# Chapitre III
# Conception

# Plan du chapitre III

III. Conception

A. Visualisation des concepts (UML : Class)

B. Comportement dynamique (UML : Sequence, Communication)

1. Sequence diagrams : principes, définitions et exemples
2. Communication diagrams : principes, définitions et exemples

C. Patrons de conception élémentaires (UML : Component)

D. Visualisation des concepts, bis (UML : Class, Package)

1. Class diagrams : éléments avancés
2. Package diagrams : principes, définitions et exemples

E. Diagrammes UML et code Java

## Buts de la conception

But de la conception :

$\Rightarrow$ Définir une architecture logicielle statique et comportementale orientée objet permettant d'écrire facilement le code source de l'application.

Attention :

En phase de conception, on ne s'intéresse pas à l'écriture spécifique du code.

Pour cela :

$\rightarrow$ Architecture dynamique à l'aide de Sequence diagrams.

$\rightarrow$ Architecture statique à l'aide de Class et Package diagrams.

$\rightarrow$ Patrons de conception élémentaires pour les bases de la POO.

Class diagrams :

principes, définitions et exemples

# III.A. Class diagrams : principes

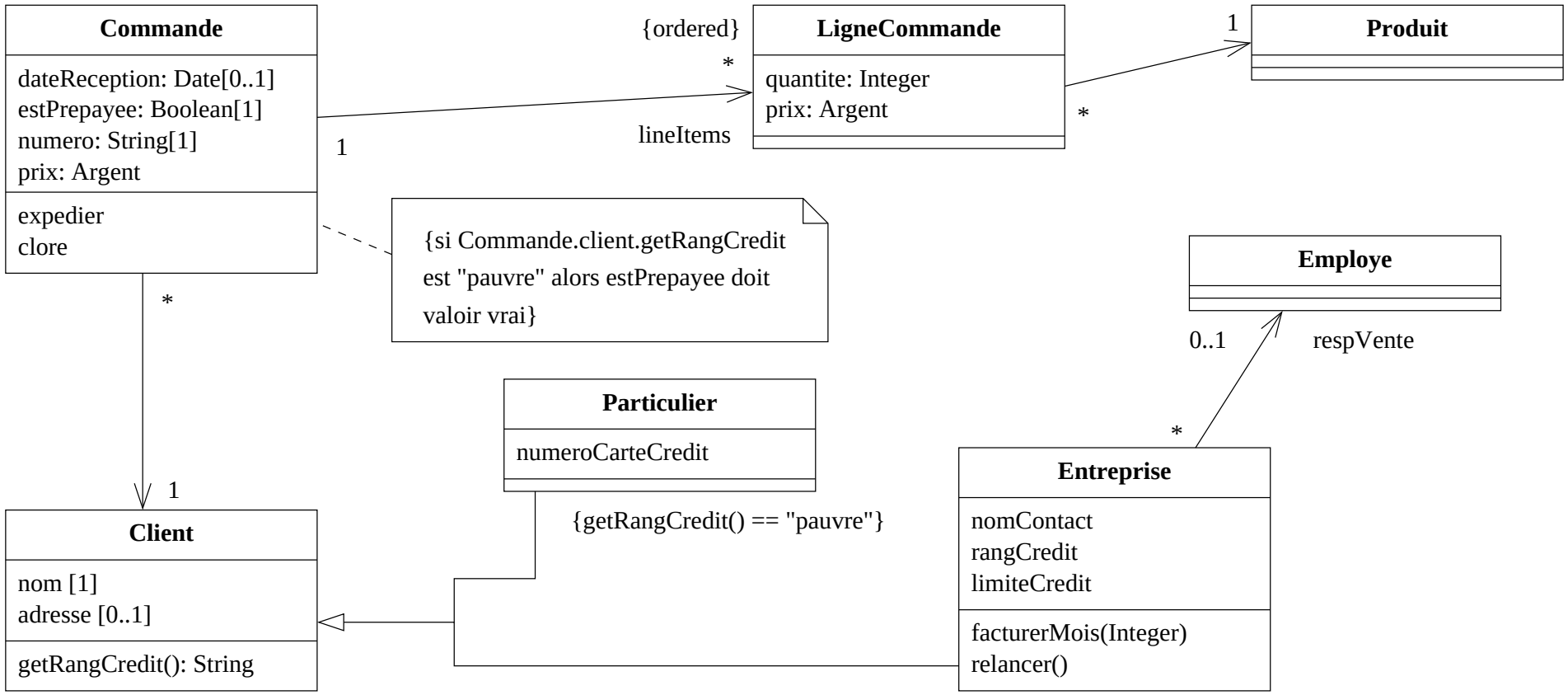
- Un Class diagram permet de représenter les types d'objets du système ainsi que les relations statiques entre ceux-ci.

- Le Class diagram constitue le noyau central de la conception de l'architecture logicielle.

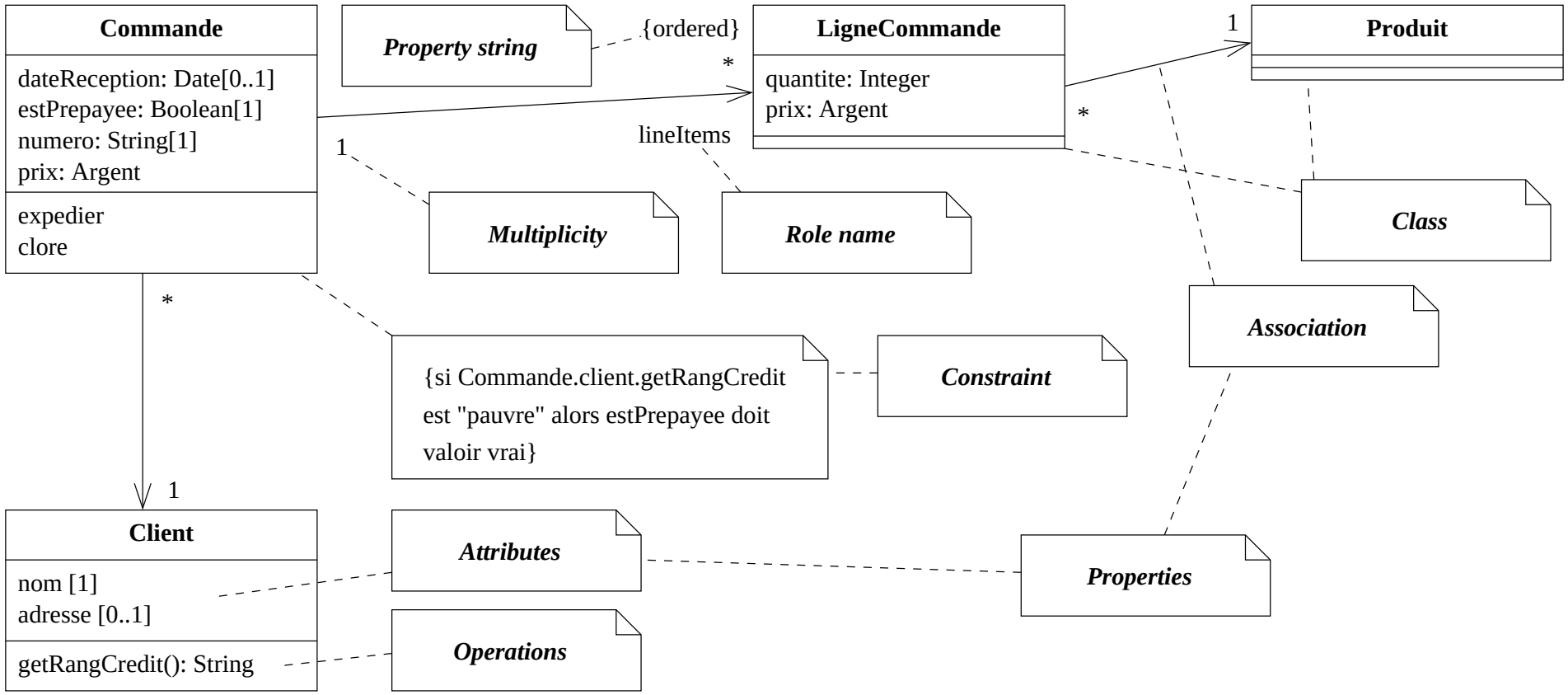
- C'est le diagramme le plus adapté (et utilisé) pour la génération de code.

- Aucune représentation de la dynamique n'y est possible, utiliser des Sequence diagrams.

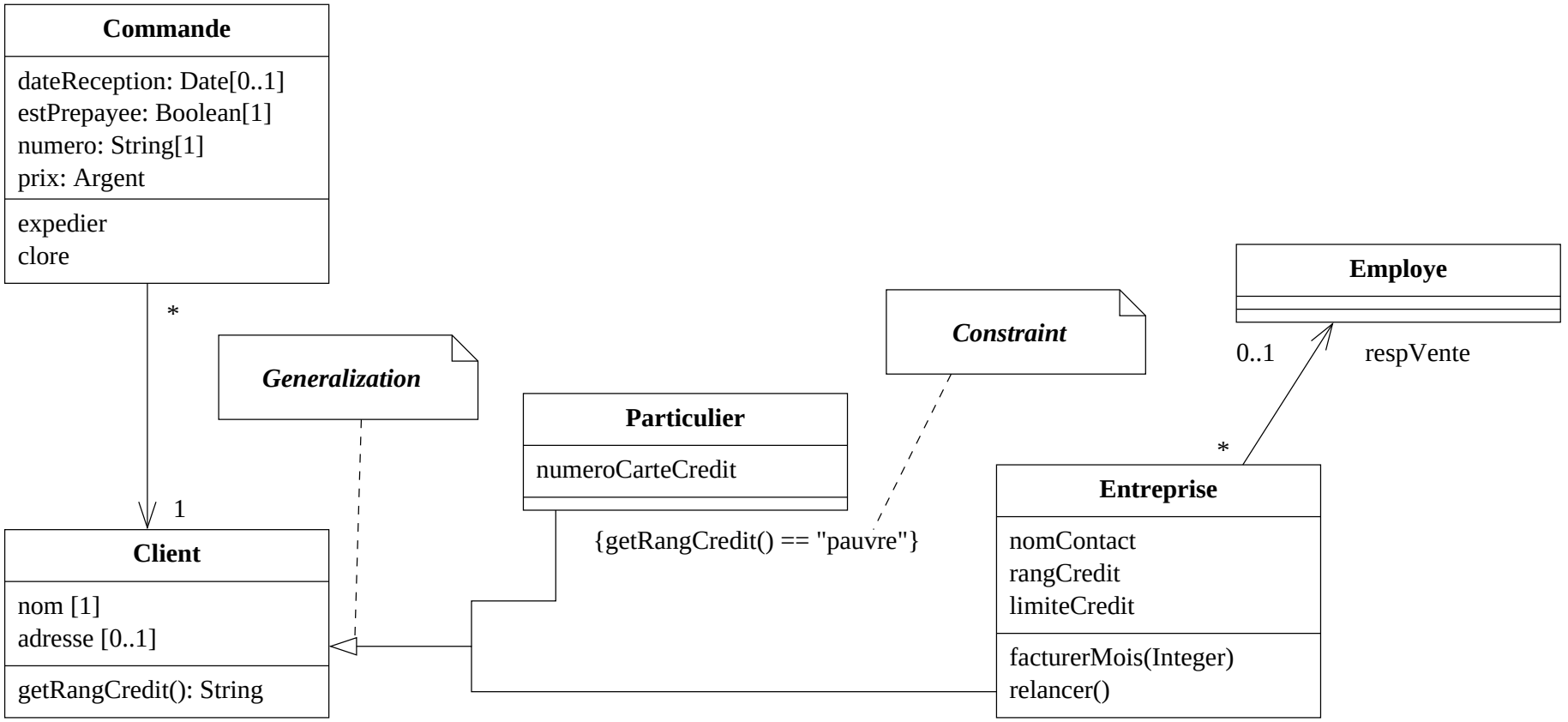
# III.A. Class diagrams : un exemple



# III.A. Class diagrams : définitions



# III.A. Class diagrams : définitions



# III.A. Class diagrams : définitions - Properties

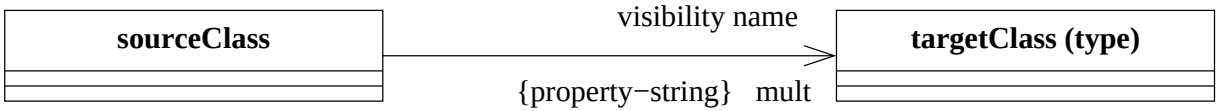
Les Properties sont soit des Attributes, soit des Associations, qui représentent tous deux la même réalité.

→ Attributes. Forme générale :

```
visibility name: type multiplicity = default {property-string}
```

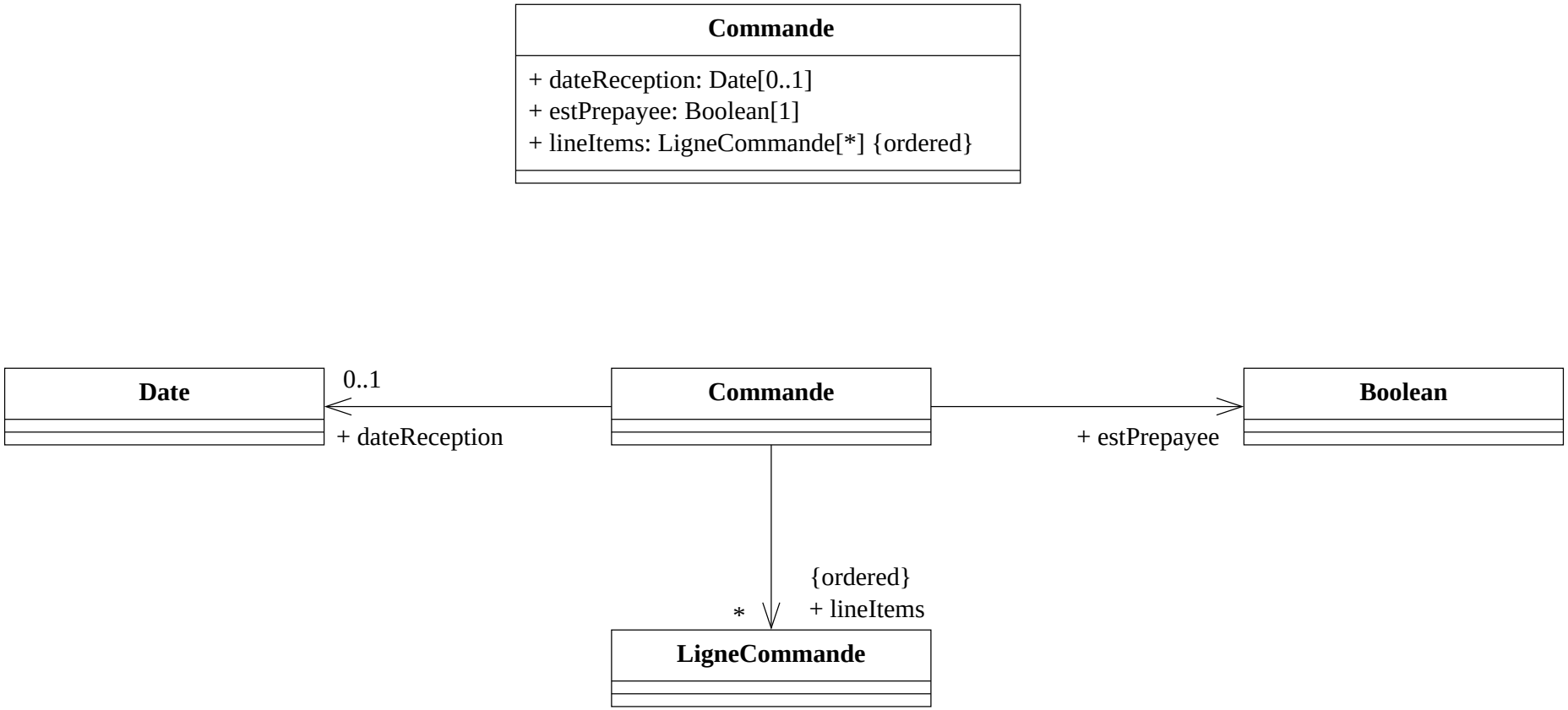
Exemple : `- nom: String [1] = "Robert" {readOnly}`

→ Associations. Forme générale :




→ Les associations peuvent être bidirectionnelles.

