
Diagramme de séquences de conception

Objectifs

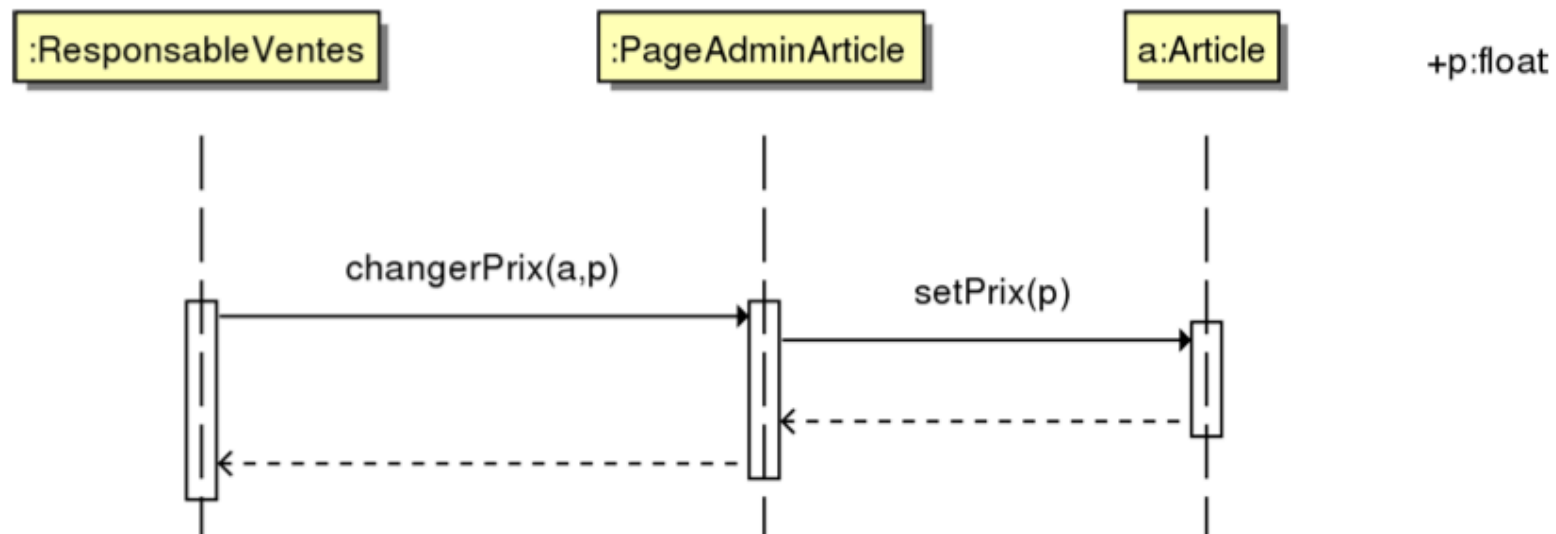
- Les **diagrammes de cas d'utilisation** modélisent à **QUOI** sert le système, en organisant les interactions possibles avec les acteurs.
- Les **diagrammes de classes** permettent de spécifier la structure et les liens entre les objets dont le système est composé : ils spécifie **QUI** sera à l'œuvre dans le système pour réaliser les fonctionnalités décrites par les diagrammes de cas d'utilisation.
- Les **diagrammes de séquences** permettent de décrire **COMMENT** les éléments du système interagissent entre eux et avec les acteurs.
 - ↳ Les objets au cœur d'un système interagissent en s'échangeant des messages.
 - ↳ Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

Exemple (1/2)

- Cas d'utilisation :

