

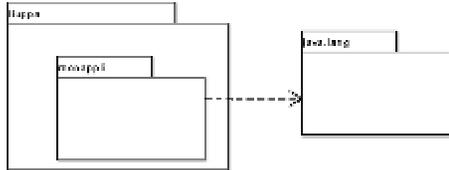
# CARTE DE REFERENCE UML 2.0

Jean-Michel Bruel – 2007

http://www.univ-pau.fr/~bruel

## Diagrammes Statiques

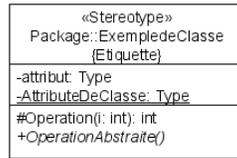
### Packages



- Groupement de classes et leurs dépendances.
- Le nom du Package préfixe le nom de la classe (e.g., MasterTI.Etudiant).

– Existe en Java/C#, namespace en C++.

### Classes



Trois parties :

1. un entête avec le nom, et éventuellement un accesseur, un stéréotype et des étiquettes
2. (optionnel) des attributs, et éventuellement pour chacun un accesseur, un stéréotype, un type, une valeur initiale.
3. (optionnel) des opérations, et éventuellement pour chacun un accesseur, un stéréotype, des paramètres typés, un type de retour

Les membres de classe sont soulignés.  
Les opérations abstraites sont en *italique*.

visibilité nom: type multiplicité = valeurParDéfaut {propriété}

+ patronyme: String [1] = "Bruel" (readOnly)

### Acces

Accesseurs (attributs, opération) :

+	public	tout le monde
#	protected	les sous-classes
-	private	privées
~	package	espace de nommage

### Associations

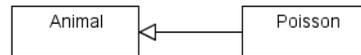


Une association de base est une ligne entre 2 classes. Les éléments suivants sont optionnels :

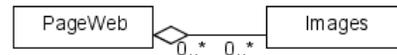
- Un nom suivi (ou précédé) d'un > (<) pour indiquer le sens de lecture
- le rôle joué par la classe dans l'association
- un stéréotype
- des directions pour indiquer la navigabilité de l'association (flèches en bout)
- des cardinalités

1	un et un seul
*	plusieurs (0..*)
0..1	optionnel (0 ou 1)
m..n	de m à n (entiers > 0)

### Généralisation/Spécialisation (Héritage)



### Agrégations



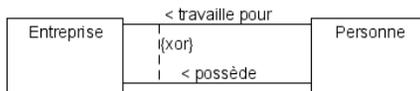
Lien fort entre classes.

### Compositions



La destruction du tout entraîne celle des parties. Cardinalité toujours 1 côté Tout.

### Contraintes



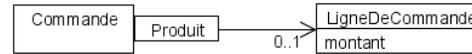
Certaines classiques ({ordered}, {xor}, {subset}).

### Contraintes OCL

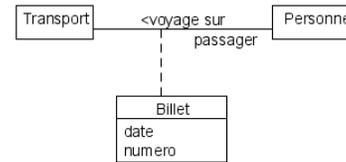


context Account inv: owner.age > 18

### Associations qualifiées



### Classes d'association



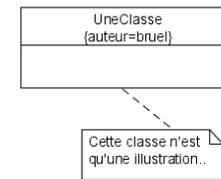
### Classes paramétrées

Liées au C++

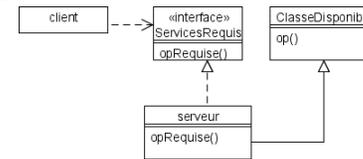


### Étiquettes et notes

Information supplémentaire

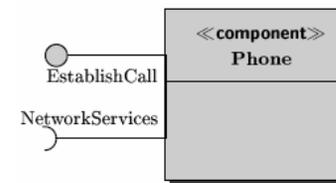


### Interfaces

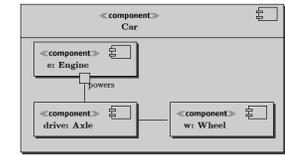


Natif en Java/C#, classe C++ ne contenant que des méthodes virtuelles. Utilisation du stéréotype «interface» La partie attributs est toujours vide.

Nouvelle notion d'interfaces requises.

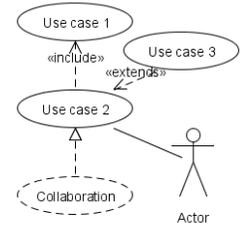


### Diagrammes de composants



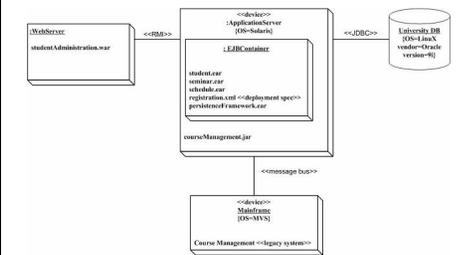
Nouveau dessin (logo ou «component»).