# III.A. Class diagrams : équivalence Attribute-Association



# III.A. Class diagrams : définitions - Properties

- Visibilité : public +, private -, éventuellement package ~ et protected #.
- Nom : le nom de l'attribut – la façon dont la classe se réfère à l'attribut.
- Type : le type de l'attribut – correspond à une restriction sur ce que l'attribut peut contenir.
- Valeur par défaut : utilisée si l'attribut n'est pas initialisé différement durant sa création.
- Le champ `{property-string}` permet d'indiquer des propriétés additionnelles : `readOnly`, `ordered`, `nonunique`, `bag`, etc.

# III.A. Class diagrams : remarques

- Attention : les visibilités changent de signification suivant les langages.
- Conseil : Ne pas les utiliser dans les diagrammes UML (ou seulement + et - pour marquer des différences pertinentes) et utiliser directement les mots-clés du langage cible si besoin.
- Par défaut, la visibilité doit être -, y compris dans le code généré. On fournit éventuellement ensuite des accesseurs.
- Attention : se méfier des classes ne contenant que des champs et leurs accesseurs, elles sont souvent le signe d'une programmation NON orientée objet.

# III.A. Class diagrams : définitions - Multiplicités

- Les plus courantes :

  `1`, `0..1`, `*`

- La forme générale : `borne_inf..borne_sup`

  $\texttt{borne\_inf} \in \mathbb{N}, \qquad \texttt{borne\_sup} \in \mathbb{N} \cup \{*\}$

- Raccourcis usuels : `x..x` est noté `x`, `0..*` est noté `*`
- Mots-clés :
  - `Optional` implique $\texttt{borne\_inf} = 0$.
  - `Mandatory` implique $\texttt{borne\_inf} \geq 1$.
  - `Single-valued` implique $\texttt{borne\_sup} = 1$.
  - `Multi-valued` implique $\texttt{borne\_sup} > 1$, en général `*`.

# III.A. Class diagrams : définitions - Operations

- Forme générale :

```
visibility name(parameter-list) : return-type {property-string}
```

- `parameter-list`. Forme générale :

```
direction name: type = default-value
```

- Exemple :

```
+ soldeDu(date:Date) : Argent
```

- Champs :
  - → `return-type` est le type du résultat de l'opération.
  - → `direction` indique si le paramètre est en entrée (`in`) (par défaut), en sortie (`out`) ou les deux (`inout`).

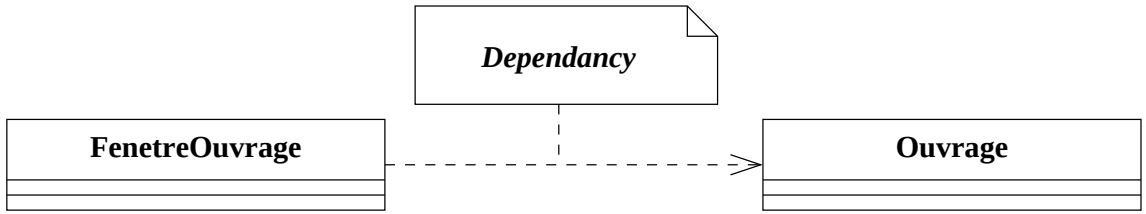
# III.A. Class diagrams : définitions - Operations

- il est souvent utile de distinguer les opérations qui modifient l'état du système de celles qui ne le font pas.

- Utiliser le `property-string` : `{query}` pour indiquer une opération qui renvoie une valeur d'une classe sans changer l'état du système.

- Par exemple : on peut échanger l'ordre dans lequel les `{query}` sont faites sans changer la sémantique du programme.

# III.A. Class diagrams : définitions - Dependancy




- Une dépendance indique que toute modification de la cible peut causer des modifications de la source.
- Beaucoup d'autres types de relation impliquent une dépendance (Association, Generalization, Composition, etc.).
- Il est très facile de déduire les dépendances à partir du code, les outils UML sont donc bien adaptés pour les obtenir.
- Conseil de conception : Minimisez les dépendances ! Particulièrement lorsqu'elles tarversent de larges zones du système. Évitez aussi les cycles.

# III.A. Class diagrams : définitions - Dependancy

| Mot-clé | Signification |
|---|---|
| «call» | La source appelle une opération de la cible. |
| «create» | La source crée une instance de la cible. |
| «derive» | La source est dérivée de la cible. |
| «instanciate» | La source est une instance de la cible. |
| «permit» | La cible autorise la source à accéder à ses fonctionnalités privées. |
| «realize» | La source implante une spécification ou interface définie par la cible. |
| «substitute» | La source peut être substituée à la cible. |
| «use» | La source a besoin de la cible pour son implantation. |

# III.A. Class diagrams : définitions - Contraintes

- Les contraintes peuvent être ajoutées n'importe où.

- Elles sont notées entre accolades, par exemple :

```
{getRangCredit() == "pauvre"}
```

- Elles peuvent être exprimés dans n'importe quel langage : langage naturel, langage de programmation, le langage de contrainte objet (OCL) d'UML.

# B. Comportement dynamique

Attention : ne pas négliger les vues dynamiques du systèmes !
Elles sont souvent porteuses d'informations capitales pour la conception et l'implantation.

Deux diagrammes UML : Sequence et Communication.

## 1. Sequence diagrams :

## principes, définitions et exemples

## III.B.1. Sequence diagrams : principes

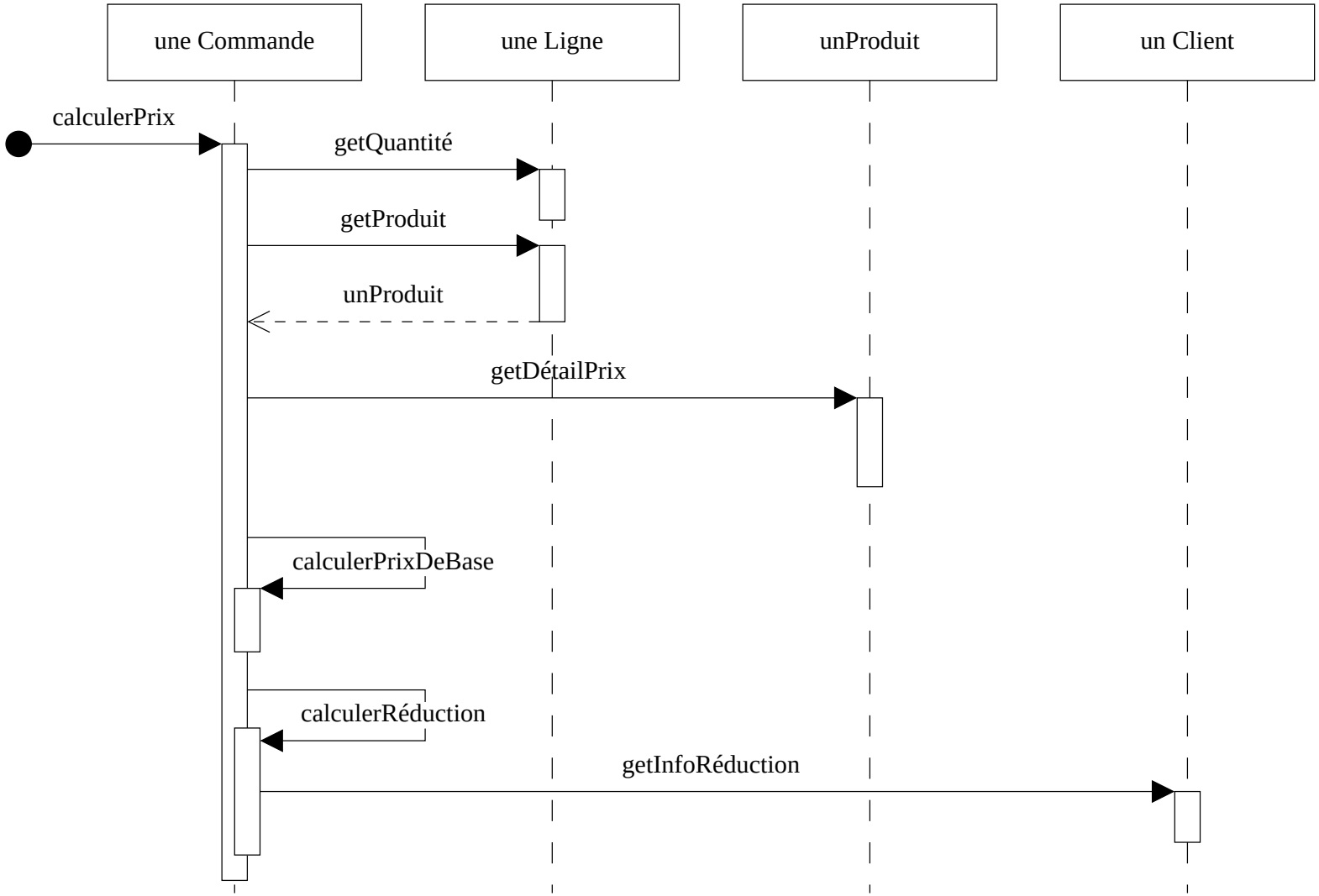
- Les diagrammes d'interaction décrivent comment les objets collaborent dans un certain comportement.

- Un Sequence diagram capture le comportement d'un scénario, montrant des exemples d'objets et de messages passés entre eux.

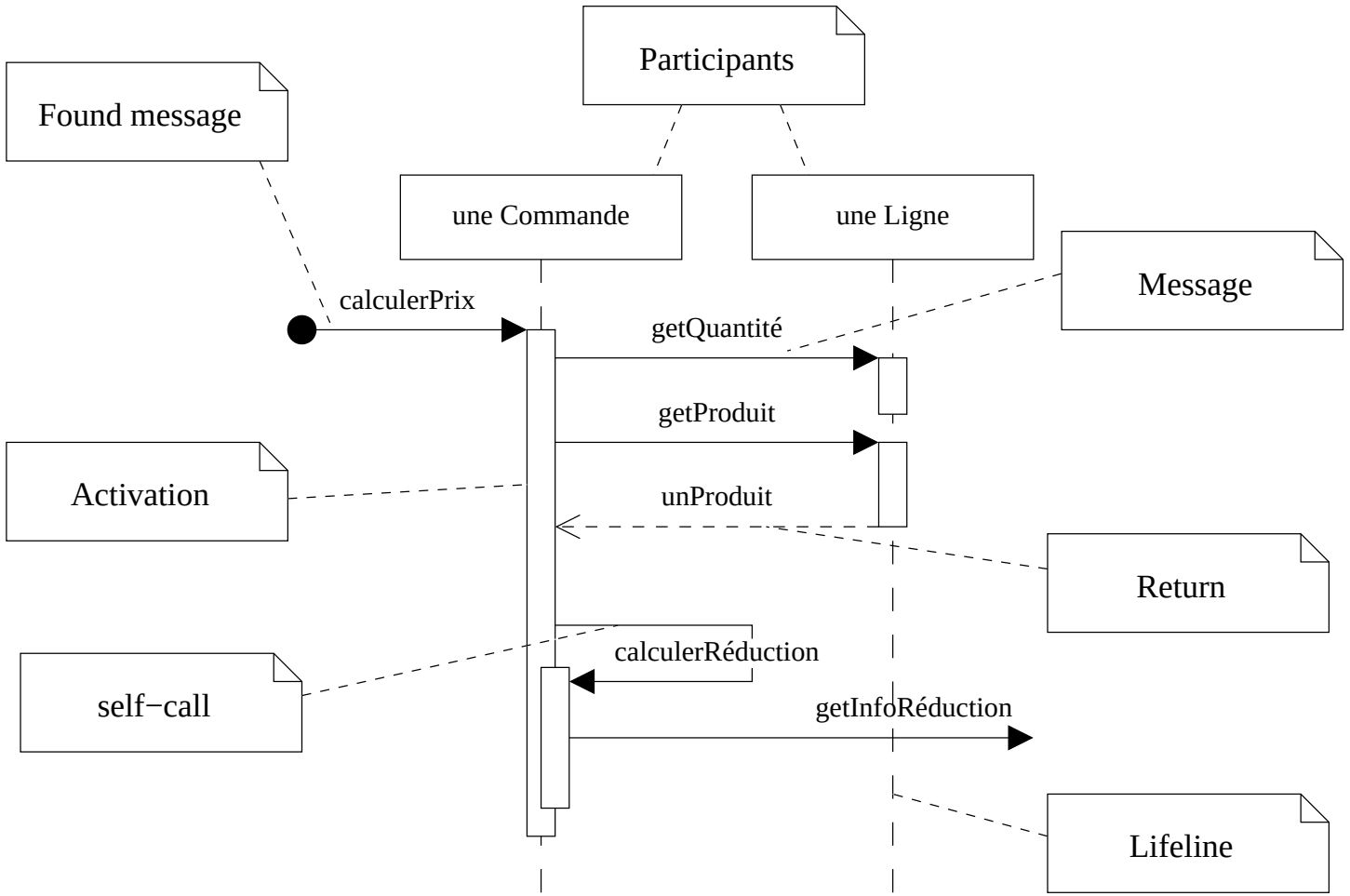
- Le grand avantage du Sequence diagram est d'illustrer comment les participants interagissent en montrant "qui fait quoi".

- Les Sequence diagrams ne sont pas très bons pour représenter la logique de contrôle, préférer pour cela les Activity diagrams ou le code lui-même.