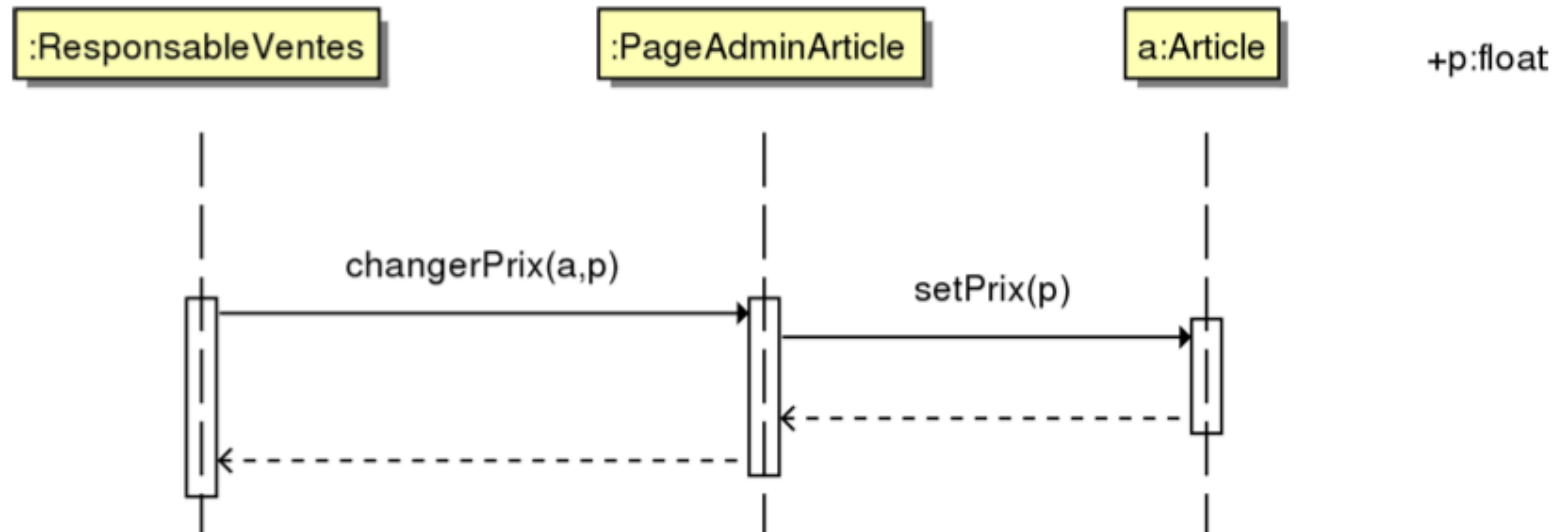


- Diagramme de Séquence correspondant

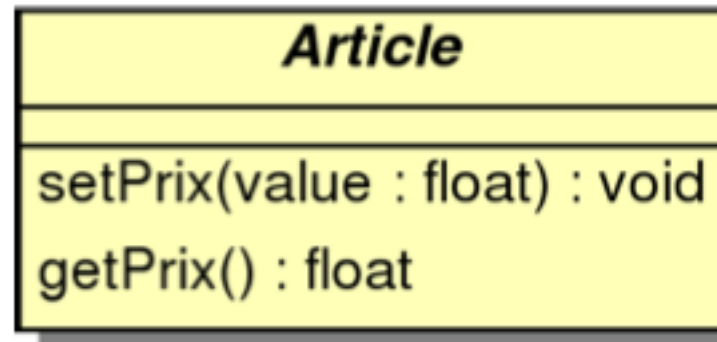


Exemple (2/2)

- Diagramme de séquences correspondant



- Opérations nécessaire dans le diagramme de classe

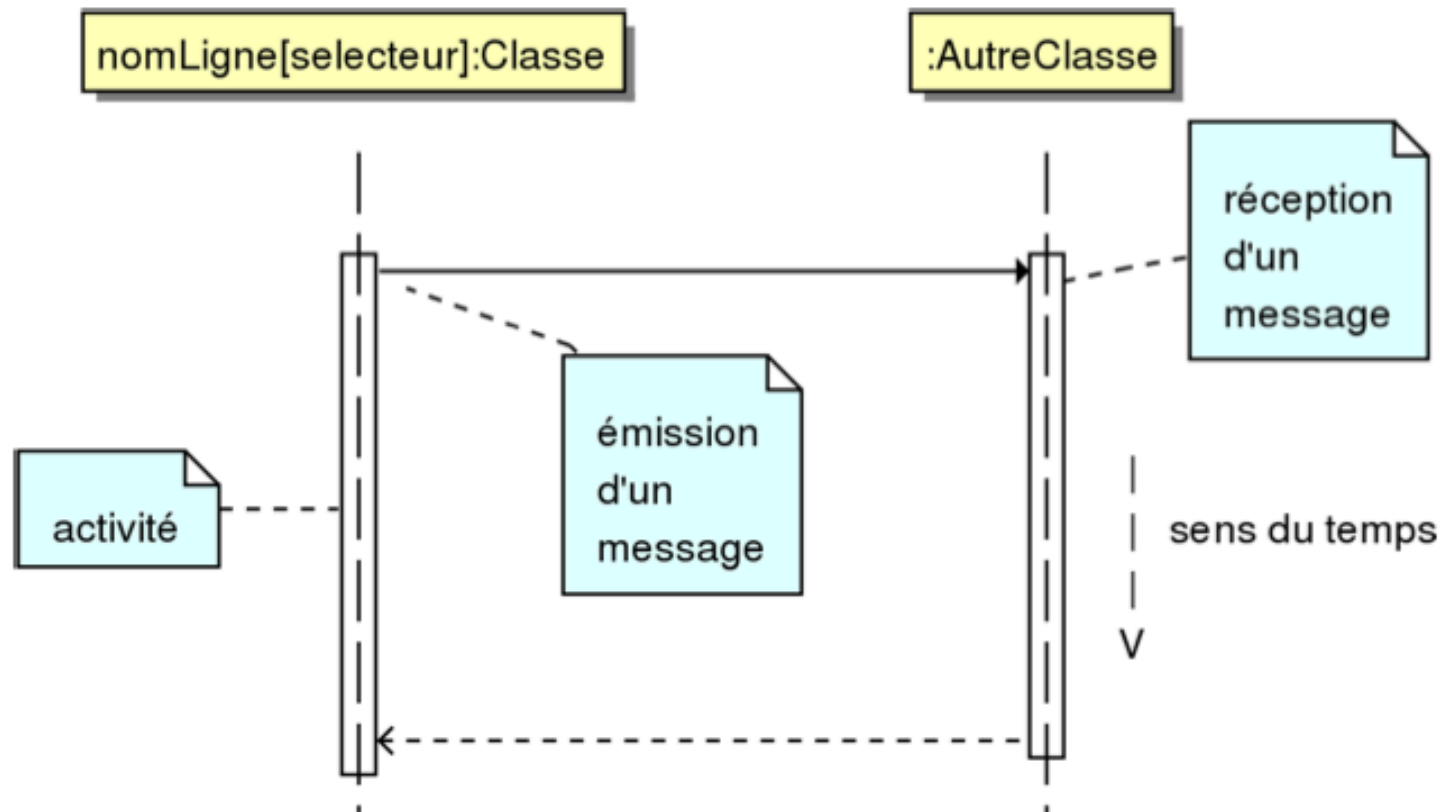


Ligne de vie

- Une ligne de vie représente un participant à une interaction (objet ou acteur).

↪ `nomLigneDeVie[selecteur]:nomClasseOuActeur`

- Dans le cas d'une collection de participants, un sélecteur permet de choisir un objet parmi n (par exemple objets [2]).



