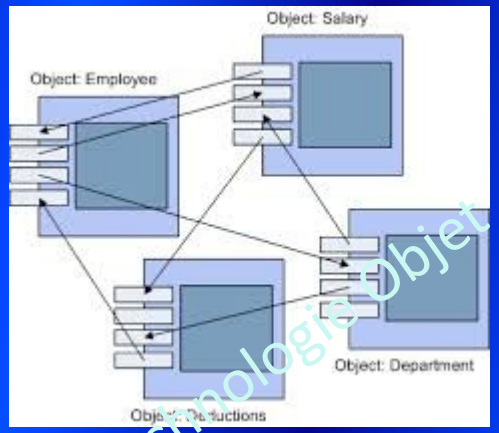
Chapitre 1

Rappel : Eléments paradigmatiques de l'orienté objet

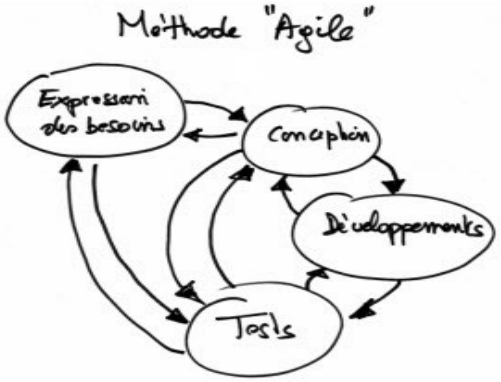


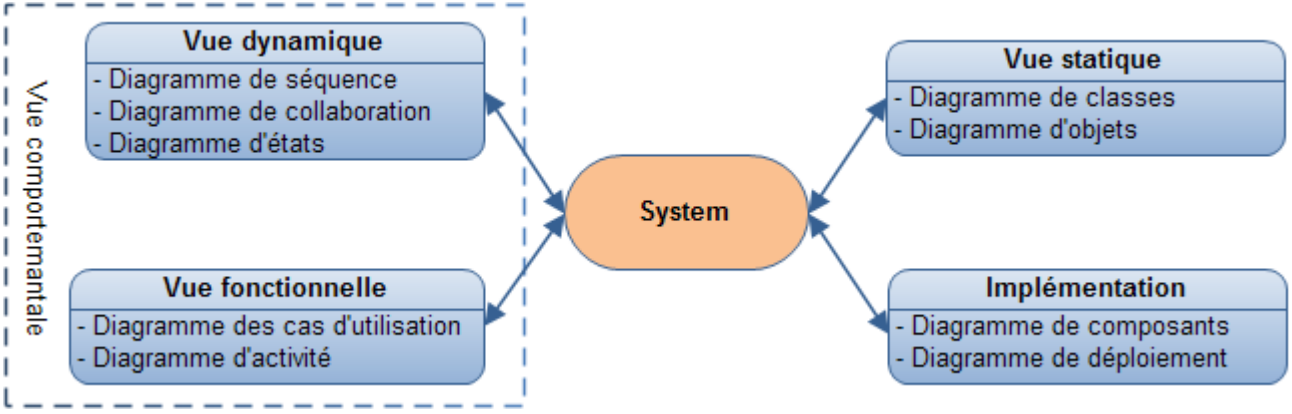
Module : Technologie objet
Master 1 IRC

Mr A. Dekhinet
Université de BATNA
Département d'Informatique

Approche Orientée Objet

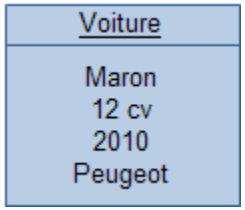
- ✓ Approche permettant de couvrir les trois phase de la conduite d'un projet Informatique
  - Analyse
  - Conception
  - Implémentation
- ✓ Le monde (Système) réel est perçu comme étant un ensemble d'objet concret ou abstrait
- ✓ Système (Selon l'approche orienté objet) : Ensemble d'objets interagissant et communicant entre eux pour réaliser les fonctions du Système.
- ✓ Emergence de méthodes : Booch, Fusion, HOOD, OMT, OOA, …
- ✓ Principales méthodes :
  - **OMT** (Object Modelling Technique), par James Rumbaugh, centre de R&D de General Electric
  - **OOD** (Object Oriented Design), par Grady Booch
  - **OOSE** (Object Oriented Software Engineering), par Ivar Jacobson, centre de développement d'Ericsson
- ✓ **OMG** : Unification et normalisation des approches orientés objets
  - ↳ UML (Unified Modeling Language) : Fusion des trois méthodes OMT, Booch et OOSE
- ✓ **UML** : Notation (Ensemble de diagrammes) pour modéliser la structure, le comportement et l'implémentation d'un système.



