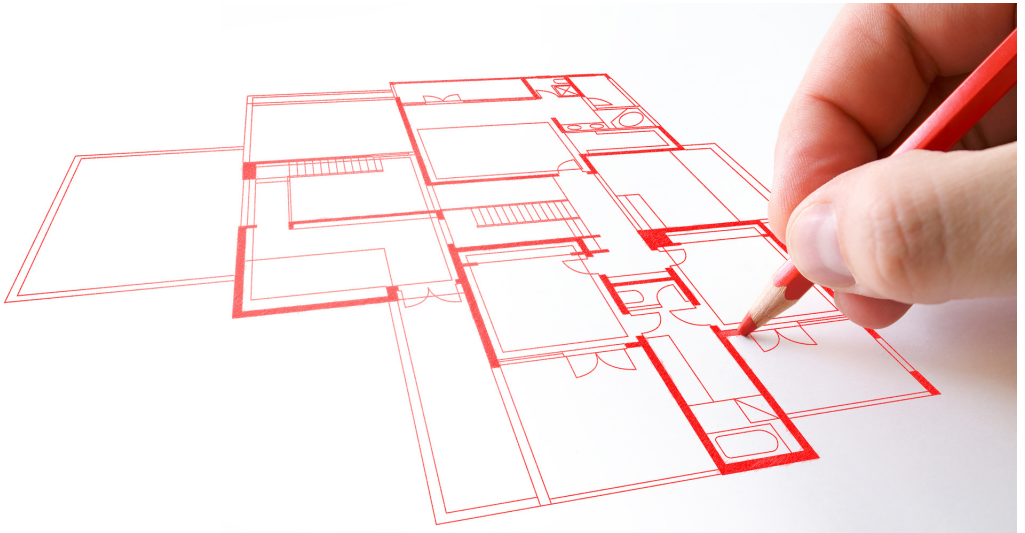
Modélisation

Principe : un modèle est une abstraction permettant de mieux comprendre un objet complexe (bâtiment, économie, atmosphère, cellule, logiciel, …).

Autre principe : *un petit dessin vaut mieux qu'un long discours*. Les modèles sont donc souvent graphiques, même si l'objet à créer n'est pas matériel.



Modélisation en informatique (1/2)

La construction d'un système d'information, d'un réseau, d'un logiciel complexe, de taille importante et par de nombreuses personnes oblige à modéliser.

Le modèle d'un système informatique sert :

- de document d'échange entre clients et développeurs
- d'outil de conception
- de référence pour le développement
- de référence pour la maintenance et l'évolution

Modélisation en informatique (2/2)

Modéliser les logiciels et utiliser des méthodologies de développement n'est pas un luxe ou une perte de temps!

D'après le Standish Group (*www.standishgroup.com*), en 2013 :

- 39 % des projets de développement logiciel ont abouti à un logiciel conforme aux attentes du client en respectant les délais et les coûts.

- 43 % des projets ont abouti à un logiciel non conforme aux attentes du client, ou ne respectant pas les délais ou les coûts.

- 18 % des projets n'ont pas abouti.

NB : le taux de succès est de 76 % pour les petits projets, mais 10 % pour les gros (au dela de 10 millions de dollars).

Modélisation en ingénierie logicielle

Langage de modélisation : une syntaxe commune, graphique, pour modéliser (OMT, UML, …).

Méthode de modélisation : procédé permettant de construire un modèle aussi correct que possible et aussi efficacement que possible (MERISE, UP, …).

Les méthodes de modélisation sont souvent des méthodes de développement qui comportent une partie modélisation.

Certaines méthodes sont associées à un langage de modélisation (par exemple MERISE).

Etapes du cyle de vie du logiciel (1/2)

Analyse des besoins et des risques

Spécifications (ou conception générale) : l'architecture générale

Conception détaillée : l'architecture détaillée et la description de tous les éléments

Codage (ou implémentation) : traduction du modèle dans un langage de programmation

Tests unitaires : vérification de chaque élément du logiciel par rapport aux spécifications

Etapes du cyle de vie du logiciel (2/2)

Intégration : vérification de l'interfaçage des différents éléments du logiciel.

Qualification (ou recette) : vérification de la conformité du logiciel aux spécifications initiales

