Architecture Logicielle

Quelques rappels sur la POO et UML

Florent Nicart

Université de Rouen

2016–2017

Contenu

Ce chapitre a pour but de faire un bref rappel sur

- Les concepts de la programmation orientée objet,
- la terminologie employée et
- les notations UML associées.

UML : Unified Modeling Language

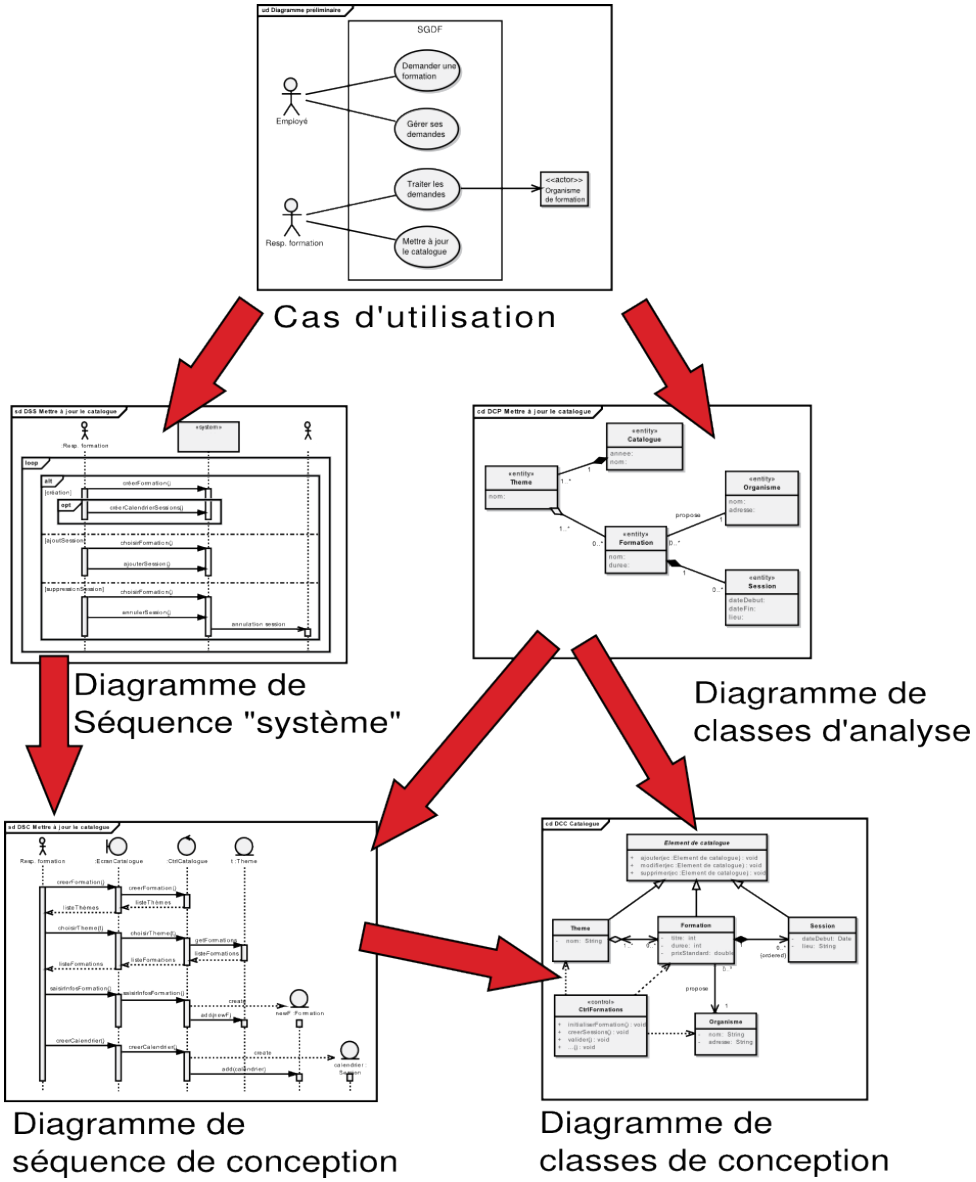
- UML est un langage de description graphique unifié [1].
- Ce n'est pas une méthode d'analyse.
- Spécifications élaborées par l'*Object Management Group*, consortium à but non lucratif.
- UML facilite le processus de conception des logiciels. Il contribue également à leur documentation et à faciliter les échanges entre développeurs.

1. Convergence de OMT(Rumbaugh), OOD(Booch) et OOSE(Jacobson).

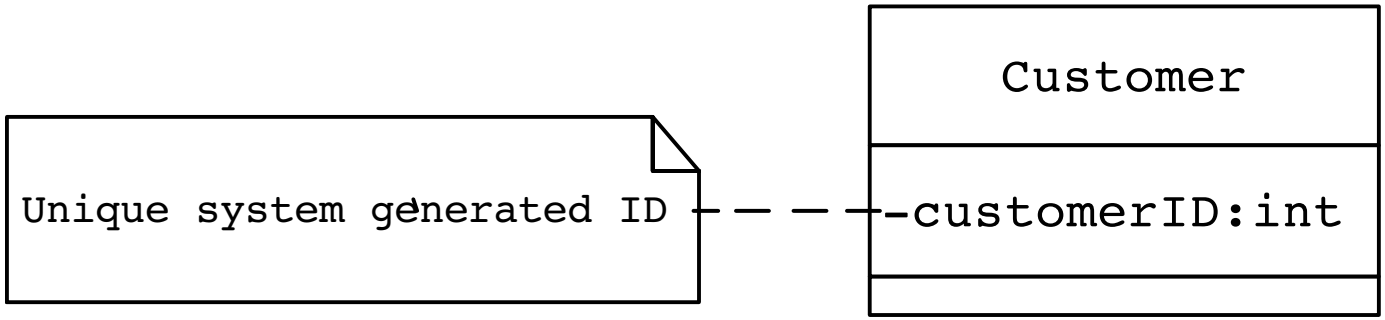
UML : Unified Modeling Language

- Diagrammes de structures :
  - **diagrammes de classes**,
  - **diagrammes d'objets**,
  - diagrammes de composants,
  - diagrammes de déploiement.
- Diagrammes de comportement :
  - diagrammes de cas d'utilisation,
  - **diagrammes de séquences**
  - diagrammes d'activité, de collaboration, etc. ...
- Diagrammes de gestion de modèles :
  - diagrammes de paquetage,
  - diagrammes de sous-systèmes,
  - diagrammes de modèles.

Démarche de modélisation



Annotations



- Comme tout langage, UML autorise les commentaires (annotations).
- Une note est représentée par une boite avec le coin supérieur droit plié reliée à l'élément annoté par une ligne en pointillés.
- Commun à tous les types de diagrammes.

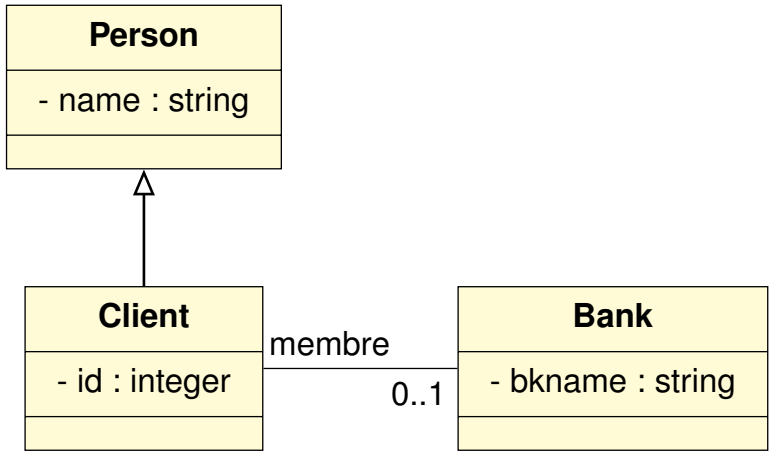

Un commentaire doit faire apparaître une information importante qui ne peut être déduite du schéma lui-même.

Diagrammes de classes et d'objets

Nous allons introduire dès maintenant deux des formalismes graphiques d'UML :

- les diagrammes de classes et d'objets,
- le rôle de chacun, utilisation liées,
- présentation sommaire (du diagramme de classes),
- les notations détaillées seront introduites au fur et à mesure pour illustrer les rappels sur les concepts de POO.

