Ligne de Produits Logiciel et Variabilité des modèles

Module IDM de 5ème année

Philippe Lahire

Pourquoi ce cours?

- Transposition du développement industriel au logiciel
- Une famille *vs* une application
- Besoins industriels :
  - Nokia
  - Airbus
  - …

Sommaire

- Concepts et définition
- Inventaire des besoins (expressivité)
- Exprimer la variabilité
- Analyse des besoins : un standard de facto
- Où introduire la variabilité
  - Métamodèle UML + variabilité
  - Un modèle EMOF + variabilité
- Démonstration d'un outil

  LdP: Feature Model + produit dérivé : Modèle EMOF

Ligne de produits Logiciel (LdP)

- **LdP** : Ensemble de systèmes partageant un ensemble de propriétés communes et satisfaisant des besoins spécifiques pour un domaine particulier

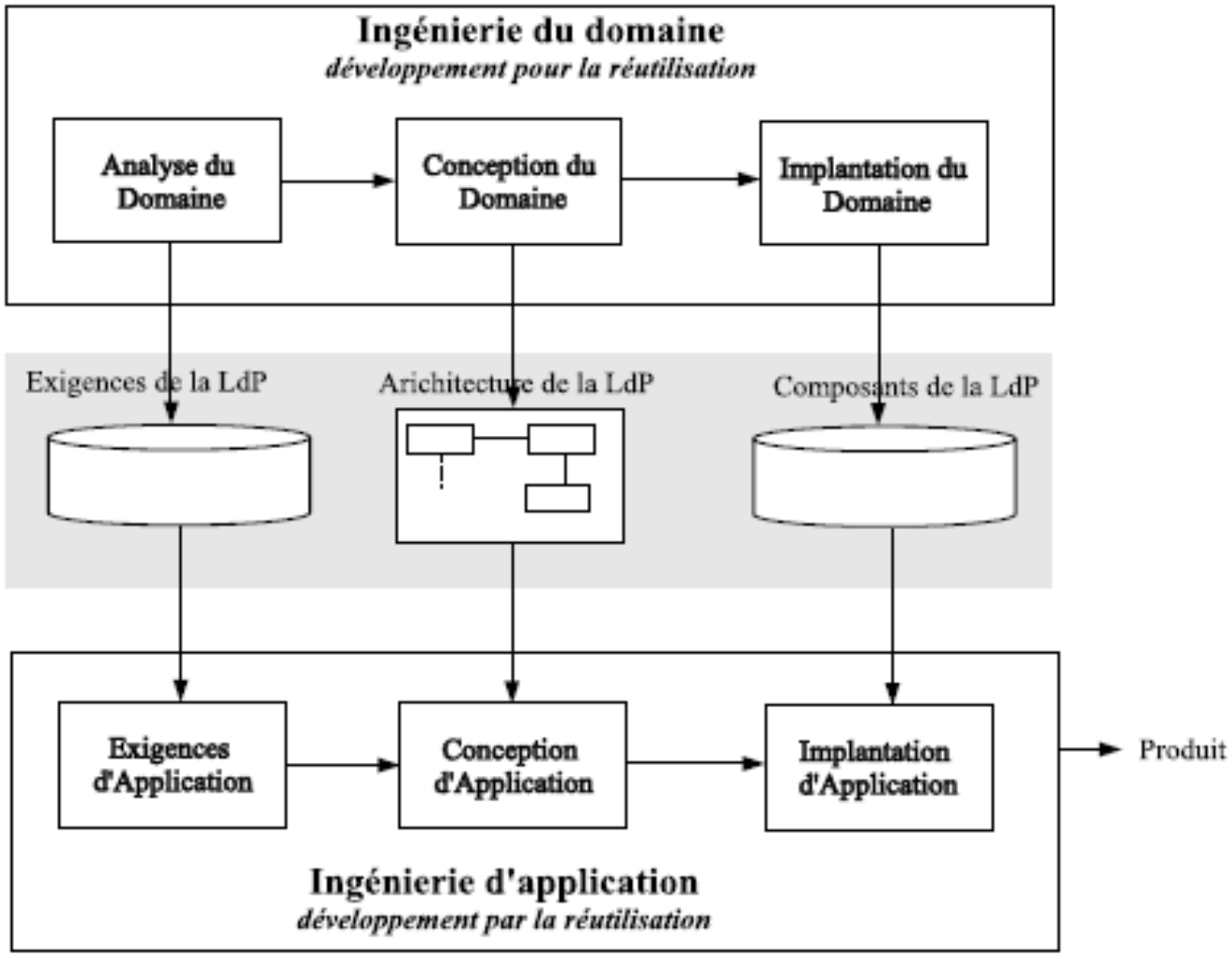
- **Domaine** : un secteur de métier ou de technologies ou de connaissances caractérisés par un ensemble de concepts et de terminologies compréhensibles par les utilisateurs de ce secteur.

Variabilité

- La variabilité regroupe l'ensemble des hypothèses montrant comment les produits, membres de la ligne de produits diffèrent

- La commonalité regroupe l'ensemble des hypothèses qui sont vraies pour tous les produits membres de la ligne de produits.

Ingénierie de domaine et d'application



Ingénierie de domaine et d'application

- Ingénierie de domaine
  - Quels sont les concepts du domaine
  - Les utilisations possibles de ces concepts

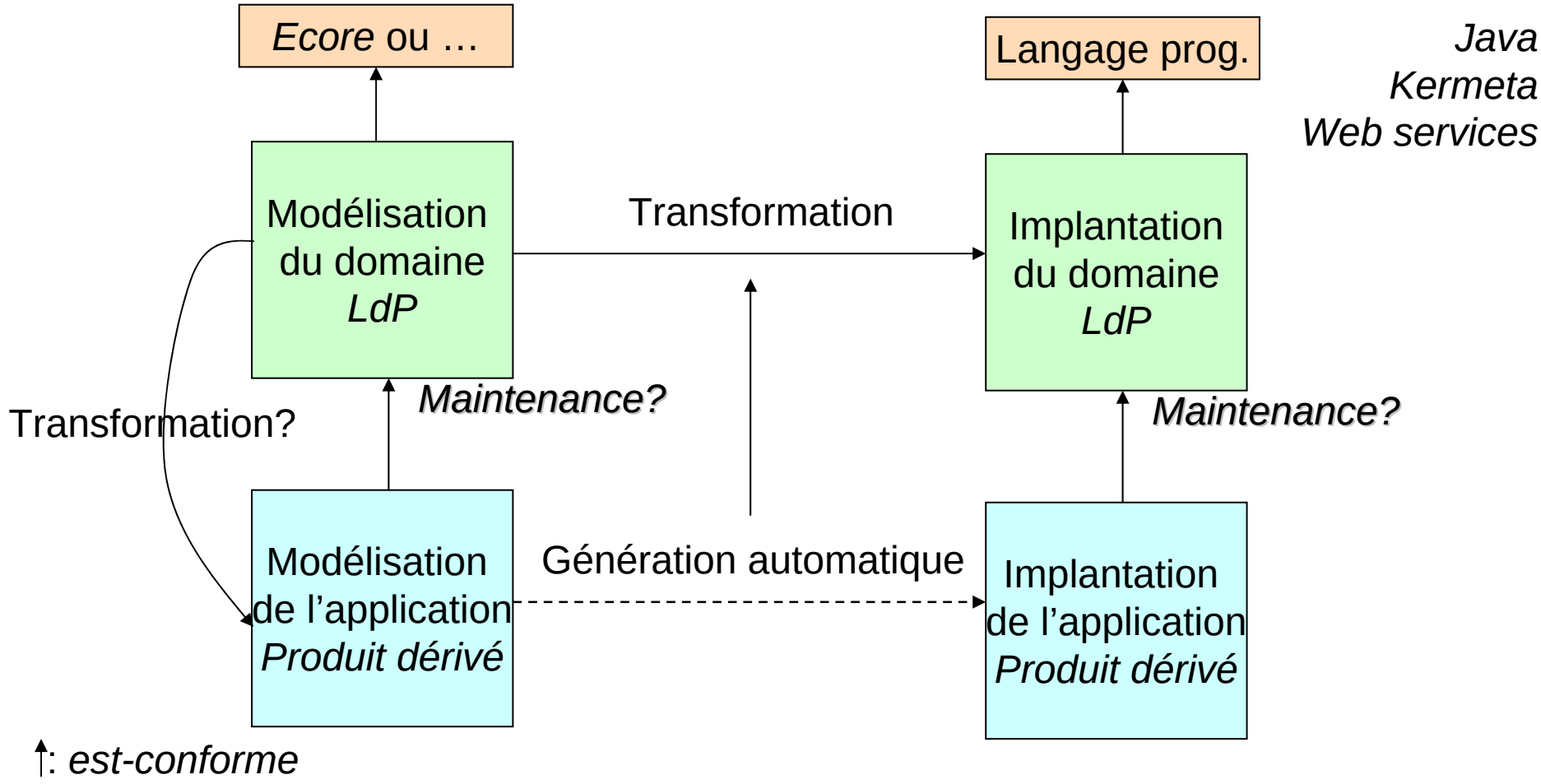
    Les uns par rapport aux autres
  - Implantation de ces concepts

- Ingénierie d'application
  - Mise en œuvre d'un produit issu du domaine
  - Passage par les mêmes étapes

Passage de la modélisation au code applicatif : approche IDM

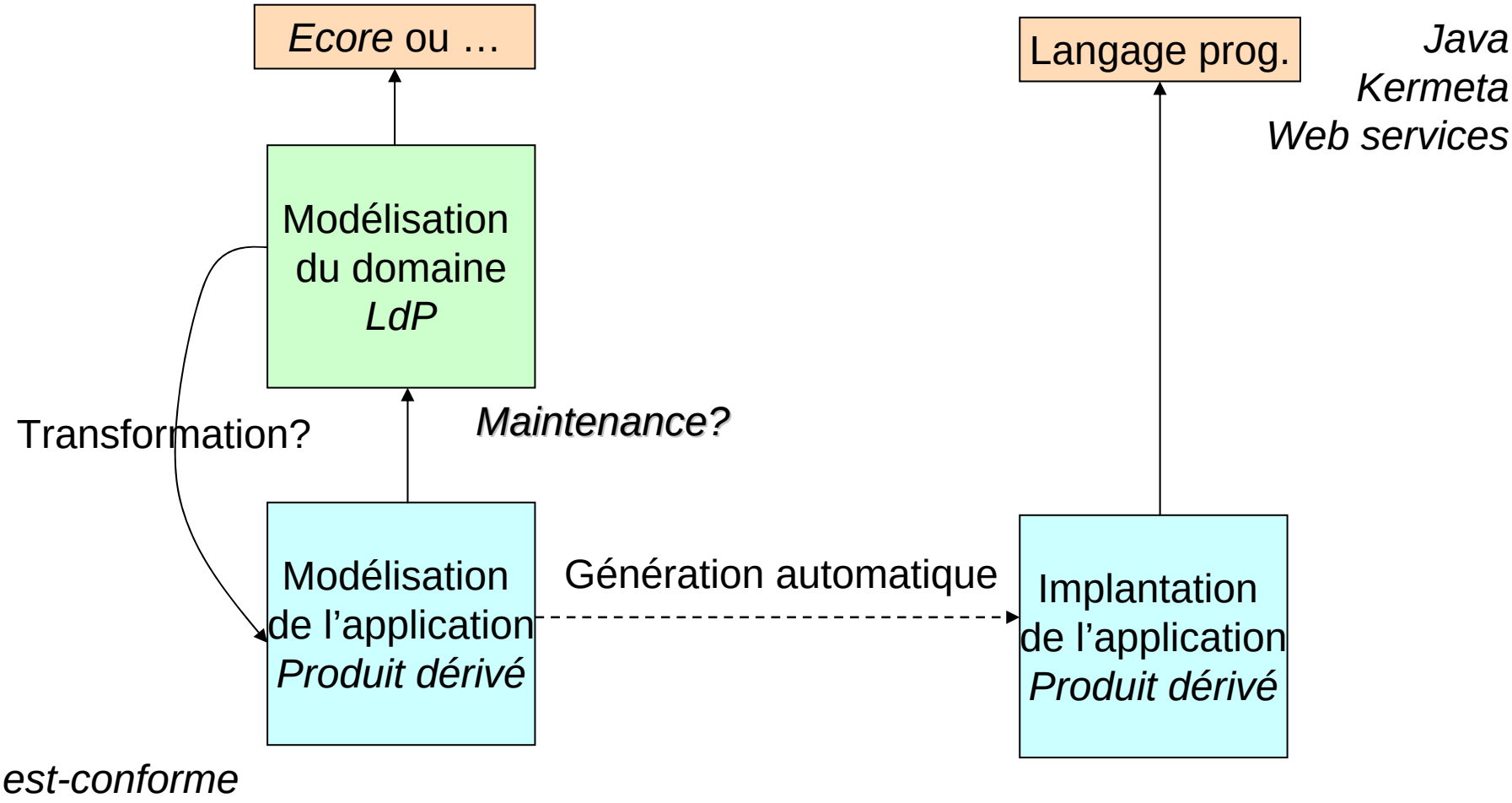
Approches (1)

Structure + comportement



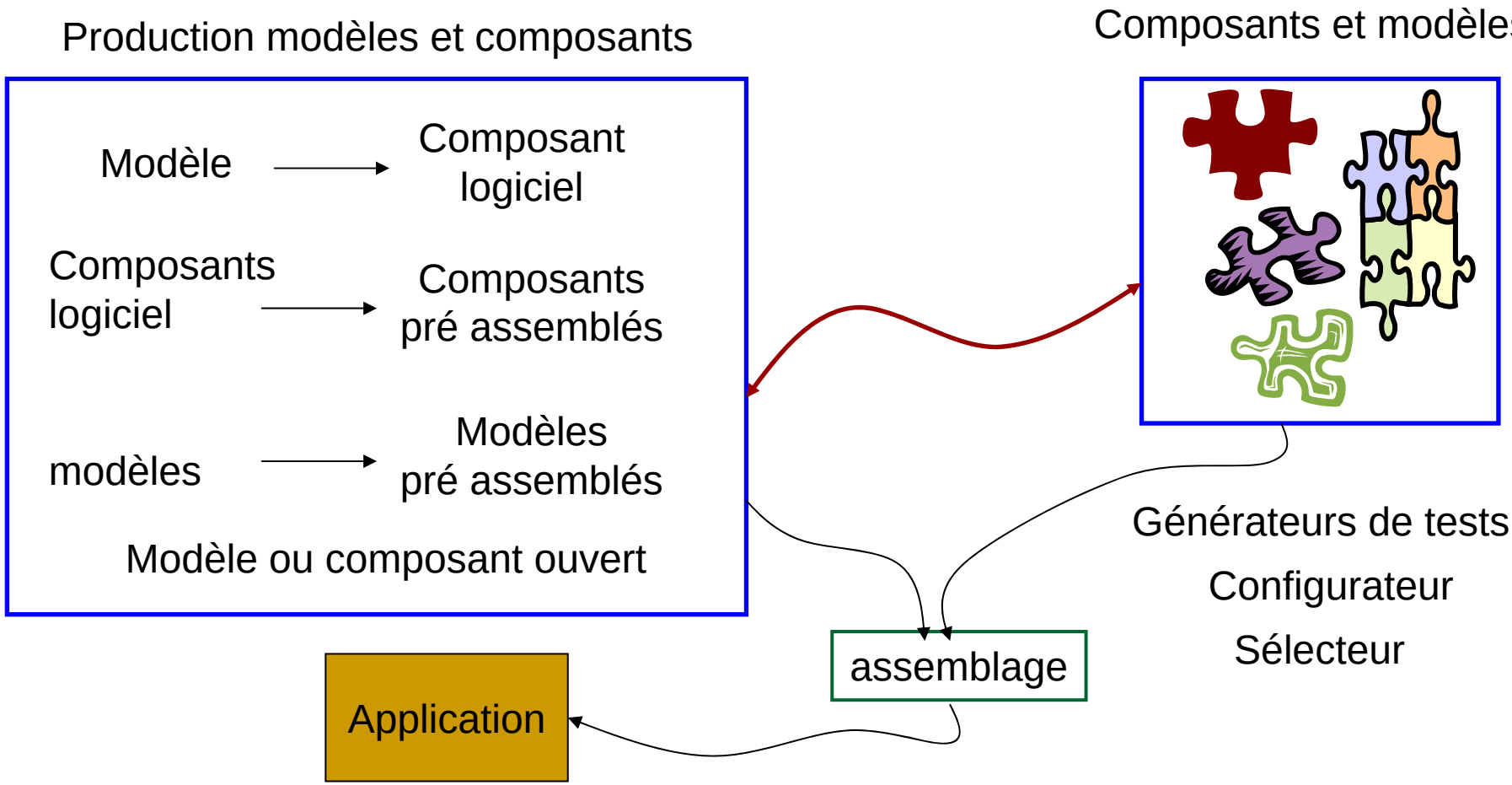
Approches (2)

Structure + comportement



Fabrique logicielle

Vers la production automatique de logiciel



Inventaire des besoins (1)

- Définir des propriétés et des fonctionnalités obligatoires
- Choix facultatifs: *0*,*1*…*n* choix parmi n
- Variantes
- Contraintes
  - de dépendance
  - Contraintes d’exclusion mutuelle

Inventaire des besoins (2)

Les propriétés communes

- Une voiture est constituée d'une carrosserie et de quatre roues
- Une personne a un nom et un prénom
- Une classe a un nom
- Une fonction a un type de retour
- Un attribut a un type
- …

Les variantes

- Une voiture est constituée d'une carosserie et de quatre roues (classique, sportive, etc.)
- Une personne a des enfants qui peuvent être des garçons ou des filles
- Un ordinateur peut être équipé d'une deuxième batterie ou d'un lecteur de DVD
- Une classe peut être concrète ou abstraite
- Une méthode peut être une procédure ou une fonction
- La partie structure d'une application peut être décrit par un programme ou un modèle
- …

Les caractéristiques facultatives

- Une voiture peut avoir un toit ouvrant
- Une personne peut avoir 0,1 ou plusieurs enfants
- Un ordinateur peut être équipé d'une deuxième batterie
- Une méthode peut être abstraite
- Une méthode peut avoir des paramètres
- Une méthode peut avoir 0 ou 1 type de retour
- Une classe peut avoir 0, 1 ou plusieurs parents
- La modélisation d'une application UML peut être constitué d'un diagramme de séquences
- …

Les dépendances

- Une voiture qui a un toit ouvrant doit avoir un intérieur cuir
- Une personne a des enfants seulement si elle est mariée
- Un ordinateur peut être équipé d'une deuxième batterie mais alors elle n'a pas de lecteur de DVD
- Un ordinateur peut avoir un processeur de 2,5 g-htz si elle a une batterie à 8 Lion
- Une méthode abstraite doit se trouver dans une classe abstraite
- Un diagramme de séquences repose aussi sur un diagramme de classes
- La présence d'un type de retour dans une méthode exclue qu'elle soit une procédure

Variabilité et cycle de vie du logiciel

- Dans le formalisme utilisé
  - Analyse des besoins (FODA…)
  - Conception (MOF, UML…) *ou les deux*
  - Implémentation (langages à objets, aspects…)
- Concerne :
  - Partie structurelle
  - Comportement
  - Contraintes

