Diagramme de séquences

Comment ?

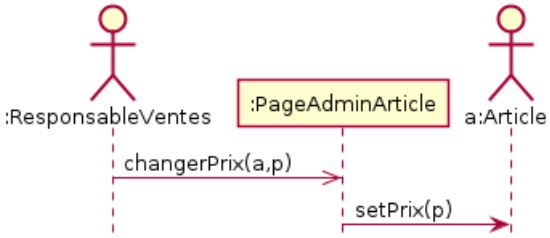
- Les diagrammes de cas d'utilisation modélisent à QUOI sert le système, en organisant les interactions possibles avec les acteurs.
- Les diagrammes de classes permettent de spécifier la structure et les liens entre les objets dont le système est composé : ils spécifie QUI sera à l'oeuvre dans le système 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 coeur d'un système interagissent en s'échangent des messages.
    - Les acteurs interagissent avec le système au moyen d'IHM (Interfaces Homme-Machine).

Interaction

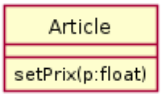
Pour être complètement spécifiée, une interaction doit être décrite dans plusieurs diagrammes UML :

- Cas d'utilisation

- Séquences



- Classes pour spécifier les opérations nécessaires



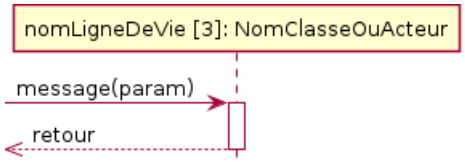
Ligne de vie

Une ligne de vie représente un participant à une interaction (objet ou acteur). La syntaxe de son libellé est :

```
nomLigneDeVie {[selecteur]}: NomClasseOuActeur
```

Une ligne de vie est une instance, donc il y a nécessairement les *deux points* (*:*) dans son libellé.

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 séquence sont les messages échangés entre les lignes de vie :

- Ils sont représentés par des flèches
- Ils sont présentés du haut vers le bas le long des lignes de vie, 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.

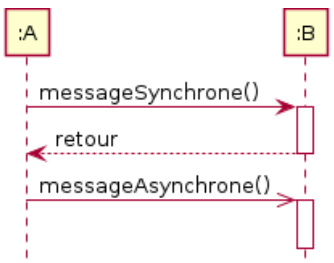
La réception des messages provoque une période d'activité (rectangle vertical sur la ligne de vie) marquant le traitement du message (spécification d'exécution dans le cas d'un appel de méthode).

Messages synchrones et asynchrones

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

- Si un objet A envoie un message synchrone à 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.



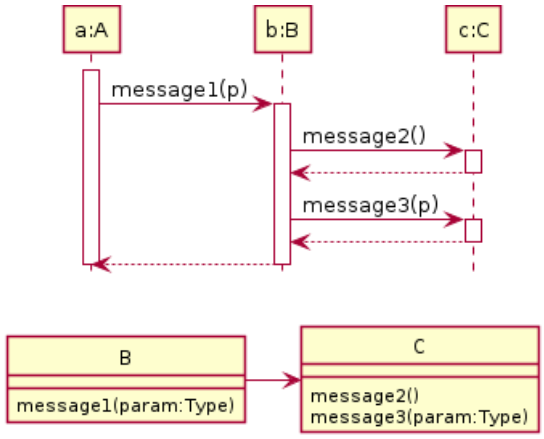
Messages synchrones et diagramme de classe

Les messages synchrones correspondent le plus souvent à une opération :

- A l'invocation, le flux contrôle passe de l'émetteur au récepteur
- L'émetteur attend la fin de l'exécution, et reprend après le retour

Les méthodes correspondant aux messages synchrones doivent être définies dans un diagramme de classes.

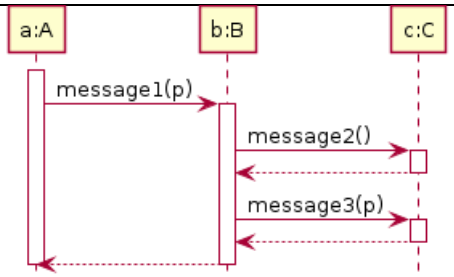
Les méthodes sont définies dans la classe du récepteur, et pas de l'émetteur du message.



La flèche dans le diagramme de classes correspond à une association unidirectionnelle, et pas à un message : la notion de message n'a aucun sens dans le contexte d'un diagramme de classes.

Echange de messages et code Java

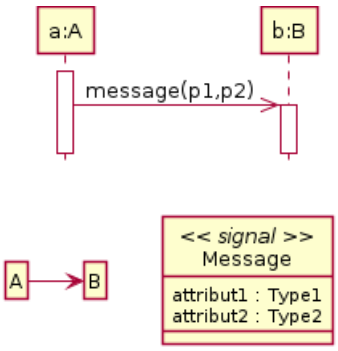
```
1  class B {
2      C c;
3      message1 (p:Type) {
4          c.message2 ();
5          c.message3 (p);
6      }
7  }
8  class C {
9      message2 () {
10         ...
11     }
12     message3 (p:Type) {
13         ...
14     }
```



Messages asynchrones et diagramme de classe

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.