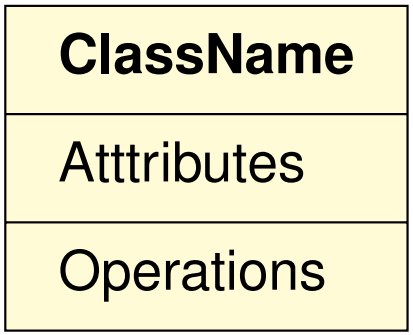
Diagrammes de classes



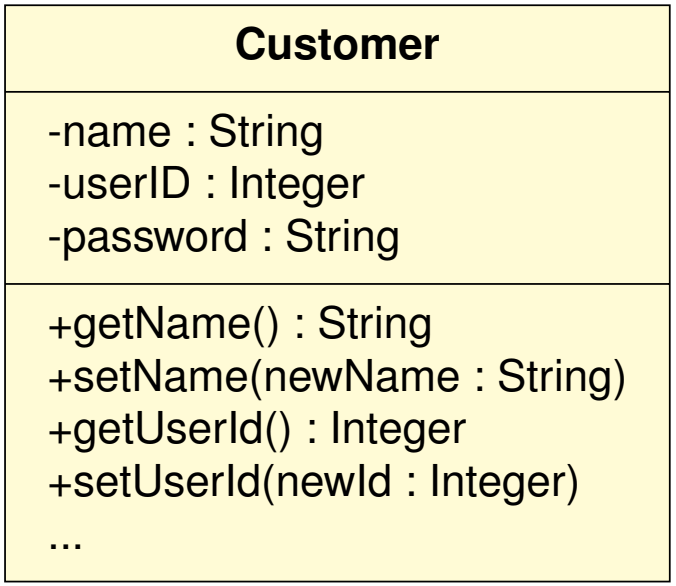
- Un diagramme de classes représente la structure statique d'un logiciel,
- il exprime l'ensemble des configurations possibles des instances des entités impliquées.
- Entités : classes, classes abstraites, interfaces.
- Relations : association, composition, agrégation, implémentation, héritage.

Représentation d'une classe

Syntaxe :



Exemple :



Une classe est représentée par une boite à trois compartiments contenant :

1. le nom de la classe,
2. ses *attributs* (ou *variables d'instance*),
3. ses *méthodes* (ou *opérations*).

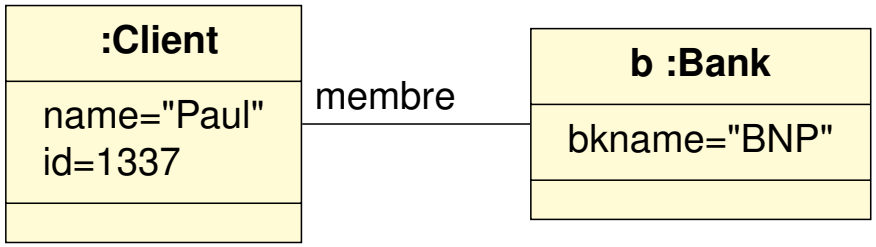
Visibilité des membres

- La visibilité des membres peut être altérée par les modificateurs *public, protégé, privé*.

Symbol	Scope
+	**Public**
#	**Protected**
-	**Private**

Specifier	Classes in the Same Package	Classes in Other Packages	Subclasses in the same Package	Subclasses in Other Packages
Public	**Access**	**Access**	**Access**	**Access**
Protected	**Access**	**No Access**	**Access**	**Access**
Friendly / Package (*No Specifier*)	**Access**	**No Access**	**Access**	**No Access**
Private	**No Access**	**No Access**	**No Access**	**No Access**

Diagrammes d'objets

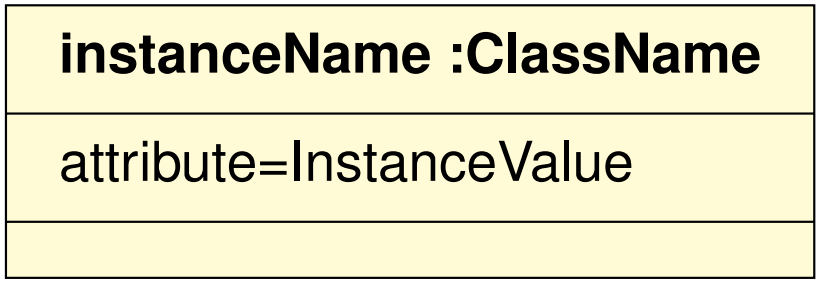


Example of a bank-client association.

- Un diagramme d'objets (ou d'instances) montre une configuration particulière d'objets (d'instances) et leur liens.
- Correspond à une instance d'un diagramme de classes :
- Entités : objets (instances de classes).
- Relations : liens (correspondant à des associations).
- Il sert à illustrer d'un cas particulier et doit porter un nom explicite.
- Il ne montre que le niveau de détails nécessaires à la compréhension du scénario.

Représentation d'un objet

Syntaxe :



Notation similaire à la classe :

- le nom est un couple : nom de l'instance (facultatif), nom de la classe(peut être omis),
- les *attributs* sont donnés lorsqu'ils sont valués,
- dernier compartiment vide.

Diagramme d'objets

Instance de diagramme de classes

Un diagramme d'objets permet de décrire un exemple possible dans un ensemble dénoté par un diagramme de classe. Exemples :

Diagramme de classes :

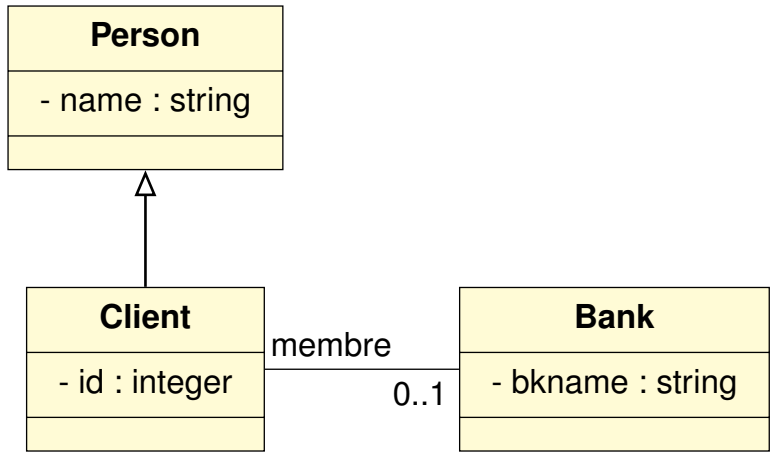
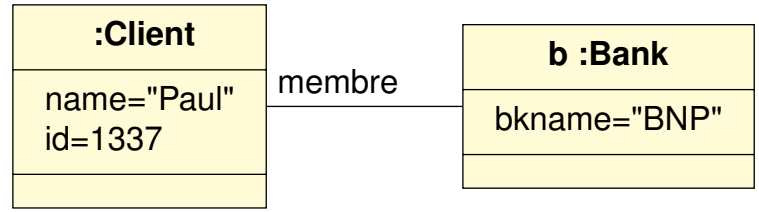


Diagramme de d'objets :



Example of a bank-client association.

- Code :

```
1  // Example of a bank-client association :
2  Bank b = new Bank("BNP");
3  return new Client("Paul", 1337, b);
```

Rappels de quelques concepts fondamentaux de la POO

Classes comme espaces de noms

Exemple en C++ :

Définition :

```
1  namespace lib {
2      namespace tools {
3          public class Utils {
4              ...
5          }
6      }
7  }
```

Utilisation :

```
1  lib.tools.Utils u=new lib.tools.Utils()
       ;
2  // Ou :
3  using namespace lib.tools;
4  Utils u=new Utils();
```

En Java, ce rôle est joué par les paquetages et les classes elles-mêmes. Les membres de classes peuvent être :

- des variables, des constantes, des énumérations
- d'instances ou de classe (membres statiques),
- des méthodes, des méthodes statiques

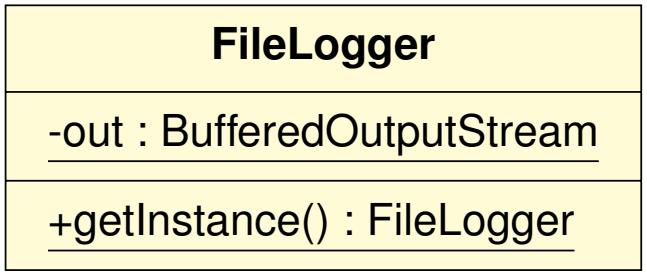
mais aussi des définitions imbriquées

- de classes et
- d'interfaces

Membres de classe (statiques)

En UML, les membres statiques sont soulignés.
En java :

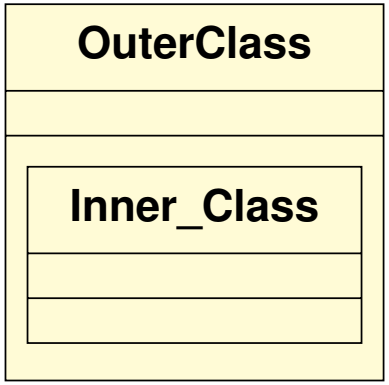
1. Les variables de classe (*attributs statiques*) peuvent être initialisés au chargement du paquetage.
2. Une méthode statique est appelée sur la classe et non sur une instance. Elle ne peut accéder qu'aux variables de classe.