  $\Rightarrow$ aspects fonctionnels et extra-fonctionnels

Exprimer la variabilité

- Héritage (modélisation, implémentation)
- Généricité (modélisation, implémentation)
- Programmation par aspects (implémentation)
- Transformation de modèles (modélisation)
- Composition de modèles (modélisation)
- Feature Models (analyse des besoins)
- …

Exprimer la variabilité

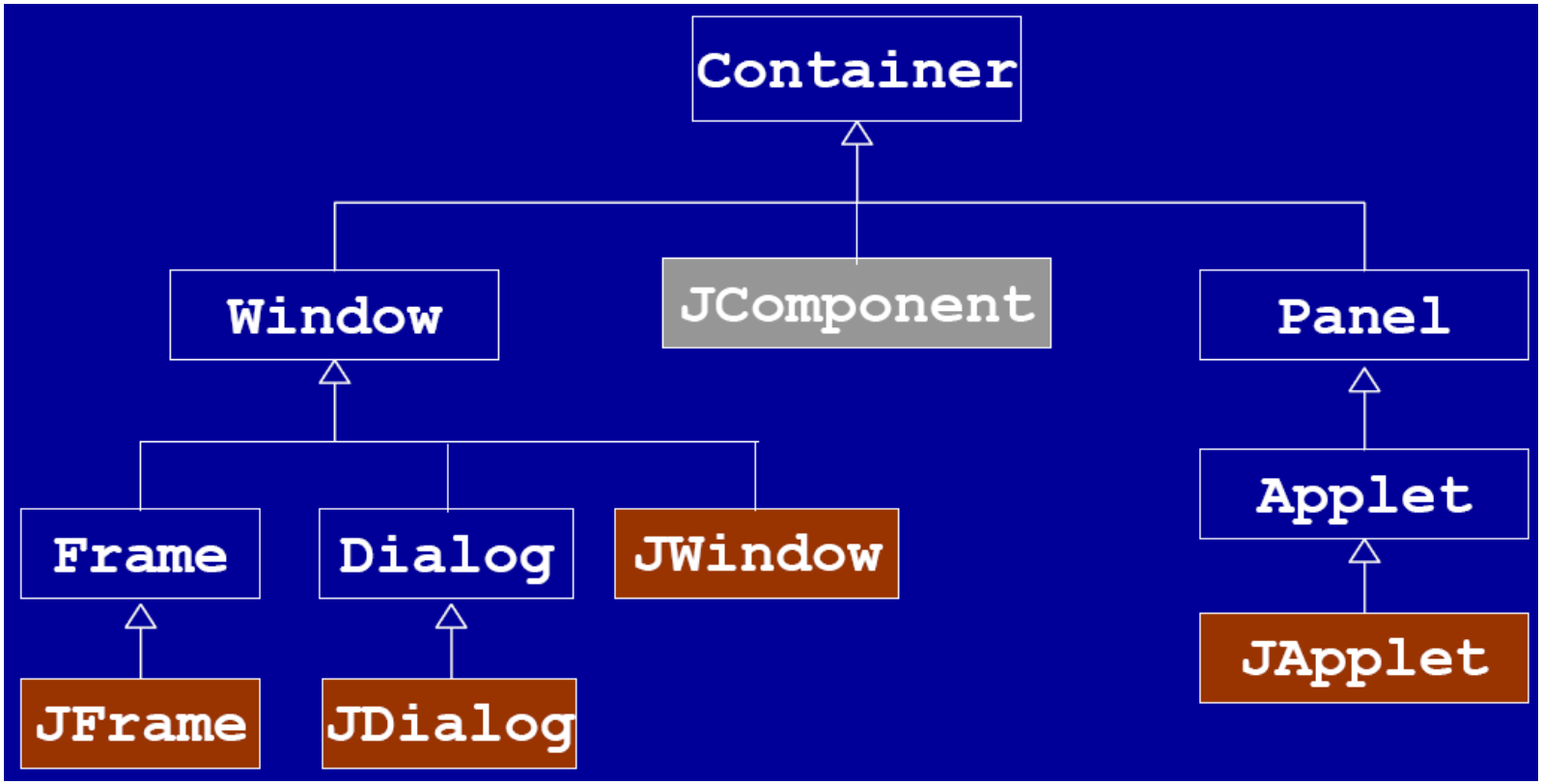
- *Héritage*
- Généricité
- Programmation par aspects
- Transformation de modèles
- Composition de modèles
- Feature Models
- …

Approche objet et Héritage

- Dédié à la définition des points communs

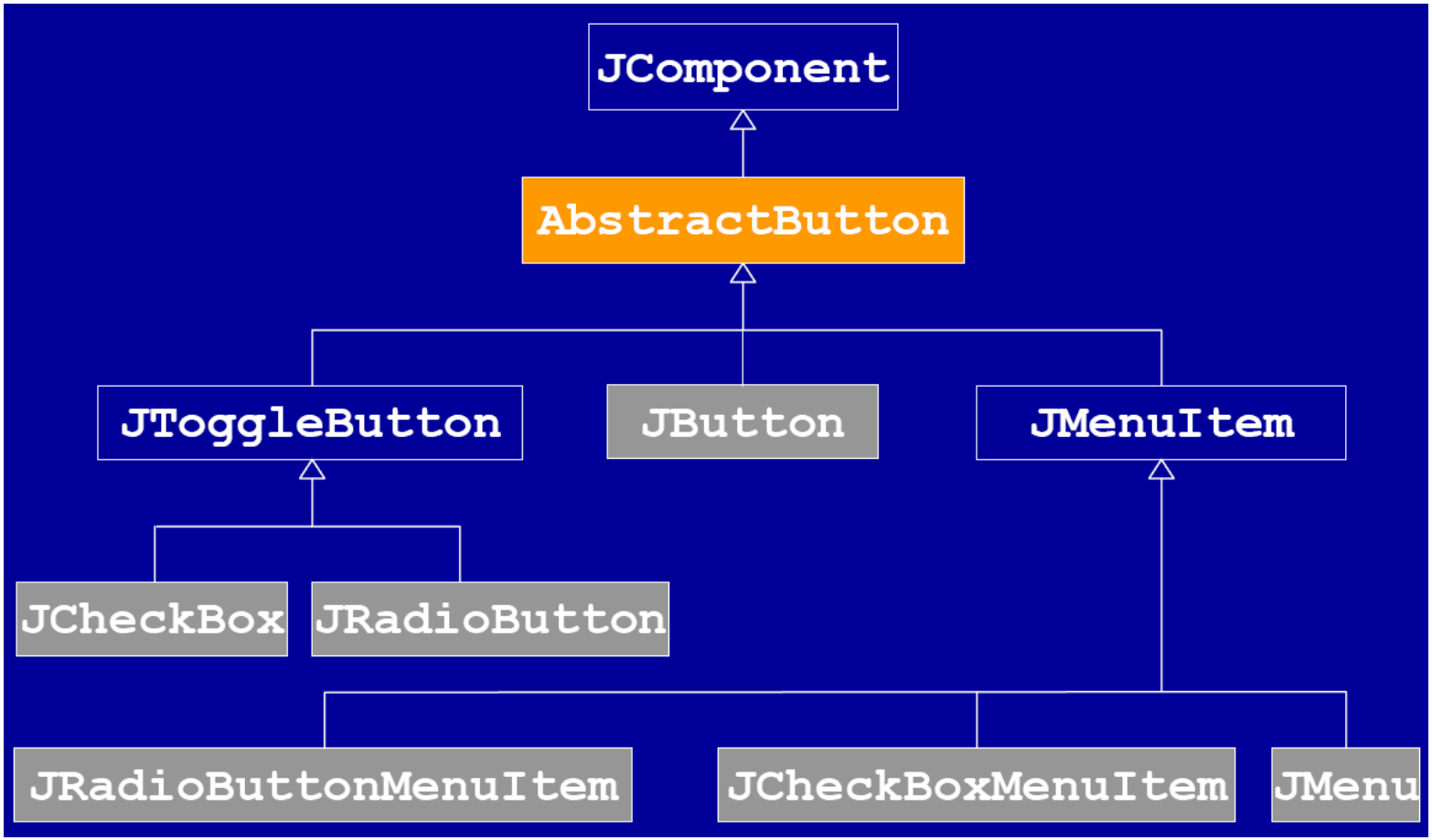
  Factorisation dans un ancêtre
- Expressivité pour la variabilité :
  - Variabilité du contenu (*redéfinition*)
  - Faible/pas de variabilité de la signature (*covariance ou non-variance*)
  - Ajout de propriétés et de méthodes (*descendants*)
  - Plusieurs variantes possibles (*plusieurs sous-arbres*)
  - Très lourd si combinaison de variantes
- Mise en œuvre: compilateur + + héritage + exécutif

Héritage



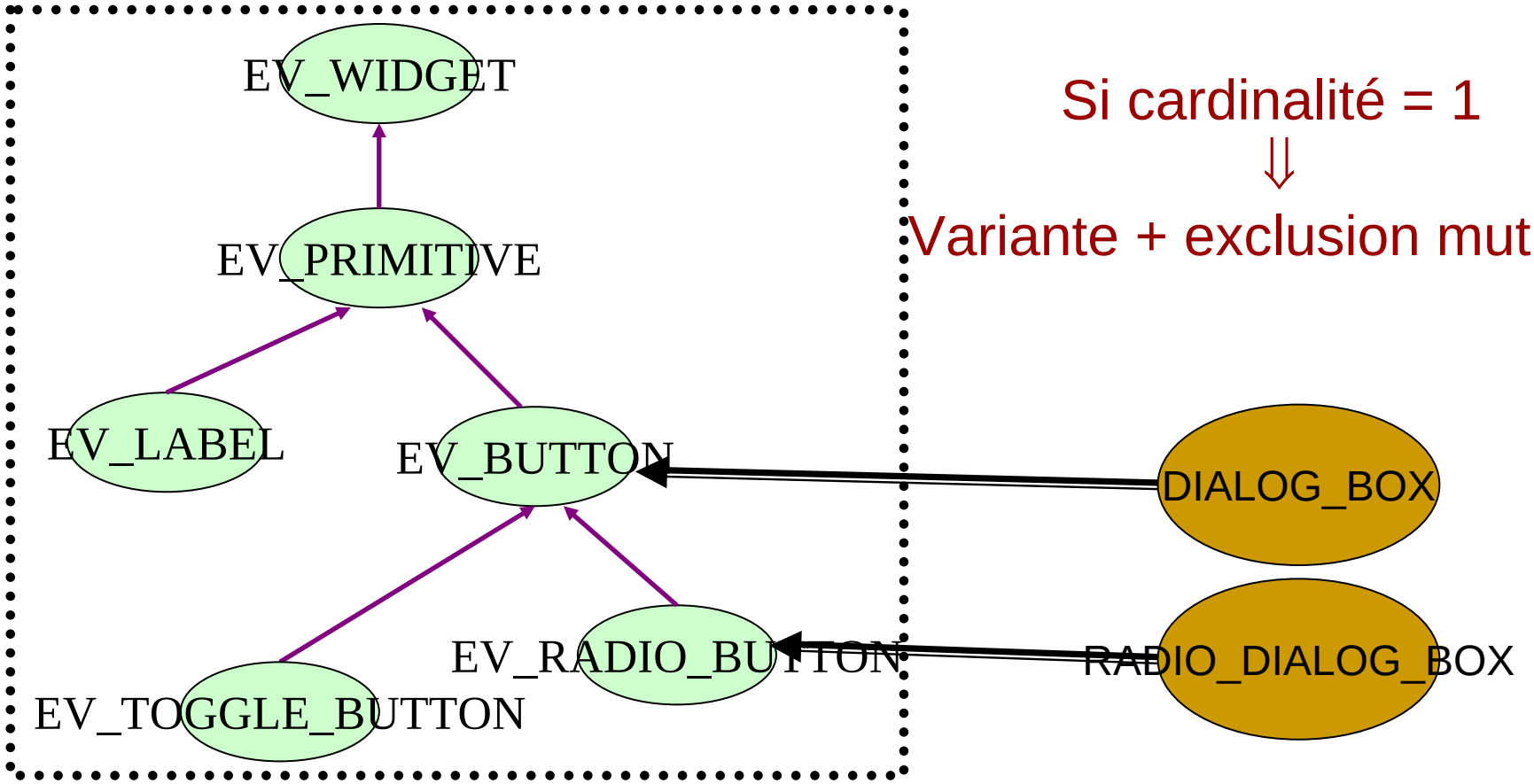
Domaine : objets graphiques
Famille de conteneurs

Héritage



Domaine : objets graphiques
Famille de Boutons

Proposer des variantes



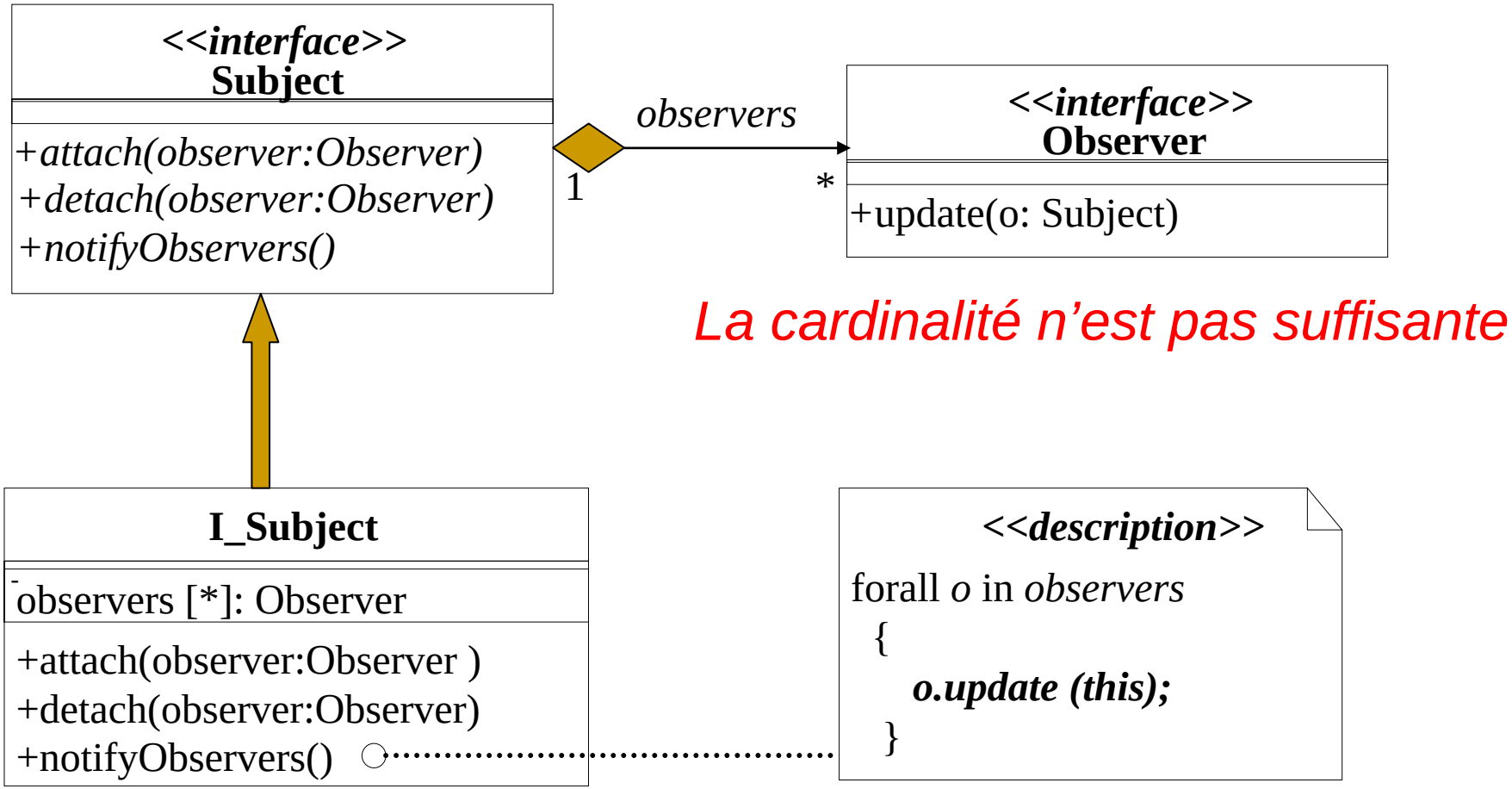
Si cardinalité = 1

⇓

Variante + exclusion mutuelle

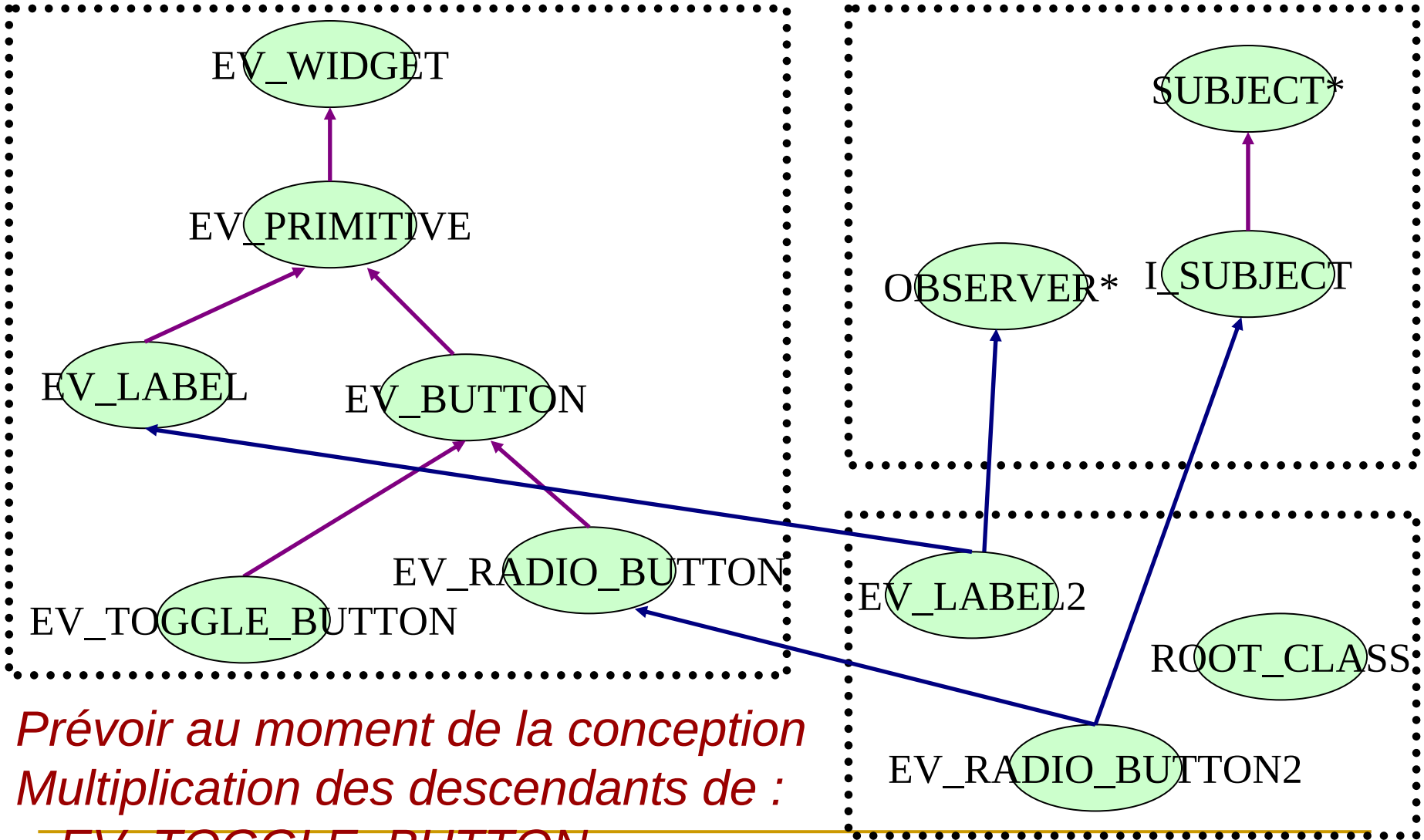
- Variantes simples par polymorphisme
- Faible support pour les variantes complexes ou impossible