# III.B.1. Sequence diagrams : un exemple



# III.B.1. Sequence diagrams : définitions



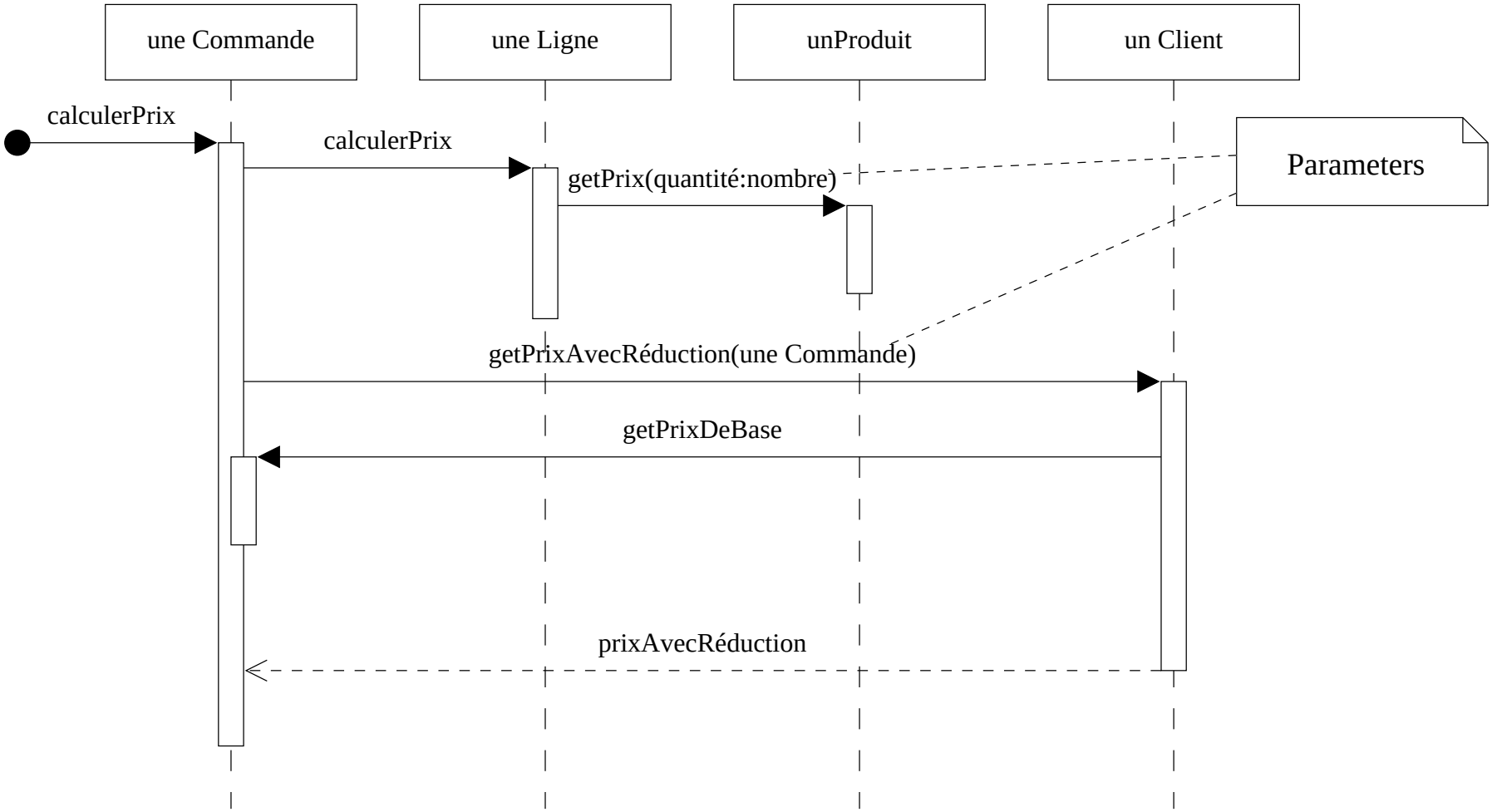
# III.B.1. Sequence diagrams : définitions

- Lecture de haut en bas, le long des lignes de vie en suivant les envois de messages.

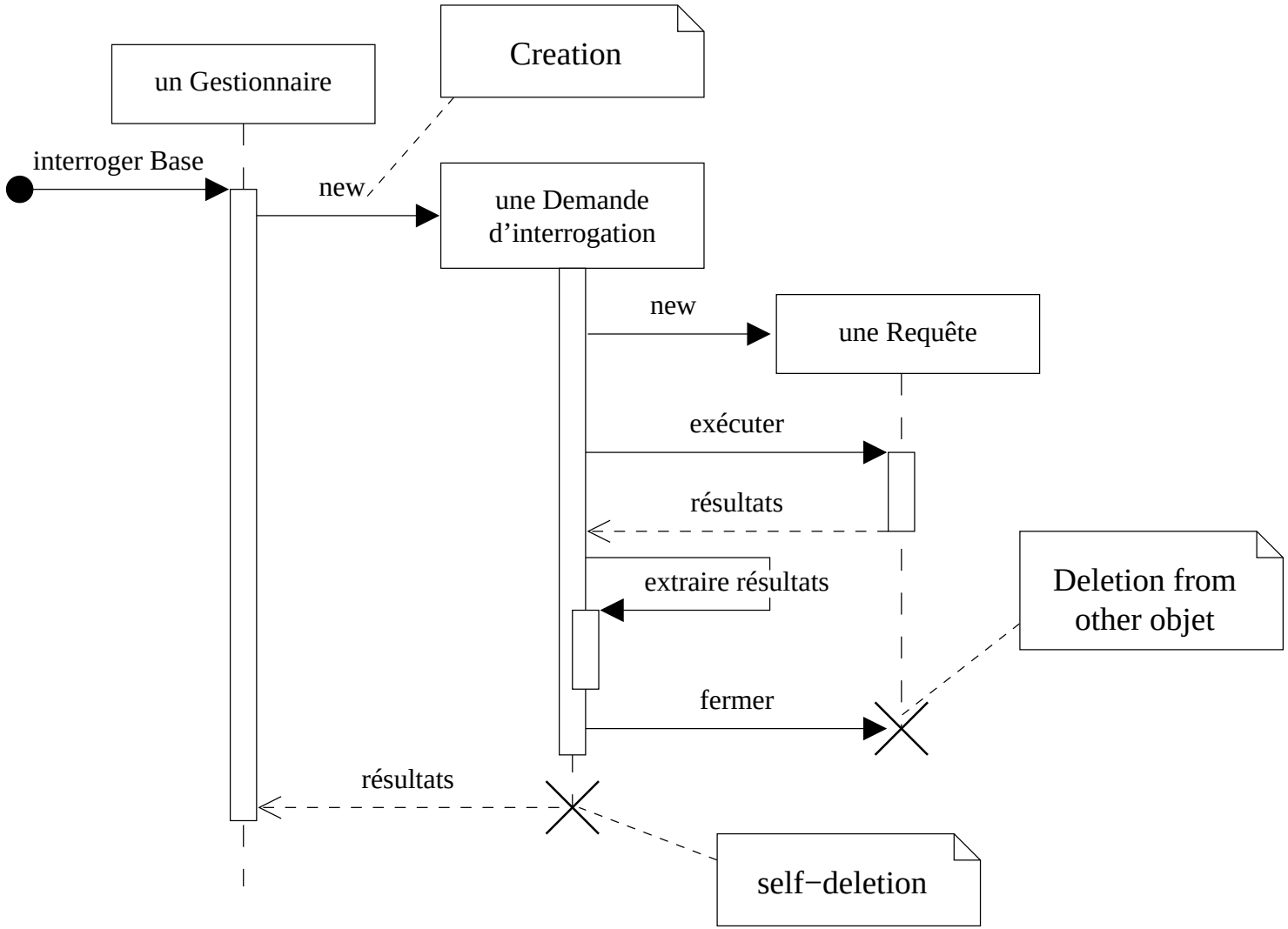
- Lecture de gauche à droite pour la création d'objets.

- La flèche de retour est optionnelle : l'utiliser lorsqu'elle apporte des informations supplémentaires.

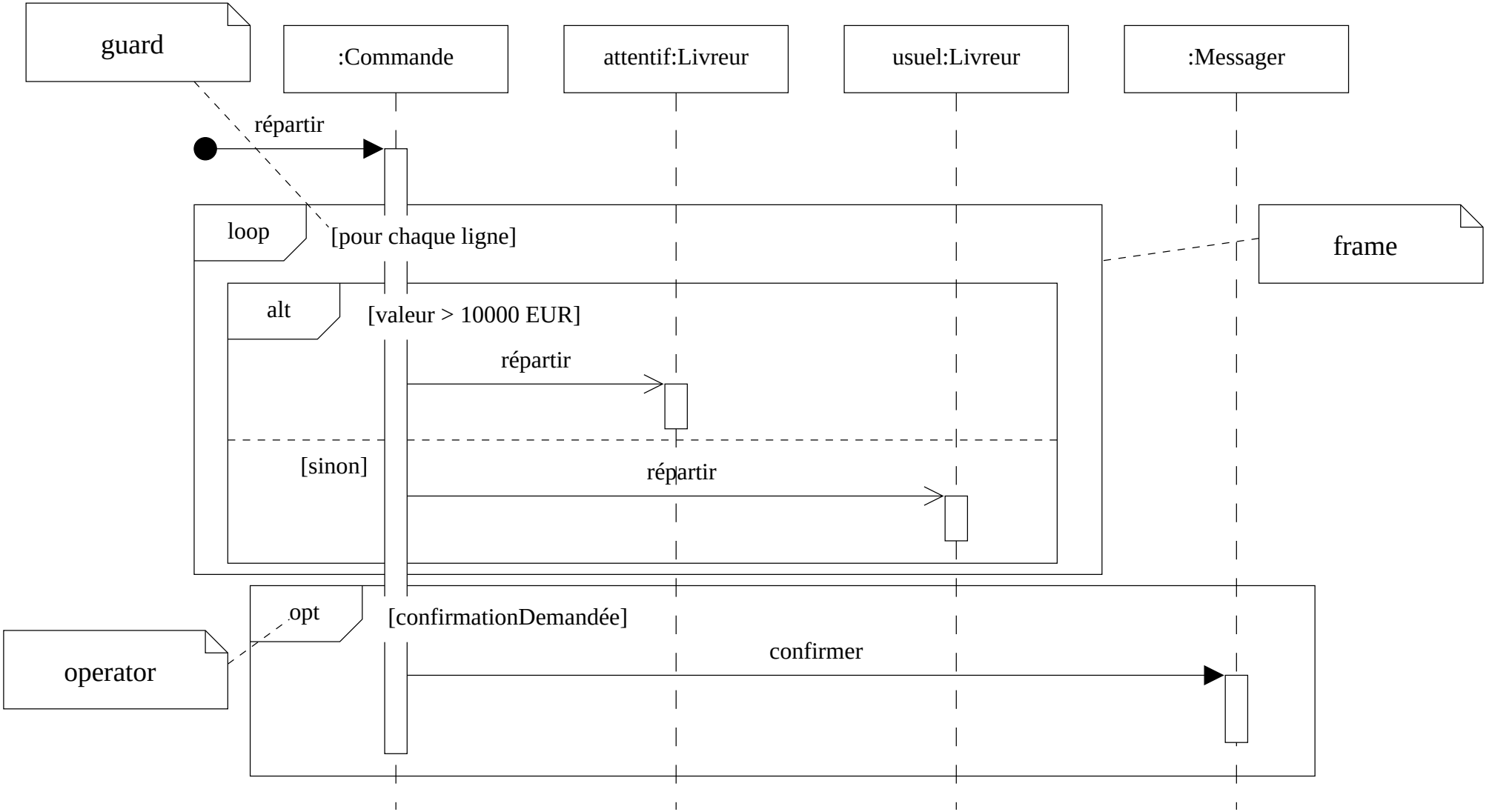
# III.B.1. Sequence diagrams : un exemple



# III.B.1. Sequence diagrams : création et destruction de participants



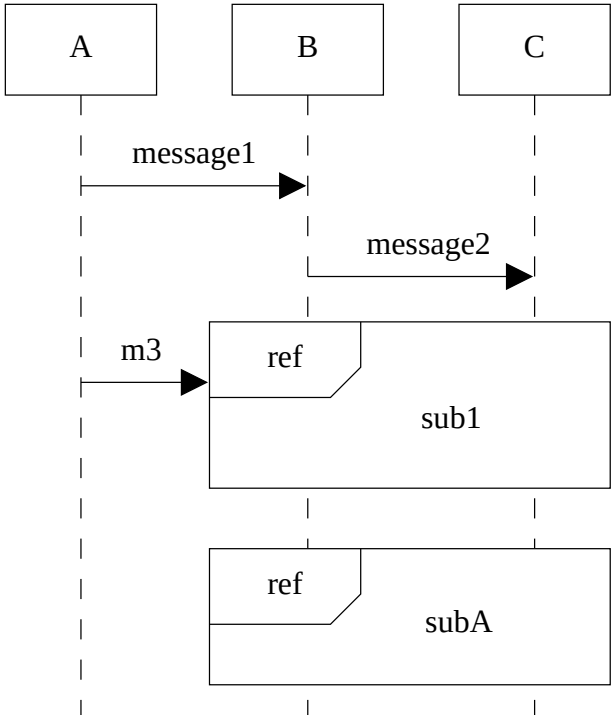
# III.B.1. Sequence diagrams : cadres d'interactions

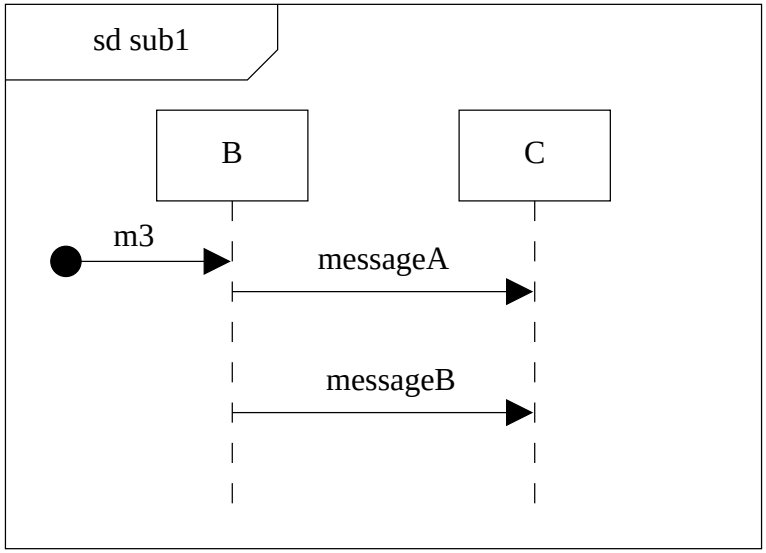


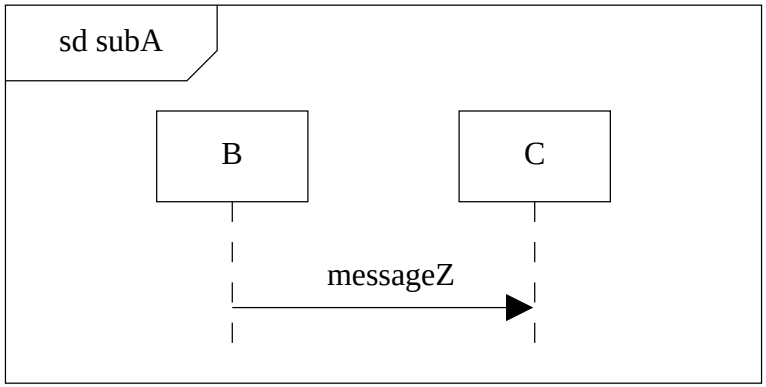
# III.B.1. Sequence diagrams : liste de opérateurs

| Opérateur | Signification |
|---|---|
| `alt` | Multiples fragments alternatifs ; seul celui dont la condition est vraie s'exécutera. |
| `opt` | Optionel ; le fragment s'exécute seulement si la condition est vraie. |
| `par` | Parallèle ; chaque fragment s'exécute en parallèle. |
| `loop` | Boucle ; le fragment peut s'exécuter plusieurs fois, et la guarde indique la base de l'itération. |
| `region` | Section critique ; le fragment ne peut avoir qu'un seul thread l'exécutant à chaque instant. |
| `neg` | Négatif ; le fragment montre une interaction invalide. |
| `ref` | Référence ; se réfère à une interaction définie sur un autre diagramme. Le cadre est tracé pour couvrir les lignes de vie impliquées dans l'interaction. |
| `sd` | Sequence diagram ; utilisé pour entourer le diagramme entier, si l'on veut. |