Syntaxe des messages

La syntaxe des messages est :

```
nomSignalOuOperation (LISTE_PARAMS)
```

avec `LISTE_PARAMS` un liste de paramètres séparés par des virgules. Dans la liste des paramètres, on peut utiliser les notations suivantes :

- pour donner une valeur à un paramètre spécifique : `nomParametre = valeurParametre`
- pour préciser que l'argument est 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.

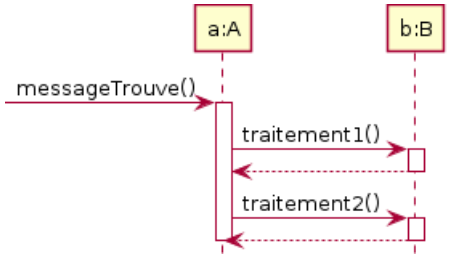
Ils sont utilisés pour spécifier le résultat de la méthode invoquée. Leur syntaxe est :

```
attributCible = nomMessageSynchroneInitial(LISTE_PARAMS) : valeurRetour
```

Les messages de retour sont représentés en pointillés.

Messages trouvés

Les diagrammes de séquences peuvent être employés pour décrire les traitements d'un système résultant de l'envoi d'un message, indépendamment de l'émetteur. Dans ce cas, l'émetteur importe peu et on le le spécifie pas.



Les messages trouvés peuvent être synchrones ou asynchrones.

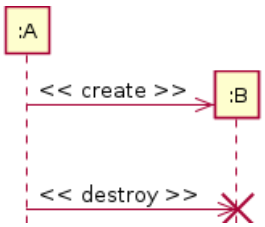
Messages perdus

Des messages perdus, on connaît l'émetteur mais pas le récepteur, à l'inverse des messages trouvés.

On utilise souvent des messages de retour perdus pour spécifier le résultat d'un message synchrone trouvé.

Création et destructiob d'objets (et de lignes de vie)

- Création : message asynchrone stéréotypé << create >> pointant vers le rectangle en tête de la ligne de vie
- Destruction : message asynchrone stéréotype << destroy >> précédant une croix sur la ligne de vie



Fragment combiné

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).

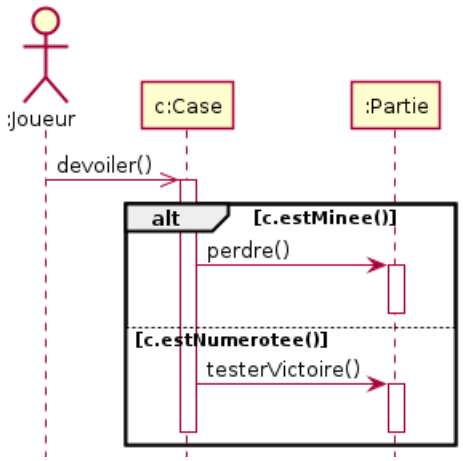
Un fragment combiné se représente de la même façon qu'une interaction. Il est représenté un rectangle dont le coin supérieur gauche contient un pentagone.

Dans le pentagone figure le type de la combinaison (appelé *opérateur d'interaction*).

Fragment alt : opérateur conditionnel

Les différentes alternatives sont spécifiées dans des zones délimitées par des pointillés.

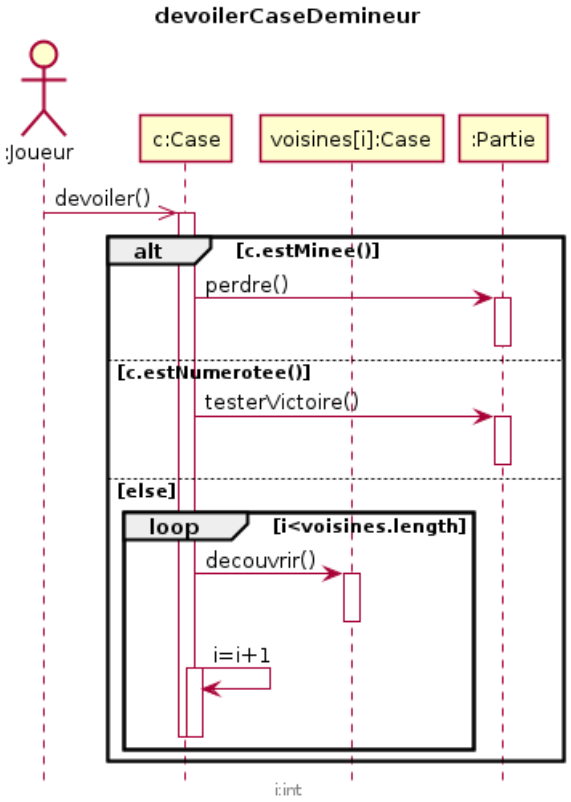
- Les conditions sont spécifiées entre crochets dans chaque zones.
- On peut utiliser une clause `[else]`



Fragment loop : opérateur d'itération

Le fragment `loop` permet de répéter ce qui se trouve en son sein.

