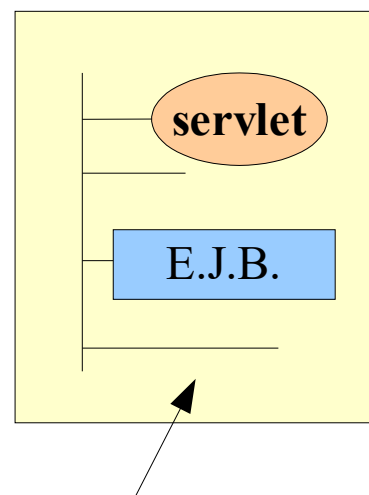
- Idée d'industrialisation.

carte mère (avec circuit imprimé)



Framework et composants matériels

Serveur d'application (avec logique pré-programmée)



Framework et composants logiciels

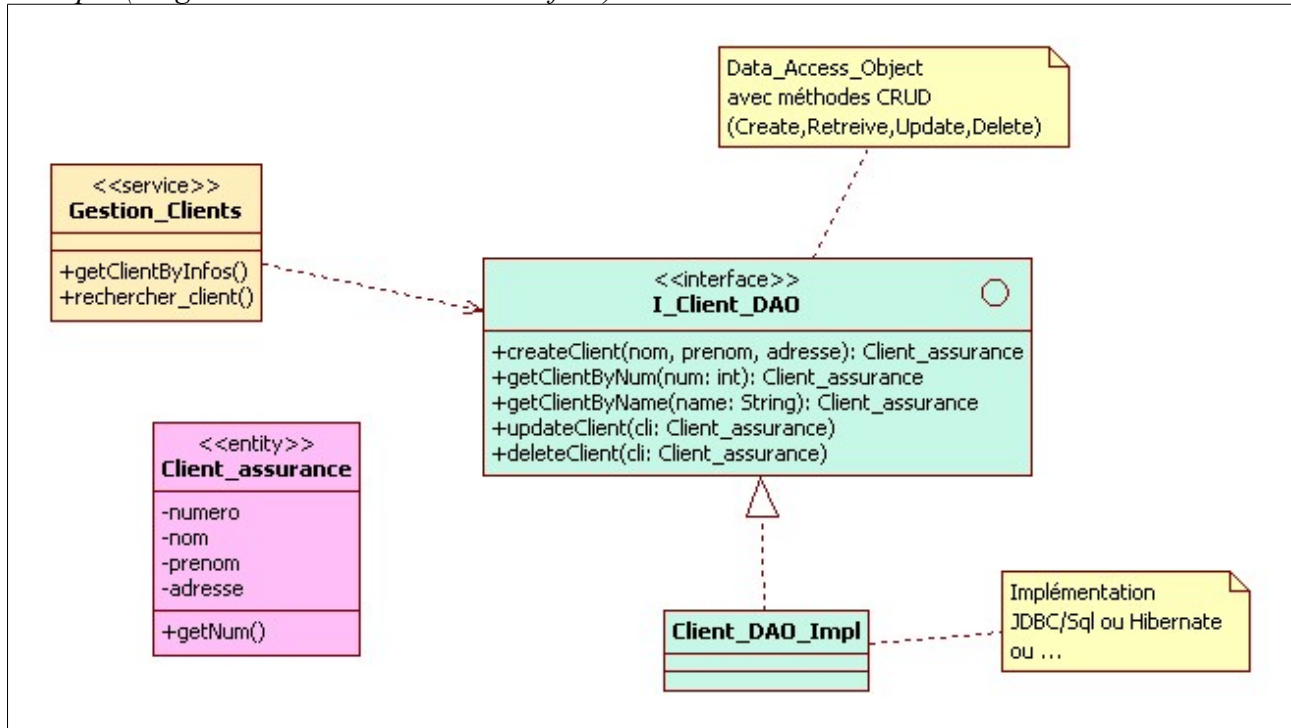
9. Éléments de conception générique

Tout schéma de conception qui peut être ré-appliqué plusieurs fois de façon quasi-systématique peut être considéré comme un élément de conception générique.

A l'inverse, certains éléments de conception très spécifiques (et applicables qu'au sein d'un cas de figure très précis et bien particulier) seront rangés dans la conception dite "détaillée" (au cas par cas).

9.1. Exemple avec interface : (DAO = Data Access Object)

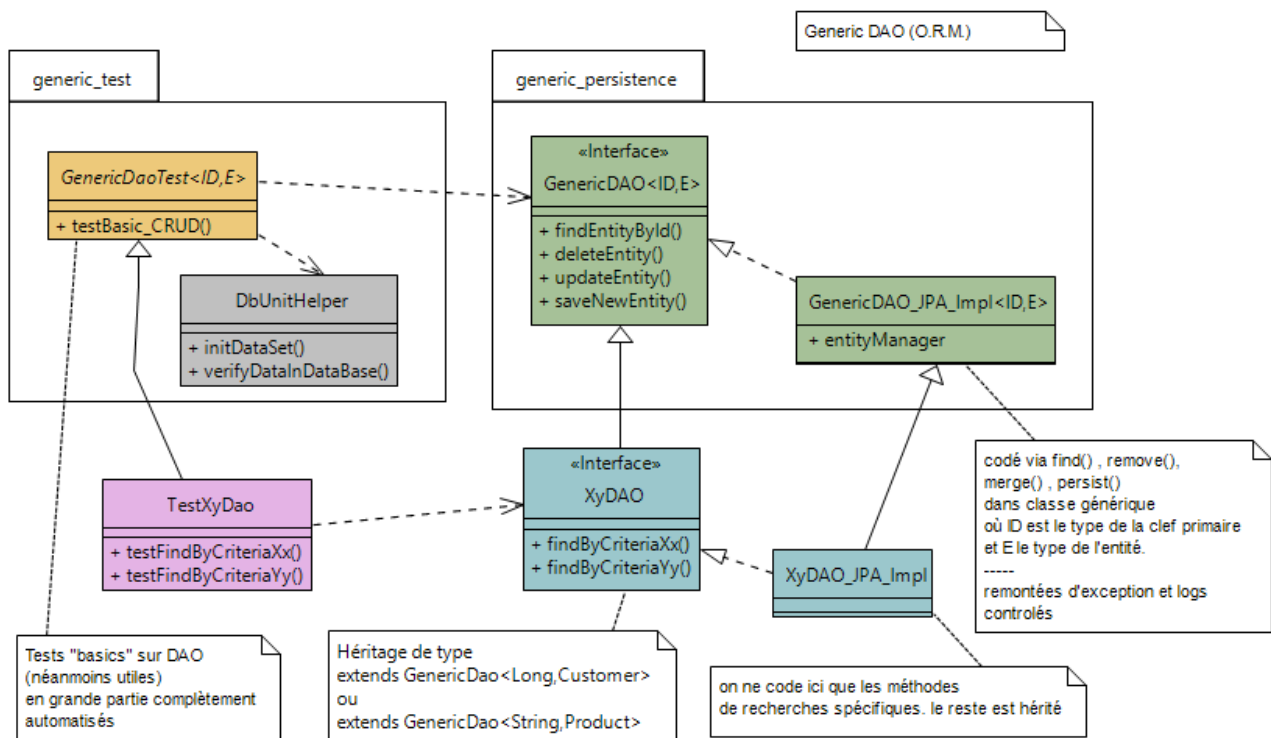
Exemple (diagramme de classes avec interface):



Un DAO est un cas particulier du design pattern "stratégie" : il s'agit d'une stratégie d'accès aux données (ex : via JDBC/SQL , via JPA/Hibernate ,).

En pouvant inter-changer des implémentations "jdbc/sql" , "jpa/hiernate" ou ".." on peut faire des avancées technologiques sans remettre en cause les appels vers le "service rendu".

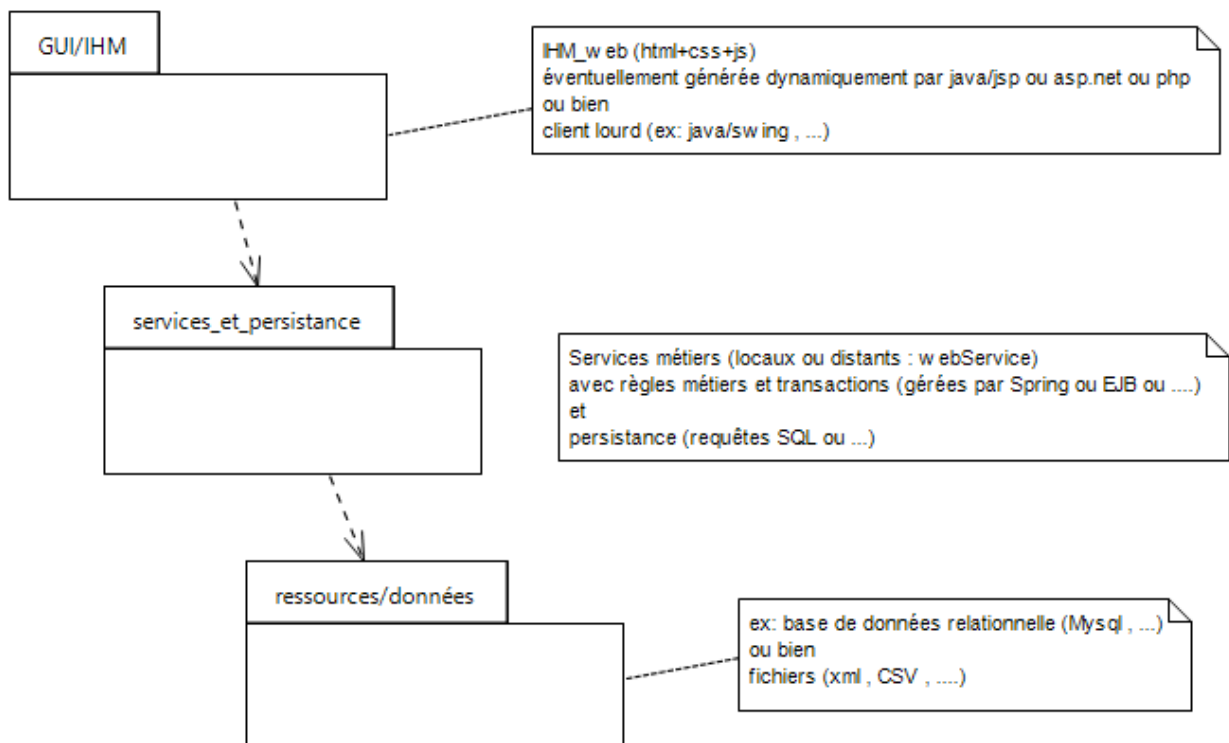
Dao générique :



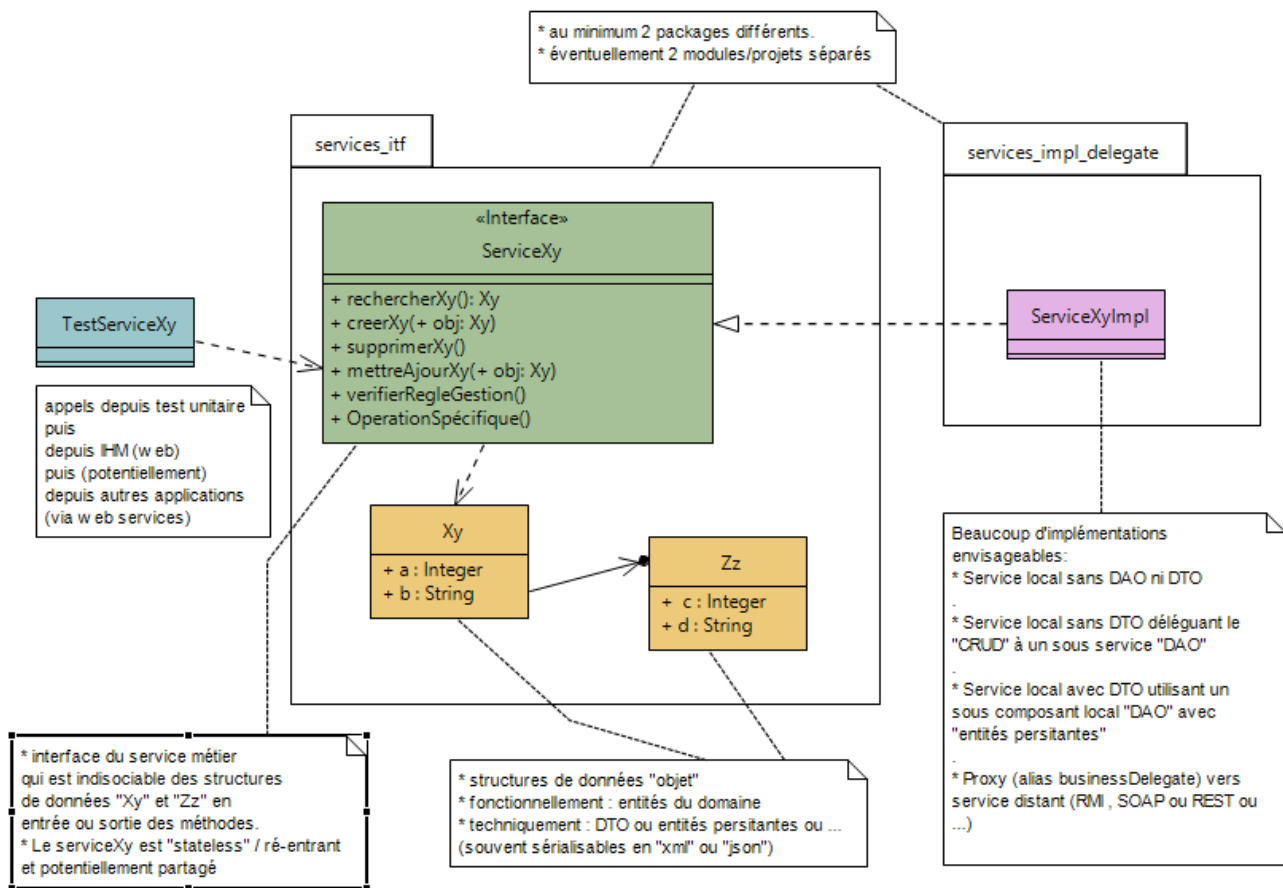
Un DAO générique permet :

- de compacter le code (moins de ligne grâce à l'héritage).
- automatiser les tests les plus simples

9.2. Modélisation des grandes couches logicielles



9.3. Variations autour de l'implémentation des services métiers

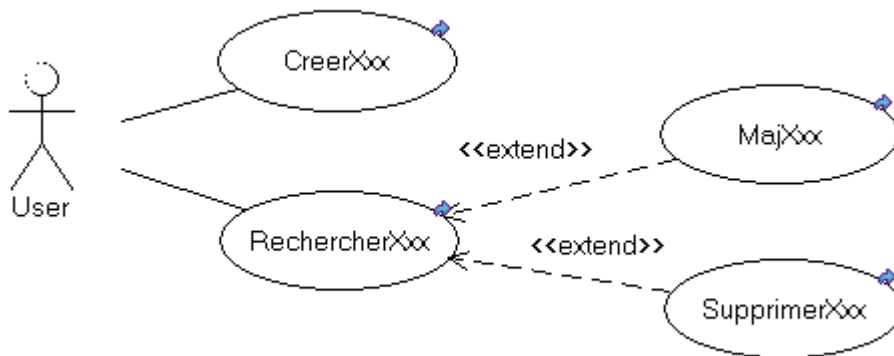


Variations à (potentiellement) faire en TPs:

- version minimaliste (sans DAO ni DTO)
- version avec DAO (mais sans DTO) / mode "DRY"
- version avec DAO et DTO en mode "service et DAO = composants indépendants"
- version avec DAO et DTO en mode "DAO = sous composant du gros composant service avec éventuelle optimisation "DAO utilisant "entité persistante" avec sous-DTO imbriqué"
- version avec délégation de l'implémentation du service (ex : vers web-service "soap").

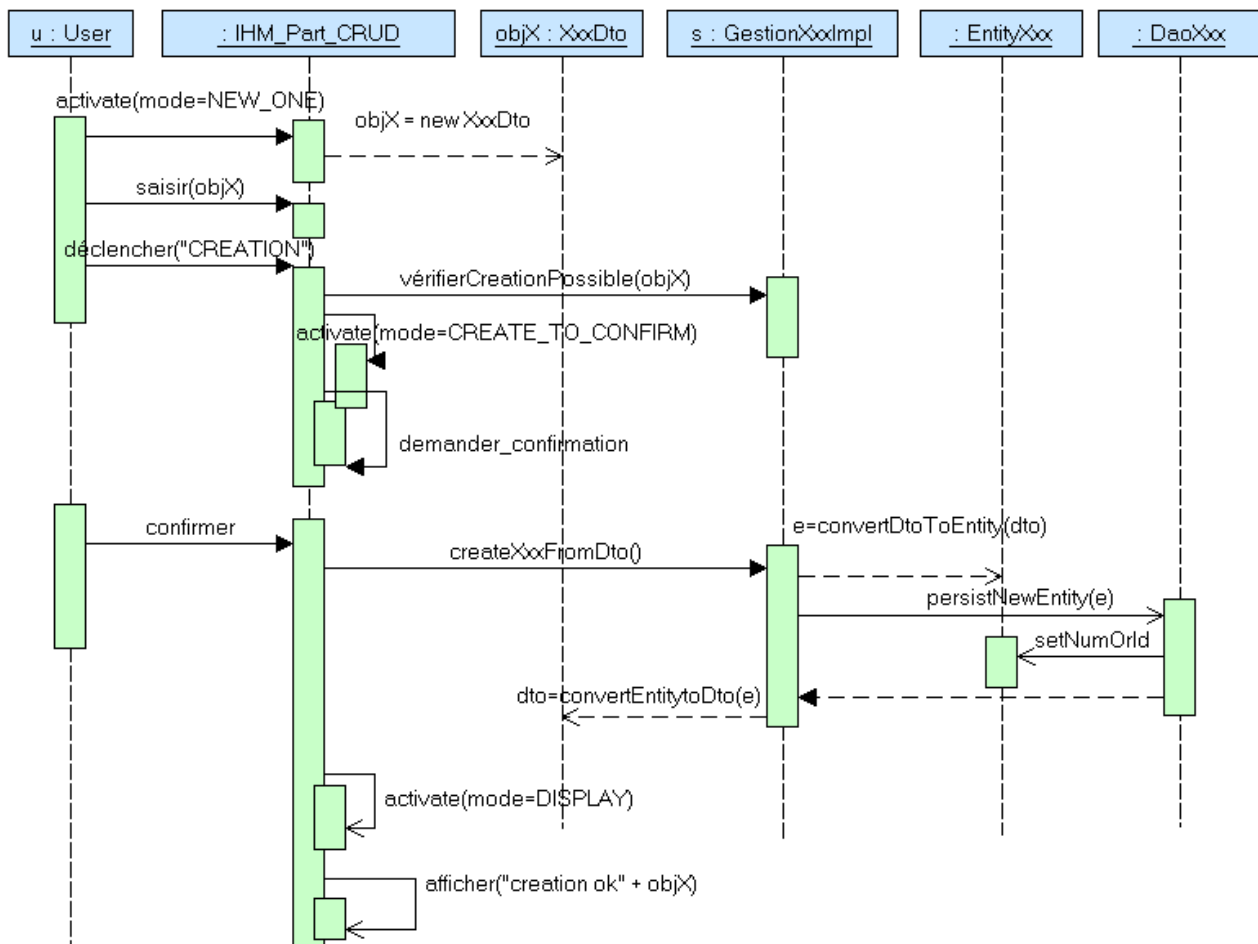
9.4. Séquences génériques accrochées à des "uses cases techniques"

Pour la syntaxe et l'idée : un exemple de diagramme de "Uses Cases" génériques :

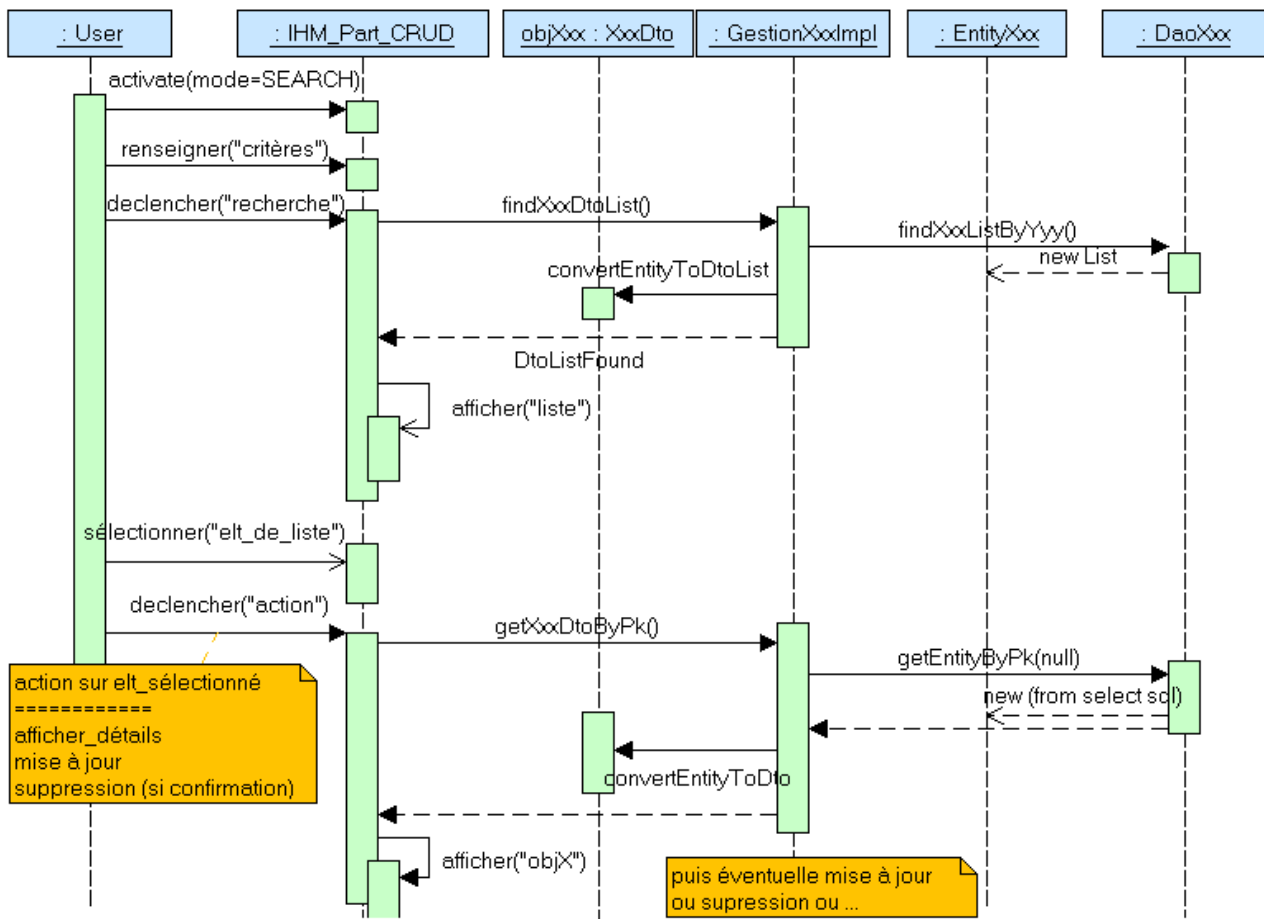


Ceci permet (au sein d'un logiciel UML élaboré) de naviguer intuitivement vers des diagrammes de séquences techniques montrant une façon de réaliser une des opérations CRUD classiques.

Exemple "seq_create_xxx_avec_dto_et_confirmation" :



diag_seq_rechercher_xxx_with_dto (liste + sélection + details) :



Remarque :

Ce diagramme de séquence UML (effectué il y a déjà quelques années) montre **une façon (parmi plein d'autres) d'effectuer une recherche** en s'appuyant sur une structure "service + DAO" avec "DTO + entité persistante" .

La séquence exacte modélisée dépend de tout un tas de choix techniques (plus ou moins pertinents ou discutables). Ce genre de diagramme permet de discuter sur les avantages et inconvénients d'une solution technique.

Une fois, le choix technique effectué par l'architecte ou le concepteur , ce genre de diagramme peut être vu comme une sorte de guide (ou plan type) pour le développeur.

III - Projection (avec ou sans M.D.A.)

1. Projection du fonctionnel dans technologies

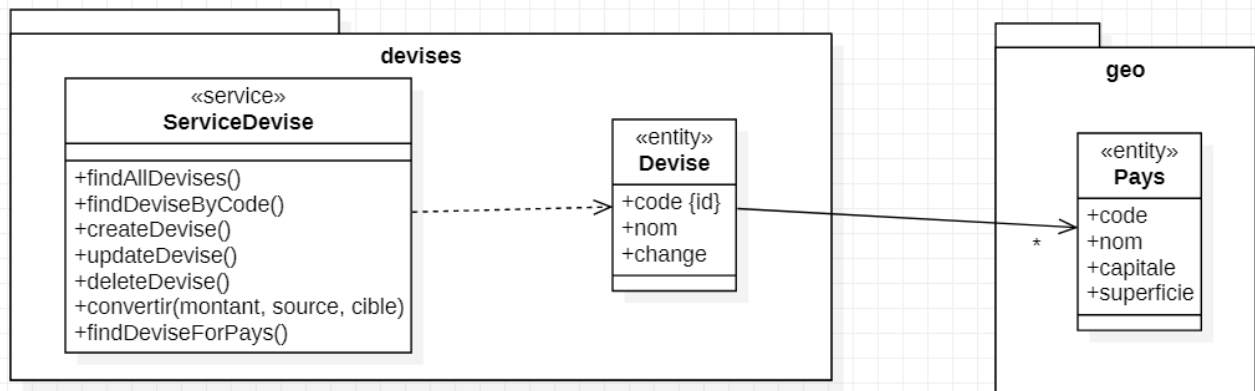
Les activités d'expression des besoins fonctionnels et d'analyse fonctionnelle mènent généralement à un jeu de diagrammes UML qui précisent assez bien :

- les structures de données nécessaires à l'application (en base , pour les échanges/communications)
- les rôles des personnes qui vont utiliser le logiciel (acteurs des "cas d'utilisation")
- les principaux "services métiers" de l'application (avec méthodes de traitements essentielles)

Mais tout ceci reste très éloigné de la structure orientée objets exacte d'une application qui dépend quant à elle de :

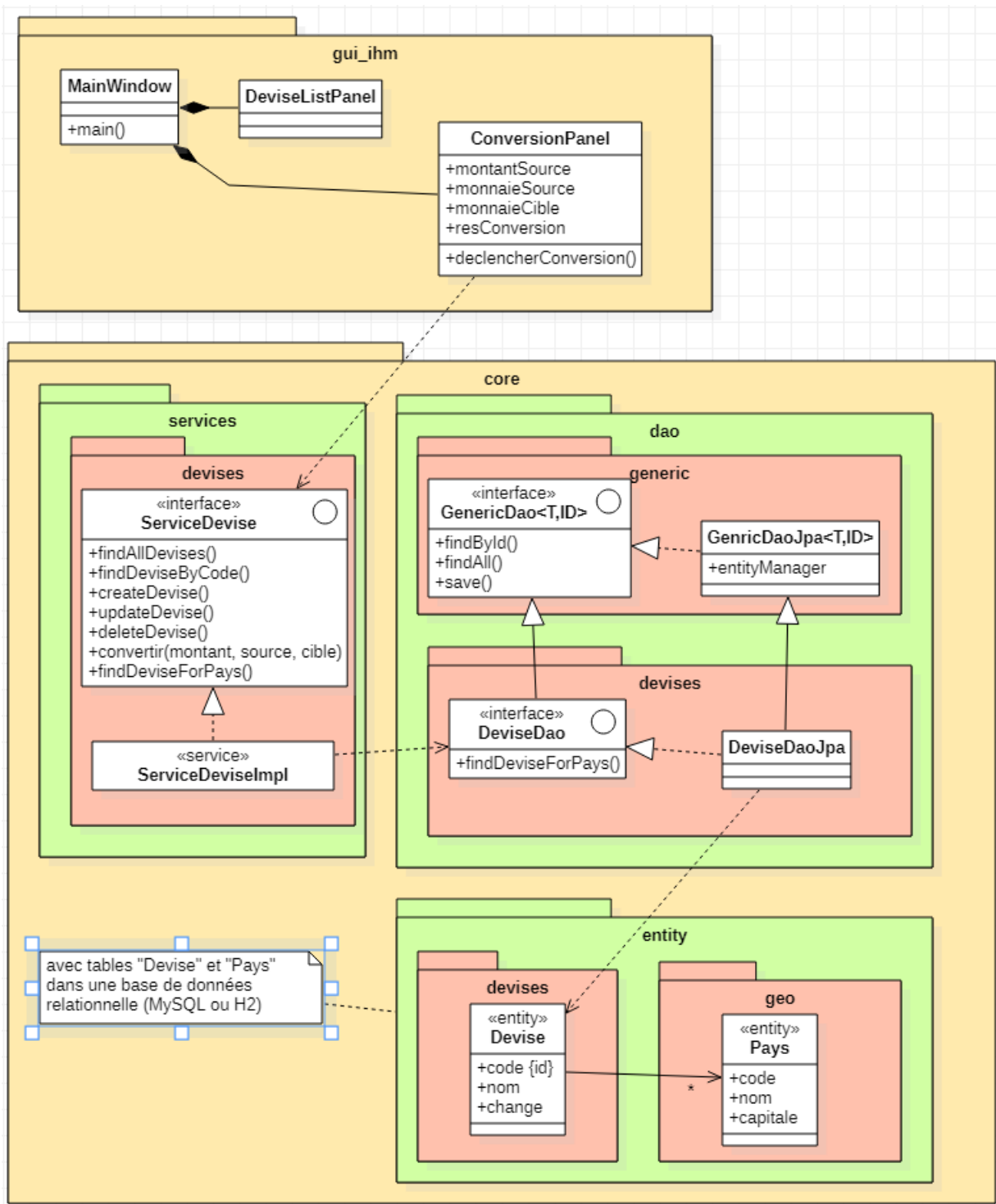
- l'architecture globale (assemblage de technologies , éventuellement distribuées)
- du choix du ou des langage(s) de programmation (avec types de données spécifiques : int ou Integer en java , number en javascript/typescript , ...)
- du choix des frameworks (ex : JPA/Hibernate , Spring , ...)
- du choix des "design patterns" mis en oeuvre (code simple ou évolué)
-

Pour bien comprendre l'étendue des alternatives , le mini exemple ci-dessus va montrer une ou deux projection(s) d'une même analyse fonctionnelle dans une (ou 2) des 36000 combinaisons de technologies possibles :



.../...

Projection possible en java embarqué et ihm multi-fenêtrées (non web):

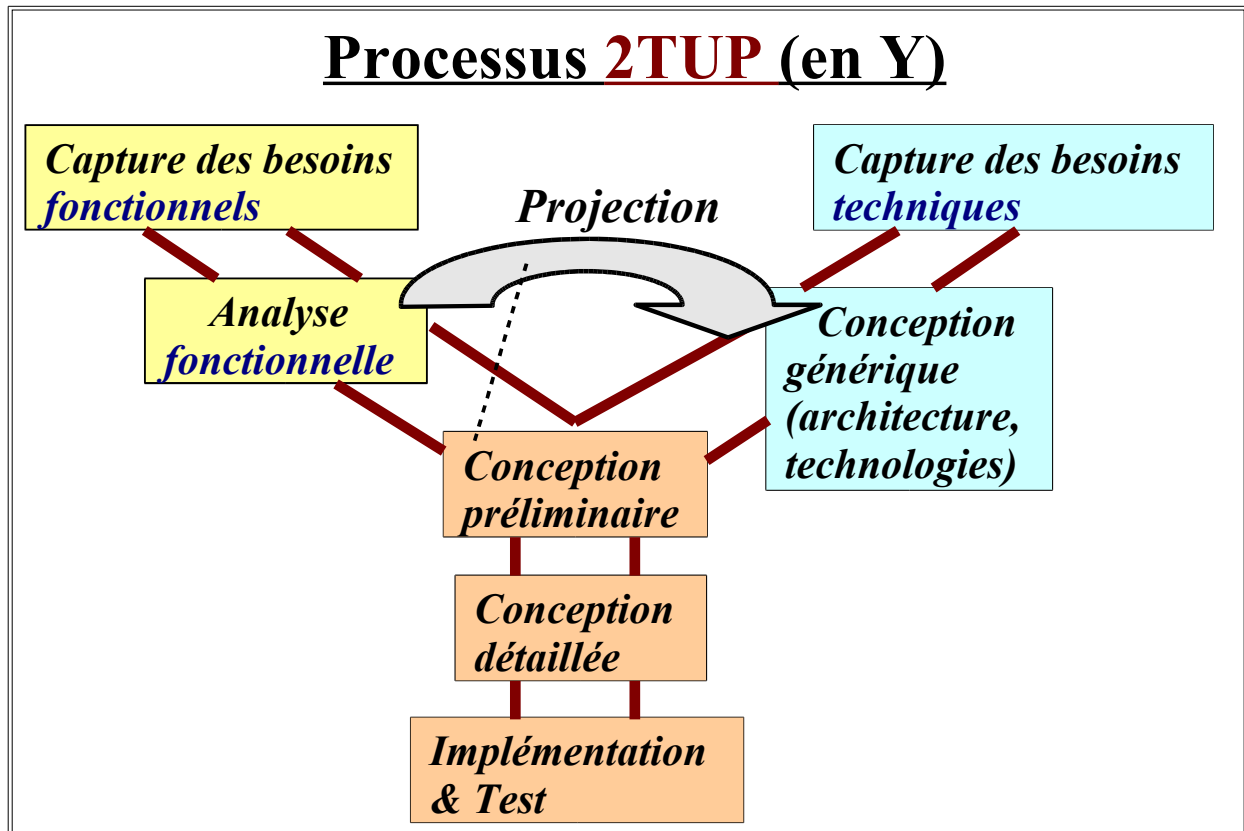


.../...

Future deuxième projection possible en mode "font-end web" + "back-end" ...

2. Objectifs de la conception préliminaire

Il s'agit ici de **projeter** le résultat de l'analyse au sein d'une architecture logique et technique définie durant l'étape "architecture" (ou "conception générique").



