Eléments de **Génie Logiciel**

Cours 2

Pr. Jean-Marc Jézéquel
IRISA – Université de Rennes
Campus de Beaulieu
F-35042 Rennes Cedex
e-mail : jezequel@irisa.fr
http://people.irisa.fr/Jean-Marc.Jezequel
X @jmjezequel

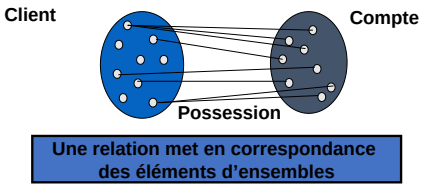
5 – Liens entre Objets

Et associations entre classes

Les objets ne sont pas définis isolément

- 2 catégories
  - Liens entre objets
    - Marie possède les comptes en banque n°12345 et n° 67890
    - Cette voiture est composée du châssis n° XX564 et des 4 roues AvG, AvD, ArG, ArD
    - => Associations entre classes
      - Une Personne possède un ou plusieurs Comptes en banque
      - Une Voiture est composée de 1 Châssis et de 4 Roues
  - Classification
    - Médor est un Chien, il est donc aussi un Mammifère
      - Médor appartient à l'ensemble des Chiens, qui est contenu dans l'ensemble des Mammifères
    - => Héritage entre classes
      - Un Chien est une sorte de Mammifère, qui est une sorte d'Animal

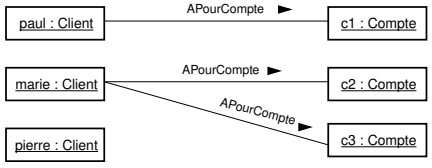
Vue ensembliste d'une relation : Graphe de la relation



Une relation met en correspondance des éléments d'ensembles

Liens (entre objets)

Un **lien** indique une connexion entre deux objets

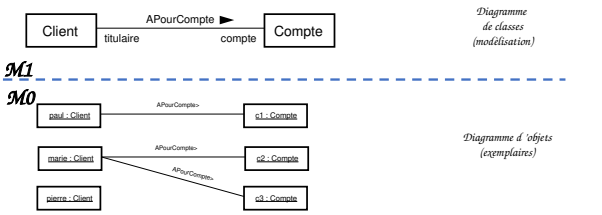


Note de style :

- les noms des liens sont des formes verbales et commencent par une majuscule
  ► indique le sens de la lecture (ex: « paul APourCompte c1 »)

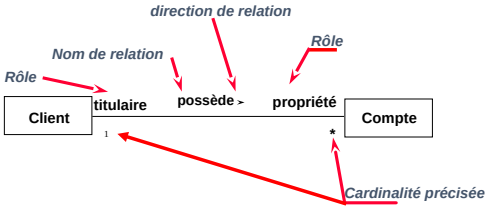
Associations (entre classes)

Une **association** décrit un ensemble de liens de même "sémantique"



Université de Rennes

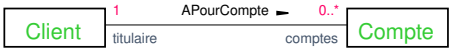
Représentation des associations : direction, rôle, cardinalité



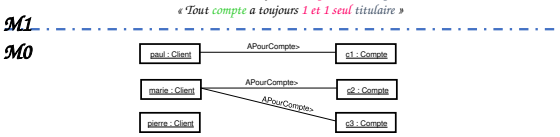
Université de Rennes

Cardinalités d'une association

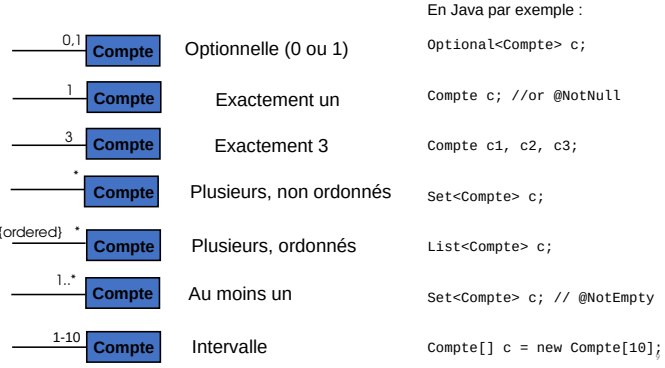
- Précise combien d'objets peuvent être liés à un seul objet source
- Cardinalité minimale et cardinalité maximale ($C_{min}..C_{max}$)
- Doivent être des constantes



