

Génie logiciel

2. UML : Unified Modeling Language

MIF17 : semestre d'automne 2014
Julien MILLE

1

Bibliographie

- Cours de MIF17 de Yannick Prié 2011-2012
<http://liris.cnrs.fr/yannick.prie/ens/11-12/MIF17>
- Cours en ligne de Laurent Audibert
<http://laurent-audibert.developpez.com/Cours-UML/html/index.html>

2

Bibliographie

- UML 2 - Synthèse de cours et exercices corrigés
B. Charroux, A. Osmani, Y. Thierry-Mieg, Pearson Education, collection Synthex, 2005
- UML 2 - Guide de référence
G. Booch, I. Jacobson, J. Rumbaugh, Campus Press, Reference 2004



3

• Introduction

- UML est une unification ...
 - ... de nombreux langages de modélisation graphique orientée objet existants (OMT, Booch, Objectory, ...)
 - ... de diagrammes et de principes de modélisation à succès (classe, séquences, cas d'utilisation, ...)
- Défini par l'OMG (Object Management Group)
- Utilisé pour spécifier, visualiser, construire et documenter les systèmes d'information
- Modélise les choix et l'interprétation d'un système à construire
- S'adapte à plusieurs méthodes de développement, plusieurs domaines applicatifs

4

- Langage de modélisation visuel, formalisme graphique
- Non propriétaire, repose sur des accords au sein de la communauté informatique
- UML **n'est pas** une méthode de conception
 - Il ne contient pas de processus de développement étape par étape
 - Pour la réussite d'un développement logiciel, il est nécessaire de suivre une méthode de conception (Rational Unified Process (RUP), méthodes AGILE, ...) qui peut utiliser (partiellement) UML
 - Vous étudierez une ou plusieurs méthode(s) de conception en MIF16
- Version étudiée : UML 2

5

• Avant UML

- Nombreux formalisme de modélisation (plus ou moins) orientés objet
 - OOD : Object Oriented Design (Booch, 1991)
 - OMT : Object Modeling Technique (Rumbaugh, 1991)
 - OOSE : Object Oriented Software Engineering (Jacobson, 1992)
 - OOM : Object Oriented Merise (Bouzeghoub, Rochfeld, 1993)
 - ...
- Concepts proches, mais notations différentes

6

- Historique

- 1994
 - Tentative de normalisation de l'OMG (Object Management Group), sans effet
 - Rumbaugh (OMT) rejoint Booch (OOD) chez Rational Software, début des travaux d'unification
- 1995
 - Présentation de la version 0.8 de la méthode
 - Arrivée de Jacobson (OOSE) chez Rational
- 1996
 - Implication de l'OMG (sous pression des industriels pour favoriser l'interopérabilité des modèles)
 - Langage unifié UML 0.9

7

- 1997

- Sortie de UML 1.0 chez Rational
- UML 1.1 adopté par l'OMG comme standard officiel

- 1997 - 2003

- Adoption par les entreprises
- UML 1.1 à 1.5 : modifications/améliorations

- 2005

- Sortie de UML 2.0 : nouveaux diagrammes, changements importants au niveau du méta-modèle, pour permettre d'utiliser UML pour la programmation

- 2007 - aujourd'hui...

- 2007 : UML 2.1 ..., 2013 : UML 2.5 Beta 2

8

- Généralités sur le formalisme d'UML

- UML se décompose en plusieurs sous-ensembles :
 - **vues** : descriptions du système dépendantes d'un point de vue, qui peut être organisationnel, temporel, architectural, géographique, logique, etc.
 - **diagrammes** : éléments graphiques représentant le contenu des vues. Un même diagramme peut être utilisé par différentes vues.
 - Exemples : diagramme de classe, de séquence, de cas d'utilisation, etc.
 - **éléments** : briques de base des diagrammes, servant à modéliser et visualiser. Un même modèle peut être utilisé dans différents diagrammes.
 - Exemples : cas d'utilisation, classe, association, etc.

9

- Diagrammes

- En tout, treize diagrammes

- Hiérarchiquement dépendants et complémentaires
- Modélisent un système dans son ensemble ou différentes parties d'un système
- UML ne fournit pas de "séquence type", de "marche à suivre" dans l'utilisation des diagrammes (rappel : UML n'est pas une méthode, contrairement à RUP, par ex.)

- Trois types principaux de diagrammes :

- Diagrammes structurels ou statiques
- Diagrammes comportementaux
- Diagrammes d'interactions ou dynamiques

10

- Diagrammes structurels ou statiques

- **Diagramme de classes** : les **classes** et leurs **associations**, utilisées pour modéliser le système
- **Diagramme d'objets** : les **instances** des classes intervenant dans le système
- **Diagramme de composants** : **éléments logiciels** du système (fichiers, bibliothèques, bases de donnée, etc.)
- **Diagramme de déploiement** : **éléments matériels** (ordinateurs, réseaux, baies de stockage, etc.) et leurs interactions, implantation des composants dans ces éléments
- Diagramme de paquetages : regroupements logiques d'éléments et leurs dépendances
- Diagramme de structure composite (apport d'UML 2.x) : structures internes des classes

11

- Diagrammes comportementaux

- **Diagramme des cas d'utilisation** : identification des possibilités d'interactions entre le système et les agents extérieurs (lien avec les besoins, les fonctionnalités attendues)
- **Diagramme d'états-transitions** : description, sous forme de machine à états finis, du comportement d'un système ou de l'un de ses composants (ex: une classe)
- **Diagramme d'activités** : description, proche de l'organigramme, des étapes intervenant dans un cas d'utilisation précis

12

– Diagrammes d'interactions ou dynamiques

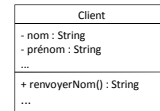
- **Diagramme de séquence** : représentation séquentielle du déroulement des traitements et des interactions **entre les éléments du système et/ou les acteurs**
- Diagramme de communication (apport d'UML 2.x) : représentation des échanges de messages **entre objets**
- Diagramme global d'interaction (apport d'UML 2.x) : enchaînements possibles des scénarios identifiés à l'aide de diagrammes de séquence
- Diagramme de temps (apport d'UML 2.x) : sémantique proche du diagramme de séquence + annotations avec intervalles de temps et instants

13

• Eléments

- Eléments de modélisation : définissent le modèle lui-même
 - Classes, objets, associations, commentaire, dépendances
- Eléments de visualisation : représentation graphique, comme :

- Conteneur de classe :



- Commentaire :



14

• Généralités sur les diagrammes

– Mots-clés

- Objectif : regrouper en "famille" des éléments similaires d'un modèle, afin de limiter le nombre d'éléments différents dans les diagrammes
- Notation : {mot-clé}
- Certains mots-clés sont prédéfinis. Exemples :
 - {abstract}, {xor}, {frozen}, ...

– Stéréotype

- Objectif : permettre la définition d'éléments, dérivés des éléments existants, avec des propriétés spécifiques adaptées à un usage spécialisé
- Certains stéréotypes sont prédéfinis. Exemples :
 - <<constructor>>, <<getter>>
 - Malgré son apparence, <<interface>> n'est pas un stéréotype

15

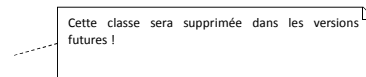
• Généralités sur les diagrammes

– Contrainte

- Relation sémantique impliquant un ou plusieurs élément(s)
- Définition de propriétés devant être vérifiées pour garantir la validité du système modélisé
- Notation : {contrainte} à côté du ou des élément(s) concerné(s)
- Exemples : {frozen}, {xor}, {x-y<10}

– Commentaire

- Annotation quelconque associée à un élément
- Notation :

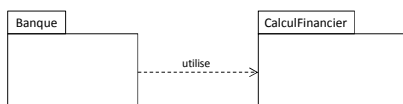


16

• Généralités sur les diagrammes

– Dépendance

- Relation d'utilisation (unidirectionnelle) d'un élément par un autre
- Exemple : un paquetage Banque à besoin d'un paquetage CalculFinancier



- La source dépend de la cible
- Une modification dans le paquetage cible peut avoir des conséquences sur le paquetage source