Principal résultat de la conception préliminaire (projection):

n "packages fonctionnels" * m "couches/niveaux techniques"

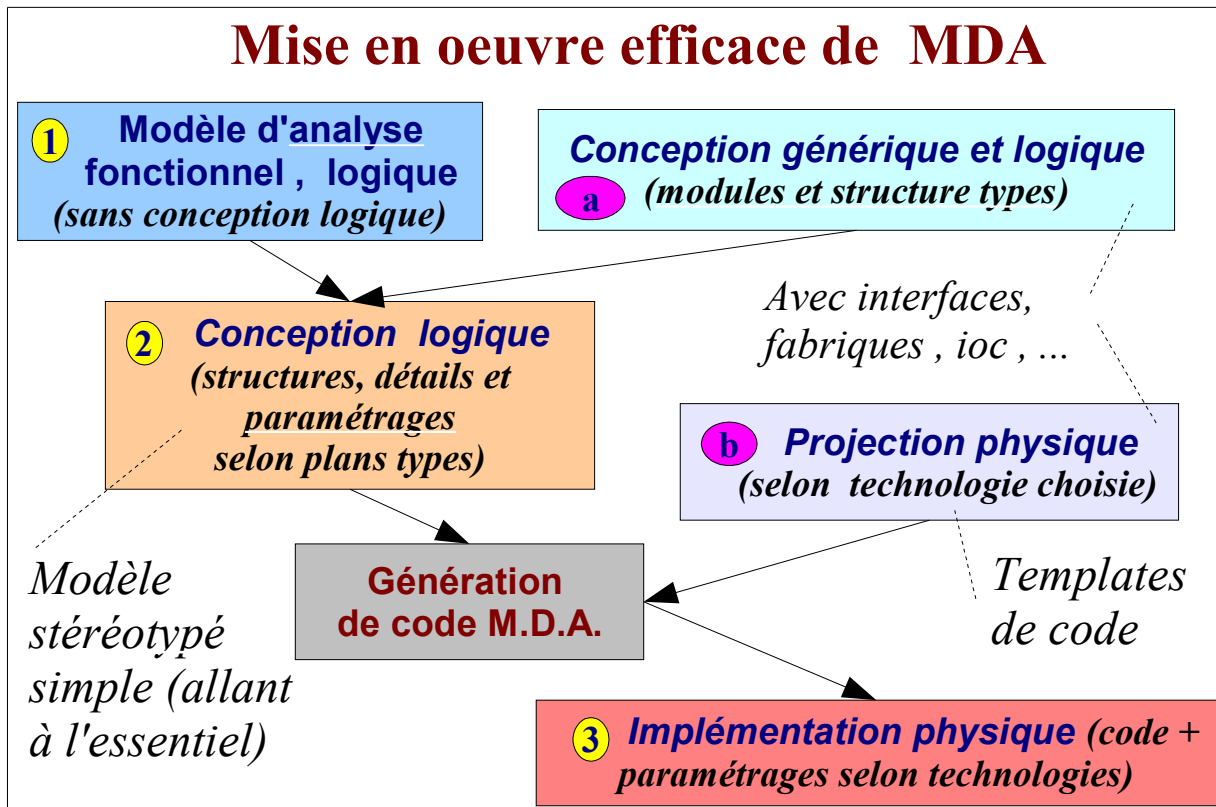
====> n*m packages dans le code à réaliser/produire:

- fr.xxx.yyy.AppliA.partieIHM.web
- fr.xxx.yyy.AppliA.*partieFonctionnelleA*.service
- fr.xxx.yyy.AppliA.*partieFonctionnelleA*.entity
- fr.xxx.yyy.AppliA.*partieFonctionnelleB*.service
- fr.xxx.yyy.AppliA.*partieFonctionnelleB*.entity
- ...

==> Il vaut mieux appliquer systématiquement certaines règles de projection (via MDA ou) pour être efficace/rapide .

3. Utilisation de MDA en conception

Approche efficace et pragmatique de la conception prenant en compte MDA:



Concrètement :

- 1) reprise des diagrammes de classes fonctionnels (fruits de l'analyse)
- a) Modélisation d'architecture générique en UML (ex : Architecture JEE avec services et DAO)
- 2) ajout de certains stéréotypes (ex : <<service>> , <<id>> , <<entity>> en vue de paramétrer la future génération de code
- b) Templates "accéléo_m2t" (modèles de code java à générer selon stéréotypes UML)
- 3) amélioration manuelle du code automatiquement généré par le plugin eclipse "accéléo_m2t" .

Point clefs :

- **stéréotype UML** = paramétrage UML interprété (par un développeur ou par un générateur automatique MDA) lors de la génération de code
- **annotations (java, php ou c#)** = paramétrage généralement interprété par un framework au runtime (lors de l'exécution du code)

Certains stéréotypes peuvent quelquefois être retranscrits en annotations proches.

Exemples :

<<id>> ---> @Id
 <<service>> ---> @WebService ou ...
 <<entity>> ---> @Entity

IV - Conception modulaire (composants)

1.1. Objectifs de la conception modulaire

==> de façon à ne pas aboutir à un **modèle trop spécifique** (liés à une ou plusieurs technologies précises), on aura tout intérêt à bien dissocier les interfaces et les implémentations des différents modules.

En fin de conception préliminaire et modulaire, on doit normalement aboutir à :

- **une collaboration inter-modules relativement rigide** (spécifier par des interfaces bien précises).
- **des implémentations de module(s) très libres** (générées automatiquement via MDA ou bien spécifiées au cas par cas lors de la conception détaillée)

1.2. Identification des modules

Composant(s)

- tailles très variables (module/système , pièce/élément/partie ,)
- éventuelle notion de classes
 - * <<**principale**>> (coeur d'un concept métier ou interface Accès/raison d'être d'1 module)
 - * <<**secondaire**>> (sous parties dépendantes ou référencées [ex: ligne de commande , adresse, ...])

D'où l'idée d'associer un composant principal (éventuellement de type façade ou bien composite) à chaque concept métier .

Un processus métier complet (accessible via un service) peut éventuellement être vu comme un gros composant composite (avec sous parties éventuellement distribuées/pilotées à distances)

Dans tous les cas, pour identifier les modules de l'application, les 2 principaux critères sont (dans l'ordre d'importance) les suivants:

1. **cohérence/responsabilité fonctionnelle (modules métiers)**
2. **responsabilités techniques (sous modules liés aux couches techniques)**

En informatique de gestion, l'un des points les plus importants de la conception préliminaire est de bien identifier :

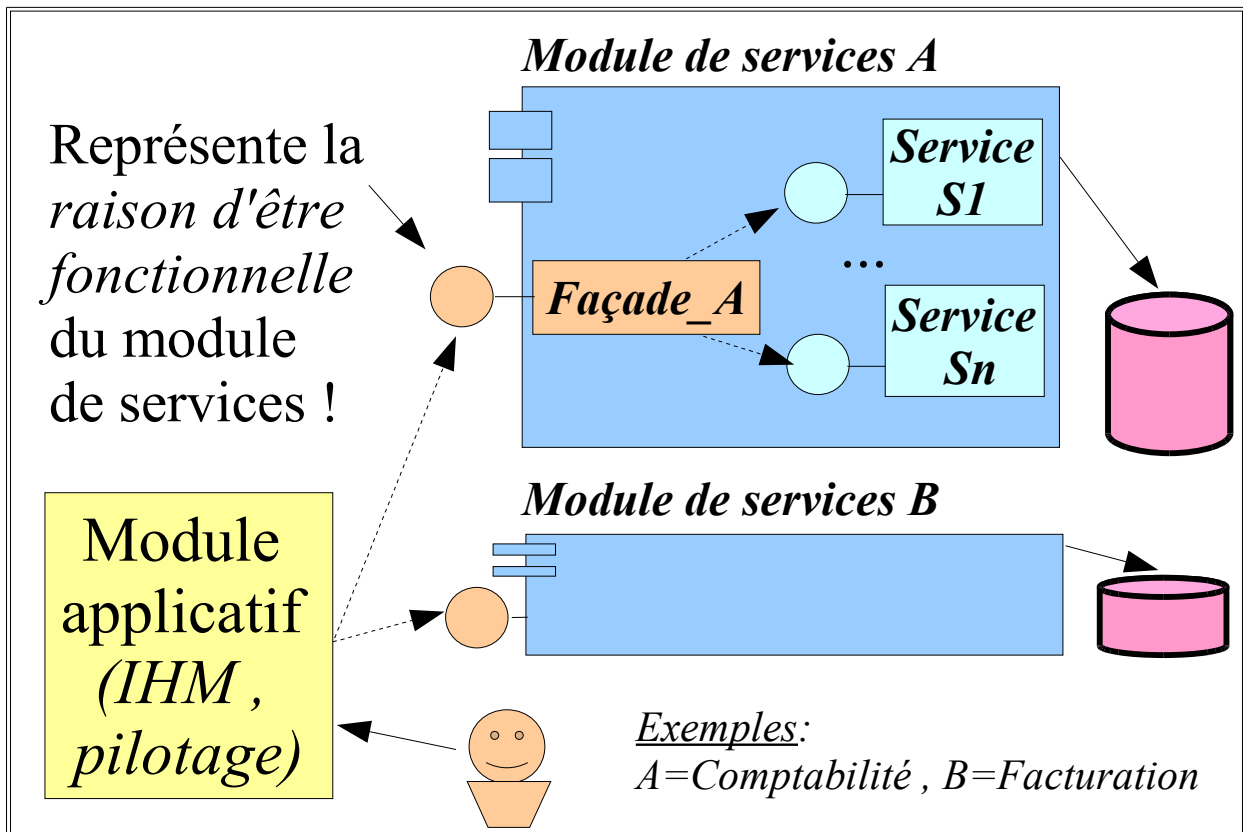
- **les services métiers**
- **les structures de données en entrées/sorties**
(au niveau des méthodes des services métiers)

Il s'agit en effet du principal contrat entre les parties "back-office" et "front-office / ihm".

Quelques conseils pertinents à ce niveau:

- **faire simple** (pour évolutions éventuelles vers d'autres technologies [ex: Service Web, ...])
- **identifiant (opaque)** plutôt que référence liée à une technologie.
- **confronter les besoins réels des modules clients et les offres de services des modules serveurs** (complétude ?, ...) un peu comme on négocie un contrat (il faut tenir compte des 2 parties , gouvernance, ...).

2. Description des modules (interfaces,façades)



D'un point de vue fonctionnel, une façade donne du sens à un module de service. Son nom doit donc être bien choisi (pour être évocateur).

D'un point de vue technique, une façade n'est qu'un nouvel élément intermédiaire de type "Façade d'accueil" qui servira à orienter les clients vers les différents services existants.

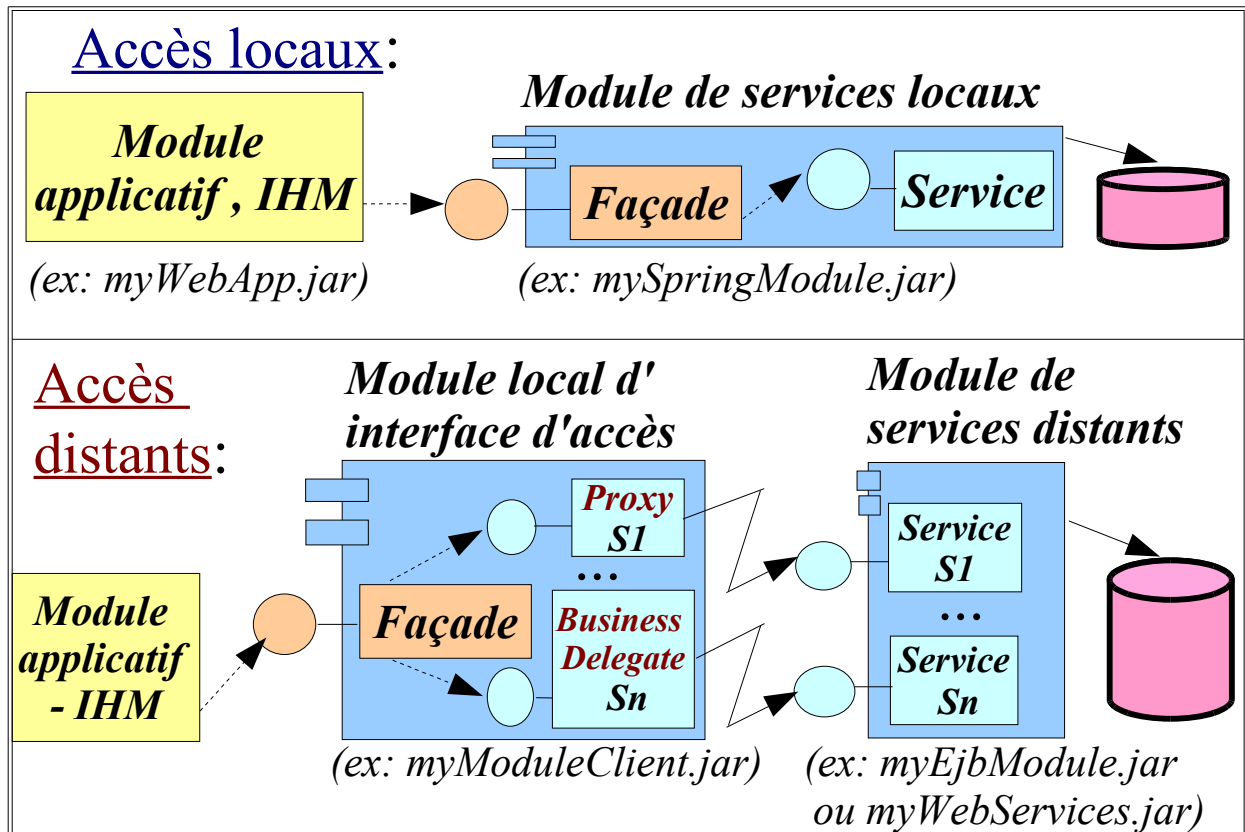
Exemple:

| Façade_Comptabilité |
|--------------------------------------|
| |
| .getServicePostesComptables() |
| .getServiceBilan() |
| .getServiceJournal() |
| .getServiceGrandLivre() |
| ... |
| .getServiceN() |

utilisation:

```
facadeCompta.getServiceBilan().xxx()
```

```
facadeCompta.getServiceGrandLivre().yyy();
```



Derrière une façade (très souvent locale) on peut éventuellement trouver des services distants.

Un "proxy" est un représentant local d'un service distant .

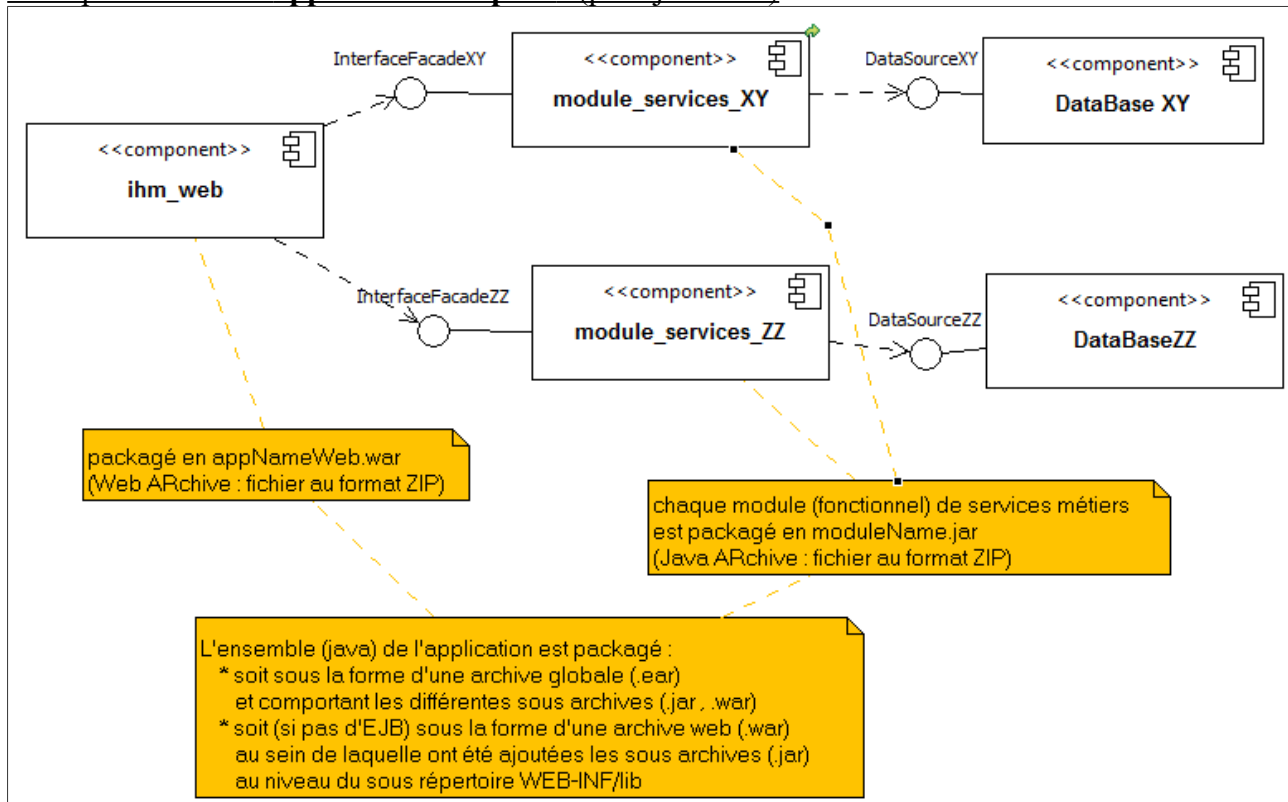
Ce terme (plutôt technique) désigne assez souvent du code généré automatiquement (à partir d'un fichier WSDL par exemple).

Pour bien contrôler l'interface locale d'un service distant (de façon à n'introduire aucune dépendance vis à vis d'une technologie particulière), on met parfois en oeuvre des objets de type "**business delegate**" qui cache dans le code privé d'implémentation tous les aspects techniques liés aux communications réseaux:

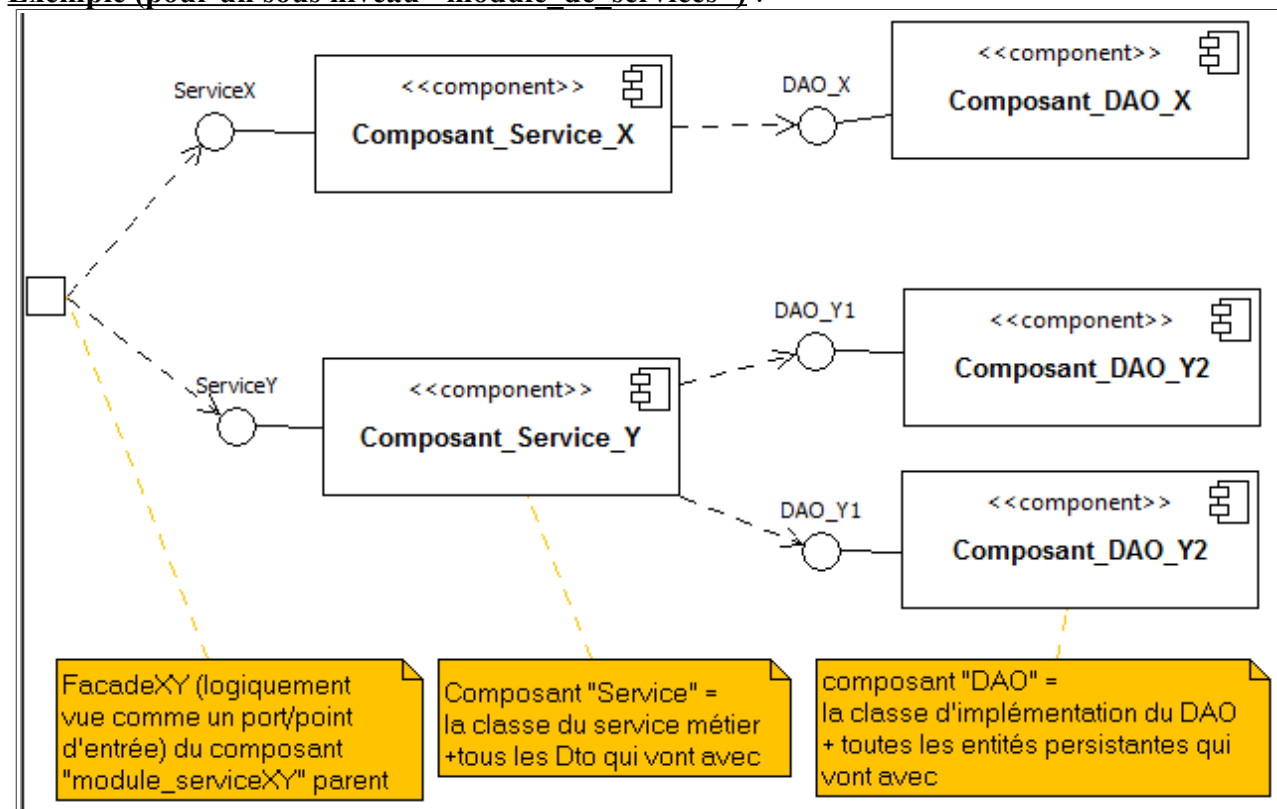
- localisation du service , connexion (lookup(EJB) ou ...)
- préparations/interprétations des messages/paramètres .
- ...

3. Modélisation des dépendances inter-modules (diagramme de composants UML)

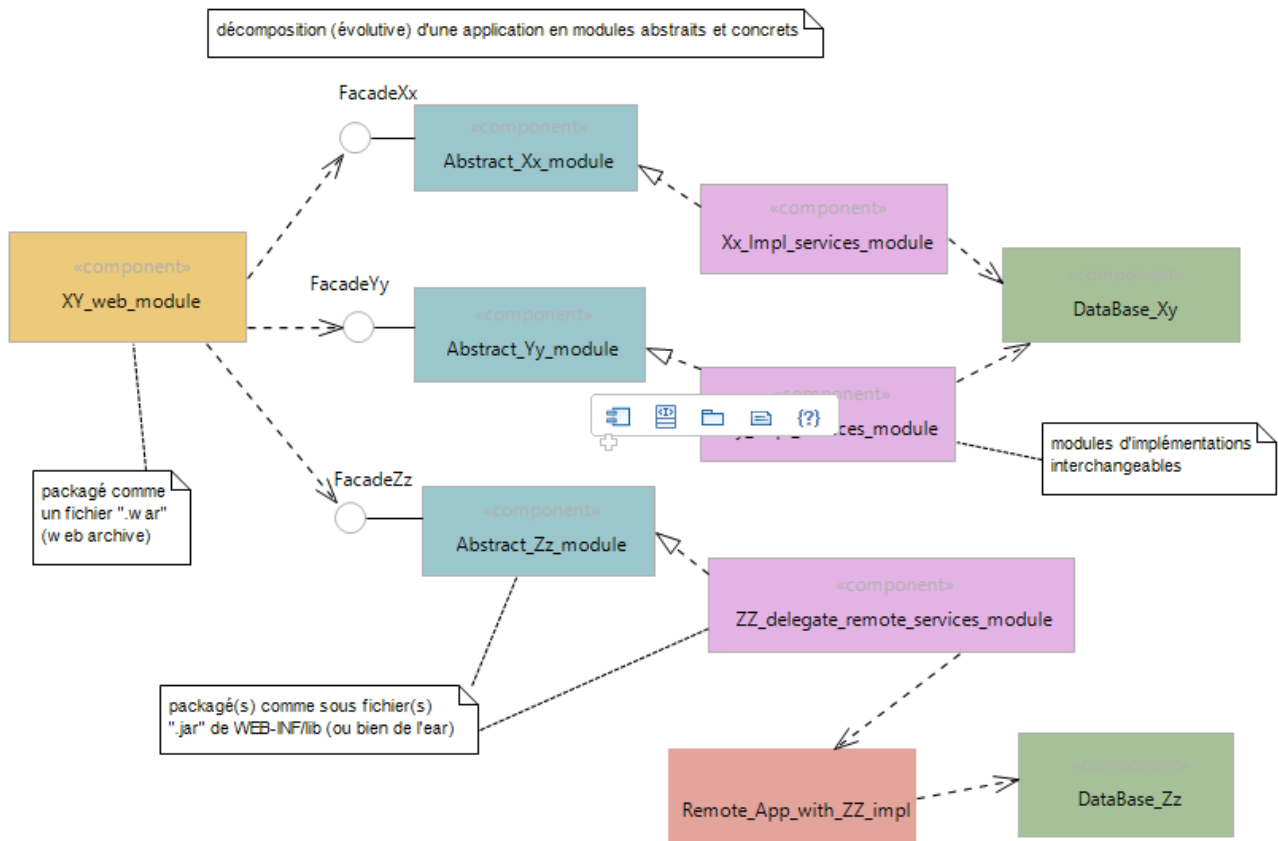
Exemple de niveau "application complète" (pour java/JEE) :



Exemple (pour un sous niveau "module_de_services") :



3.1. Diagramme de composants avec dépendances affinées



Un module abstrait comporte essentiellement :

- des **interfaces**
- des types de données en entrées/sorties des appels de méthodes (ex : DTO)
- une éventuelle "façade"

Un module d'implémentation comportera :

- le code effectif d'implémentation (en local)

ou bien

- un code qui délègue les traitements vers des services extérieurs

V - Éléments de conception détaillée

1. les conceptions détaillées

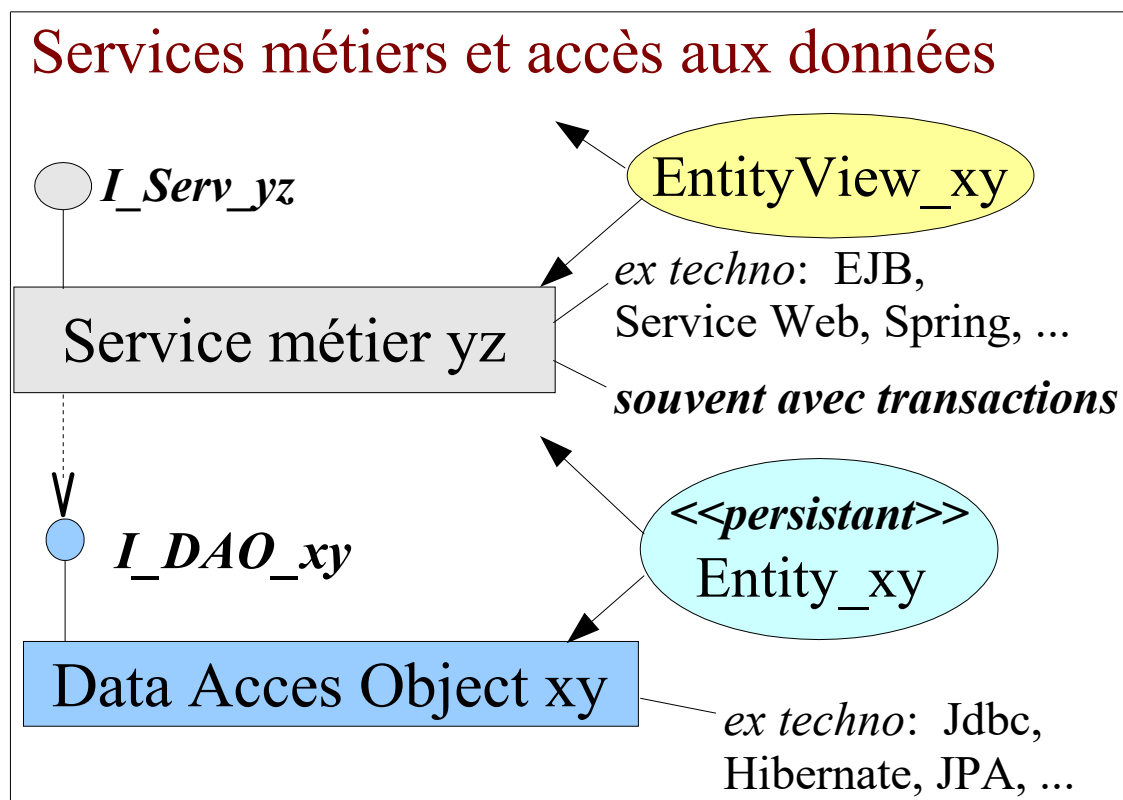
Lors de la **conception détaillée**, on doit choisir et spécifier des implémentations assez précises qui dépendent très grandement des technologies et framework employés.

Parmi les principaux points à détailler, on peut citer:

- **Les fonctionnements internes de l'interface graphique (IHM)**
 - * navigations, flots d'actions, structure composite (s'il y a lieu)
 - * framework choisi (et mécanismes)
- **Les types de données précis (int, String,)**
- **Les mécanismes de persistance.**
 - * api / framework choisi (et mécanismes)
 - * structure de la base de données, mapping objet/relationnel

NB: Dans beaucoup de cas, certains éléments de la conception détaillée peuvent être ré-utilisables plusieurs fois et être donc "modélisés de façon générique" (pour paramétrer ensuite de la génération de code MDA).

2. Conception détaillée des services métiers et de la persistance



Vues "métier" (alias DTO / VO)

Une "*vue métier*" est un *objet sérialisable* et qui servira à faire communiquer un service métier avec un client (Web ou ...) potentiellement distant. [synonymes classiques: *Value Object* ou *Data Transfert Object*].

Outre son aspect technique lié aux appels éventuellement distants, une vue métier est utile pour que:

- * La couche N (Appli-Web,...) ne voit pas directement les entités persistantes remontées par la couche N-2 (D.A.O.).
- * L'application puisse manipuler des vues souvent simplifiées (sans tous les détails des tables relationnelles ou des objets persistants).

2.1. Structure de la base de données et mapping objet/relationnel

Ce sujet est en soi très vaste .

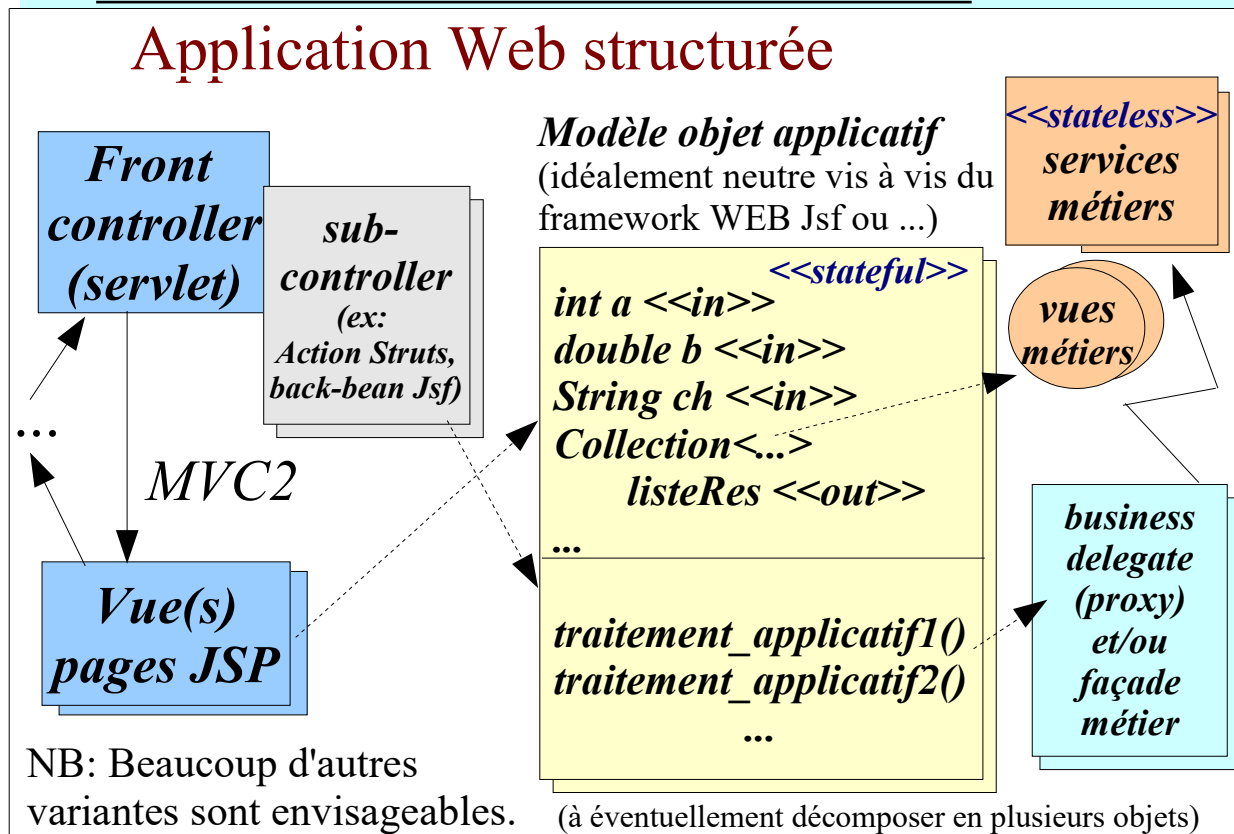
L'idéal est d'étudier en détail une technologie de mapping objet/relationnel tel qu'hibernate pour voir ce qu'il est possible de faire. ==> la modélisation UML sera alors ensuite plus aisée et naturelle.