Messages

- **Les principales informations contenues dans un diagramme de séquences sont**

- ↳ les messages échangés entre les lignes de vie, présentés dans un ordre chronologique.
- ↳ Un message définit une communication particulière entre des lignes de vie (objets ou acteurs).
- ↳ Plusieurs types de messages existent, dont les plus courants :
 - ↳ l'envoi d'un signal ;
 - ↳ l'invocation d'une opération (appel de méthode) ;
 - ↳ la création ou la destruction d'un objet.

- **La réception des messages provoque une période d'activité (rectangle vertical sur la ligne de vie) marquant le traitement du message (spécification d'exécution dans le cas d'un appel de méthode).**

Principaux type de messages

- Un message **synchrone** bloque l'expéditeur jusqu'à la réponse du destinataire. Le flot de contrôle passe de l'émetteur au récepteur.

↳ Typiquement : appel de méthode Si un objet A invoque une méthode d'un objet B, A reste bloqué tant que B n'a pas terminé.



↳ On peut associer aux messages d'appel de méthode un message de retour (en pointillés) marquant la reprise du contrôle par l'objet émetteur du message synchrone.

- Un message **asynchrone** n'est pas bloquant pour l'expéditeur. Le message envoyé peut être pris en compte par le récepteur à tout moment ou ignoré.

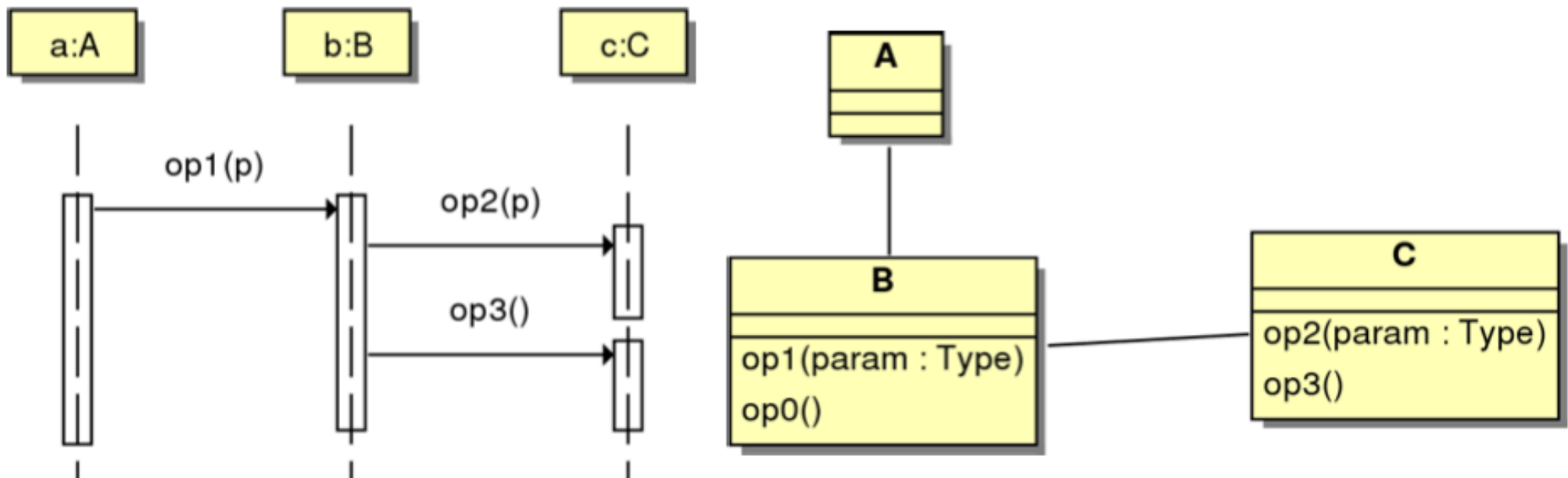
↳ Typiquement : envoi de signal (voir stéréotype de classe « signal »).



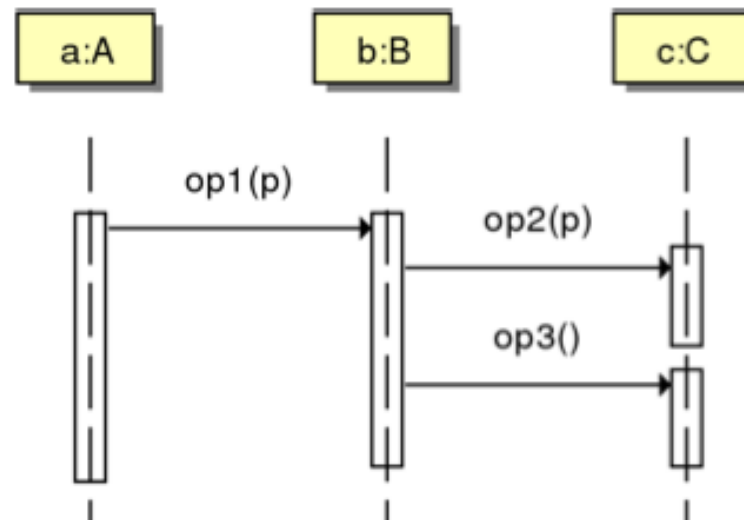
Correspondance messages / opérations

- Les messages synchrones correspondent à des opérations dans le diagramme de classes.