« Tout client a toujours 0 ou plusieurs comptes »
« Tout compte a toujours 1 et 1 seul titulaire »



Université de Rennes Cardinalité d'une association

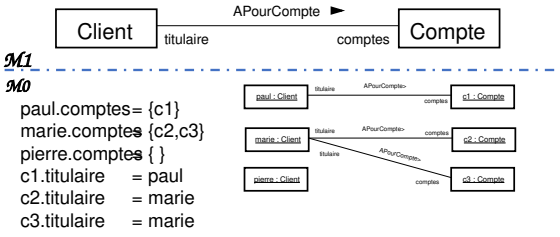


En Java par exemple :

```
Optional<Compte> c;
Compte c; //or @NotNull
Compte c1, c2, c3;
Set<Compte> c;
List<Compte> c;
Set<Compte> c; // @NotEmpty
Compte[] c = new Compte[10];
```


Université de Rennes

Utiliser les rôles pour «naviguer»



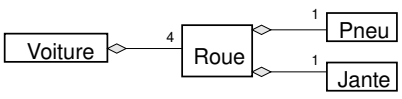
paul.comptes= {c1}
marie.comptes= {c2,c3}
pierre.comptes= { }
c1.titulaire = paul
c2.titulaire = marie
c3.titulaire = marie

Nommer en priorité les rôles

Université de Rennes

Composition

Notion intuitive de "composants" et de "composites"



composition =
cas particulier d'association
+ contraintes décrivant la notion de "composant"...

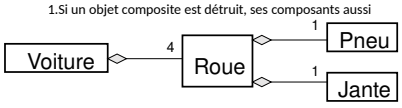
Université de Rennes

Composition

Contraintes liées à la composition :

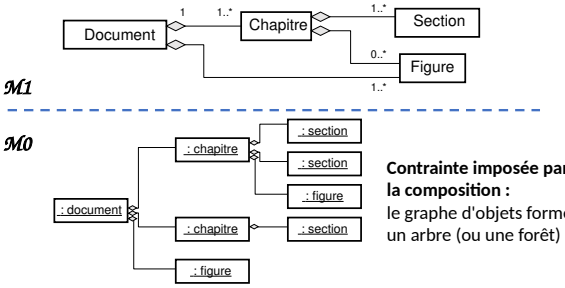
1. Un objet *composant* ne peut être que dans 1 seul objet *composite*
2. Un objet *composant* n'existe pas sans son objet *composite*

1.Si un objet composite est détruit, ses composants aussi



Dépend de la situation modélisée !
(Ex: vente de voitures vs. casse)

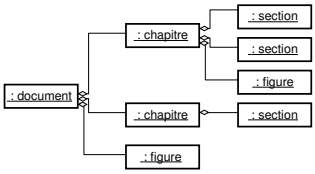
Composition



Contrainte imposée par la composition :
le graphe d'objets forme un arbre (ou une forêt)

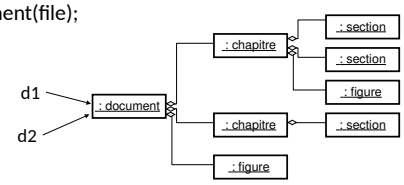
Copie d'objets composites

- Document d1 = new Document(file);



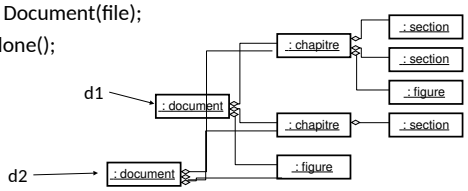
Affectation

- Document d1 = new Document(file);
- Document d2 = d1;



Copie simple

- Document d1 = new Document(file);
- Document d2 = d1.clone();



slido

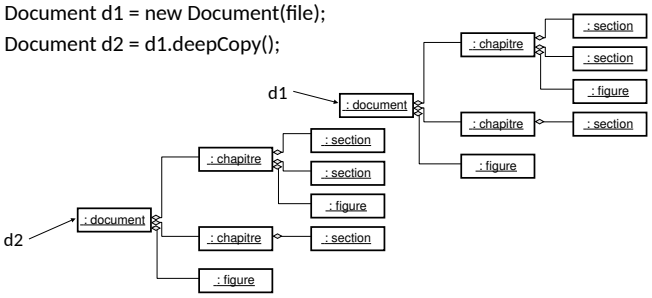
Please download and install the Slido app on all computers you use

Quel est le problème de ce clone() vis-à-vis de la composition?