Disposer de fonctionnalités facultatives



Ex: patron de conception Observeur

Disposer de fonctionnalités facultatives



- *Prévoir au moment de la conception*
- *Multiplication des descendants de :*
  - *EV_TOGGLE_BUTTON*
  - *EV_RADIO_BUTTON*

Exprimer la variabilité

- Héritage
- *Généricité*
- Programmation par aspects
- Transformation de modèles
- Composition de modèles
- Feature Models
- …

Classes génériques

- Bien adapté aux structures de données
  - Famille de listes : listes de personnes, listes de fenêtre…
- Support naturel de la notion de famille de type
- Les paramètres génériques définissent la variation
- Permet les variantes pour des types sans filiation
- Possibilité de contrainte (généricité contrainte)
  - Utilise des variantes basées sur l'héritage
- Les propriétés facultatives : idem précédemment

⇨Un complément de l'héritage

Exprimer des variantes

class Liste <P>

+get_element : P

+add_element (e: P)

+goto_Element (e: P)

+remove (e: P)

+forth

Description d'une famille : les listes de qq chose

Liste <Window>
Liste <Person>
Liste<Class>

class Liste <P -> Window>

+get_element : P

+add_element (e: P)

+goto_Element (e: P)

+remove (e: P)

+forth

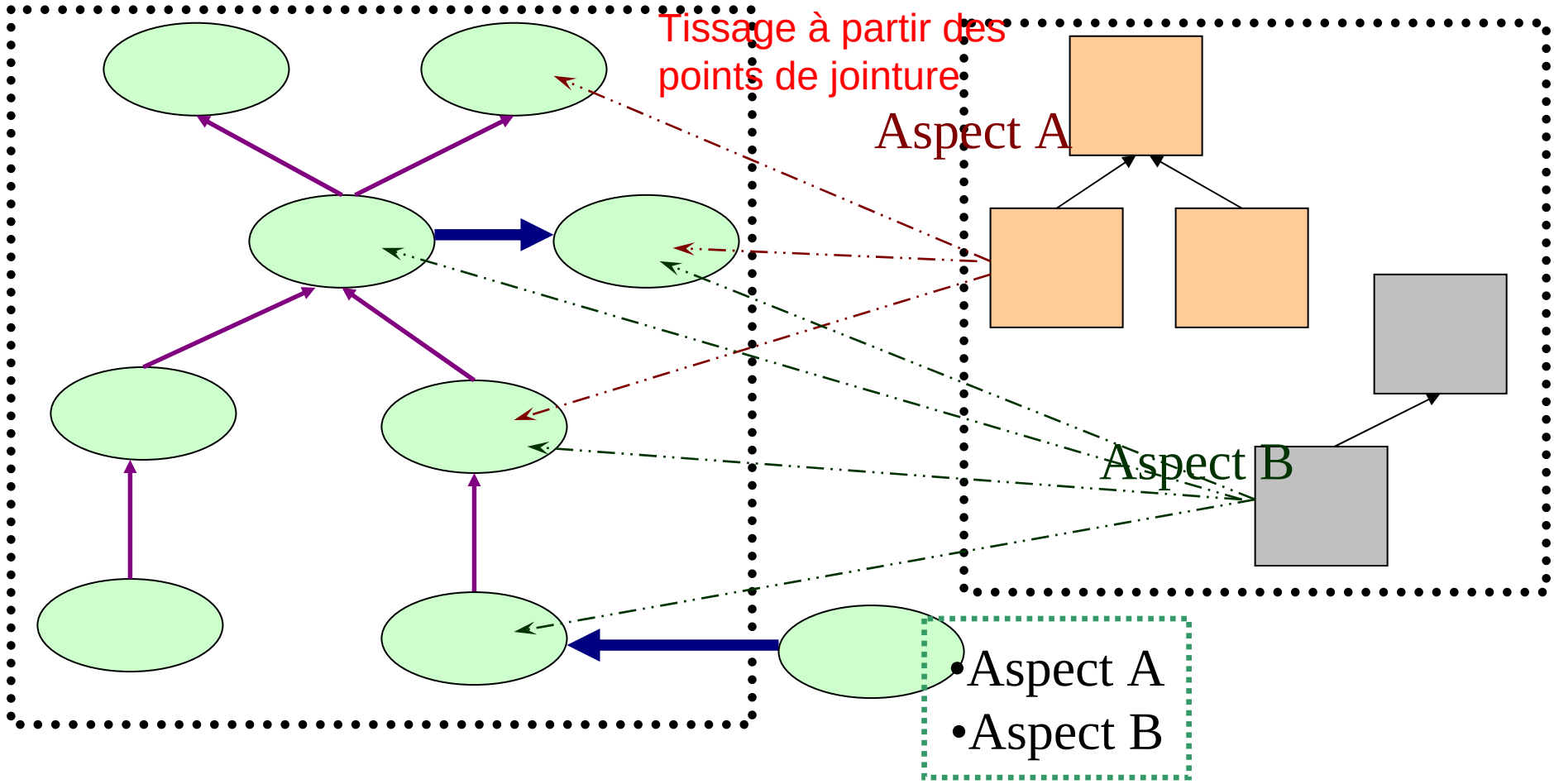
Description d'une famille : les listes de *window*

Mise en œuvre : Un compilateur (généricité)

Exprimer la variabilité

- Héritage
- Généricité
- *Programmation par aspects*
- Transformation de modèles
- Composition de modèles
- Feature Models
- …

Modélisation par aspects

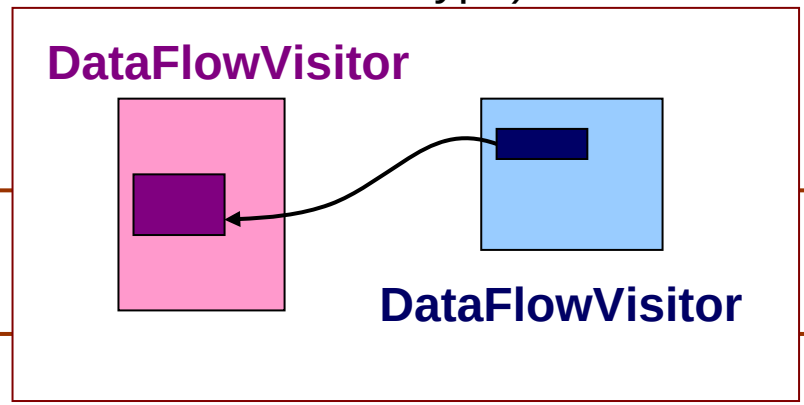


Encapsulation : types de données vs. préoccupations

Modèle de point de jointure (Kermeta)

DataFlowHandling

```
abstract class DataFlowVisitor inherits AbstractVisitor
{
operation visitSystem(system : System) : Void is abstract
operation visitDataSinkType(sink : Kernel::DataSinkType) : Void is abstract
operation visitDataSourceType(source : Kernel::DataSourceType) : Void is abstract
...
}
...
```



SynchronisationHandling

```
require "DataFlowVisitor.kmt"
aspect class DataFlowVisitor
{
operation visitSynchronization(synchronization : Synchronization) : Void is abstract
}
...
```

Pas de redéfinition possible

Modèle de point de jointure (AspectJ)

```
class Person {
protected Person partner ;
protected int age ;

public Person (int a) {
   setAge(a);
   partner = new Person(0);
}
public setPartner (Person s) {
    if (partner.age = = 0)
        partner = s;
    else
        throw new … ;
}
…}
```

Method call

Constructor call

Field reference

Field assignment

Exception handler execution

Définir la jonction aspect/programme

Type de point de jointure

pointcut basicPointcut (Person *p*) : …

pointcut basicPointcut (Person *p*) :

call (public void Person.* (.., Person)) **&&** target (*p*)

Syntaxe de déclaration de pointcut:

- le nommer
- quel contexte d'exécution
- Sélection points de jointure

Method modifier

Return type

Class container

Method name

Method parameters

Associate an object to p

Spécifier le traitement à exécuter

```
pointcut viewNames (Person pe, Person pa) :
    target (pe) && args (pa) && call (public void Person.* (.., Person))

before ( Person pe, Person pa) :
    viewNames (Person pe, Person pa)
{
    if  (pe.age == pa.age) proceed (pe, pa);
   else
        System.out.println (“Error …”)
}
```

Match joint point + set context

Action à exécuter

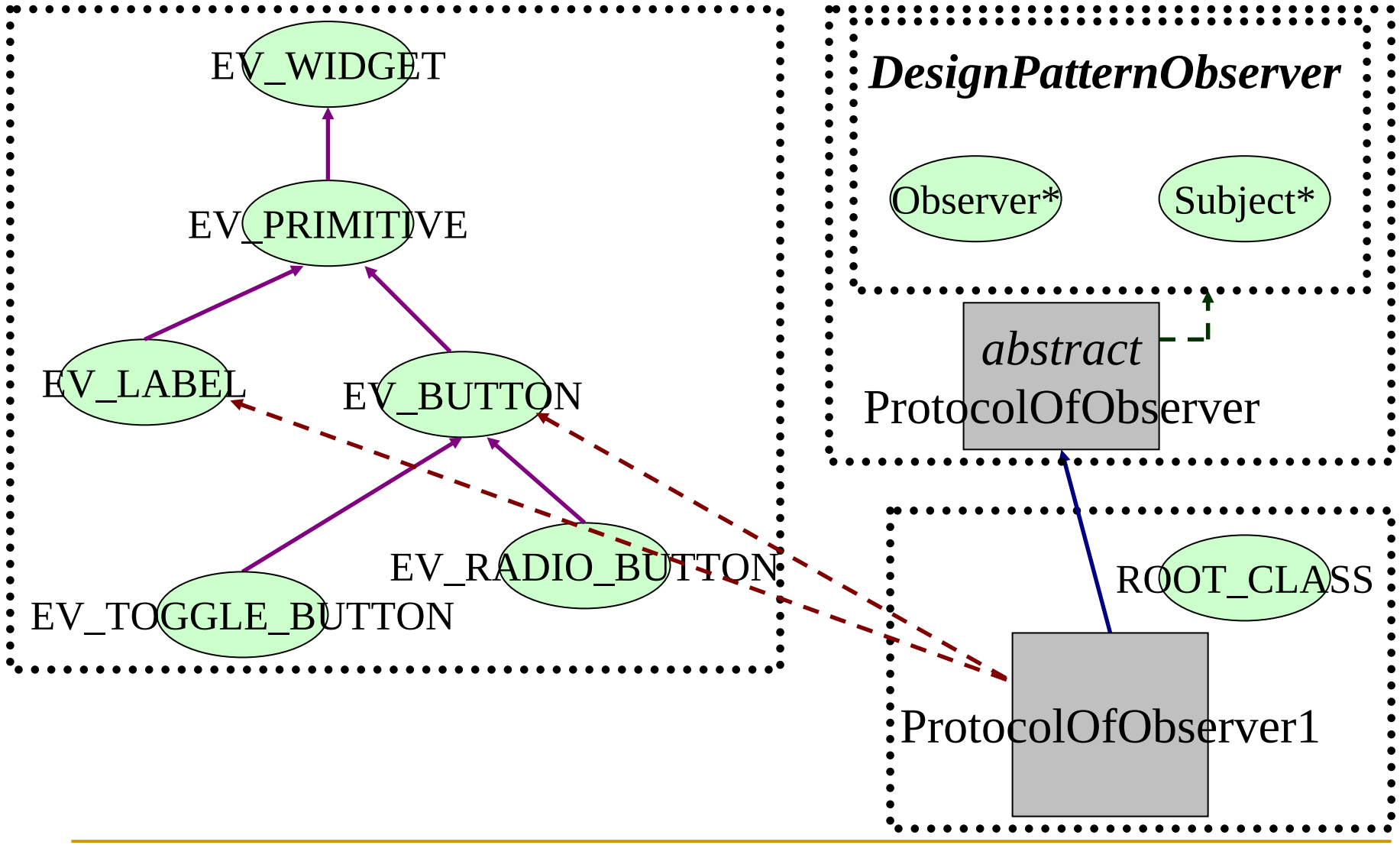
autres *advices*:

- before
- after
- around

Un exemple d'aspect

```
aspect PersonCrossCutting {
pointcut viewNames (Person pe, Person pa) :
  target (pe) &&  args (pa) &&  call (public void Person.* (.., Person))
before (Person pe, Person pa) :
   viewNames (Person pe, Person pa) {
    System.out.println (pa.name + "will become the partner of" + pe.name); }
after ( Person pe, Person pa) :
   viewNames (Person pe, Person pa) {
    System.out.println (pa.name + "became the partner of" + pe.name);  }
around (Person pe, Person pa) :
   viewNames (Person pe, Person pa) {
    if  (pe.age == pa.age) proceed (pe, pa);
   else  System.out.println ("Error : the age of persons are not compatible") }
…
}
```

Schéma général



Design pattern Observer (description)

```
import DesignPatternObserver ;
public abstract aspect ProtocolOfObserver
{
   abstract pointcut specificStateModification () ;
   pointcut stateModification (Subject s) : target (s) && specificStateModification () ;
   after (Subject s) : stateModification (s) { s.notifyObservers () ;}

   private Vector Subject.observers = new Vector () ;
   public void Subject.attach (Observer o) { observers.addElement (o); }
   public void Subject.detach (Observer o) { observers.removeElement (o); }
   public void Subject.notifyObservers () {
    for (int i = 0 ; i < observers.size () ; i++ ) ( (Observer) observers [i] ).update (this) ;
    }
}
```

Description du patron en dehors du contexte de réutilisation

Design pattern Observer (utilisation)

```
public aspect ProtocolOfObserver1 extends ProtocolOfObserver
{
   declare parents :  EV_BUTTON implements Subject ;
   pointcut specificStateModification () : call (void EV_BUTTON.click () ) ;

   declare parents :  EV_LABEL implements Observer ;
   public void EV_LABEL.update (Subject s) {
       EV_COLOR c = new EV_COLOR (…);
       set_foreground_color (c);
   }
}
```

Intégration pour gérer boutons et labels

Expressivité offertes par les aspects

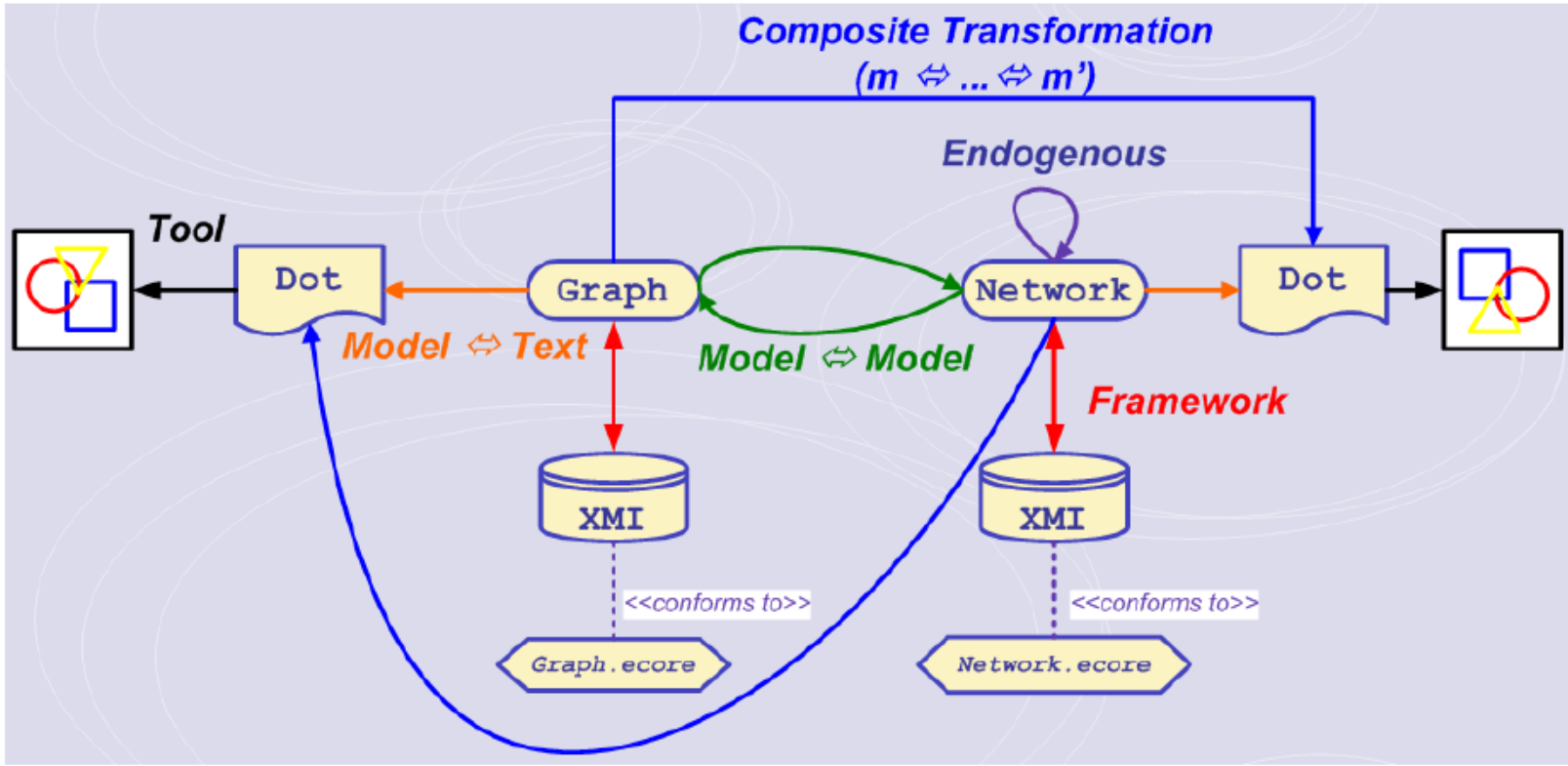
- Disposer de caractéristiques facultatives
  - Il suffit d'appliquer ou pas l'aspect
- Proposer des variantes
  - Un aspect par variante
  - Ces aspects peuvent devenir complexes
- Gestion des dépendances
  - Difficile à spécifier
  - Complexification des aspects