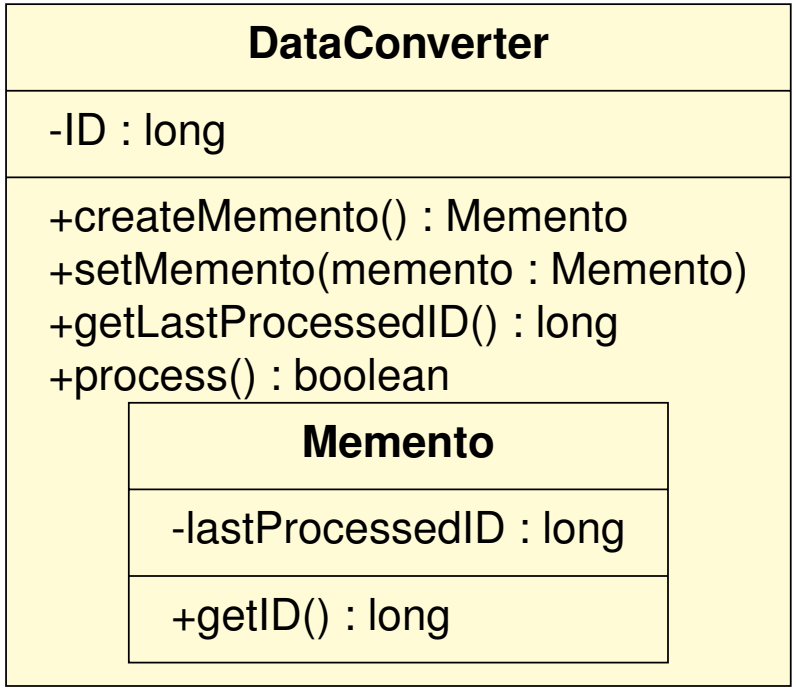



```
1  public class Test {
2      private static Date date = new
           Date();
3      ...
4
5      public Test() { ... }
6
7      public static void getTestDate()
           {
8          return date;
9      }
10 }
```

```
1  Date dt=Test.getTestDate();
```

Définitions imbriquées





UML permet la définition d'une classe à l'intérieur d'une autre mais ce concept n'est pas généralisé :

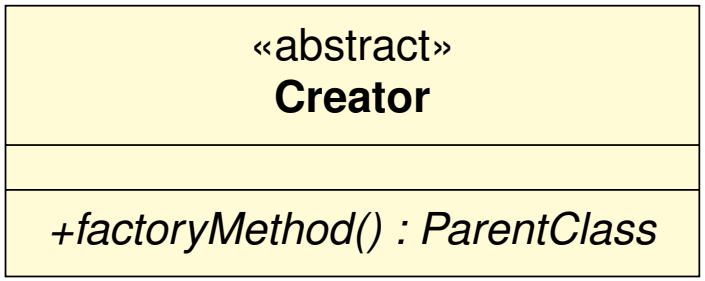
- Naturel en Java et C#
- seulement struct et enum en C++.

```
1  x=new DataConverter.Memento();
```


Imbrication uniquement de l'espace de nom !

Classes et méthodes abstraites

- Une méthode sans définition (sans corps) est dite abstraite (mot clé *abstract* en Java). Sa définition sera fournie dans une classe dérivée.
- Une classe qui possède au moins une méthode abstraite est dite abstraite. (Peut être forcé avec le mot clé *abstract*). Un telle classe ne peut être instanciée.



En UML, les classes et méthodes abstraites s'écrivent en italique.

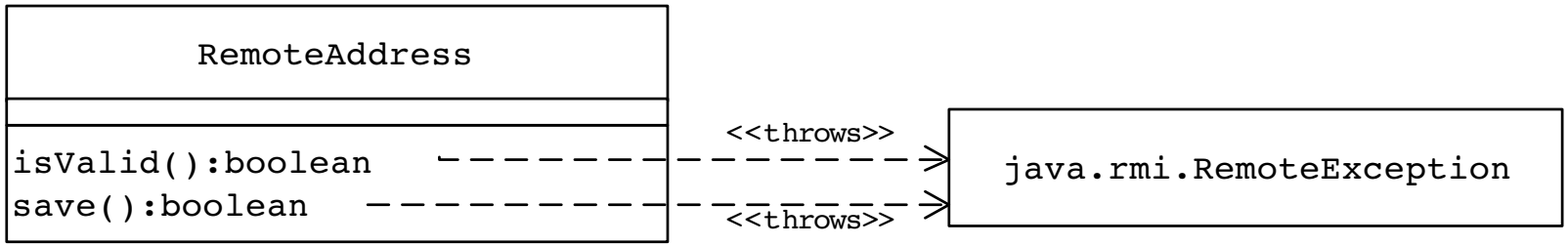
- La notation « *abstract* » existe aussi pour les classes.

Les exceptions

- En Java, **throws** indique que la fonction définie est susceptible de déclencher une exception.

```
1  public class Test {
2      public boolean isValid() throws java.rmi.RemoteException {
3          ...
4      }
5      public void boolean() throws java.rmi.RemoteException {
6          ...
7      }
8  }
```

- L'équivalent en UML est une ligne en pointillés annotée avec *throws* entre la méthode et la boite de l'exception.



La généricité

- Les types génériques permettent d'employer une classe en typant, à l'utilisation, certains de ses types internes.

```
1  public class ArrayList<E> extends AbstractList<E> implements List<E> {
2  ^^Iboolean ^^Iadd(E e) { ... }
3          ...
4      }
5  }
6  // Utilisation :
7  List<String> sl=new ArrayList<String>();
8  sl.add("coucou");
```

- En UML, les paramètres génériques sont indiqués en haut à droite. Ex :

Signatures et surcharge

Définition (Signature de fonction)

La signature d'une fonction est définie par le nom de la fonction, le nombre et le type de ses paramètres.

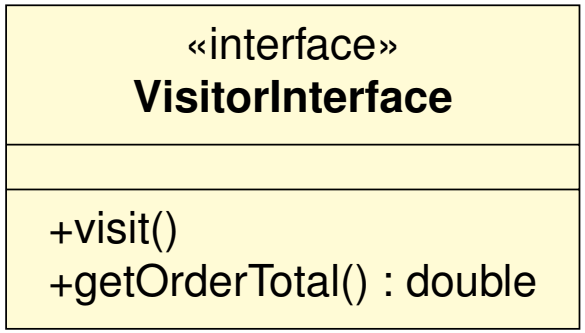
Le type de retour et le nom des paramètres sont exclus :

```
1 // Definissent la meme fonction :
2 List getList(int i);
3 List getList(int rank);
4 Iterator getList(int rank);
```

Surcharge et polymorphisme *ad hoc* :

```
1 // Definissent differente fonctions :
2 List merge(List a, List b);
3 Vector merge(Vector a, Vector b);
```

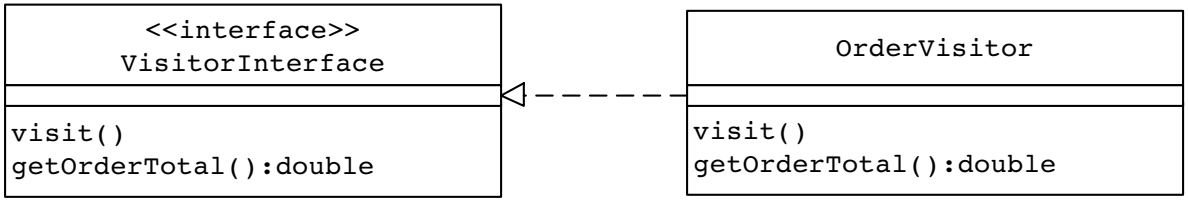
Les interfaces



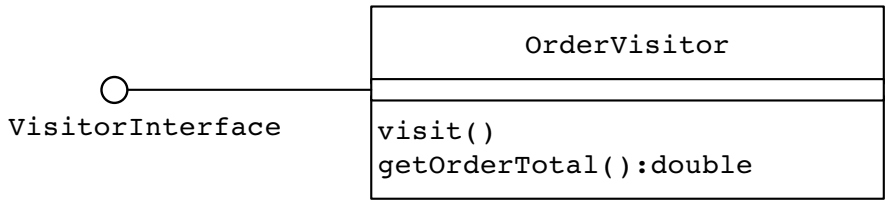
- Une interface permet de spécifier un vocabulaire : un ensemble de méthodes.

- Une interface peut être vue comme une classe abstraite sans attribut (*cf* C++).
- En UML, une interface se représente comme une classe dont le nom est préfixé du mot *interface* entre chevrons.