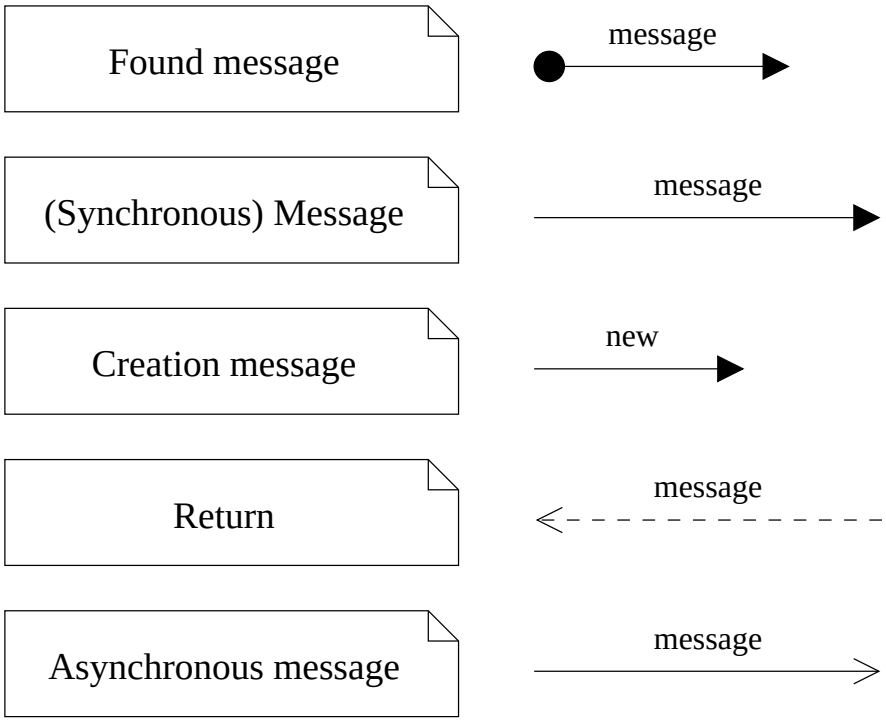
# III.B.1. Sequence diagrams : cadres “ref”



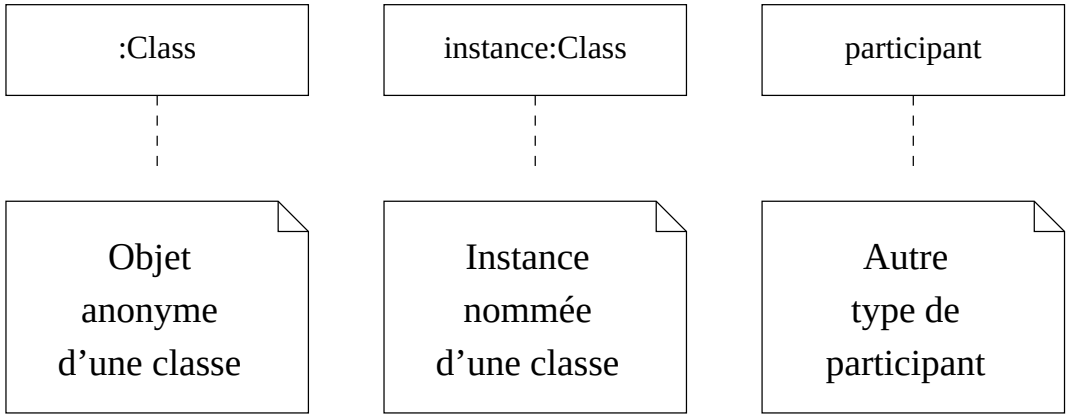



# III.B.1. Sequence diagrams : différentes flèches



# III.B.1. Sequence diagrams : types de participants



## III.B.1. Sequence diagrams : en guise de conclusion

- Les Sequence diagrams sont bons pour montrer les collaboration entre objets, moins pour donner une définition précise du comportement.
- Pour étudier les comportement d'un seul objet à travers plusieurs cas d'utilisation, utiliser un State machine diagram.
- Pour étudier un comportement sur plusieurs cas d'utilisation ou threads, utiliser un Activity diagram.
- Pour avoir une vue d'ensemble des connexions, utiliser un Communication diagram.
- Pour voir les contraintes temporelles, utiliser un Timing diagram.

## 2. Communication diagrams :

## principes, définitions et exemples

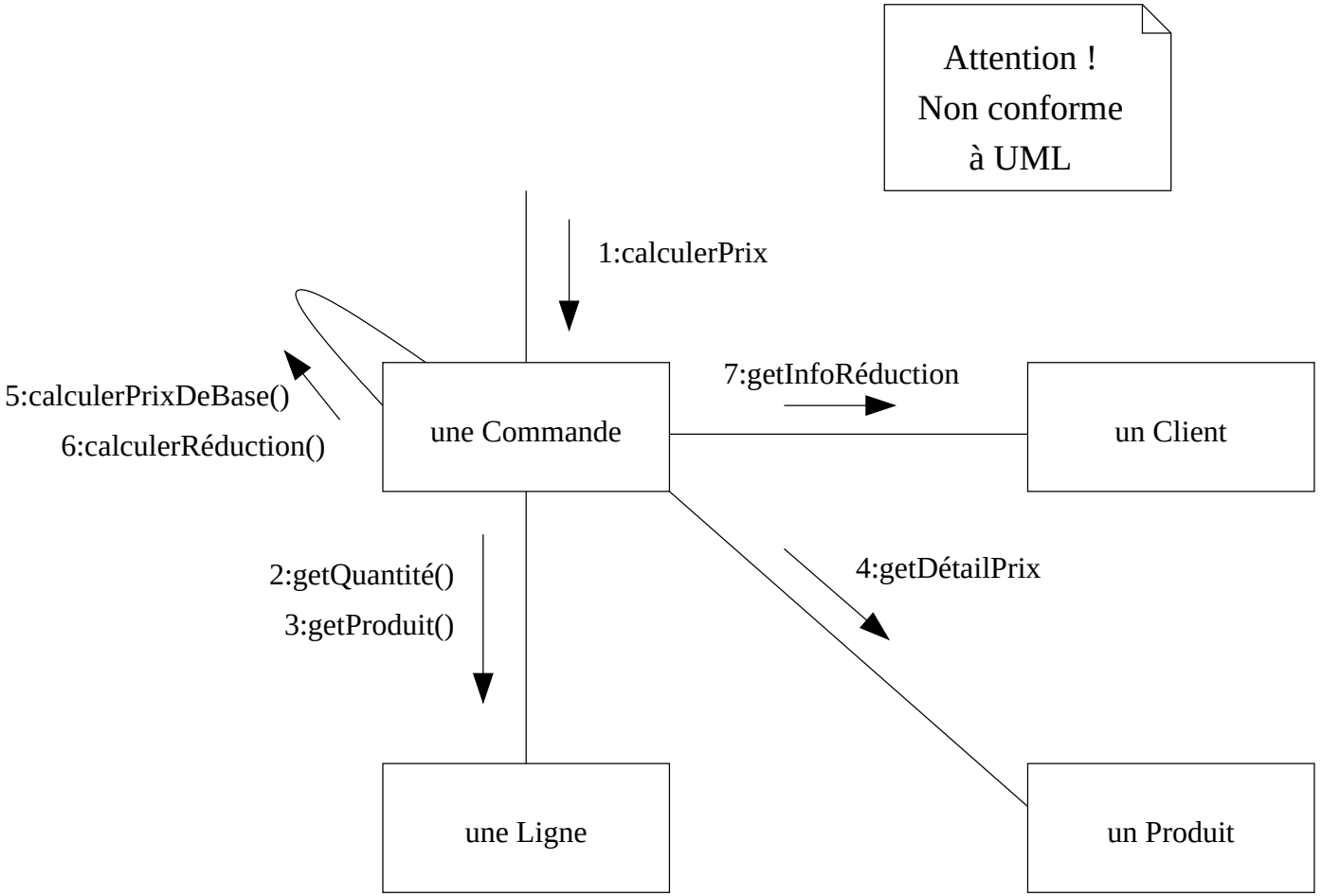
# III.B.2. Communication diagrams : principes

- Le Communication diagram fait ressortir les liens de données entre les différents participants d'une interaction.

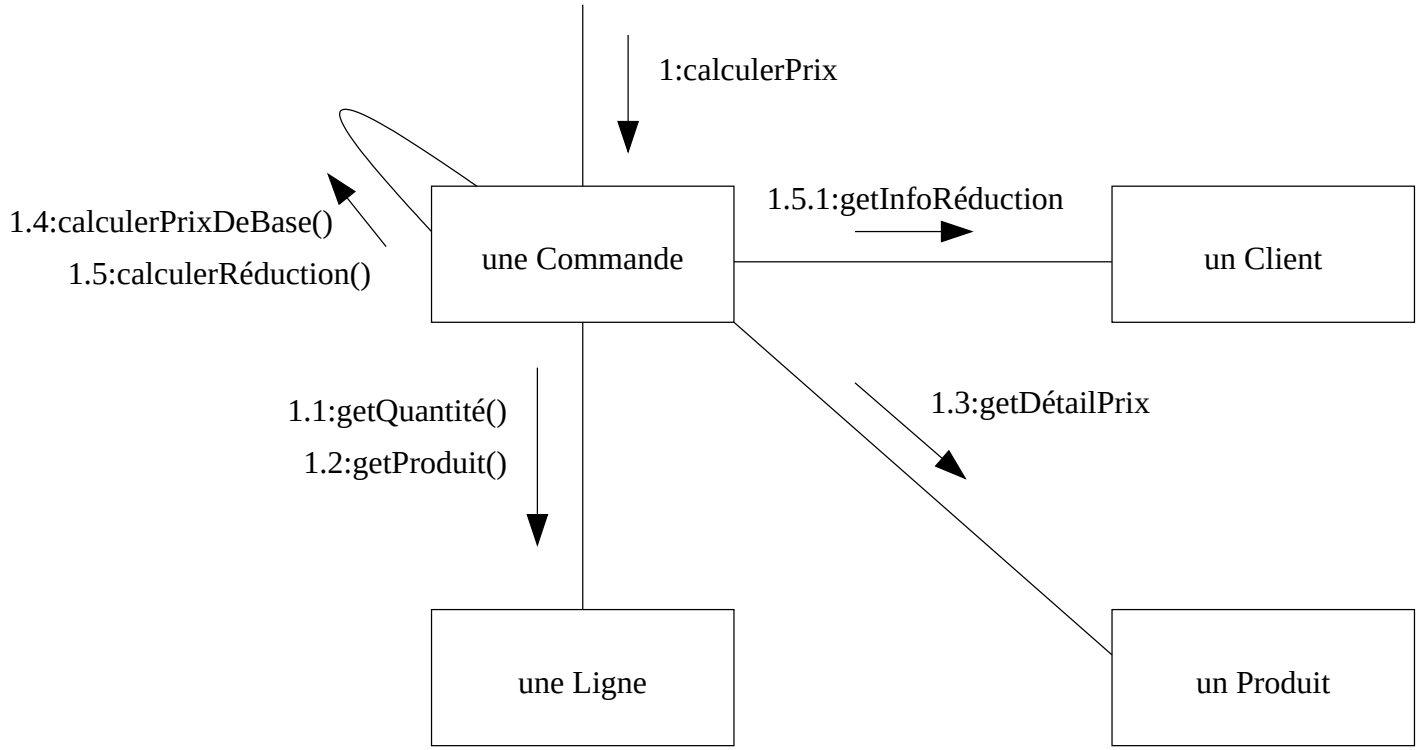
- Pas de ligne de vie verticale, mais un placement libre des participants.

- Possibilité de numéroter les messages pour en montrer le séquencement.