Envoyer un message et attendre la réponse pour poursuivre son activité revient à invoquer une méthode et attendre le retour pour poursuivre ses traitements.



implantation des messages synchrones



- code

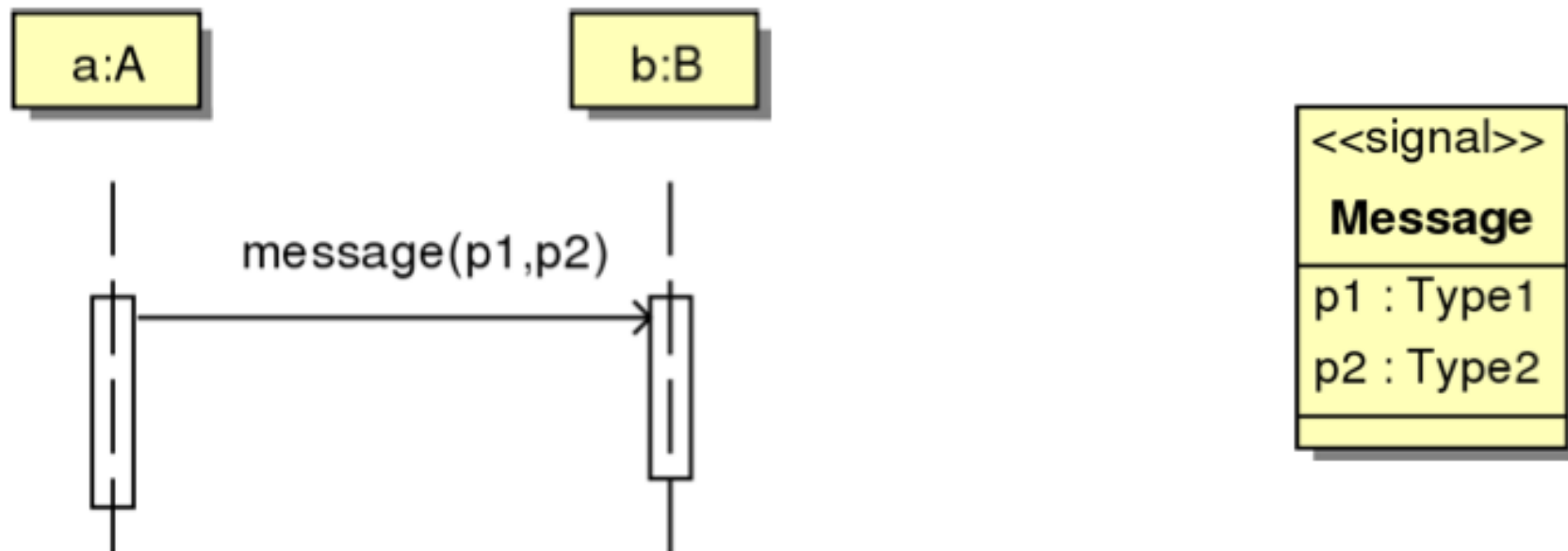
```
class B {  
    C c;  
    op1(p:Type){  
        c.op2(p);  
        c.op3();  
    }  
}
```

```
class C {  
    op2(p:Type){  
        ...  
    }  
    op3(){  
        ...  
    }  
}
```

Correspondance messages / signaux

- Les messages asynchrones correspondent à des signaux dans le diagramme de classes.

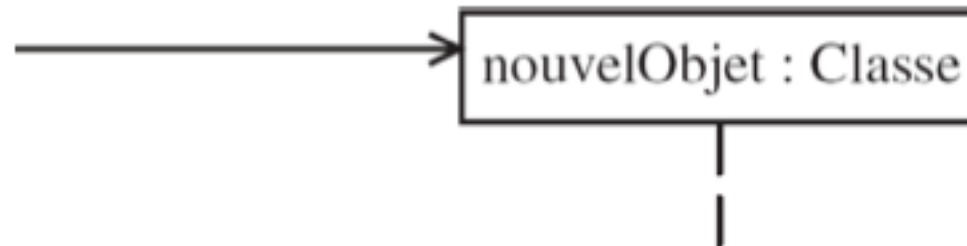
Les signaux sont des objets dont la classe est stéréotypée « signal » et dont les attributs (porteurs d'information) correspondent aux paramètres du message.



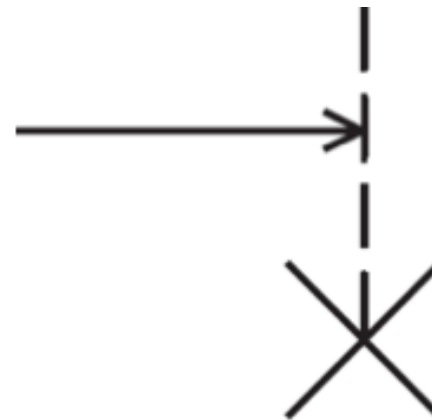
Création et destruction de lignes de vie

- La **création** d'un objet est matérialisée par une flèche qui pointe sur le sommet d'une ligne de vie.

↳ On peut aussi utiliser un message asynchrone ordinaire portant le nom « create ».



- La **destruction** d'un objet est matérialisée par une croix qui marque la fin de la ligne de vie de l'objet.



Messages complets, perdus et trouvés

- Un **message complet** est tel que les événements d'envoi et de réception sont connus.

↳ Un message complet est représenté par une flèche partant d'une ligne de vie et arrivant à une autre ligne de vie

- Un **message perdu** est tel que l'événement d'envoi est connu, mais pas l'événement de réception.



↳ La flèche part d'une ligne de vie mais arrive sur un cercle indépendant marquant la méconnaissance du destinataire.

↳ Exemple : broadcast.

- Un **message trouvé** est tel que l'événement de réception est connu, mais pas l'événement d'émission.