17

• Diagramme de classes

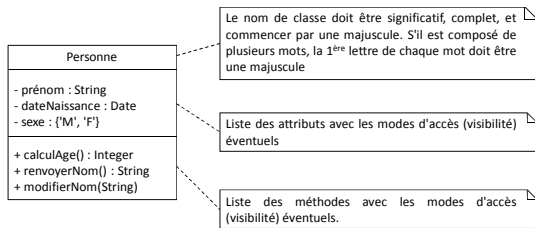
- Le plus connu et utilisé
- Représentation graphique des **classes** (pas des objets) et les relations entre ces classes
- Proche du modèle entité-relation utilisé en bases de données relationnelles

– Remarque sur le concept d'instance

- Instance = en UML, notion plus générale que l'objet. Un objet est une instance de classe, mais une instance peut être une instance de plusieurs autres éléments de modélisation. Par ex., un lien est une instance d'association

18

– Représentation d'une classe

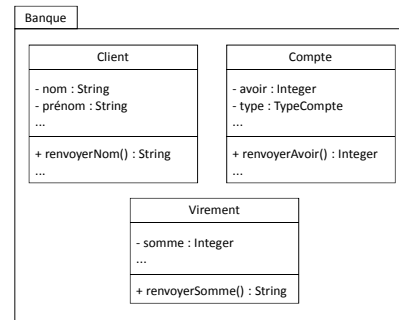


- Attributs et méthodes : **propriétés** (= membres)
- Classes considérées comme existantes pour les attributs "simples" (correspondent aux types de base) :
 - Integer, Real, String, Boolean, Character

19

– Paquetage (*package*)

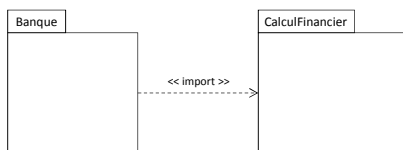
- Regroupement logique de classes



20

– Paquetage : dépendances

- Dépendance purement fonctionnelle : un paquetage A utilise (dépend d') un paquetage B si, par exemple, une classe de A comporte une méthode qui appelle une méthode de B



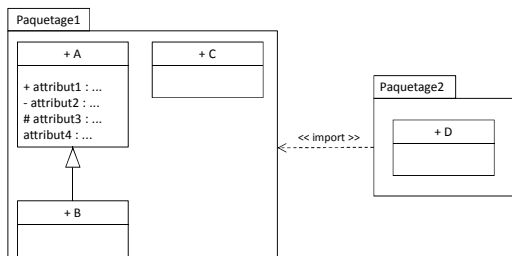
21

– Encapsulation

- Modificateurs d'accès (ou de visibilité) :
 - Caractère + : propriété ou classe **publique**, visible partout
 - Caractère - : propriété ou classe **privée**, visible uniquement dans la classe
 - Caractère # : propriété ou classe **protégée**, visible dans la classe et ses classes dérivées
 - Aucun caractère, ni mot-clé : propriété ou classe visible **uniquement dans le paquetage** où la classe est définie

22

– Encapsulation



- Attributs de A visibles (accessibles) :
 - dans B : attribut1, attribut3, attribut4
 - dans C : attribut1, attribut4
 - dans D : attribut1

23

– Syntaxe de déclaration

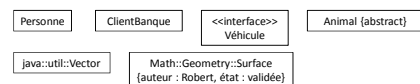
- Arguments entre [] = facultatifs

- Nom de classe

```

[<< stéréotype >>] [<< interface >>]
[NomPaquetage1::NomPaquetage2:: ... ::NomPaquetageN::]
NomClasse [ { [abstract], [auteur : valeur], [état : valeur], ... } ]
  
```

- Exemples :



24

– Syntaxe de déclaration

• Attribut

- Syntaxe :
`[+, -, #] [/] NomAttribut : NomClasse [EnsembleValeurs
 ['[' multiplicité ']'] [= valeur(s) initiale(s)] {propriété}`
- Exemples :
`+ nom : String (frozen)
 - rayon : Real = 0
 chambres : Chambre [1..200]
 typePersonnel : {'VACATAIRE', 'PERMANENT', 'STAGIAIRE'}`

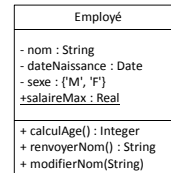
• Méthode

- Syntaxe :
`[+, -, #] NomMéthode ([Paramètres]) : [ClasseRetour] {propriété}`
- Syntaxe de la liste des paramètres
`[in|out|inout] NomParam1 : ClasseParam1, ...`
- Exemples :
`+ getNom() : String
 # calculMinMax(in tab : Table, out min : Integer, out max : Integer) : Boolean
 + executer(com : Commande) {abstract}`

25

– Attributs et méthodes de classe

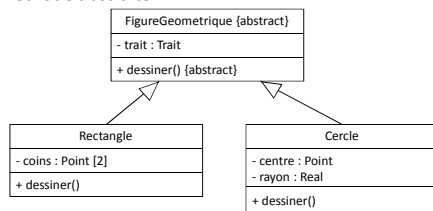
- Une propriété est dite "**de la classe**" si elle est propre à la classe (partagée par toutes ses instances) et non propre à un objet
- En Java et C++, de telles propriétés sont déclarées **static**
- Représentation : souligné



26

– Méthodes et classes abstraites, interfaces

- Une méthode est abstraite si son en-tête est connue mais pas la manière dont elle peut être **réalisée**
- Une classe est abstraite si elle contient au moins une méthode abstraite

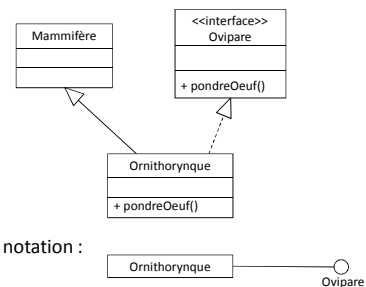


- La méthode abstraite peut être **réalisée** dans les classes dérivées (si une classe dérivée ne réalise pas toutes les méthodes abstraites, elle reste elle-même abstraite)

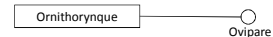
27

– Interface

- Par définition, toutes les méthodes de l'interface sont abstraites
- Une classe **réalise** une interface



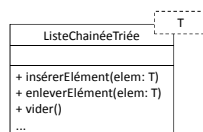
- Autre notation :



28

– Classe paramétrable

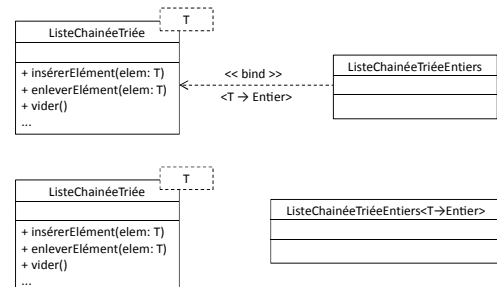
- Objectif : regrouper les comportements associés à la structure de la classe indépendamment des objets qu'elle contient
- Souvent utilisée pour les classes correspondant à des "collections" d'élément(s) : tableau dynamique, liste (simplement chaînée, doublement chaînée, triée ou non, avec dénombrement, ...), table de hashage, ensemble, ...
 - » le paramètre est alors le type d'objet contenu
- Disponible en C++ (patrons de classe = templates) et en Java (depuis la version 1.5)



29

– Classe paramétrable

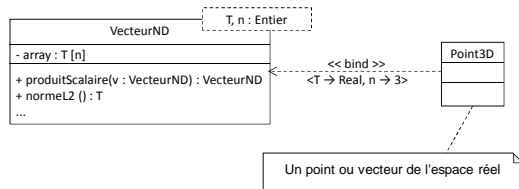
- Paramétrisation d'une classe paramétrable = instantiation des paramètres (*binding*). Deux notations possibles :



30

– Classe paramétrable

- Paramètres possibles
 - Aucun type spécifié : le paramètre est lui-même un type ! (ex: T)
 - Type spécifié : le paramètre est une variable
- Cas de plusieurs paramètres
 - Exemple : type des éléments et dimension



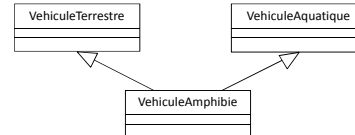
31

– Relation entre classes

- Relation d'héritage
 - But : factoriser les propriétés communes à certaines entités
 - Généralisation / spécialisation : si une classe B hérite d'une classe A
 - » A est une **généralisation** de B, B est une **spécialisation** de A
 - » A est une **classe mère**, une **superclasse**
 - » B est une **classe fille**, une **sous-classe**, une **classe dérivée**