ⓘ Start presenting to display the poll results on this slide.

Copie profonde

- Document d1 = new Document(file);
- Document d2 = d1.deepCopy();



Copie d'objets complexes / Identité vs Egalité

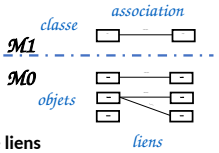
1. Affectation
   - d1 = d2;
2. Copie simple
   - d1 = d2.clone();
3. Copie profonde
   - d1 = d2.deepClone();

1. Test d'identité
   - d1 == d2
2. Test d'égalité
   - d1.equals(d2);
3. Test d'égalité profonde
   - d1.deepEquals(d2);

Support plus ou moins direct en fonction du langage
- Java : seulement 1 et ~2
- Python : 1, 2 (copy) et 3 (deepcopy)

Association vs. Liens

- Un **lien** lie deux **objets**
- Une **association** lie deux **classes**
- Un **lien** est une instance d'**association**
- Une **association** décrit un ensemble de **liens**



- Des **liens** peuvent être ajoutés ou détruits pendant l 'exécution, (ce n 'est pas le cas des associations)

6 – Classification

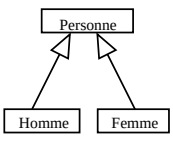
Héritage entre classes

Les objets ne sont pas définis isolément

- 2 catégories
  - Liens entre objets
    - Marie possède les comptes en banque n°12345 et n° 67890
    - Cette voiture est composée du châssis n° XX564 et des 4 roues AvG, AvD, ArG, ArD
    - => Associations entre classes
      - Une Personne possède un ou plusieurs Comptes en banque
      - Une Voiture est composée de 1 Châssis et de 4 Roues
  - Classification
    - Médor est un Chien, il est donc aussi un Mammifère
      - Médor appartient à l'ensemble des Chiens, qui est contenu dans l'ensemble des Mammifères
    - => Héritage entre classes
      - Un Chien est une sorte de Mammifère, qui est une sorte d'Animal

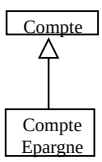
Héritage et Polymorphisme : Généralisation / Spécialisation

Une classe peut être la généralisation d'une ou plusieurs autres classes.
Ces classes sont alors des spécialisations de cette classe.



"Super classe"
Cas général

"Sous classes"
Cas spécifique



On dit que CompteEpargne **hérite** *de Compte*

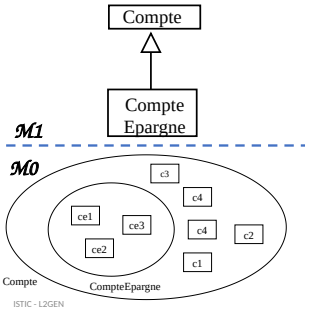
Deux points de vue complémentaires

- Mécanisme de classification : sous-typage
  - relation X est-une-sorte-de Y (est substituable à)
  - organisation des systèmes complexes (cf. Linnaeus)
  - réutilisation d'interface
- Mécanisme d'extension de module
  - ajout de fonctionnalités dans la sous-classe
  - "customisation" et combinaison de composants logiciels
  - réutilisation de code

Héritage : Vision ensembliste

Tout objet instance d'une sous-classe appartient également à l'extension de la superclasse

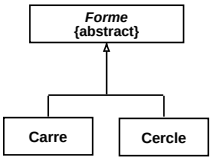
Liskov's Substitution Principle (LSP)
Partout où un Compte est attendu, on peut toujours utiliser un CompteEpargne à la place