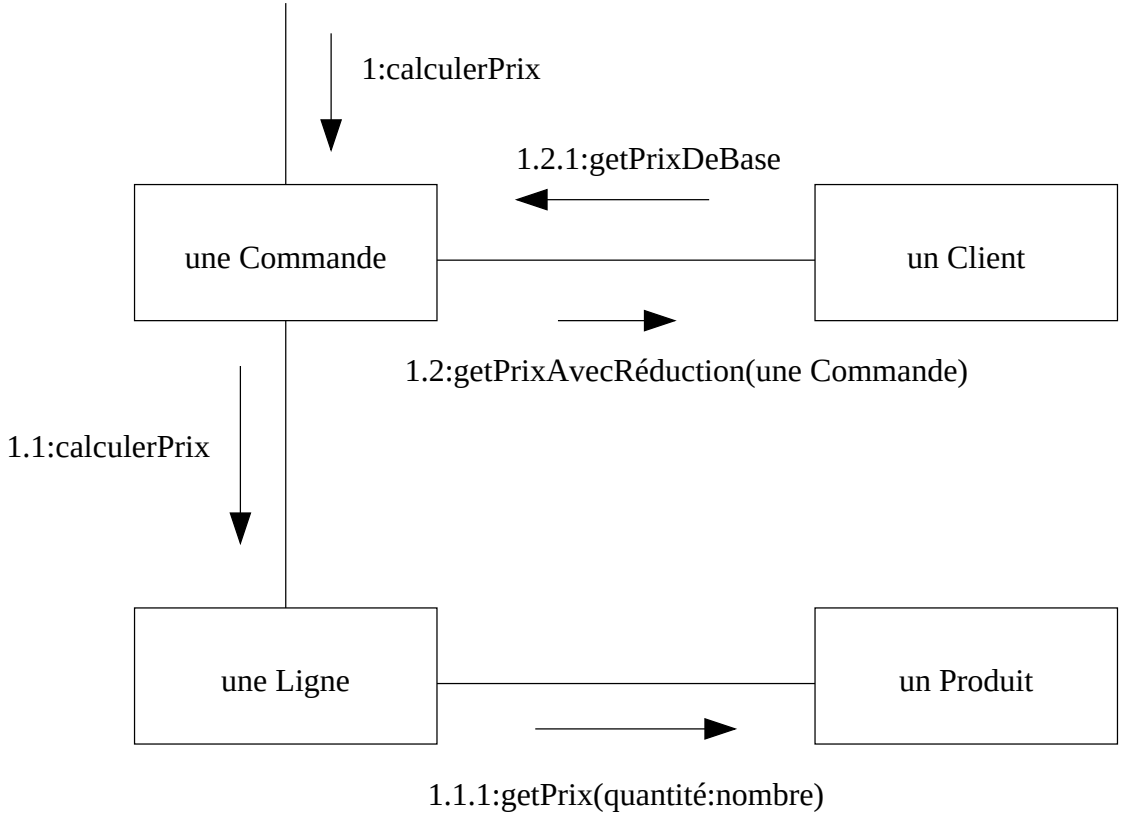
# III.B.2. Communication diagrams : un exemple



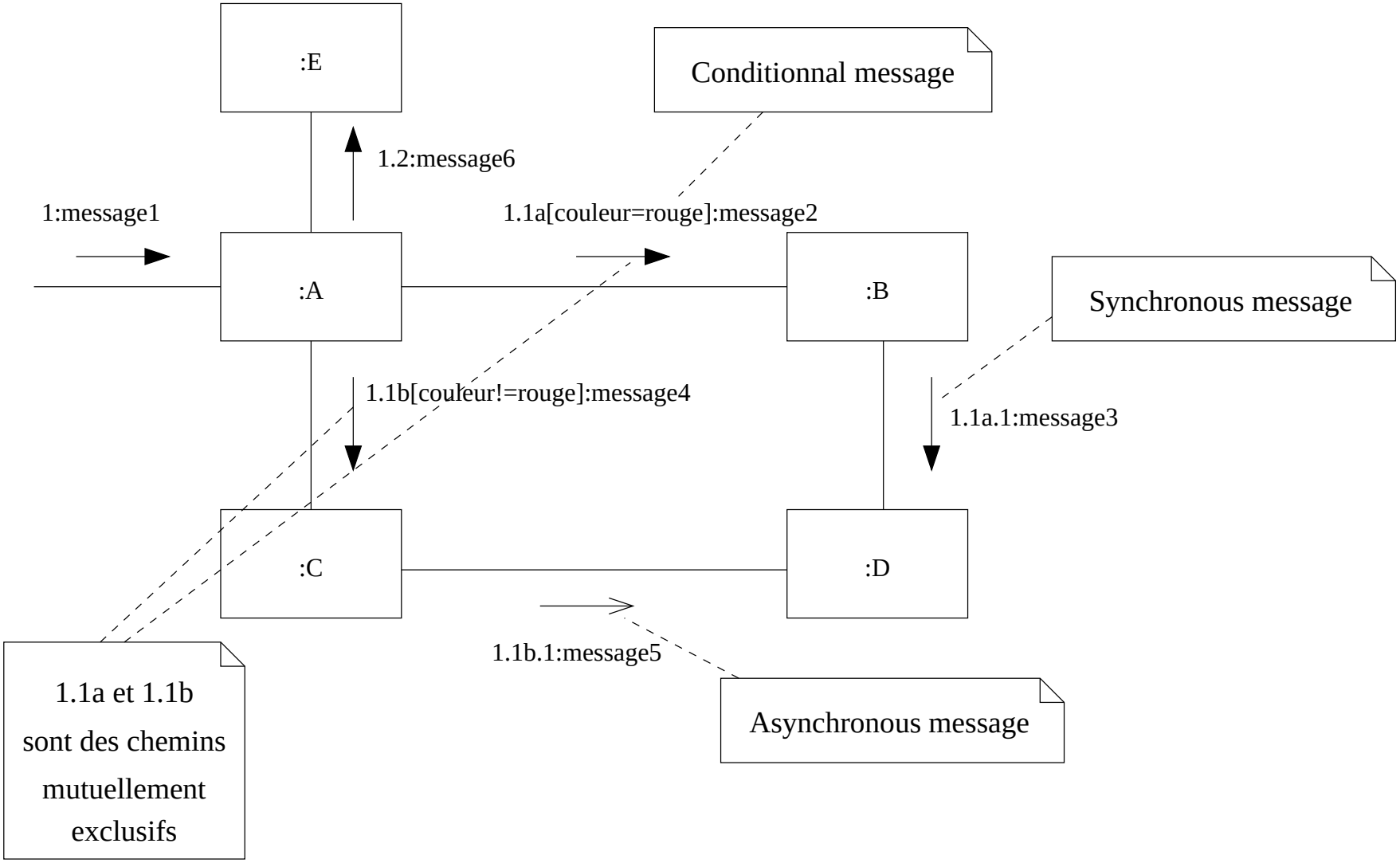
# III.B.2. Communication diagrams : un exemple



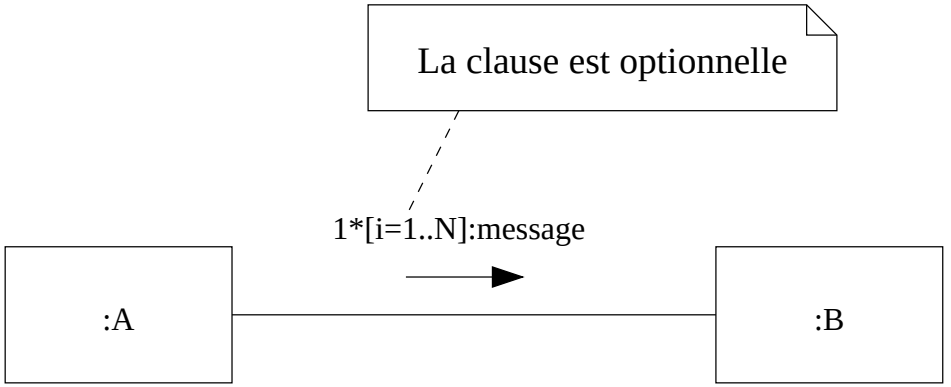
# III.B.2. Communication diagrams : un exemple



# III.B.2. Communication diagrams : messages conditionnels



# III.B.2. Communication diagrams : itérations ou boucles



## III.B.2. Communication diagrams : en guise de conclusion

- Les Communication diagrams donnent une bonne vue d'ensemble des interactions entre participants.

- Il permettent de bien voir l'architecture de communication (centralisée ou répartie) et la délégation des responsabilités.

- Les annotations de séquencement sont nettement moins pertinentes : les utiliser pour montrer un exemple simple plutôt que pour être complet, puis tracer un Sequence diagram (ou un Activity diagram) pour le détail.

# C. Patrons de conception élémentaires

# 1. Component diagram

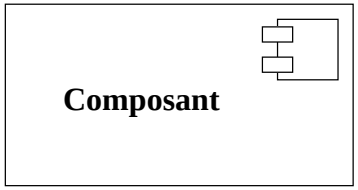
# III.C.1. Component diagram : notion de composant

- La notion de composant est différente de la notion de classe, mais correspond au même type d'abstraction.

- Il s'agit plutôt de découper l'application en entités cohérentes du point de vue des fonctionnalités majeures.

- On peut les rapprocher d'un découpage en packages.

# III.C.1. Component diagram : exemple



## 2. Les patrons de base de conception

Principes :

- On conçoit les classes comme des acteurs qui interagissent.
- On affecte à ces acteurs des responsabilités.
- Les responsabilités sont de deux types : savoir, et savoir-faire.

## III.C.2. Buts des patrons de base

→ Mettre en place les bases d'une architecture logicielle.

→ Orienter la conception des éléments de façon "objet".

→ Se donner des moyens d'évaluer la pertinence des choix de conception.

# III.C.2. Les patrons de base

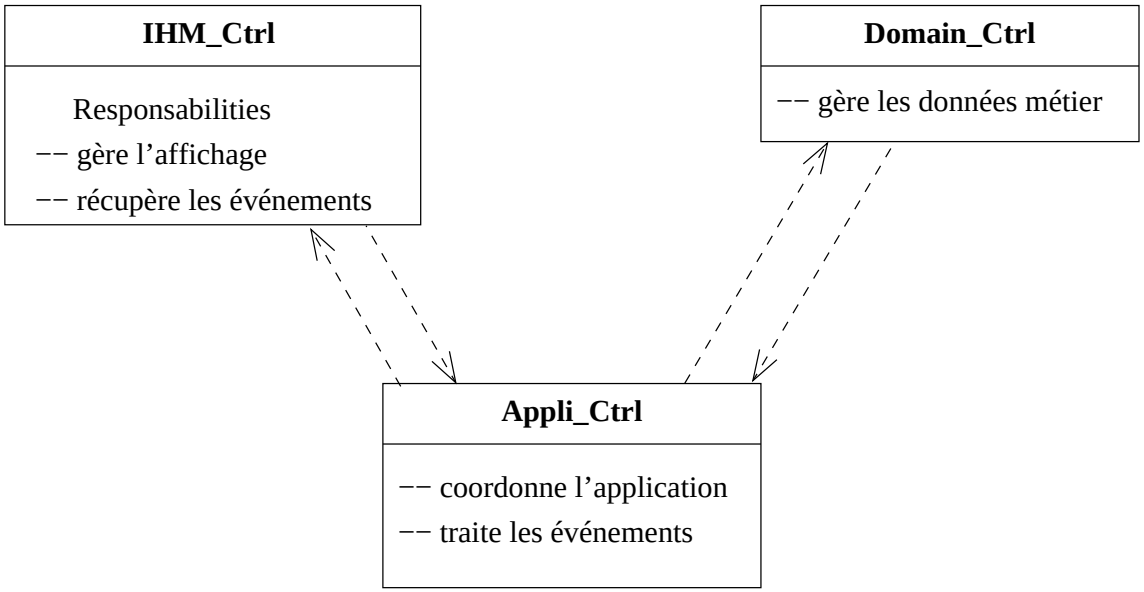
- Créateur.
  Définir la stratégie de création des objets.
- Expert en information.
  Mettre en place les modalités d'attribution de responsabilités.
- Faible couplage.
  Donner un moyen d'évaluer la pertinence des choix de conception.
- Contrôleur.
  Définir un modèle d'architecture interne.
- Forte cohésion.
  Réfléchir aux choix de répartition des responsabilités.

# D. Visualisation des concepts, bis

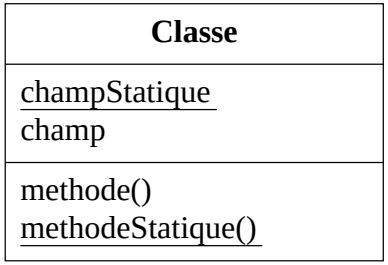
## 1. Class diagrams :

## éléments avancés

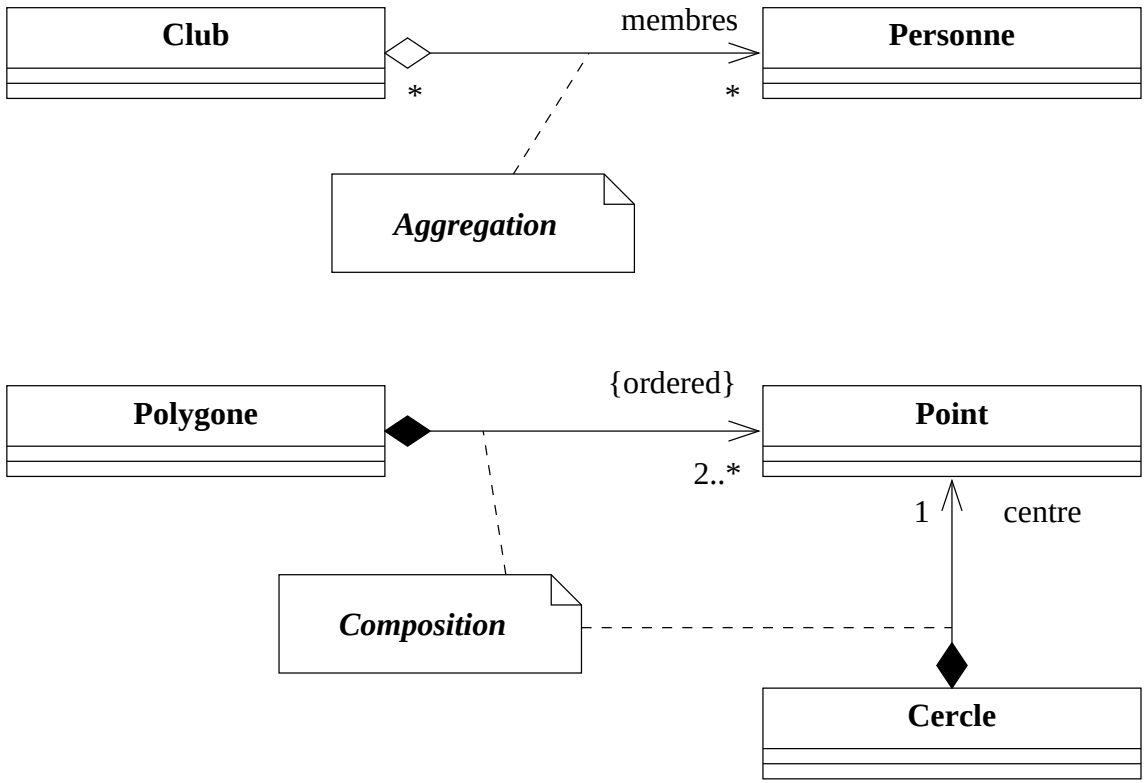
# III.D.1. Class diagrams : responsabilités



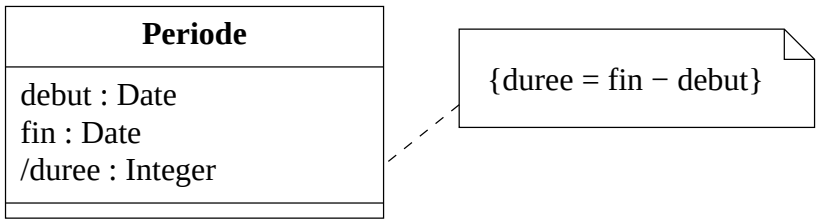
# III.D.1. Class diagrams : éléments statiques