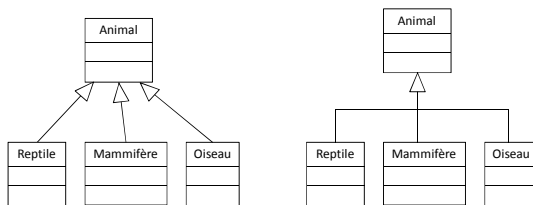
– Héritage multiple possible



32

• Relation d'héritage

- Une classe fille possède toutes les propriétés (avec les mêmes modes d'accès) que sa ou ses classe(s) mère(s), avec éventuellement des propriétés supplémentaires
- Une classe fille peut redéfinir certaines méthodes de sa ou ses classe(s) mère(s) (voir remarque sur classe abstraite)
- Lorsqu'une classe à plusieurs classes filles, deux notations équivalentes :



33

• Association

– Modèle général

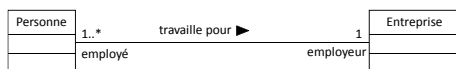


- Nom : forme verbale
- Sens de lecture de l'association indiqué éventuellement par ►
- Multiplicité : nombre (ou intervalle de nombres) d'instance(s) que l'association peut impliquer. Plus précisément :
 - » Multiplicité côté A : nombre (ou intervalle de nombres) d'instances de A pouvant être associée à **UNE** instance de B **simultanément** (et inversement)
 - » Aucune multiplicité = implicitement, 1
 - » Multiplicité quelconque : 0..* ou *
- Rôle : forme nominale décrivant le statut de la classe dans l'association. Peut contenir un mode d'accès

34

• Association

- Exemple : une personne travaille pour une et une seule entreprise. L'entreprise emploie au moins une personne. L'entreprise est l'employeur des personnes qui travaillent pour elle et une personne a un statut d'employé dans l'entreprise.



- Attention : différence avec le modèle entité-relation (E-R) utilisé en base de données : les multiplicités sont inversées par rapport aux cardinalités du modèle E-R !
- L'inversion des multiplicités est une erreur fréquente sur les premiers diagrammes de classe (solution = apprendre par cœur l'exemple précédent ?)

35

• Association avec sens de navigation

- Exemple : un polygone est défini par un ensemble de points jouant le rôle de sommets. Un point peut être sommet de plusieurs polygones. Les sommets du polygone ne sont accessibles que par la classe et ses descendants. On considère qu'il est inutile que les points aient un lien vers les polygones dans lesquels ils sont sommets.

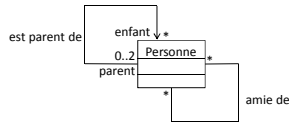


- Dans cet exemple, la navigation est possible uniquement du polygone vers les points : un polygone connaît les points qui lui servent de sommets, mais un point ne connaît pas les polygones dans lesquels il est sommet
- Incidence sur l'implémentation Java ou C++ : la classe Polygone contiendra une collection (au sens large : tableau dynamique, liste chaînée, ...) de pointeurs/références vers Point. La classe Point ne contiendra aucun attribut de type Polygone.

36

• Réflexivité, symétrie/asymétrie

- Exemple : une personne peut être parent d'autres personnes et enfant d'au plus deux personnes connues. Deux personnes peuvent être amies (on considère que l'amitié est réciproque).
- Les associations de parenté et d'amitié sont **réflexives** : elles associent la classe Personne avec elle-même



- La relation de parenté est asymétrique, donc orientée (si une instance A est parent d'une instance B, l'inverse ne peut pas être possible simultanément)
- La relation d'amitié est symétrique (=non-orientée). Il est inutile de spécifier des rôles dans ce cas.

37

• Association avec contraintes

- Sur un rôle d'une association seule, ou entre deux associations
- Utilisation du langage OCL (Object Constraint Language)
 - Vocabulaire : ordered, subset, xor, complete, disjoint, ...

Exemple 1 : les sommets dans un polygone sont ordonnés :



Exemple 2 : une personne travaille dans un service, et elle peut, en plus, gérer ce service. Plusieurs employés peuvent cogérer un service.



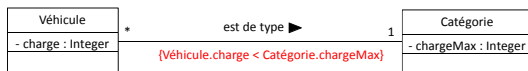
38

• Association avec contraintes

- Exemple 3 : une personne (sans distinction de classe entre étudiant et enseignant) est associée à une matière, soit parce qu'elle l'enseigne, soit parce qu'elle la suit (mais jamais les deux simultanément)



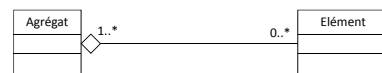
- Exemple 4 : un véhicule a des attributs propres, entre autres sa charge, et est d'une certaine catégorie. La charge maximale autorisée pour un véhicule dépend uniquement de la catégorie, pas du véhicule lui-même



39

• Agrégation

- Forme particulière d'association, représentant une inclusion structurelle ou comportementale
- Elle fait intervenir une classe agrégat et une classe élément
- Non nommée, structure d'arbre sous-jacente

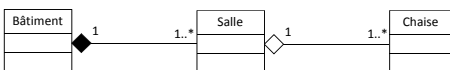


- Du point de vue de la classe jouant le rôle d'élément, l'agrégation n'est pas exclusive : une même instance élément peut être agrégée à plusieurs instances agrégat
- La vie de l'Elément **n'est pas liée** à celle de l'Agrégat : la création (resp. destruction) de l'Agrégat **n'entraîne pas** la création (resp. destruction) de ses Eléments

40

• Agrégation

- Exemple : à l'université, un bâtiment d'enseignement dispose d'un certain nombre de salles et de chaises. A un instant donné, une chaise est obligatoirement à l'intérieur d'une salle. Une chaise peut être déplacée dans une autre salle selon les besoins.

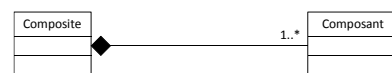


- Un bâtiment est donc composé d'une ou plusieurs salle(s), l'existence de la salle étant conditionnée par l'existence du bâtiment
 - relation de composition = cas particulier de l'agrégation (détaillé à la diapositive suivante)
- La salle est composée, entre autres, de chaises, mais une chaise n'est pas éternellement liée à la même salle (l'existence de la chaise n'est pas conditionnée par l'existence de la salle)
 - relation d'agrégation

41

• Composition

- Agrégation forte, notion d'inclusion structurelle
- Elle fait intervenir une classe Composite et une classe Composant
- Non nommée, structure d'arbre sous-jacente

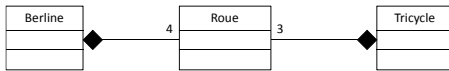


- Multiplicité =1, côté Composite (implicite)
- Lien exclusif : une instance Composant ne peut être liée qu'à une instance Composite
- La vie du Composant est liée à celle du Composite : la création (resp. destruction) du Composite entraîne la création (resp. destruction) de ses Composants

42

• Composition

- Exemple : on représente divers moyens de locomotion à roue.



- Remarque : la relation de composition est exclusive pour une instance de Composant, pas pour la classe Composant elle-même
- La composition peut être remplacée par l'ajout d'un attribut de classe Composant dans la classe Composite. Transformation de l'exemple précédent :

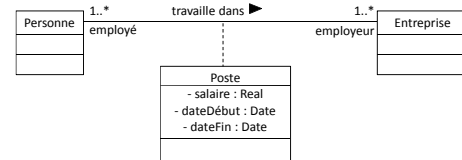


43

• Classe-association

- Une association peut être raffinée et avoir ses propres propriétés, qui ne sont disponibles dans aucune des classes qu'elle lie. Seules les classes peuvent avoir des attributs et méthodes. L'association devient alors une classe-association

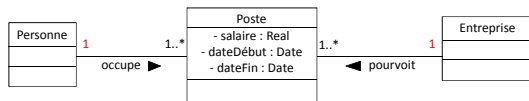
- Exemple : une personne à travaillé dans des entreprises, à des périodes données et avec un certain salaire.