Syntaxe des messages

- **La syntaxe des messages est :**

- ↳ `nomSignalOuOperation(parametres)`

- **La syntaxe des arguments est la suivante :**

- ↳ `nomParametre=valeurParametre`

- **Pour un argument modifiable :**

- ↳ `nomParametre:valeurParametre`

- **Exemples :**

- ↳ `appeler("Capitaine Hadock", 54214110)`

- ↳ `afficher(x,y)`

- ↳ `initialiser(x=100)`

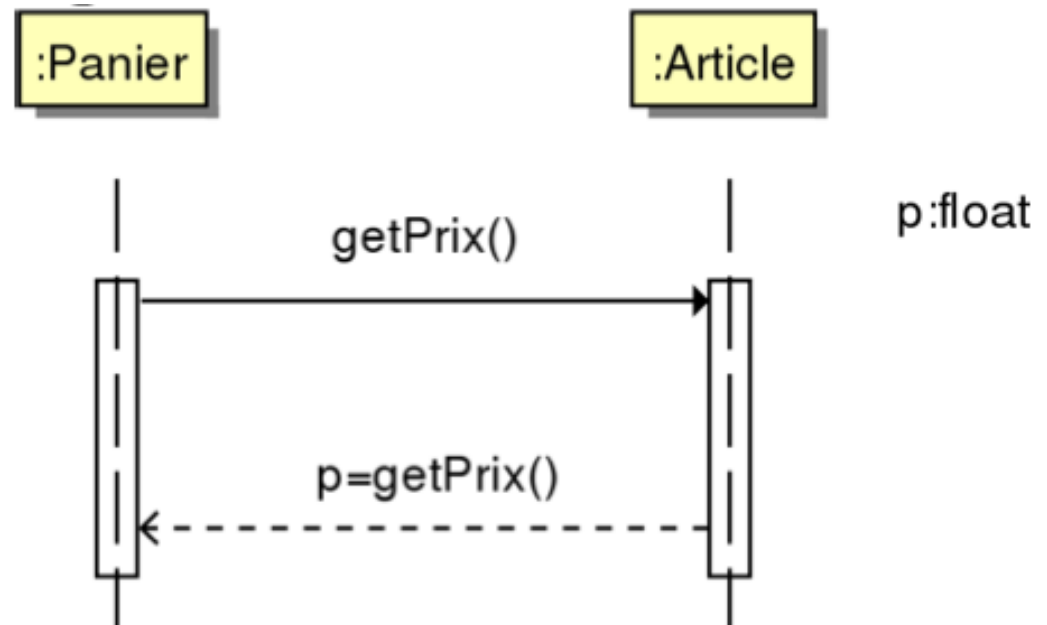
- ↳ `f(x:12)`

Messages de retour

- Le récepteur d'un message synchrone rend la main à l'émetteur du message en lui envoyant un message de retour
- Les messages de retour sont optionnels :

↳ la fin de la période d'activité marque également la fin de l'exécution d'une méthode.

↳ Ils sont utilisés pour spécifier le résultat de la méthode invoquée.



Le retour des messages asynchrones s'effectue par l'envoi de nouveaux messages asynchrones.