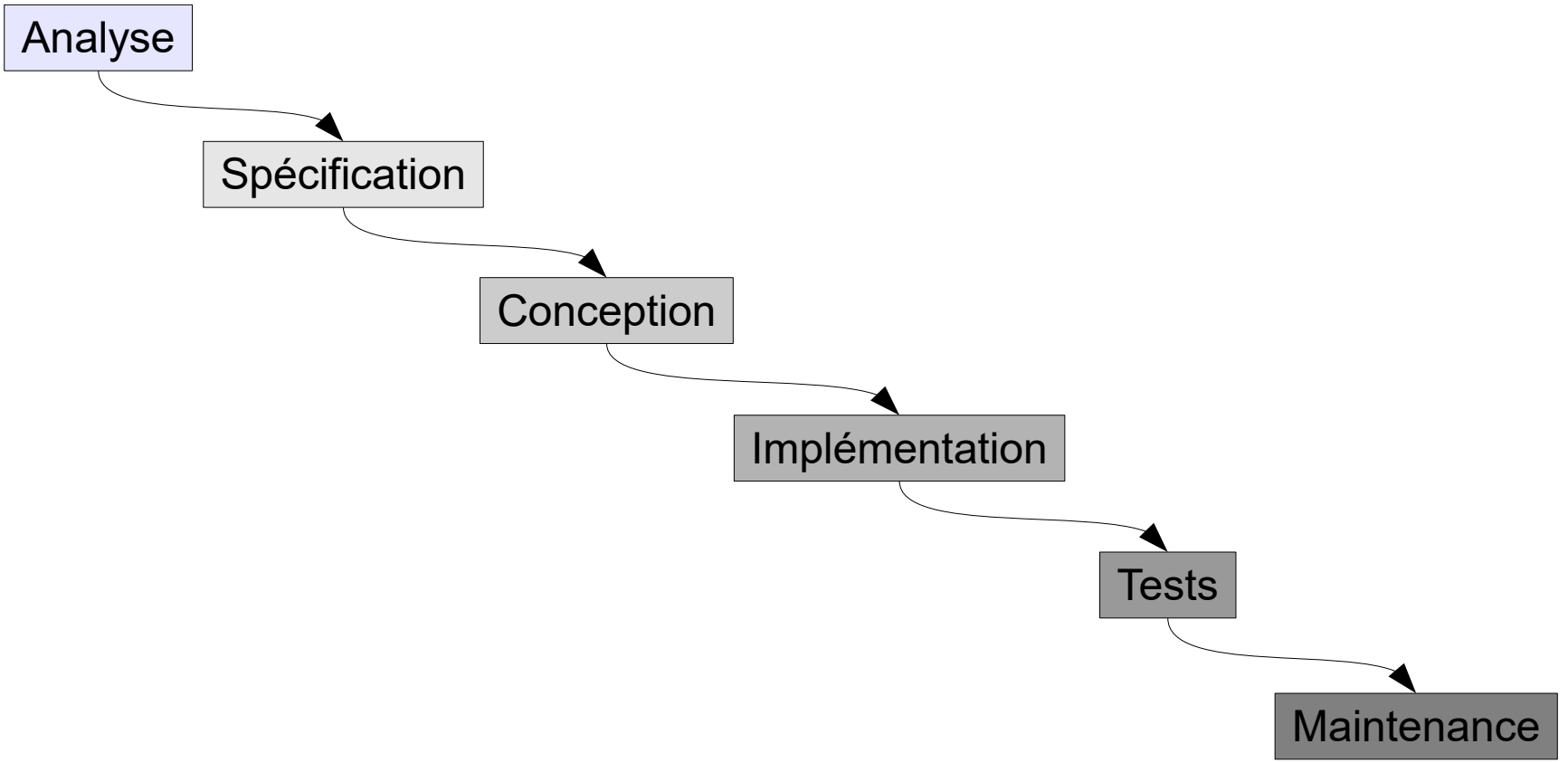
Documentation

Mise en production (déploiement)

Maintenance corrective et évolutive

Méthodes de développement (1/5)

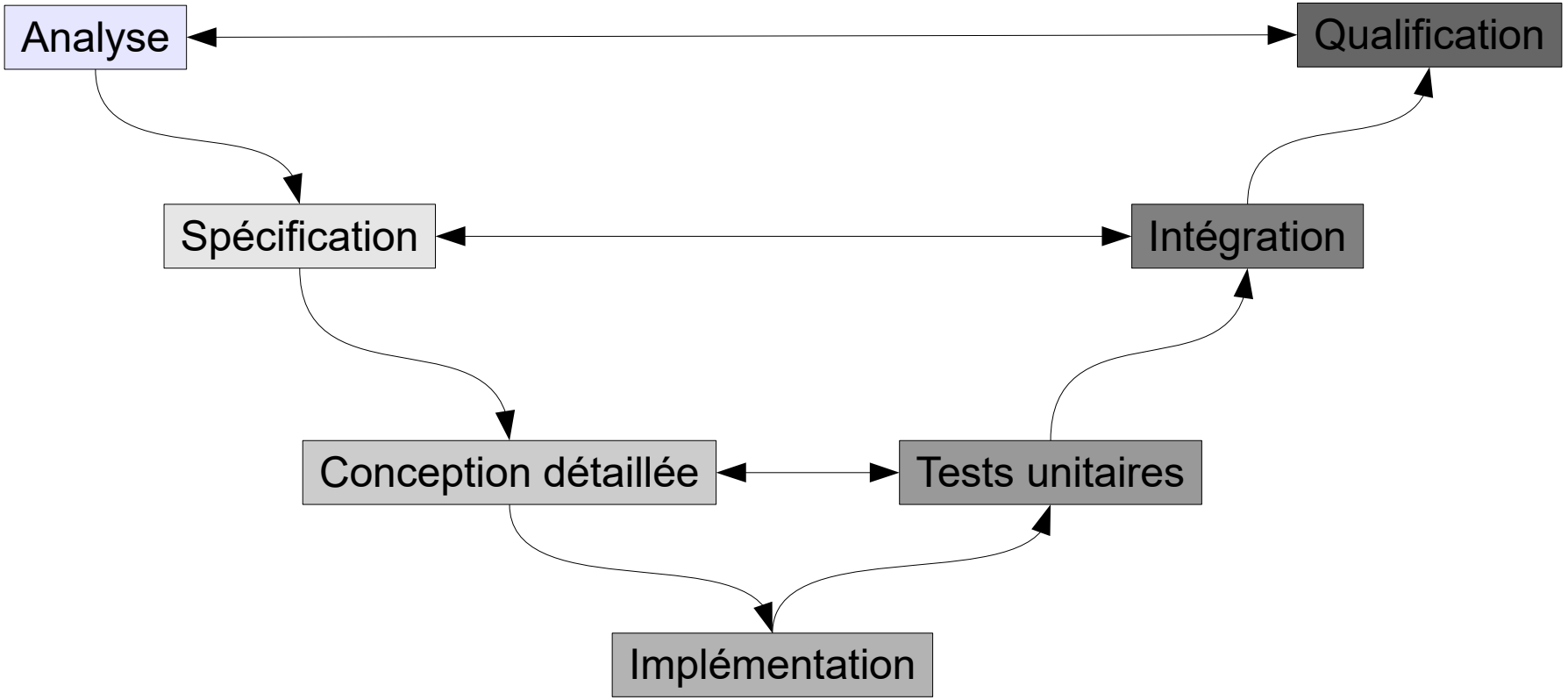
Méthodes en cascade



Méthodes de développement (2/5)