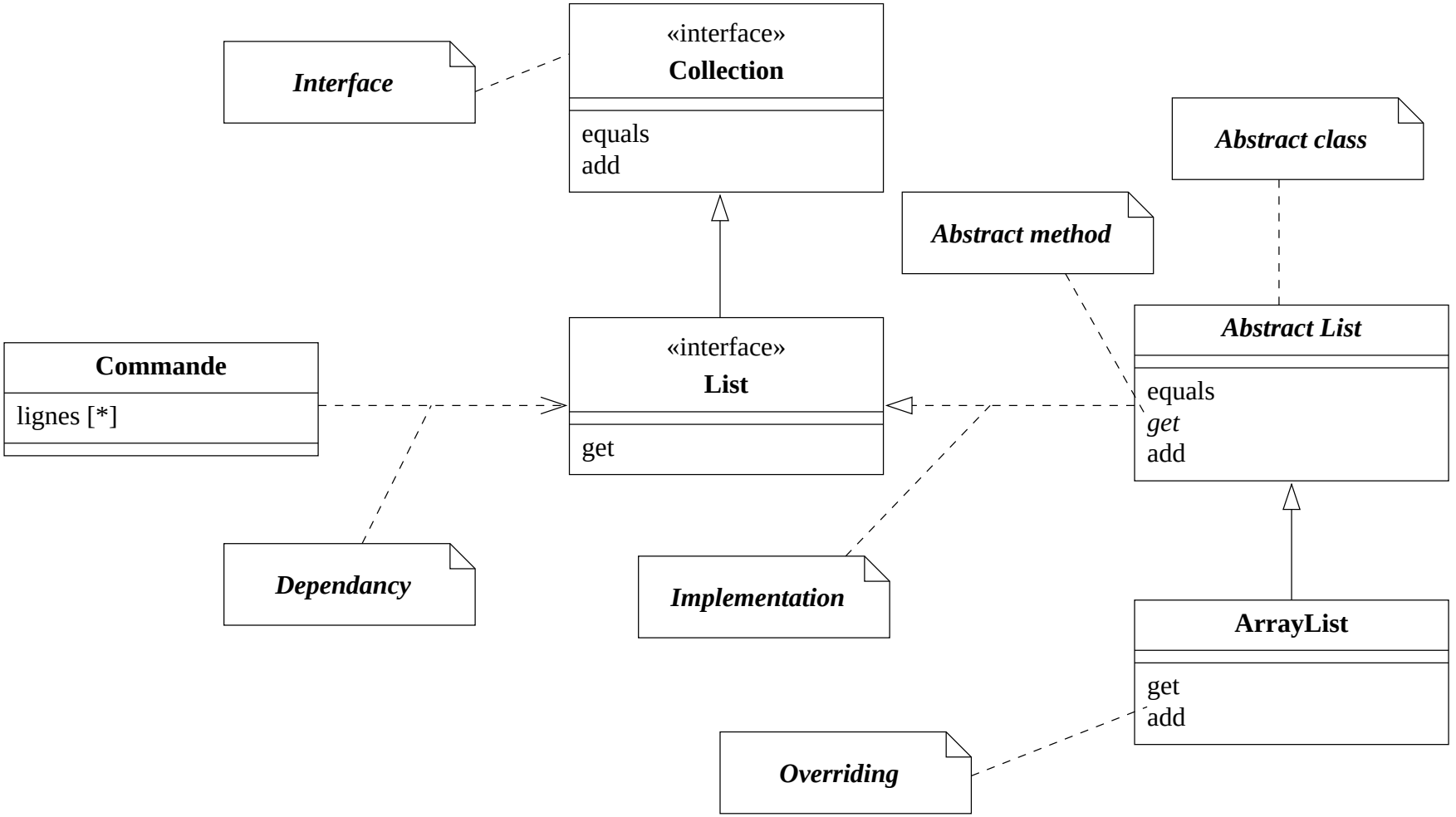
# III.D.1. Class diagrams : aggrégation et composition



# III.D.1. Class diagrams : attributs dérivés



# III.D.1. Class diagrams : interfaces et classes abstraites



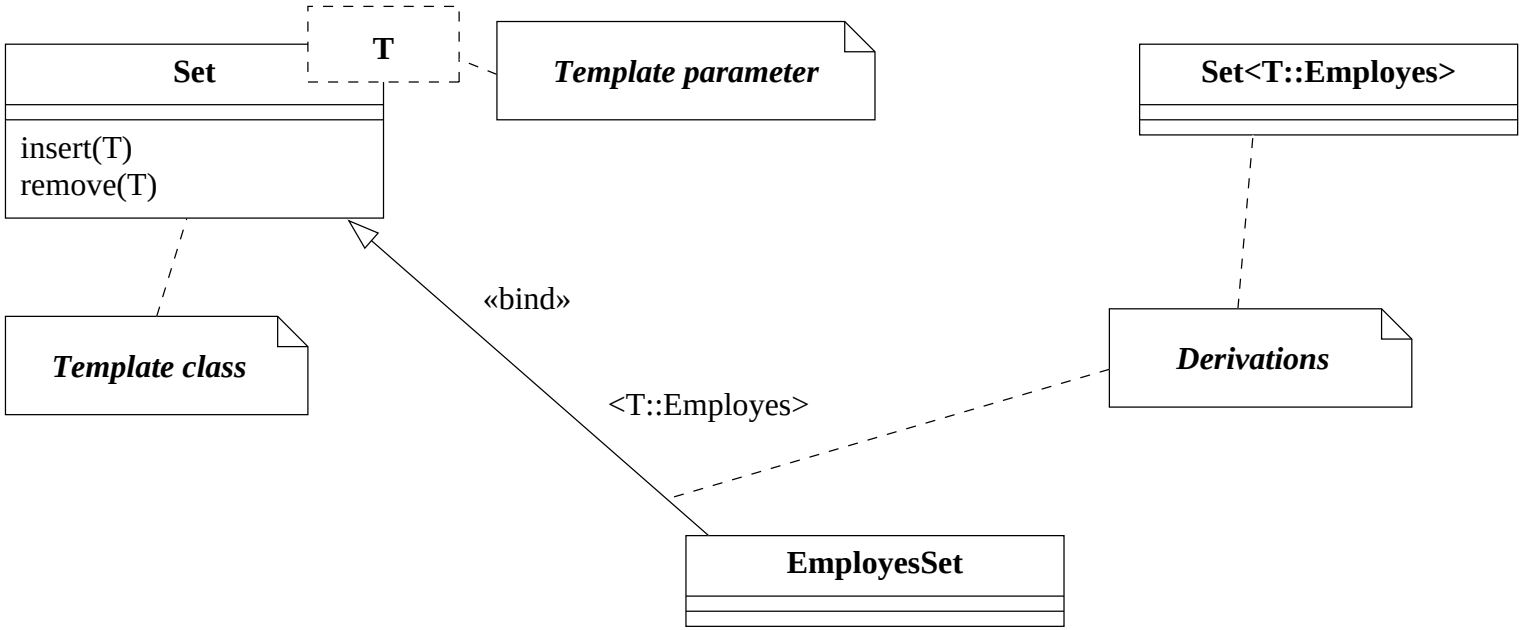
# III.D.1. Classification et généralisation

Combiner !

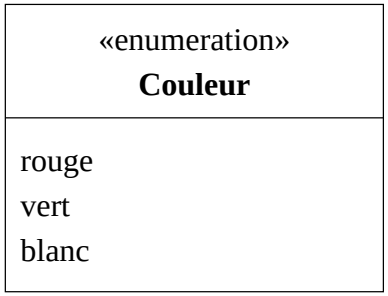
1. Lassie est un Colley.
2. Un Colley est un chien.
3. Les chiens sont des animaux.
4. Colley est une race.
5. Chien est une espèce.

- Généralisation = “Toute instance de A est instance de B” .
- Seule la généralisation est transitive.
- On peut enchaîner classification puis généralisation.

# III.D.1. Class diagrams : classes paramétrées



# III.D.1. Class diagrams : énumérations



## 2. Package diagrams :

## principes, définitions et exemples

# III.D.2. Package diagrams

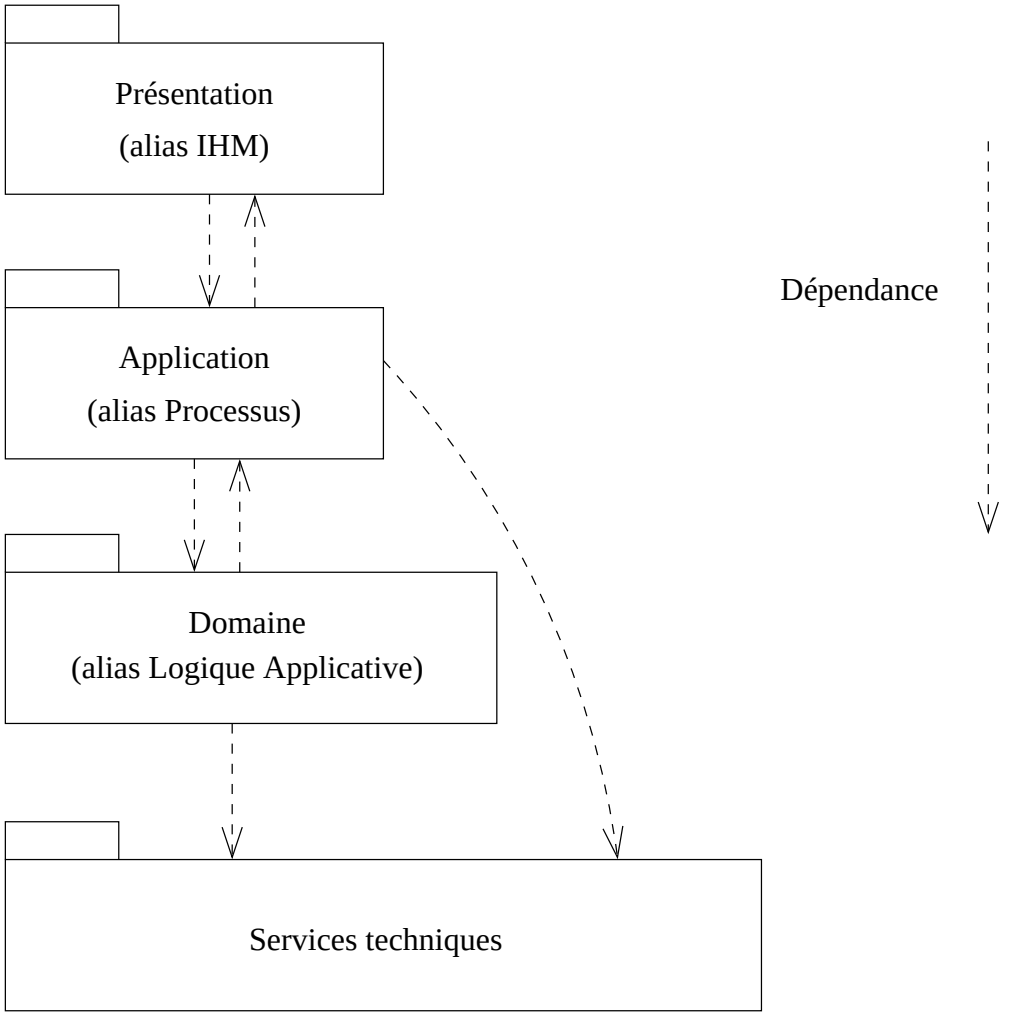
Architecture logique en couches :

- Séparation des fonctions : réduction du couplage et des dépendances, amélioration de la cohésion, de la clarté et du potentiel de réutilisation.

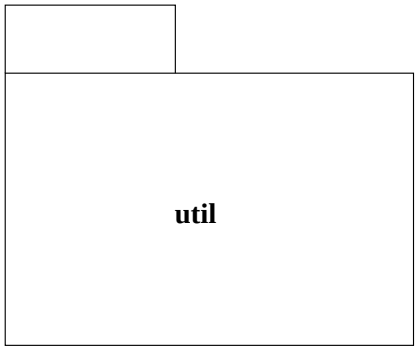
- La complexité est encapsulée et décomposable. La segmentation facilite le développement en équipes.

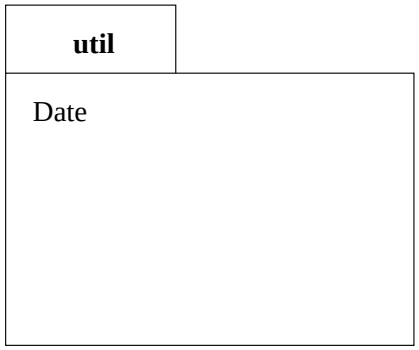
- On peut remplacer certaines couches par de nouvelles implantations. Certaines couches peuvent être distribuées.