Héritage dans les langages à objets

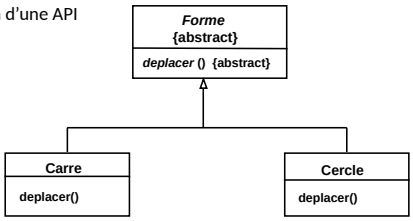
- Java / Dart
```
class CompteEpargne extends Compte {
 void ajouterIntérêts() {…}
}
```
- Python
```
class CompteEpargne (Compte):
 def ajouterIntérêts(self):
```
- C++/ C#
```
class CompteEpargne : Compte {
 void ajouterIntérêts() {…}
}
```
- Swift
```
class CompteEpargne : Compte {
    func ajouterIntérêts() {…}
}
```
- Kotlin
```
class CompteEpargne : Compte(){
    fun ajouterIntérêts() {..}
}
```


- Capturent des comportements communs
- Ne peuvent donc pas être instanciées
- Servent à structurer un système
- Peuvent avoir des opérations dont l'implantation est absente
  - *pure virtual* en C++
  - *abstract* en Java, C#, Kotlin etc.
- Classes sans instances immédiates



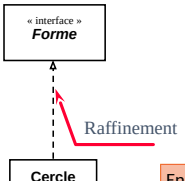
Une instance de «**Forme**» est obligatoirement une instance de la classe **Carre** ou de la classe **Cercle**

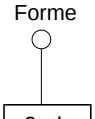
- Opération sans corps d'une classe abstraite
  - Définition d'une API



Interfaces et « lollipop »

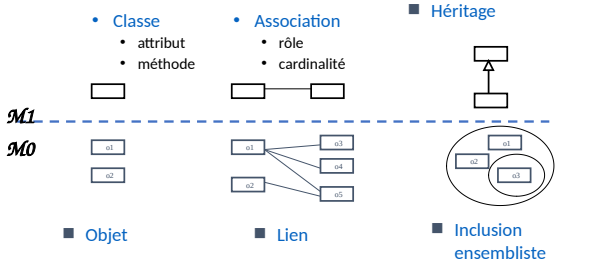
- Interface = classe « totalement » abstraite
  - Dont toutes les méthodes sont abstraites





En Java :
```
class Cercle implements Forme {
    …
}
```


Synthèse des concepts de base



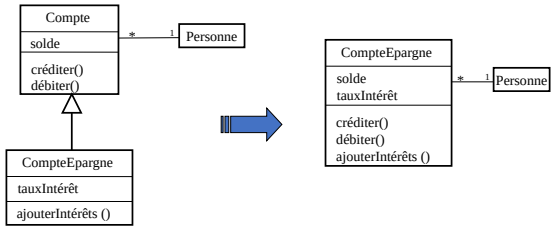
Deux points de vue complémentaires

- Mécanisme de classification : sous-typage
  - relation X est-une-sorte-de Y (est substituable à)
  - organisation des systèmes complexes (cf. Linnaeus)
  - réutilisation d'interface

- Mécanisme d'extension de module
  - ajout de fonctionnalités dans la sous-classe
  - "customisation" et combinaison de composants logiciels
  - réutilisation de code

Relation d'héritage

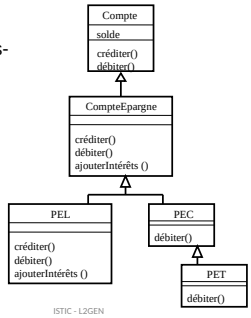
Les sous-classes « *héritent* » des propriétés des super-classes (attributs, méthodes, associations, etc.)



Relation d'héritage et redéfinitions

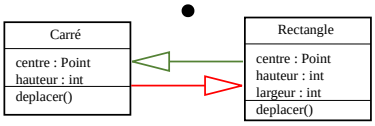
Une opération peut être "redéfinie" dans les sous-classes

Permet d'associer des méthodes spécifiques à chaque pour réaliser une même opération



Carré vs. Rectangle

?



slido

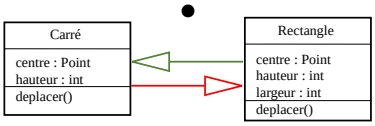
Please download and install the Slido app on all computers you use

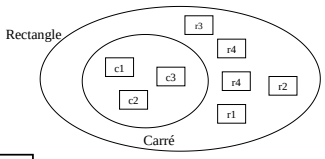
Rectangle hérite t'il de Carré ou bien est-ce l'inverse?

ⓘ Start presenting to display the poll results on this slide.

Carré vs. Rectangle

?