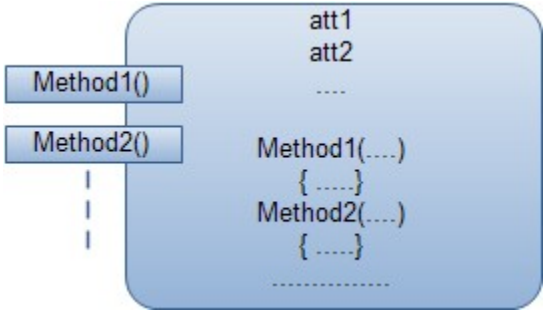


Classe et Objets

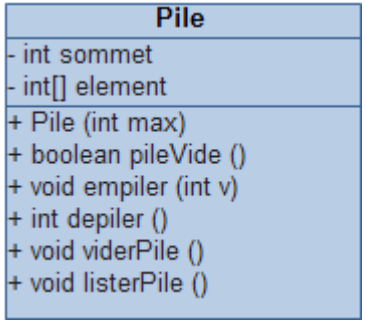
- ✓ Principaux éléments paradigmatiques : Objet, Classe, Encapsulation, Héritage, Polymorphisme, Surcharge, Généricité, …
- ✓ **Objet** (**Identité + Etat + Comportement)** : Entité atomique caractérisée par :
  - une identité
  - un état : Valeurs instantanées des propriétés (attributs)
  - un comportement : Ensembles d'opérations (méthodes) interface



- ✓ **Classe** : Un modèle (Moule) représentant une famille d'objets ayant :
  - La même sémantique
  - Le même comportement
  - Les mêmes attributs
  - Les mêmes relations avec d'autres objets
    - L'objet est une **instance** de la classe
    - La classe constitue un **Type** : Pile p1; p1 = new Pile(10);
    - Réutilisation, Prototypage, …



- ✓ **Encapsulation** (information hiding, abstraction, intégrité) : Dissimuler les détails internes de l'objet, en fournissant aux autre objets une interface d'utilisation. Les attributs ne sont manipulable qu' à travers l'interface.
- ✓ **Visibilté** des attributs et des méthodes :
  - - attribut ou méthode privée (private) : visible a la classe
  - # attribut ou méthode protégée (protected) : visible aux classes dérivées
  - + attribut ou méthode (public) : Interface

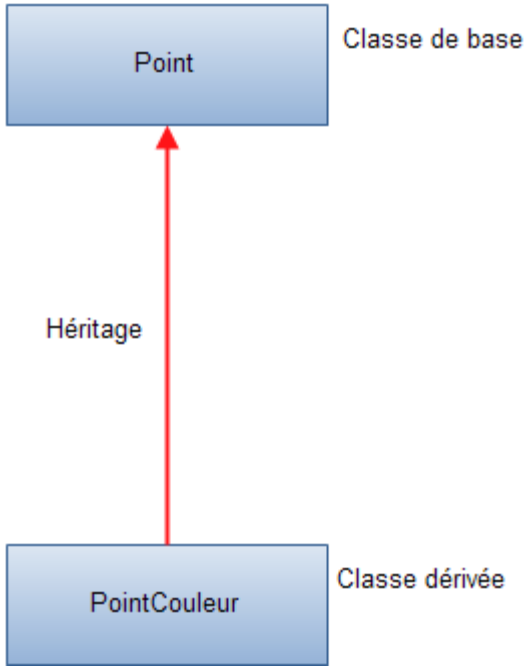


Héritage

- ✓ Relation entre classes : Héritage, Composition, Agrégation, Association
- ✓ **Héritage** : Evite la duplication et favorise la réutilisation
  - Mécanisme de transmission des propriétés (attributs ou méthodes) d'une classe de base à une autre derivée.
  - Deux types d'héritage : simple et multiple .
  - C++, Eiffel et Python supporte l'héritage multiple. Java, ADA et Objective-C ne le propose pas directement.
  - L'héritage multiple en C++ : **class** Cfille : **public** CMere1, **public** CMere2, …
  - Java propose le concept d'interface, notion proche de l'héritage multiple :
    class MyClass **implements** interface1, interface2, …
  - Spécialisation/Généralisation (Raffinement/Factorisation) : Sous classe/Super classe .