On peut spécifier entre crochets à quelle condition continuer.



Remarques

- Les fragments peuvent s'imbriquer les uns dans les autres
- On peut émettre des messages réflexifs et dans ce cas, on définit une activité "dans" l'activité
- Lorsqu'on décrit une opération dans le détail, il est permis (mais pas recommandé) de placer des commandes sur les flèches au lieu de messages correspondant à des opérations ou des signaux
- Tous les éléments d'un diagramme doivent être définis. Typiquement, les attributs doivent correspondre :
    - soit à des attributs définis dans un diagramme de classes au niveau de la ligne de vie contrôlant le flux d'exécution
    - soit à des attributs définis localement au diagramme de séquence (ici, `i`)

Opérateurs de flux de contrôle

- **opt** (*facultatif**) : Contient une séquence qui peut ou non se produire. Dans la protection, vous pouvez spécifier la condition sous laquelle elle se produit.
- **alt** : Contient une liste des fragments dans lesquels se trouvent d'autres séquences de messages. Une seule séquence peut se produire à la fois.
- **loop** : Le fragment est répété un certain nombre de fois. Dans la protection, on indique la condition sous laquelle il doit être répété.
- **break** : Si ce fragment est exécuté, le reste de la séquence est abandonné. Vous pouvez utiliser la protection pour indiquer la condition dans laquelle la rupture se produira.
- **par** (*parallel*) : Les événements des fragments peuvent être entrelacés.
- **critical** : Utilisé dans un fragment `par` ou `seq`. Indique que les messages de fragment ne doivent pas être entrelacés avec d'autres messages.
- **seq** : Il existe au moins deux fragments d'opérande. Les messages impliquant la même ligne de vie doivent se produire dans l'ordre des fragments. Lorsqu'ils n'impliquent pas les mêmes lignes de vie, les messages des différents fragments peuvent être entrelacés en parallèle.
- **strict** : Il existe au moins deux fragments d'opérande. Les fragments doivent se produire dans l'ordre donné.

Opérateurs d'interprétation de la séquence

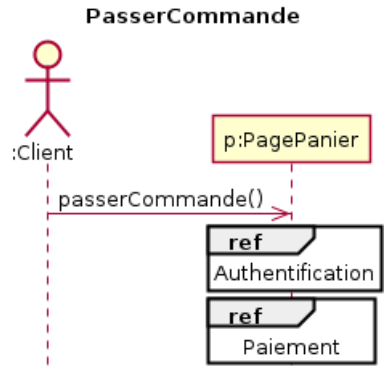
- **consider** : Spécifie une liste des messages que ce fragment décrit. D'autres messages peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.
- **ignore** : Liste des messages que ce fragment ne décrit pas. Ils peuvent se produire dans le système en cours d'exécution, mais ils ne sont pas significatifs quant aux objectifs de cette description.
- **assert** : Le fragment d'opérande spécifie les seules séquences valides. Généralement utilisé dans un fragment Consider ou Ignore.
- **neg** : La séquence affichée dans ce fragment ne doit pas se produire. Généralement utilisé dans un fragment Consider ou Ignore.

(source : msdn)

Réutilisation de séquences

Un fragment `ref` permet d'indiquer la réutilisation d'un diagramme de séquences défini par ailleurs.

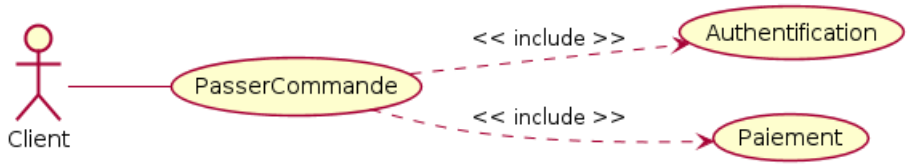
En supposant qu'il existe un diagramme intitulé `Authentification` et un autre `Paiement`, on peut établir le diagramme suivant :



Utilisation des diagrammes de séquence

Les diagrammes de séquences sont principalement utilisés pour :

- Documenter des cas d'utilisation. Dans ce cas, un acteur est toujours présent.



- Définir des opérations. Dans ce cas, on initie souvent le diagramme par un message trouvé et on est particulièrement rigoureux dans la définition des éléments du modèle.