# III.D.2. Architecture logique : exemple d'architecture en couches

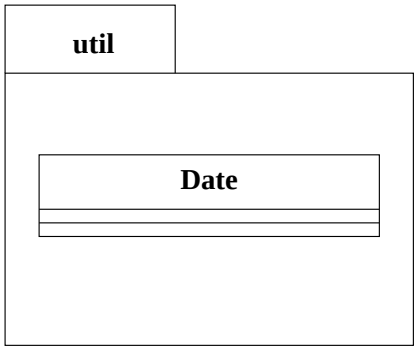


# III.D.2. Package diagrams : exemples

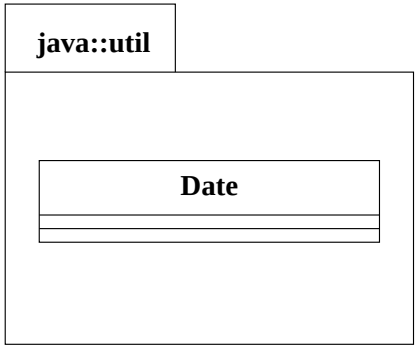


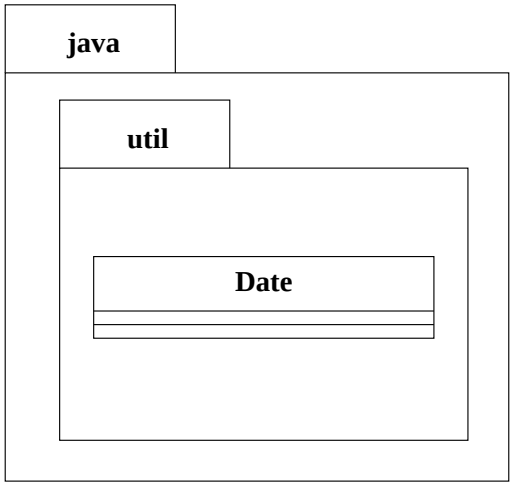

Contenu listé dans la boîte



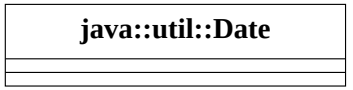

Contenu dessiné dans la boîte




Nom de package complètement qualifié

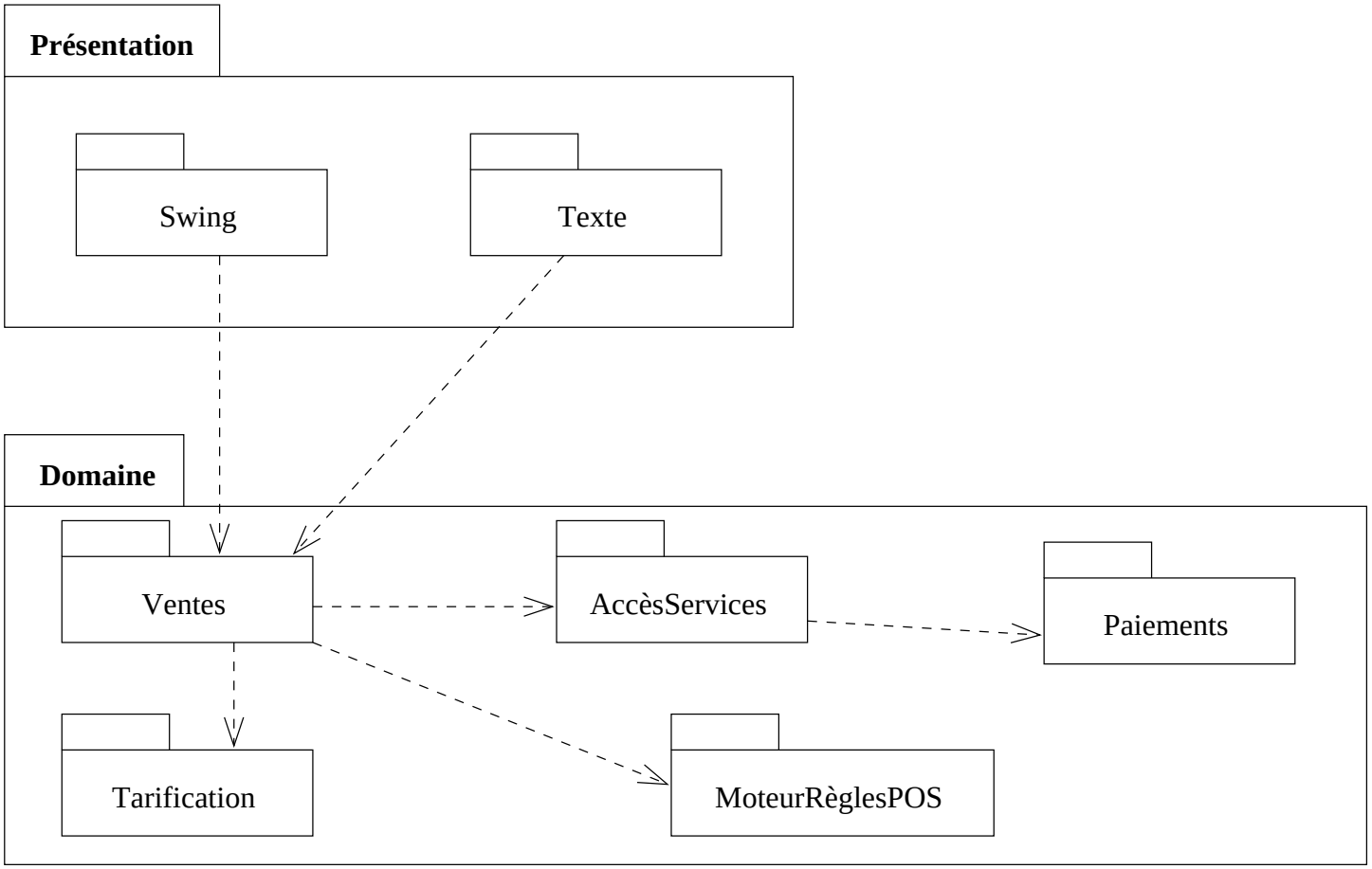



Packages imbriqués

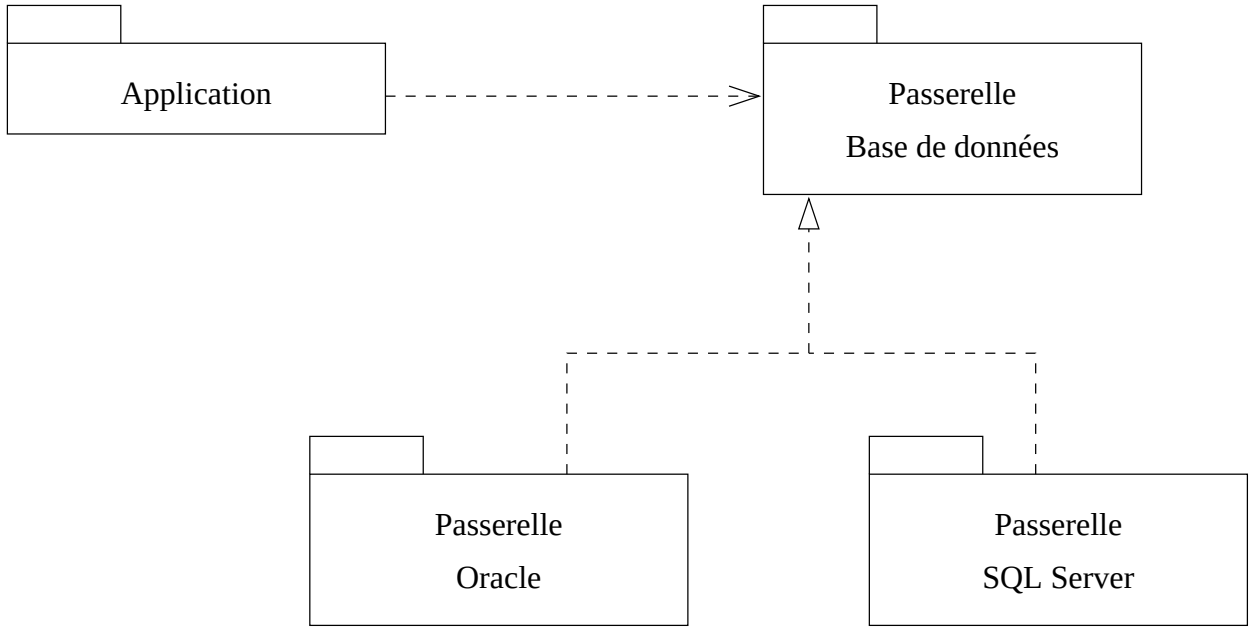



Nom de classe complètement qualifié

# III.D.2. Package diagrams : dépendances



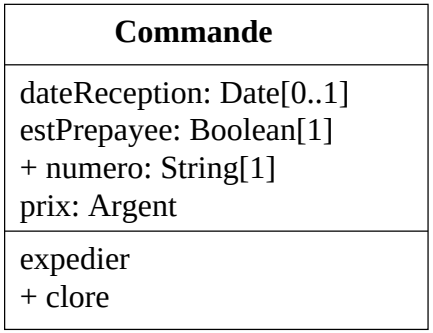
# III.D.2. Package diagrams : implantation



# E. Diagrammes UML et code Java

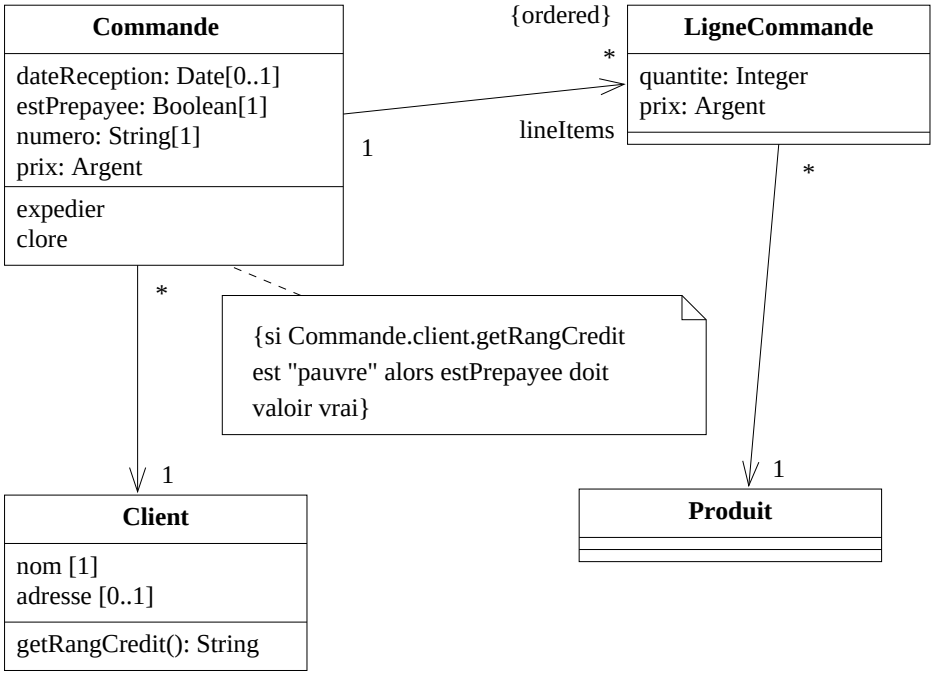
# 1. Class diagrams

# III.E.1. Class diagrams : génération de code



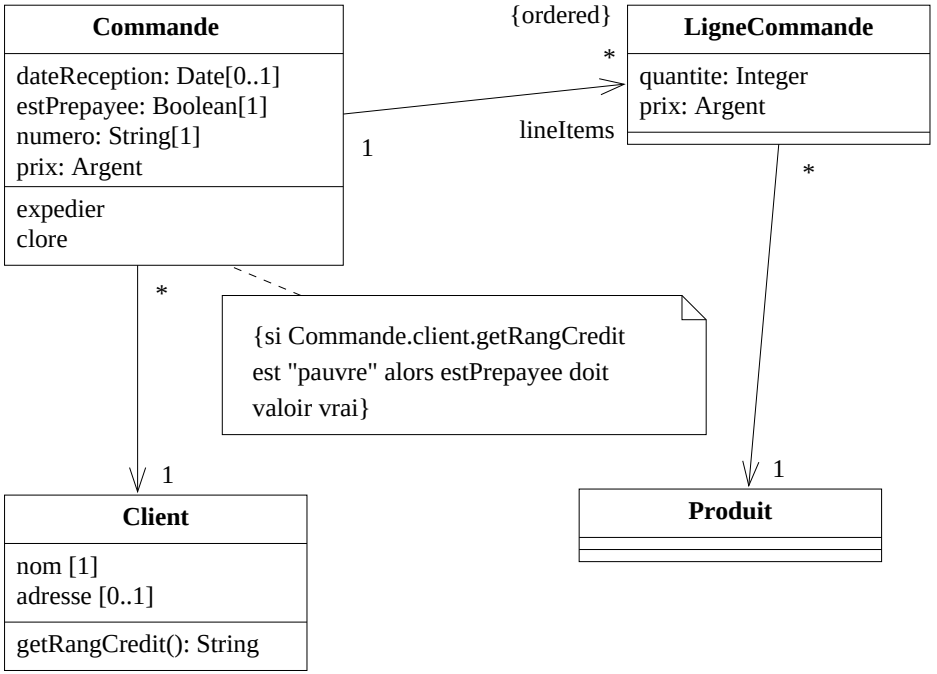

```
class Commande {
    private Date dateReception;
    private boolean estPrepayee;
    public String numero;
    private Argent prix;
    void expedier() {}
    public void clore() {}
}
```

# III.E.1. Class diagrams : génération de code



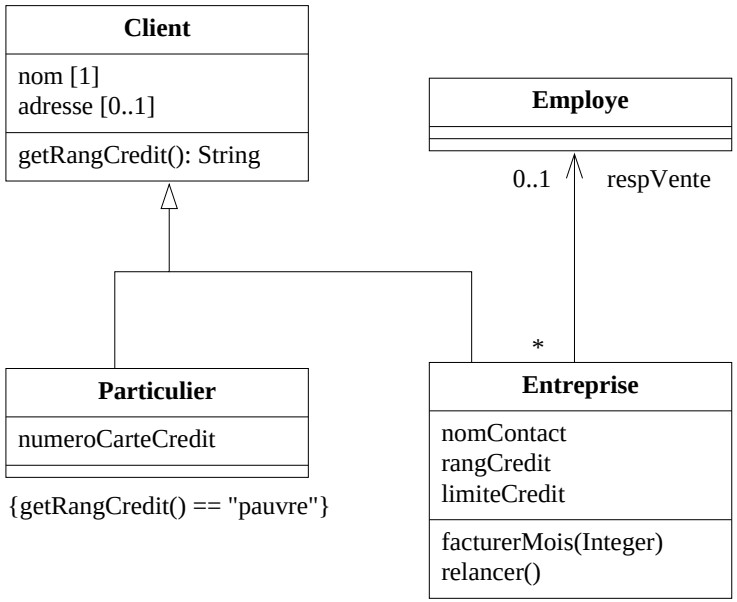

```
class Commande {
    // Si Commande.client.getRangCredit est
    // "pauvre" alors estPrepayee doit valoir vrai.
    private Date dateReception;
    private boolean estPrepayee;
    private String numero;
    private Argent prix;
    private List lineItem; // de LigneCommande
    void expedier() {}
    void clore() {}
}
class LigneCommande {
    private int quantite;
    private Produit produit;
    private Argent prix;
}
class Produit {}
class Argent {}
class Client {
    private nom;
    private adresse;
    String getRangCredit();
}
```

# III.E.1. Class diagrams : génération de code




```
class Commande {
    // Si Commande.client.getRangCredit est
    // "pauvre" alors estPrepayee doit valoir vrai.
    private Date dateReception;
    private boolean estPrepayee;
    private String numero;
    private Argent prix;
    private List lineItem; // de LigneCommande
    void expedier() {}
    void clore() {}
}
class LigneCommande {
    private int quantite;
    private Produit produit;
    Argent prix() {};
}
class Produit {}
class Argent {}
class Client {
    private nom;
    private adresse;
    String getRangCredit();
}
```

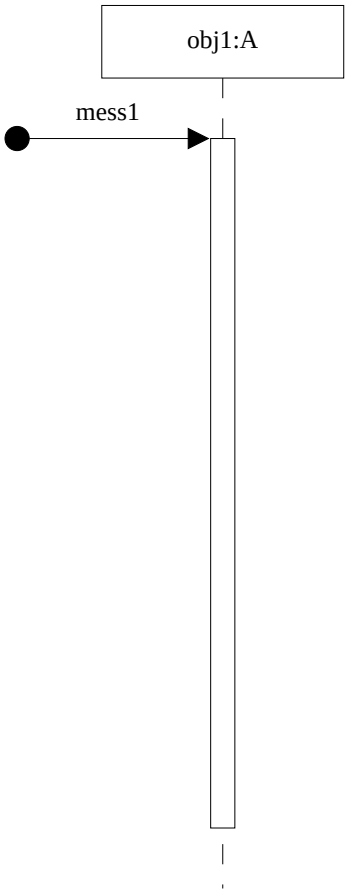
# III.E.1. Class diagrams : génération de code




```
class Client {
    private nom;
    private adresse;
    String getRangCredit();
}
class Particulier extends Client {
    // getRangCredit() == "pauvre"
    private numeroCarteCredit;
}
class Entreprise extends Client {
    private nomContact;
    private rangCredit;
    private limiteCredit;
    private Employe respVente;
    void facturerMois(int x) {}
    void relancer() {}
}
class Employe {}
```