44

• Classe-association

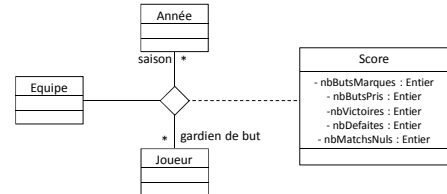
- Une classe-association peut être remplacée par une classe, en séparant la relation initiale
- Attention au changement de multiplicité par rapport à la version avec classe-association
- Transformation du diagramme précédent : une instance de Poste ne concerne qu'une instance de Personne et une instance d'Entreprise



45

• Association n-aire

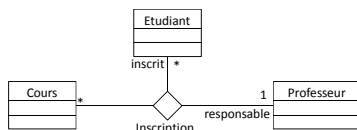
- Association entre au moins trois classes
- Chaque instance de l'association est un t-uple de valeurs provenant chacune de leurs classes respectives
- Exemple : une équipe réalise un score, pour une saison donnée avec un gardien de but particulier. Celui-ci peut faire l'objet d'un transfert dans le courant de la saison et obtenir un score différent selon les équipes.
- Ici, l'association n-aire est aussi une classe-association, car le score comporte plusieurs propriétés



46

• Association n-aire

- Multiplicités : chaque valeur de multiplicité correspond à une paire d'objets aux autres extrémités
- Exemple : un étudiant suit le cours d'un professeur responsable pendant un semestre. Il ne va pas choisir de suivre le même cours auprès de plusieurs responsables, mais il peut suivre plusieurs cours d'un même professeur
- Pour une paire (cours, étudiant), il n'existe qu'un professeur. Pour une paire (étudiant, professeur), il existe plusieurs cours. Pour une paire (cours, professeur), il existe plusieurs étudiants



47

- Mise en œuvre du diagramme de classes dans une méthode de développement




- Souvent plusieurs diagrammes de classes successifs lors du développement d'un système
- Le diagramme des classes n'est pas définitif et peut être raffiné plusieurs fois

- Diagramme de classes initial = **modèle du domaine**

- Contient uniquement les classes **métier** (=du domaine)
- Modèle des entités du « monde réel »
- Indépendant de toute implémentation (jamais de classes Java dans le modèle du domaine !)
- Seuls les attributs apparaissent, pas les méthodes
- La plupart des méthodes de développement logiciel distinguent l'activité d'**analyse** de celle de **conception** : le modèle du domaine est écrit durant l'**analyse**

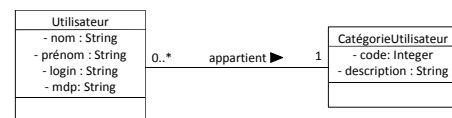
48

- Diagramme de classes participantes (ou classes « techniques »)
 - Réalisé durant l'**analyse**
 - Ajout des classes « techniques » liées aux choix de conception (interfaces utilisateur, patrons de conception, ...)
 - Trois types de classes :
 - Classes de type **métier** : reprises du modèle du domaine, éventuellement complétées
 - Classes de type **dialogue** : permettent les interconnexions entre l'IHM et ses utilisateurs. Ce sont typiquement les écrans proposés à l'utilisateur : les formulaires de saisie, les résultats de recherche... Elles proviennent directement de l'analyse de la maquette.
 - » Au moins un dialogue pour chaque association entre un acteur et un **cas d'utilisation** (voir diagramme des C.U.)
 - » En général, les dialogues vivent seulement le temps du déroulement du cas d'utilisation concerné

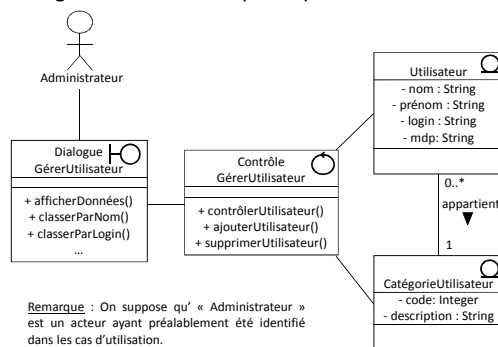
- Classes de type **contrôle** : font la jonction entre les dialogues et les classes métier, en permettant aux écrans de manipuler des informations détenues par un ou plusieurs objets métier.
- Associations entre les types de classe
 - Les dialogues ne peuvent être reliés qu'aux contrôles ou à d'autres dialogues (en général, associations unidirectionnelles)
 - Les classes métier ne peuvent être reliées qu'aux contrôles ou à d'autres classes métier
 - Les contrôles ont accès à tous les types de classes
- Notations
 - Métier : 
 - Dialogue : 
 - Contrôle : 

- Diagramme de classes d'implémentation
 - Réalisé durant la **conception**
 - Ajout des classes liées à l'implémentation dans un langage orienté objet
- Cas des classes conteneurs/collections
 - L'utilisation de classes de conteneurs/collections d'objet (`std::list`, `std::vector`, `ArrayList`, `PriorityQueue`, ...), que l'on peut faire apparaître dans le diagramme d'implémentation, est implicitement contenue dans les associations (multiplicités 0.* , 1..*) des diagrammes précédents (modèle du domaine, diagramme de classes participantes)
 - Dépend du niveau de détail choisi

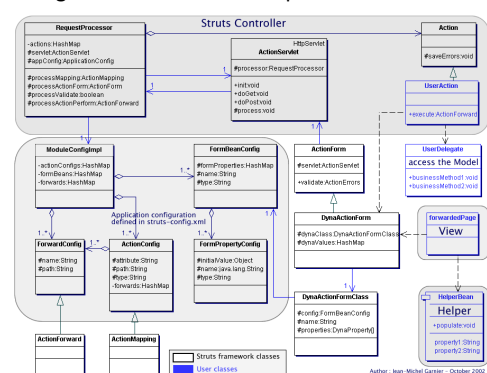
- Transformation du modèle du domaine en diagramme de classes participantes
 - Exemple : *pour accéder à un service via une interface, l'utilisateur doit s'authentifier à l'aide de son login et de son mot de passe. Il existe différentes catégories d'utilisateur*
 - Modèle du domaine :



- Diagramme de classes participantes



- Diagramme de classes d'implémentation



• Diagramme d'objets

– Objectifs

- Permet de représenter un instantané du système
- Utilisé pour illustrer une structure complexe, difficile à comprendre avec le diagramme de classes seul (association réflexive, association n-aire, ...)

– Instanciation du diagramme de classe

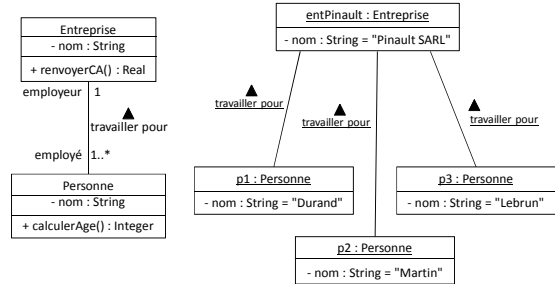
- Objet : instance de classe
- Lien : instance d'association

– Représentation des objets

- Noms des instances soulignés
- Deux compartiments
 - Nom de l'objet : Classe
 - Liste des attributs = Nom de l'attribut : Classe [=valeur]

55

– Exemple de diagramme de classes et diagramme d'objets associé :



56

• Diagramme de composants

– Un composant est une entité logique

- Exemples de composants :
 - Composant d'extension : plugin, codec, pilote
 - Composant de bibliothèque
 - Base de données

– Intérêt du composant

- Permet la réutilisabilité de tout ou partie de l'application
- Indépendance de son évolution vis-à-vis des applications qui l'utilisent.
- La classe, de par ses connexions figées (les associations avec les autres classes matérialisent des liens structurels), ne comporte pas la notion de réutilisabilité.

57

– Propriétés du composant

- Fournit un service bien précis
- Pour être réutilisable, les fonctionnalités qu'il encapsule doivent être cohérentes entre elles et génériques (par opposition à spécialisées)
- Comporte une ou plusieurs interfaces requises ou offertes (implémentées)
- Comportement interne masqué, seules ses interfaces sont visibles
- Réalisé par un ensemble de classes
- La seule contrainte pour pouvoir substituer un composant par un autre est de respecter les interfaces requises et offertes

58

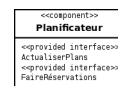
- Un composant étant un classeur structuré, on peut en décrire la structure interne. L'implémentation d'un composant peut être réalisée par d'autres composants, des classes ou des artefacts
- Les éléments d'un composant peuvent être représentés dans le symbole du composant, ou à côté en les reliant au composant par une relation de dépendance
- Pour montrer les instances des composants, un diagramme de déploiement doit être utilisé

59

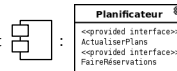
– Représentation

- Par un classeur structuré

– Stéréotypé « component » :



– Avec l'icône de composant :



- Accompagné de la représentation explicite de ses interfaces requises et offertes



60

• Représentation compacte

- Interfaces requises (représentées par un demi-cercle)
- Interfaces offertes (représentées par un cercle)



- Autre possibilité : le stéréotype « *component* » est rendu inutile par la représentation même du composant :



• Notion de port

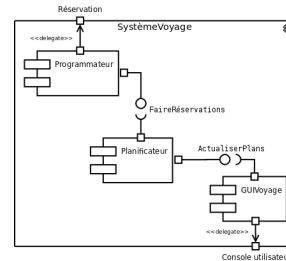
- Point de connexion entre le composant et son environnement
- Représenté par un petit carré sur le bord du classeur. On peut faire figurer le nom du port à proximité de sa représentation.
- Généralement, un port est associé à une interface requise ou offerte.