Cependant, sans trop rentrer dans les détails, on peut tout de même indiquer 3 grands axes pour conduire la mise en oeuvre:

| <i>Approches</i> | <i>Principes et caractéristiques</i> |
|--|--|
| top-down (objet ==> relationnel) | On effectue d'abord une modélisation complètement objet (UML) en ajoutant (en UML ou java) quelques annotations de type "pk (primary key)" et l'on utilise ensuite un générateur de structure relationnelle compatible (DDL). |
| down-top (relationnel ==> objet) | On effectue une modélisation non objet pour la base de données (ex: MCD de Merise avec AMC-Designor) et l'on utilise ensuite un générateur de structure objet compatible (ex: HibernateSynchronizer). |
| meet-in-the-middle (séparés puis assemblés) | On effectue deux modélisations séparées (Objet/UML et relationnelle/MCD) puis on confronte les 2 structures en effectuant un mapping spécifique. => c'est la seule solution si l'on part de 2 existants |
| <i>mixte</i> | Lorsque l'on a le choix , une solution mixte de type "top-down" (pour dégrossir) puis "meet-in-the-middle"(pour peaufiner) est la plus naturelle pour les concepteurs maîtrisant bien UML et les technologies O.R.M. (ce qui n'est pas toujours le cas : base ou modèle MCD existant , habitudes MCD , ...). |

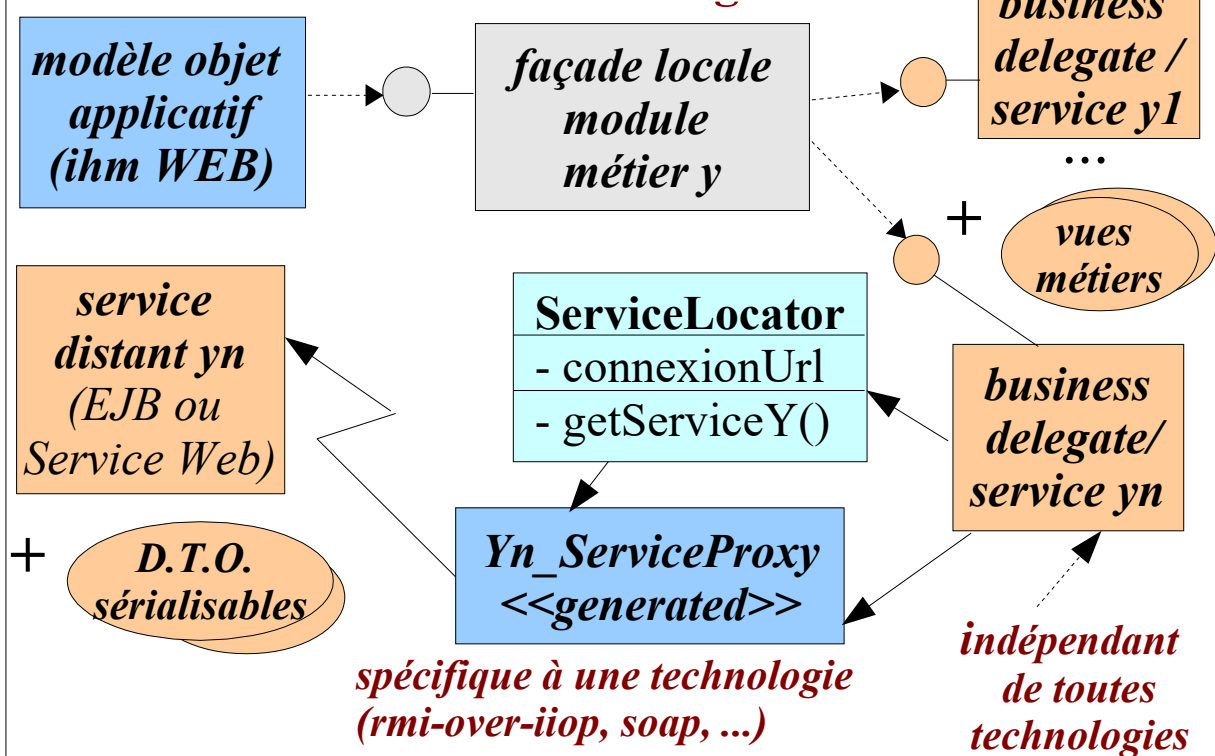
3. Conception détaillée de l'IHM (couche présentation)

3.1. Vue d'ensemble sur les constituants d'une IHM

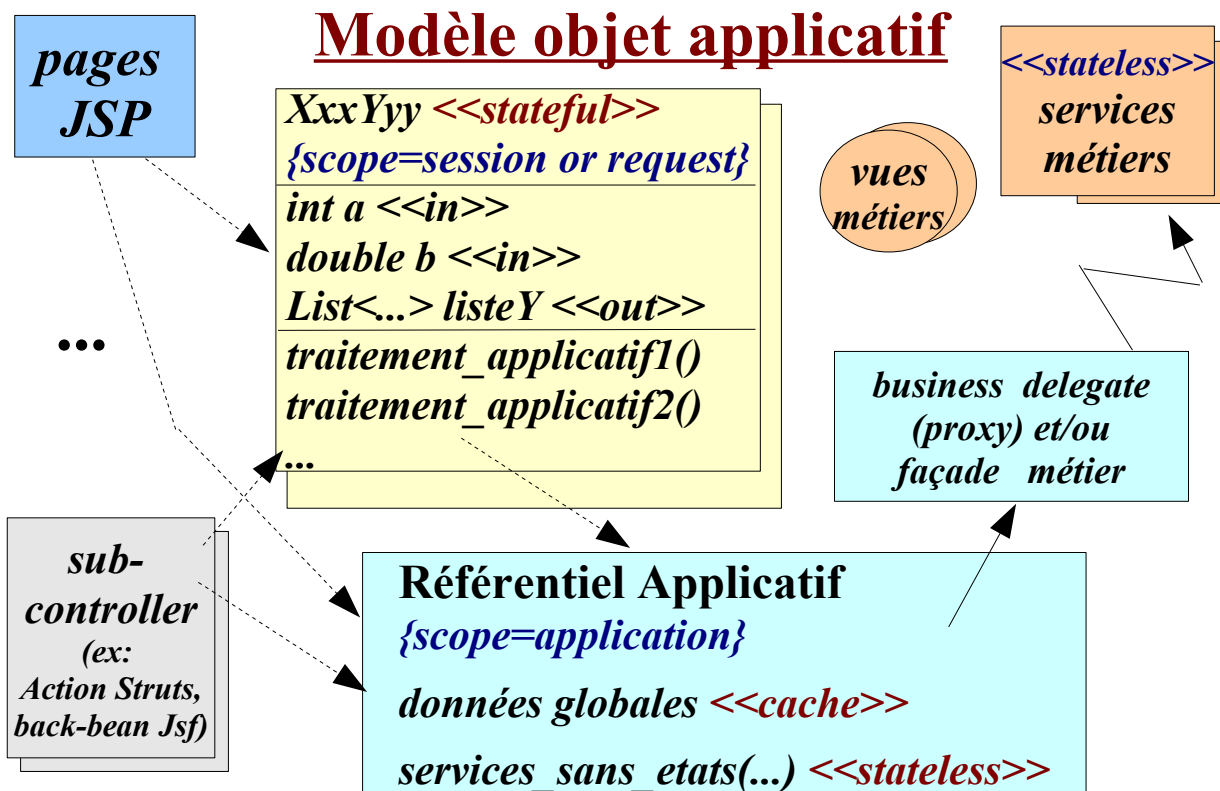


3.2. Conception détaillée des appels distants (RPC)

Facade locale et "business delegate"



3.3. Conception détaillée du modèle applicatif



Un objet central de type "**Référentiel_applicatif**" *retournant des données fixes (en cache pour obtenir de bonnes performances)* et proposant une liste d'opérations sans états (du type `getLabelXFromCodeY(...)`, ...) peut être très pratique pour structurer le modèle applicatif (directement utilisé par l'IHM au sein du modèle MVC).

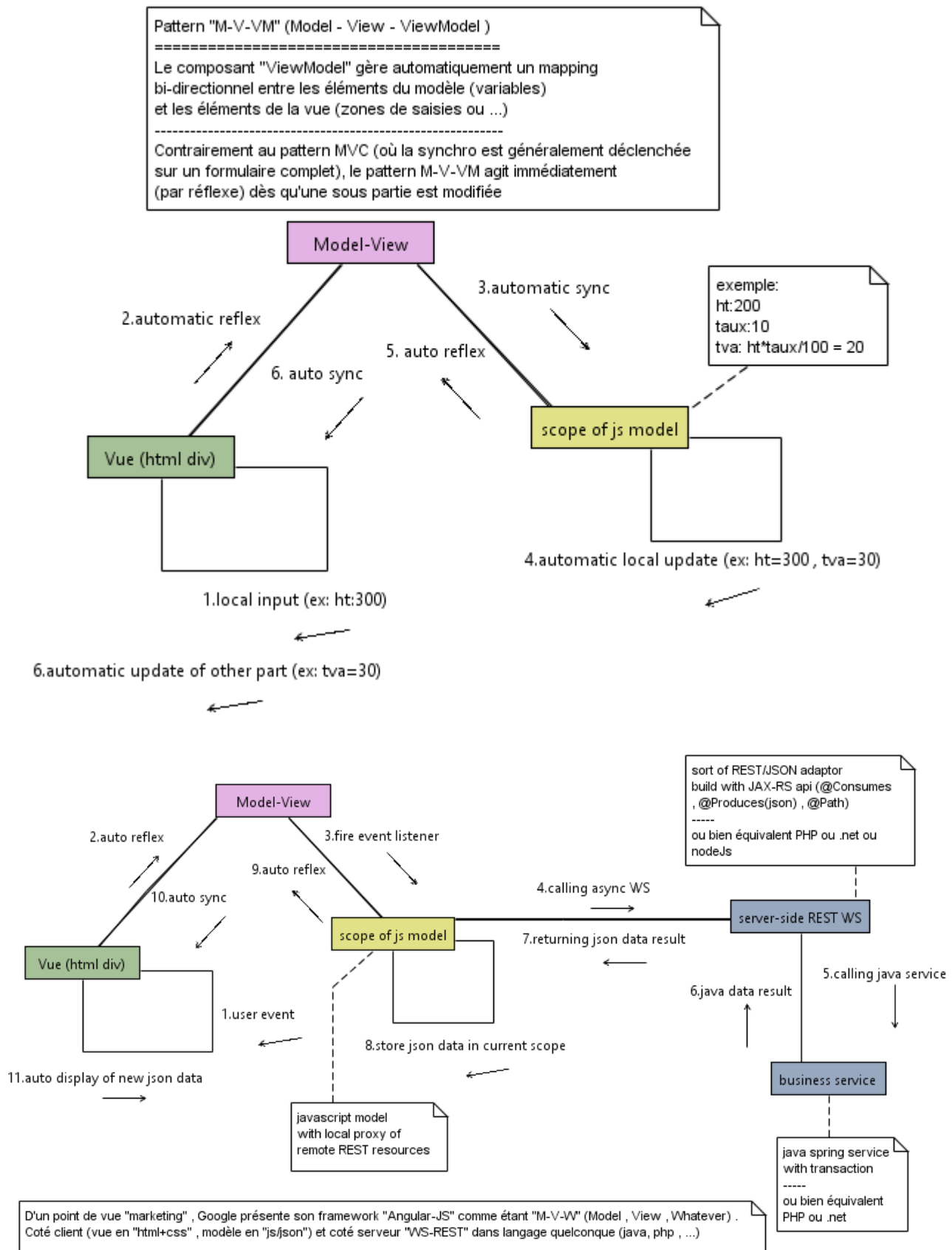
Important :

L'utilité (ou pas) d'un référentiel applicatif dépend essentiellement de la taille de l'application à développer et du contexte technologique .

Des intermédiaires de type "référentiel" et/ou "façade" sont utiles dans le cadre de grosses applications (avec plein de services) et sont quelquefois inutiles sur de petites applications ou bien si l'on utilise des technologies très sophistiquées qui automatisent les liaisons (ex : CDI de JEE6) .

3.4. Comportement détaillé d'une IHM moderne avec M-V-VM

Les diagrammes de communication UML2 ci dessous montrent le principe de fonctionnement d'un framework (ex : *AngularJs* , ...) basé sur le design pattern "M-V-VM" :



4. Quelques notations détaillées et/ou pointues

Autres détails sur les caractéristiques d'une classe

+ compteur : int = 0

valeur initiale

- directions[4]:string

tableau

+ f1([in] num : int , [out] chRes : string) : int

+ afficher() { **abstract** }

méthode abstraite (sans code)

+ getValueX() { **query** }

requête ne modifiant pas l'objet

+ setTauxInteret(taux: float)

*méthode de classe que l'on peut
invoquée sans instance.*

5. Templates/Generics (notations avancées, ...)

Utilité des templates/generics (classes paramétrées)

Préliminaire: différence importante entre java et le c++:

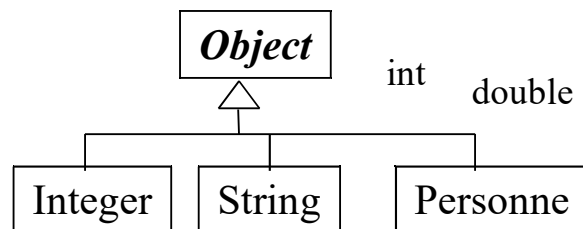
Un vector en C++ est dépendant du type des éléments qui y seront rangés

→ `std::vector<int>` ,
`std::vector<Personne*>`

En C++, chaque nouvelle classe est un type de donnée complètement déconnecté des autres :

int double Personne C2

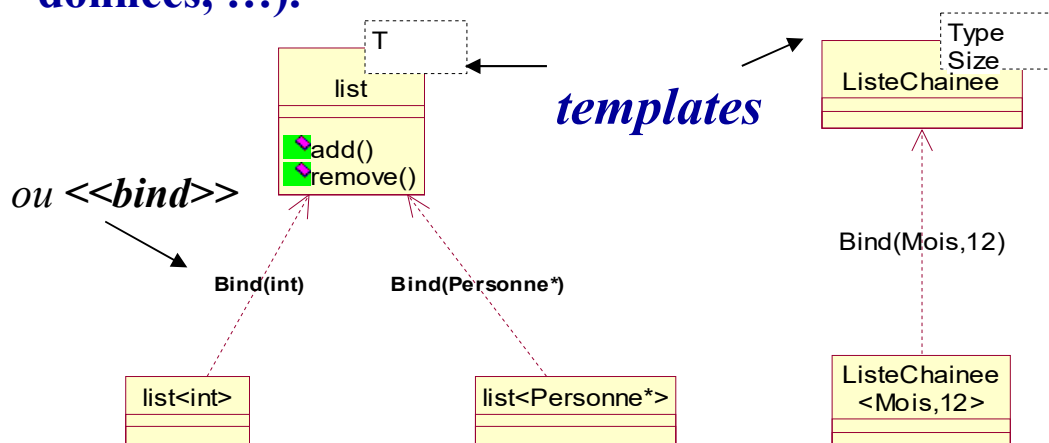
En **Java**, toute nouvelle classe hérite systématiquement de Object :



La classe prédéfinie *ArrayList* de java peut servir à stocker des références sur des sous classes quelconques de Object. ==> Les **Generics** de **Java** ne servent qu'à rendre plus précis certains types de données (ex: `List<Personne>`).

Templates/Generics (classes paramétrées)

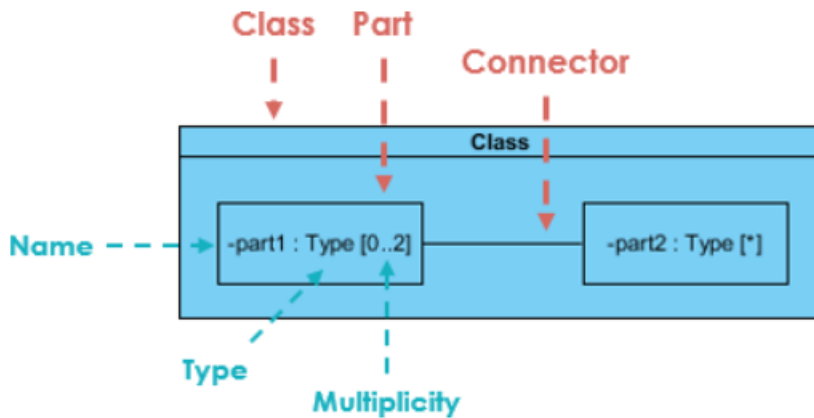
Un **template** est un **modèle générique de classe** pouvant être **paramétré par des paramètres formels (type de données, ...)**.



Liste_chainée<Mois,12> est une classe de réalisation issue du template.

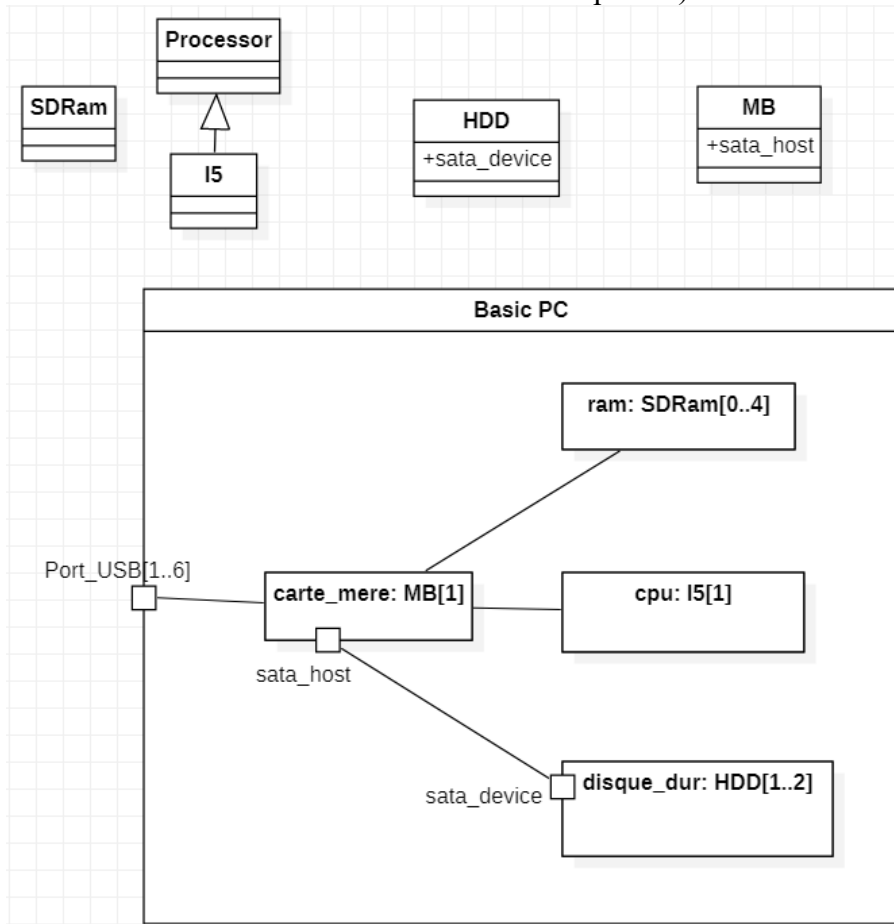
6. Diagramme de structure composite

Il s'agit essentiellement d'une extension au diagramme de classes permettant de **montrer des sous parties** (avec **multiplicités précises**) avec des **connecteurs** et d'éventuels ports d'accès .



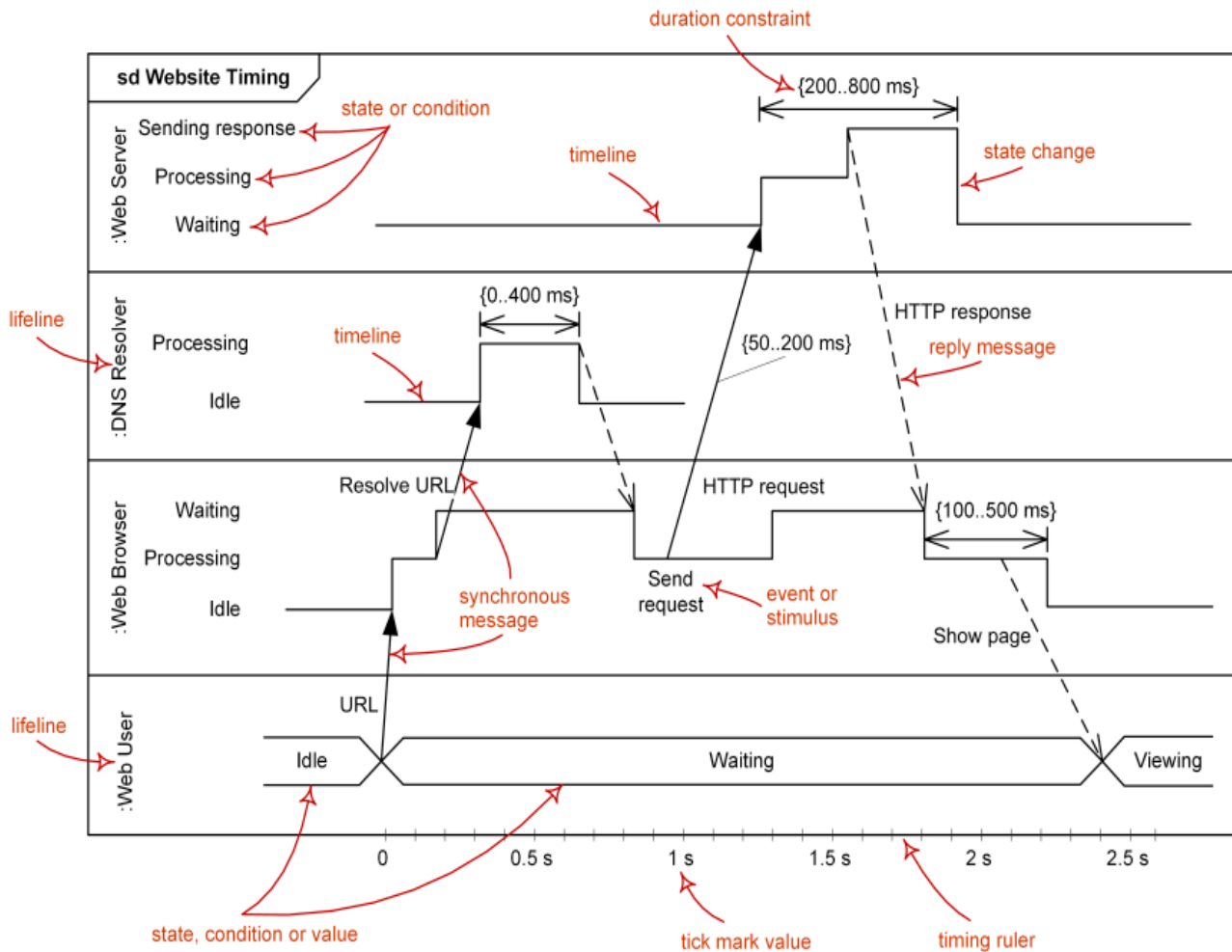
(illustration tirée de la documentation "visual-paradigm").

Avec Star-UML 3 , on peut obtenir le résultat suivant (avec "suppress attributes" coché et "suppress reception" décoché dans le menu contextuel de la classe composite) :



7. Aperçu sur diagramme de timing UML2

Exemple issu de "<https://www.uml-diagrams.org/timing-diagrams.html>" :



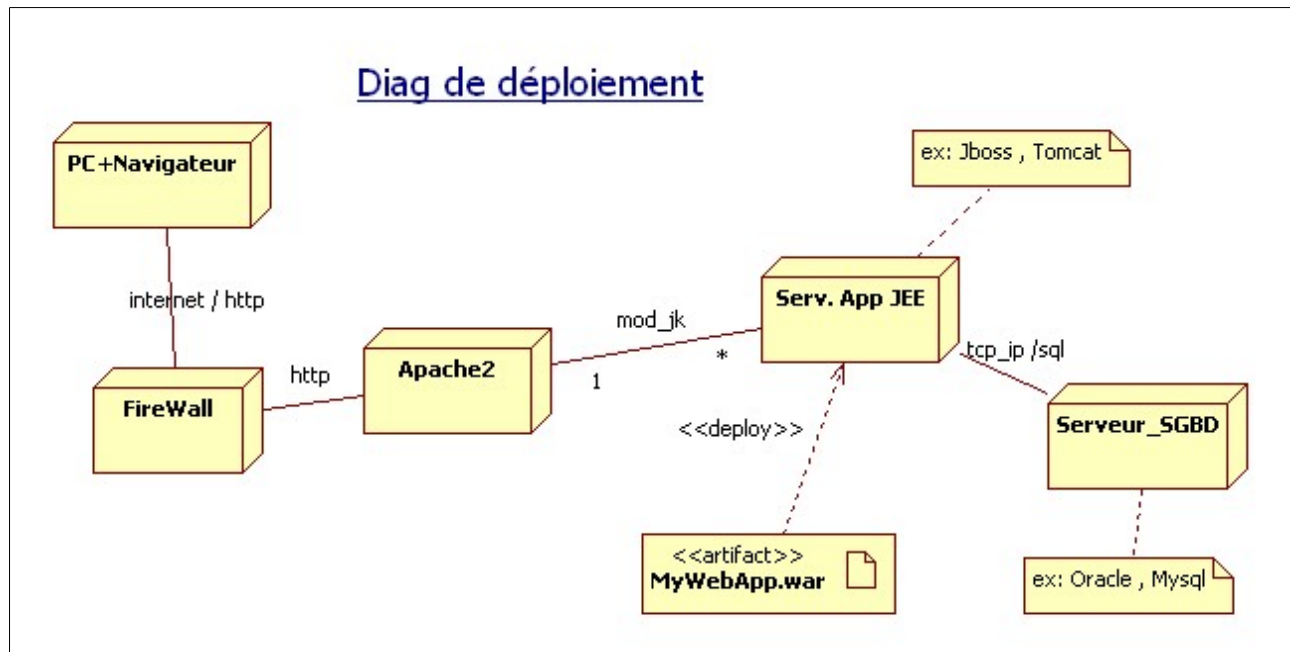
--> Un type de diagramme très pointu (géré par assez peu d'outils UML) permettant essentiellement d'indiquer des contraintes de temps et des synchronisations entre des éléments communiquant/interagissant entre eux .

VI - Implémentation , tests , itérations

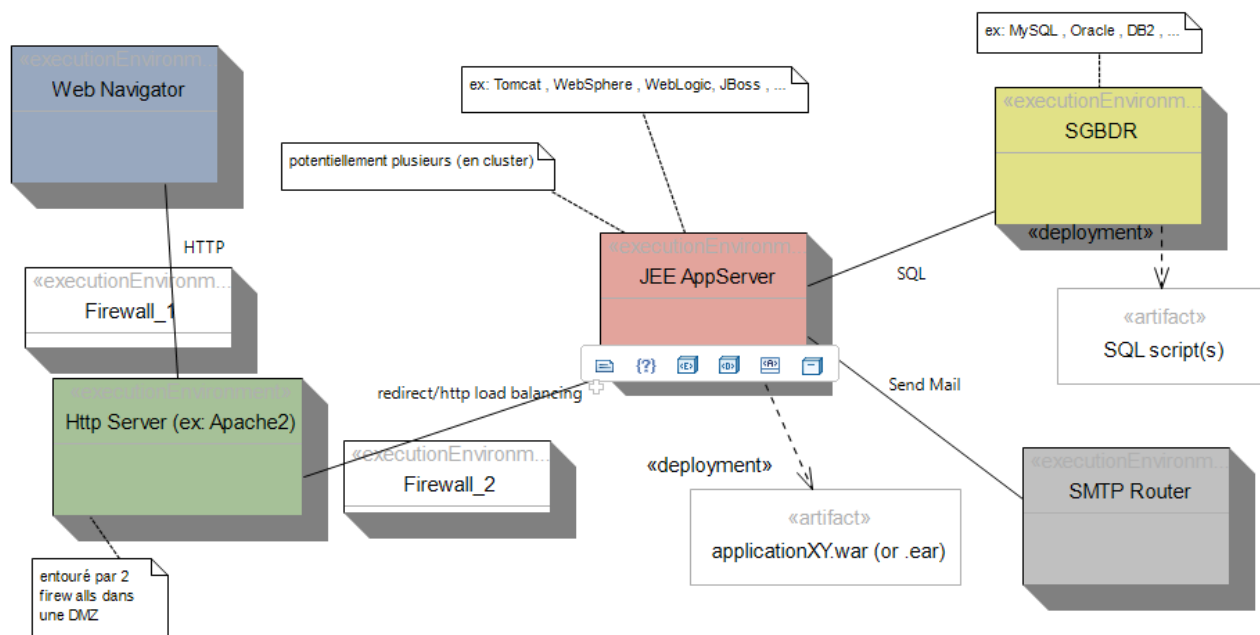
1.1. Environnements de développement, recette & production

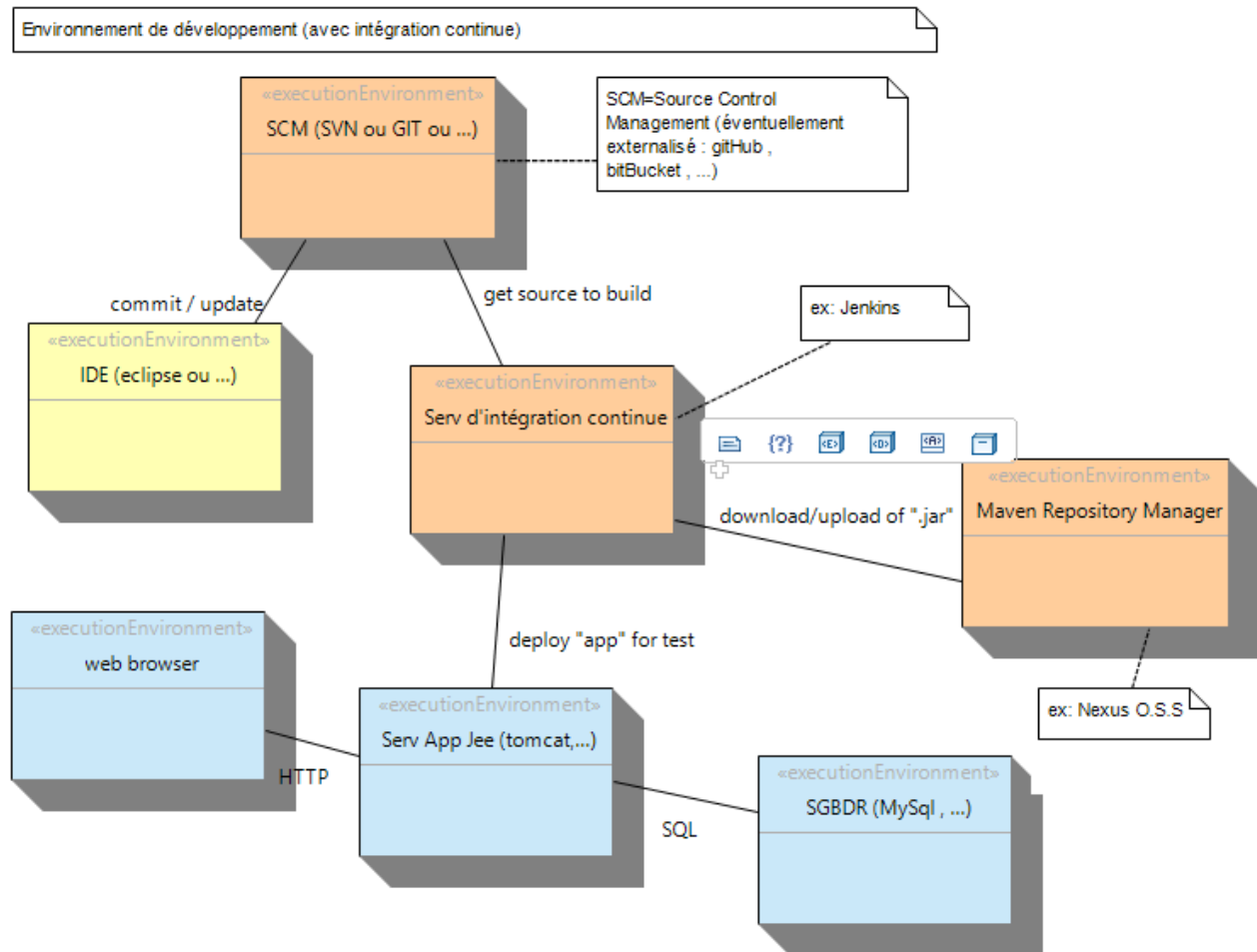
Sachant qu'il ne faut pas sous estimer la spécification de l'environnement cible (intégration / pré-production ,) , un diagramme de déploiement UML permet d'indiquer :

- les grandes lignes de la topologie d'une partie du S.I. (serveurs , liaisons réseaux,)
- les éléments à installer/déployer ou configurer (ex: base de données , applications , ...)



avec variantes :





1.2. Tests , retours , critiques , itérations

De façon à progresser , il est **indispensable** d'*effectuer un suivi* de ce qui sera développé en aval de la modélisation effectuée .

Modélisation initiale (premières idées) ==> implémentation & tests

==> bonnes et mauvaises critiques ==> nouvelle itération dans le cycle (modélisation ré-ajustée si nécessaire ,)

Bonne pratique !!!

ANNEXES

VII - Annexe – UML et mapping objet-relationnel

1. Cas de figures (down-top , top-down , ...)

1.1. Top-down (modèle logique --> base de données)

Dans le cas d'une application toute neuve (sans existant à reprendre) , la base de donnée n'existe pas et elle peut alors être vue comme un des "dérivés physiques" du modèle logique UML .

Certaines règles de conception permettent ainsi de produire un schéma relationnel (avec tables , clefs primaires et étrangères) à partir d'un modèle logique UML (stéréotype <<id>> , associations , rôles).

1.2. Down-top (base de données existante --> modèle logique)

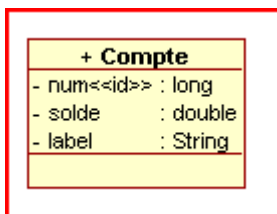
Dans un certain nombre de cas, la base de données existe (parce qu'elle est partagée avec un projet existant ou alors parce qu'il s'agit d'une migration d'une application existante vers Java) avant que la conception de l'application ne soit réalisée .

Le passage d'un modèle relationnel à un modèle logique UML s'effectue essentiellement en appliquant (dans le sens inverse) les règles de conception de l'approche "top-->down" .

2. Correspondances essentielles "objet-relationnel"

2.1. Entité (UML) / Table simple

| <i>Eléments UML</i> | <i>Eléments relationnels correspondants</i> |
|--|---|
| Classe (avec stéréotype <<entity>>) | Table relationnelle |
| Attribut(s) avec stéréotype <<id>> (identifiant) | Clef primaire (1 colonne ou +) |
| Attribut simple (de type String , int , double) | Colonne ordinaire de la table (VARCHAR , INTEGER , ...) |



```

----> Create Table Compte(
    num LONG PRIMARY KEY ,
    solde DOUBLE,
    LABEL VARCHAR(64));
  
```

Quelques conseils généraux:

- Eviter des clefs primaires formées par des IDs significatifs (ex: (nom,prenom)) en base , mais préférer des IDs de type compteurs qui s'incrémentent lorsque c'est possible.
- ...

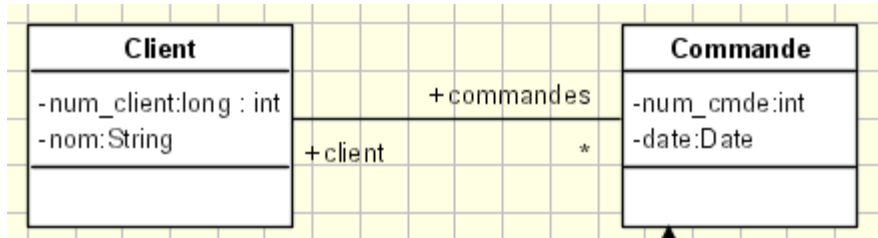
2.2. Association UML / Jointure entre tables

| Eléments UML | Eléments relationnels correspondants |
|----------------------------------|--|
| Association UML (1-n) avec rôles | Clef étrangère (proche du nom du rôle UML affecté à l'élément de multiplicité 1 ou 0..1) |
| ... | |

1-1

clef étrangère avec **unique=true**

1-n



T_Client

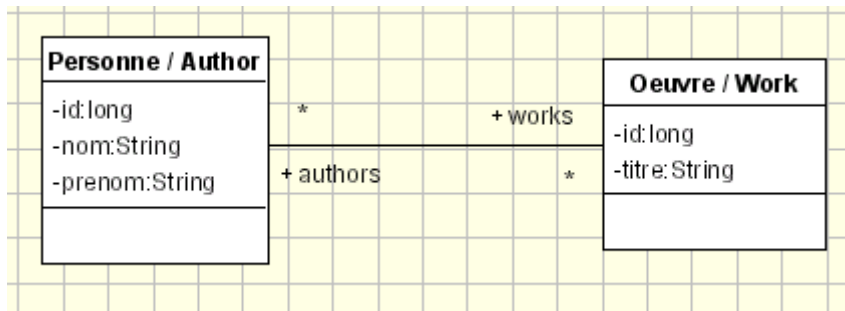
num_client (pk) , nom ,

many-to-one

T_Commande

num_cmde (pk) , client (fk), date,

n-n



T_Author

auth_id (pk) , nom , prenom ,

Author_Work

author_id (fk) , work_id(fk)

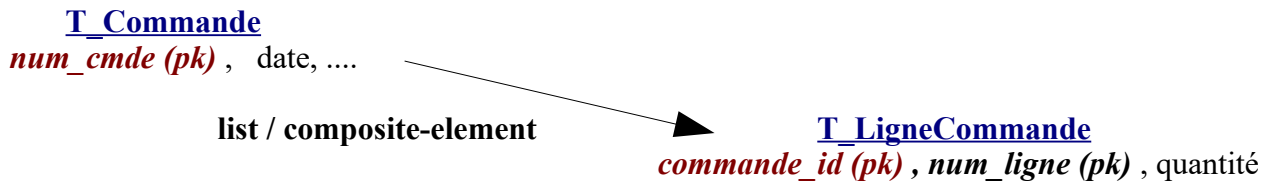
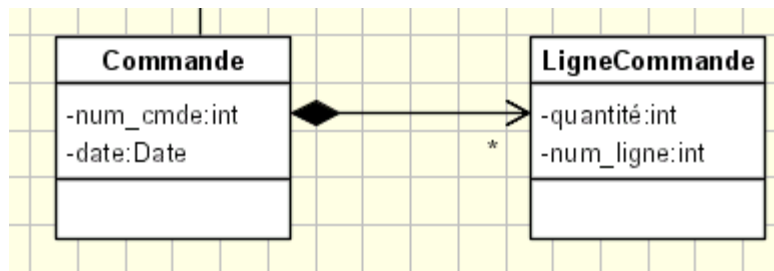
many-to-many

T_Work

w_id (pk) , titre , ...