Cycles en V

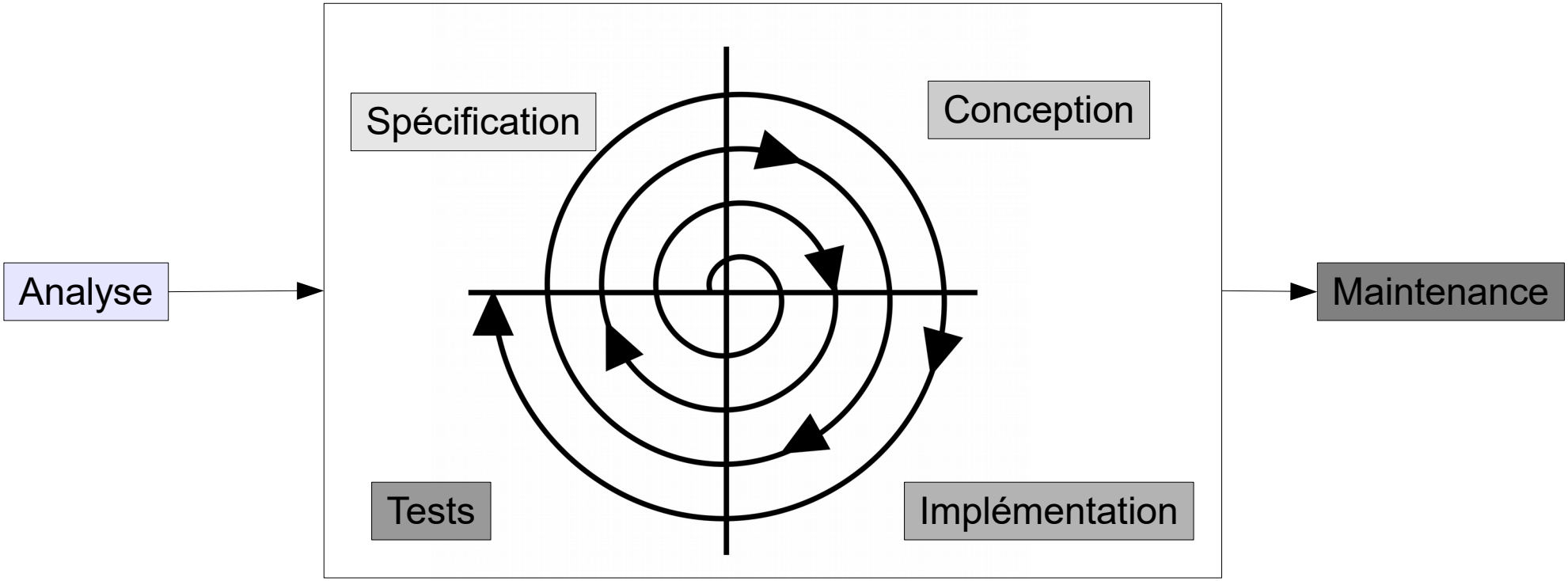
Les phases de test/intégration/qualification produisent des résultats qui amènent à modifier les éléments produits dans les phases en vis-à-vis, limitant les aller-retours tout au long de la chaîne de développement.



Méthodes de développement (3/5)

Cycles en spirale

Méthode itérative : plusieurs cycles sont nécessaires avant d'obtenir un résultat correct



Méthodes de développement (4/5)

Méthodes agiles

Méthodes itératives, incrémentales (on débute par un noyau qui est enrichi au fur et à mesure) et adaptatives (acceptation des changements, planification légère).

Les méthodes agiles sont largement utilisées : Scrum, Agile Unified Process, ExtremeProgramming, …

Il existe de nombreux autres types de méthodes : *chaos model* (on se focalise sur le plus urgent), *slow programming* (on prend son temps pour améliorer la qualité), ...

Méthodes de développement (5/5)

Rapport du Standish Group (*www.standishgroup.com*), en 2013

Projets de développement logiciel utilisant la méthode en cascade :

- 14 % ont abouti correctement (fonctionnalités, délais, coûts)
- 57 % ont abouti à un résultat non correct
- 29 % n'ont pas abouti

Projets de développement logiciel utilisant une méthode agile :

- 42 % ont abouti correctement
- 49 % ont abouti à un résultat non correct
- 9 % n'ont pas abouti

Mais les compétences et motivations des participants sont plus cruciales que la méthodologie ...

UP : une méthode de développement pour UML (1/2)

UP (Unified Process) est une méthode agile issue d'autres méthodes et langages propres à la POO, en particulier OMT (Object Modeling Technology) et OOSE (Object Oriented Sofware Engineering).

La première version date de 1998 et les évolutions ultérieures visent à coller au langage UML.

UP est itérative, incrémentale, pilotée par les cas d'utilisations (les besoins) et centrée sur l'architecture du logiciel.

UP : une méthode de développement pour UML (2/2)

Un cycle UP comporte 4 phases contenant chacune des itérations :