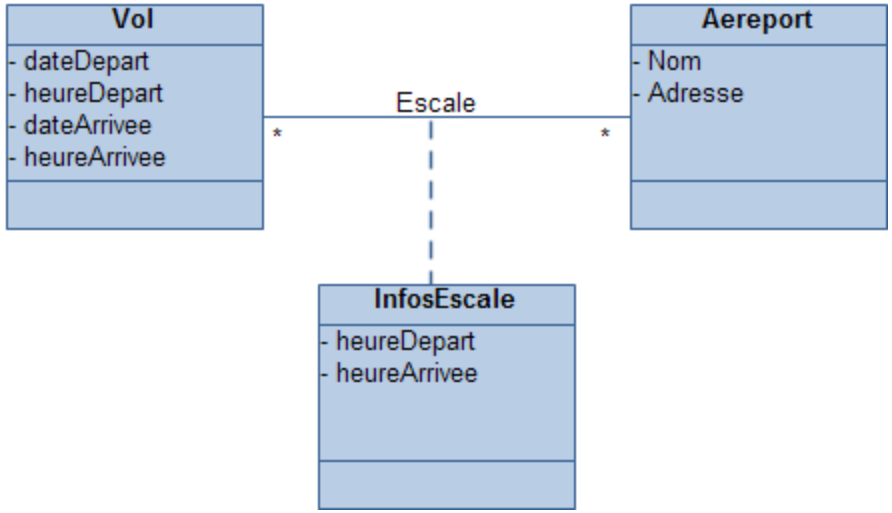
```
public class Point {
        private int x ;
        private int y ;
        public Point(int x, int y) {
                this.x = x ;
                this.y = y ;
        }
        public void deplacer(int dx, int dy) {
                x += dx ;
                y += dy ;
        }
        public void afficher() {
                System.out.println("Point ("+x+", "+y+")") ;
        }
}
```

```
public class PointCoul extends Point {
        private int couleur ;
        public void colorer(int couleur) {
                this.couleur = couleur ;
        }
}
```



Association et Agrégation

- ✓ Principe de susbtitution de **Liskov**
  - Permet de déterminer si une relation d'héritage est bien employée comme classification.
  - **Principe** : Si la classe **S** est un sous-type de la classe **T**, alors on peut substituer des objets de type **S** à des objets de type **T** sans altérer les propriétés désirables d'un modèle.
- ✓ **Association**
  - Capture la relation ou le lien sémantique entre classes . Equivalente à l'association dans le modèle E-A .
  - Classe d'association : Association promue en classe, elle peut être porteuse de nouveaux attributs.
- ✓ **Agrégation :** Exprime un lien de composition entre classes (IsPartOf, ComposeDe, ElementDe, …) : Relation de composé et composants
- ✓ **Composition** : Agrégation forte , la suppression du composé implique la suppression des composants et vice-versa.



Polymorphisme

- ✓ **Polymorphisme :** Mécanisme permettant à un élément de prendre plusieurs formes., Il complète l'héritage.
- ✓ Types de polymorphisme :
  - Surcharge : Overloading, Ad hoc
  - Redéfinition : Overriding, polymorphisme d'héritage
  - Généricité : Polymorphisme paramétrique
- ✓ **Surcharge :** La même méthode peut réaliser des fonctions différentes (différentes implémentations) en fonction du nombre et du type des paramètres passés (paramètres formels).
  - Surcharge des opérateurs (C++) : Redéfinir par surcharge les opérateurs du langage

```
Matrice M1[], M2[], M3[]
M1 = M1 + M2              // l'opérateur + redéfini pour les types matrices
```

  - Surcharge des méthodes : Définir par surcharge plusieurs méthodes de même nom au sein d'une classe, réalisant des opérations différentes

```
class figure {
        perimetre(r) {…};              // Cercle
        perimetre(l,g) {…};            // Rectangle
        perimetre( d1,d2,d3) {…};      // Triangle
}
```

- ✓ **Redéfinition :** La même méthode peut figurer dans la classe de base et la classe dérivée. L'exécution est réalisée en fonction de l'objet associé à l'appel **.**

- ✓ **Généricité**
  - Des types paramètrent d'autres types.
  - Paramètres sur les classes (classe générique) ou sur les méthodes (méthodes génériques)

Généricité : Pile générique

```
Public class Stack<T> {
  private int size;
  private T[] elements;
  private int top;

 public Stack(int size) { this.size = size; elements = (T[])new
Object[size];top = -1; }

 public void push(T value) {
       if(isFull()){
            throw new StackFullException("Cannot push "+value+",
Stack is full");
       }
       elements[++top] = value;
  }

  public T pop() {
       if(isEmpty()){
            throw new StackEmptyException("Stack is empty");
       }
       return elements[top--];
  }

  public boolean isEmpty() { return (top == -1); }

  public boolean isFull() { return (top == size - 1); }
}
```

```
public class StackGeneric {
  public static void main(String[] args) {
       Stack<Integer> MyStack = new Stack<Integer>(10); //
Creation of Generic Stack
       MyStack.push(11);
       MyStack.push(21);
       MyStack.push(31);
       MyStack.push(41);
       MyStack.push(51);

       System.out.print("Popped items: ");
       System.out.print(MyStack.pop()+" ");
       System.out.print(MyStack.pop()+" ");
       System.out.print(MyStack.pop()+" ");
       System.out.print(MyStack.pop()+" ");
       System.out.print(MyStack.pop()+" ");

  }
}
```