Réalisation d'interface



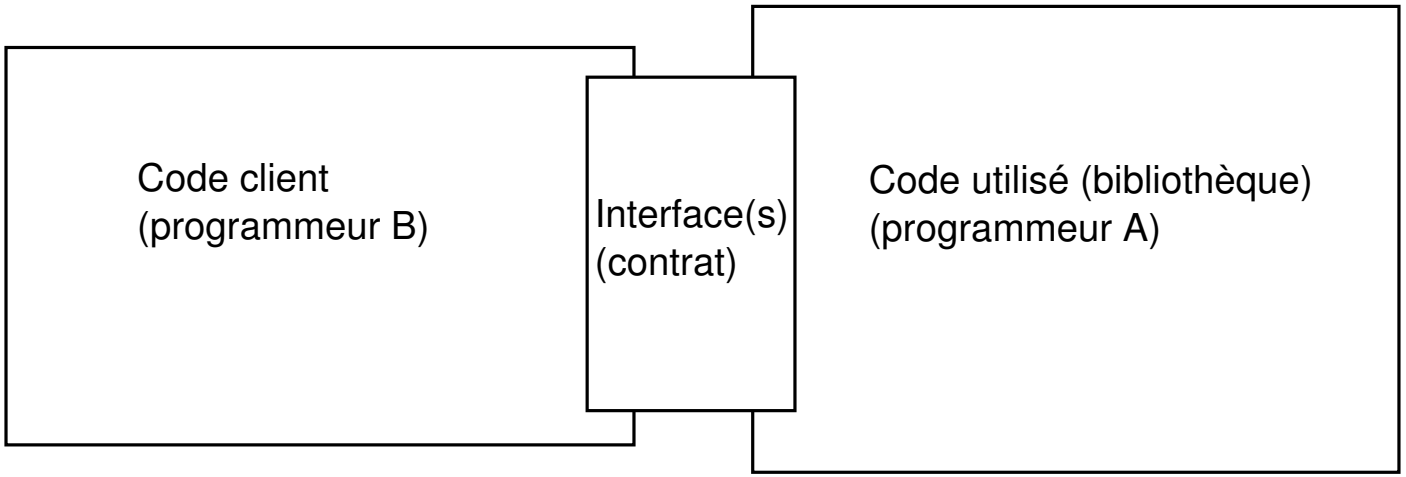
- On dit qu'une classe *implémente* ou *réalise* une interface lorsqu'elle fournit une implémentation pour (toutes) les méthodes de cette dernière.
- Une classe peut implémenter plusieurs interfaces.
- Une notation abrégée permet de ne pas redéfinir les interfaces connues ou déjà définie par ailleurs :



Les interfaces

Programmation par contrats

- Les interfaces sont la base de la programmation par contrat,
- Une interface définit un contrat d'utilisation entre un composant et son client (le code qui l'utilise)
- par extension entre les développeurs !



Les interfaces

Opérations et sémantique

- Un contrat implique une sémantique que l'on ne retrouve pas forcément dans la définition des méthodes.
- À ce niveau, on doit parler d'**opérations** plutôt que de méthodes.

Opération ***Brancher*** :

⇐

brancher

Les interfaces

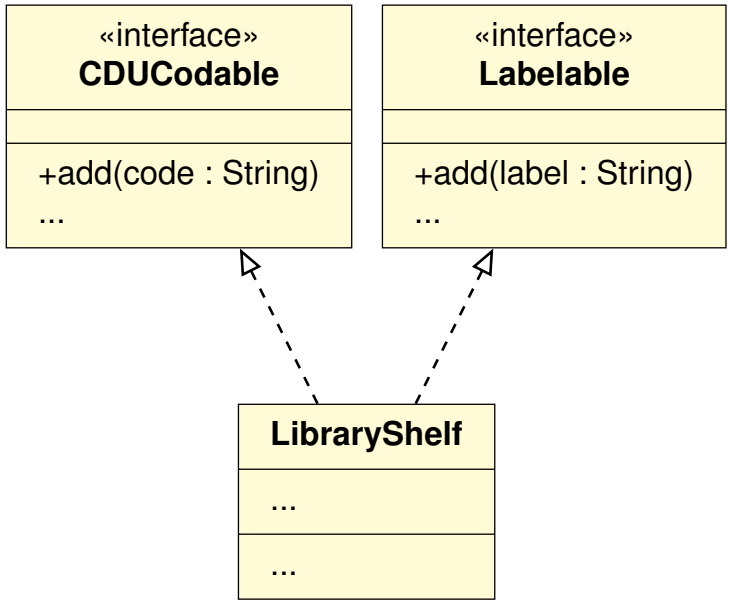
Opérations et sémantique

Mais en y regardant de plus près :

Les interfaces

Opérations et sémantique

Exemple en POO : conflit entre CDU [2] et étiquettes d'étagères.

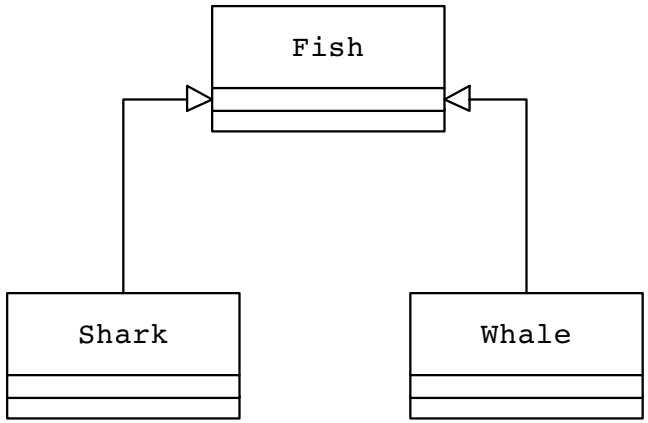


Quel sens donner à `LibraryShelf.add(String)` ?

2. Codification Décimale Universelle.

L'héritage

- L'héritage permet de *spécialiser* le concept représenté par une classe.
- On parle de *super-classe* (ici *Fish*) et de *sous-classe* (ici Shark et Whale).



Ou encore de *classe mère* ou *classe de base* et On encore de *classe fille* ou *classe dérivée*.

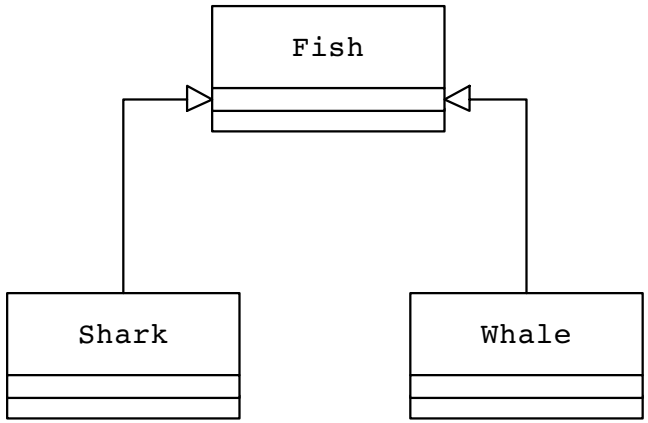
On dit aussi que :

- *Shark* et *Whale* sous-classent *Fish*.
- *Shark* et *Whale* spécialisent *Fish*.
- *Fish* généralise *Shark* et *Whale*.

Ce processus de hiérarchisation s'appelle *classification*.

L'héritage

- L'héritage permet de *spécialiser* le concept représenté par une classe.
- On parle de *super-classe* (ici *Fish*) et de *sous-classe* (ici Shark et Whale).



Ou encore de *classe mère* ou *classe de base* et On encore de *classe fille* ou *classe dérivée*.

On dit aussi que :

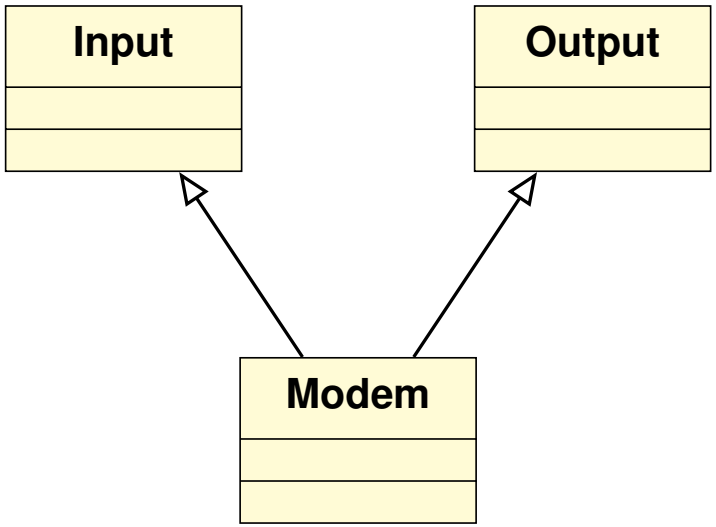
- *Shark* et *Whale* sous-classent *Fish*.
- *Shark* et *Whale* spécialisent *Fish*.
- *Fish* généralise *Shark* et *Whale*.

Ce processus de hiérarchisation s'appelle *classification*.

 Réaction du public attendue !