Mise en œuvre : Un tisseur + un compilateur

Exprimer la variabilité

- Héritage
- Généricité
- Programmation par aspects
- *Transformation de modèles*
- Composition de modèles
- Feature Models
- …

Transformation de modèles (1)



Besoin d'une autre représentation: *calcul, visualisation, nouvelles fonctionnalités…*

Transformation de modèles (2)

- Adapter un modèle ou un programme
  - Permettre sa manipulation par un autre paradigme (variabilité de présentation)
  - L'enrichir par de nouvelles fonctionnalités (variabilité de fonctionnalité)

    tissage d'aspects, refactoring…
  - Réutiliser un algorithme existant réalisé pour un autre modèle (variabilité de comportement)
- Au niveau des modèles ou du code (passage par une réification : ex: compilateur)

Exprimer la variabilité

- Héritage
- Généricité
- Programmation par aspects
- Transformation de modèles
- *Composition de modèles*
- Feature Models
- …

Composition de modèles

- Une transformation avec n entrées
  - L'enrichir par de nouvelles fonctionnalités (variabilité de fonctionnalité)
  - Adapter un comportement par fusion (variabilité de comportement)
- Au niveau des modèles ou du code (passage par une réification : ex: compilateur)

Exprimer la variabilité

- Héritage
- Généricité
- Programmation par aspects
- Transformation de modèles
- Composition de modèles
- *Feature Models*
- …

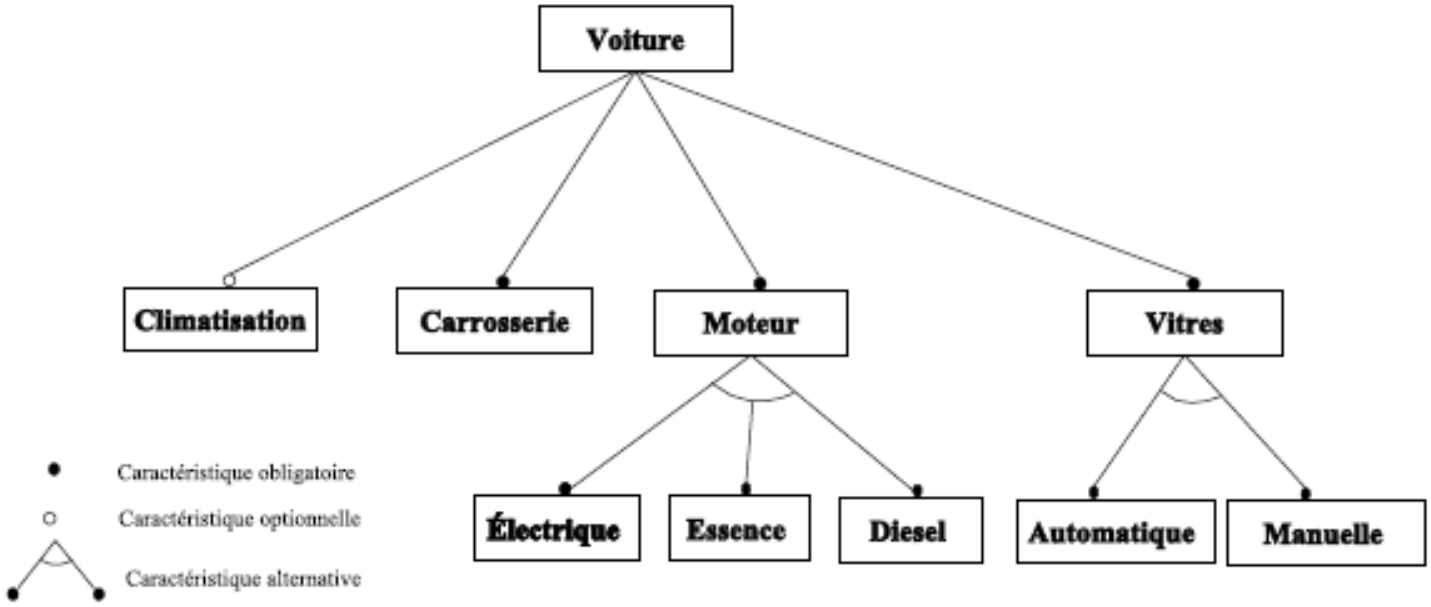
Analyse des besoins (*feature model*)

- Analyse du domaine
- Prise en compte des principaux critères de variabilités
- Modélisation
  - Structure
  - Comportement?
- Description hiérarchique
- Nombreuses extensions
- …

Que des noms de « feature »

Peut être complété par une approche
qui intègre la variabilité des aspects dynamiques

Un exemple de ligne de produits

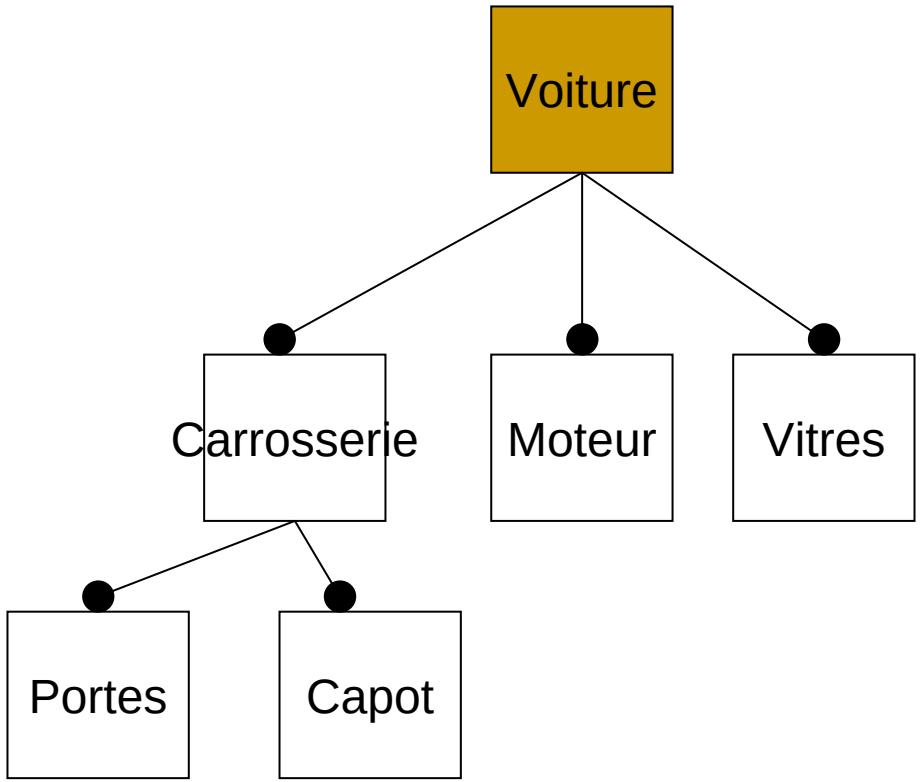


Feature Model (1)

Concept

Features

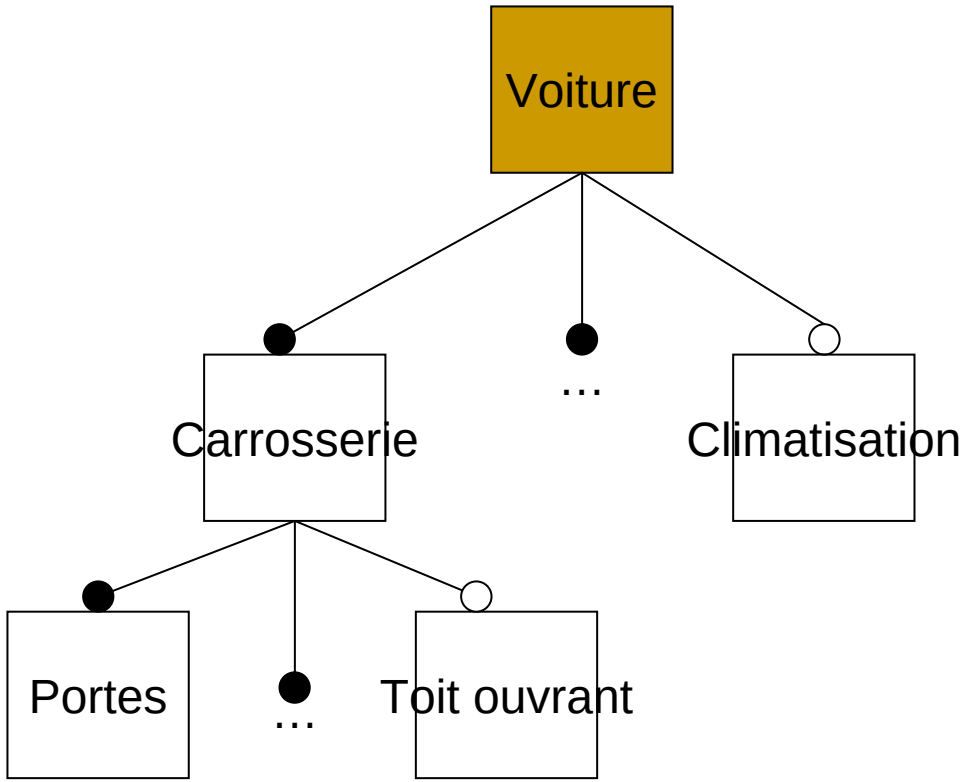
Sub-features



- ✓ définition des parties communes
- ✓ toute *sub-feature* obligatoire n'est pas commune

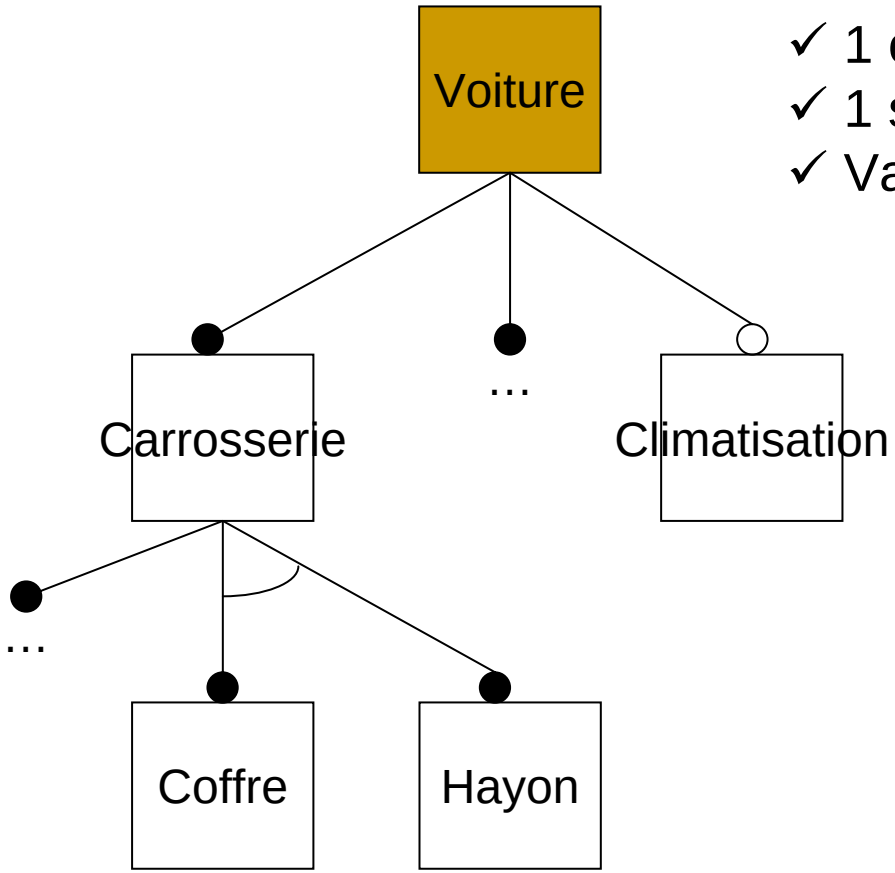
Mandatory Features

Feature Model (2)



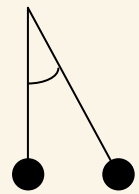
Optional Features

Feature Model (3)

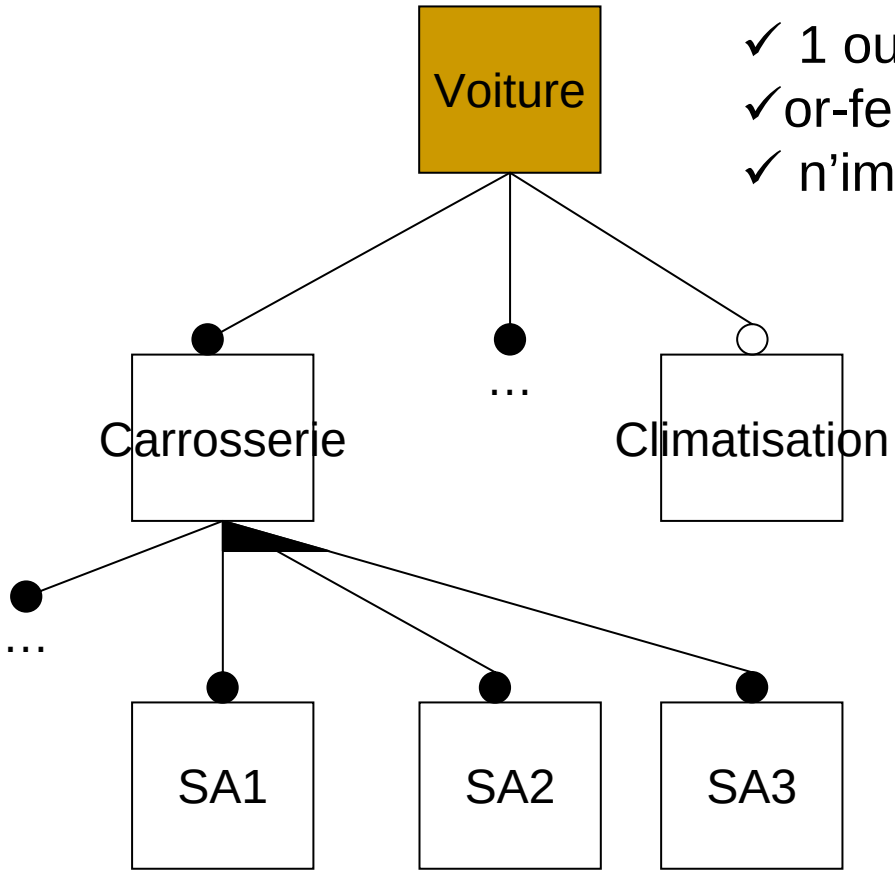


- ✓ 1 ou plusieurs séries de variantes
- ✓ 1 seule série de variantes = dimension
- ✓ Variantes optionnelles ou obligatoires

Alternative Features

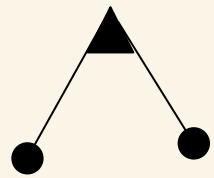


Feature Model (4)

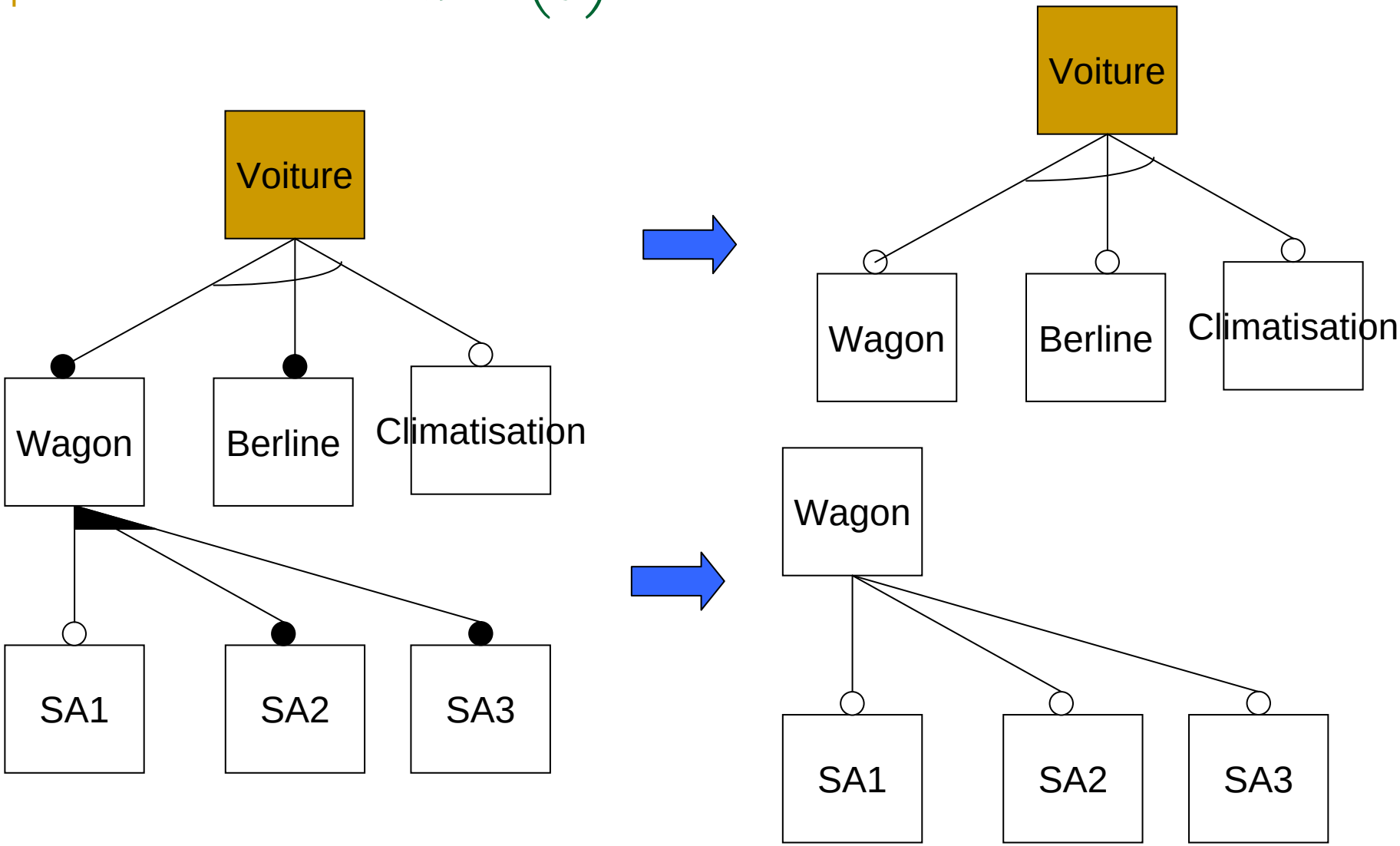


- ✓ 1 ou plusieurs séries de or-features
- ✓or-features optionnelles ou obligatoires
- ✓ n'importe quel sous-ensemble non vide