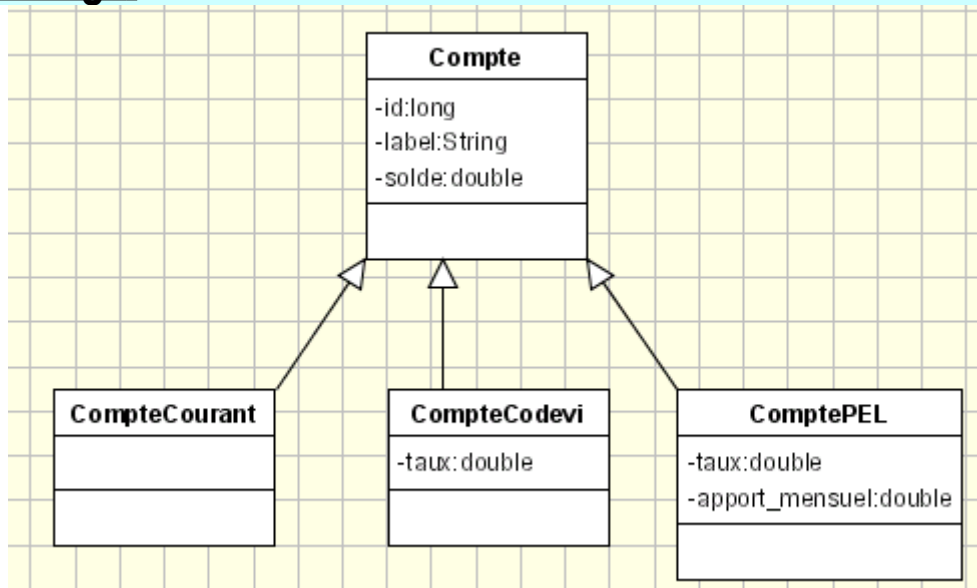
2.3. Composition UML / table "détails" avec clef primaire composée

| Eléments UML | Eléments relationnels correspondants |
|---|--|
| Composition UML (agrégation forte / losange noir) | Clef primaire composée du côté de la table "détails" . Ou bien éventuellement jointure simple avec opérations en cascade ("cascade delete", ...) |
| ... | |



3. Correspondances "objet-relationnel" avancées

3.1. Héritage



Solution/Stratégie 1 : "Une table par hiérarchie de classe" – propriété discriminante

Une seule grande table permet d'héberger les instances de toute une hiérarchie de classe.

Pour distinguer les instances des différentes sous classes , on utilise une propriété discriminante (à telle valeur correspond telle sous classe).

Cette stratégie (relativement simple) est assez pratique et adaptée dans le cas où il y a peu de différences structurelles entre les sous classes .

Solution/Stratégie 2 : "Une table par classe " (joined-subclass)

1 table avec points communs (liée à la classe de base et avec id/pk)

+

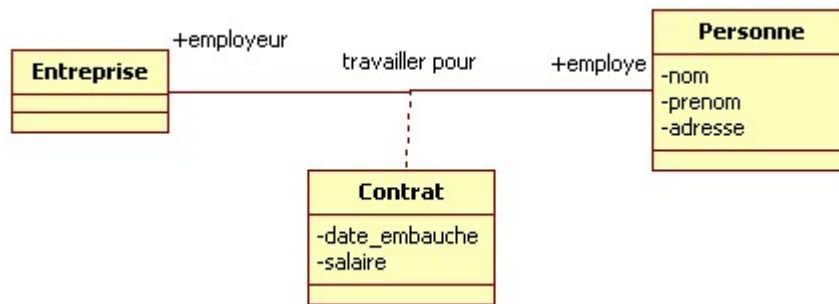
1 table pour chaque classe fille avec "colonnes suppléments" et clef étrangère fk référençant l'id de la table des points communs .

Solution/Stratégie 2 : "Une table par classe concrète"

Cette stratégie n'est pas idéale car elle casse un peu le lien entre sous classe et super classe .

==> problème sur polymorphisme .

3.2. Classe d'association & Table de jointure avec attributs



Contrat est une classe d'association (en UML) . Cette classe comporte des attributs (date d'embauche, salaire,...) qui détaillent l'association "travailler pour" (avec multiplicités non indiquées dans l'exemple ci dessus).

NB: Une modélisation UML à base de classe d'association est de niveau assez conceptuel (un peu comme le MCD de Merise).

==> D'un point de vue plus technique:

- un objet de type "Contrat" sera un objet intermédiaire (en mémoire) entre un objet "Entreprise" et un objet "Personne" .
- Une table "Contrat" (dans la base de données) sera très souvent une table de jointure avec:
 - * une clef primaire composée : (idEntreprise, idPersonne)
 - * des colonnes supplémentaires : date_embauche , salaire ,

3.3. Profil UML pour Données relationnelles

Une tentative de normalisation des aspects "données relationnelles" sous forme de "UML Profile" (extension normalisée pour UML) a été initiée par Rose/IBM au début des années 2000 mais n'a pas été réellement suivie.

On peut donc considérer, que les stéréotypes à utiliser pour paramétrer les aspects relationnels sont libres (<<id>> ou <<pk>> par exemple pour indiquer la clef primaire).

Idée à débattre:

Modèle UML stéréotypé (avec <<id>> ,)

==> génération de code MDA (ex: Accéléo)

==> DDL/SQL (Create table

==> reverse engineering (via Open ModelSphere ou ...)

====> MLD (Merise / relationnel)

4. Stéréotypes UML pour O.R.M.

Bien qu'il n'existe pas de standard véritablement suivi/utilisé sur le sujet, on peut utiliser des stéréotypes UML proches de ceux-ci :

| Stésréotypes UML | sémantique | Traduction JPA |
|------------------|--|----------------|
| <<entity>> | Entité métier persistante avec clef primaire | @Entity |
| <<id>> | Clef primaire (ou partie de clef primaire) | @Id |
| | | |
| | | |
| | | |
| | | |

Les associations @OneToOne , @OneToMany , @ManyToMany , peuvent être déduites à partir des multiplicités UML.

Pour paramétrer certains détails (coté secondaire des relations , générateur de clef primaire ,) ou pourra éventuellement utiliser quelques **valeurs étiquetées UML (tag Values) considérées quelquefois comme des propriétés des stéréséotypes**:

- **generator = auto** (pour auto_increment) ou ...
- **joinColumn** = nom_colonne_clef_étrangère
- **table** = nom_table_si_différent_classeUML
- **inverse=true** (ou secondary = true) pour le coté secondaire d'une relation bidirectionnelle (coté où il y a mappedBy="..." en JPA)
- **notNull=true**
- **length=32**
-

VIII - Annexes – Design Principles

1. Présentation des "design principles"

Grands principes de conception orientée objet

Une autre série de "design patterns" appelés <<*design principles*>> et élaborés par *Bertrand MEYER* et *Robert MARTIN* permettent (si besoin) d'approfondir certains points et de formaliser un peu plus quelques principes objets fondamentaux.

On parle alors en terme de *principes* ou de *règles* là où jusqu'ici *GRASP* s'exprimait essentiellement en terme de *bonnes pratiques*.

Essentiellement liée à la structure générale des modules , cette série de patterns est tout à fait adaptée à la conception préliminaire.

Préambule (sur le vocabulaire employé dans ce chapitre)

La série des « design principles » utilise intensément le terme « package » à interpréter ici au sens « module » ou « artifact » plutôt que « namespace » .

Autrement dit « package » est ici à comprendre comme « packaging » (ex : archive « .jar ») .

2. Gestion des évolutions et dépendances

Gestion des évolutions et dépendances (1)

OCP (Open-Close P.) / Principe d'ouverture-fermeture:

Un module doit être ouvert aux extensions
mais fermés aux modifications.

LSP (Liskov Substitution Principle) / Principe de substitution de Liskov:

Les méthodes qui utilisent des objets d'une classe doivent pouvoir utiliser des objets dérivés de cette classe sans même le savoir (*substitution parfaite par une sous classe*).

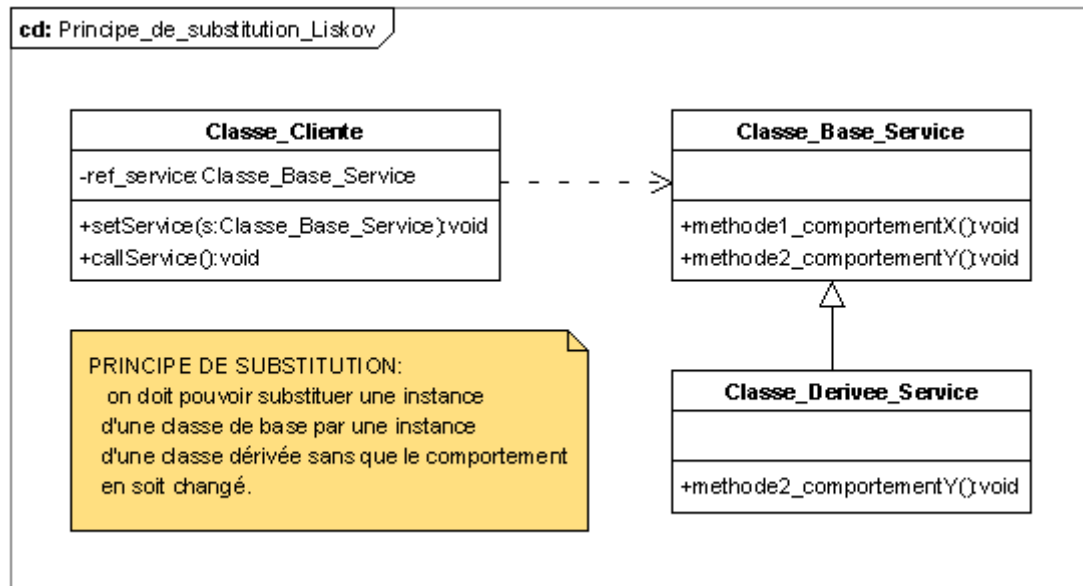
.../...

2.1. OCP et Principe de substitution (de Liskov)

Exemple (ouverture/fermeture):

- * Un bon schéma d'héritage (avec niveau abstrait , sous classes concrètes et polymorphisme approprié) est naturellement ouvert à des **extensions** potentielles qui prendront la forme de **nouvelles sous classes**.
- * A l'inverse un code basé sur de multiples *if (xxx instanceof Cxx) ...else if(...instanceof Cyy)* est assez fermé car il *doit malheureusement être modifié pour évoluer*.
- * D'un point de vue technique le LSP favorise l'OCP.
- * D'un point de vue sémantique, *un LSP trop artificiel peut conduire un des implémentations non applicables (avec éventuelles exceptions à gérer)* , ce qui n'est pas mieux qu'un unique test "instanceof" (idéalement lié à toute une sous branche de l'arbre d'héritage).

Un objet d'une classe CX doit normalement pouvoir être substitué par n'importe quel objet d'une sous classe CY sans que le comportement en soit changé.



Autrement dit : La sémantique (ou le comportement abstrait) ne doit pas être changé entre l'implémentation d'une opération dans une classe et dans une sous classe .

2.2. Principe d'inversion des dépendances

Gestion des évolutions et dépendances (2)

DIP (Dependency Inversion P.) / Principe d'inversion des dépendances:

- A. Les modules de haut niveau ne doivent pas dépendre de modules de bas niveau. Tous deux doivent dépendre d'abstractions
- B. Les abstractions ne doivent pas dépendre de détails. Les détails doivent dépendre d'abstractions.

ISP (Interface Segregation P.) / Principe de séparation des interfaces:

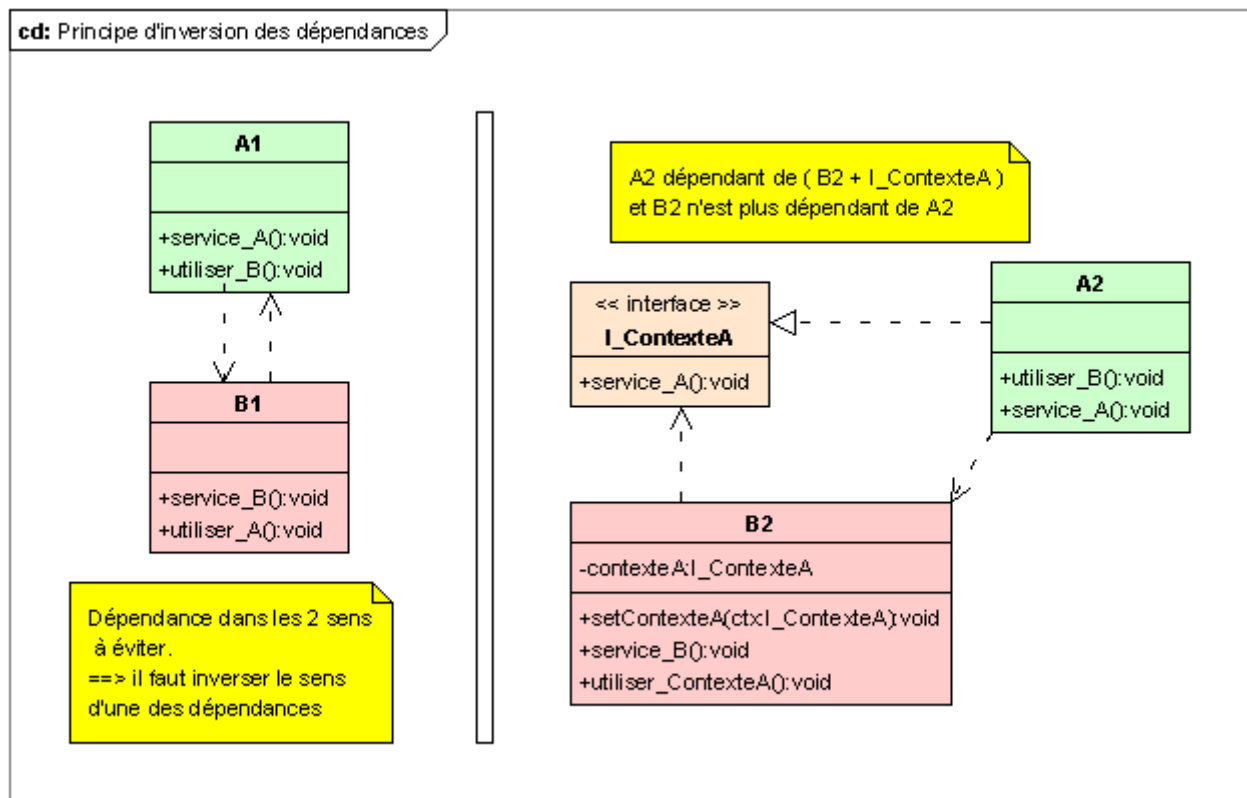
Les clients ne doivent pas être forcés de dépendre d'interfaces qu'ils n'utilisent pas.

Autrement dit:

DIP_A : Un module de haut niveau doit toujours voir un service de plus bas niveau de façon abstraite (interface).

Pour qu'un module de bas niveau soit réutilisable par plusieurs modules de haut niveau, il doit interagir avec ceux-ci via des "callback" déclarées sur des contextes abstraits (*ex: méthodes événementielles et "listener"*)

ISP : Ne pas hésiter à faire en sorte qu'une classe implémente plusieurs interfaces complémentaires. Les futurs clients ne seront alors dépendants que d'un seul point d'entrée abstrait.



Il vaut mieux éviter des dépendances dans les 2 sens entre deux classes ou packages (ou modules) :

A-->B et B-->A

On introduit alors une interface (I_ContexteA) de type "interface sortante coté B" devant être ultérieurement implémentée par A et on met en place une méthode d'enregistrement de contexte I_ContexteA coté B: ".setContexteA(IA ...) ou constructeur ...".

Les nouvelles dépendances sont alors les suivantes:

A-->I_ContexteA (en l'implémentant)

B-->I_ContexteA (en l'utilisant)

A-->B

NB:

- L'interface I_ContexteA est censée être packagée coté B (dans le même package)
- Ce principe est très utilisé pour le traitement des événements.

Exemple concret :

B2 = class Button (ou JButton) devant être réutilisable (dans package awt/swing)

I_ContexteA = interface ActionListener

A = Fenetre quelqonce utilisant un bouton

coté "bouton" , l'enregistrement du contexte se fait via addActionListener(...)

3. Organisation d'une application en modules

Organisation de l'application en modules

REP (Reuse/Release Equivalence P.) / Principe d'équivalence

Réutilisation/Livraison:

La granularité en termes de réutilisation est le package.
Seuls des packages livrés sont susceptibles d'être réutilisés.

CRP (Common Reuse P.) / Principe de réutilisation commune:

Réutiliser une classe d'un package, c'est réutiliser le package entier.

CCP (Common Closure P.) / Principe de fermeture commune:

Les classes impactées par les mêmes changements doivent être placées dans un même package.

Autrement dit:

Ne pas hésiter à modifier un élément interne (et caché/privé) d'un package car ceci n'a pas d'impact négatif sur la réutilisation du package.

Décomposer s'il le faut un gros package en un ensemble de petits packages pour affiner les dépendances (*pour ne redéployer que ce qui a changé et pour favoriser de futures évolutions*).

4. Gestion de la stabilité de l'application

Gestion de la stabilité de l'application

ADP (Acyclic-Dependencies P.) / Principe des dépendances

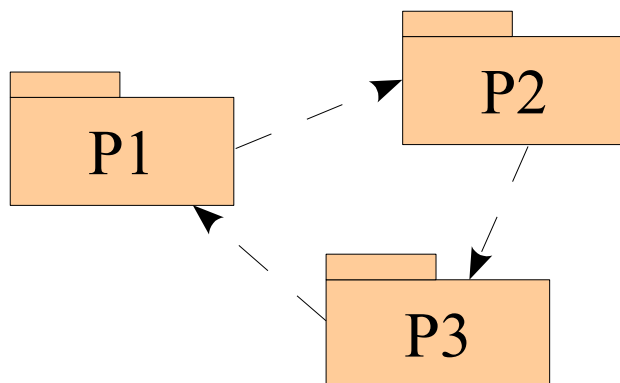
acycliques: Les dépendances entre packages doivent former un graphe acyclique (sans dépendance(s) circulaire(s)).

SDP (Stable Dependencies P.) / Principe de relation

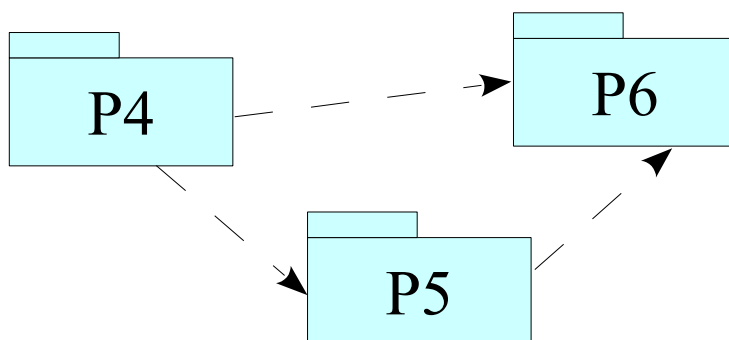
dépendance/stabilité: Un package doit dépendre uniquement de packages plus stables que lui.

SAP (Stable Abstractions P.) / Principe de stabilité des

abstractions: Les packages les plus stables doivent être les plus abstraits. Les packages instables doivent être concrets. Le degré d'abstraction d'un package doit correspondre à son degré de stabilité.



pas bien !



bien !

exemple: implémentation "*myfaces / Apache*"
du framework WEB "*JSF*"

myfaces-api.jar

package "javax.faces"
(standard stable)

myfaces-impl.jar

package "org.apache.myfaces"
*(implémentation qui
évolue à chaque version)*



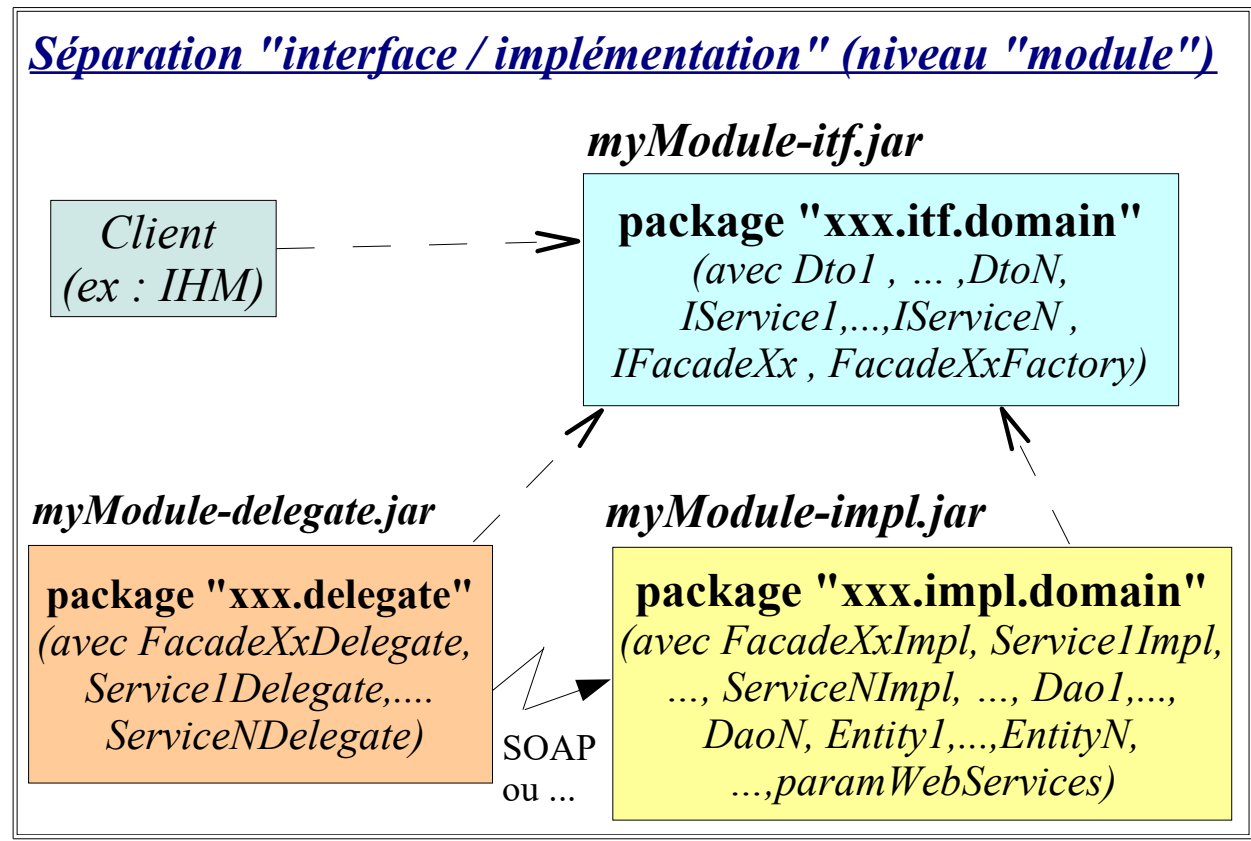
Application courante :

client_module -----> **moduleXY_itf**

(implémenté via)

mod_XY_local_impl ou bien **mod_XY_delegate_impl** ----> services_distant

(selon ".jar" présent dans le classpath) .



NB : Il existe des "métriques de packages" qui permettent de mesurer le respect des principales règles de "bonne conception orientée objet" .

IX - Annexe - Patterns "GOF" et "JEE"

1. Liste des principaux "design patterns" du GOF

| Design Pattern | Description |
|---|---|
| Abstract factory (fabrique abstraite) | Instanciation indirecte de familles de produits. (ex: Look & Feel "Windows" ou "X11/Motif" Java / SWING). En changeant de fabrique on génère des objets différents (look1, look2,) qui ont néanmoins les mêmes fonctionnalités (même interfaces). |
| Adapter (adaptateur) | Intermédiaire permettant de convertir l'interface (figée) d'une classe existante avec celle attendue par un client. |
| Bridge (pont) | Correspondance (pont) entre 2 hiérarchies de classes (ex: XXX - XXXImpl de AWT). Séparation de l'abstraction et de la représentation/implémentation. |
| Builder (monteur) | Fabriquer des parties via des objets monteurs dirigés par un objet directeur. Séparer la construction de la représentation. |
| Chain of responsibility (Chaîne de responsabilité) | Requête avec: 1. 1 émetteur. 2. des récepteurs chaînés entre eux (hop, hop ,hop jusqu'à traitement effectif [+ éventuelle valeur ajoutée sur les intermédiaires aux responsabilités bien définies]). ==> exemple ACL = Access Control List. |
| Command (Commande) | Encapsulation d'une requête dans une méthode (ex: execute()) d'un objet commande. ==> Liste de commandes ==> permet undo/redo et le déclenchement d'une même commande depuis plusieurs parties de l'IHM (menu, toolbar, bouton poussoir). |
| Composite | Composition récursive à niveau de profondeur quelconque (héritage + composition combinés) |
| Decorator (Décorateur / Enveloppe transparente) | Enveloppe transparente rajoutant de nouvelles fonctionnalités. |
| Facade | Interface unifiée pour la totalité d'un sous système / accueil plein de variantes (ex : façade agnostique , ...) |
| Factory method (méthode de fabrication) | Création indirecte d'instance déléguée au niveau d'une méthode de type "create()" [beaucoup de variantes] |
| Flyweight (poids mouche) | Comment gérer plein de petits objets ? ==> petit objet partagé (avec une partie interne intrinsèque) et avec des méthodes comportant une référence sur un contexte (partie externe à la charge du client). |
| Interpreter (interpréteur) | Grammaire (phrases / expressions à interpréter , exemple: $2x+y+5*z/3$). ==> construction d'un arbre syntaxique dont les nœuds sont des objets. Fonction interpréter() récursive et polymorphe. |
| Iterator (itérateur) | Traverser (balayer/parcourir) une collection (liste/tableau) sans avoir à connaître la structure interne et de façon à accéder à chacun des éléments. |
| Mediator (médiateur) | Intermédiaire commun à un paquet d'objet (coordonnant les interactions)==> pour réduire le couplage. |
| Memento | Petit objet secondaire (avec interfaces large et fine) permettant de mémoriser l'état d'un objet principal de façon à restaurer les valeurs de celui-ci plus tard (Respect de l'encapsulation, permet un undo). |
| Observer (observateur) | Définit une dépendance de un à plusieurs 1 Mise à jour / des Notifications (callback) |

| | |
|--|---|
| | 1 Diffusion / des Souscripteurs |
| Prototype | Créer de nouveaux objets à partir d'un exemplaire déjà instancié (clonage). (ex: Palette d'objets graphiques à cloner). |
| Proxy (proximité) | Fournir un substitut pour accéder indirectement à un objet (souvent distant). |
| Singleton | Une seule instance pour une certaine classe (Méthode statique pour créer (ou obtenir) cette instance. |
| State (état) | Différents sous-objets ayant une interface commune codent différents comportements d'un objet englobant ayant plusieurs états (comme s'il changeait de classe). |
| Strategy (stratégie) | Classe abstraite d'algorithmes interchangeables pour un certain contexte. [externaliser une responsabilité avec variantes] |
| Template method (Patron de méthode) | Algorithme abstrait basé sur un même squelette (méthode d'une sur-classe) dont différentes sous tâches (parties) sont codées comme des méthodes polymorphes dans diverses sous classes. |
| Visitor (visiteur) | Sur un objet que l'on peut parcourir (ex : hiérarchie de noeuds d'un arbre ou liste) on va déclencher (via des visiteurs actifs) des opérations génériques durant une traversée. ==> objets visitables et visiteurs doivent être pensés et modélisés pour être compatibles. |
| | |
| ... | |

2. Design Patterns fondamentaux du GOF

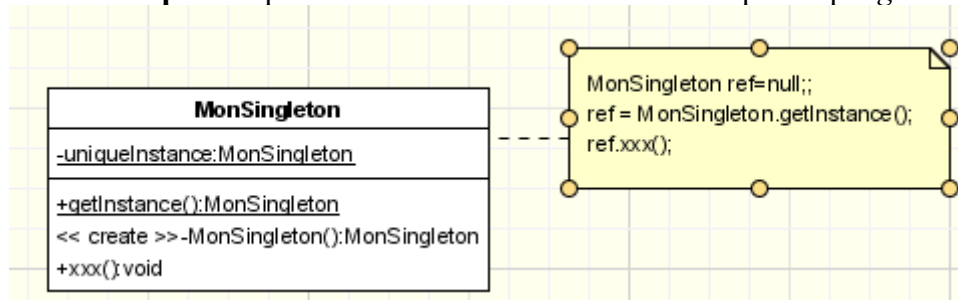
2.1. Singleton

Objectif : Une et une seule instance pour une classe donnée.

Solution classique:

- variable de classe (static) **uniqueInstance** = null
- méthode de classe (static) **getInstance()**

```
{ if (uniqueInstance == null) {
    uniqueInstance = new Xxx();
}
return uniqueInstance; // instance nouvellement ou anciennement créée .
}
```
- **constructeur privé** pour interdire des "new" directs sans passer par *getInstance()*



NB : il faudra penser à ajouter "**synchronized**" (ou autre) sur *getInstance()* dans un contexte "multi-thread" .

Avantages du singleton:

Etre sûr qu'une seule instance sera utilisée à un niveau donné permet:

- **d'optimiser la mémoire**
- **de gérer un contexte (avec données partagées) à un endroit central bien précis (initialisation , lecture/écriture , ...)**

D'autre part, l'appel d'une méthode statique est très pratique pour récupérer l'instance unique depuis un endroit quelconque du programme.

Le Singleton est assez souvent utilisé sur les fabriques et les façades (une seule instance suffit souvent)

Eventuels inconvénients:

Effet "Kitchen Sink" (siphon) ==> Le singleton ramène à lui tous les appels "static" , ce qui peut poser quelques problèmes si les choses doivent évoluer (changement ? changement partiel ? , ...).

--> On peut dans certains cas , mettre en oeuvre des "pseudo singleton" liés à un contexte bien précis (ex : contexte Spring ou module angular ou ...)

Exemple de code (Singleton) :

```
public class ProduitDaoFactory {
    private static ProduitDaoFactory uniqueInstance = null;

    public synchronized static ProduitDaoFactory getInstance() {
        if(uniqueInstance==null){
            uniqueInstance=new ProduitDaoFactory();
        }
        return uniqueInstance;
    }

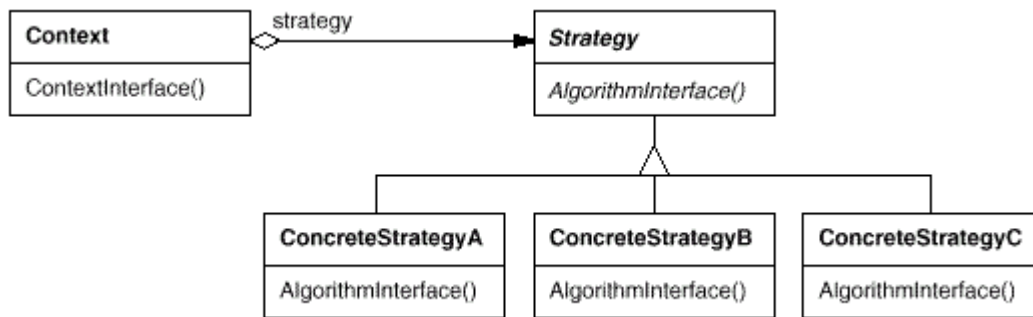
    private ProduitDaoFactory() { super(); }

    private String commonData; //shared inside singleton (with get/set)
    public String getCommonData() { return commonData; }
    public void setCommonData(String commonData) { this.commonData = commonData; }
    //...
}
```

```
private void subTestSingleton(){
    ProduitDaoFactory produitDaoFactory = ProduitDaoFactory.getInstance();
    String data = produitDaoFactory.getCommonData();
    Assert.assertTrue(data.equals("my shared data inside singleton"));
}

@Test
public void testSingleton(){
    //ProduitDaoFactory prodDaoFactory = new ProduitDaoFactory();-
    //impossible si constructeur privé
    ProduitDaoFactory prodDaoFactory = ProduitDaoFactory.getInstance();
    prodDaoFactory.setCommonData("my shared data inside singleton");
    subTestSingleton();
}
```


2.2. Stratégie



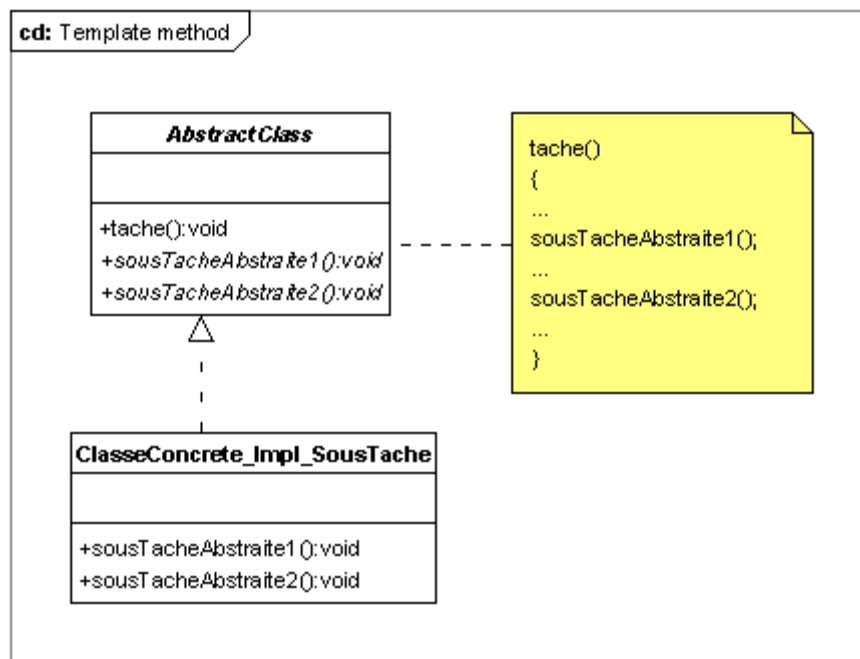
NB: Le design pattern "*stratégie*" met non seulement l'accent sur le *polymorphisme* mais également sur le fait d'*externaliser une certaine responsabilité annexe vis à vis du contexte de départ*. C'est généralement une très bonne idée favorisant nettement la **modularité** de l'ensemble.

NB: Le design pattern "*D.A.O.*" (Data Access Object) peut être vu comme un cas particulier de stratégie (une stratégie d'accès aux données)

"DataSource" est une stratégie pour établir une connexion à une base de données .

2.3. Patron/modèle de méthode (Template method)

Factoriser ce qui est commun , différencier le spécifique .