L'héritage

Simple ou multiple

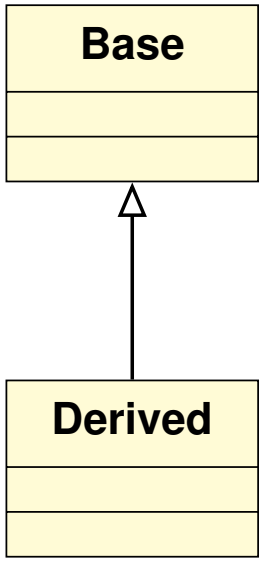


- Certains langages supportent l'héritage (classification) multiple : C++, Eiffel, Python, ...
- Java ne supporte que l'héritage simple :

```
1 class Shark extends Fish {
2 ...
3 }
```

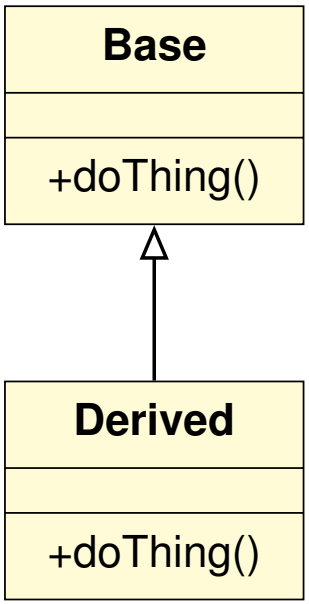
Héritage

- La classe dérivée hérite de l'intégralité du contenu de la classe de base.
- L'interface « publique » de la classe de base est incluse dans celle de la classe dérivée.
- Changement de visibilité uniquement possible : protégé $\rightarrow$ public (par redéfinition)



- Ressemble à un copier coller du code de la classe de base...
-  En Java : assemblage dynamique à l'exécution
- danger si la classe de base change voir ocp plus tard

Protections contre l'héritage



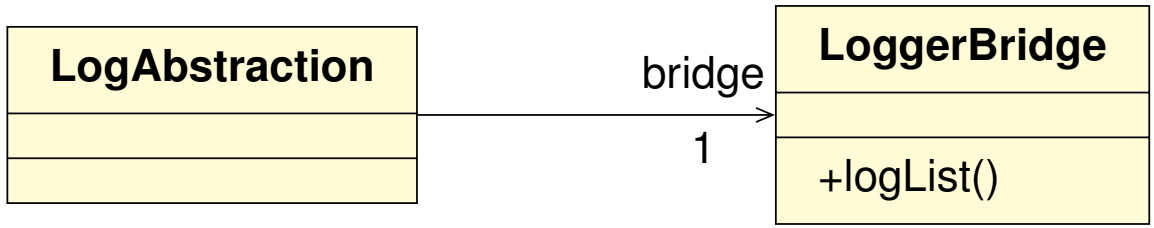
- Surcharger une méthode dans une classe dérivée s'appelle une **redéfinition**.
- En Java, on peut empêcher la redéfinition d'une méthode en la déclarant `final`.
- De la même manière, on peut interdire la dérivation d'une classe entière la déclarant `final`.

Les associations

- Les associations correspondent à la possibilité données aux instances de se faire mutuellement référence.
- C'est ce qui constitue l'architecture d'une application OO et que retranscrit le diagramme de classe.
- L'association *simple* symbolise ces références.
- l'agrégation et la composition sont des raffinements sémantiques de l'association simple (notion de propriété, de composant et de tout)...

L'association simple

- Il y a *association* entre une classe A et une classe B si A contient les informations permettant d'atteindre B :



- On parle de navigation de A vers B :

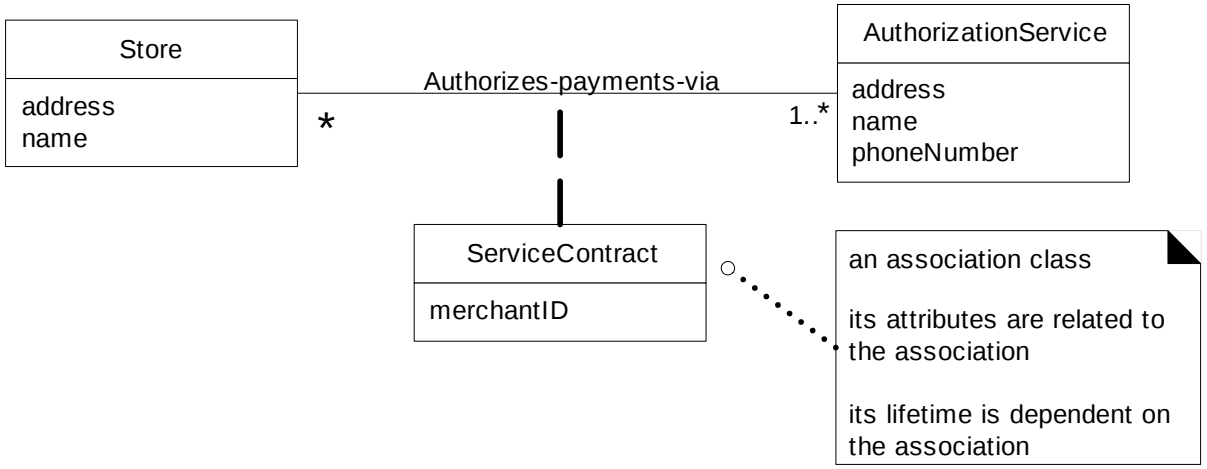
```
1 LogAbstraction la=...
2 LoggerBridge lb=la.getLogBridge(...);
```

- Les relations sont sujettes à multiplicité (ex : collections, références nulles, ...) et peuvent être bidirectionnelles.

Notation	Description
1	**No More than One**
0..1	**Zero or One**
*	**Many**
0..*	**Zero or Many**
1..*	**One or Many**

Classes d'association

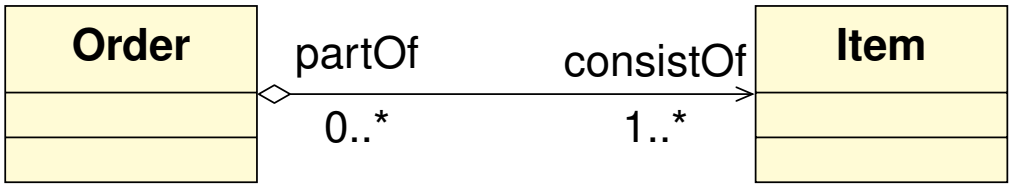
- Les associations peuvent être représentées par des *classes d'association* :



- Une ligne en pointillés relie la ligne de la relation à la classe qui la représente.
- La classe de relation permet de munir la relation d'attributs et de méthodes.

L'agrégation

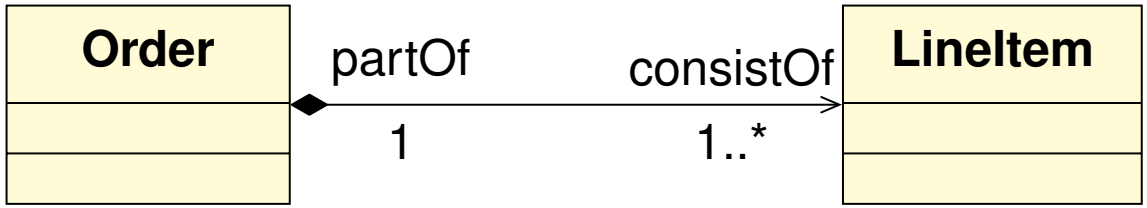
- Une classe A *agrège* une classe B lorsque A possède une (ou plusieurs) référence(s) de type B.
- Le losange identifie la classe qui représente l'*agrégat* (le « tout ») :



- Un composé peut appartenir plusieurs agrégats.
- Ce type de relation comprend aussi des cardinalités.
- Implémentation identique à l'association simple :

```
1  class Order {
2  private:
3      List<LineItem> items;
4  public:
5      Order() {
6          items=new List<LineItem>();
7          items.add(new LineItem("Free Gift", 0));
8      }
9  };
```

La composition



- La composition est une agrégation avec une sémantique plus forte :
  - le composé ne partage pas ses composants,
  - les cycles de vie des composés et composants sont liés (le composant n'existe pas sans son composé).
- Le losange est noircit,
- la cardinalité du coté composé est toujours au maximum de 1.

La composition

- La composition peut être réalisée naturellement en C++ en incluant les composants « physiquement »dans le composé :

```
1 class A {
2 private:
3     B b;
4 public:
5     A() :b() {}
6 };
```

Implémentation de la composition

Quelque soit le langage

La composition est un concept qui nécessite des précautions pour être implémentée :

- s'assurer que B est privée,
- instancier B depuis A (ou copier un référence),
- prendre soin de ne jamais retourner B, une copie si nécessaire.

```
1  public class A {
2       private B b;
3
4       public A() { b=new B(); }
5       public setB(B p) { b=p.clone(); }
6
7       getB() { ^^Ireturn b.clone(); }
8  }
```

- En résumé, s'assurer que A possède l'unique référence pointant vers B. (Action du ramasse-miette)