61

• Composant englobant

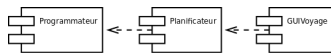
- Composant complexe contenant des sous-composants.
- A l'intérieur, un port peut être relié directement à un autre port situé sur la limite du composant englobant. Il peut être stéréotypé « *delegate* », et appelée *connecteur de délégation*.
- L'utilisation des ports permet de modifier la structure interne d'un classeur sans affecter les clients externe



62

• Diagramme de composants

- La relation de dépendance est utilisée dans les diagrammes de composants pour indiquer qu'un élément de l'implémentation d'un composant fait appel aux services offerts par les éléments d'implémentation d'un autre composant.



- Lorsqu'un composant utilise l'interface d'un autre composant, on peut utiliser la représentation de la figure précédente en imbriquant le demi-cercle d'une interface requise dans le cercle de l'interface offerte correspondante.

63

• Diagramme de déploiement

– Objectif

- Expliquer comment un système décrit statiquement dans un modèle sera concrètement déployé sur une architecture physique distribuée
- Décrire la disposition physique des ressources matérielles qui composent le système et montre la répartition des composants sur ces matériels
- Modéliser l'environnement d'exécution d'un projet

64

– Notion de nœud

• Rôle

- Chaque ressource est matérialisée par un nœud
- Le diagramme de déploiement précise comment les composants sont répartis sur les nœuds et quelles sont les connexions entre les composants ou les nœuds

• Représentation

- Nœud (à gauche) et instance de nœud (à droite)



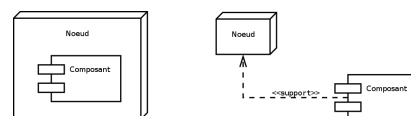
• Classeur structuré

- Peut être stéréotypé « *device* »
- Un nœud est un classeur et peut donc posséder des attributs (quantité de mémoire, vitesse du processeur, ...)

65

• Affectation d'un composant à un nœud

- Deux représentations possibles : soit placer le composant dans le nœud, soit les relier par une relation de dépendance stéréotypée « *support* » orientée du composant vers le nœud



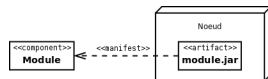
• Relations entre nœuds

- Support de communication physique, protocole (« Ethernet/LAN », « Bluetooth », « TCP/IP », ...)

66

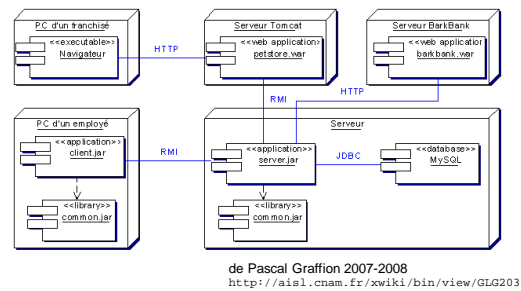
• Notion d'artefact

- Un **artefact** correspond à un élément concret existant dans le monde réel (document, exécutable, fichier, tables de bases de données, script, ...)
- Représentation par classeur stéréotypé « artefact »
- L'implémentation des modèles (classes, ...) se fait sous la forme de jeu d'artefacts. On dit qu'un artefact peut **manifester**, c'est-à-dire résulter et implémenter, un ensemble d'éléments de modèle (utilisation du stéréotype « manifest »)



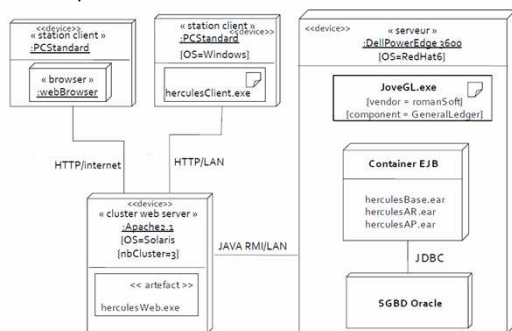
67

– Exemples



68

– Exemples



de Yannick Prié 2011-2012
http://liris.cnrs.fr/yannick.prie/ens/11-12/MIF17

69

• Traduction du diagramme de classes vers Java/C++

– Implémentation de certains concepts objet vue au premier cours

- Définition d'une classe
- Encapsulation : public, protected, private
- Membres statiques (attributs et méthodes de classe)
- Héritage
- Méthode abstraite (= virtuelle pure), classe abstraite, interface

– Implémentation restant à étudier

- Association (binaire/n-aire, classe-association, avec contrainte et navigabilité, agrégation, composition)

70

– Traduction directe du modèle du domaine !

- Les exemples d'association qui suivent apparaîtraient plutôt dans le modèle du domaine
- Si une méthode de développement était appliquée, des diagrammes des classes intermédiaires devraient être réalisés avant d'écrire le code (diagramme des classes d'implémentation...)

71

• Association binaire

- Multiplicité (1,1) ou (0..1, 0..1)



- L'association, si elle est navigable dans les deux sens, est implémentée par ajout dans la classe A (resp. B) d'un unique attribut de la classe B (resp. A).

A.java

```
public class A {
    private B roleB;
    ...

    public A() {
        roleB = null;
    }
    public void modifierRoleB(B b) {
        roleB = b;
    }
}
```

72

```

    public B renvoyerRoleB() {
        return roleB;
    }
    public void nom_association_AB(B b) {
        roleB = b;
        b.modifierRoleA(this);
    }
    ...
};

```

- En plus des accesseurs et mutateurs, ajout d'une méthode qui maintient la cohérence de l'association (si une instance de A est liée à une instance de B, alors l'instance de B est obligatoirement liée à l'instance de A).
- Utilisation de la référence null pour symboliser l'absence d'instance associée

B.java

```

public class B {
    private A roleA;
    ...
    public B() {
        roleA = null;
    }
}

```

73

```

    public void modifierRoleA(A a) {
        roleA = a;
    }
    public A renvoyerRoleA() {
        return roleA;
    }
    public void nom_association_AB_inverse(A a) {
        roleA = a;
        a.modifierRoleB(this);
    }
    ...
};

```

Ailleurs.java

```

...
A a1=new A(), a2=new A();
B b1=new B(), b2=new B();

a1.nom_association_AB(b1);
a2.nom_association_AB(b2);

```

74

A.hpp

```

class A
{
    private:
        B *pRoleB;
        ...
    public:
        A()
        {
            pRoleB = NULL;
        }

        void modifierRoleB(B &b)
        {
            pRoleB = &b;
        }

        B *renvoyerRoleBPtr() {
            return pRoleB;
        }
}

```

75

```

    void nom_association_AB(B &b) {
        pRoleB = &b;
        b.modifierRoleA(*this);
    }
    ...
};

```

B.hpp

```

class B {
    private:
        A *pRoleA;
        ...
    public:
        B()
        {
            pRoleA = NULL;
        }

        void modifierRoleA(A &a)
        {
            pRoleA = &a;
        }
}

```

76

```

A *renvoyerRoleAPtr()
{
    return pRoleA;
}
void nom_association_AB_inverse(A &a)
{
    pRoleA = &a;
    a.modifierRoleB(*this);
}
...
};

```

Ailleurs.cpp

```

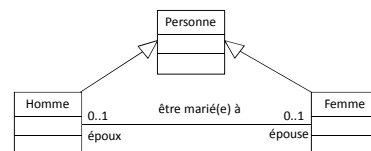
...
A a1, a2;
B b1, b2;

a1.nom_association_AB(b1);
a2.nom_association_AB(b2);

```

77

- Association binaire
 - Exemple avec deux sous-classes d'une classe mère commune



Homme.java

```

public class Homme extends Personne {
    private Femme epouse;
    ...

    public Homme() {
        epouse = null; // Célibataire par défaut
    }
    public void modifierEpouse(Femme f) {
        epouse = f;
    }
}