Exemple:

// code commun de la tache au niveau de la classe abstraite:

```

void dessinerAvecCouleur(String couleur) {
  choisirCouleur(couleur); // code commun (factorisé au niveau de la classe abstraite)
  dessiner(); // sous tache abstraite avec code différent pour ligne , rectangle , ...
}
  
```

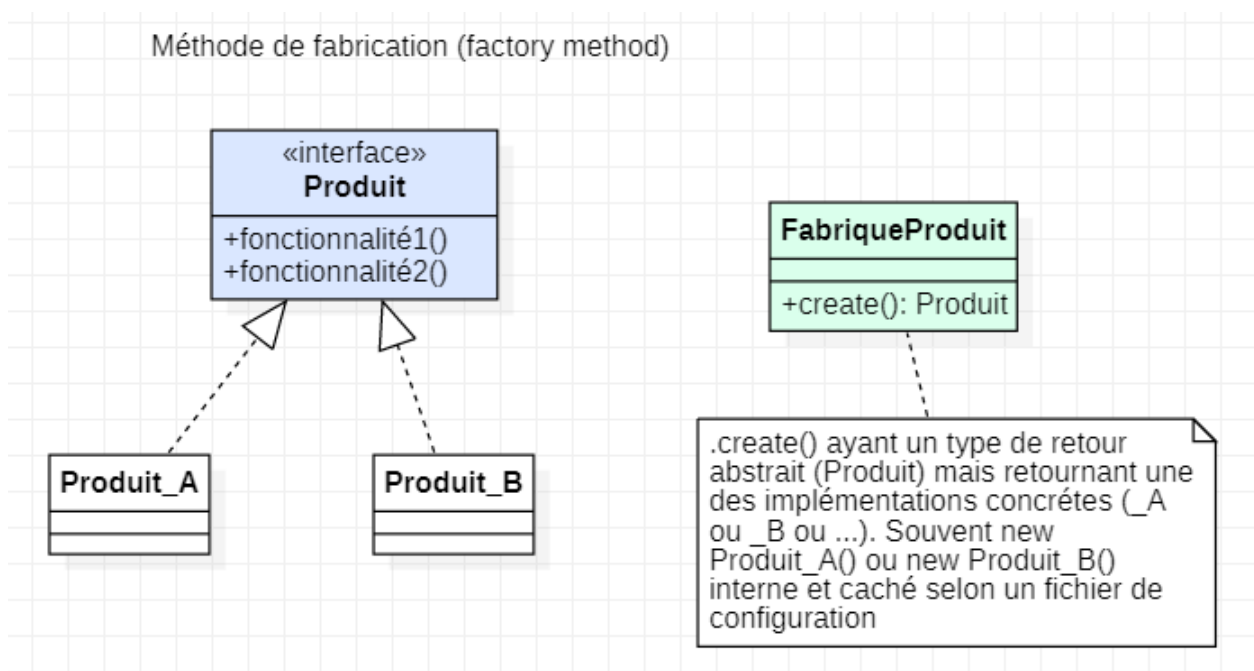
2.4. Méthode de fabrication (factory method)

Problème courant: comment masquer le type exact d'un objet à créer ainsi que les détails de son initialisation ?

Solution courante: Utiliser une méthode de fabrication sur un objet intermédiaire (souvent appelé "fabrique"). On évite ainsi une instantiation directe du genre `xxx = new CXxx("v1","v2");`

Ce code est alors caché au sein d'une méthode `".create()"` éventuellement "static".

Le code interne de `".create()"` peut alors utiliser des mécanismes quelconques (ex: fichier de configuration) pour créer et initialiser un composant `CXxxV1` ou `CXxxV2` et le retourner ensuite de façon abstraite (type de retour de `.create()` = interface `IXxx`)



Variantes: le code interne de la méthode de fabriquer (ex: `create()`) peut :

- instancier une nouvelle instance via `new`
- retourner un objet pré-construit et rangé dans un pool
- déclencher une instantiation générique via `Class.forName("package.NomClasse").newInstance()`
-

Exemple :

produitDao.properties

```
#produitDao=tp.dao.ProduitDaoSimu
produitDao=tp.dao.ProduitDaoJdbc
#produitDao=tp.dao.ProduitDaoJpa
```

```
public class ProduitDaoFactory {
//...
public ProduitDao createProduitDao() {
    ProduitDao dao = null;
    String daoImplClassName = MyPropertiesUtil.propertyValueFromEntryOfPropertyFile(
        "produitDao.properties", "produitDao");
    try {
        logger.info("daoImplClassName="+daoImplClassName);
        /*
        if(daoImplClassName.equals("tp.dao.ProduitDaoSimu") )
            dao = new tp.dao.ProduitDaoSimu();
        else if(daoImplClassName.equals("tp.dao.ProduitDaoJdbc") )
            dao = new tp.dao.ProduitDaoJdbc();
        else if(daoImplClassName.equals("tp.dao.ProduitDaoJpa") )
            dao = new tp.dao.ProduitDaoJpa();
        */
        dao = (ProduitDao) Class.forName(daoImplClassName).newInstance();
    } catch (Exception e) {
        e.printStackTrace();
    }
    return dao;
}
}
```

```
package tp.util;
```

```

import java.io.InputStream;
import java.util.Properties;

import org.slf4j.LoggerFactory;

public class MyPropertiesUtil {
    private static org.slf4j.Logger logger = LoggerFactory.getLogger(MyPropertiesUtil.class);

    public static Properties propertiesFromCPRelativePathFile(String relativePathFile){
        Properties props = new Properties();
        try {
            InputStream inStream = Thread.currentThread().getContextClassLoader()
                .getResourceAsStream(relativePathFile);
            props.load(inStream);
        } catch (Exception e) {
            logger.error("cannot load properties file : " + relativePathFile , e.getMessage() );
        }
        return props;
    }

    public static String propertyValueFromEntryOfPropertyFile(
        String relativePathPropertyFile, String propertyName)
    {
        String propValue=null;
        Properties props = propertiesFromCPRelativePathFile(relativePathPropertyFile);
        propValue=props.getProperty(propertyName);
        return propValue;
    }
}

```

NB : Le design pattern "injection de dépendances" (alias "ioc") présenté dans le chapitre "design patterns JEE et n-tiers" est une évolution du design pattern "factory" (sorte de "méga fabrique" qui fabrique et assemble plein de composants) .

2.5. Facade

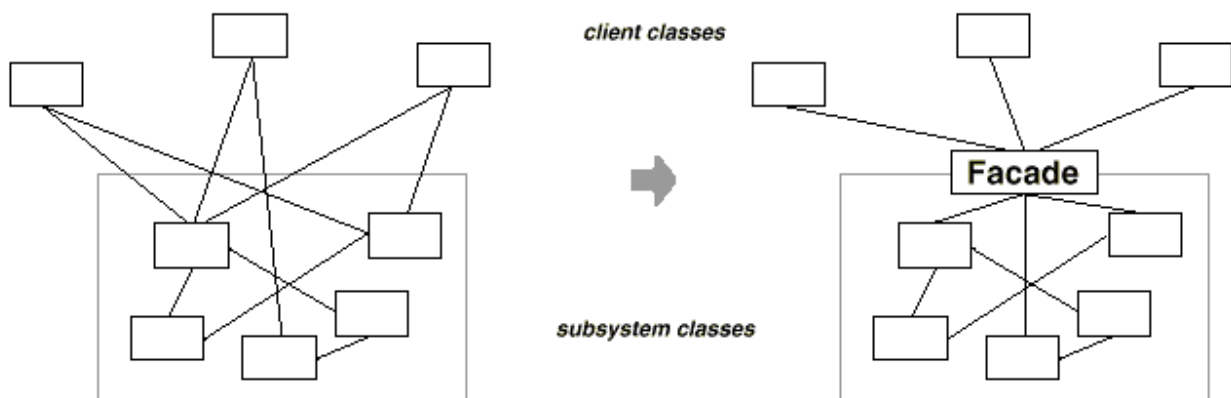
Le design pattern "façade" comporte plein de variantes.

Au sens le plus général, il s'agit de mettre en place une façade (simple à utiliser) de façon à indirectement bénéficier des services rendus par différents éléments d'un module (quelquefois complexe).

- Une façade de type "accueil/redirection" est une façade d'entrée unique pour un module complet ==> on est alors dépendant que d'un seul élément central (qui souvent redirige).
- Une façade "technologiquement agnostique" est un objet (que l'on instancie via un simple new) et qui cache toutes les technologies utilisées en arrière plan (ex: Spring, ...) .

Problème à résoudre (GOF):

Un module sans façade comporte de multiple points d'entrée que l'on est obligé de connaître pour initialiser les appels. D'autre part, ce n'est pas facile de s'y retrouver en cas d'évolution (quelles répercussions suite à tel changement ?).



Solution générale:

Introduire un nouvel élément intermédiaire qui jouera le rôle de "façade / accueil / point d'entrée unique" pour ce module.

Variantes de la solution:

| <i>Variantes</i> | <i>Caractéristiques</i> | <i>Astuces (?)</i> |
|---|---|---|
| Nouvelle classe reprenant toutes les méthodes de toutes les classes internes (délégation en interne) | cette classe peut devenir énorme si le module est grand. | solution monolithique pas très évoluée |
| Simple intermédiaire de type "accueil" redirigeant vers les classes internes du module | ceci permet de ramener à 1 le nombre de point d'entrée à connaître mais le code des appels est, après redirection de l'accueil, de nouveau directement dépendant des éléments internes du module. | solution moyenne : accueil sans abstraction. |
| Intermédiaire de type "accueil/façade/fabrique" redirigeant vers des éléments concrets internes qui sont retournés comme des éléments abstraits . | Un seul point d'entrée à connaître + vision externe abstraite (pouvant être indépendante de la structure interne du module) ==> évolution interne facilitée. | solution astucieuse (très bien) mais un peu longue à programmer (interfaces , façade ,) |

NB : On appelle "**façade agnostique**", une façade qui masque volontairement la technologie utilisée en interne dans le module (exemple : "spring" ou "cdi") .

NB : "agnostique" (signifiant littéralement "non croyant") est ici à prendre au sens "je ne crois pas à la prédominance de spring ou cdi pour encore les 20 ans à venir" et je préfère basé mon code sur les fondamentaux du langage de programmation ("static" , "singleton" , "constructeur" , ...)

Exemple :

```
package tp.service;
public interface MyFacade {
    //GestionProduits , GestionConversion et GestionTva sont ici 3 interfaces de services
    public GestionProduits getGestionProduits();
    public GestionConversion getGestionConversion();
    public GestionTva getGestionTva();

    public void cleanUpResources(); //si nécessaire (libérer ressources internes)
}
```

```

public class MyFacadeImpl implements MyFacade {

    private GestionConversion gestionConversion=null;
    private GestionTva gestionTva=null;
    private GestionProduits gestionProduits=null;
    //... + singleton + ...

    private MyFacadeImpl(){
        //En version "Agnostique", une façade cache entièrement la technologie "Spring" ,
        // "CDI" ou "IOC maison" qui prend en charge les composants derrière la façade

        MyXmlBeanFactory myXmlIocFactory = MyXmlBeanFactory.getInstance();
        myXmlIocFactory.initIocConfigFromXmlFile("myIocConfig.xml");
        this.gestionConversion = (GestionConversion)
            myXmlIocFactory.getBean("serviceGestionConversion");
        this.gestionTva = (GestionTva) myXmlIocFactory.getBean("serviceGestionTva");
        this.gestionProduits = (GestionProduits) myXmlIocFactory.getBean("serviceGestionProduits");
    }

    public GestionProduits getGestionProduits() {return gestionProduits;}
    public GestionConversion getGestionConversion() {return gestionConversion;}
    public GestionTva getGestionTva() {return gestionTva;}

    public void cleanUpResources() {
        gestionProduits.cleanUpResources(); //à adapter avec Spring ou autre sur vrai projet
    }
}

```

Exemple d'utilisation de la façade :

```

MyFacade myFacade = MyFacadeImpl.getInstance();
double sommeFrancs = myFacade.getGestionConversion().euroToFrancs(15);
System.out.println("15 euros : " + sommeFrancs + " francs");
double sommeTva = myFacade.getGestionTva().getTva(20.0 , 200);
System.out.println("TVA (20.0) pour 200 Euros ==>" + sommeTva);
myFacade.cleanUpResources();

```

Relativité de l'importance des façades :

La mise en oeuvre du design pattern "façade" est généralement intéressante sur un gros projet (application de grande taille) .

Associer façade et "business delegate" (voir chapitre "design patterns pour jee et n-tiers") peut éventuellement être intéressant pour bien structurer le code d'une application cliente (ou middleware) déléguant des appels vers de nombreux serveurs .

L'aspect "agnostique" n'est généralement intéressant que s'il est appliqué à fond.

Mieux vaut ne pas appliquer de "semi indépendance technologique" si 80 % de l'application est à fond basé sur un framework bien spécifique du type "CDI" ou "Spring" .

2.6. Décorateur (enveloppe transparente)

Provenance du nom "décorateur":

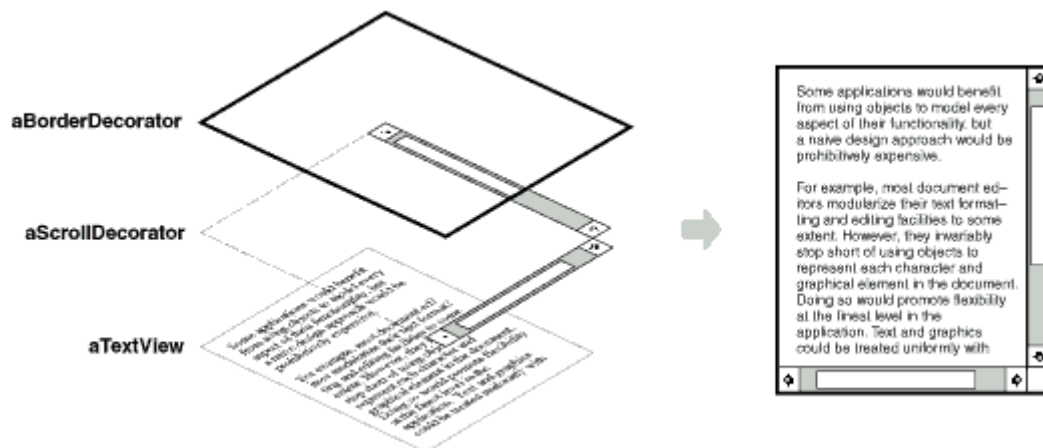
Soit "Composant Visuel" une classe correspondant à un composant graphique (sorte de fenêtre) de base.

Cette classe comporte les méthodes classiques "Afficher() , ..."

On souhaite maintenant **manipuler de façon uniforme** différents types de composants graphiques avec de nouvelles fonctionnalités:

- Vue (éditeur) de texte (toute simple).
- Vue de texte avec ascenseurs et gestion automatique du scrolling.
- Vue de texte avec bordures de redimensionnement automatique et titre.
- Vue de texte combinant bordures et ascenseurs.

NB: les éventuelles décorations (ascenseurs, bordures) sont toujours gérées automatiquement.



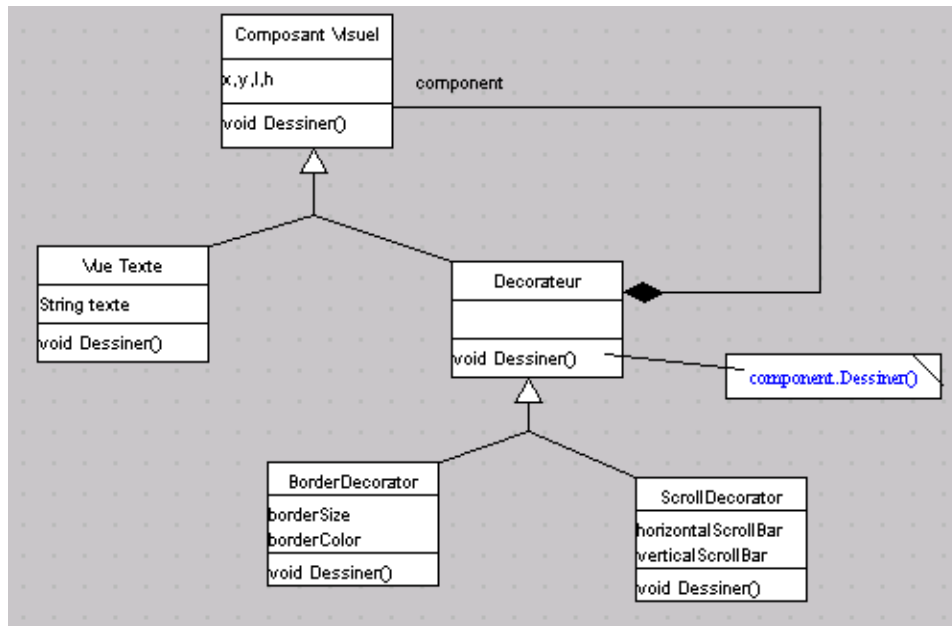
Solution intuitive mais moyenne:

N'utiliser que l'héritage et créer plein de classes telles que "Composant graphique avec bordures" , "Composant graphique avec Ascenseurs" , ...

Problème: cette solution est un peu trop statique (figée). Si l'on souhaite une classe combinant bordures et ascenseurs , il faut effectuer un nouvel héritage (éventuellement multiple).

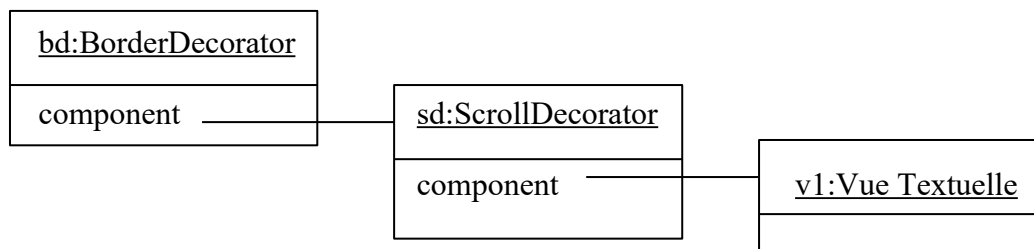
Solution plus flexible proposée (Design Patter "Decorator"):

.../...



Il suffit d'effectuer une série d'imbrications pour obtenir une vue avec bordures et ascenseurs.

D'autre part, la classe Décorateur hérite de "Composant Visuel" et est donc vue comme un composant visuel ordinaire (bien qu'il ait une composition en interne).



Sémantique "enveloppe transparente":

Le design pattern ci-dessus correspond à la notion de "**enveloppe transparente**".

Le code d'utilisation ne voit l'objet manipulé que comme un composant de base ordinaire.

L'**objet réellement manipulé** peut en fait être une **enveloppe intermédiaire** qui:

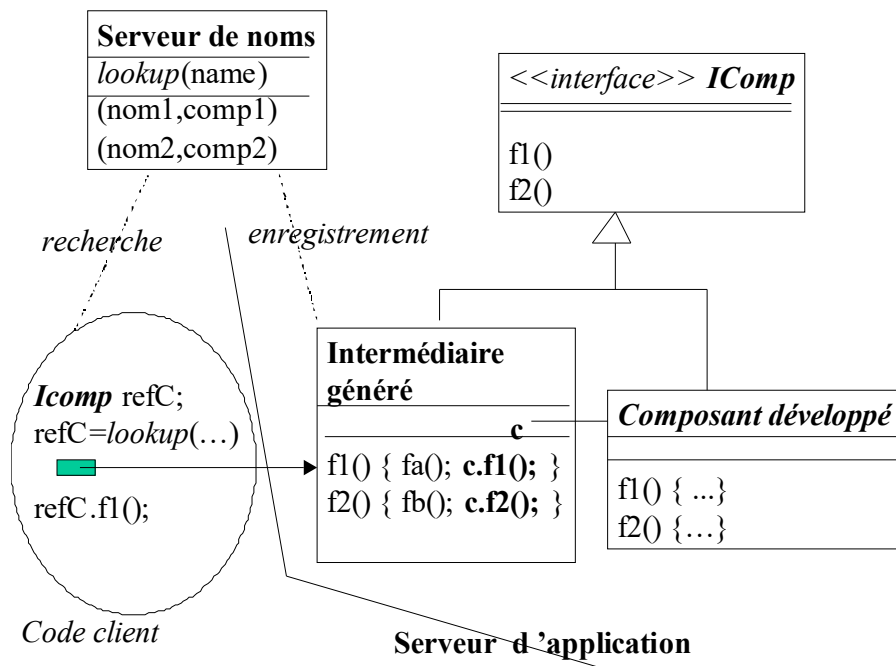
- **ajoute de nouvelles fonctionnalités**
- **délègue les traitements de base au niveau de l'objet imbriqué.**

Quelques exemples concrets du Design pattern "Decorateur":

- classe **JScrollPane** de Java/Swing
- classe **BufferedReader** et **InputStreamReader** de *java.io* ,

Interposition dans les serveurs d'applications:

La plupart des serveurs d'applications (MTS de Microsoft, J2EE , WebLogic, WebSphere,) utilisent des astuces dérivées de ce design pattern pour ajouter de nouvelles fonctionnalités aux composants que l'on déploie dedans.



Nouvelles fonctionnalités couramment apportées: gestion des transactions, de la sécurité et de la montée en charge.

Limitations et extension:

Pour être applicable , le design pattern décorateur doit partir d'un élément de base bien défini qu'il faudra envelopper.

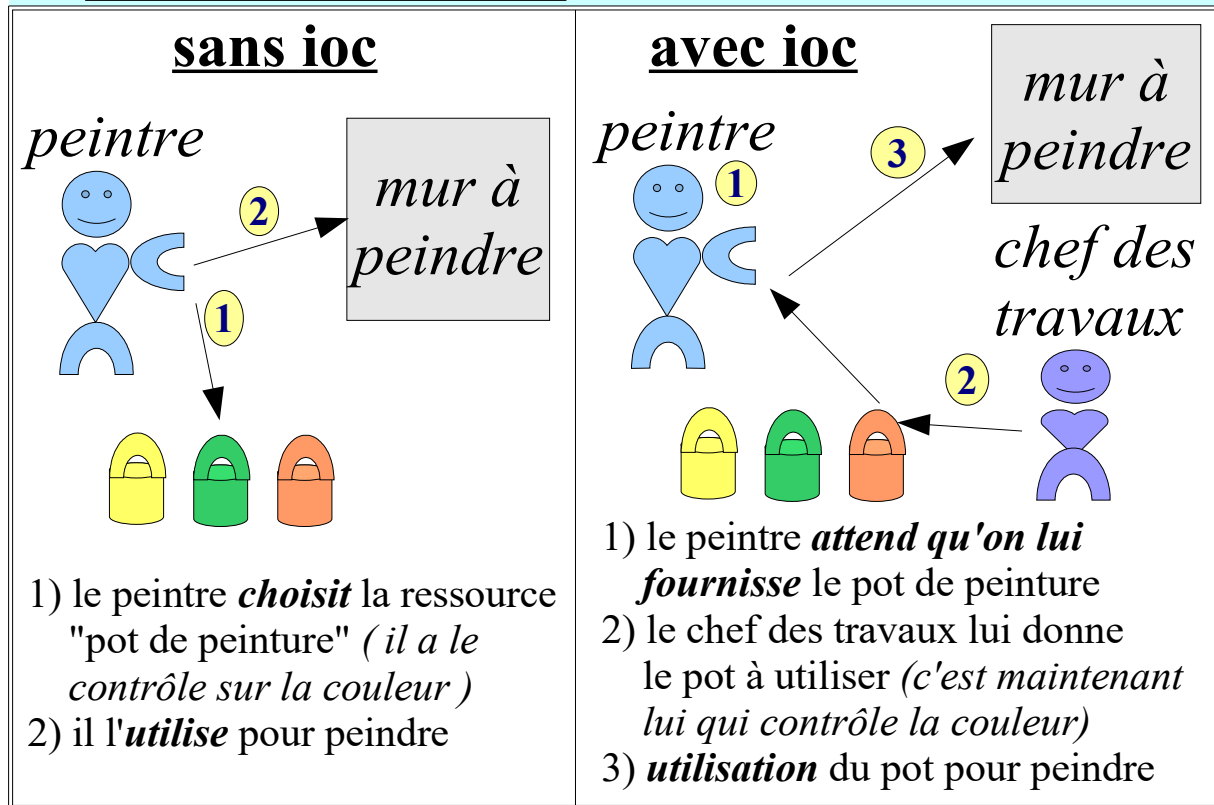
Cet élément de base doit comporter un certain nombre de méthodes à conserver dans les décorateurs. **Ce paquet de méthodes doit être en nombre fini et les prototypes de ces méthodes doivent être connus et stables.**

Lorsque les prototypes des méthodes ne sont pas connus et/ou lorsque le nombre de méthode de l'élément de base peut varier on doit alors utiliser la *programmation par aspects (A.O.P.)*.

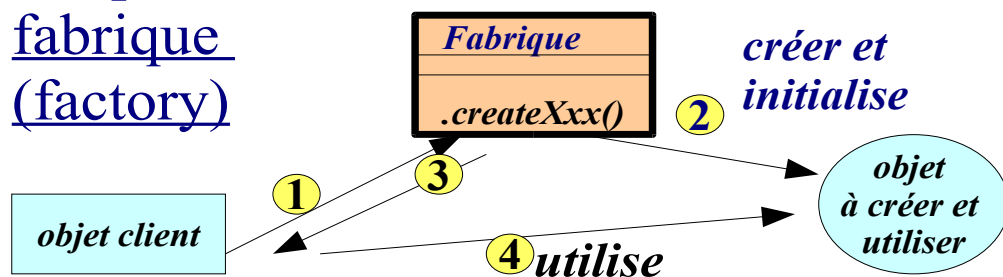
3. Injection de dépendances

Ce "Design Pattern" a 2 noms : "I.O.C." ou bien "injection de dépendances"

3.1. IOC = inversion of control



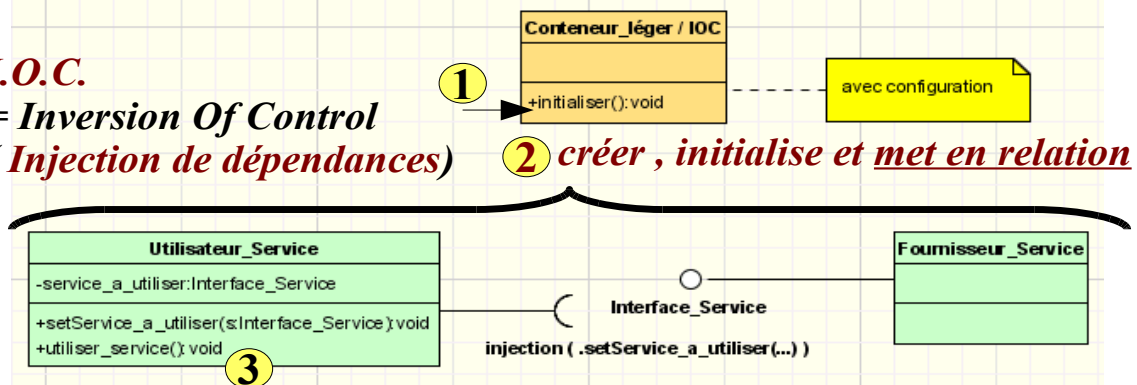
Simple fabrique (factory)



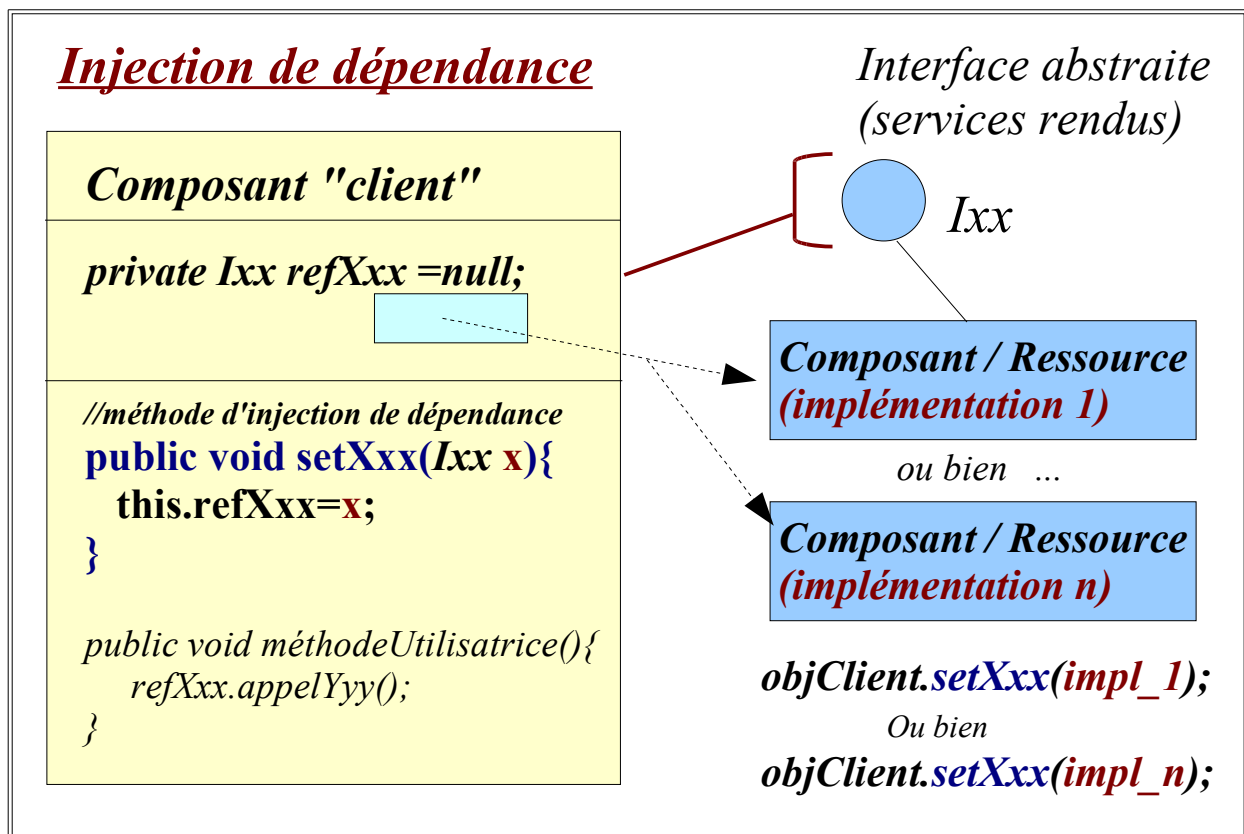
I.O.C.

= **Inversion Of Control**

(**Injection de dépendances**)



3.2. injection de dépendance



Le *design pattern* "IOC" (*Inversion of control*) correspond à la notion d'**injection de dépendances abstraites**.

Concrètement au lieu qu'un composant "client" trouve (ou choisisse) lui même une ressource avant de l'utiliser, cet **objet client exposera une méthode** de type:

public void setRessources(AbstractRessource res)

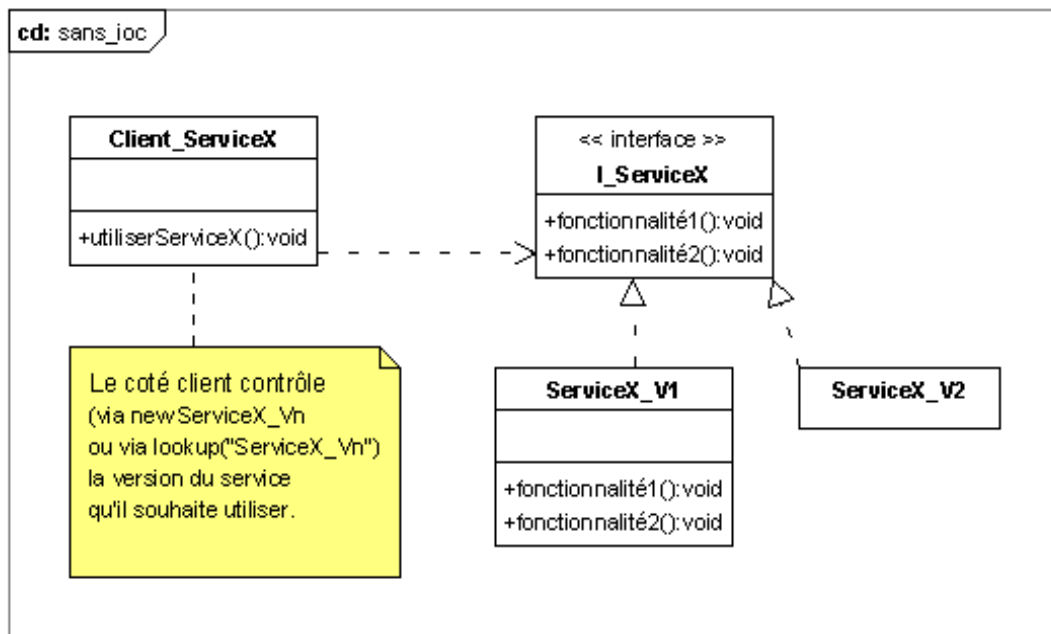
ou bien un constructeur de type:

public CXxx(AbstractRessource res)

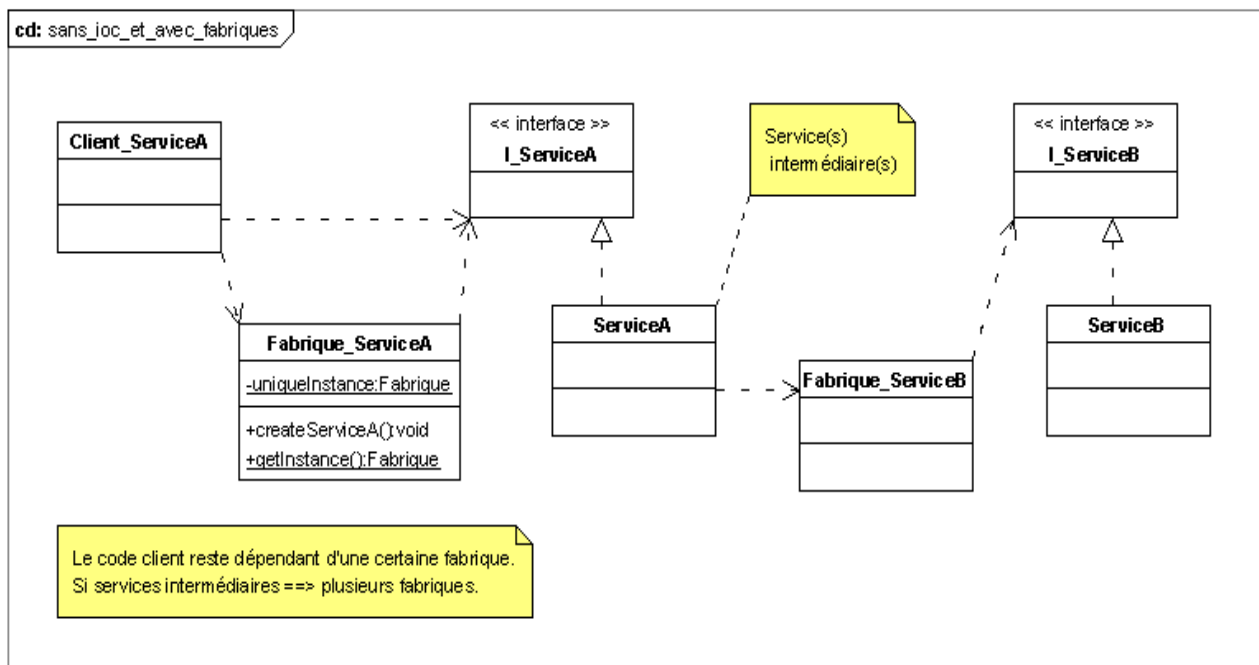
permettant qu'on lui fournisse la ressource à ultérieurement utiliser.