Le Polymorphisme

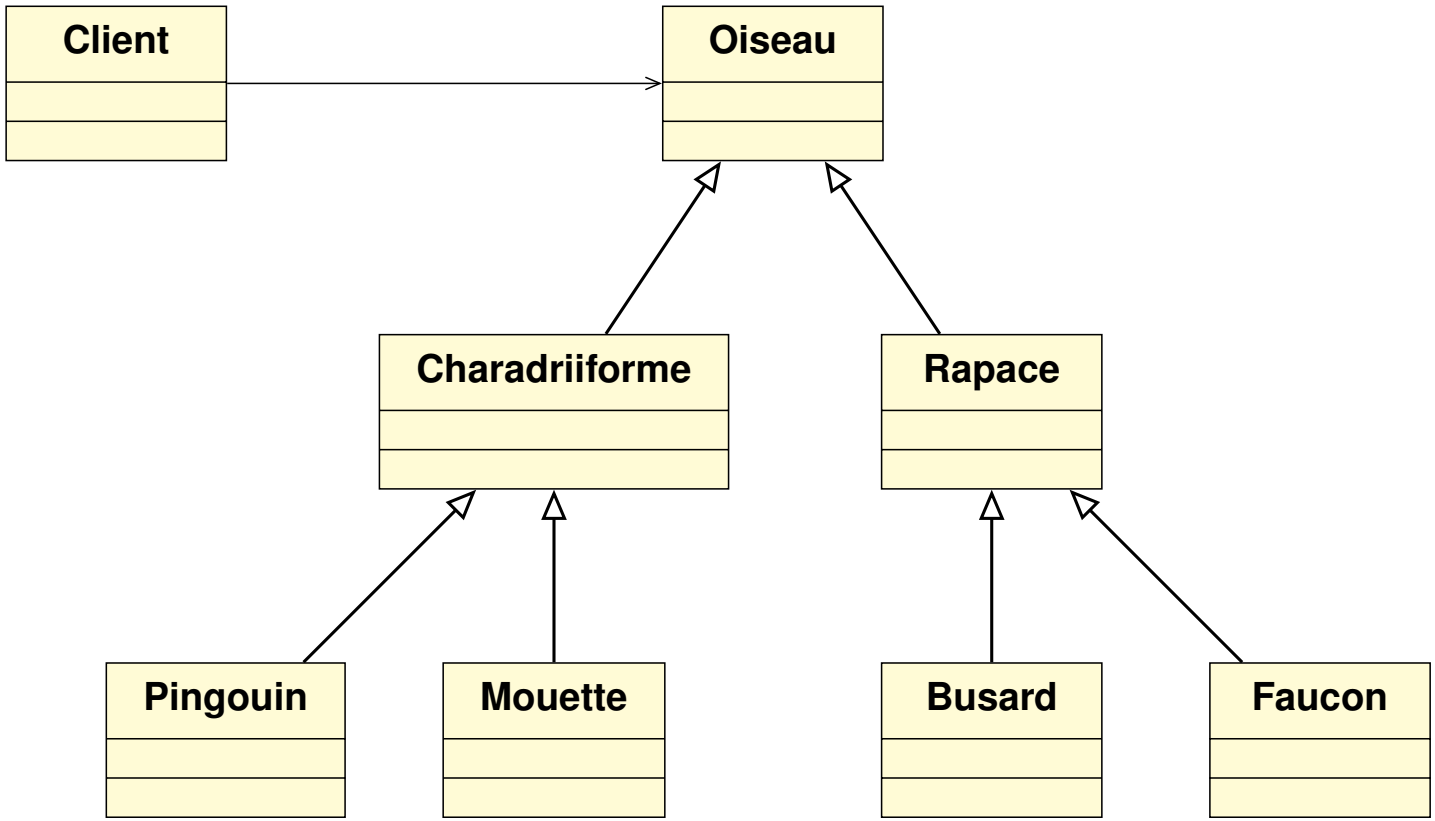
- Le polymorphisme (par sous-typage [3]) repose sur les associations.
- Il consiste à distinguer au niveau des références leur type **statique** de leur type **dynamique**
- i.e. le type de l'objet pointé n'est pas nécessairement le même que celui utilisé au moment de la déclaration de la référence :

```
1 Animal ani;
2 ani=new Penguin();
```

- le sous-typage s'étend aux interfaces. (Polymorphisme au delà de la taxonomie)

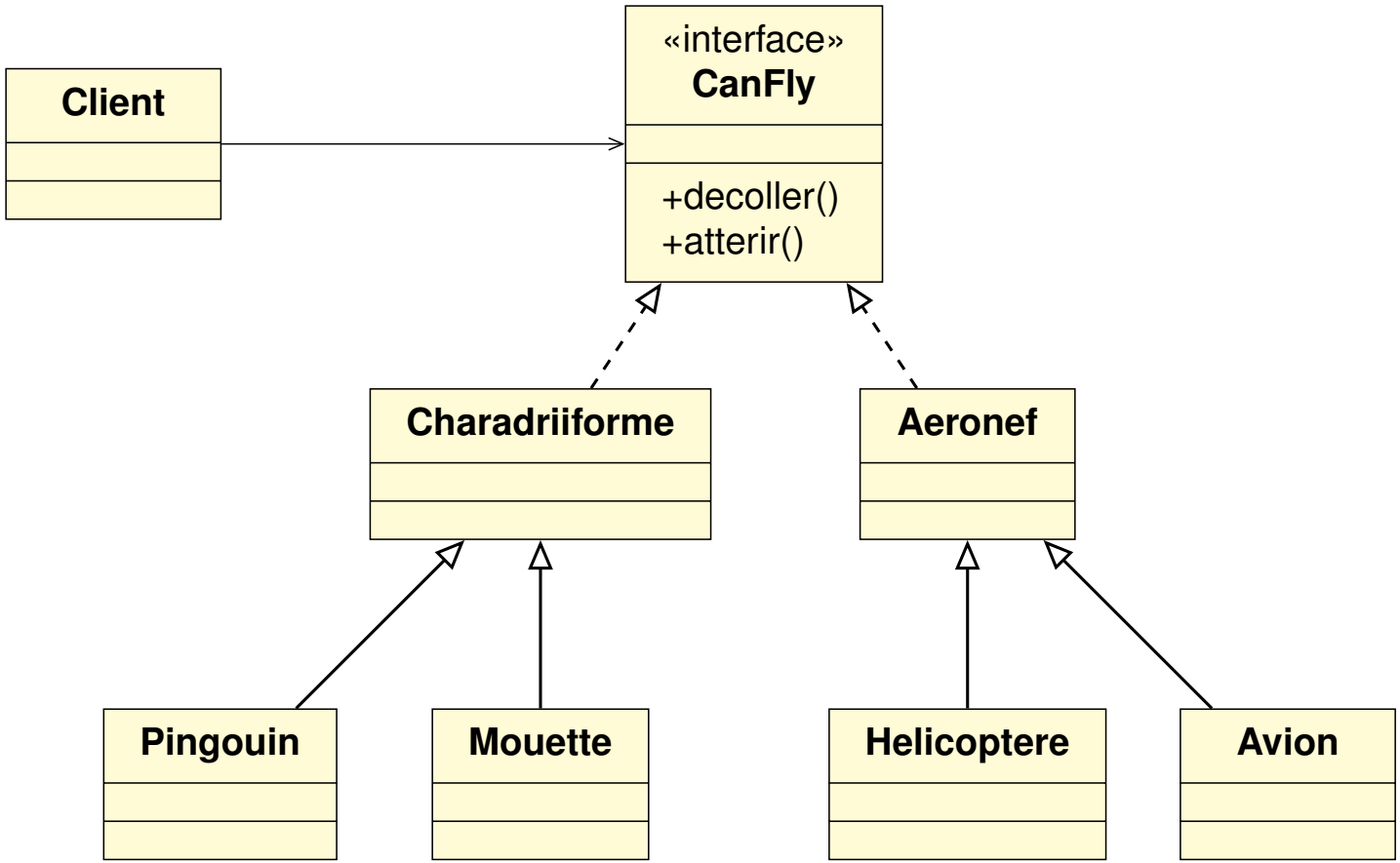
3. par opposition au polymorphisme ad hoc vu précédemment.

Polymorphisme et interfaces



- Si le type de la référence est une classe, alors le type de l'instance appartient à l'arborescence d'héritage de cette classe.

Polymorphisme et interfaces



- Si le type de la référence est une interface, alors le type de l'instance appartient la forêt des arborescences d'héritage des classes implémentant cette interface.

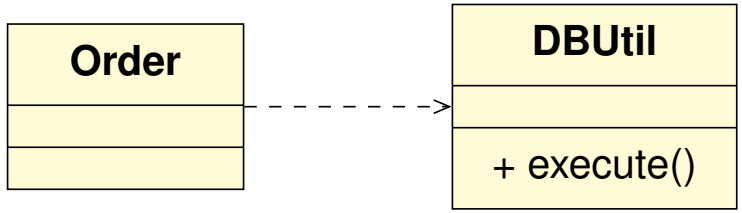
Les dépendances

- Une dépendance indique qu'une modification de la cible aura un impact sur la source.

Une classe *C* dépendra d'une classe (ou une interface) *S* lorsque

- *C* envoie un message à *S*,
- *C* contient *S* dans ses données,
- *C* utilise *S* comme paramètre d'une méthode.