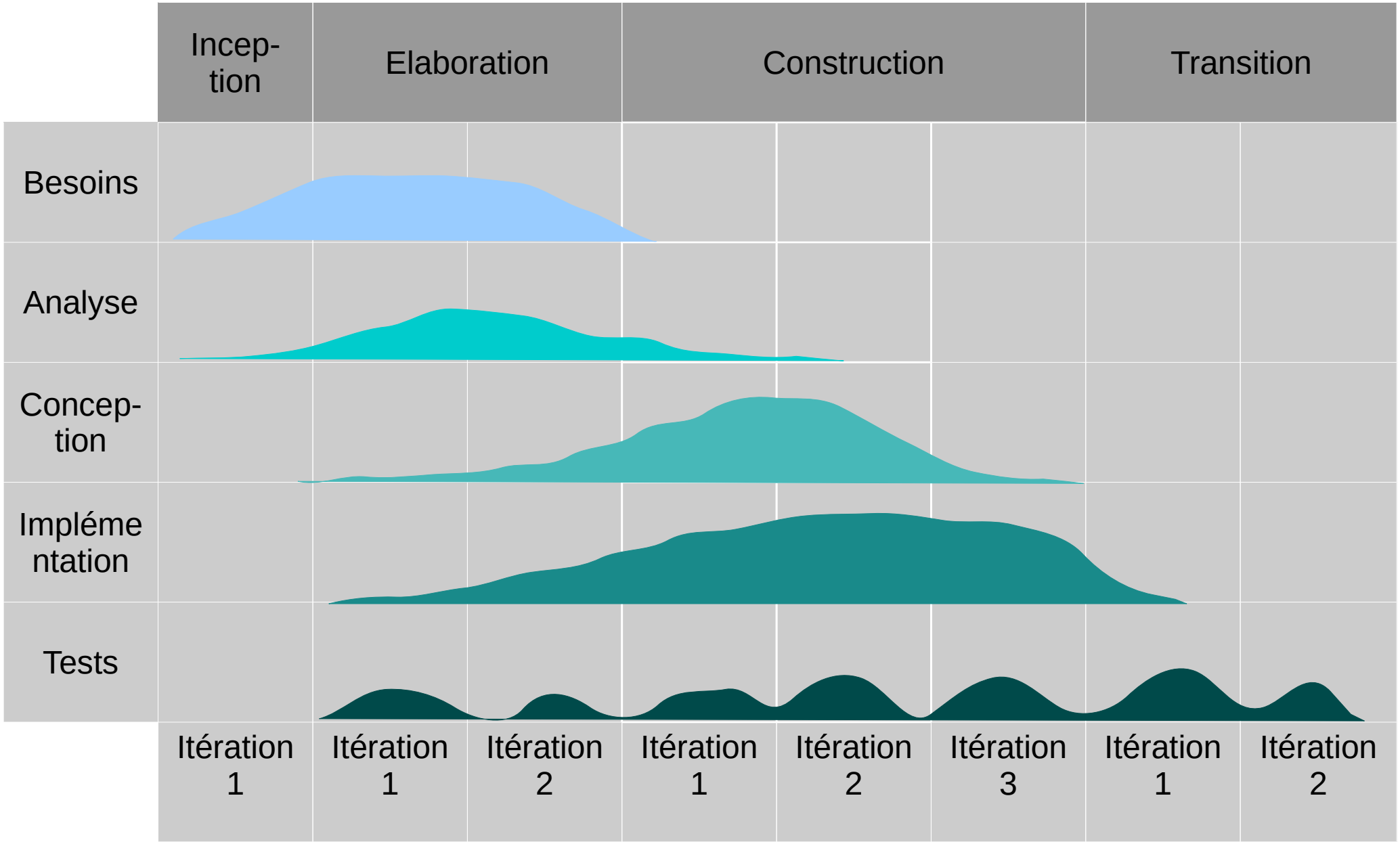
Inception : définir le projet, les coûts, les risques, premier jet d'architecture

Elaboration : analyse des besoins, spécification de l'architecture, début de conception

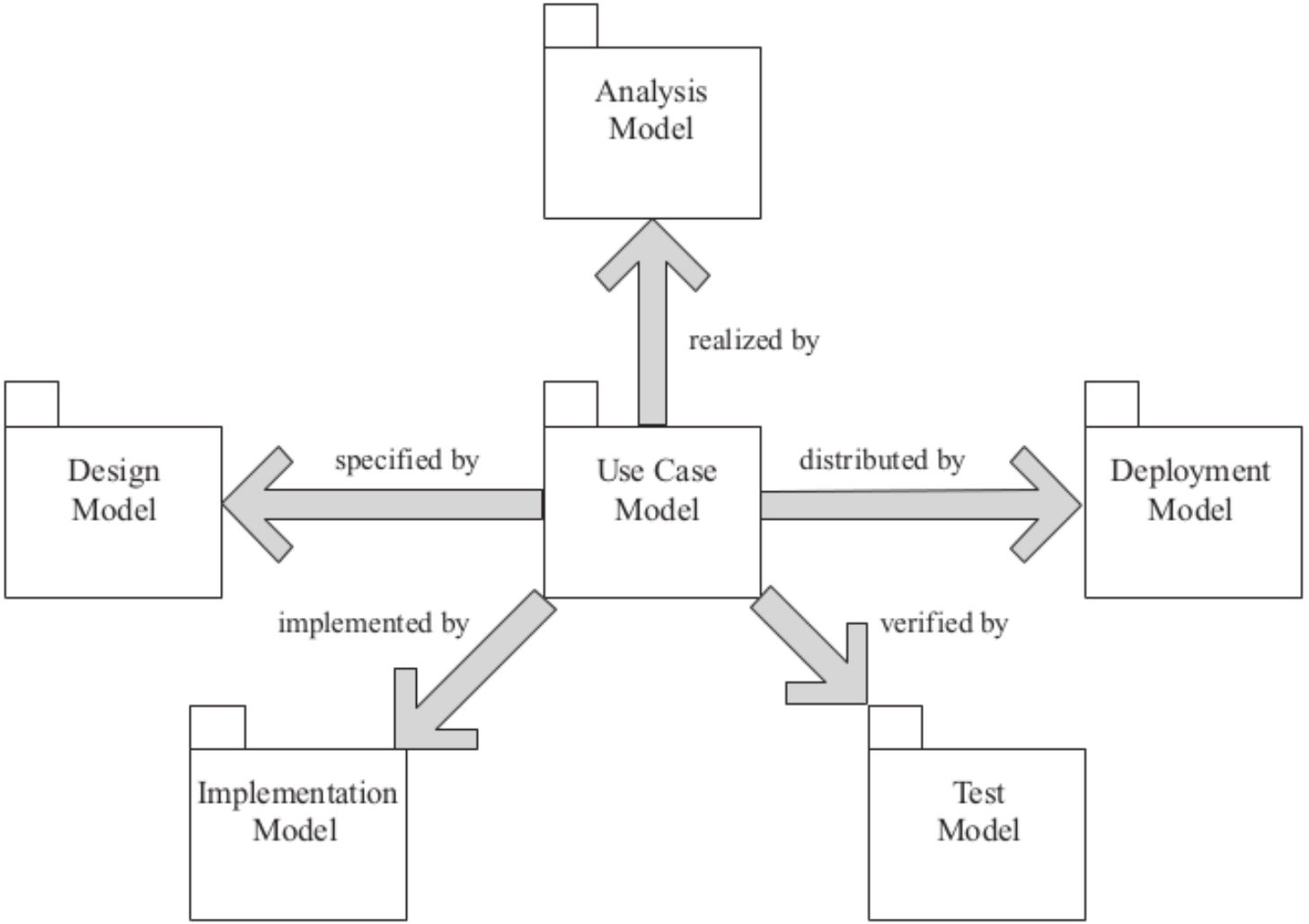
Construction : conception détaillée, implémentation, tests, intégration