```

78

```

    public Femme renvoyerEpouse() {
        return epouse;
    }
    public void marrierA(Femme f) {
        epouse = f;
        f.modifierEpoux(this);
    }
};

Femme.java
public class Femme extends Personne {
    private Homme epoux;
    ...

    public Femme() {
        epoux = null; // Célibataire par défaut
    }
    public void modifierEpoux(Homme h) {
        epoux = h;
    }
    public Femme renvoyerEpouse() {
        return epoux;
    }
}

```

79

```

    public void marrierA(Homme h) {
        epoux = h;
        h.modifierEpouse(this);
    }
};

Ailleurs.java
Homme h1 = new Homme();
Homme h2 = new Homme();

Femme f1 = new Femme();
Femme f2 = new Femme();

h1.marrierA(f1);
h2.marrierA(f2);

h1.marrierA(f2); // h2 est encore marié à f2 !
f1.marrier(h2);

```

80

```

Homme.hpp
class Homme : public Personne
{
    private:
        Femme *pEpouse;
        ...
    public:
        Homme()
        {
            pEpouse = NULL; // Célibataire par défaut
        }
        void modifierEpouse(Femme &f)
        {
            pEpouse = &f;
        }
        Femme *renvoyerEpousePtr()
        {
            return pEpouse;
        }
}

```

81

```

    void marierA(Femme &f)
    {
        pEpouse = &f;
        f.modifierEpoux(*this);
    }
    ...
};

Femme.hpp
class Femme : public Personne
{
    private:
        Homme *pEpoux;
        ...
    public:
        Femme()
        {
            pEpoux = NULL; // Célibataire par défaut
        }
}

```

82

```

void modifierEpoux(Homme &h)
{
    pEpoux = &h;
}
Homme *renvoyerEpouxPtr()
{
    return pEpoux;
}
void marierA(Homme &h)
{
    pEpoux = &h;
    h.modifierEpouse(*this);
}
...
};

```

83

```

Ailleurs.cpp

Homme h1, h2;
Femme f1, f2;

h1.marierA(f1);
h2.marierA(f2);

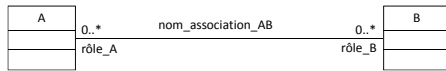
h1.marierA(f2); // h2 est encore marié à f2 !
f1.marierA(h2);

```

84

- Association binaire

- Multiplicité (0..*, 0..*)



- L'association, si elle est navigable dans les deux sens, est implémentée par ajout dans la classe A (resp. B) d'un attribut correspondant à une "collection" (au sens large) d'éléments. Chaque élément représente un lien vers une instance de la classe B (resp. A).
- Collection : peut être implémentée en tableau dynamique, liste chaînée, ...
 - » Java : Vector, List, ArrayList, Set, ...
 - » C++ : std::vector, std::list, ...
- Lien : référence en Java, pointeur en C++
- Vérification qu'une même instance n'est pas liée plusieurs fois

85

A.java

```
public class A {
    private Vector<B> listRoleB;
    ...

    public A() {
        listRoleB = new Vector<B>();
        ...
    }

    public int renvoyerNbRoleB() {
        return listRoleB.size();
    }

    public void ajouterRoleB(B b) {
        if (estRoleA(b)==false)
            listRoleB.add(b);
    }

    public boolean estRoleA(B b) {
        return listRoleB.contains(b);
    }
}
```

86

```
public B renvoyerRoleBIndice(int index) {
    return listRoleB.elementAt(index);
}
public void nom_association_AB(B b) {
    if (estRoleA(b)==false) {
        listRoleB.add(b);
        b.ajouterRoleA(this);
    }
}
... // + méthodes pour vider listRoleB, etc.
};
```

B.java

```
public class B {
    private Vector<A> listRoleA;
    ...

    public B() {
        listRoleA = new Vector<A>();
        ...
    }

    public int renvoyerNbRoleA() {
        return listRoleA.size();
    }
}
```

87

```
public void ajouterRoleA(A a) {
    if (estRoleB(a)==false)
        listRoleA.add(a);
}
public boolean estRoleB(A a) {
    return listRoleA.contains(a);
}

public A renvoyerRoleAIndice(int index) {
    return listRoleA.elementAt(index);
}
public void nom_association_AB_inverse(A a) {
    if (estRoleB(a)==false) {
        listRoleA.add(a);
        a.ajouterRoleB(this);
    }
}
... // + méthodes pour vider listRoleA, ...
};
```

88

A.hpp

```
class A
{
private:
    std::vector<B *> listRoleB;
    ...

public:
    A() {...}
    int renvoyerNbRoleB() const
    {
        return listRoleB.size();
    }
    void ajouterRoleB(B &b)
    {
        if (estRoleA(b)==false)
            listRoleB.push_back(&b);
    }
    bool estRoleA(B &b) const
    {
        std::vector<B *>::const_iterator it;
        bool trouve = false;

```

89

```
for (it=listRoleA.begin();
     it!=listRoleA.end() && trouve=false; it++)
{
    if (*it==&b) // Comparaison d'adresses
        trouve = true;
}
return trouve;
}
B *renvoyerRoleBIndicePtr(int index)
{
    return listRoleB[index];
}
void nom_association_AB(B &b)
{
    if (estRoleA(b)==false)
    {
        listRoleB.push_back(&b);
        b.ajouterRoleA(*this);
    }
}
... // + méthodes pour vider listRoleB, etc.
};
```

90

B.hpp

```

class B
{
private:
    std::vector<A *> listRoleA;
    ...

public:
    B() {...}
    int renvoyerNbRoleA() const
    {
        return listRoleA.size();
    }
    void ajouterRoleA(A &a)
    {
        if (estRoleB(a)==false)
            listRoleA.push_back(&a);
    }
    bool estRoleB(A &a) const
    {
        std::vector<A *>::const_iterator it;
        bool trouve = false;

```

91

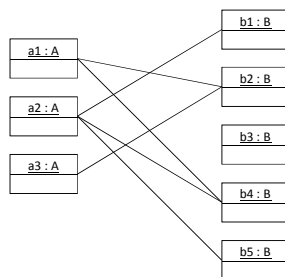
```

        for (it=listRoleA.begin();
             it!=listRoleA.end() && trouve=false; it++)
        {
            if (*it==&a) // Comparaison d'adresses
                trouve = true;
        }
        return trouve;
    }
    A *renvoyerRoleBIndicePtr(int index)
    {
        return listRoleA[index];
    }
    void nom_association_AB_inverse(A &a)
    {
        if (estRoleB(a)==false)
        {
            listRoleA.push_back(&a);
            a.ajouterRoleB(*this);
        }
    }
    ... // + méthodes pour vider listRoleA, etc.
};

```

92

- Diagramme d'objet illustrant des instances liées par des instances d'association binaire...



93

- ... et implémentation correspondante :

Ailleurs.java

```

...
A a1, a2, a3;
B b1, b2, b3, b4, b5;

a1=new A(); a2=new A(); a3=new A();
b1=new B(); b2=new B(); ... b5=new B();

// Une solution parmi d'autres :
a1.nom_association_AB(b2);
a1.nom_association_AB(b4);

a2.nom_association_AB(b1);
b4.nom_association_AB_inverse(a2);
a2.nom_association_AB(b5);

b2.nom_association_AB_inverse(a3);

```

94

Ailleurs.hpp

```

...
A a1, a2, a3;
B b1, b2, b3, b4, b5;

// Une solution parmi d'autres :
a1.nom_association_AB(b2);
a1.nom_association_AB(b4);

a2.nom_association_AB(b1);
b4.nom_association_AB_inverse(a2);
a2.nom_association_AB(b5);

b2.nom_association_AB_inverse(a3);

```

95

- Association binaire

— Exemple de multiplicité (0..*, 0..*) : un étudiant peut suivre plusieurs UE. Une UE est suivie par plusieurs étudiants. Le nombre d'UE est fini mais suffisamment grand pour que l'on puisse considérer * comme borne supérieure de la multiplicité côté UE. Idem pour le nombre d'étudiants. Dans cet exemple, on ne cherche pas à représenter à quel semestre ou avec quel intervenant l'étudiant suit une UE :



96

Etudiant.java

```
public class Etudiant {
    private Vector<UE> listUESuivies;
    ...

    public Etudiant() {
        listUESuivies = new Vector<UE>();
        ...
    }

    public int renvoyerNbUESuivies() {
        return listUESuivies.size();
    }

    public void ajouterUE(UE u) {
        if (suitDeja(u)==false) {
            listUESuivies.add(u);
        }
    }

    public boolean suitDeja(UE u) {
        return listUESuivies.contains(u);
    }
}
```