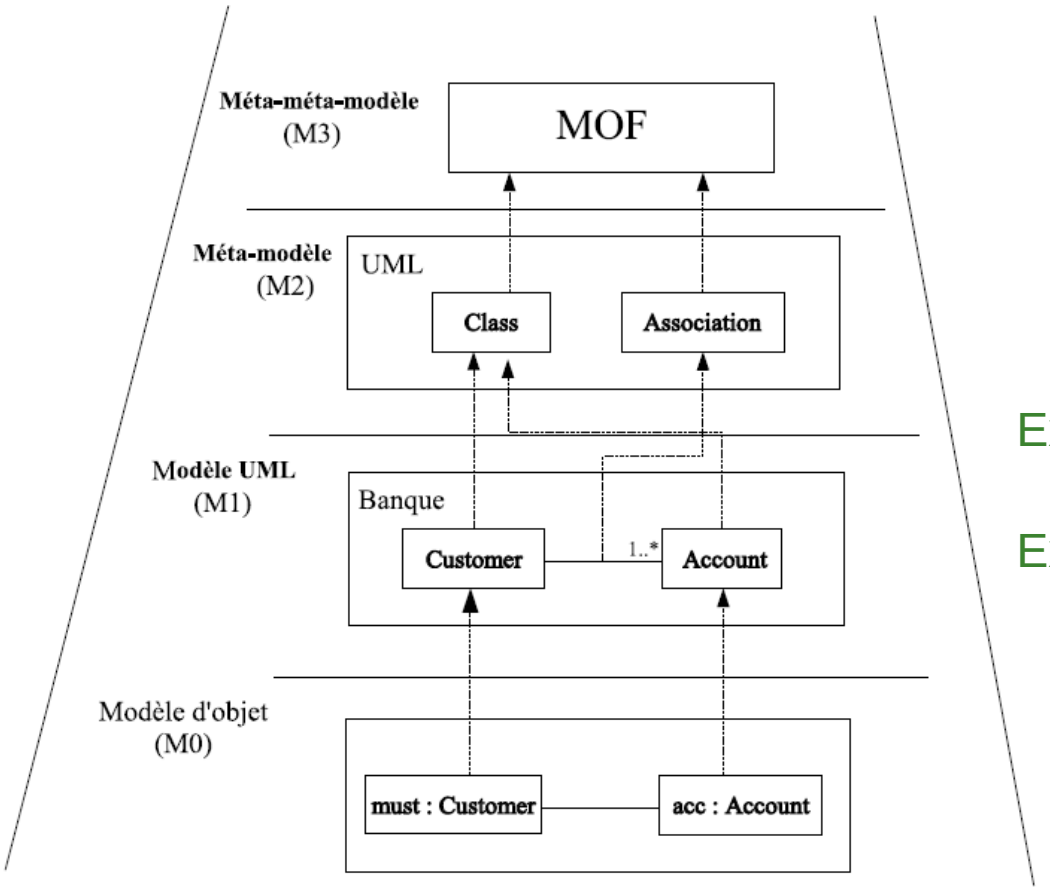
Or-Features



Feature Model (5)



Méta niveaux et ajout de la variabilité

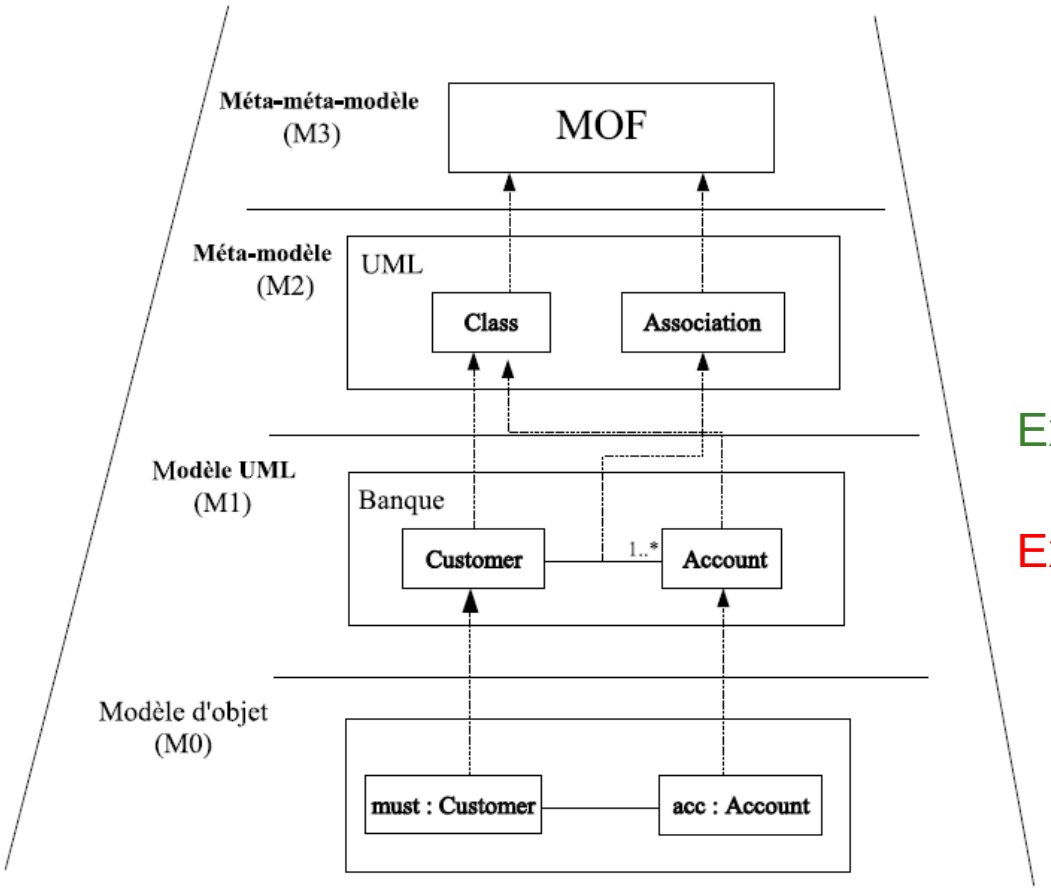


Extension d'UML

Extension d'un dialecte d'UML

Extension d'un modèle spécifique

Méta niveaux et ajout de la variabilité



Extension d'UML

Extension d'un dialecte d'UML

Extension d'un modèle spécifique

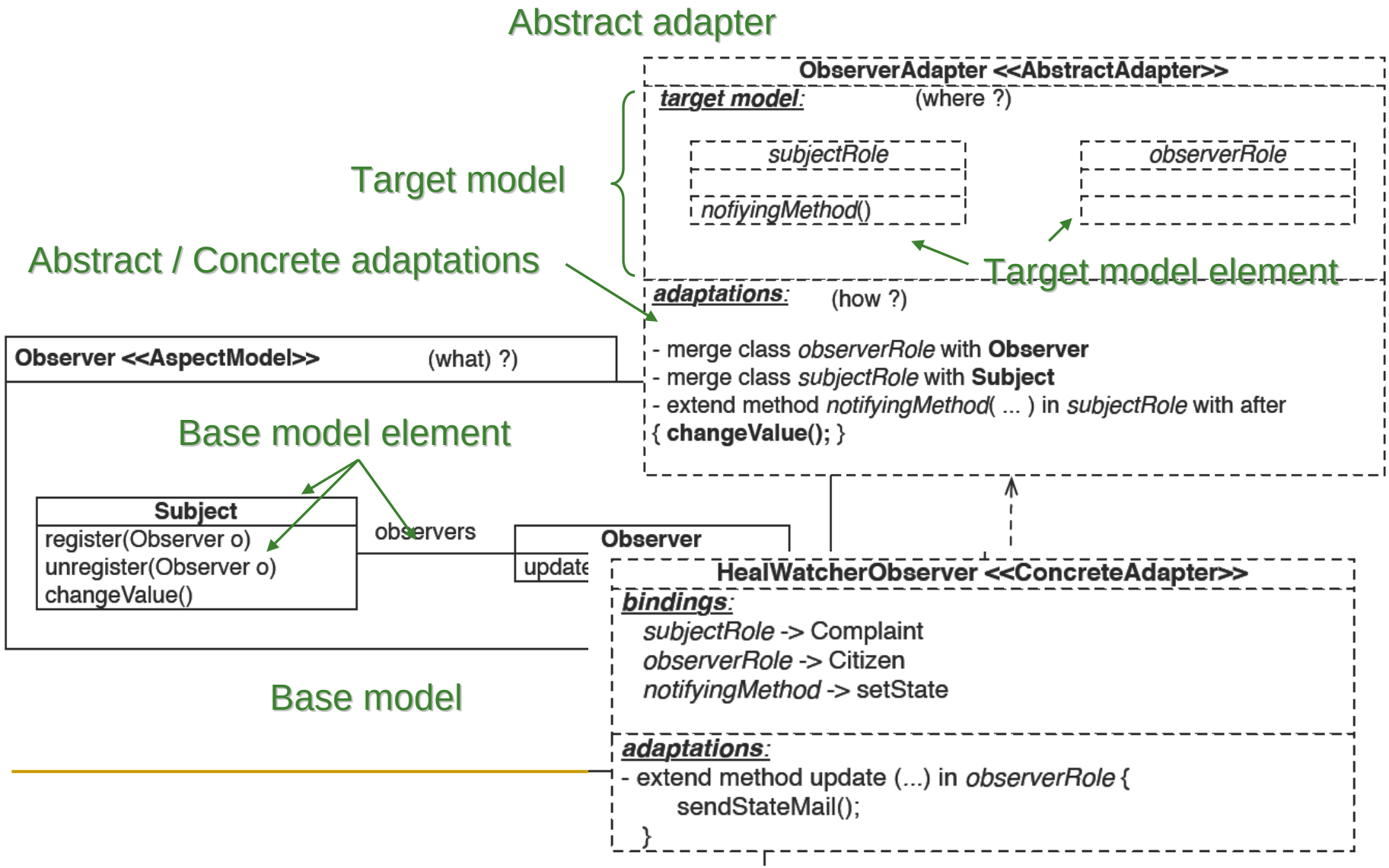
Extension d'un modèle spécifique

Diagramme de classes uniquement

- Construire le modèle sans variabilité
- Introduction de la variabilité
  - Choisir les éléments variables
  - Définir l'expressivité de la variabilité (pour chacun)
  - Définir les types de contrainte
  - Positionner les éléments par rapport aux contraintes
- Mise en œuvre du moteur de dérivation
- Interface pour la spécification du produit