Syntaxe des messages de retour

- **La syntaxe des messages de retour est :**

↳ `attributCible = nomOperation(params) : valeurRetour`

- **La syntaxe des paramètres est :**

↳ `nomParam = valeurParam`

Ou

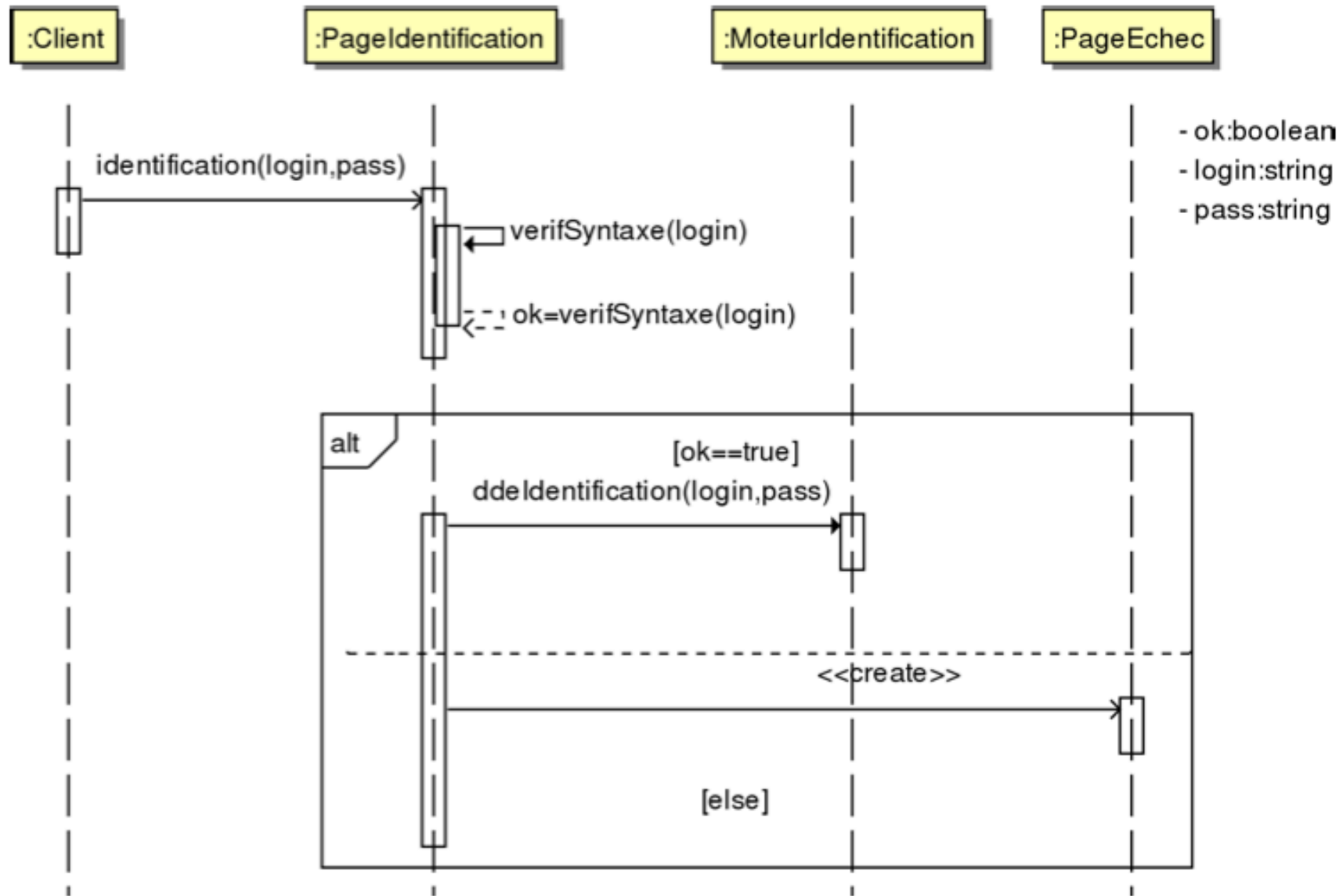
↳ `nomParam : valeurParam`

- **Les messages de retour sont représentés en pointillés.**

Fragment combine

- **Un fragment combiné permet de décomposer une interaction complexe en fragments suffisamment simples pour être compris.**
 - ↳ Recombiner les fragments restitue la complexité.
 - ↳ Syntaxe complète avec UML 2 : représentation complète de processus avec un langage simple (ex : processus parallèles).
- **Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone.**
 - ↳ Dans le pentagone figure le type de la combinaison (appelé « opérateur d'interaction »).

Exemple de fragment avec l'opérateur conditionnel



Type d'opérateurs d'interaction

- **Opérateurs de branchement (choix et boucles) :**

- ↳ alternative, option, break et loop ;

- **Opérateurs contrôlant l'envoi en parallèle de messages :**

- ↳ parallel et critical region ;

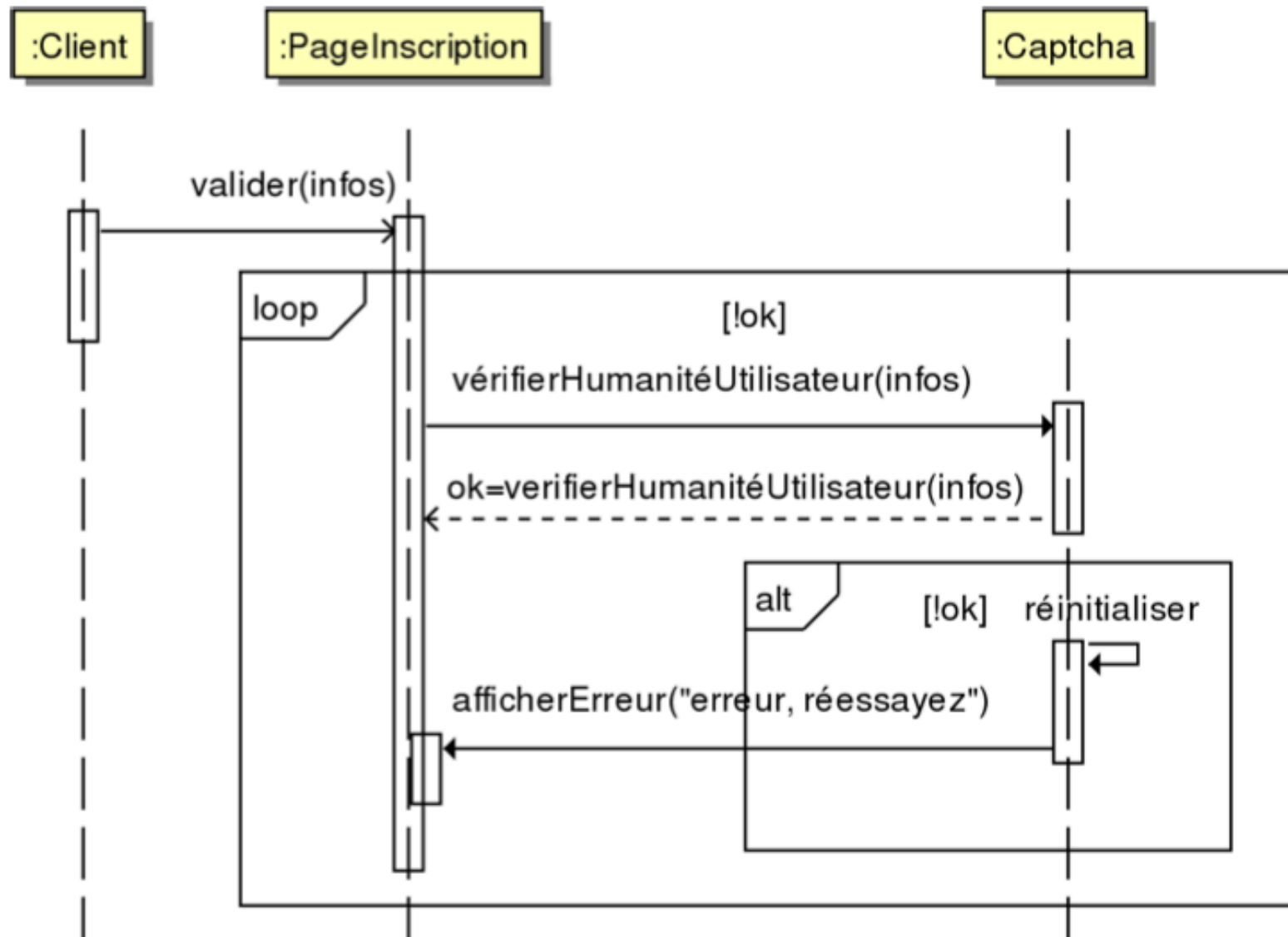
- **Opérateurs contrôlant l'envoi de messages :**

- ↳ ignore, consider, assertion et negative ;

- **Opérateurs fixant l'ordre d'envoi des messages :**

- ↳ Weak, sequencing et strict sequencing.