Un tel composant est beaucoup plus réutilisable .

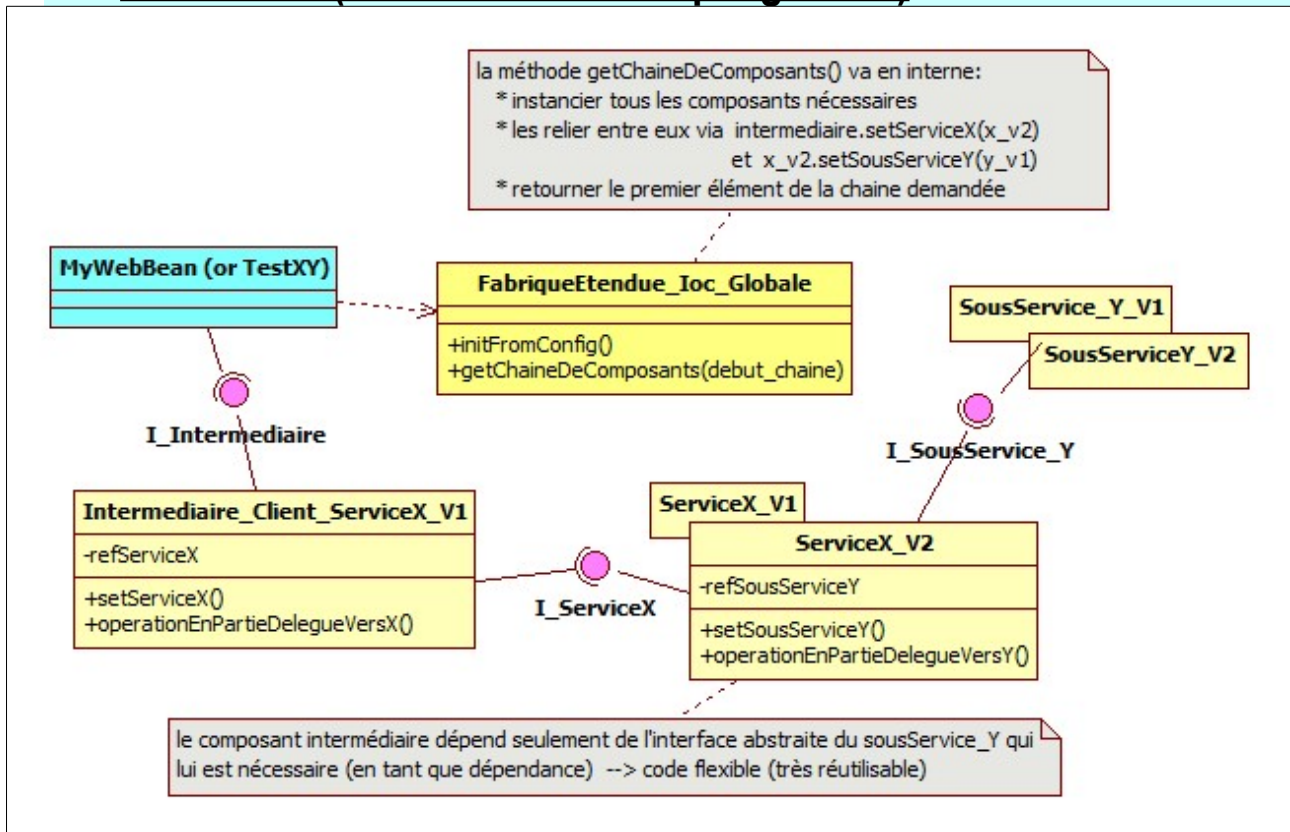
3.3. sans I.O.C. ni fabrique



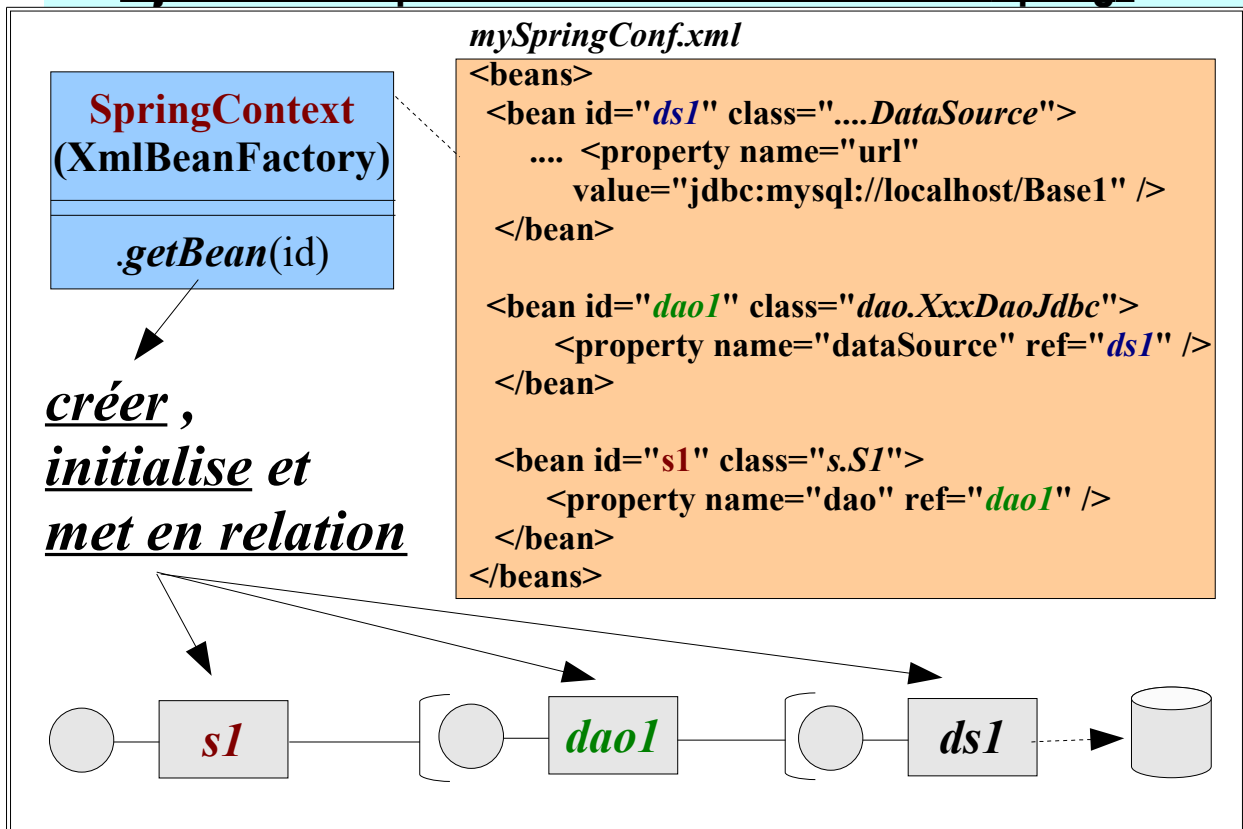
3.4. sans I.O.C. et avec de multiples petites fabriques



3.5. avec I.O.C. (et donc avec fabrique globale)



3.6. injection de dépendances avec le framework "Spring"



Autre framework d'injection : "CDI : Context and Dependency Injection" .

X - Annexe - Infrastructure Eclipse EMF / UML

1. Infrastructure pour UML, MDA, ...

1.1. Challenge

Comment modéliser et automatiser des processus de développement qui reposent sur des éléments mouvants (architectures et technologies qui évoluent sans cesse,) ?

1.2. Principales normes de l'OMG

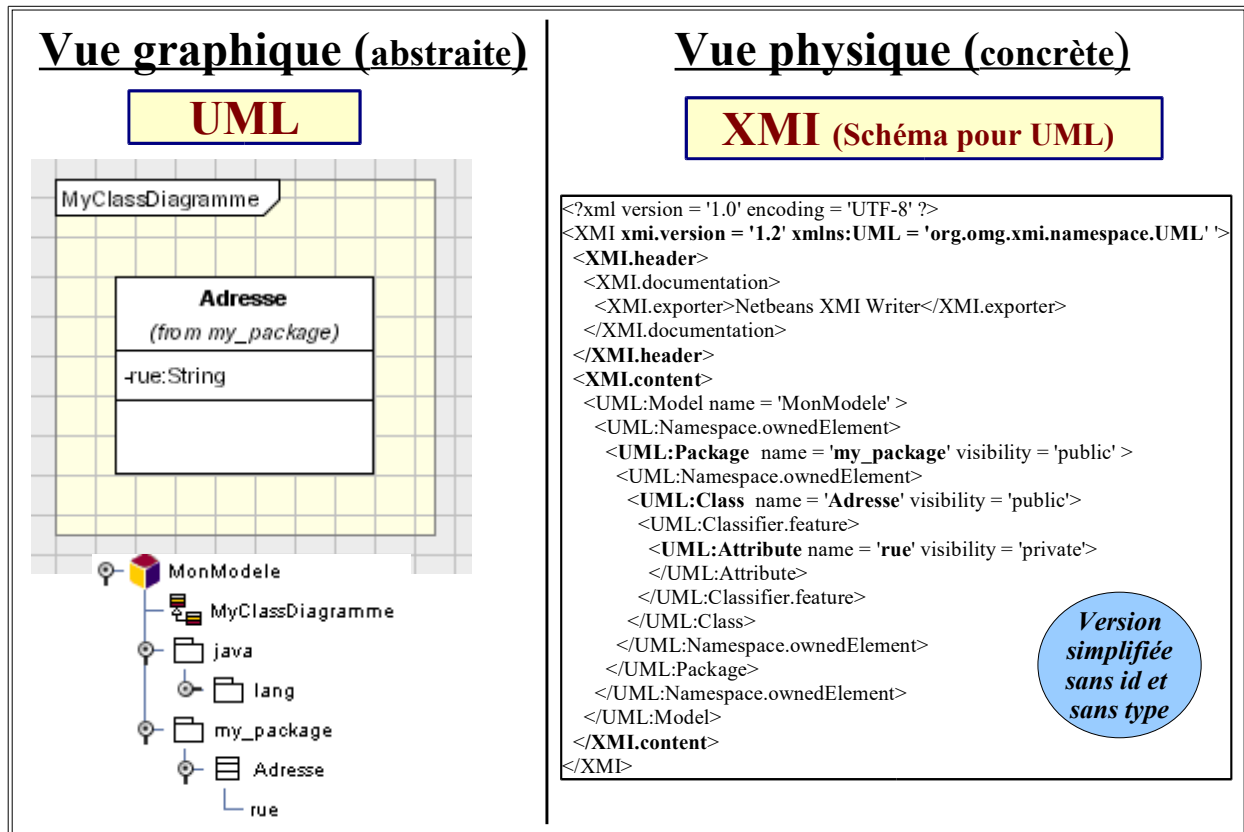
| | |
|--|--|
| UML (Unified Modeling Language) | Formalisme (notations + sémantiques) permettant de bâtir divers diagrammes correspondants à différents vues d'une modélisation orientée objet d'un système informatique |
| M.D.A. (Model Driven Architecture) | Cadre formalisé dans lequel pourra s'inscrire un processus de développement piloté par l'élaboration de divers modèles : CIM : Computation Independant Model ==> <i>niveau conceptuel / expr. besoins</i> PIM : Platform Independant Model ==> <i>modèle d'analyse</i> PSM : Platform Specific Model ==> <i>modèle de conception</i> L'objectif principal de MDA est de pouvoir paramétrer certains passages automatisés d'un modèle à un autre (partiellement). |
| M.O.F. (Meta Object Facility) | Meta-meta-modèle basé sur l'IDL de CORBA |
| X.M.I. (XML MetaData Interchange) | Fichiers XML permettant d'échanger des modèles entre divers produits (générateurs de diagrammes, outils de développement java ,) |
| CORBA / OMA <i>ORB = Object Request Broker</i> <i>(négociateur de requêtes objets) = bus logiciel généralement construit sur TCP/IP et IIOP</i> <i>CORBA = Common ORB Architecture</i> | Architecture objet distribuée basée sur un bus logiciel (l'ORB) et sur différents services annexes (nommage: COSNaming,). |
| CWM (Common Warehouse Metamodel) | Meta-(méta-) modèle commun pour les entrepôts de données et référentiels (repository avec stockage et récupération d'éléments d'une modélisation UML) |
| SPEM (Software Process Engineering MetaModel) | Meta modèle pour processus de modélisation (U.P. ,) |
| xxx Profile for UML (avec xxx = XML , Real-Time ,) | Extension standardisée d'UML vis à vis d'un certain domaine (schéma XML , temps-réel , ...) |

---> Combinaison de technologies pour offrir une infrastructure solide pour des **AGL** (Atelier de génie logiciel).

2. XMI

Le **XML Metadata Interchange (XMI)** est un standard de l'**OMG** pour échanger les metadonnées via XML. Il peut être utilisé pour toutes metadonnées dont le metamodelle peut-être exprimé en

Meta-Object Facility (MOF). L'usage le plus commun de XMI est l'échange de modèles UML.



2.1. Combinaisons éventuelles d'outils complémentaires

Outils UML généraliste (pour expression des besoins , analyse ,)

====> export XMI

----- ==> import XMI

vers plugin UML pour IDE (eclipse,)

et/ou vers ou outils MDA (AndroMDA , accéleo,)

2.2. Limitations de xmi et des imports/exports

Les opérations d'import/export (au format XMI) entre différents outils UML sont en pratique rendues délicates pour les raisons suivantes:

- différentes versions d'UML (UML 1.4 , UML2.0 , UML2.1)
- différentes versions d'XMI (xmi 1.1 , xmi 1.2 , xmi 2)
- différents moyens de stocker les coordonnées des éléments des diagrammes UML (dans fichier xmi , dans fichier annexe ,)

==> Entre des outils compléments différents (de différents éditeurs ou de différentes époques) , les opérations d'import/export sont souvent partielles (on ne récupère qu'une partie des modèles) ou quelquefois carrément inopérantes (en cas d'incompatibilité).

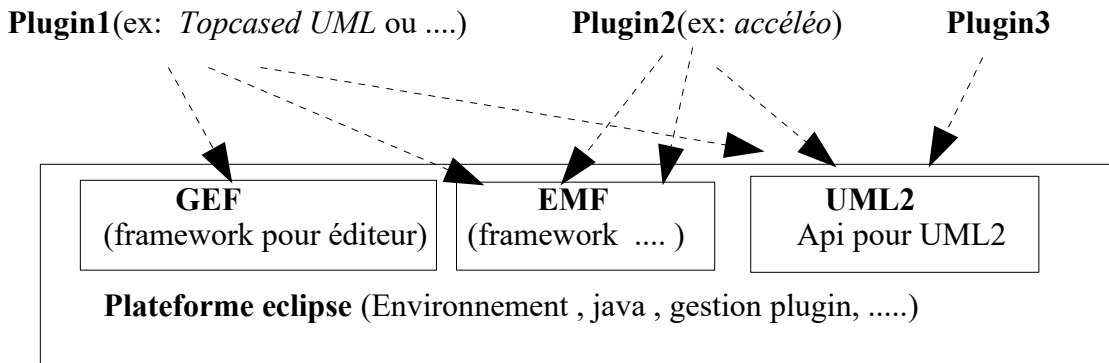
Bonne nouvelle: les versions récentes des spécifications UML et XMI sont de plus en plus précises et les imports/exports sont ainsi assez bien gérés avec des outils très récents.

Il y a tout de même des petites pertes d'informations.

2.3. Socle technique apporté par eclipse (ou équivalent)

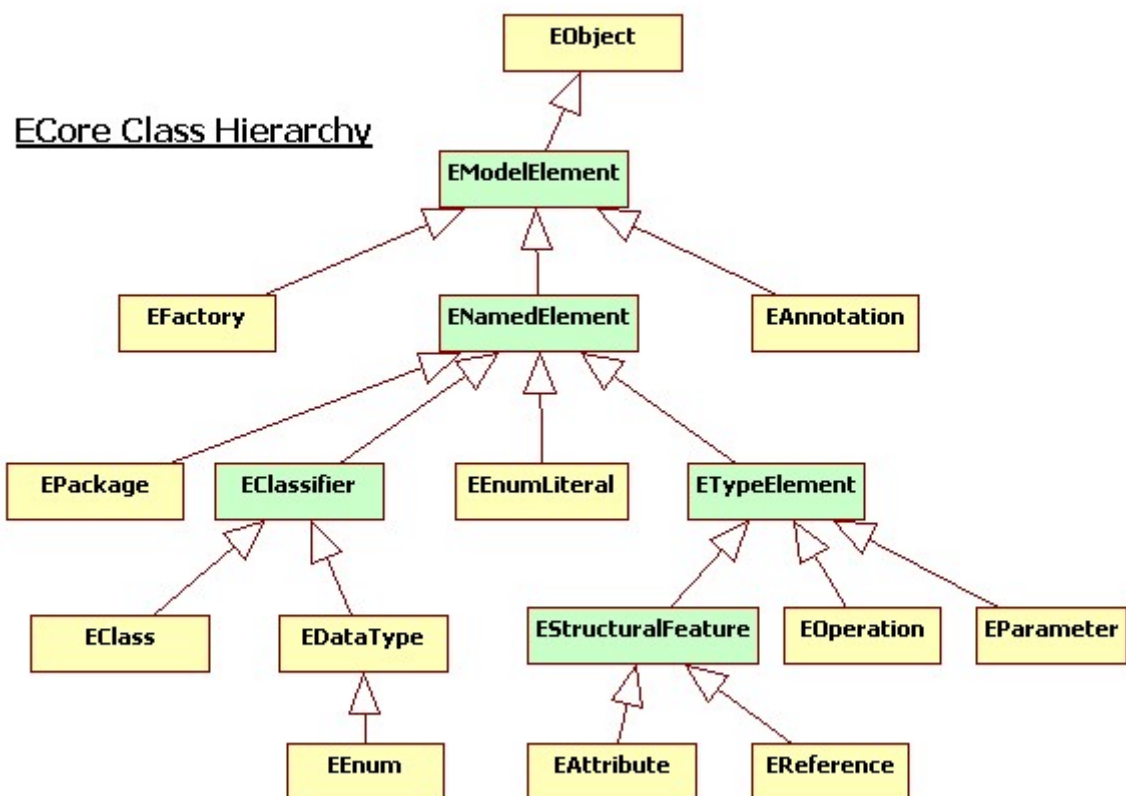
Les I.D.E. les plus populaires dans le monde du développement JAVA (eclipse , netbeans,)

offrent un socle technique important qui est très souvent utilisé en interne par les outils UML.



Ainsi , utilisant les mêmes formats internes (.xmi , .uml) en utilisant le même socle technique (GEF/EMF/UML2) , les différents plugins récents pour eclipse (Eclipse UML/Omondo , Topcased UML , accéléo ,) parviennent simplement à dialoguer/coopérer très efficacement.

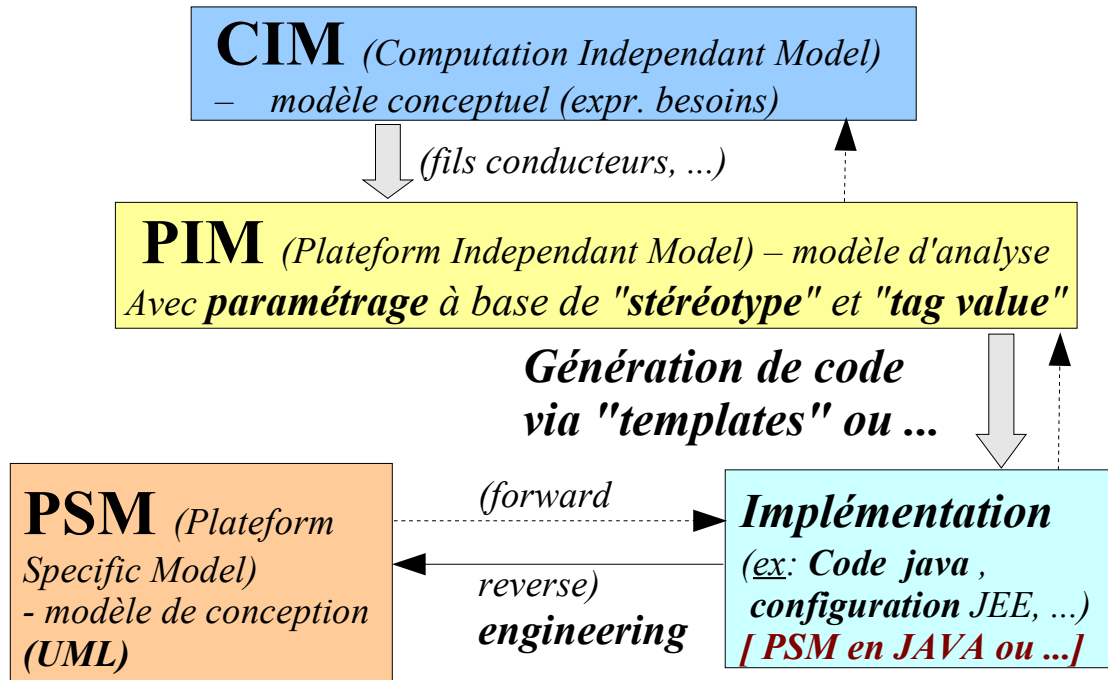
Méta modèle "ECore" (pour UML ou ...) fourni par la plateforme "Eclipse":



XI - Annexe - M.D.A. (principes)

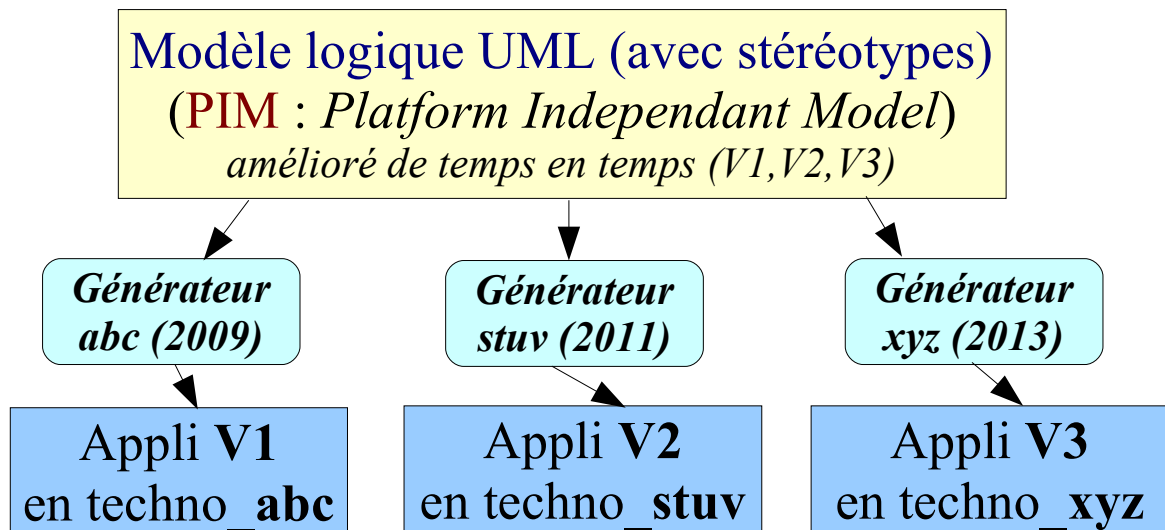
1. MDA

MDA : Model Driven Architecture

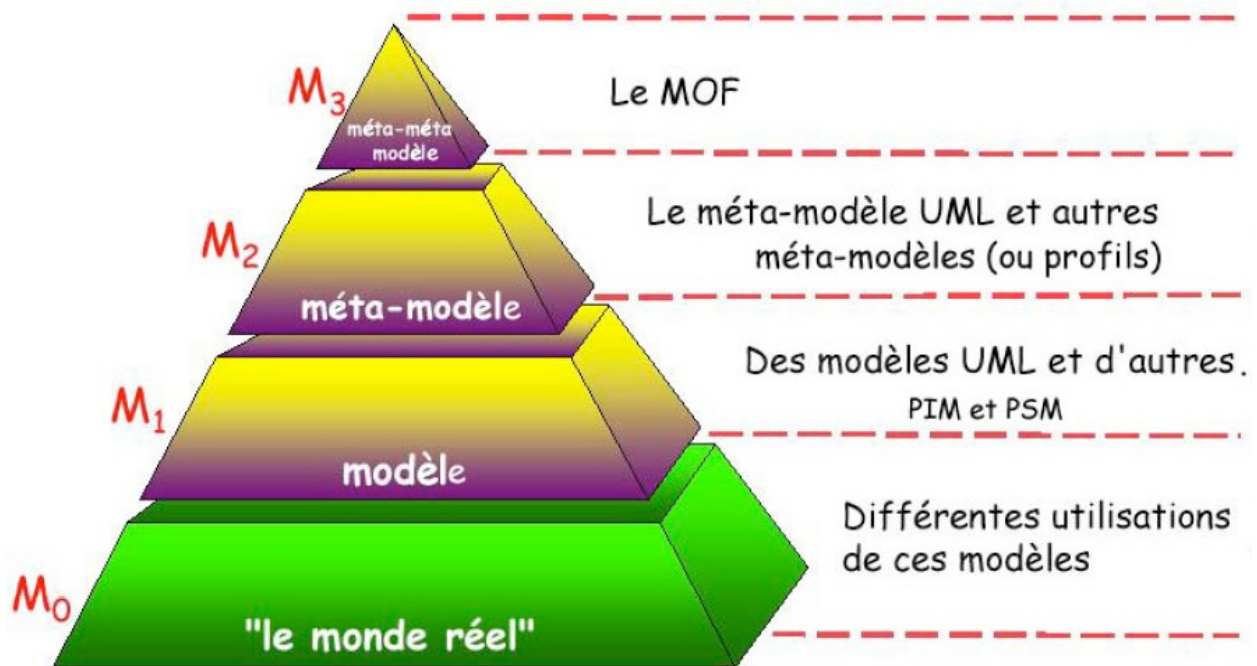


Objectif(s) de MDA:

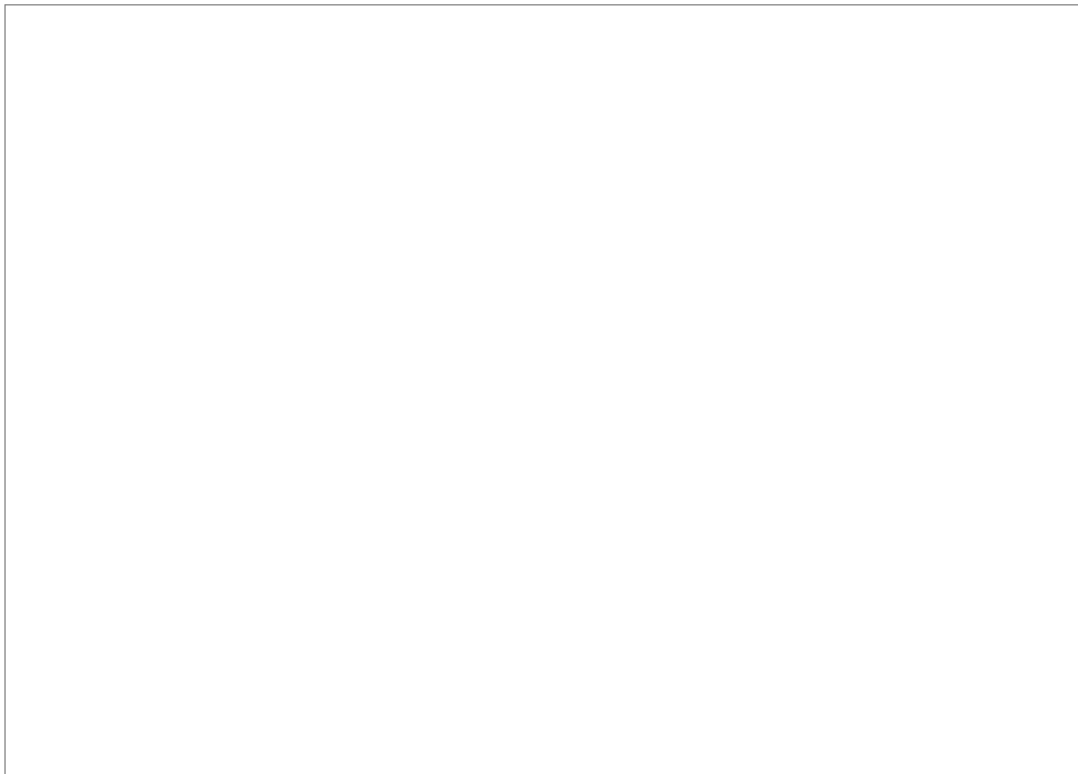
- Capitaliser sur les modèles logiques (idéalement *stables*) et sur des générateurs de code paramétrable (*templates*).
- (Re-)générer du code dans une technologie ou une autre .

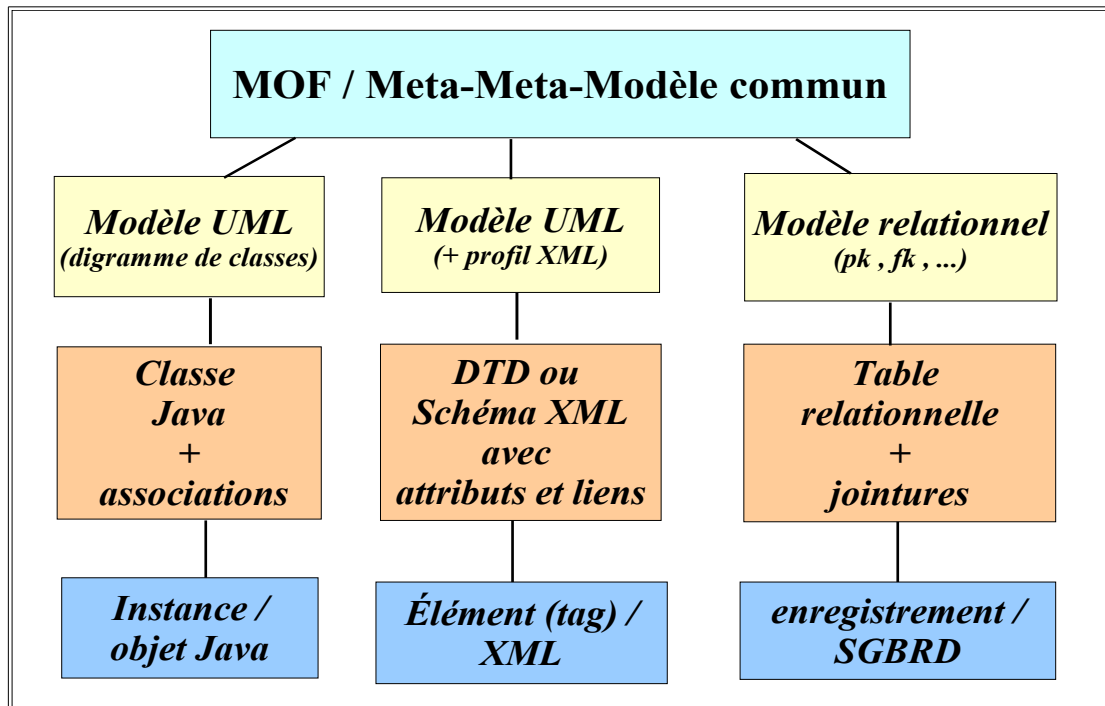


1.1. Modèles , Méta-Modèles et Méta-Méta-Modèles



- Concrètement , les niveaux M2 et M3 sont des standards à utiliser (MOF , UML,).
Le niveau M3 (MOF) parle en terme d'*entités* de *relations* et de *packages*. et les généralités abstraites de XMI (basé sur MOF et XML) sont à ce niveau .
Le niveau M2 (UML , Profil UML,) parle en terme de *classes* , d'*attributs* et de *méthodes* .
Des **versions concrètes de XMI** sont à ce niveau .
- Le niveau M1 correspond aux modèles et diagrammes à générer (ex: Classe "**Personne**")
- Le niveau Mo est le monde "réel" des objets (ex: ["**Dupond**" , "rue xxx ..." , ...])





NB: En pratique , au sein d'une application , plusieurs technologies ont besoin de coopérer :

- **Modèle relationnel** et **base de données** associée pour la **persistance**
- **Modèle objet** pour la **gestion des processus** et la **représentation des entités**
- **Documents XML** pour les **échanges**

Grosso-modo **en conception**, à partir d'un modèle assez abstrait issu de l'analyse (PIM) , on à besoin de générer **plusieurs petits modèles spécifiques (PSM)** devant être **cohérent entre eux** :

- Le schéma relationnel d'une base de données (**xxx.sql**)
- La structure des échanges XML (Schéma XML <---> Profil XML pour UML)
- La structure des objets métiers et de représentation (Java)

1.2. Quelques produits "MDA" concrets:

| <i>Outils/Générateur MDA</i> | <i>Editeur</i> | <i>Open Source ?</i> | <i>Caractéristiques</i> |
|------------------------------|----------------|----------------------|---|
| Andro MDA | | oui | Pionnier en MDA , différences selon versions, prévu pour être intégré/piloté par Maven. |
| Accéléo | Obéo (France) | oui | Très simple à paramétrer , très bonne intégration dans eclipse |
| ... | | | |

XII - Annexe – Outils / éditeurs UML

1. Quelques outils UML (Editeurs , AGL)

| Outils/AGL UML | Editeur | Open Source ? | Caractéristiques |
|--|--|--|--|
| Rational Rose --> Rational XDE ---> RSA / RSM | Rational ---> IBM | non | Ancien leader du marché (dans les années 1996/2003) .Bon produit (très complet) mais assez cher .(Rational Software Modeler) |
| Together | Borland, Microfocus | non | Outil très ancien. Évolution récente ? |
| Star UML 1.x | | oui | Produit gratuit assez complet (très inspiré de Rational Rose) .L'ancienne version 1.x n'a pas évolué depuis 2005 et n'évoluera plus (développé ancien langage "Delphi"). |
| Star UML 2 et 3 | MKLabs | partiellement | Nouvelle version de star uml entièrement redéveloppée en nodeJs (méta model json). licence d'environ 90 euros , version d'évaluation gratuite . Produit simple et intuitif . |
| Enterprise Architect | Sparx | Non mais pas cher (250 euros) | Basé sur environnement Microsoft .NET Outil assez complet , bonne ergonomie |
| MagicDraw UML | NoMagic | non | Bon produit , intègre très bien les normes récentes mais prix caché . |
| Visual Paradigm | VP | Non mais pas cher (90 euros) | Bon outil UML (complet et stable) |
| PowerAMC Designor | SDP | non | Outils pour MCD/Merise avec maintenant une partie UML |
| Visio (avec partie UML) | Microsoft | Non (environ 400 euros) | Outil graphique généraliste avec partie UML |
| Objectteering UML | Softeam (fr) | non | ergonomie très moyenne (ancien produit) |
| Modelio | Softeam (fr) | Cœur open source , extensions payantes | Bonne ergonomie , fichiers très propriétaires avec néanmoins import/export XMI. |
| Papyrus UML (plugin eclipse) | Topcased.org → Polarsys. Projet eclipse | oui | Bon Plugin UML pour eclipse (bien/ très complet mais un peut sembler compliqué au départ) |
| UML-Designer et sirius (uml + ...) | OBEO | oui | UML-Designer est assez proche de Papyrus mais ne pourra néanmoins être considéré comme concurrent sérieux que lorsque les innombrables bugs auront disparus !!! |
| GenMyModel | Startup près de Lille | non | Outil UML en ligne (nécessitant un simple navigateur) . Beaucoup de limitations en version gratuite . |
| ... | | | |

XIII - Annexe - Mise en oeuvre MDA via Accéléo

Présentation d'accéléo3_M2T (plugin eclipse)

Accéléo est un **plugin eclipse** de type "**générateur de code MDA basé sur des templates**". Ce générateur permet de lancer l'exécution d'une série de générateurs de code paramétrables. Chaque générateur de code s'écrit comme un template (modèle de code à générer).