Exemple : un métamodèle pour l'AOM

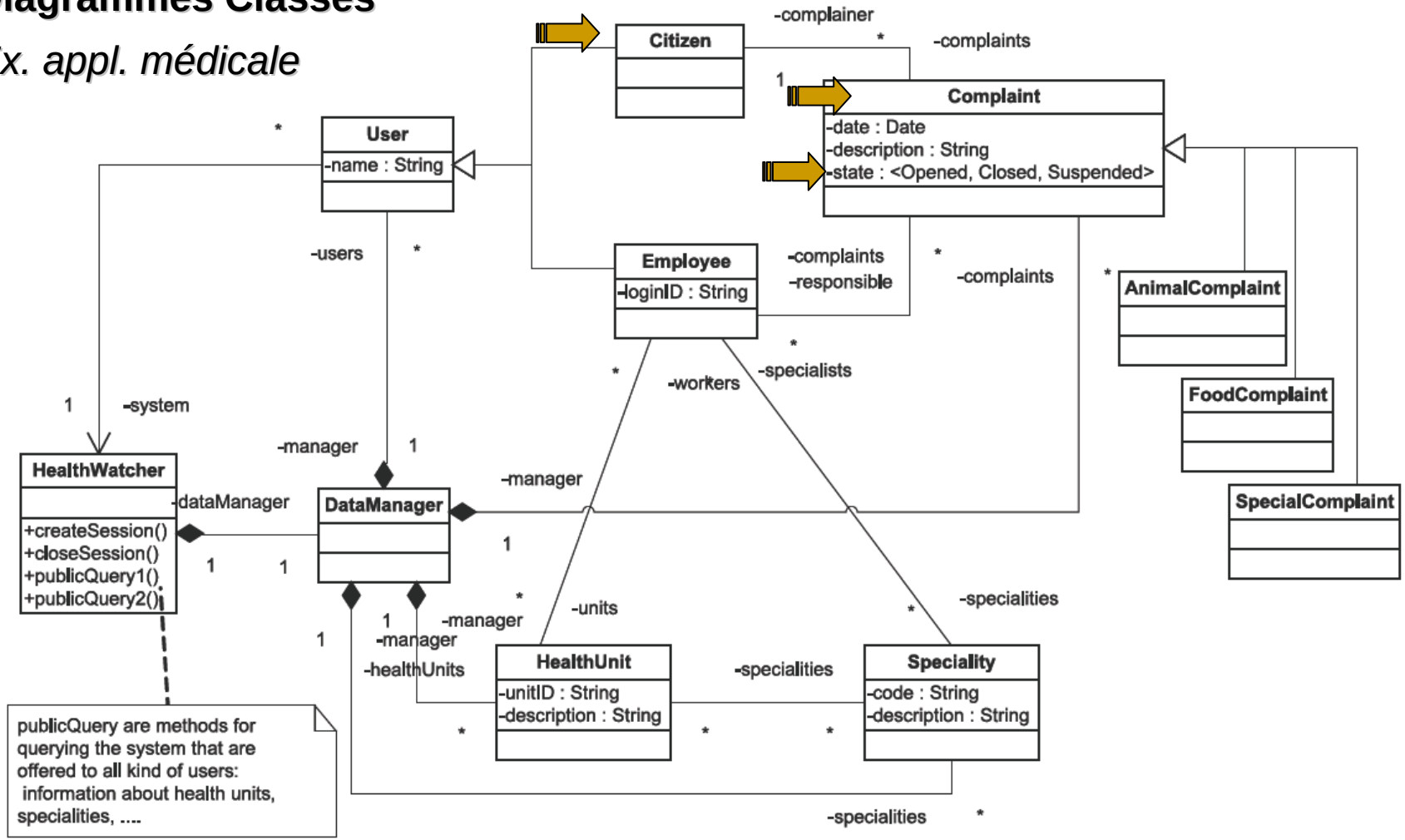
Explication des concepts



Exemple : un métamodèle pour l'AOM

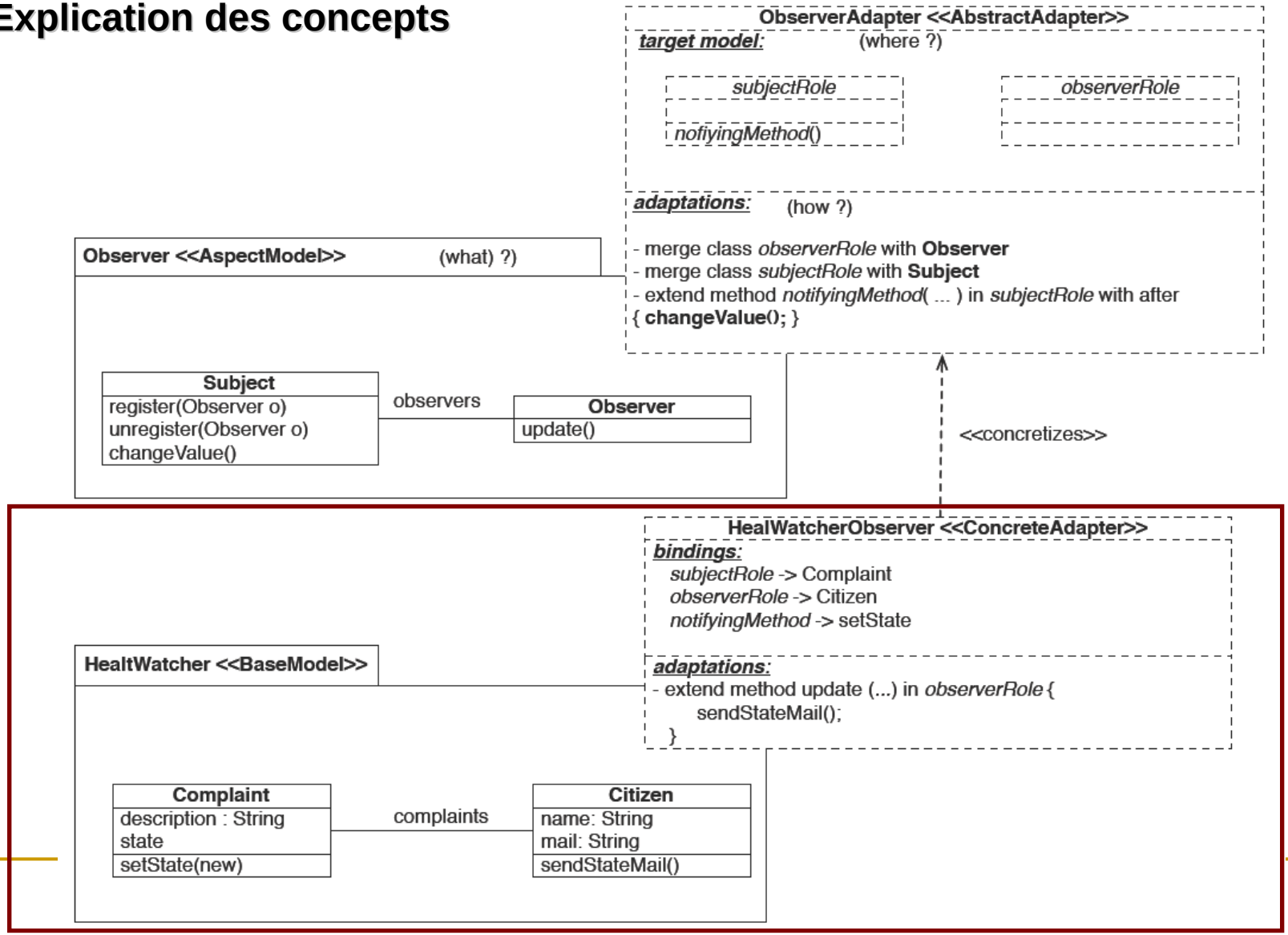
Diagrammes Classes

Ex. appl. médicale



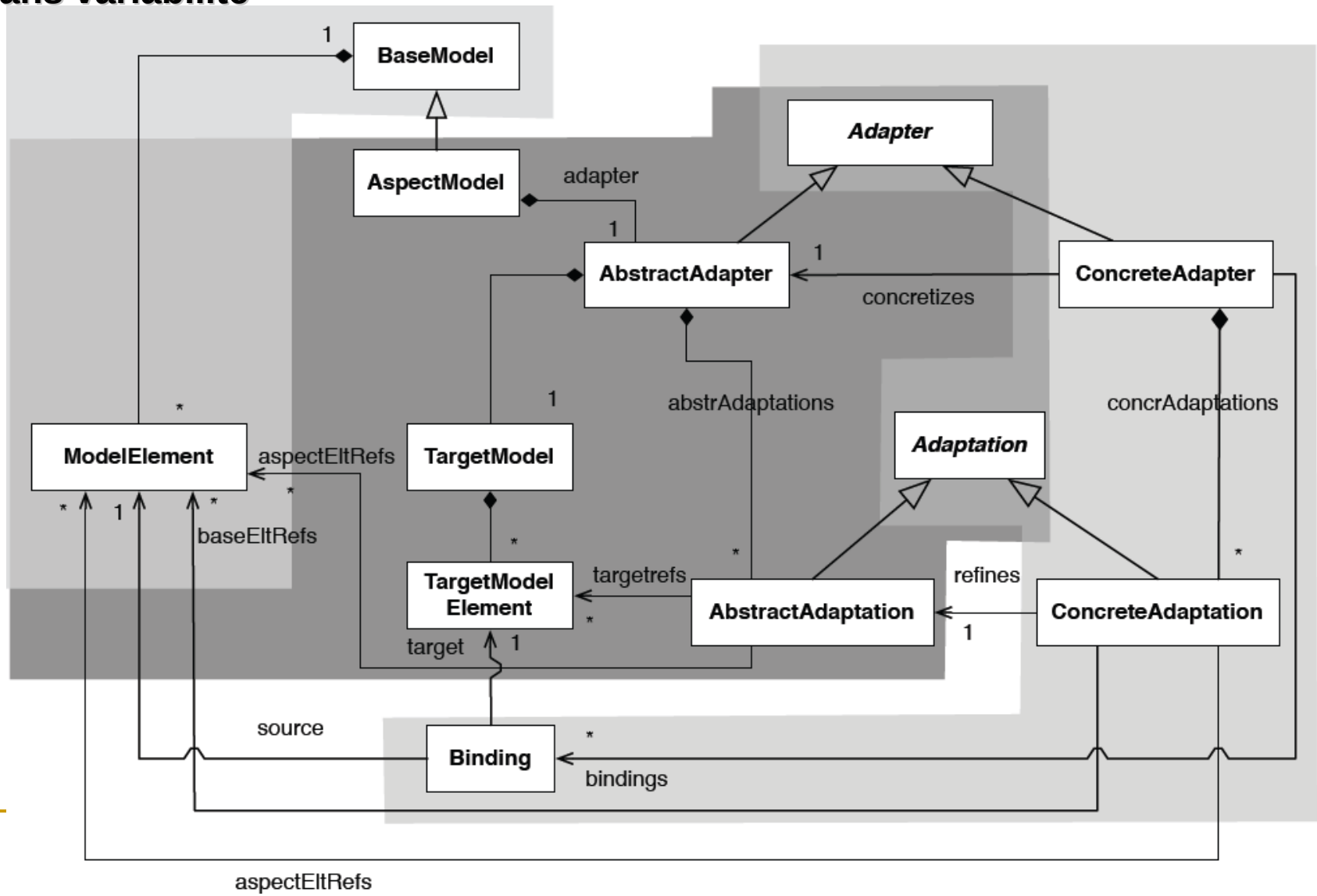
Exemple : un métamodèle pour l'AOM

Explication des concepts

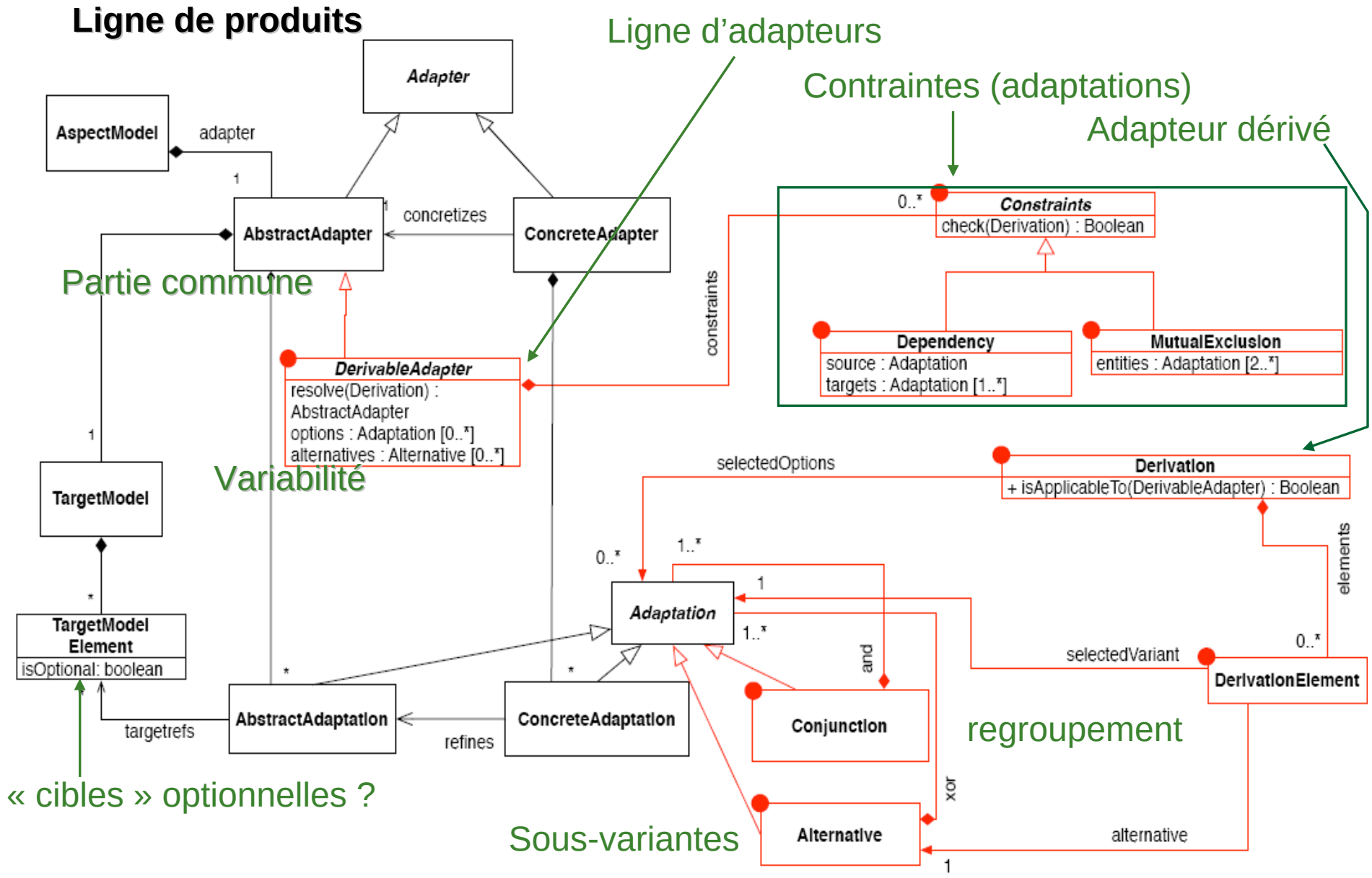


Exemple : un métamodèle pour l'AOM

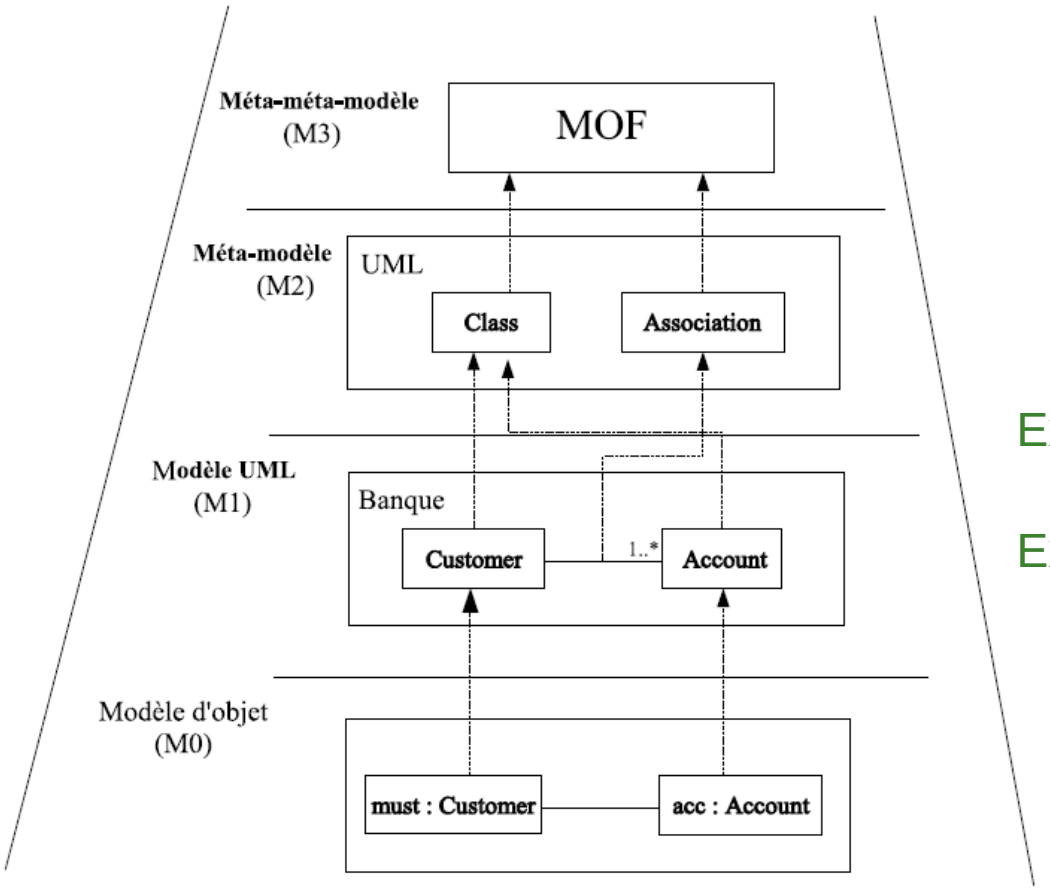
Sans variabilité



Exemple : un métamodèle pour l'AOM



Méta niveaux et ajout de la variabilité



Extension d'UML

Extension d'un dialecte d'UML

Extension d'un modèle spécifique

Extension d'un métamodèle : UML

- Modélisation de la variabilité
  - Cas d'utilisation
  - Diagrammes statiques (ex: classes)
  - Diagrammes dynamiques (ex: séquences)
- Modélisation des contraintes

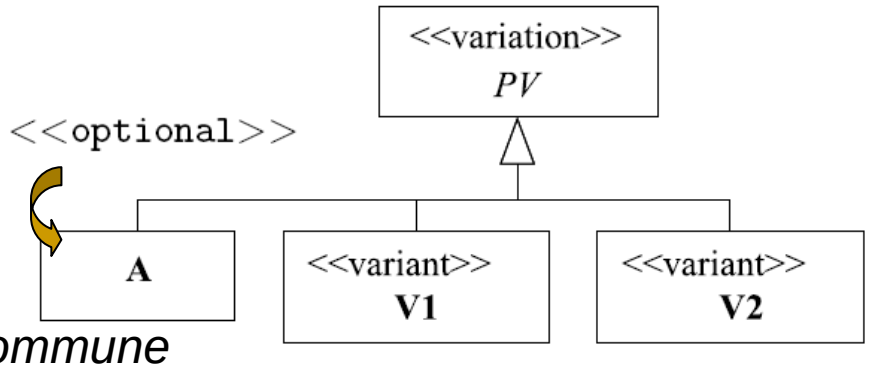
  Génériques, propre à une LDP
  - OCL

    Exemple : thèse de Ziadi
  - Autres
- Dérivation de produit
  - Diagrammes statiques (ex: classes)
  - Diagrammes dynamiques (ex: séquences)

Modélisation de la variabilité (1)

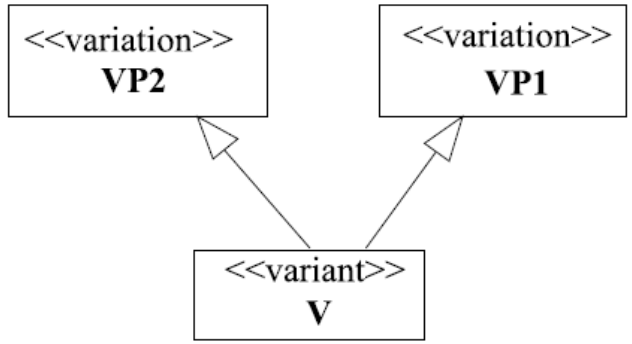
Diagrammes Classes

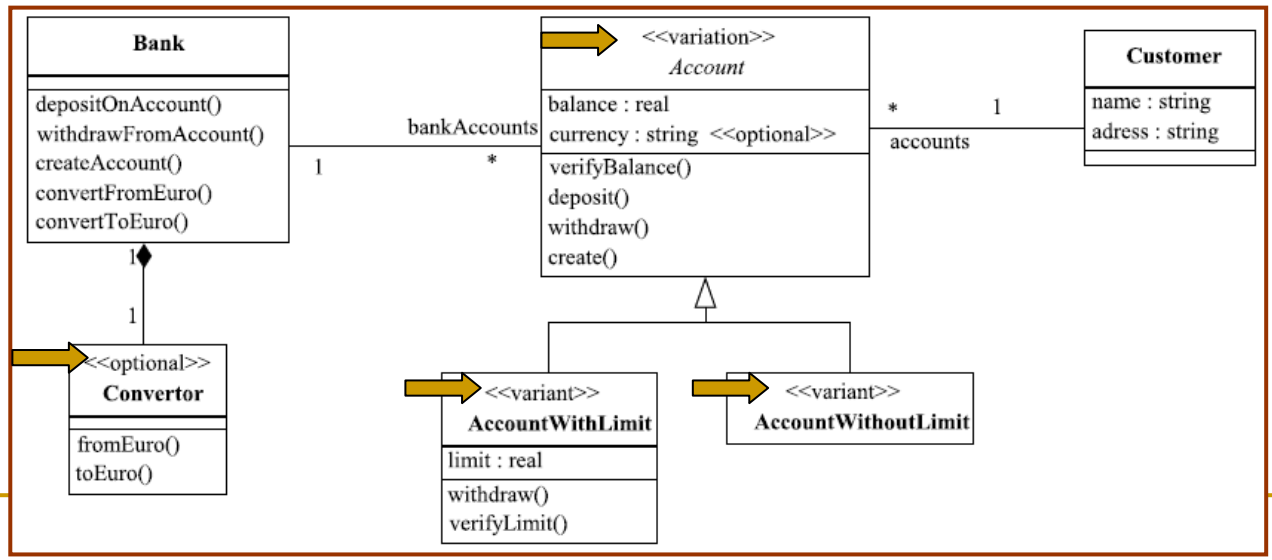
1 point de variation



commune

2 points de variation



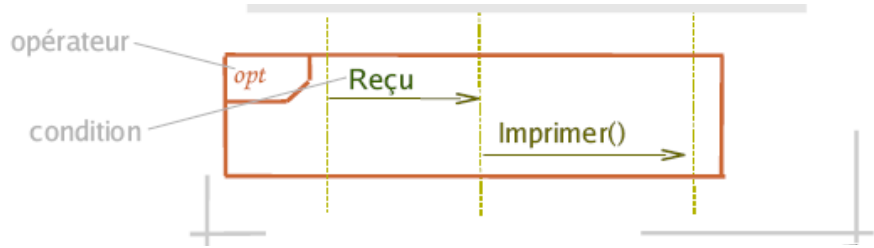


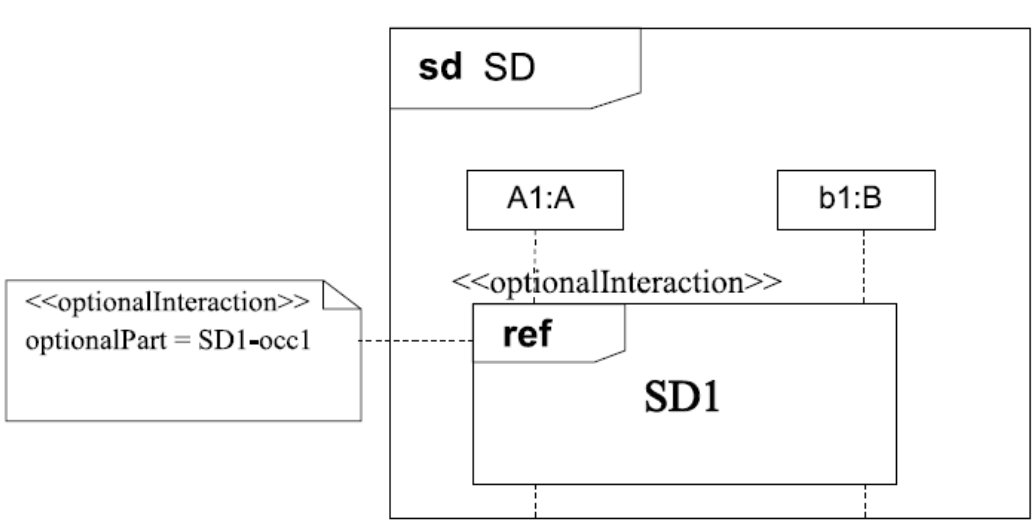
Modélisation de la variabilité (2)