Opérateur de boucle



Syntaxe de l'opérateur loop

- **Syntaxe d'une boucle :**

- ↳ `loop(minNbIterations ,maxNbIterations)`

- ↳ La boucle est répétée au moins `minNbIterations` fois

- ↳ avant qu'une éventuelle condition booléenne ne soit testée (la condition est placée entre crochets dans le fragment)

- ↳ Tant que la condition est vraie, la boucle continue, au plus `maxNbIterations` fois.

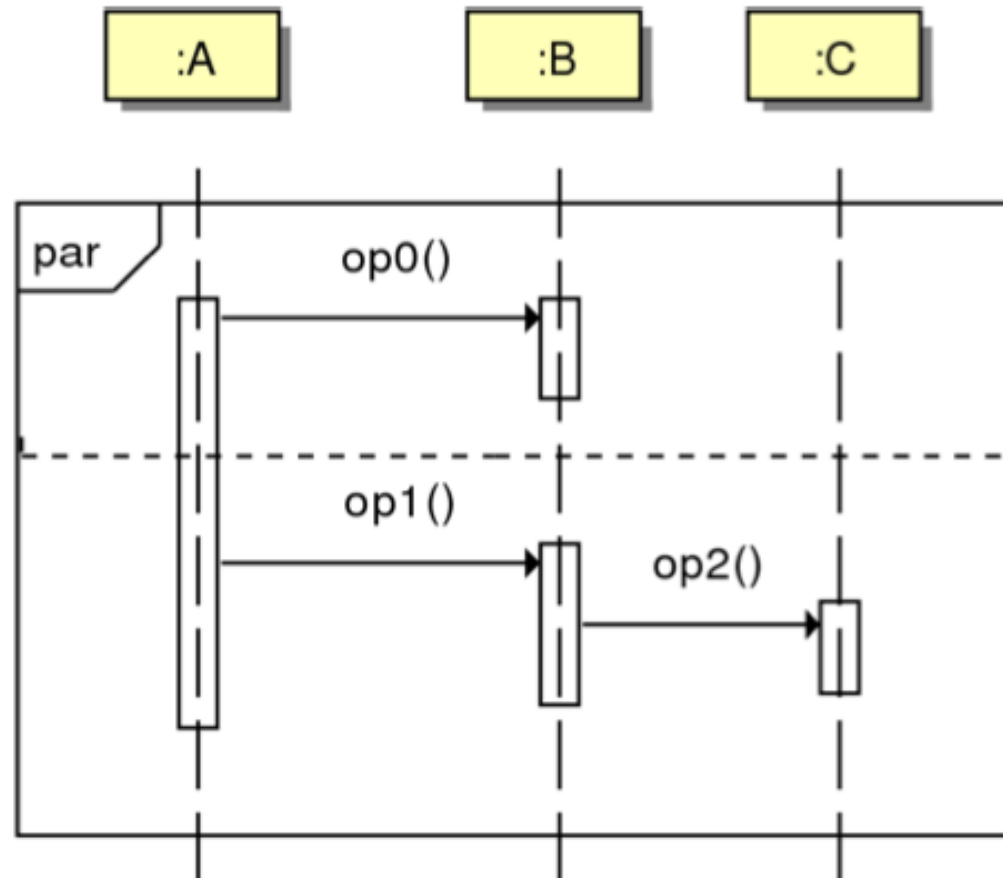
- **Notations :**

- ↳ `loop(valeur)` est équivalent à `loop(valeur,valeur)`.

- ↳ `loop` est équivalent à `loop(0,*)`, où `*` signifie « illimité ».