Via une syntaxe adéquate le code d'un template peut récupérer toutes les informations d'un modèle UML (classes, attributs, opérations, stéréotypes, ...) et ainsi générer du code précis en relation directe avec les éléments de la modélisation .

On peut générer n'importe quel type de fichier texte:

- du code source java (**xxx.java**)
- des fichiers de configuration xml (**xxx.xml**)
- des pages html , jsp ou autres (**xxx.xhtml** , **xxx.jsp**)
- des structures de base de données (**xxx.sql** , ...)
-

Les principaux intérêts du produit accéléo sont les suivants:

- **générer tout un tas d'éléments cohérents entre eux** (mêmes noms logiques au sein des différentes parties : configuration xml , code java , ihm web ,)
- **Bonne séparation des tâches:**
 - l'*expert technique* (architecte ,) écrit les templates de génération de code (en connaissant à fond toute les subtilités d'une certaine technologie).
 - l'*analyste fonctionnel et le concepteur/développeur* utilisant des générateurs déjà opérationnels n'ont plus qu'à se focaliser sur les aspects métiers (algorithmes liés au règles de gestion, structure du modèle logique UML , ...)
- **(re-)générer régulièrement du code restant bien conforme au modèle UML** (pas de divergence , plans toujours à jour) ---> maintenance plus aisée.
- **gagner en productivité** sur un gros projet (ou bien sur une suite de petits projets basés sur les mêmes technologies). Attention: il faut comparer le *temps passé à écrire les modèles de code à générer* avec le *temps gagné grâce à la génération de code* .

1. Point d'entrée de la génération de code MDA : un modèle logique UML stéréotypé

Le générateur de code accéléo est un plugin pour l'IDE eclipse.

La méthode la plus simple pour produire un modèle UML consiste à utiliser un autre plugin pour eclipse de type "éditeur/modéleur UML".

Le plugin "**TopCased UML**" est un assez bon plugin UML pour eclipse.

Cet éditeur UML produit des fichiers "**xxxx.uml**" (au format **UML 2.x**) qui sont directement interprétables par le **générateur MDA accéléo** (sans transformation).

Seules difficultés notables:

- Installer les bonnes versions des plugins (topcased UML, accéléo) dans la bonne version d'eclipse.
- La préparation des stéréotypes applicables au niveau de "Topcased Uml" est assez délicate (fichier annexe à préparer , , intégration compliquée,)

2. Versions d'accéléo

Le produit accéléo est conçu par l'entreprise OBEO (France/Nantes).

Ce produit a évolué radicalement entre les versions 2.x et la version 3.

- Les versions 2.x utilisaient une syntaxe propriétaire (<% %> ou [% %])
- La version 3 utilise maintenant la syntaxe normalisée issue des spécifications "(MOF)M2T" de l'OMG.
- D'autre part, l'intégration du plugin "accéléo" au sein d'eclipse est assez différente entre les versions 2.x et 3 (fichier de paramétrage et de lancement différents).
- On peut donc considérer que les plugins Accéléo 2.x et 3.x sont bien différents (même s'ils apportent à peu près les mêmes fonctionnalités).

La suite de ce document correspond à ACCELEO 3 (ECLIPSE ACCELO MODEL TO TEXT).

3. Installation du plugin accéléo3 (M2T) sous eclipse

Help/software update ...

<http://download.eclipse.org/modeling/m2t/updates/site.xml> / *M2T_ACCELEO_SDK 3.0.1*

également besoin du plugin UML2 :

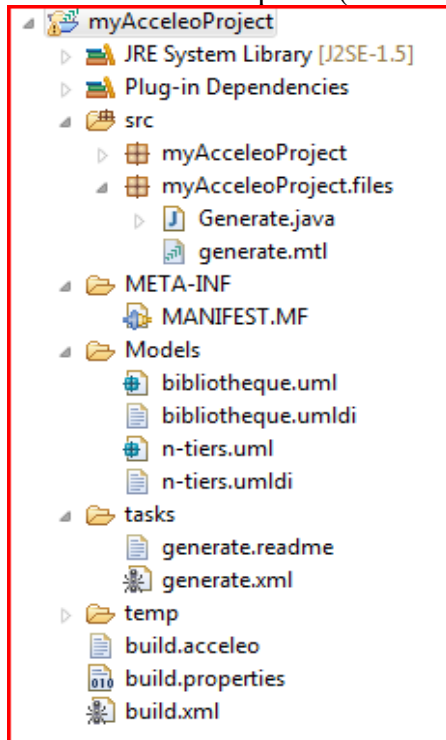
<http://www.eclipse.org/modeling/mdt/?project=uml2>

---> **mdt-uml2-Update-3.1.1.zip** + update from archive.

4. Création d'un projet "accéléo" sous eclipse

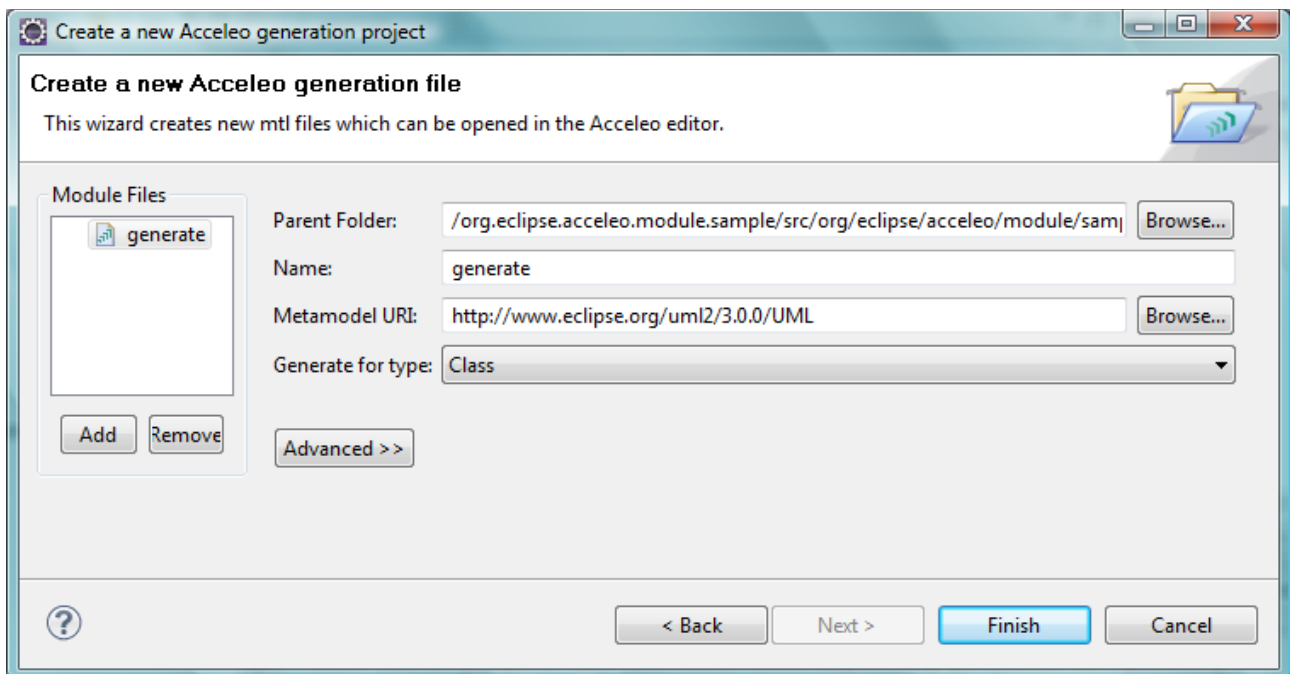
L'utilisation d'accéléo nécessite la mise en place d'un projet de "génération de code" (new other/... /accéléo model to text / Accéléo Project). Celui-ci contiendra :

- les templates de génération de code (+ classes utilitaires en arrière plan)
- d'éventuels modèles UML (ici ou ailleurs)
- un éventuel script ant (build.xml) de lancement



A côté de ce projet accéléo, il y a aura des projets classiques (java, ejb, dynamic web project, ...) dont une grande partie du code sera généré par accéléo et une partie sera complétée (code ajouté par le développeur).

NB: Lors de la création du projet Accéléo, choisir "<http://www.eclipse.org/uml2/3.0.0/UML>" comme "metamodel URI".



5. Exemple de "template" (modèle de code à générer)

generateJavaBean.mtl

```
[comment encoding = Cp1252 /]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]

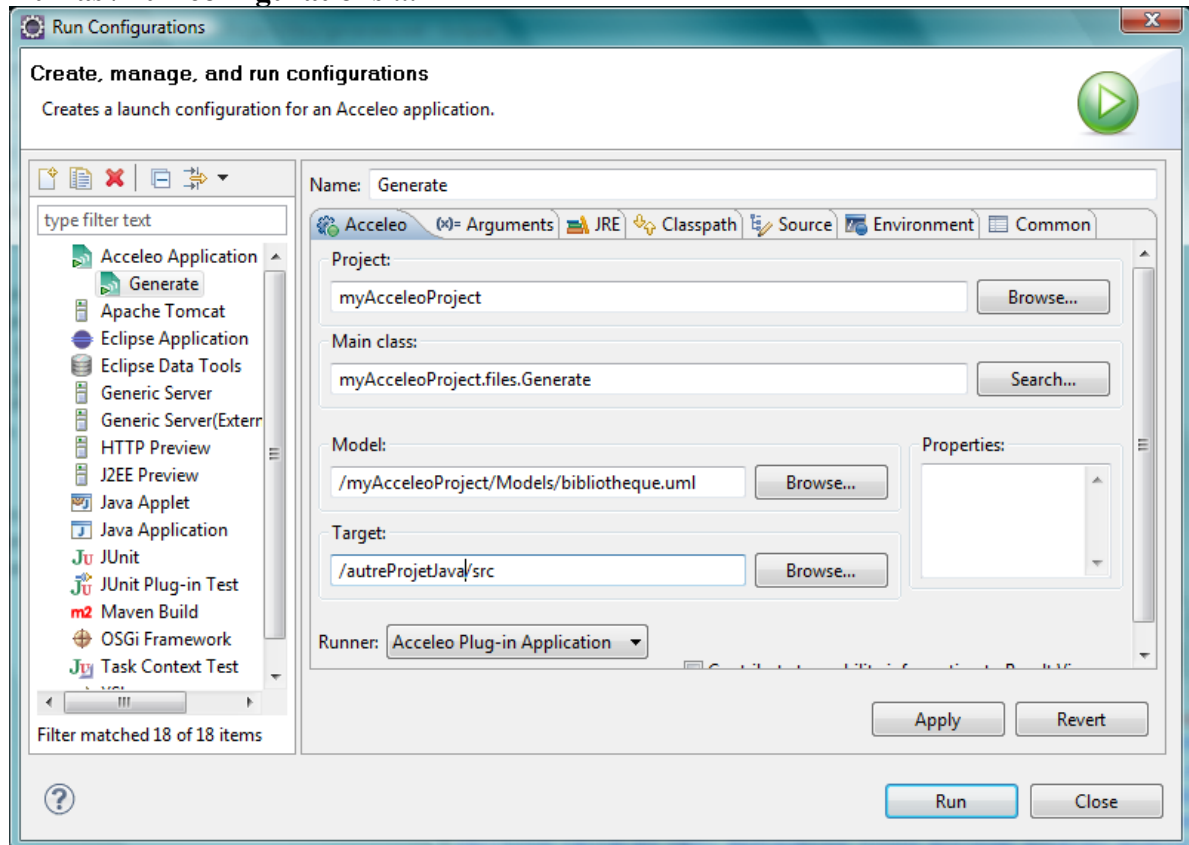
[template public generate(c: Class)]
  [comment @main /]
  [file (c.name.concat('.java'), false)]

  public class [c.name.toUpperFirst()/] {
    [for (p: Property | c.attribute)]
    private [p.type.name/] [p.name/];
    [/for]
    [for (p: Property | c.attribute)]
    public [p.type.name/] get[p.name.toUpperFirst()/]() {
      return this.[p.name/];
    }
    [/for]
    [for (o: Operation | c.ownedOperation)]
    public [o.type.name/] [o.name/]() {
      // TODO should be implemented
    }
    [/for]
  }
[/file]
[/template]
```

6. lancement unitaire d'une génération de code

Pour lancer une génération de code à partir d'un template à déclencher:

- 1) se placer sur le modèle à déclencher (generate.mtl ou ...)
- 2) **Run as / run configurations ...**



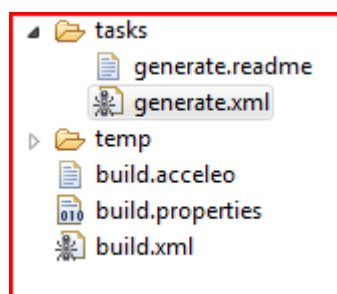
et choisir le fichier "modèle uml" de départ
le répertoire où le code doit être généré.

- 3) **Run as / Launch Acceleo Application**

7. lancement d'une génération de code via script ant

Si l'on tient à lancer d'un seul coup plusieurs générations de code à partir de plusieurs templates, on peut s'appuyer sur un script ANT .

Il faut pour cela générer un répertoire (folder) de nom "**tasks**" à la racine du projet. Quelques instants plus tard (lors d'une génération ou ...) , les fichiers **generate.xml** et **generate.readme** seront automatiquement générés dans le répertoire tasks.



Il faudra éventuellement mettre à jour la propriété `ECLIPSE_HOME` dans tasks/generate.xml :

```
<property name="ECLIPSE_HOME"
```

```
value="C:\\Prog\\java\\eclipse\\e36\\helios_je_e_m2e_acceleo\\eclipse"/>
```

On pourra ensuite s'inspirer de generate.readme pour écrire un script ant (**build.xml**) ressemblant au suivant:

```
<?xml version="1.0" encoding="UTF-8"?>

<project basedir="." default="generateSample" name="myAcceleoProjectSample">
  <import file="../myAcceleoProject/tasks/generate.xml"/>
  <property name="MODEL" value="${basedir}/Models/bibliotheque.uml"/>
  <property name="TARGET" value="${basedir}/temp"/>
  <target name="generateSample" description="Generate files in '${TARGET}'">
    <antcall target="generate" >
      <param name="model" value="${MODEL}"/>
      <param name="target" value="${TARGET}"/>
    </antcall>
    <!-- eventuels autres antcall ici -->
  </target>
</project>
```

8. Module , sous modules et importation

generate.mtl

```
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]
[import utilJava /]
....
```

utilJava.mtl

```
[comment encoding = Cp1252 /]
[module utilJava('http://www.eclipse.org/uml2/3.0.0/UML')/]
...
```

Un module xxx correspond à un fichier xxx.mtl
et via l'instruction [import yyy/] ou bien [import pla::plb::plc:yyy /] on peut importer/inclure un sous module yyy .

Remarque: la syntaxe des fichiers .mtl ressemble à celle d'xml (avec des [] à la place des <>).

9. Templates , sous templates et applications

Un template correspond à une partie de code qui sera (directement ou indirectement) généré.
Chaque template à un nom et est attaché à un type d'objet sur lequel il est applicable.

Dans l'exemple suivant , le template "operationBody" est applicable sur des objets de type "Operation" :

```
[template public operationBody(o: Operation)]
public [o.type.name/] [o.name/] () {
    // [protected (o.name)]
    // TODO should be implemented
    // [/protected]
}
[/template]
```

Ce (sous-)template pourra être appelé par un template de plus haut niveau:

```

[template public generate(c: Class)]
  [comment @main /]
[file (c.name.concat('.java'), false)]

public class [c.name.toUpperFirst()/] {
  ...
  [/for (o: Operation | c.ownedOperation)]
  [o.operationBody() /]
  [/for]
}
[/file]
[/template]

```

Lors de son activation/invoque, un template est souvent préfixé par l'objet sur lequel il s'applique (**o.operationBody()** /] dans l'exemple précédent .

Seuls certains templates sans argument sont activés sans être préfixés .

Exemple:

```

[template public importBlock()]
// [protected ('for imports')]
import java.util.*;
// [/protected]
[/template]

```

invoqué via

```
[importBlock()/]
```

En décomposant des templates en sous templates, on arrive ainsi à une structure modulaire .

9.1. Pré-conditions

Via la syntaxe **? (condition)** , il est possible d'indiquer une condition à vérifier pour qu'un template puisse être déclenché :

Exemple:

```

[template public genAssociation(p : Property) ? (owningAssociation <> null and isOrdered)]
...
[/template]

```

On peut aussi écrire différentes versions d'un template de même nom et avec des conditions différentes:

```

[template public genAssociation(p : Property)? (owningAssociation <> null and not isOrdered)]
...
[/template]

```

9.2. Post-traitement

Via la syntaxe **post (...)** on peut indiquer un traitement qui sera effectué après la génération initiale du template pour re-formater le résultat généré .

Ceci est essentiellement pratique pour supprimer via trim() le retour chariot "\n" final lorsque le résultat d'un sous template est prévu pour être ultérieurement concaténé avec d'autres éléments sur

une même ligne.

Exemple:

```
[template public javaName(c : Class) post ( trim() ) ]
    [c.name.toUpperFirst() /]
[/template]
```

9.3. Variable

Dans un template (et dans d'autres blocs), on peut définir des variables via la syntaxe
{ nomVar : Type = valeurInitiale ; }

Exemple:

```
[template public genXxx(c : Class)
    { nomClasse : String = javaName();
      nomComplet : String = packageName().concat('.').concat(nomClasse);
    }]
.... [nomClasse/]
.... [nomComplet/] ...
[/template]
```

9.4. Délimitation des parties à ne pas écraser lors d'une re-génération

```
[template public operationBody(o: Operation)]
public [o.type.name/] [o.name/] () {
    int i=0; //partie re-générée à chaque fois
        // [protected (o.name)]
            int j=0; //code initial (première génération)
            // partie à coder par le développeur
            // le code amélioré ici ne sera pas écrasé
        // [/protected]
    }
[/template]
```

==> code généré :

```
public int methodeXy(){
int i=0 ;
// start of user code of methodeXy
    int j=0; //...
// end of user code of methodeXy
}
```

et si l'operation "methodeXy" n'existe plus dans le modèle UML , son ancien code se retrouve automatiquement sauvegardé dans un fichier annexe "xxxxx.lost+found.txt" .

10. Quelques éléments de syntaxe

```
[template public xyz(c : Class)]
    [file ('fichierXyz', false , 'Cp1252')]
    [comment file(pathName,append_mode,jeuxCaractères ) /]

    [for (e : TypeElement | collectionAparcourir)]
        [if (condition)]
        ....
```

```

    [/for]
  [/file]
[/template]

```

[comment /]

ou bien

```

[comment]
...
...
[/comment]

```

```

[for (e : TypeElement | collectionAparcourir)]
  [if (condition)]
    ...
    [elseif (condition_alternativeA) /]
    ...
    [elseif (condition_alternativeB) /]
    ...
  [/if]
[/for]

```

NB: la boucle for peut être étendue avec before() , separator() et after() de façon à rapidement formater une liste ou séquence d'éléments.

```

[ for (Sequence{1, 2, 3}) before ('liste: ') separator (' ') after (';')] [self/] [for]

```

générera le texte suivant:

liste: 1, 2, 3;

variable (finale/constante) locale à un bloc "let":

```

[let var1 : String = expression_xy()]
  ... [var1/] ...
[/let]

```

--> seul intérêt = stocker dans une variable le résultat d'une expression pour ne pas la ré-exécuter plusieurs fois .

Attention : le block [let ...] [/let] à une signification différente du mot clef "let" d'OCL .

11. Générateurs sophistiqués

11.1. Requêtes (query)

Une requête comporte un nom et un type en entrée (comme pour les templates)

Elle peut cependant retourner autre chose que des "String" .

Une "query" peut retourner des collections , des objets ,

Son coeur est exprimé en **OCL** .

Exemple:

```
[query public getPublicAttributes(c : Class) : Set(Property) =
  c.attribute->select(visibility = VisibilityKind::public)
/]
```

11.2. Boucle sur les attributs et opérations

```
[for (p: Property | c.attribute)]
  private [p.type.name/] [p.name/];
[/for]
[for (o: Operation | c.ownedOperation)]
  public [o.type.name/] [o.name/]() {
    // TODO should be implemented
  }
[/for]
```

NB : cette version simpliste est à améliorer de façon à :

- tenir compte des types de données à adapter (ex : Integer vers int ou ...)
- générer automatiquement les "getter/setter"
- retourner une valeur par défaut compatible avec le type de retour (ex : return 0 ; ou return null ;)

11.3. Prise en compte des stéréotypes

```
[query public queryAsNoStereotype(e : Element) : Boolean =
  e.getAppliedStereotypes()->size() < 1 /]

[query public queryAsStereotype(e:Element,sName:String):Boolean =
  e.getAppliedStereotypes()->exists(s | s.qualifiedName=sName) /]
```

et

```
[if (c.queryAsStereotype('n-tiers::entity'))]
@Entity
[/if]
```

11.4. Génération de pages JSP ou xhtml

Les modèles ".mtl" sont structurés avec des balises dont les noms sont volontairement délimités par des "[" "]" plutôt que par des "<" ">" comme le fait le standard UML.

Ceci permet donc d'écrire facilement des templates de pages HTML ou JSP au sein desquels les balises JSF ou HTML à générer seront délimitées par des "<" ">" et où les expressions d'extraction d'informations depuis le modèle UML seront délimitées par des "[" "]" .

D'autre part, de façon à générer automatiquement certaines balises de saisies (ex : `<input type="text" />` ou bien `<h:inputText value="#{mBean.property}" />`) on pourra prévoir des stéréotypes de type "`<<in>>`" ou "`<<input>>`" , "`<<out>>`" ou "`<<output>>`" dans le modèle UML .

Exemple simple :

generateSimpleHtml.mtl

```
[comment encoding = Cp1252 /]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]

[template public generate(c: Class)]
  [comment @main /]
```

```
[file (c.name.concat('.html'), false)]
<html>
<head>
  <title>[c.name/]</title>
</head>
<body>
  <form>
    [for (p: Property | c.attribute)]
      [p.name/]: <input type="text" /> <br/>
    [/for]
  </form>
</body>
</html>
[/file]
[/template]
```

11.5. Génération d'un fichier global (basé sur modèle complet)

```
[template public genSqlDropTable(m: Model)]
[for (c : Class | m.eAllContents())]
  [if (c.queryAsStereotype('n-tiers::entity'))]
DROP TABLE IF EXISTS [c.name/];
  [/if]
[/for]
[/template]
```

et

generateSql.mtl

```
[comment encoding = UTF-8 /]
[module generateSql('http://www.eclipse.org/uml2/3.0.0/UML')/]

[template public generateSqlFile(m: Model) ]
[comment @main /]
[file ('myGeneratedDataBaseStructure.sql', false)]
CREATE DATABASE IF NOT EXISTS [m.name/];
USE [m.name/];
[m.genSqlDropTable()/]
[m.genSqlCreateTable()/]
[/file]
[/template]
```

11.6. Propriétés de configuration générale

Solution 1: via fichier "xxxx.properties" annexe au modèle UML et au template.

...

utilisation depuis un template:

....

Solution 2 : via des propriétés d'une instance d'un modèle UML :

...

utilisation depuis un template:

....

11.7. Génération d'une structure relationnelle (sql)

```
[template public genSqlCreateTable(m: Model) ]
[for (c : Class | m.eAllContents())]
    [if (c.queryAsStereotype('n-tiers::entity'))]
        [c.sqlCreateTable()/]
    [/if]
[/for]
[/template]

[template public sqlCreateTable(c: Class)]
CREATE TABLE [c.name/](
    [for (p: Property | c.attribute)]
        [p.name/] [p.sqlStringTypeFromUmlProperty()/]
        [p.sqlColumnAttrFromUmlProperty()/],
    [/for]
    [c.foreignKeyStringWithTypeFromAssociationsOfUmlClass()/]
    PRIMARY KEY ([c.getStringOfPkFieldList()/]);
[/template]

[template public sqlStringTypeFromUmlProperty(p:Property)
post(trim())
{ umlType : String = p.type.name;} ]
    [if (umlType = 'int' or umlType = 'short' or
umlType='Integer')]
        integer
    [elseif (umlType = 'long')]
        integer
    [elseif (umlType = 'double')]
        double
    [elseif (umlType = 'float')]
        float
    [elseif (umlType = 'java.util.Date' or umlType='Date')]
        DATE
    [elseif (umlType = 'String')]
        VARCHAR(64)
    [else] [comment par défaut --> integer comme type de fk /]
        [if (p.type.oclIsTypeOf(Enumeration)) ]
            VARCHAR(24)
        [else]
            integer
        [/if]
    [/if]
```

[/template]

XIV - Essentiel outil "Papyrus UML"

1. Utilisation de Papyrus UML (éditeur UML2 eclipse)

(MDT) **Papyrus (UML)** est un éditeur UML open source qui fonctionne sous forme de *plugin* pour l'environnement de développement **ECLIPSE** (très utilisé pour le développement d'applications en JAVA). Le tout étant basé sur JAVA, **Papyrus_UML** est "multi-plateforme" et fonctionne entre autres sur **Windows**, **Unix/Linux**, **Mac**.

Historiquement, les premières versions de Papyrus ont été conçues par le **CEA**.

Un projet similaire "**Topcased UML**" avait été pris en charge par "**Airbus + écoles et universités proches de Toulouse**". En 2011/2012/2013/2014, ces deux produits sont actuellement en train de fusionner (en même temps que s'effectue une reprise officielle de la partie "éditeur UML papyrus" par la communauté "eclipse").

En tant que "(sous) projet eclipse officiel", l'éditeur "**Papyrus UML**" peut s'intégrer (par simple ajout) dans le tout dernier **eclipse** (ex : 4.3 / Kepler en 2013/2014).

En tant que "sous partie" des versions récentes de "**Topcased UML RCP**", "**Papyrus UML**" peut être rapidement utilisé dans **un environnement tout intégré** (avec générateur de code java et de documentation).

L'un des principaux intérêts des outils "Topcased UML" et "Papyrus UML" tient dans le fait qu'en tant que "plugins bien intégrés à eclipse", on peut les utiliser conjointement avec d'autres plugins importants (ex: accéléo_M2T, gendoc2, ...).

Ainsi à partir d'un même outil ECLIPSE, on peut:

- **créer/paramétrer des modèles UML** (diagrammes avec stéréotypes)
- **(re-)générer automatiquement de la documentation au format ".doc" ou ".odt"** (avec le plugin intégré gendoc2) pour produire des spécifications
- **(re-)générer automatiquement une bonne partie du code de l'application** (avec le plugin "accéléo_M2T" / MDA).

Ceci permet de travailler efficacement avec de bon atouts pour obtenir des éléments produits (spécifications, code, tests) bien **cohérents** entre eux.

NB: Pour bien utiliser cette plateforme de développement (et ses différents plugins), il faut savoir effectuer les **bons paramétrages** à chaque niveau:

- bien paramétrer les modèles UML (avec des stéréotypes adéquats)
- bien paramétrer la génération de documentation (avec des "templates" de "docs")
- bien paramétrer la génération de code (avec des "templates" de code)

Tout ceci correspond au final à **un investissement en "temps de mise au point"** qui peut se rentabiliser sur des projets importants.

1.1. Intégration/installation de Papyrus UML

Préalable : une machine virtuelle JAVA (idéalement JDK 1.6 ou 1,7) doit être installée sur le poste de développement.

Il y a au final deux grands modes d'intégration de "MDT Papyrus UML":

- intégration pré-établie (prête à être téléchargée)
---> **Topcased RCP** (*Rich Client Platform*)
- intégration spécifique (à construire soit même en partant d'ECLIPSE et en y ajoutant un à un tous les plug-ins jugés utiles : Papyrus_UML , accéléo_M2T , gendoc2 , ...)

NB:

En 2013, pour la version "Eclipse 4.3 / Kepler" , le mode opératoire pour installer le plugin "MDT Papyrus" est le suivant :

Menu Help / Software Update , ...

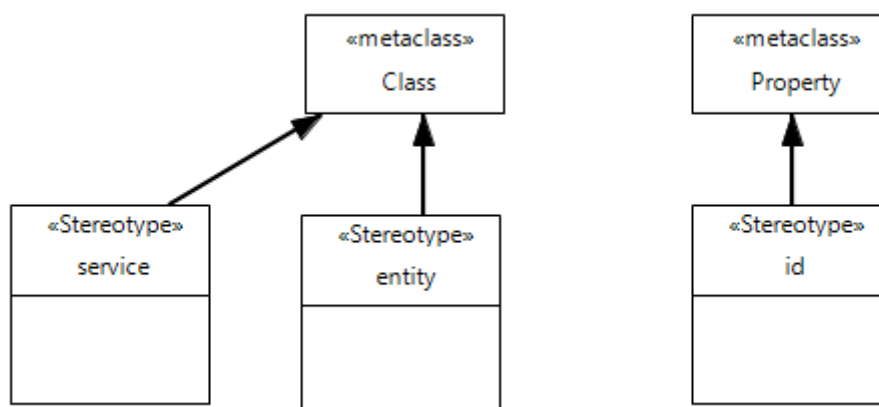
releases kepler / **Modeling** / (modeling components) / **papyrus**

1.2. Création d'un profil UML (avec stéréotypes)

Pour placer des stéréotypes (ex: <<entity>> , <<id>>) dans un modèle UML , il faut d'abord les créer dans un fichier de type "**UML profile**".

Mode opératoire:

- **New / Other ... / Papyrus / Papyrus model**
- **Select "Profile" et "UML profile diagram" (+include template "primitives types").**
- Créer de nouveaux **stéréotypes** (depuis la palette et en renseignant leurs **noms**).
- Placer et paramétrer des "**metaclass**" (ex: "Class" , "Property" , ...) .
- Relier les "**stéréotypes**" aux "**metaclass**" via des flèches d'extension pour indiquer "**sur quoi les stéréotypes seront applicables**".
- Bien sauvegarder le fichier généré (et choisir une **version**).



NB: un stéréotype peut éventuellement comporter des propriétés.

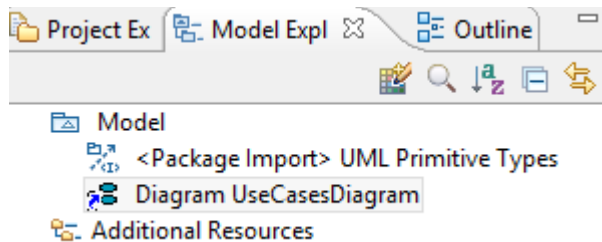
Quelques idées de stéréotypes:

| | |
|--|--|
| <<entity>> | Entité persistante |
| <<id>> | Identifiant (proche de "clef primaire") |
| <<service>> | Service métier |
| <<requestScope>> , <<sessionScope>> , <<applicationScope>> | Scope / portée des objets de la partie "IHM_WEB" |
| <<stateful>> , <<stateless>> | Avec ou sans état (traitements ré-entrants et partagés ?) |
| <<facade>> , <<dao>> , <<dto>> , ... | Design pattern / pour la conception |
| <<in>> , <<out>> , <<inout>> , <<select>> | Pour paramétrer les fonctionnalités souhaitées coté IHM (entrée/saisie , sortie/affichage , sélection , ...) |
| <<transactional>> , <<CRUD>> | Fonctionnalités diverses attendues (transactionnel , ...) |
| <<module_web>> , <<module_services>> , <<database>> | Types de composants |
| ... | |
| ... | |

1.3. Création et initialisation d'un modèle UML (pour application)

Mode opératoire:

- Créer un nouveau modèle via le menu "New / Other.../ Papyrus/ Papyrus Model"
- Sélectionner "UML" et (include template "primitives types")
- Sélectionner éventuellement le nom et le type de diagramme initial (ex : UsesCases ou "Class") (NB : d'autres diagrammes pourront être ajoutés ultérieurement au modèle).
- Se situer au sein du modèle UML via la vue "Model explorer" :



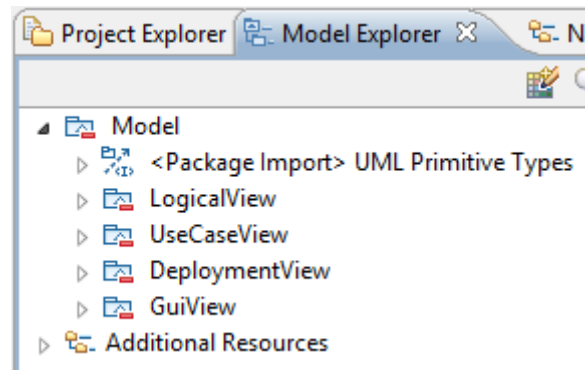
- Sélectionner la racine du modèle UML (Model) et ajuster la propriété "profiles" en "cliquant" sur "+" et en sélectionnant le fichier ".....profile.uml" .
- ...
- Bien (re-)sauvegarder le fichier du modèle UML.

Remarque :

L'assistant de création de modèle "uml papyrus" ne crée pour l'instant qu'un point de départ très rudimentaire (rien ou un diagramme au choix à la racine du modèle).

Si l'on souhaite mieux organiser la structure interne d'un modèle papyrus UML , on peut éventuellement se placer à la racine "Model" et créer (via le menu contextuel "add Child / new Model") des sous modèles de type "UseCaseView" , "LogicalView" de façon à mieux ranger les parties du modèle applicatif.