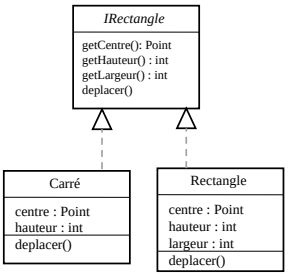




Parfois besoin de dissocier sous-typage et héritage d'implantation!

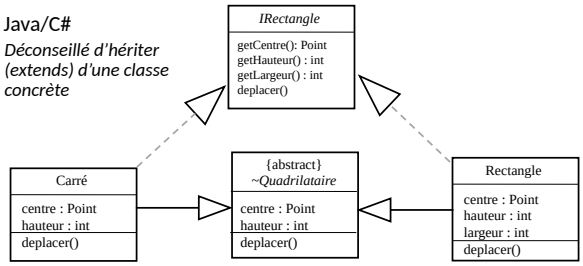
Distinguer Sous-Type de Sous-Classe

- En Java/C#



Distinguer Sous-Type de Sous-Classe

- En Java/C#
  - *Déconseillé d'hériter (extends) d'une classe concrète*



7 – Polymorphisme

Et liaison dynamique

Polymorphisme et liaison dynamique

- Polymorphisme : possibilité de changer de forme

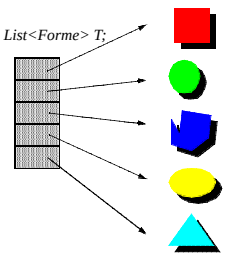
  Forme f; c = new Cercle(); r = new Rectangle();
  f = c;
  f = r;
- Liaison dynamique : l'effet de l'appel d'une opération d'un objet dépend de sa forme effective à l'exécution
  - f.imprimer(); -- différent selon que f est Cercle ou Rectangle
- Espace de nommage réduit et uniforme
  - Mise en facteur des parties communes
  - Possibilité de "conteneurs" hétérogènes

Polymorphisme : exemple

- T contient des Formes
  - en fait des instances de sous-types de Forme
- Programmes de type :

```
T[0] = new Carré();
T[1] = new Cercle();
...
for (Forme f : T)  f.imprimer();
```

- Si ajouts ultérieurs:
  - e.g. triangle
- Pas de modifications globales
  - case-less programming



Parcours d'arbre orienté objet

- Un Groupe contient
  - Des cercles, des rectangles…
  - …et d'autres groupes
- Ce qui forme une structure d'arbre

```
void deplacer(double dx, double
dy){
  for (Forme forme : formes) {
     forme.deplacer(dx, dy);
  }
}
```

```
void deplacer(double dx, double dy){
  formes.forEach(f-
>f.deplacer(dx,dy));
}
```

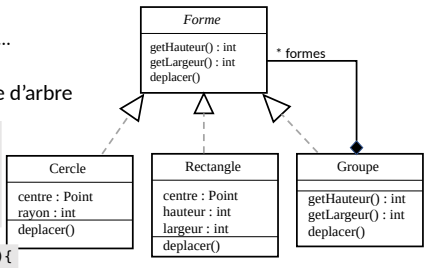
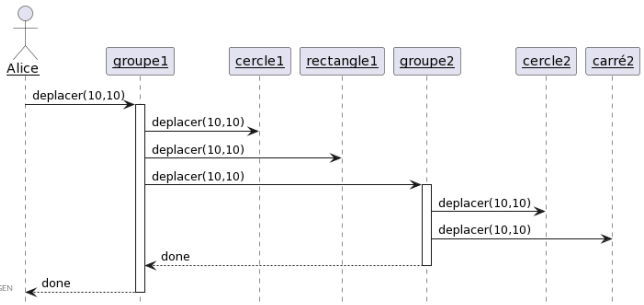
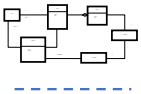


Illustration de la dynamique avec un diagramme de séquence



Synthèse diagrammes de classes et d'objets

- Un diagramme de classes
  - Défini l'ensemble de tous les états possibles
  - les contraintes doivent toujours être vérifiées



- Un diagramme d'objets
  - décrit un état possible à un instant t, un cas particulier
  - doit être conforme au modèle de classes

- Les diagrammes d'objets peuvent être utilisés pour
  - expliquer un diagramme de classe (donner un exemple)
  - valider un diagramme de classe (le "tester")