Transition : tests, évolution, maintenance, déploiement

UP et les tâches de développement



Les modèles dans UP



UML (Unified Modeling Language)

UML est le langage standard pour la modélisation objet. Dernière version 2.5 (mars 2015).

UML est édité par l'**OMG** (Object Managment Group) responsable de la normalisation des technologies objet.

Les spécifications UML font environ 800 pages et couvrent tous les aspects du développement logiciel, offrant des outils de modélisation de l'architecture générale aux plus petits détails.

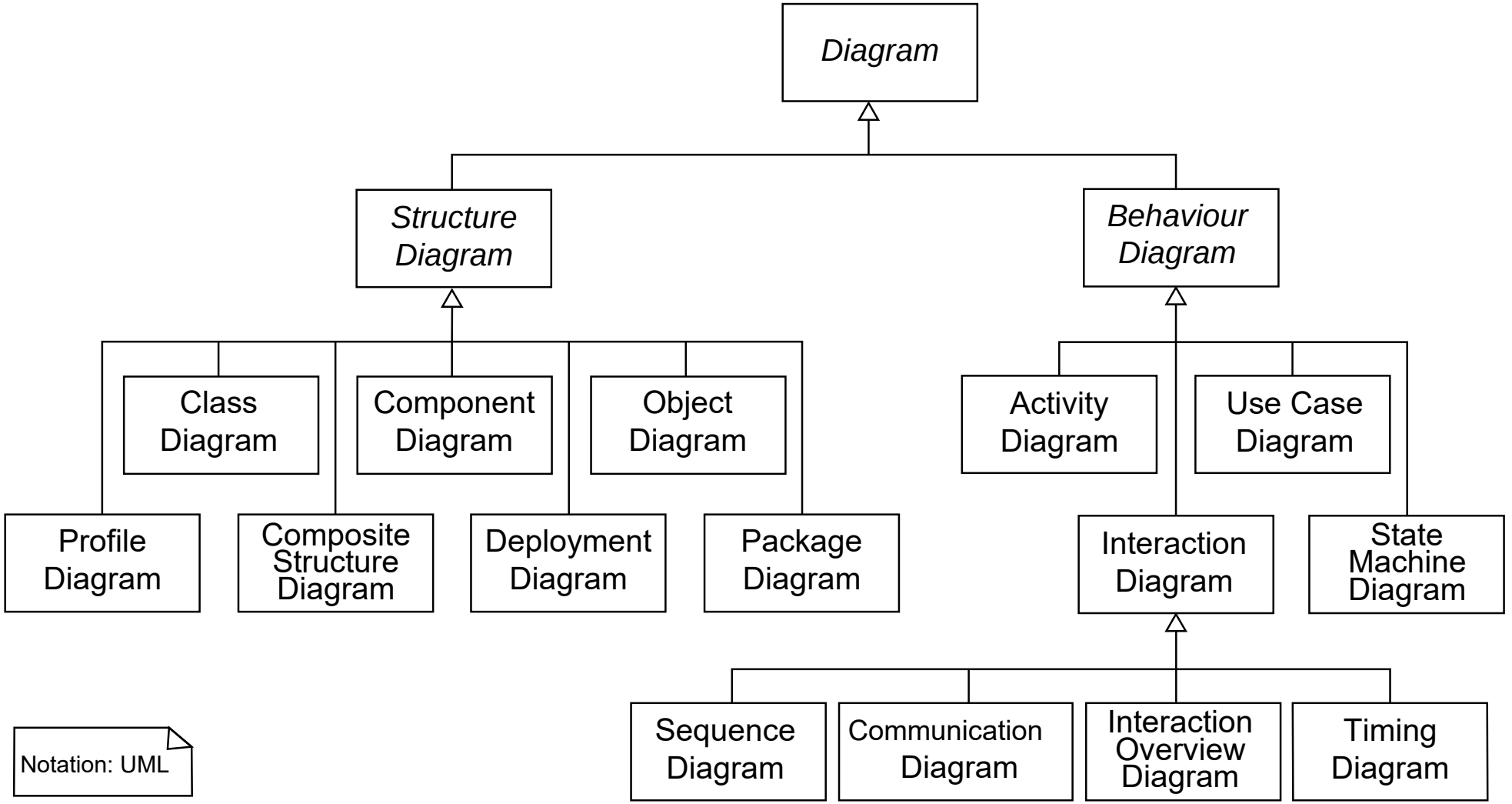
UML n'est qu'un langage

UML propose de nombreux diagrammes pouvant être plus ou moins renseignés, de façon plus ou moins formelle (sur papier, sur machine).

Un modèle UML correctement renseigné permet aux IDE prévus pour cela de générer le squelette des classes.