97

```
public UE renvoyerUESuivieIndice(int index) {
    return listUESuivies.elementAt(index);
}

public void suivre(UE u) {
    if (suitDeja(u)==false) {
        listUESuivies.add(u);
        u.ajouterEtudiant(this);
    }
}

... // + méthodes figurant dans le diagramme
};
```

UE.java

```
public class UE {
    private Vector<Etudiant> listEtudiants;
    ...
    public UE() {
        listEtudiants = new Vector<Etudiant>();
        ...
    }

    public int renvoyerNbEtudiants() {
        return listEtudiants.size();
    }
}
```

98

```
public void ajouterEtudiant(Etudiant etu) {
    if (estDejaSuivie(etu)==false)
        listEtudiants.add(etu);
}

public boolean estDejaSuivie(Etudiant etu) {
    return listEtudiants.contains(etu);
}

public Etudiant renvoyerEtudiantIndice(int index) {
    return listEtudiants.elementAt(index);
}

public void inscrire(Etudiant etu) {
    if (estDejaSuivie(etu)==false) {
        listEtudiants.add(etu);
        etu.suivre(this);
    }
}

... // + méthodes figurant dans le diagramme
};
```

99

Etudiant.hpp
class Etudiant

```
{
    private:
        std::vector<UE *> listUESuivies;
        ...

    public:
        Etudiant() {...}
        int renvoyerNbUESuivies() const
        {
            return listUESuivies.size();
        }
        void ajouterUE(UE &u)
        {
            if (suitDeja(u)==false)
                listUESuivies.push_back(&u);
        }
        bool suitDeja(UE &u) const
        {
            std::vector<UE *>::const_iterator it;
            bool trouve = false;
        }
}
```

100

```
for (it=listUESuivies.begin();
     it!=listUESuivies.end() && trouve=false; it++)
{
    if (*it==&u) // Comparaison d'adresses
        trouve = true;
}
return trouve;
}

UE *renvoyerUESuivieIndicePtr(int index)
{
    return listUESuivies[index];
}

void suivre(UE &u)
{
    if (suitDeja()==false)
    {
        listRoleB.push_back(&b);
        b.ajouterRoleA(*this);
    }
}

... // + méthodes figurant dans le diagramme
};
```

101

UE.hpp

```
class UE
{
    private:
        std::vector<Etudiant *> listEtudiants;
        ...

    public:
        UE() {...}
        int renvoyerNbEtudiants() const
        {
            return listEtudiants.size();
        }
        void ajouterEtudiant(Etudiant &etu)
        {
            if (estDejaSuivie(etu)==false)
                listEtudiants.push_back(&etu);
        }
        bool estDejaSuivie(Etudiant &etu) const
        {
            std::vector<Etudiant *>::const_iterator it;
            bool trouve = false;
        }
}
```

102

```

        for (it=listEtudiants.begin();
             it!=listEtudiants.end() && trouve=false; it++)
        {
            if (*it==&etu) // Comparaison d'adresses
                trouve = true;
        }
        return trouve;
    }
    Etudiant *renvoyerEtudiantIndicePtr(int index)
    {
        return listEtudiants[index];
    }
    void inscrire(Etudiant &etu)
    {
        if (estDejaSuivie(etu)==false)
        {
            listEtudiants.push_back(&etu);
            etu.ajouterUE(*this);
        }
    }
    ... // + méthodes pour vider listRoleA, etc.
};

```

103

- Dans la classe Etudiant (resp. UE), la liste listUESuivies (resp. listEtudiants) n'est pas propriétaire des instances d'UE (resp. Etudiant). Les listes contiennent des **références (Java)/pointeurs (C++)** vers des instances qui sont supposées **avoir été créées auparavant** !
- L'action d'inscrire un étudiant à une UE ne crée pas de nouvelle instance d'Etudiant ni d'UE (car l'association n'est pas une relation de composition dans le cas présent)
- A partir des implémentations des associations (0..1, 0..1) et (0..*, 0..*), on peut déduire aisément l'implémentation d'une association (0..1, 0..*)

104

- Association avec sens de navigation

- Dans cet exemple, la navigation est possible uniquement du polygone vers les points : un polygone connaît les points qui lui servent de sommets, mais un point ne connaît pas les polygones dans lesquels il est sommet.



- Incidence sur l'implémentation Java ou C++ : dans la classe Point, aucun attribut ne fait intervenir la classe Polygon

105

Polygone.java

```

public class Polygone {
    protected Vector<Point> listSommets;
    ...

    public Polygone() {
        listSommets = new Vector<Point>();
    }
    ...

    public int renvoyerNbSommets() {
        return listSommets.size();
    }

    public void ajouterSommets(Point pt) {
        listSommets.add(pt);
    }

    public boolean aPourSommets(Point pt) {
        return listSommets.contains(pt);
    }
}

```

106

```

    public UE renvoyerSommetsIndice(int index) {
        return listSommets.elementAt(index);
    }
    ... // + méthodes figurant dans le diagramme
};

```

Point.java

```

public class Point {
    public int x, y;
    ...
    public Point() {x=0; y=0;}
};

```

Polygone.hpp

```

class Polygone
{
protected:
    std::vector<Point *> listSommets;
    ...

public:
    Polygone() {...}
}

```

107

```

    int renvoyerNbSommets() {
        return listSommets.size();
    }
    void ajouterSommets(Point &pt) {
        listSommets.add(&pt);
    }
    ...
};

```

Point.hpp

```

class Point
{
public:
    int x, y;

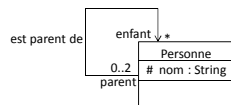
    Point() {x=0; y=0;}
    ...
};

```

108

- Association réflexive

- Exemple : une personne peut être parent d'autres personnes et enfant d'au plus deux personnes connues.



- La relation de parenté est asymétrique : si une instance A est parent d'une instance B, l'inverse ne peut pas être possible simultanément.
- Une personne ne peut pas être son propre parent
- Vérifications effectués lors de la création des liens de parenté

109

Personne.java

```

public class Personne {
    protected String nom;
    protected Vector<Personne> listEnfants;
    protected Personne parent[];

    public Personne(String s) {
        nom = new String(s);
        listEnfants = new Vector<Personne>();
        parent = new Personne[2];
        parent[0] = null;
        parent[1] = null;
    }

    public boolean estParentDe(Personne p) {
        return listEnfants.contains(p);
    }
}

```

110

```

public void devenirParentDe(Personne p) {
    if (p.estParentDe(this)==false && p!=this) {
        listEnfants.add(p);
        if (p.parent[0]==null)
            p.parent[0] = this;
        else if (p.parent[1]==null)
            p.parent[1] = this;
        else {
            // écrase parent[0]
            p.parent[0].listEnfants.remove(p);
            p.parent[0] = this;
        }
    }
}
};

```

111

Personne.hpp

```

class Personne
{
protected:
    std::string nom;
    std::vector<Personne *> listEnfants;
    Personne *parent[2];

public:
    Personne(const std::string &s)
    {
        nom = s;
        parent[0] = NULL;
        parent[1] = NULL;
    }

    bool estParentDe(Personne &p) const
    {
        std::vector<Personne *>::const_iterator it;
        bool trouve = false;

        for (it=listEnfants.begin();
             it!=listEnfants.end() && trouve=false; it++)
        {

```

112

```

        if (*it==&p) // Comparaison d'adresses
            trouve = true;
    }
    return trouve;
}

void devenirParentDe(Personne &p)
{
    if (p.estParentDe(*this)==false && &p!=this)
    {
        listEnfants.push_back(&p);
        if (p.parent[0]==NULL)
            p.parent[0] = this;
        else if (p.parent[1]==NULL)
            p.parent[1] = this;
        else { // écrase parent[0]
            ... // Retire &p de
                // p.parent[0]->listEnfants
            p.parent[0] = this;
        }
    }
}
};

```

113

- Contraintes

- Exemple 1 : les sommets dans un polygone sont ordonnés :



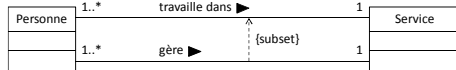
- La notion d'ordre est déjà présente dans la plupart des collections d'objet :

- » Un Vector (Java) ou un std::vector (C++) : accès via un itérateur ou un indice
- » Une LinkedList (Java) ou une std::list (C++) : accès via un itérateur

114

• Contraintes

- Contrainte d'inclusion : *une personne travaille dans un service, et elle peut, en plus, gérer ce service. Plusieurs employés peuvent cogérer un service*



- Contrainte d'exclusivité : *une personne (sans distinction de classe entre étudiant et enseignant) est associée à une matière, soit parce qu'elle l'enseigne, soit parce qu'elle la suit (mais jamais les deux simultanément)*



- Ces contraintes ne peuvent pas être représentées par le type de collection. Elles doivent être maintenues lors de la création des liens.