Les dépendances



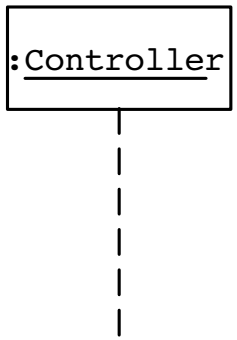
- Sur cet exemple, une modification de *execute()* (et donc de l'interface de *DBUtil*) aura un impact sur le fonctionne de la classe *Order* qui l'utilise.
- Utile pour indiquer une dépendance en l'absence d'association/héritage/implémentation.

Les diagrammes de séquences

Diagrammes de séquences

Acteurs

- Les diagrammes de séquences complètent les diagrammes de classes en exhibant l'ordre des flux d'évènements entre les différentes acteurs.
- Il permet de modéliser les comportements typiques du système.
- La chronologie se lit de haut en bas.
- Chaque acteur est représenté par une boite étiquetée par : *[<nom_du_rôle>] : [<Nom_du_type>]*,
- sa ligne de vie est représentée en pointillé :

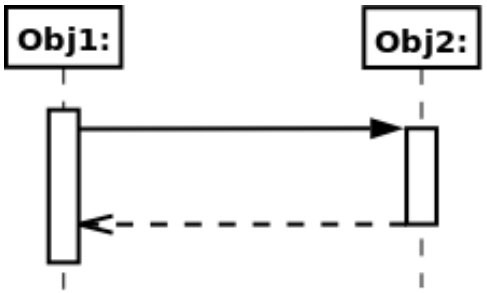


Diagrammes de séquences

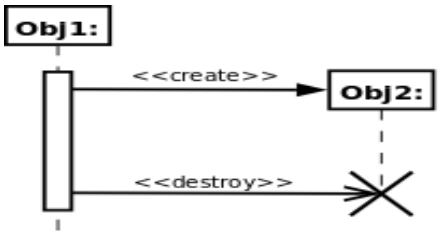
Évènements

- Messages asynchrones :

- Messages synchrones :

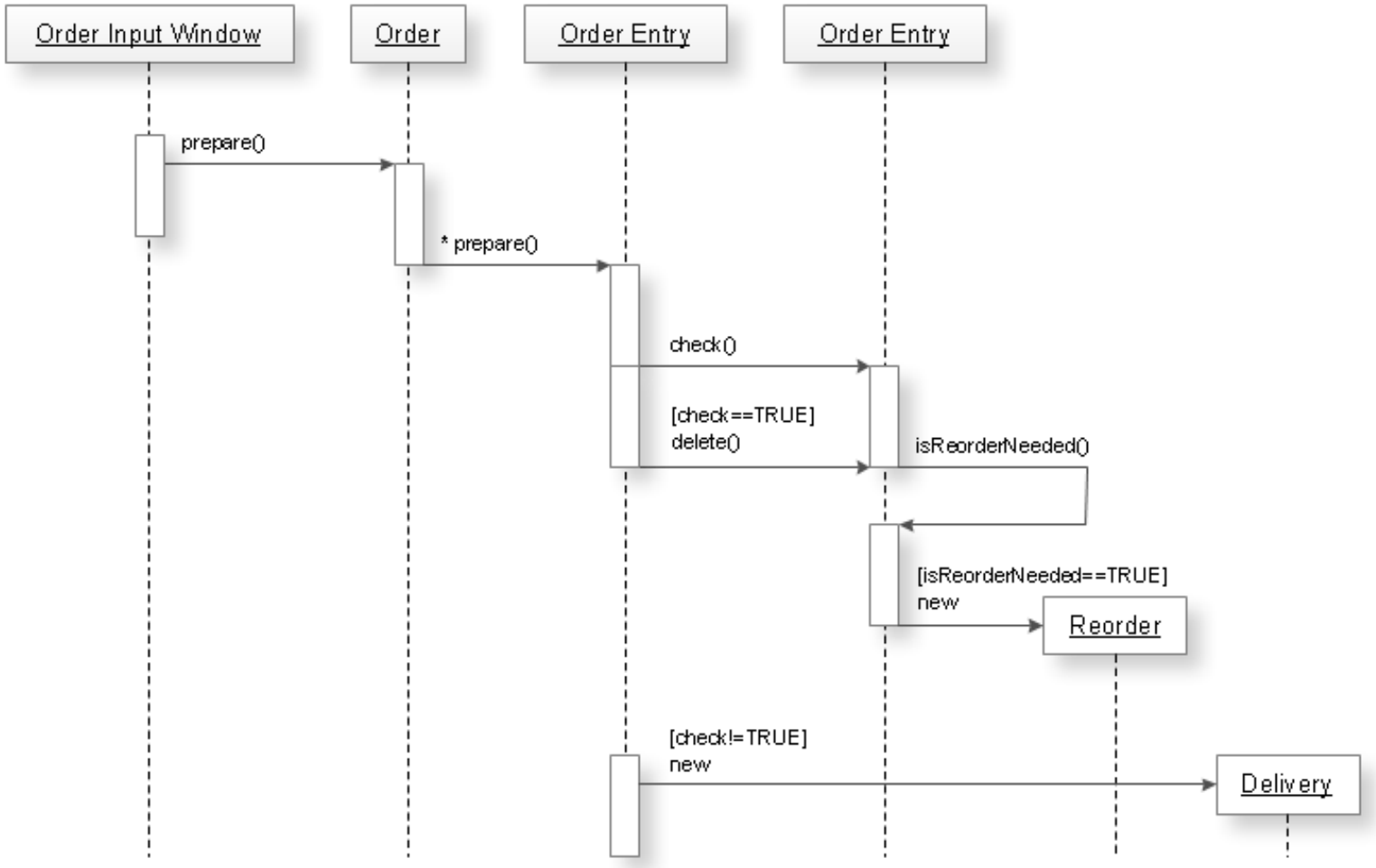


- Messages création/desctruction :