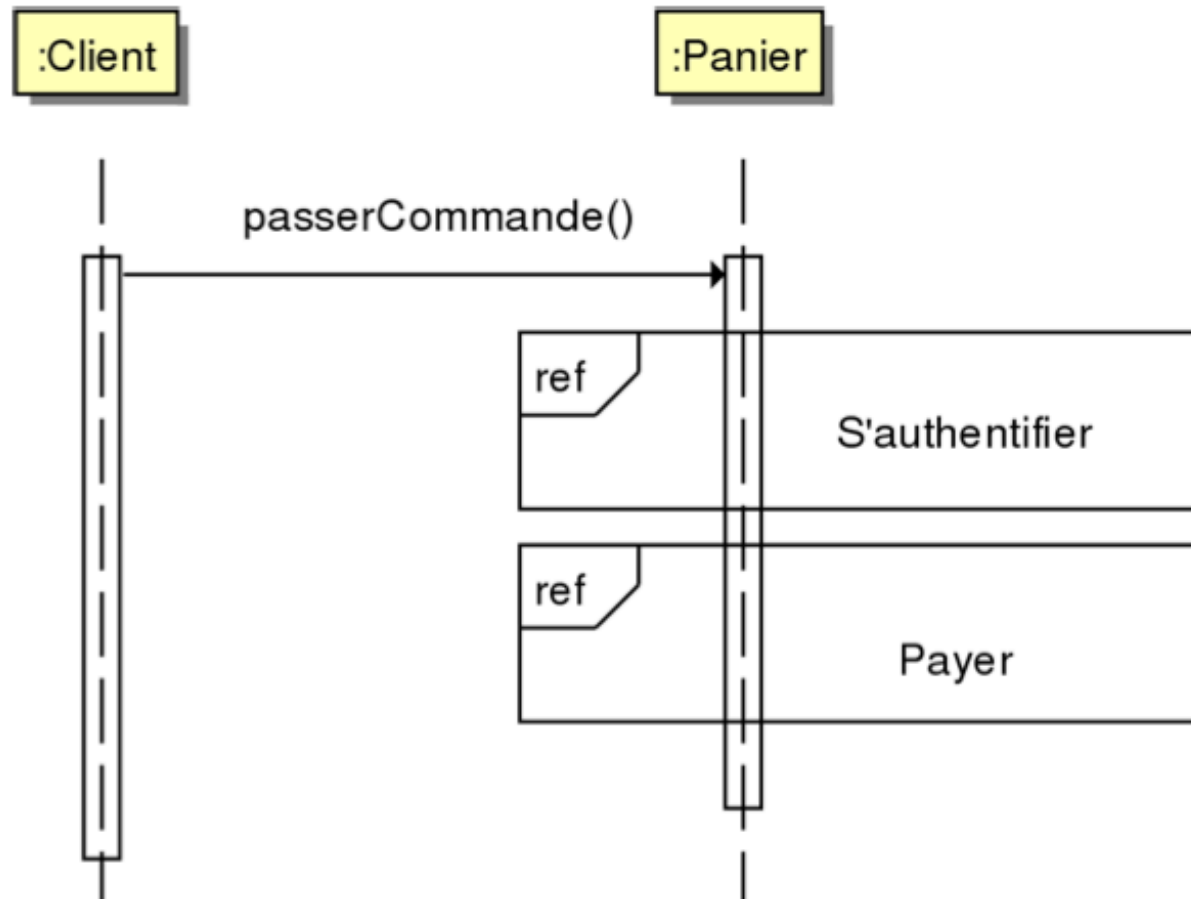
Opérateur parallèle

- L'opérateur **par** permet d'envoyer des messages en parallèle.
- Ce qui se passe de part et d'autre de la ligne pointillée est indépendant.



Réutilisation d'une interaction

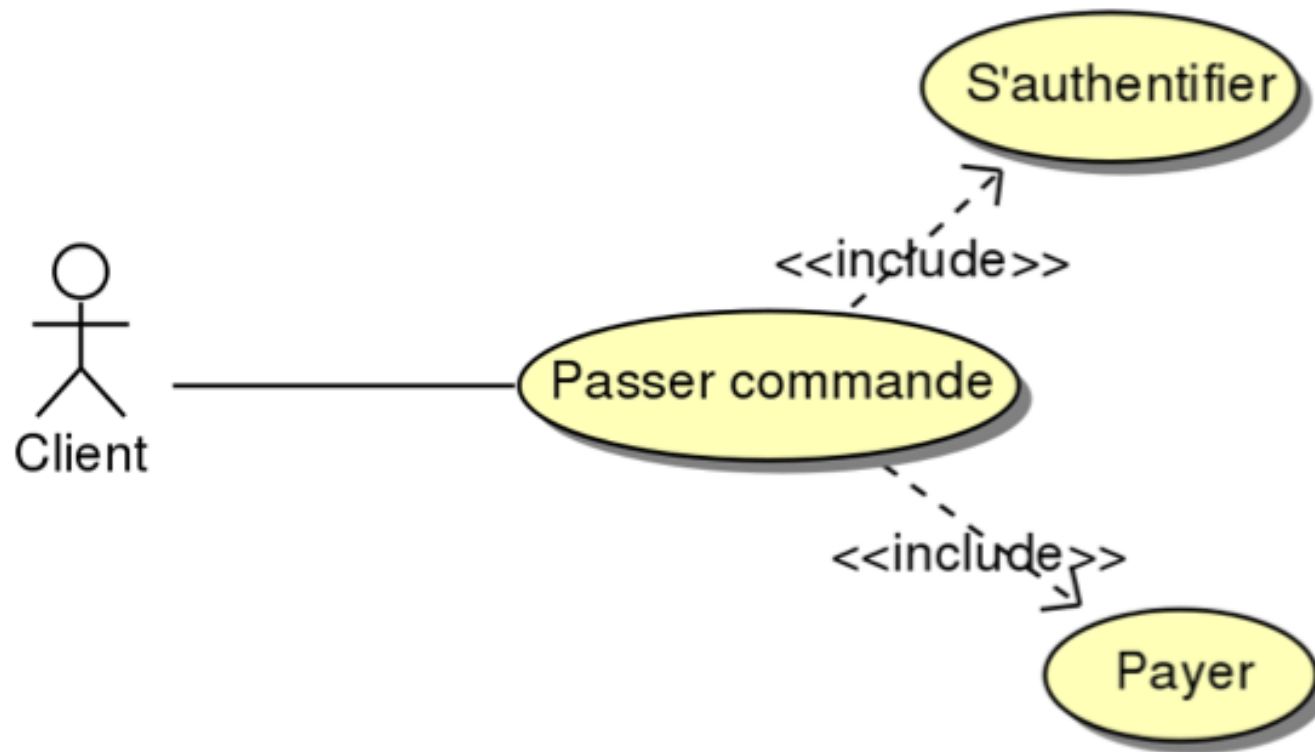
- Réutiliser une interaction consiste à placer un fragment portant la référence « ref » là où l'interaction est utile.



- On spécifie le nom de l'interaction dans le fragment.

Du CU au DSC

- Chaque cas d'utilisation donne lieu à un diagramme de séquences



- Les inclusions et les extensions sont des cas typiques d'utilisation de la réutilisation par référencement

Utilisation d'un DS pour spécifier une méthode

- Une interaction peut être identifiée par un fragment sd (pour pour « sequence diagram »)précisant son nom

➤ Un message peut partir du bord de l'interaction, spécifiant le comportement du système après réception du message, quel que soit l'expéditeur