### Diagrammes de cas d'utilisation



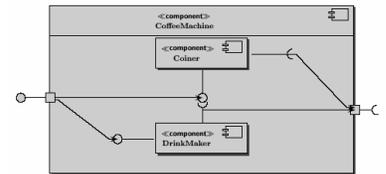
Cas = ensemble de scénarios reliés par un but commun. Acteurs (rôle d'utilisateur du système), limites du systèmes (rectangle), fonctionnalités principales et liens entre elles (extension, inclusion)

### Diagrammes de déploiement



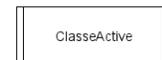
### Diagrammes de composites

Ports, connecteurs, connecteurs de délégation



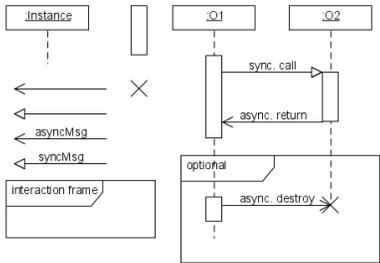
### Classes actives

Anciennement en cadre gras.

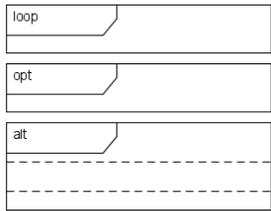


## Diagrammes dynamiques

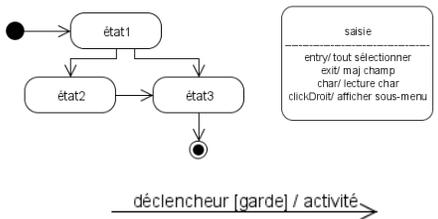
### Diagrammes de séquences



Objets (boîtes en haut), lignes verticales (existence des objets – « lignes de vie »), lignes rectangulaires (activité de l'objet), flèches (envois de messages). L'objet émetteur doit avoir une association avec l'objet receveur. Création d'objet (apparition de l'objet à la fin du message), destruction (une croix X terminant une ligne de vie).



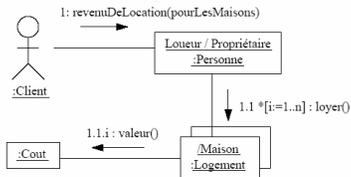
### Diagrammes d'états



Etats, transitions entre états, activités internes  
Eventuellement superEtats

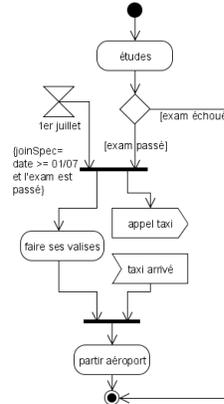
### Diagrammes de communication

Anciennement diagramme de collaboration.

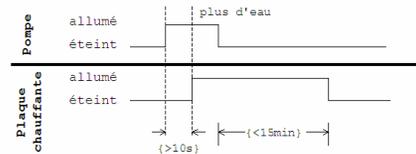


## Diagrammes d'activité

Processus métiers, workflows, enchaînements d'activités.

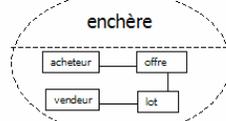


### Diagrammes de Timing



### Collaborations

Pas un diagramme officiel UML.



participant / rôle : classe  
utilisé e.g., pour les patterns

### Principaux stéréotypes standards

<<abstract>>	classe	pas d'instance
<<artifact>>	composants	instance
<<call>>	dépendance	entre opérations
<<component>>	composants	création instance
<<extend>>	cas d'utilisation	extension

<<frozen>>	relation	immuable
<<implement>>	composants	spécification
<<include>>	cas d'utilisation	réutilisation
<<realize>>	interface	realisation
<<utility>>	classe	sans instance
<<use>>	dépendance	utilisation

## Codage et conception

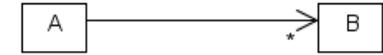
### Classe et associations

<pre> &lt;&lt;abstract&gt; Personne - nom: String - ageMajorite: int = 18 # dateNaissance: Date + setNom(s: String): + getNom(): String;         </pre>	<pre> public abstract class Personne {     private String nom;     private static int         ageMajorite = 18;     protected Date         dateNaissance;     public void         setNom(String s) {         //...     }     public String         getNom() { //...     } }         </pre>
<pre> class Etudiant         </pre>	<pre> public class Etudiant extends Personne { //... }         </pre>
<pre> &lt;&lt;interface&gt; IEtatCivil + setNom(s: String): + getNom(): String;         </pre>	<pre> public abstract class Personne implements I Scolarite, I EtatCivil { //... }         </pre>
<pre> package Universite; import Scolarite; //...         </pre>	<pre> public class Personne {     private Personne         conjoint;     private Personne         amis[];     //... }         </pre>



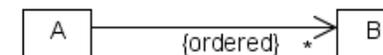
```

public class A
{ private B b; //...
}
        
```



```

public class A
{ private B b[]; //...
}
        
```



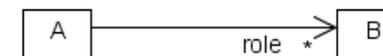
```

public class A
{ private List b = new
    ArrayList(); //...
}
        
```



```

public class A
{ private Hashtable b
    = new Hashtable(); //...
}
        
```



```

public class A
{ private B role[]; //...
}
        
```

### Design Patterns

exemple du pattern Etat.