Un modèle UML formel et complet permet la génération automatique du code du programme (modulo le corps de certaines méthodes). C'est l'utilisation préconisée par le MDA (Model Driven Architecture).

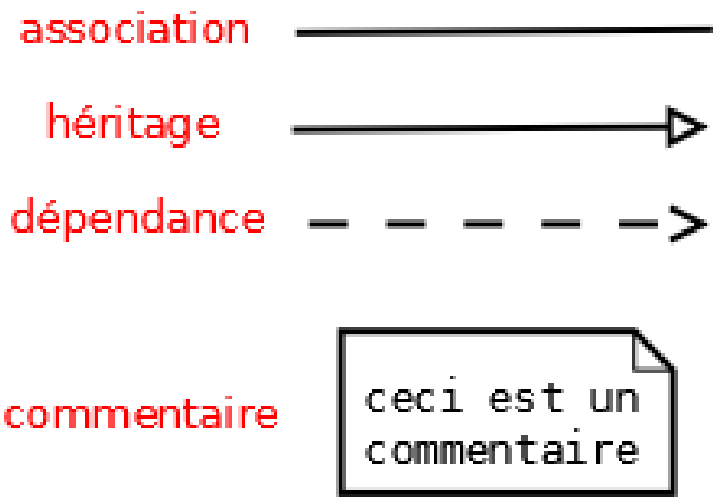
Les principaux diagrammes UML



Règles communes à tous les diagrammes UML

Les stéréotypes, mots-clés ajoutant de la sémantique aux diagrammes, sont donnés entre guillemets (`<<actor>>`, `<<realize>>`, ...) ou entre accolades (`{abstract}`, `{use}`, ...).

Une partir de la syntaxe est commune à tous les diagrammes :



UP et les diagrammes UML (1/2)

UP prévoit la construction de plusieurs modèles qui combinent chacun différents diagrammes UML.

Tâche des besoins → modèle de cas d'utilisation : *diagrammes de cas d'utilisation et de séquences*

Tâche d'analyse → modèle d'analyse : *diagrammes de classes, d'objets, de paquetages et de communication*

Tâche de conception → modèles de conception : *diagrammes de classes, d'objets, de paquetages, de séquences, d'états et d'activités*

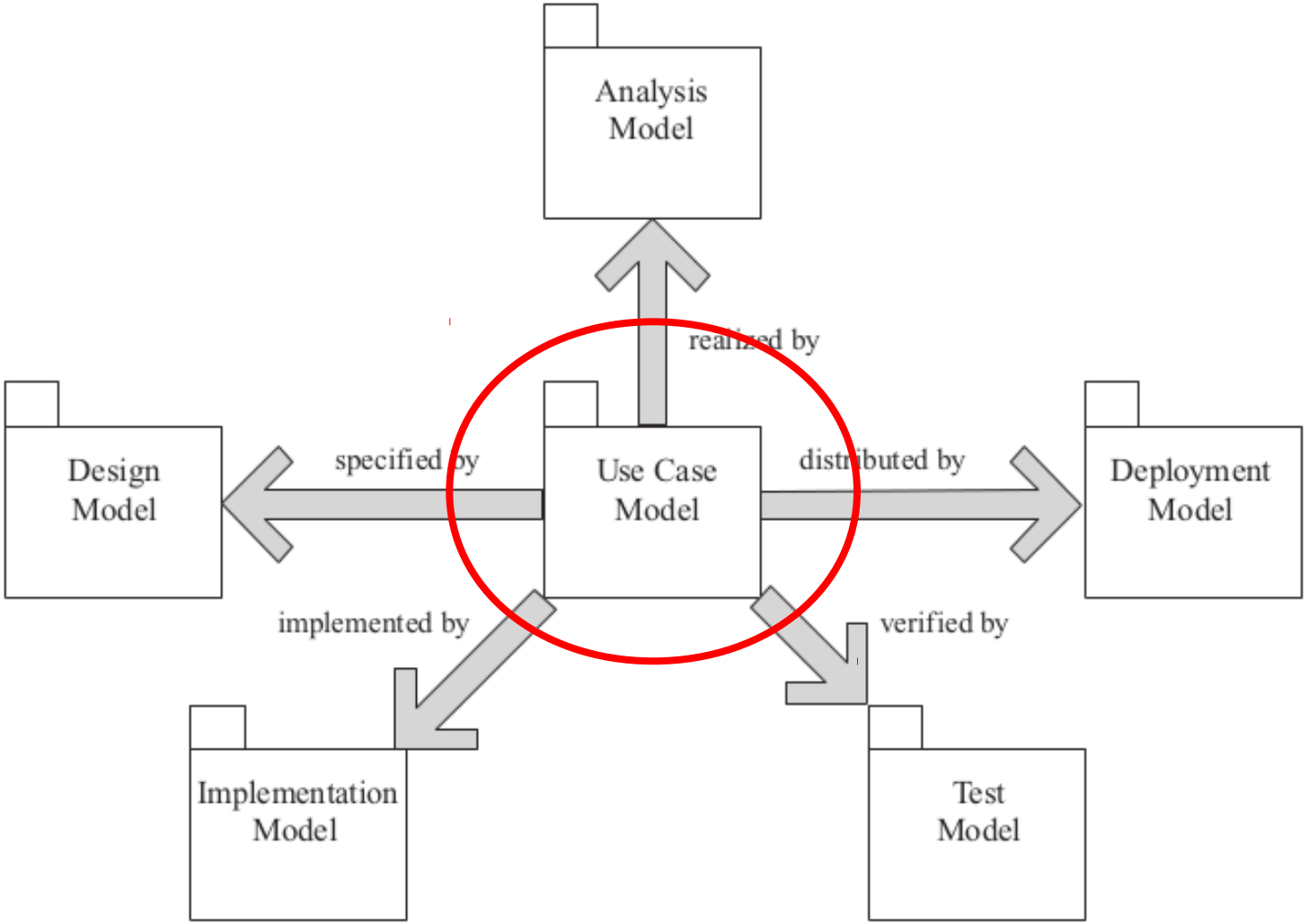
UP et les diagrammes UML (2/2)

Tâche d'implémentation → modèle d'implémentation : *diagrammes de composants et de séquences*

Tâche de déploiement → modèle de déploiement : *diagramme de déploiement et de séquences*

Tâche de test → modèle de test : *tous les diagrammes*

Les modèles dans UP



Modèle de cas d'utilisation

Dans UP, le modèle de cas d'utilisation décrit le système d'un point de vue fonctionnel.