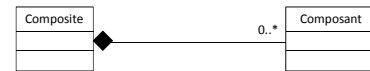
115

• Agrégation

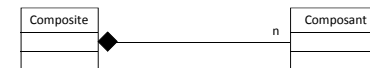
- Implémentation similaire à une association binaire

• Composition

- Inclusion structurelle d'un composant dans un composite



- La vie du Composant est liée à celle du Composite : la création (resp. destruction) du Composite entraîne la création (resp. destruction) de ses Composants
- La composition peut être implémentée par l'ajout d'un attribut de classe Composant dans la classe Composite.
- Cas particulier : multiplicité fixe côté Composant



116

Composant.java

```

public class Composant {
    private Composite comp;
    ...

    public Composant() {comp = null;}
    void modifierComposite(Composite c) {comp = c;}
    ...
};

```

Composite.java

```

public class Composite {
    private Composant tabComposants[];
    ...

    public Composite() {
        tabComposants = new Composant[n];
        for (int i=0; i<n; i++) {
            tabComposants[i] = new Composant();
            tabComposants[i].modifierComposite(this);
        }
    }
    ...
}

```

117

Composant.hpp

```

class Composant
{
private:
    Composite *pComp;
    ...

public:
    Composant() {pComp = NULL;}
    void modifierComposite(Composite &c) {pComp = &c;}
    ...
};

```

Composite.hpp (version 1)

```

class Composite
{
private:
    Composant tabComposants[n];
    ...

public:

```

118

```

Composite()
{
    for (int i=0; i<n; i++)
        tabComposants[i].modifierComposite(*this);
    ...
};

```

Composite.hpp (version 2)

```

class Composite
{
private:
    Composant *tabPtrComposants[n];
    ...

public:
    Composite()
    {
        for (int i=0; i<n; i++)
        {
            tabPtrComposants[i] = new Composant();
            tabPtrComposants[i]->modifierComposite(*this);
        }
    }
}

```

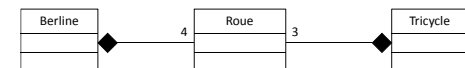
119

```

~Composite()
{
    for (int i=0; i<n; i++)
        delete tabPtrComposants[i];
    ...
};

```

- Si une classe est Composante de plusieurs Composites, l'attribut Composite (référence ou pointeur) dans la classe Composant ne peut être conservé. Exemple :

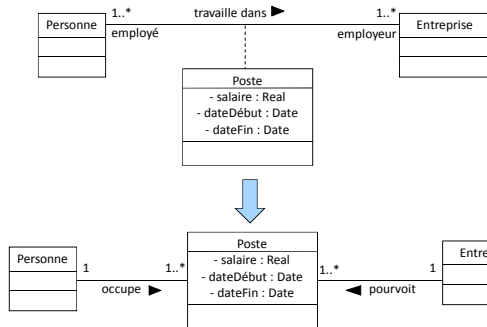


- Si la multiplicité n'est pas fixe, remplacement du tableau de taille fixe par un tableau dynamique ou autre collection (Vector, LinkedList, ..., std::vector, std::list, ...)

120

- Classe-association

- Implémentation de la classe-association par ajout d'une classe



121

Personne.java

```
public class Personne {
    private Vector<Poste> listPostesOccupes;
    ...

    public Personne() {
        listPostesOccupes = new Vector<Poste>();
    }
    ...

    public int renvoyerNbPostesOccupes() {
        return listPostesOccupes.size();
    }

    public void ajouterPoste(Poste pos) {
        if (aOccupePoste(pos)==false)
            listPostesOccupes.add(pos);
    }

    public boolean aOccupePoste(Poste pos) {
        return listPostesOccupes.contains(pos);
    }
}
```

122

```
public Poste renvoyerPosteOccupeIndice(int index) {
    return listPostesOccupes.elementAt(index);
}
public void occuperPoste(Poste pos) {
    if (aOccupePoste(pos)==false) {
        listPostesOccupes.add(pos);
        pos.modifierPersonne(this);
    }
}
...
};
```

Entreprise.java

```
public class Entreprise {
    private Vector<Poste> listPostesPourvus;
    ...

    public Entreprise() {
        listPostesPourvus = new Vector<Poste>();
    }
    ...

    public int renvoyerNbPostesPourvus() {
        return listPostesPourvus.size();
    }
}
```

123

```
public void ajouterPoste(Poste pos) {
    if (aPourvuPoste(pos)==false)
        listPostesPourvus.add(pos);
}

public boolean aPourvuPoste(Poste pos) {
    return listPostesPourvus.contains(pos);
}

public Poste renvoyerPostePourvuIndice(int index) {
    return listPostesPourvus.elementAt(index);
}

public void pourvoir(Poste pos) {
    if (aPourvuPoste(pos)==false) {
        listPostesPourvus.add(pos);
        pos.modifierEntreprise(this);
    }
}
...
};
```

124

Poste.java

```
public class Poste {
    private Personne employe;
    private Entreprise employeur;
    private Date dateDebut, dateFin;
    private float salaire;

    public Poste() {
        dateDebut = new Date();
        dateFin = new Date();
    }

    public void modifierEmploye(Personne p) {
        employe = p;
    }

    public Personne renvoyerEmploye() {
        return employe;
    }

    public void modifierEmployeur(Entreprise ent) {
        employeur = ent;
    }

    public Entreprise renvoyerEmployeur() {
        return employeur;
    }
    ...
}
```

125

Personne.hpp

```
class Personne
{
private:
    std::vector<Poste *> listPostesOccupes;
    ...
public:
    Personne() {...}

    int renvoyerNbPostesOccupes()
    {
        return listPostesOccupes.size();
    }
    void ajouterPoste(Poste &pos)
    {
        if (aOccupePoste(pos)==false)
            listPostesOccupes.push_back(&pos);
    }
    bool aOccupePoste(Poste &pos)
    {
        ... // Recherche de &pos dans listPostesOccupes
        return ...;
    }
}
```

126

```

Poste *renvoyerPosteOccupePtrIndice(int index)
{
    return listPostesOccupes[index];
}
void occuperPoste(Poste &pos)
{
    if (aOccupePoste(pos)==false)
    {
        listPostesOccupes.push_back(&pos);
        pos.modifierPersonne(*this);
    }
}
... // + méthodes figurant dans le diagramme
};

```

Entreprise.hpp

```

class Entreprise
{
private:
    std::vector<Poste *> listPostesPourvus;
    ...
public:
    Entreprise() {...}

```

127

```

int renvoyerNbPostesPourvus()
{
    return listPostesPourvus.size();
}
void ajouterPoste(Poste &pos)
{
    if (aPourvuPoste(pos)==false)
        listPostesPourvus.push_back(&pos);
}
bool aOccupePoste(Poste &pos)
{
    ... // Recherche de &pos dans listPostesPourvus
    return ...;
}
Poste *renvoyerPostePourvuPtrIndice(int index)
{
    return listPostesPourvus[index];
}

```

128

```

void pourvoir(Poste &pos)
{
    if (aPourvuPoste(pos)==false)
    {
        listPostesPourvus.push_back(&pos);
        pos.modifierEntreprise(*this);
    }
}
...
};

```

Poste.hpp

```

class Poste
{
private:
    Personne *pEmploye;
    Entreprise *pEmployeur;
    Date dateDebut, dateFin;
    float salaire;
public:
    Poste() { pEmployer=NULL; pEmployeur=NULL; }

```

129

```

void modifierEmploye(Personne &p)
{
    pEmploye = &p;
}

Personne *renvoyerEmployePtr()
{
    return pEmploye;
}

void modifierEmployeur(Entreprise &ent)
{
    pEmployeur = &ent;
}

Entreprise *renvoyerEmployeurPtr()
{
    return pEmployeur;
}
...
};

```

– Même principe pour les associations n-aire.

130