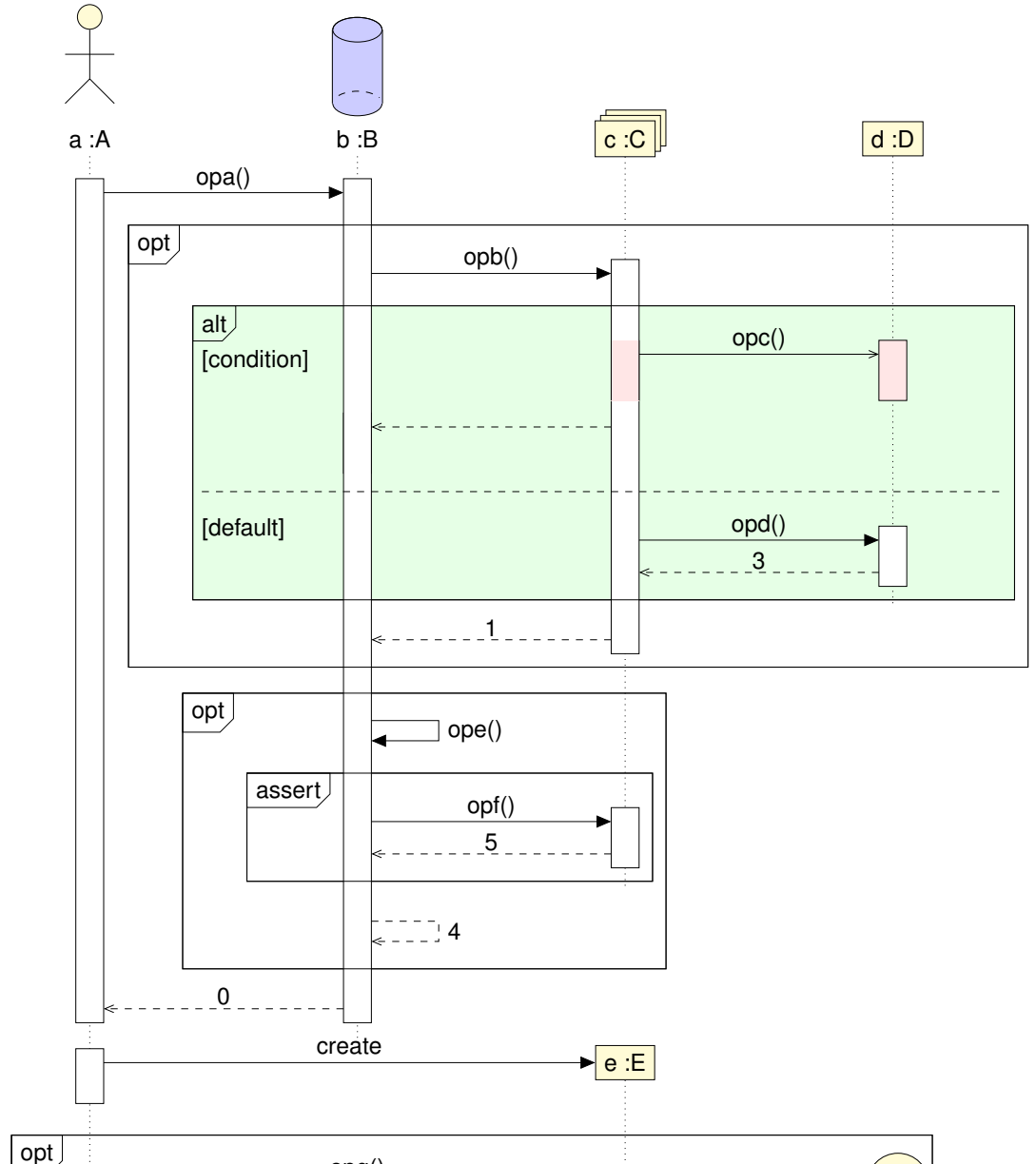
Diagrammes de séquences

Exemples



Diagrammes de séquences

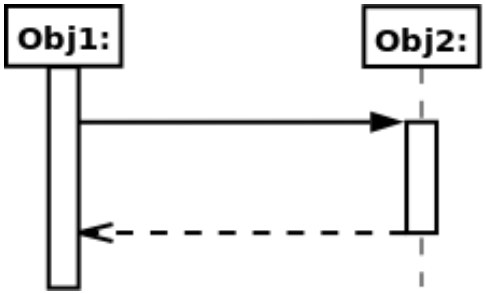
Exemples



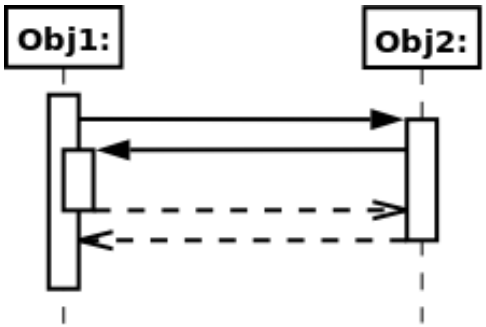
Diagrammes de séquences

Objets actifs

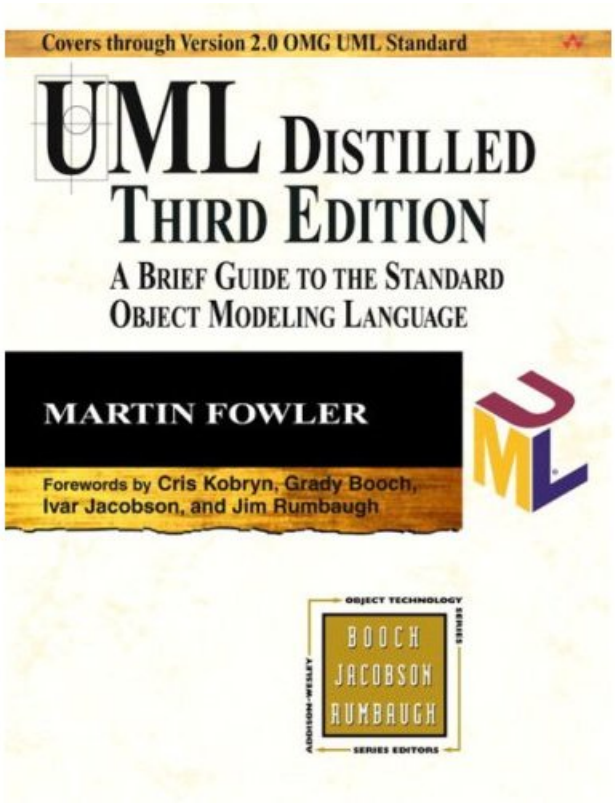
- Objets actifs et passifs :



- Exécutions simultanées (ré-entrance) :

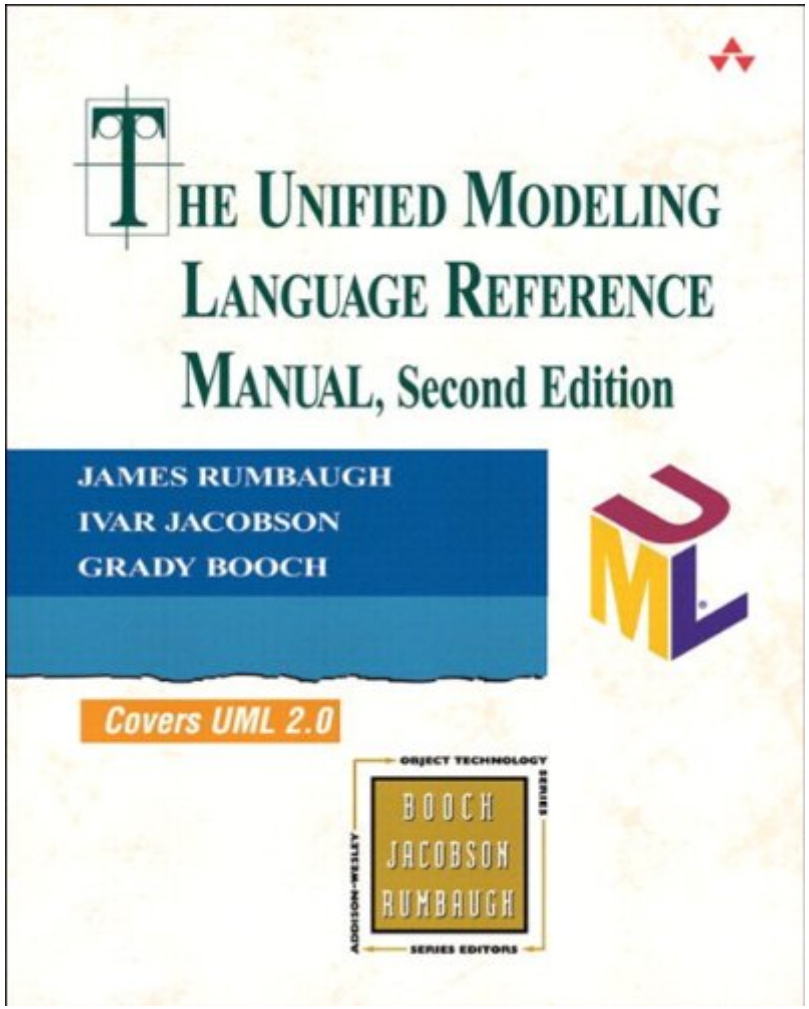


Pour aller plus loin...



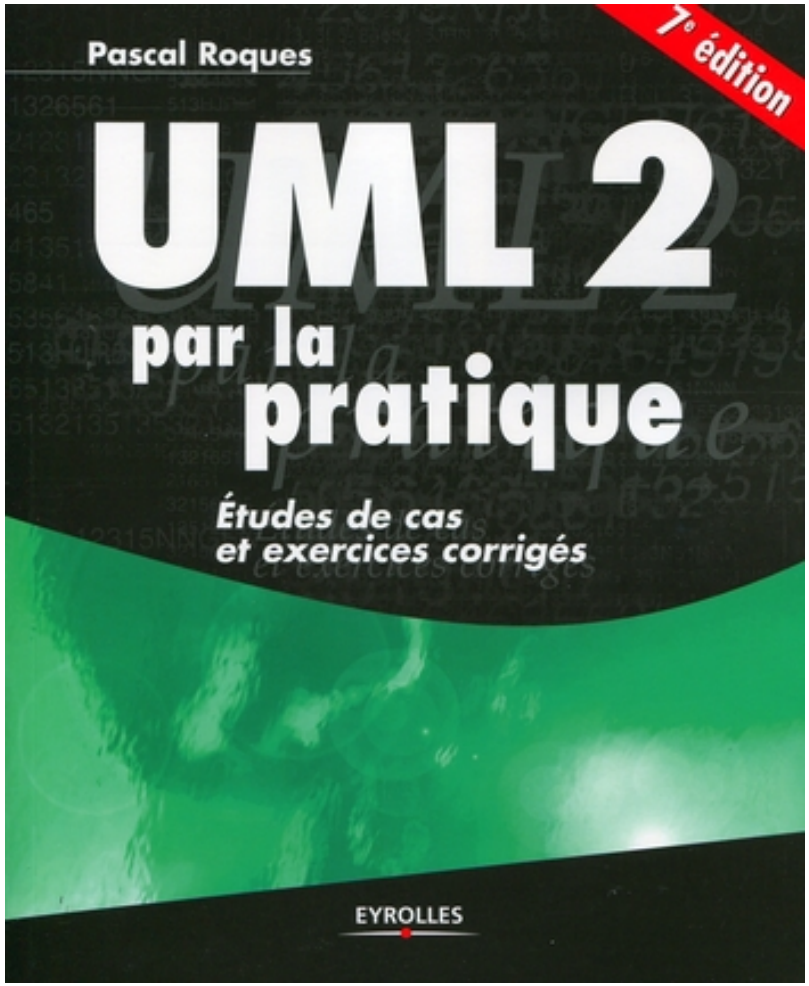
UML Distilled : A Brief Guide to the Standard Object Modeling Language, *Martin Fowler*, Addison Wesley.
ISBN-13 : 978-0321193681.

Pour aller plus loin...



The Unified Modeling Language Reference Manual, *James Rumbaugh, Ivar Jacobson, Grady Booch*, Addison-Wesley Professional. ISBN-13 : 978-0321718952.

Pour aller plus loin...



UML2 par la pratique,
Pascal Roques, Eyrolles.
ISBN-13 : 978-2212114805.