Il se construit en relation entre les utilisateurs/clients et les développeurs.

Il permet de définir les cas d'utilisation et de détailler leurs déroulements à l'aide principalement des diagrammes de cas d'utilisation, de séquences et de communication.

Il permet d'identifier des éléments internes au système et d'entamer la définition de l'architecture du système dans le modèle d'analyse.

Diagramme de cas d'utilisation (1/2)

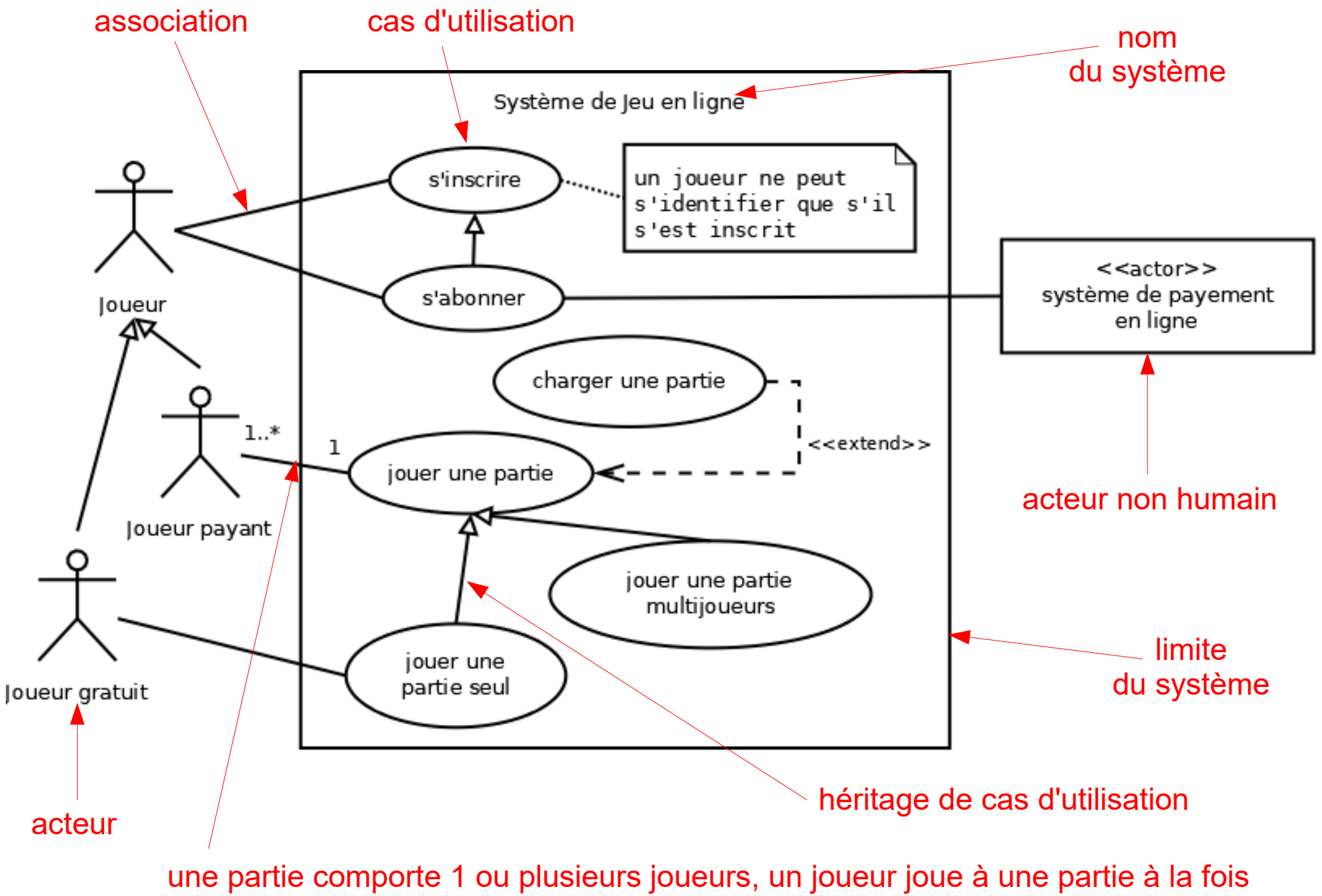
Le diagramme de cas d'utilisation est utilisé pour spécifier les fonctionnalités que le système va offrir aux utilisateurs.

Un cas d'utilisation est une action pouvant être réalisée par un acteur avec le système.

Le système représente le logiciel à modéliser. Sa limite sépare ce qui fera partie du logiciel et les acteurs externes.