Exemple :

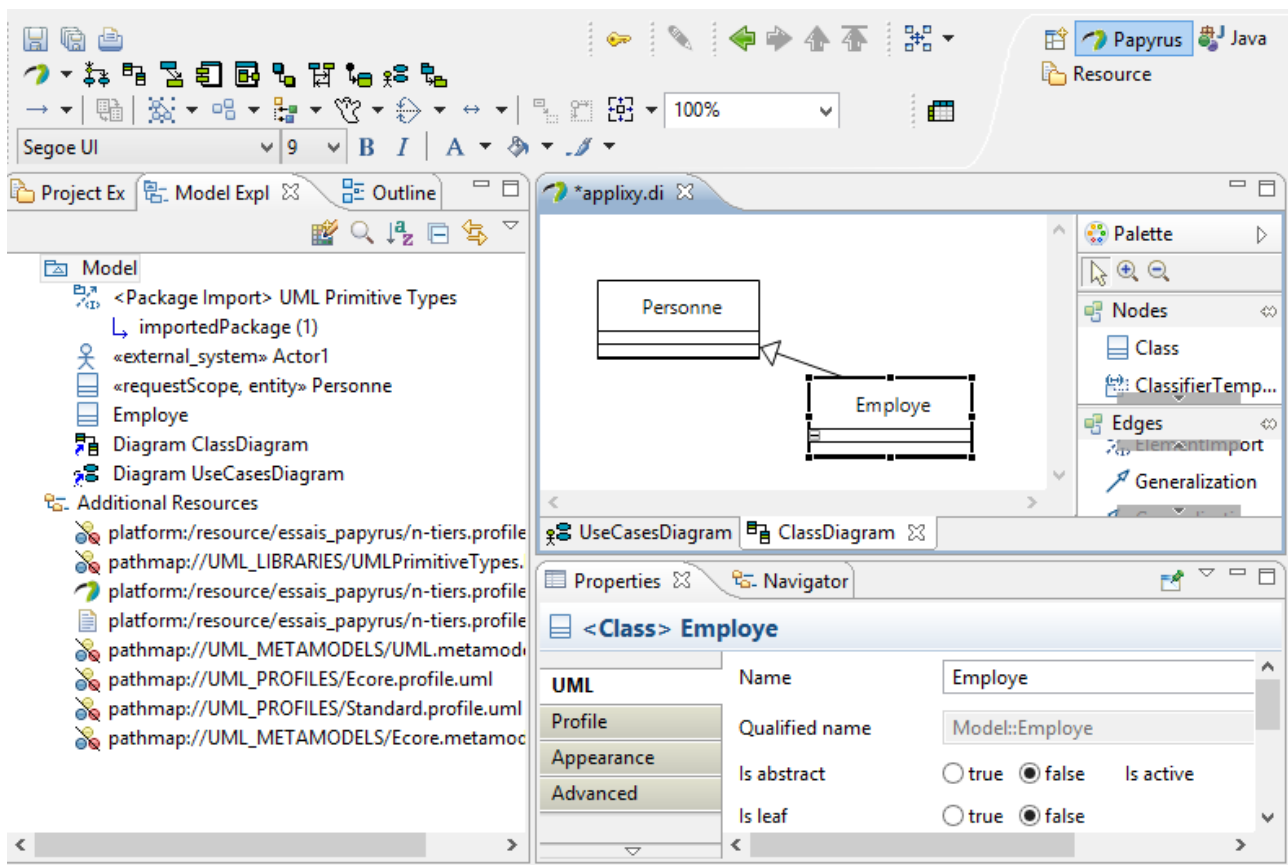


1.4. Utilisation générale de l'outil Papyrus UML

Chaque modèle UML est sauvegardé dans trois fichiers complémentaires:

- le fichier **".uml"** comporte tous les éléments significatifs du modèle UML (packages , classes ,). C'est à partir de ce fichier que l'on peut extraire les informations essentielles du modèle UML pour générer du code (via un outil MDA tel qu'accéléo_m2t par exemple)
- les fichiers **".notation"** et **".di"** comportent les coordonnées (x,y,...) des éléments internes des diagrammes UML.

==> pour éditer graphiquement un modèle UML, il faut ouvrir (via un double-clic) le fichier **".di"** (ou l'ensemble). Le fichier **".uml"** de même nom sera alors automatiquement pris en charge et mis à jour.



De façon intuitive, la vue "Model Explorer" permet de naviguer dans l'arborescence interne du

modèle UML et la vue "Properties" permet de fixer les propriétés de l'élément sélectionné. Une palette permet de choisir le type d'élément à ajouter au modèle.

- pas de menu contextuel "add attribute/property" ni "add operation/method" mais des éléments "**Property**" et "**Operation**" à récupérer dans la palette et à ajouter aux classes du diagramme.
- Souvent besoin de cacher (via **Filter/hide compartment** ou **Filter /**) certains éléments secondaires.


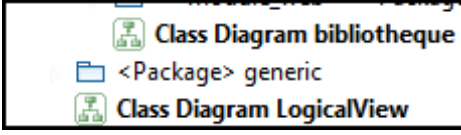
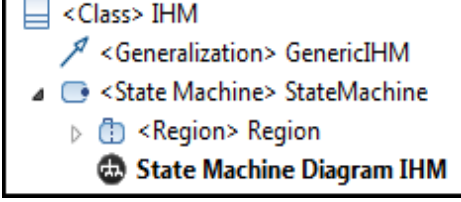
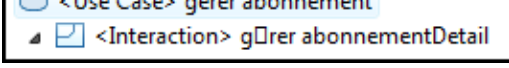
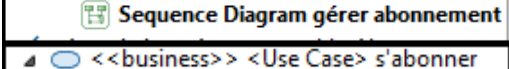
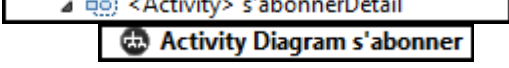
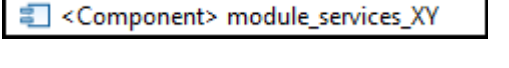
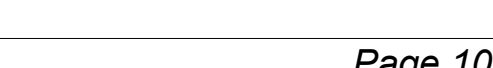
1.5. Organisation conseillée des packages et des diagrammes

L'un des principaux "**points forts**" des outils "**Topcased UML**" et "**Papyrus UML**" c'est de bien gérer la **cohérence** entre les **packages** et les **diagrammes** de façon à pouvoir naviguer de façon efficace dans l'arborescence d'un modèle UML.

Mode opératoire conseillé:

- Créer un élément UML (ex: package , classe, ...) dans un diagramme et le paramétrer.
- **Sélectionner un package existant (modélisé auparavant)** dans l'arborescence "**model explorer**", puis activer le menu "**add class diagram**" en donnant un nom explicite à ce diagramme (ex : packageXY_ClassDiagram), etc / etc
- Par la suite (une fois la création/association effectuée) , un **double-clic** ultérieur sur le package au sein du diagramme permettra de **naviguer d'un niveau vers un sous niveau** (de détails).

Sous diagrammes classiques pour indiquer les détails:

| Eléments du modèle UML | diagramme(s) classique(s) associé(s) pour les détails | Exemple(s) |
|------------------------|---|--|
| Package | Diagramme de classes (ou ...) |   |
| Classe | Diagramme d'états (state machine) ou diag. de structure composite |  |
| Use Case | diagramme d'activités et/ou diagrammes de séquences |     |
| Composant | Diagramme de (sous)composants ou de structure composite , ... |  |

Remarques importantes:

- Pour bien organiser un modèle UML, il faut réfléchir le plus tôt possible à la décomposition en différents packages.
- Il est toujours possible de renommer ou déplacer à la souris un package (après coup / par la suite) via le "model_explorer".
- Cette "bonne organisation" des éléments du modèle UML est surtout utile pour que des scripts des templates (.docx/.odt) pour gendoc2 puissent efficacement retrouver les diagrammes de façon à générer automatiquement une bonne documentation.

1.6. Edition d'un diagramme de classes (spécificités)

Après avoir sélectionné une association, on peut activer le menu contextuel "Filters/All-No-Managed connectors labels" ce qui fait apparaître la boîte de dialogue suivante :

Select the labels to display.

| Label Role | Displayed Text |
|--|----------------------|
| <input type="checkbox"/> / <Association> class1_class2_1 | class1_class2_1 |
| <input type="checkbox"/> Name | class1_class2_1 |
| <input type="checkbox"/> 0..1 SourceMultiplicity | |
| <input type="checkbox"/> SourceRole | + class1 |
| <input type="checkbox"/> Stereotype | [No Text To Display] |
| <input type="checkbox"/> 0..1 TargetMultiplicity | |
| <input type="checkbox"/> TargetRole | + class2 |

Ceci est très pratique pour afficher ou cacher certains détails associés aux associations (rôles ,)

Dans la fenêtre des propriétés:

***Paramétrer les détails d'une d'association (navigabilité, agrégation, composition,)**

Member End

Name:

Owner:

Navigable: ☐ true ☒ false

Aggregation:

Multiplicity:

Member End

Name:

Owner:

Navigable: ☒ true ☐ false

Aggregation:

Multiplicity:


```

classDiagram
    class Voiture {
    }
    class Roue {
    }
    Voiture "1" -- "4..4" Roue : voiture_roue_1
  
```


* Paramétrer une opération (nom, type de retour, paramètres ,)

| | | | |
|-------------|---|------------|---|
| Name | rechercherExemplaireParNumero | | |
| Is abstract | <input type="radio"/> true <input checked="" type="radio"/> false | Is leaf | <input type="radio"/> true <input checked="" type="radio"/> false |
| Is query | <input type="radio"/> true <input checked="" type="radio"/> false | Is static | <input type="radio"/> true <input checked="" type="radio"/> false |
| Concurrency | sequential | Visibility | public |
| Method | <div> <div>↑</div> <div>↓</div> <div>+</div> <div>×</div> <div>✎</div> </div> | | |
| | Owned parameter | | |
| | <div> <div>↑</div> <div>↓</div> <div>+</div> <div>×</div> <div>✎</div> </div> | | |
| | <div> <div>↔</div> <div><Parameter></div> <div>num : Integer</div> </div> | | |

Create a new Parameter

Name

num

Is exception

☐ true ☒ false

Is stream

☐ true ☒ false

Direction

in

Visibility

public

Default value

<Undefined>

+

✎

×

Type

<Prim...teger

...

+

✎

×

Is ordered

Is unique

Effect

Multiplicity

UML

Profile

pour un paramètre d'entrée , laisser direction="in", **pour la valeur de retour , direction = return**

Name

return

Is exception

☐ true ☒ false

Is stream

☐ true ☒ false

Direction

return

Visibility

public

Default value

<Undefined>

+

Type

<<entity>> <Class> Exemple

...

+

résultat →

```
rechercherExemplaireParNumero(num: Integer): Exemple
```

et (dans le sous onglet "Appearance"), décocher "param direction" , "visibility" et "modifiers":

Appearance

Advanced

Requirement

Label customization

☐ Parameters Direction

☒ Parameters Name

☒ Parameters Type

☐ Parameters Multiplicity

☒ Name

☐ Parameters Modifiers

☒ Visibility

☒ Return Type

☐ Parameters Default Value

☒ Modifiers

* Choix d'un (ou plusieurs) stéréotype(s) à appliquer:

Sélectionner un des éléments du modèle (classe ou propriété ou package ou)
et sélectionner un stéréotype via le sous onglet "profile" :

Properties Documentation

<<service>> <Class> GestionExemplaires

UML

Profile

Applied stereotypes:

service (from n-tiers)

Applicable Stereotypes:

| Stereotype | Information |
|---------------|------------------------|
| local | n-tiers::local |
| pageScope | n-tiers::pageScope |
| remote | n-tiers::remote |
| repository | n-tiers::repository |
| requestScope | n-tiers::requestScope |
| sessionScope | n-tiers::sessionScope |
| stateful | n-tiers::stateful |
| stateless | n-tiers::stateless |
| transactional | n-tiers::transactional |

Applied Stereotypes:

service

NB : Pour que certains stéréotypes applicables soient proposés, il faut qu'au préalable au moins un profile UML (autre fichier ".uml" comportant un paquet de stéréotypes) ait été associé à la racine du modèle via le sous onglet "profile" des propriétés.

* sélectionner un type de données pour une propriété ou le cacher :

Sélectionner une propriété de la classe,

Si l'on souhaite (en analyse) cacher le type encore indéfini , il faut alors décocher "type" dans le sous onglet "appearance"

Appearance

Advanced

Requirement

Label customization

☒ Type

☒ Multiplicity

☐ Default Value

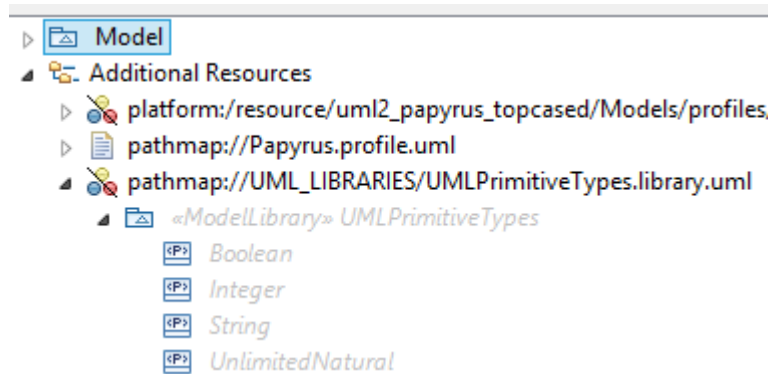
☒ Visibility

☒ Is Derived

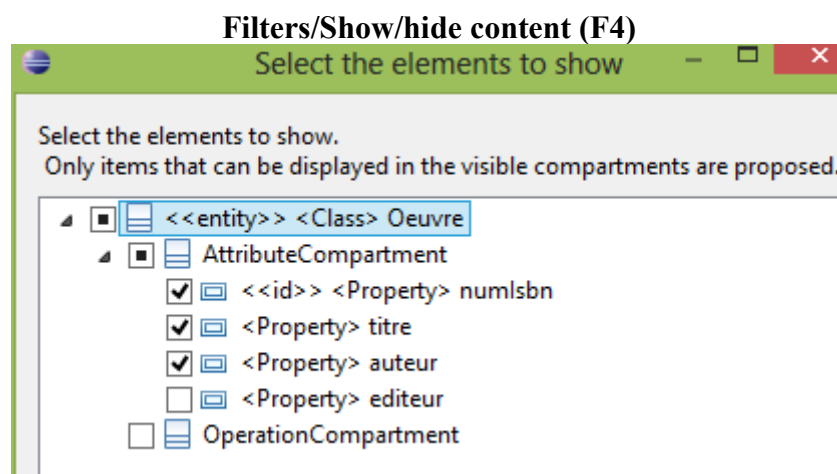
☒ Name

Si l'on souhaite (en conception détaillée), préciser un type de données parmi les types primitifs

prédéfinis d'UML, il faut depuis le sous onglet "UML" , le choisir via "..." en face "type :"



* Montrer ou cacher **Graphiquement** les différents éléments (propriétés/opérations) d'une classe:



* spécifier une **agrégation** ou une **composition** d'un coté d'une **association** :

Aggregation → pour obtenir un losange blanc sur l'extrémité inverse .

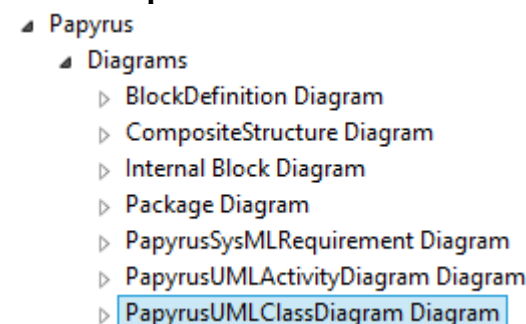
Aggregation → pour obtenir un losange noir sur l'extrémité inverse .

* spécifier une **classe d'association**

→ d'abord placer une association ordinaire et une classe ordinaire , relier ensuite par une liaison de type "AssociationClass" dans la sous palette "Edge" en partant de l'association et en pointant vers la classe . [Bug mi-2013 : les pointillés disparaissent lorsque l'on ferme et ré-ouvre le diagramme]

1.7. Préférences/options sur l'éditeur Papyrus UML

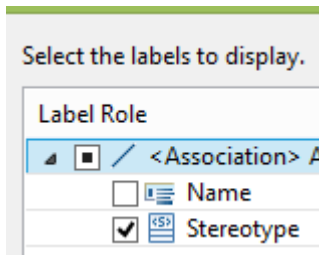
Windows/preferences/



1.8. Edition d'un diagramme de Use Cases (spécificités)

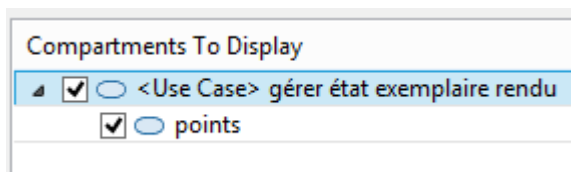
* **Montrer ou cacher les différents éléments d'une association sélectionnée:**

Filters / No Connector Label ou / Managed Connector Label



* **Montrer ou cacher un point d'extension (lié à un <<extend>>) :**

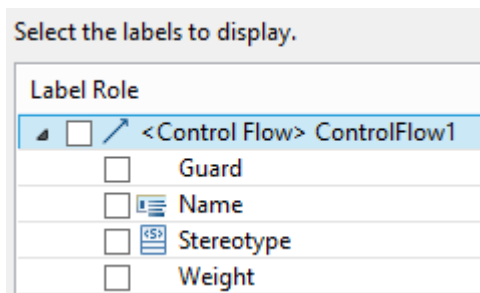
Filters/ Show/hide compartments



1.9. Edition d'un diagramme d'activités (spécificités)

* **Montrer ou cacher les détails d'une liaison (controlFlow) :**

Filters / No Connector Label ou / Managed Connector Label



* **Paramétrage des pins (output et input) autour d'un flot d'objet(s) :**

Placer un "object flow" entre deux actions via la palette.

Au moment de l'établissement de la liaison, la boîte de dialogue suivante apparaît alors pour paramétrer le nom et/ou le type des objets (données/document) qui seront véhiculés d'une activité à l'autre. A partir de ce paramétrage, l'éditeur "papyrus" va automatiquement construire des "pins" de même nature de chaque côté de la liaison.

Please fill information for pins creation

Pins initialization ?

Name: data

Type: [] ...

* **Précautions à prendre pour bien paramétrer les liaisons d'entrées et de sorties** autour d'un "losange de décision" ou d'un "fork" :

→ De façon à contourner certains bugs temporaires de papyrus, il vaut mieux bien définir la (ou les) entrée(s) avant de définir la (ou les) sortie(s).

* **Paramétrage d'un "call behavior action" pour naviguer d'un diagramme d'activité à un sous autre :**

Create a new Call Behavior Action

Create a new Behavior ?

☐ Create behavior

Behavior type: Activity -> Behavior

Name: Activity1

Element owner: <Activity> calcul , confirmation et paiement de l'acompte

Or assign an existing one

☒ Select behavior

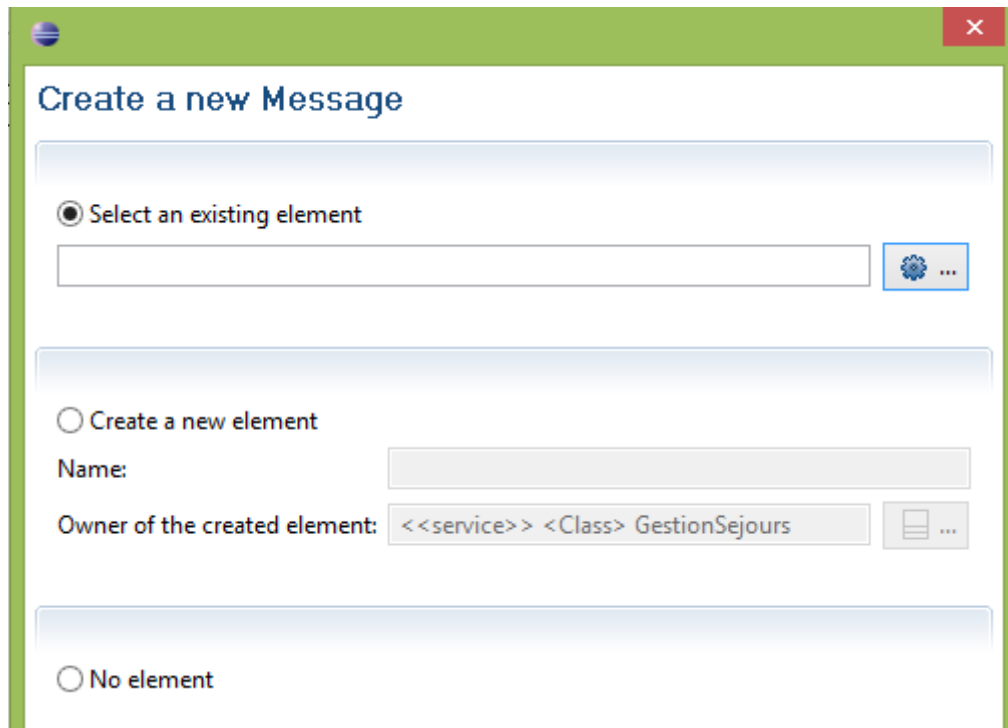
Behavior: [] ...

NB : de façon à bien contrôler la position du sous diagramme dans la hiérarchie du "model explorer" , on peut soit pré-crée le sous diagramme pour le sélectionner ensuite , soit bien paramétrer l'élément propriétaire ("owner") lors d'une création à la volée.

1.10. Edition d'un diagramme de séquences (spécificités)

Mode opératoire:

- créer un diagramme de séquence (idéalement en tant que détail d'un "use case")
- placer des **lignes de vie ("LifeLine")** et préciser les **types représentés** (acteurs, classes, ...) en effectuant des "glisser/poser" d'un élément (acteur ou classe) du "model explorer" vers l'entête du "LifeLine" et confirmer l'opération via un click sur "set represent ...".
- placer des **blocs d'exécution** sur les lignes de vie
- placer des **messages** entre un bloc d'exécution et un autre
- **paramétrer** les messages (*saisir un nom* ou bien *sélectionner une opération disponible au niveau de type d'objet qui reçoit le message*)













Remarque importante :

Lorsque (via l'option "create new element" de cette boîte de dialogue) l'on crée de nouvelles méthodes/opérations dans la classe de l'objet qui reçoit le message, celles-ci sont présentes dans le "model explorer" mais n'apparaissent pas automatiquement dans les diagrammes de classes. Pour faire graphiquement apparaître les nouvelles opérations dans les classes d'un diagramme de classes, il faut activer le menu contextuel "filters/ show/hide contents" et sélectionner les nouvelles méthodes (supplémentaires) à afficher.

NB: On peut également placer des fragments combinés (avec mot clef "alt", "opt", "loop", ...) d'UML2.

1.11. Edition d'un diagramme d'états (spécificités)

* Paramétrage interne d'un état (do , exit , entry) :

| | | | | |
|-----------------|--|---|-------|-------------|
| State invariant | <Undefined> |    | Entry | <Undefined> |
| Do activity |  <Opaque Behavior> choix période, transport |    | Exit | <Undefined> |
| Submachine | <Undefined> |    | | |

1.12. Edition d'un diagramme de composants (spécificités)

RAS

1.13. Edition d'un diagramme de déploiement (spécificités)

RAS

1.14. Génération de code java (via le générateur par défaut de Topcased)

- Ouvrir (si besoin) la vue "Navigator"
- Sélectionner le fichier appXY.uml et activer le menu contextuel "Code Generator / generateJava".

→ le code généré apparaît alors dans le projet courant .

2. Génération de documentation (gendoc2)

Gendoc2 est un **plugin eclipse** permettant de *générer de la documentation* (au format ".docx" de word ou bien ".odt" de OpenOffice) *à partir des informations extraites dans un modèle UML* (".uml" et ".notation" , ".di").

NB: à partir des formats ".odt" ou ".docx" , il est assez facile de **générer une version ".pdf"** .

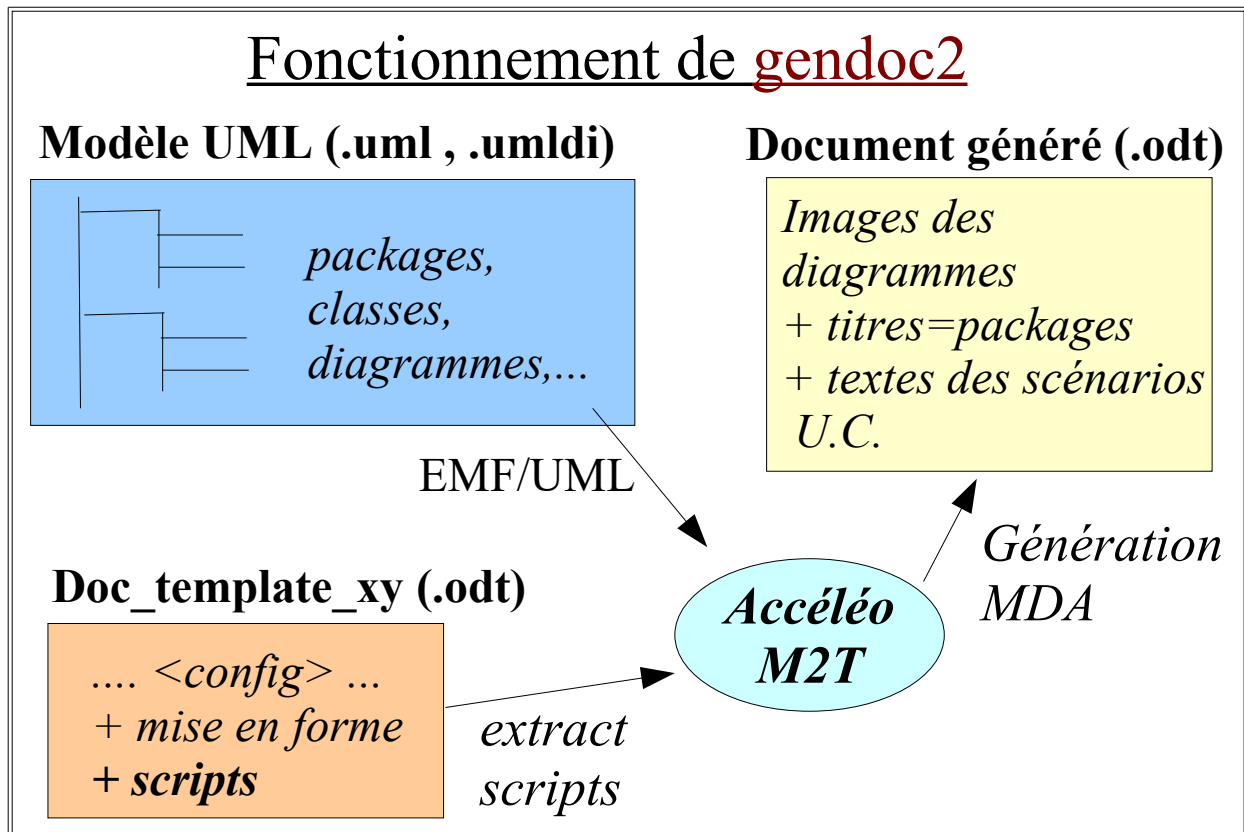
Gendoc2 est déjà **intégré à Topcased_RCP >=5** et prêt à l'emploi.

L'ancienne version "gendoc" utilisait en interne une ancienne version du générateur MDA "accéléo"
La nouvelle version "gendoc2" utilise en interne la nouvelle version 3 d'accéléo (accéléo_M2T).

NB: En combinant des *fichiers générés par gendoc2* avec des *fichiers statiques* , on peut assez rapidement produire des **spécifications** assez complètes de bonnes qualités et toujours cohérentes

avec les dernières versions des modèles .

2.1. Principe de fonctionnement de gendoc2



NB: Le déclenchement du processus de génération de documentation s'effectue simplement en:

- se plaçant sur un fichier modèle "doc_template.odt" (ou bien .docx)
- activant le menu contextuel "Generate Documentation".

NB2 : Les exemples de configurations qui suivent sont adaptés à l'éditeur "papyrus".

2.2. Paramétrages généraux (configuration, contexte(s))

Un fichier modèle de documentation à générer (doc_template) doit comporter (généralement dès le début) un **bloc de configuration XML** qui sera pris en compte par gendoc2 et qui ne sera pas affiché au sein de la documentation produite.

Ce bloc de configuration sert essentiellement à préciser les **chemins d'accès** nécessaires pour localiser le modèle UML , le "template" initial et la documentation à générer.

Syntaxe et exemple:

```

<config>
<param key='workspace' value='c:\tp\tp-uml\my-topcased-uml-wksp' />
<param key='project' value='${workspace}\uml2_papyrus_topcased' />
<param key='appName' value='bibliotheque' />
<param key='model' value='${project}\Models\applications\${appName}\${appName}.uml' />
<output path='${project}/Documentation/${appName}/Generated/exprBesoins.odt' />
</config>
<context model='${model}' importedBundles='gmf;papyrus' searchMetamodels='true'/>
  
```


Ensuite , dans le reste du fichier modèle à générer, on pourra trouver un ou plusieurs blocs (éventuellement complémentaires) de type `<context>` pour préciser des **chemins internes au modèle UML** qui seront considérés comme des **bases** (ou points de départs) de l'**extraction d'informations UML via des scripts**.

```
...
<context element='Model/UseCaseView' /><gendoc>
.... script basé sur Model/UseCaseView
</gendoc>

<context element='Model/LogicalView' /><gendoc>
.... script basé sur Model/LogicalView
</gendoc>
...
```

2.3. Généralités sur les scripts de gendoc2

Un script pour gendoc2 est encadré par la balise XML `<gendoc>....</gendoc>`

Il comporte des instructions entre [] qui seront interprétées par accéléo M2T .

Ces instructions entre [] servent essentiellement à :

- boucler sur les éléments internes du modèle UML
- filtrer les éléments recherchés selon divers critères (types, ...)
- effectuer des opérations de mise en forme (concaténation, ...)
- ...

Des sous (sous) boucles de type ([for] ... [/for] imbriqués) sont possibles et assez fréquentes.

Syntaxe fondamentale: [for (nomVar :TypeElementUML | surQuoiOnBoucle)] [nomVar/] [/for]

Attention à ne pas placer trop d'élément de type "espace" ou "saut de ligne" car ceux-ci seront répétés en boucle lors de la génération de documentation

Cette contrainte explique pourquoi les fermetures des instructions ne sont pas souvent placées de façon symétrique par rapport aux ouvertures (décalages fréquents dans l'indentation).

Exemple:

```
<context element='Model/UseCaseView' />
Expression des besoins fonctionnels (Uses Cases) ici en texte caché (open office ou word)
<gendoc>
[for (uc:UseCase|self.ownedElement->filter(UseCase))]
U.C. "[uc.name/]"
```

```
[for(ligne:String|uc.getDocumentation().splitNewLine())][ligne /]
[/for]
```

```
[for (a:Activity|self.ownedElement->filter(Activity) )]
[for(subActivitydiag : Diagram | a.getPapyrusDiagrams())]
```

```
<image object='[subActivitydiag.getDiagram() /]' keepW='true'>
```



```
</image>[/for][/for]
```

```
[/for]</gendoc>
```

2.4. Scripts avec images/diagrammes

```
<gendoc>
```

```
[for(diag : Diagram | self.getPapyrusDiagrams())]
```

```
<image object='[diag.getDiagram() /]' keepW='true'>
```

*... Ce bloc normalement vide ... généré avec "insertion/cadre" de openOffice ...
 ... est obligatoire
 ... et sert à délimiter la surface de l'image qui sera issue d'un diagramme UML
 ... cette délimitation tiendra compte des paramètres keepH et keepW de <image >*

```
</image>[/for]
```

```
[for (p:Package|self.ownedElement->filter(Package))]
```

```
[p.name/]
```

```
[for(ligne:String|p.getDocumentation().splitNewLine())][ligne /][[/for]
```

```
[for(subdiag : Diagram | p.getPapyrusDiagrams())]
```

```
<image object='[subdiag.getDiagram() /]' keepW='true'>
```



```
</image>[/for][/for]
```

```
[/for]</gendoc>
```

2.5. Document maître pour fédérer plusieurs fichiers générés

On peut éventuellement utiliser un fichier "*spécifications_fonctionnelles.odm*" au format "document maître de OpenOffice" pour fédérer:

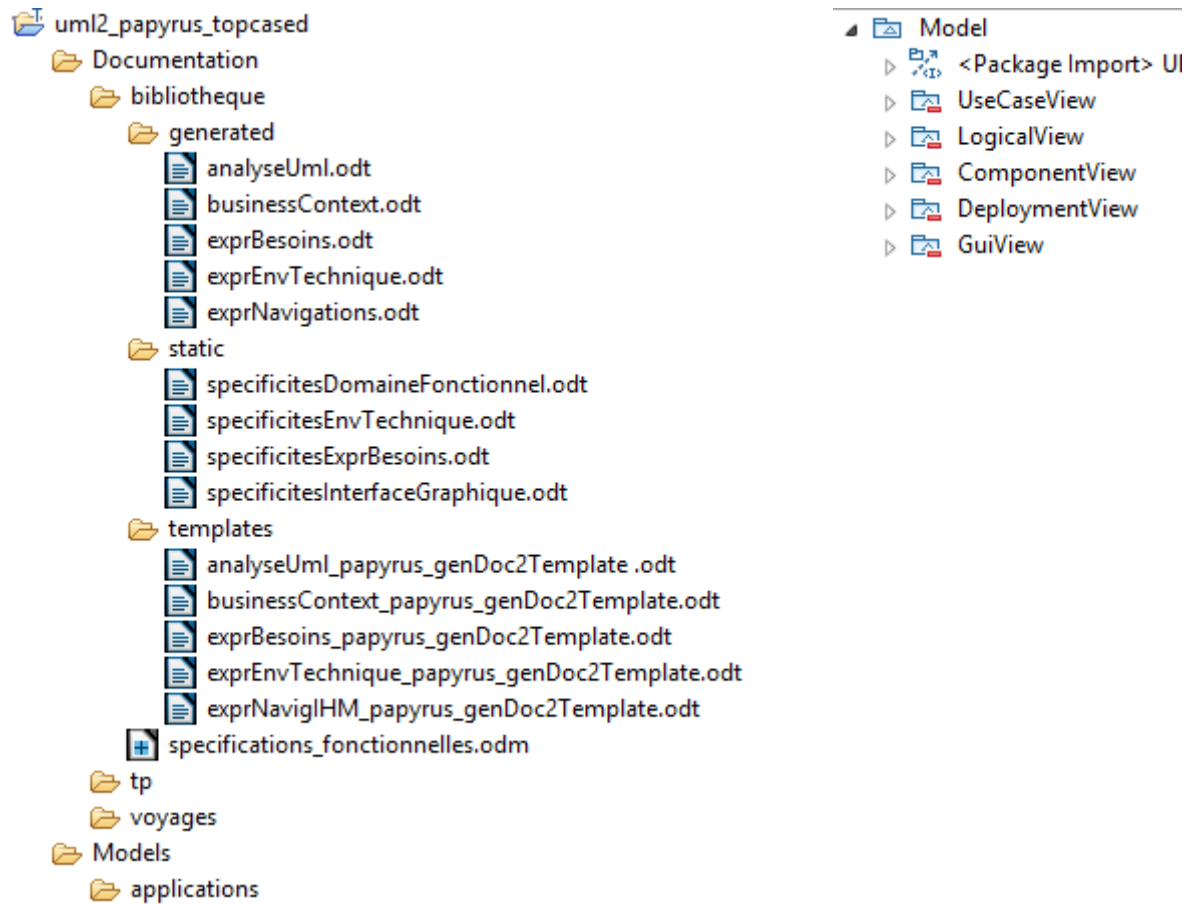
- des fichiers issus d'une génération automatique "UML/gendoc2"
- des fichiers éditer manuellement avec des contenus très spécifiques
- ...

Ceci permet en outre :

- d'obtenir une bonne numérotation des chapitres
- de construire facilement une table des matières globale
- de facilement exporter le tout au format pdf.

Exemple d'organisation de la documentation :

* analyseUml_....Template génère la doc structurée (diag classes , packages , diagrammes d'états et diag. séquences) ---- à partir de LogicalView et à partir des séquences attachées aux UseCase(View)



* exprBesoins.....Template génère l'expression des besoins (uses cases + scénarios + diag d'activités) à partir de UseCasesView

* exprEnvTechnique..Template génère le diagramme de déploiement à partir de DeploymentView

* businessContext...Template génère businessContext à partir du modèle annexe "context_XXX.uml"

* exprNavigIHM.....Template génère des diagrammes sur l'IHM (structure + navigations) depuis la partie GuiView