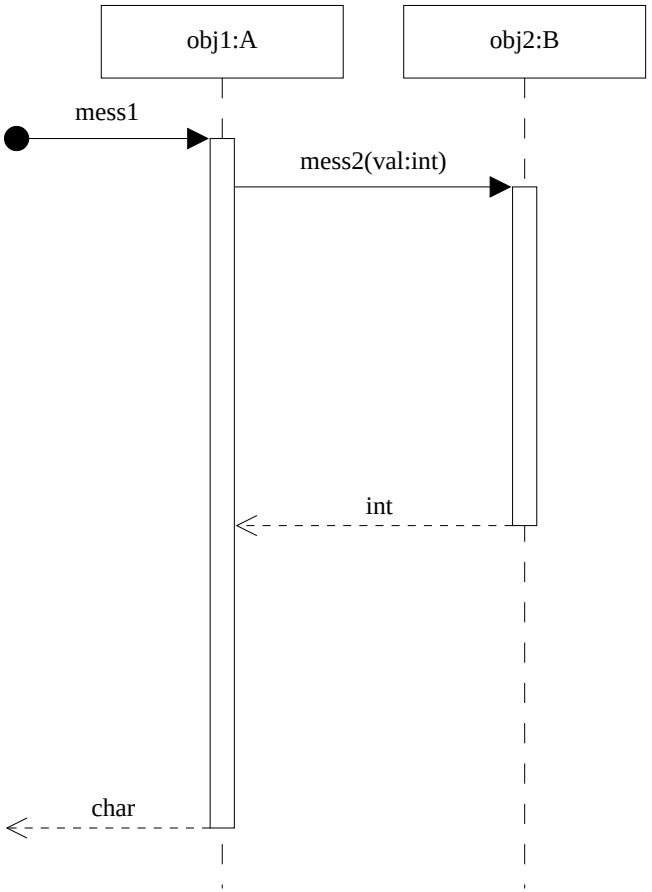
## 2. Sequence diagrams

# III.E.2. Sequence diagrams : génération de code



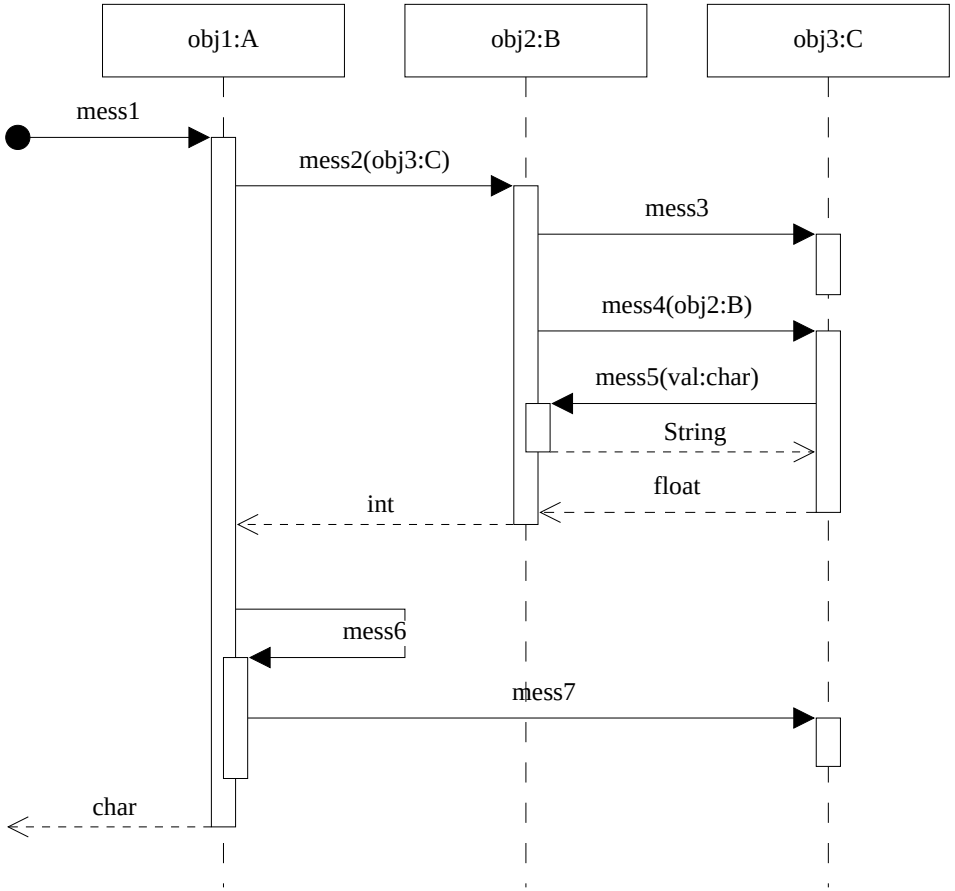

```
class A {
    void mess1() {}
}
```

# III.E.2. Sequence diagrams : génération de code



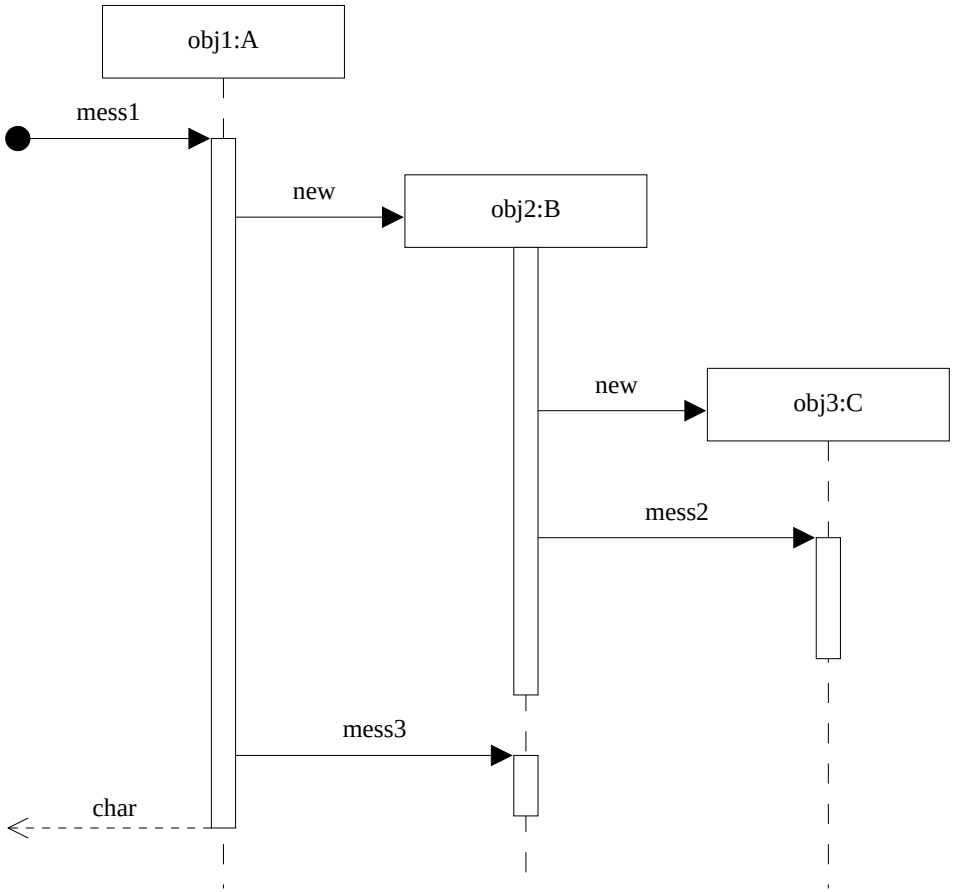

```
class A {
    private B obj2;
    private int val;

    char mess1() {
        int x;
        x = obj2.mess2(val);
    }
}

class B {
    int mess2(int y) {}
}
```

# III.E.2. Sequence diagrams : génération de code



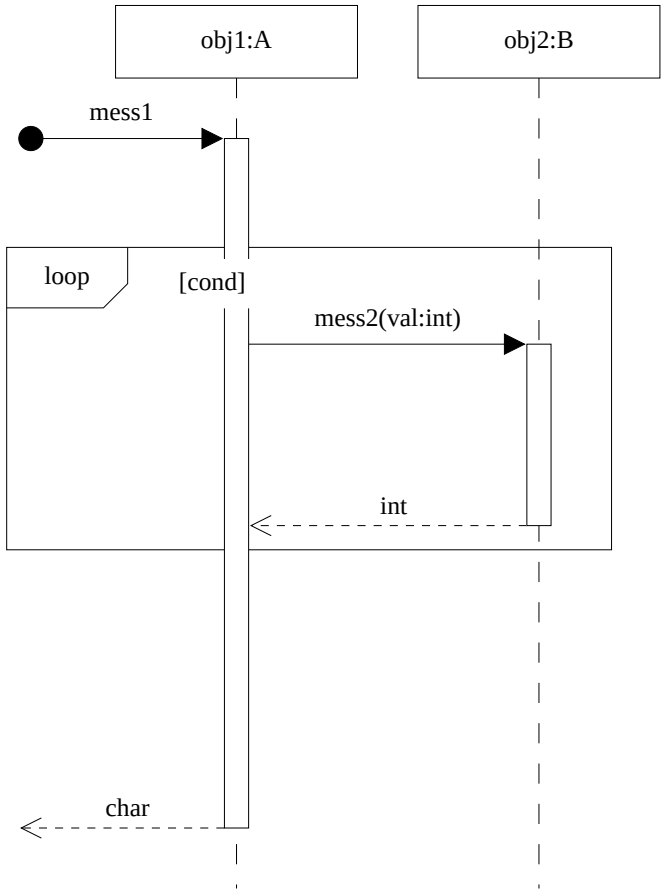

```
class A {
    private B obj2;
    private C obj3;
    char mess1() {
        int x ;
        x = obj2.mess2(obj3);
        mess6(); // this.mess6()
    }
    void mess6() {
        obj3.mess7();
    }
}
class B {
    int mess2(C obj) {
        obj.mess3();
        float y = obj.mess4(this);
    }
    String mess5(char z) {}
}
class C {
    private char val;
    void mess3() {}
    float mess4(B obj) {
        obj.mess5(val);
    }
    void mess7() {}
}
```

# III.E.2. Sequence diagrams : génération de code



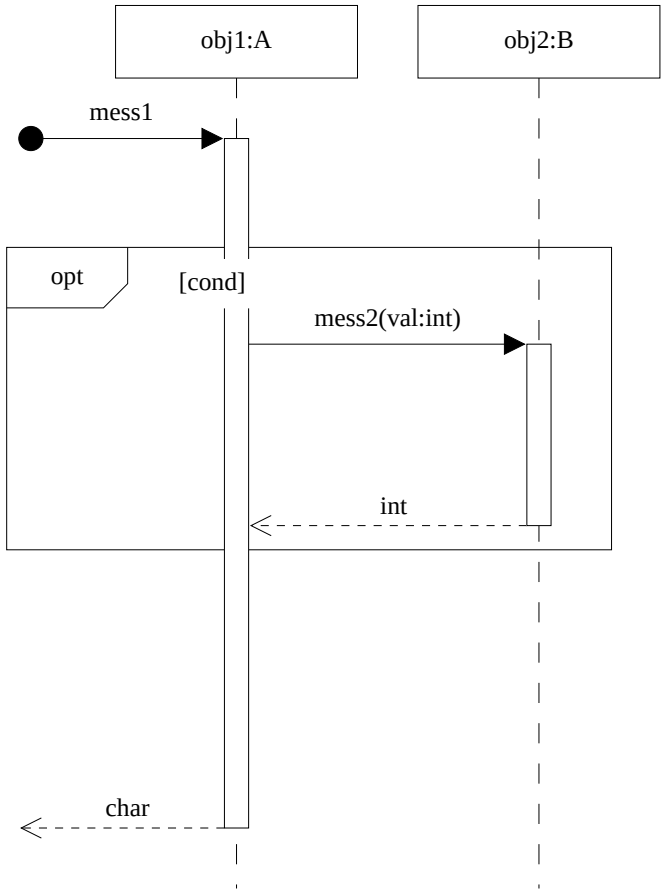

```
class A {
    char mess1() {
        B obj2 = new B();
        obj2.mess3();
    }
}

class B {
    B() {
        C obj3 = new C();
        obj3.mess2();
    }
    void mess3() {}
}

class C {
    void mess2() {}
}
```

# III.E.2. Sequence diagrams : génération de code



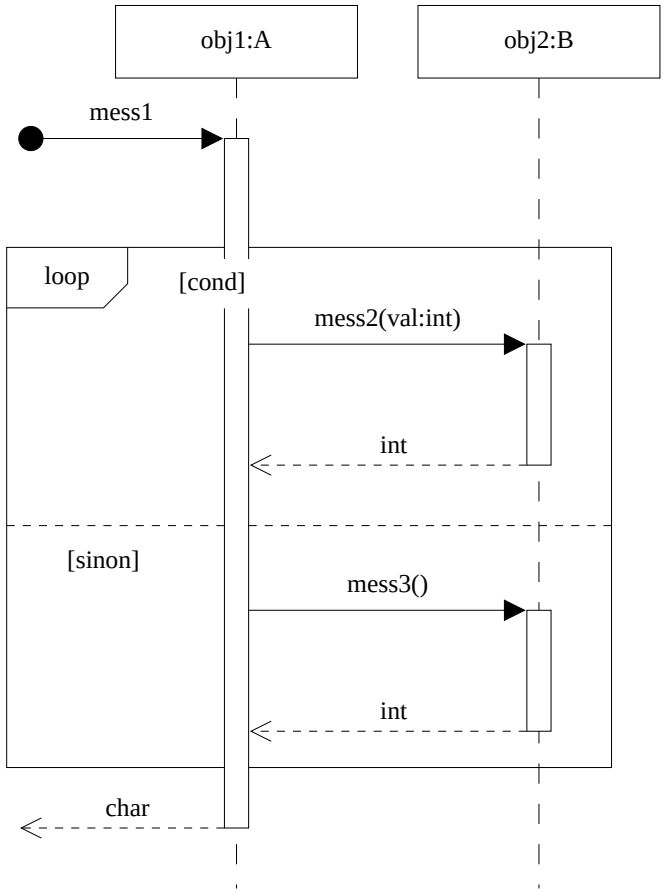

```
class A {
    private B obj2;

    char mess1() {
        while (cond) {
            int x = obj2.mess2(val);
        }
    }
}

class B {
    int mess2(int y) {}
}
```

# III.E.2. Sequence diagrams : génération de code




```
class A {
    private B obj2;

    char mess1() {
        if (cond) {
            int x = obj2.mess2(val);
        }
    }
}

class B {
    int mess2(int y) {}
}
```

# III.E.2. Sequence diagrams : génération de code




```
class A {
    private B obj2;

    char mess1() {
        int x;
        if (cond) {
            x = obj2.mess2(val);
        } else {
            x = obj2.mess3();
        }
    }
}

class B {
    int mess2(int y) {}
    int mess3() {}
}
```