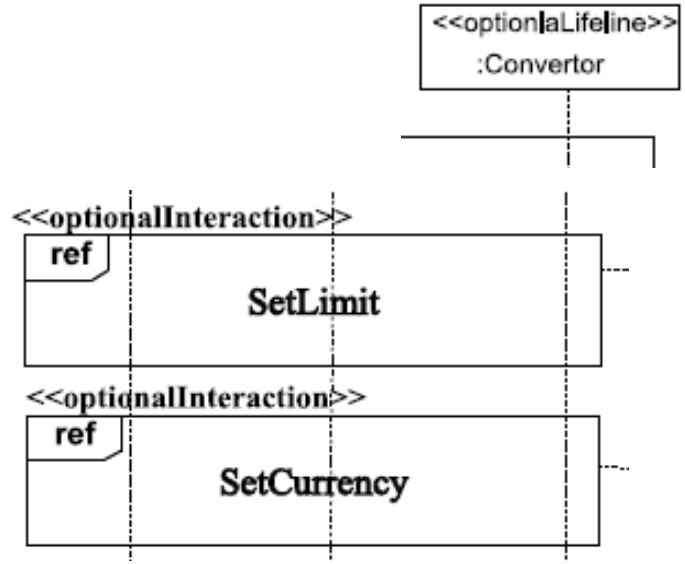
Diagrammes de séquences

Interaction optionnelle

≠ OPT





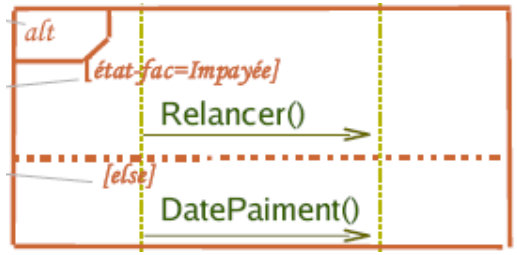


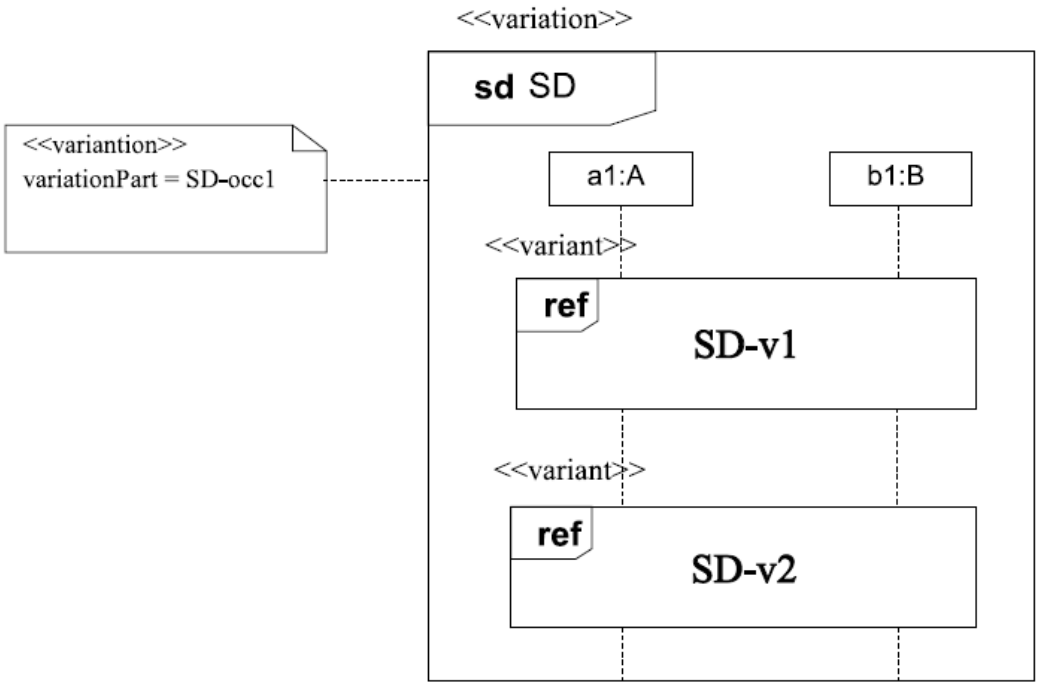
SD1 un diagramme de séquence optionnel…

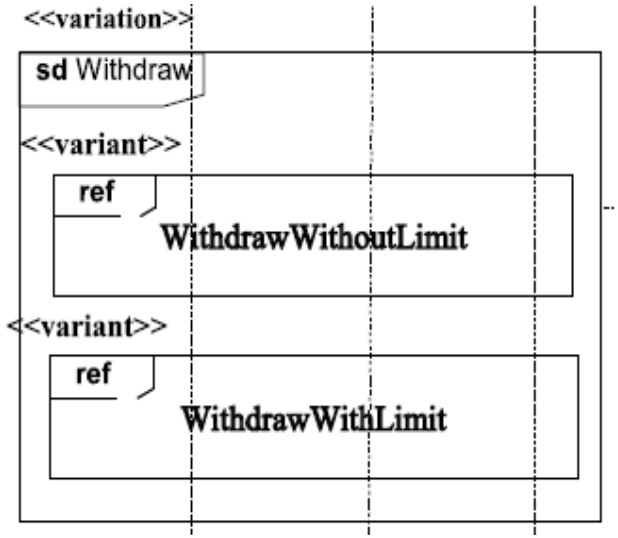
Modélisation de la variabilité (3)

Diagrammes de séquences

Variantes d'interaction ≠ ALT





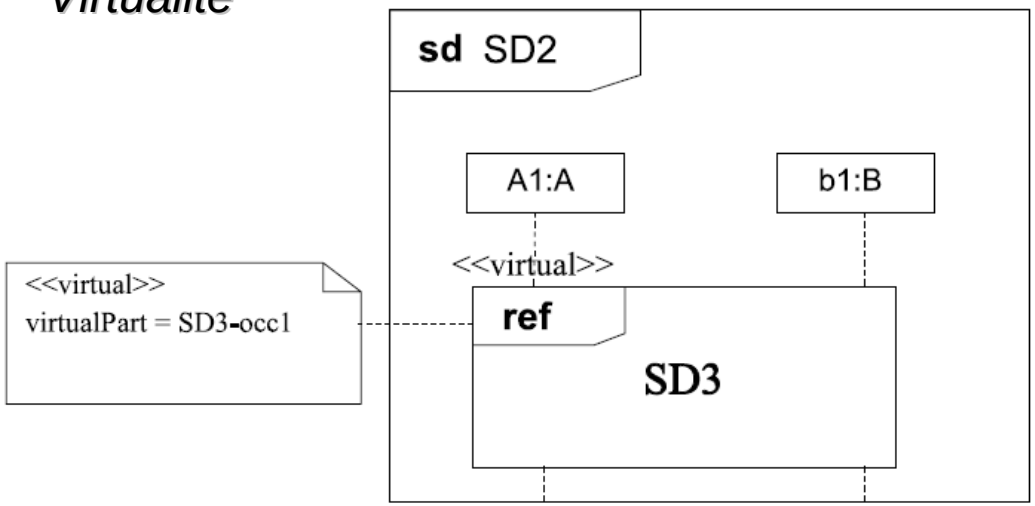


SD un choix entre plusieurs diagrammes de séquence: SD-V1…

Modélisation de la variabilité (4)

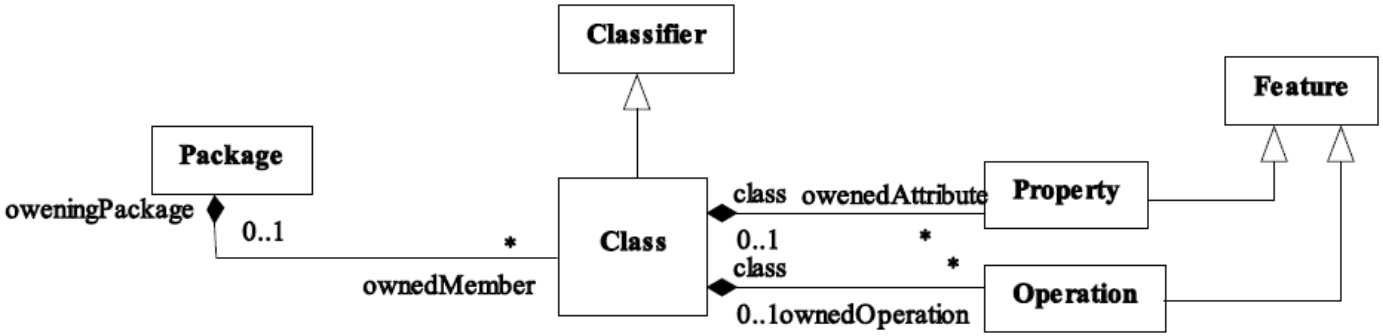
Diagrammes de séquences

Virtualité

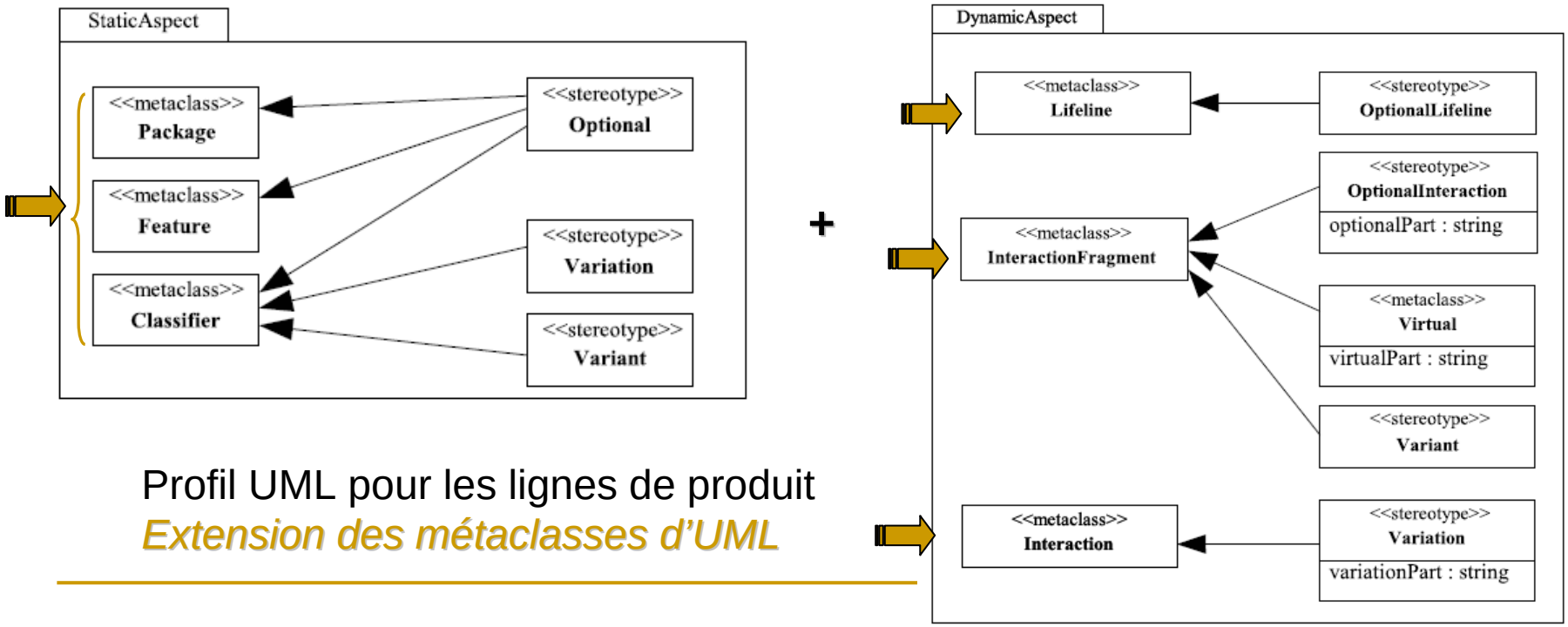


La virtualité d'une interaction signifie que son comportement peut être redéfini par une autre interaction de raffinement associée à un produit particulier.

Synthèse extension UML



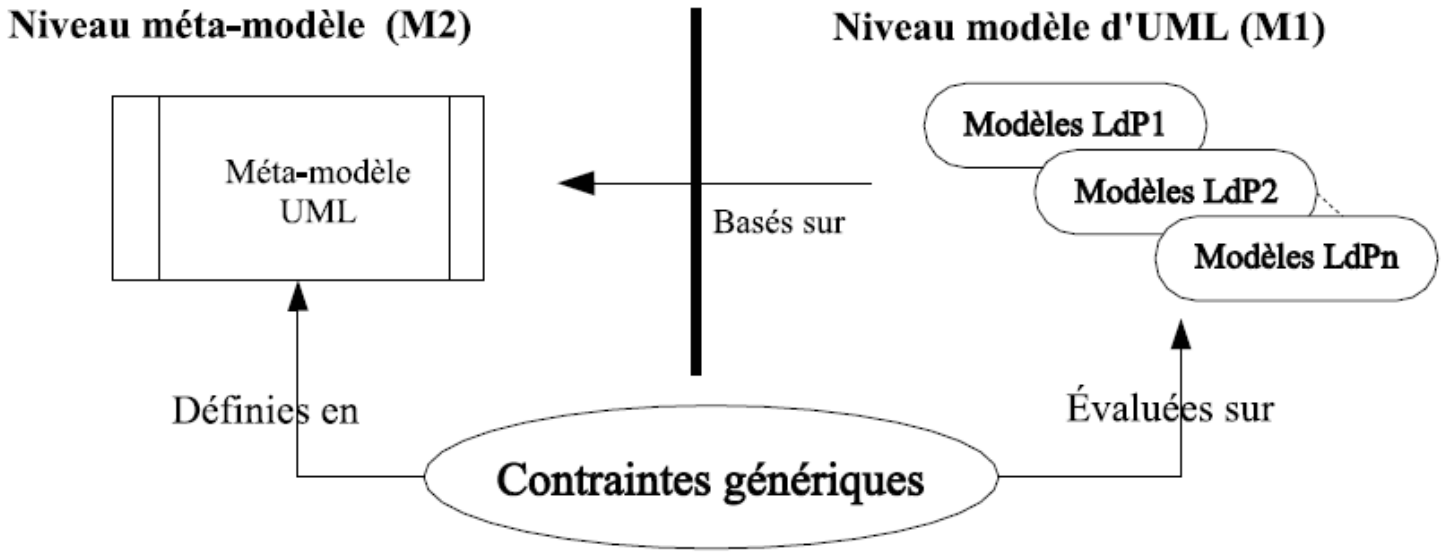
Diag. Classes
sans variabilité



Profil UML pour les lignes de produit
Extension des métaclasses d'UML

Modélisation des contraintes (1)

Contraintes génériques

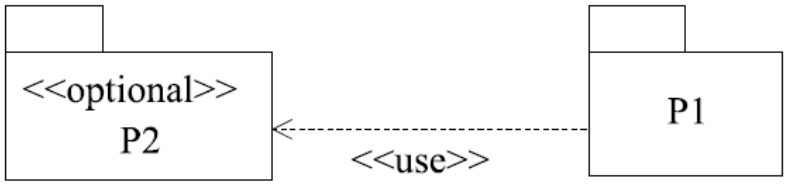


Exemple : un élément commun ne doit pas pouvoir utiliser d'éléments variables

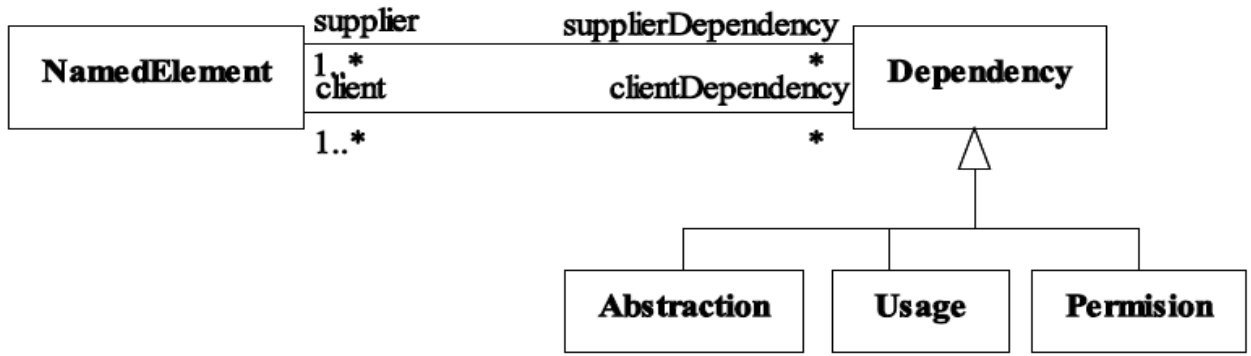
Définition: Un élément dans un modèle de LdP est dit variable s'il est optionnel ou variant. Dans les autres cas, il est dit commun.

Modélisation des contraintes (2)

Contraintes génériques

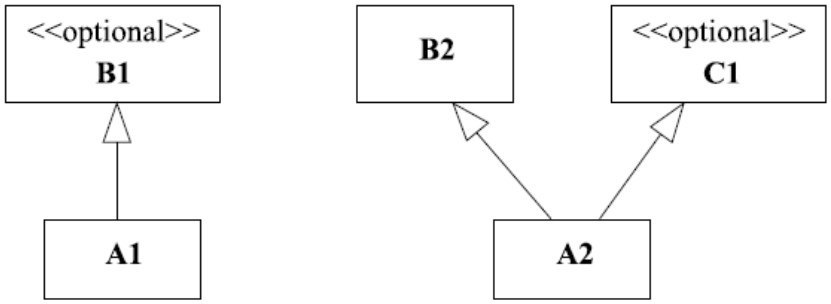



```
context Dependency inv :
  self.supplier ->exists(S| S.isStereotyped('optional') or
    S.isStereotyped('variant')) implies
  self.client -> forAll( C| C.isStereotyped('optional') or
    C.isStereotyped('variant'))
```

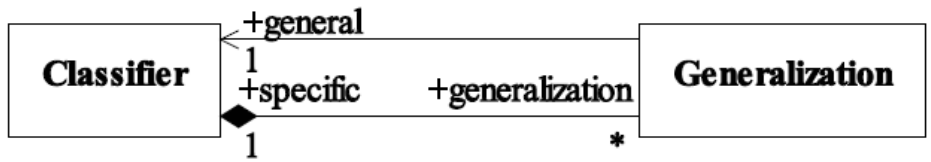


Modélisation des contraintes (3)

Contraintes génériques

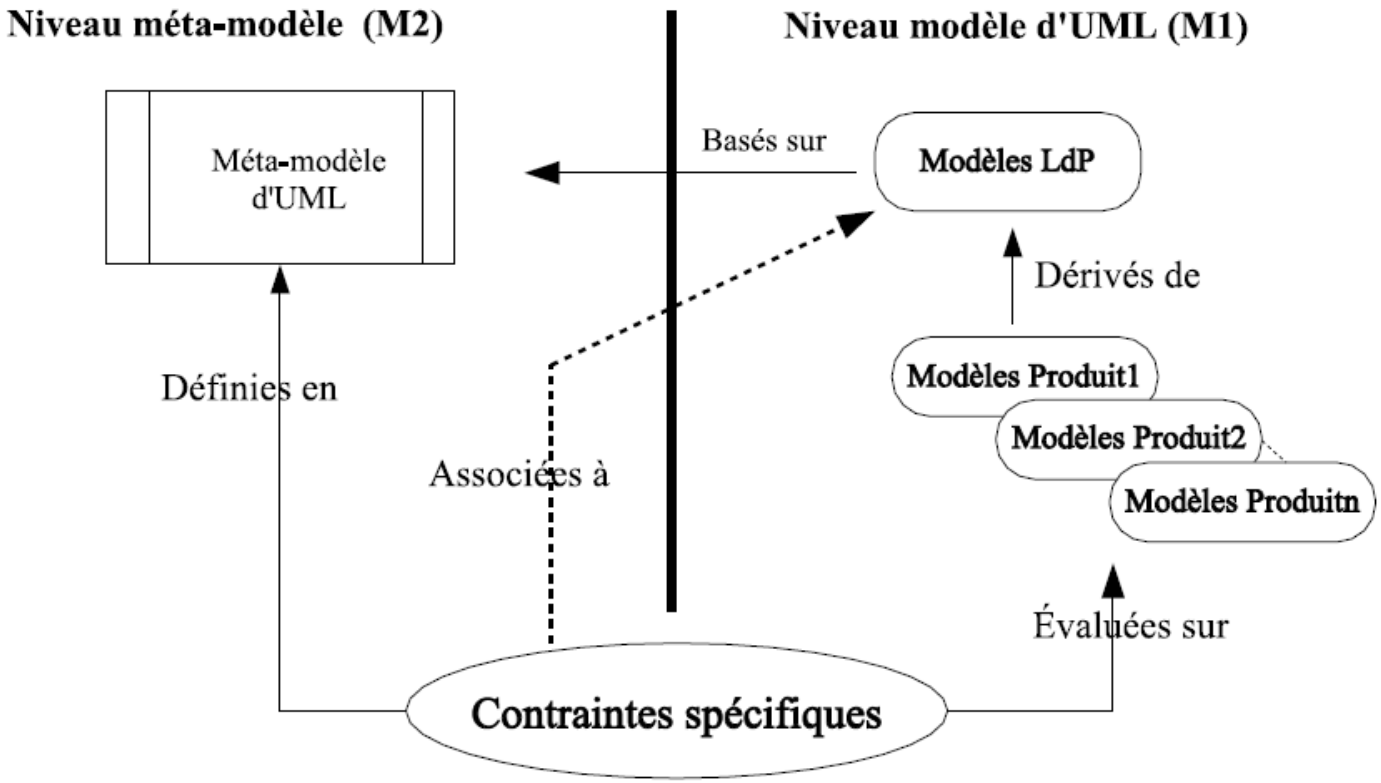



```
context Generalization inv :
  (self.general.isStereotyped('optional') or
  self.general.isStereotyped('variant')) implies
  (self.specific.isStereotyped('optional') or
  self.specific.isStereotyped('variant'))
```



Modélisation des contraintes (4)

Contraintes de la LdP



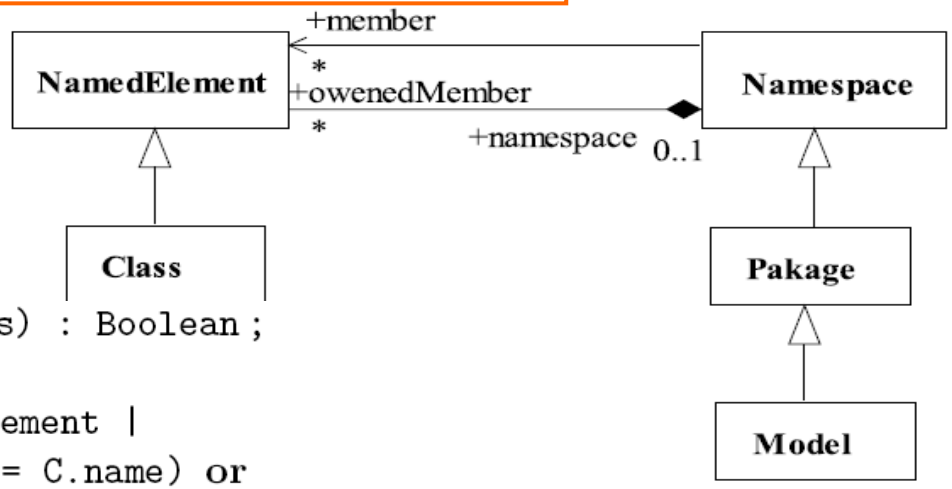
Modélisation des contraintes (5)

Contraintes de la LdP

```
context Model inv :
    self.presenceClass('C1') implies self.presenceClass('C2')
```

context Model **inv** :
(self.presenceClass('AccountWithLimit') implies
not self.presenceClass('AccountWithoutLimit')) and
(self.presenceClass('AccountWithoutLimit') implies
not self.presenceClass('AccountWithLimit'))

- Présence
- Exclusion mutuelle




```
context Namespace : :presenceClass(C : Class) : Boolean ;
      presenceClass =
       self.ownedMember->exists(el : NamedElement |
        (el.oclIsKindOf(Class) and cl.name = C.name) or
        ( el.isKIndOf(Namespace) and el.presenceClass(C)))
```

Association Feature Model / MOF

Partie statique

- On utilise le meilleur des deux :
  - Feature Model : la variabilité (famille de produits)
  - MOF/UML (ici EMF) : la description d'un produit
- On passe d'un modèle à l'autre
  - Par transformation (kermeta)
- Importance de la cohérence du produit :
  - Contraintes de la LdP
  - Raffinement du produit après dérivation

Démonstration

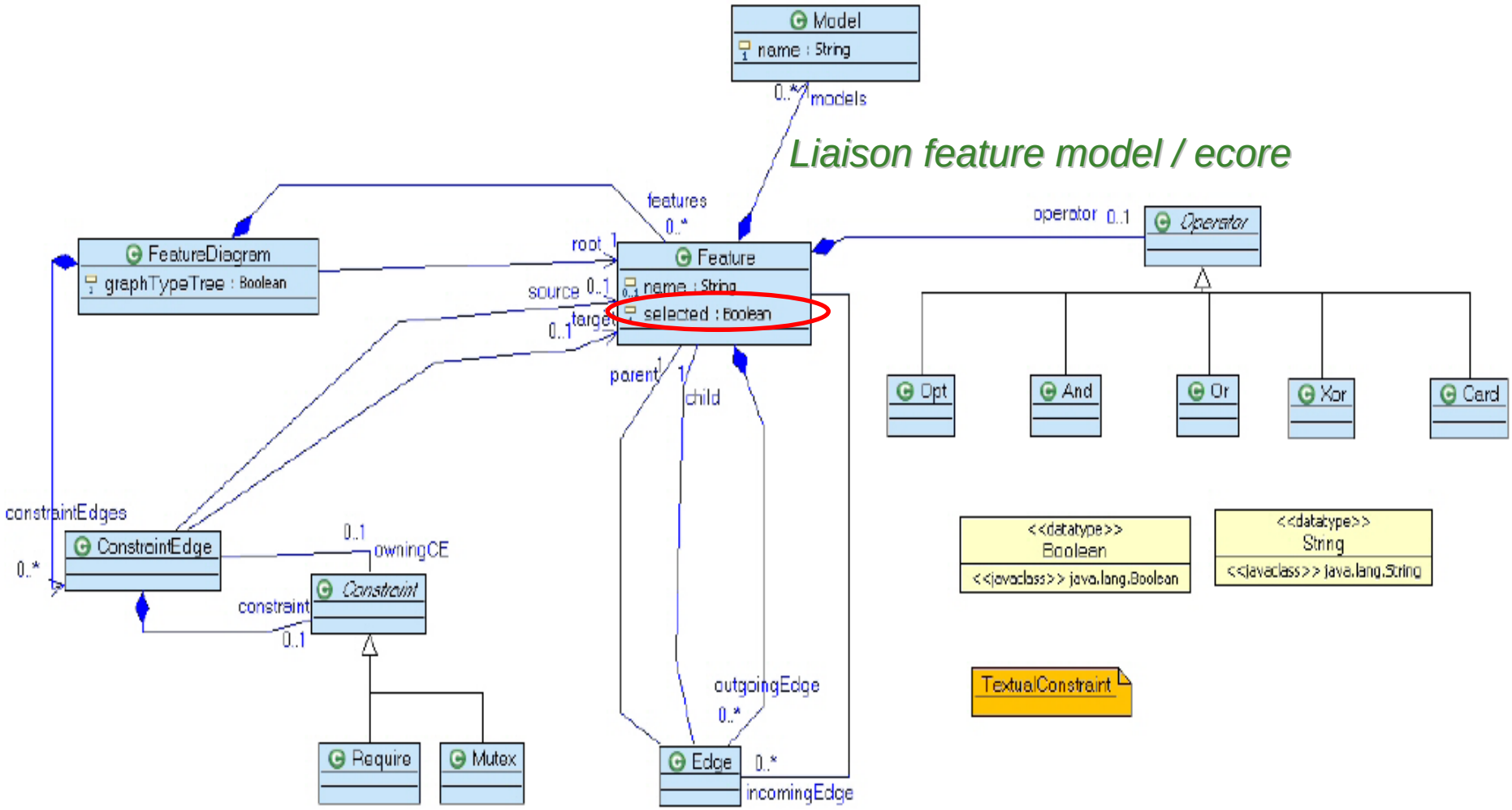
- Description du catalogue
  - Instance du métamodèle « feature model »
  - Instances du métamodèle Ecore
  - Lien entre les deux
- Description d'un produit
  - Utilise « selected »
- Dérivation
- Raffinement cohérent du produit

Vérification :
Contraintes génériques
Contraintes LDP

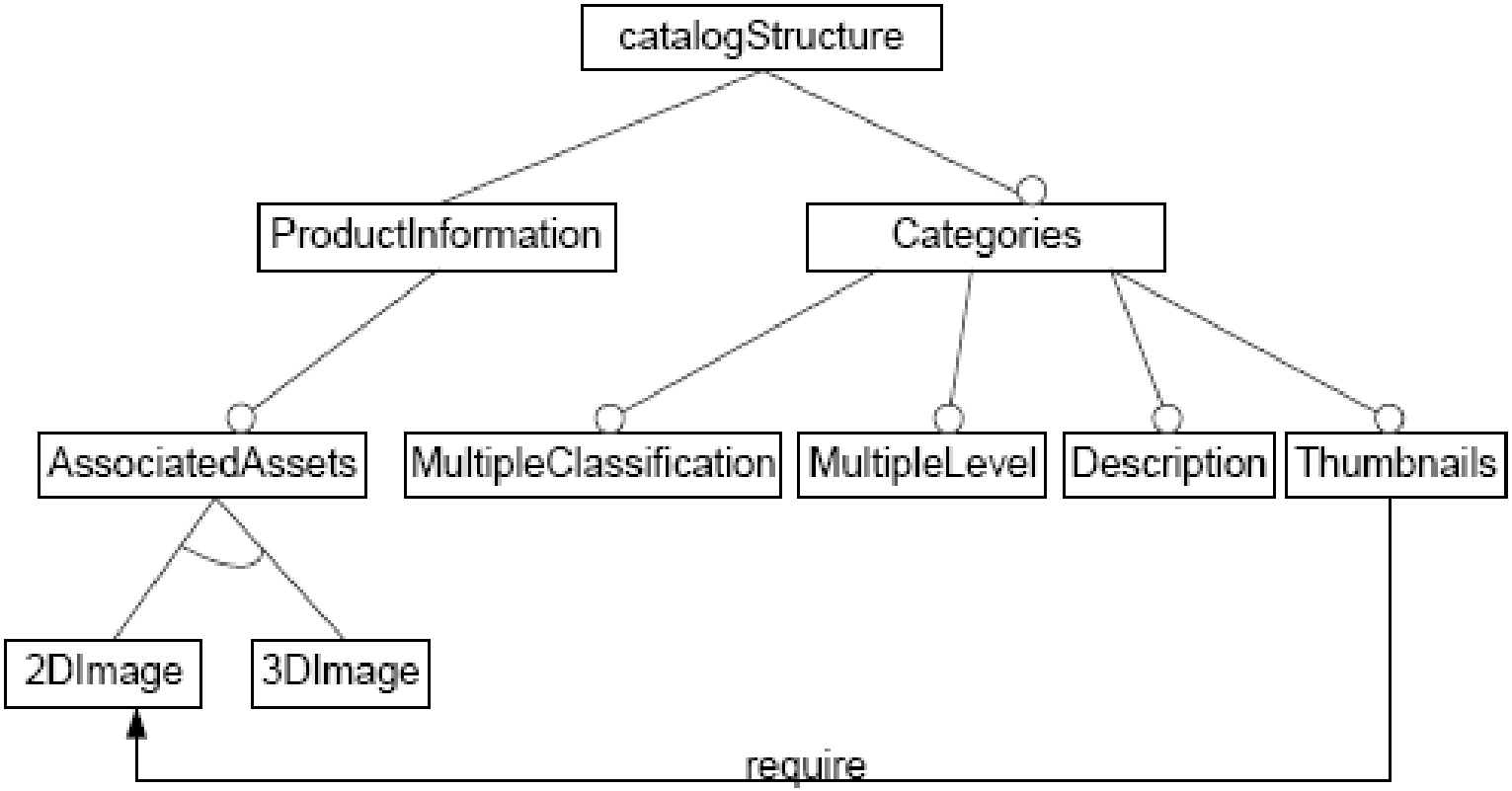
Processus de dérivation

- Vérification des contraintes
  - Génériques *(ex: suppression partie commune)*
  - LDP *(violation « requires »)*
- Composition des fichiers Ecore (*Kompose*)
  - Utilise « selected »
  - Fusion de modèles
- Production modèle du produit dérivé
- Raffinement cohérent du produit
  - Utilise un DSL

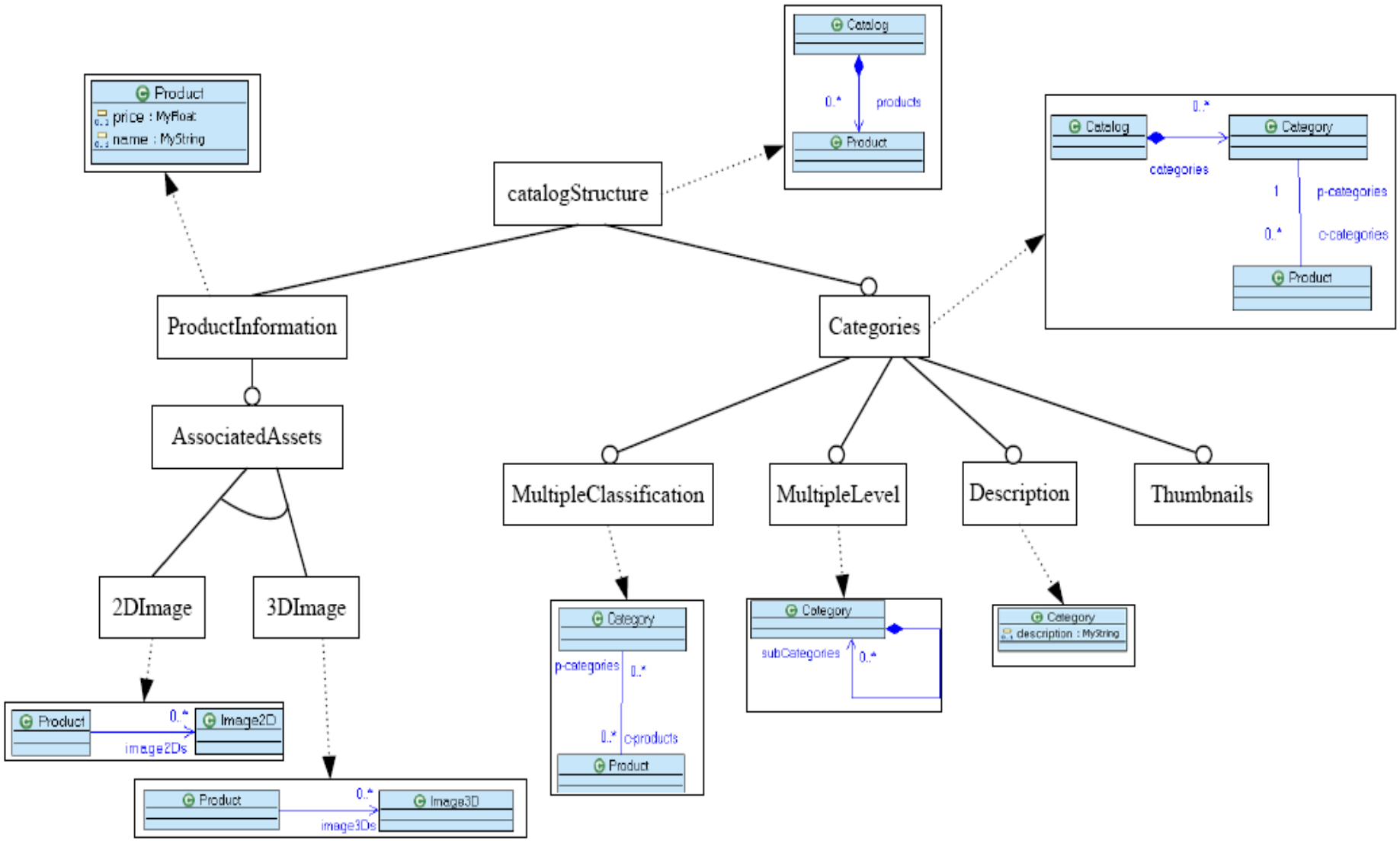
Le métamodèle « Feature model »



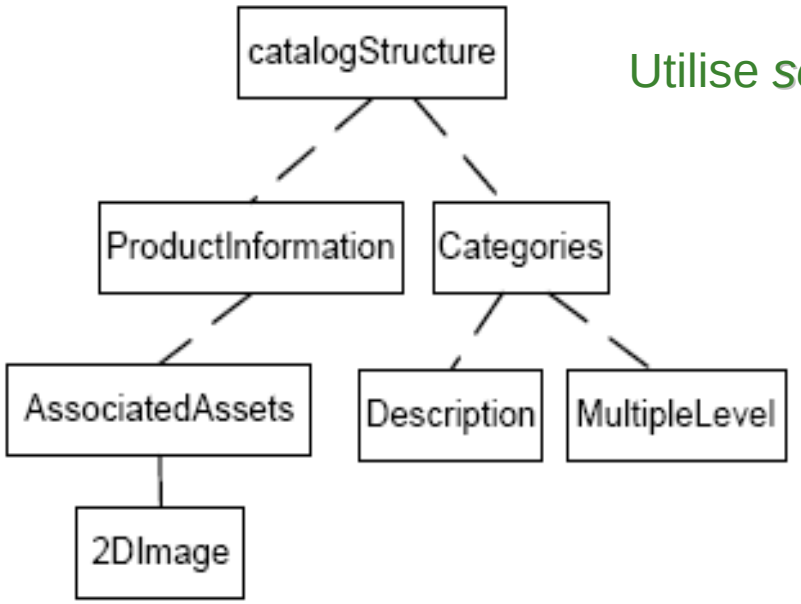
Variabilité d'un catalogue de produits



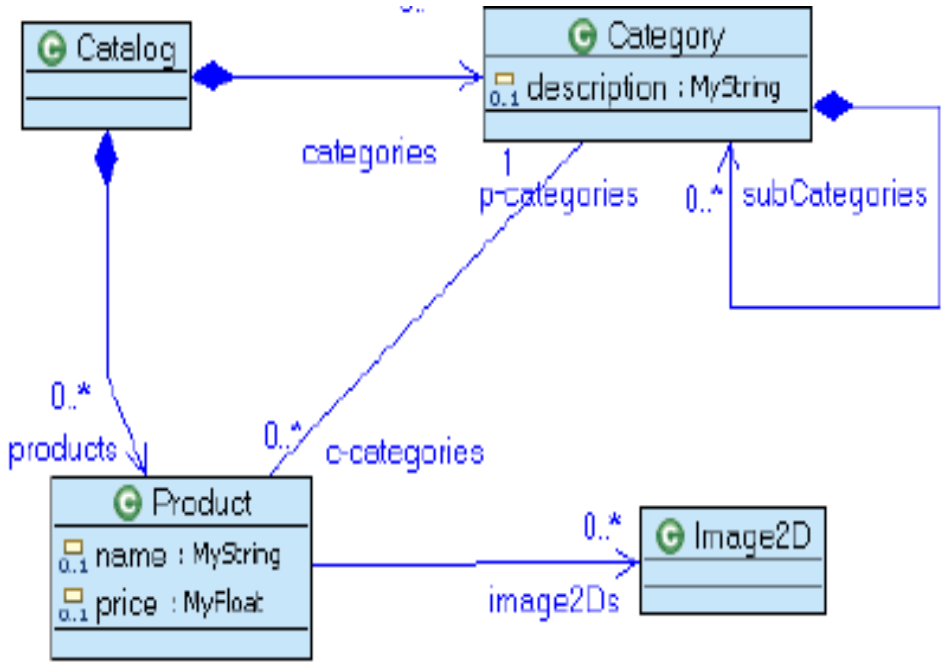
Catalogue de produits complet



Un catalogue (produit dérivé)



Utilise *selected*



Association Feature Model / MOF

Partie dynamique

- Non existante
- Il faudrait :
  - Annoter le code du programme (LdP de programmes)
  - Annoter le code Kermeta associé (LdP de modèles)
- On pourrait :
  - Adapter les travaux sur les diagrammes de séquences
- Mise en œuvre
  - Etendre la syntaxe et le moteur kermeta