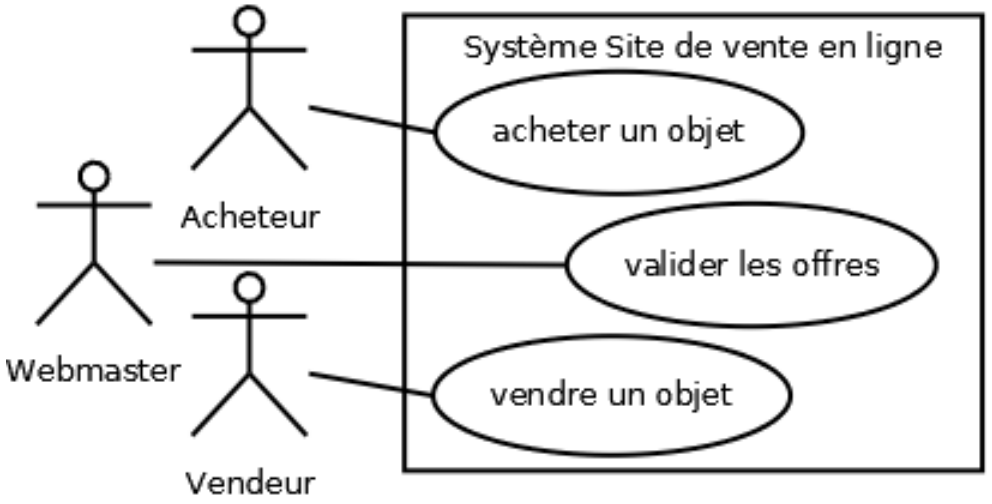
Un acteur peut être un utilisateur, un autre système, un SGBD extérieur au système, etc.

Diagramme de cas d'utilisation (2/2)

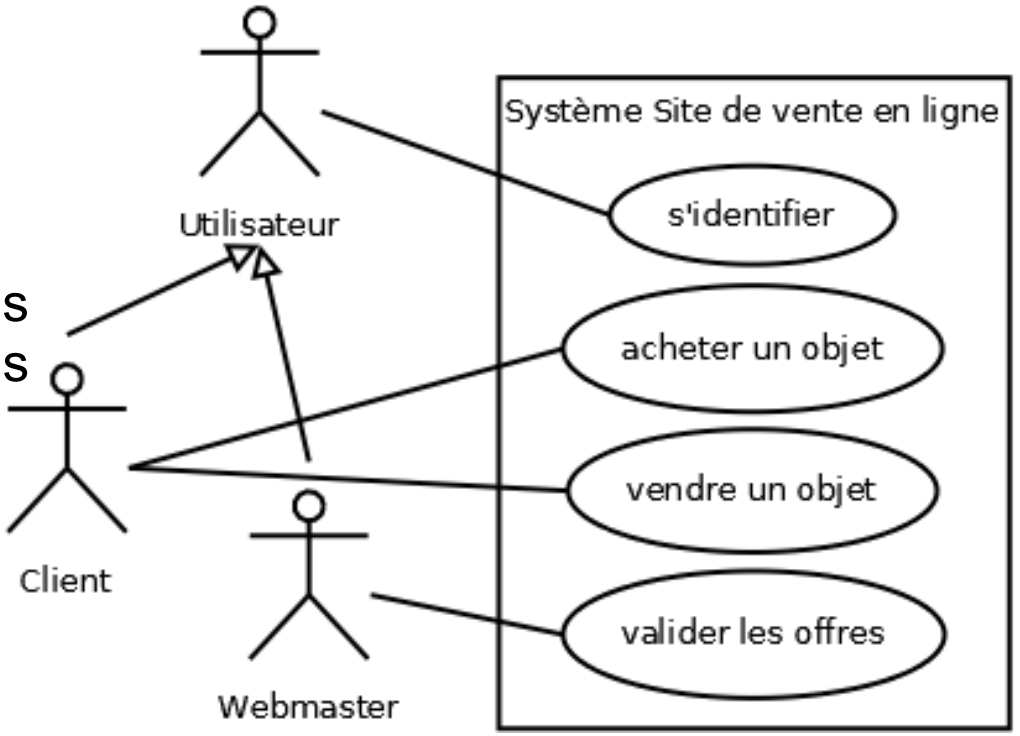


Acteurs (1/3)

Les acteurs sont des rôles, pas forcément des utilisateurs. Une même personne physique peut occuper plusieurs rôles, et inversement.

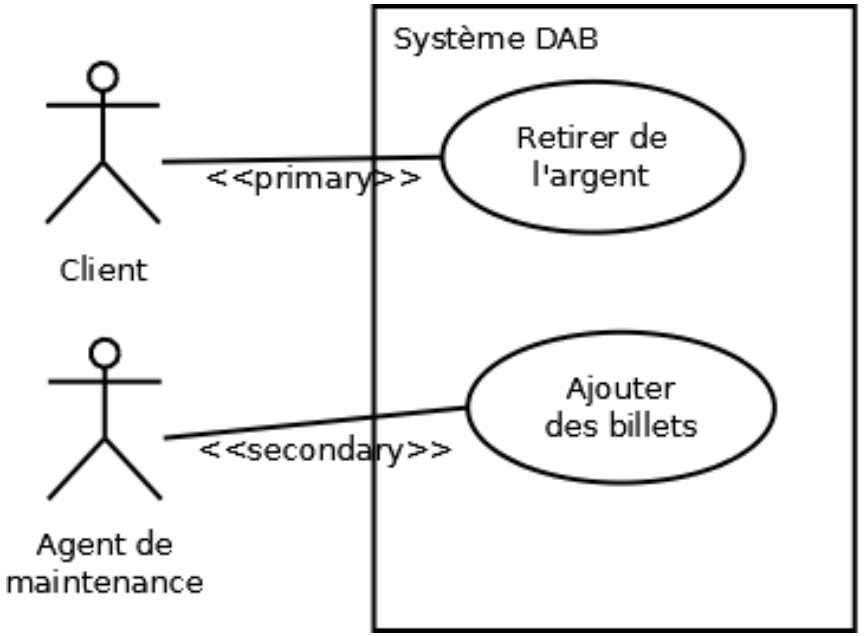


Les acteurs peuvent être organisés par héritage pour factoriser des cas d'utilisation :



Acteurs (2/3)

On peut distinguer utilisateurs primaires (qui utilisent les fonctionnalités du système) et secondaires (qui assurent les fonctionnalités et/ou maintiennent le système).



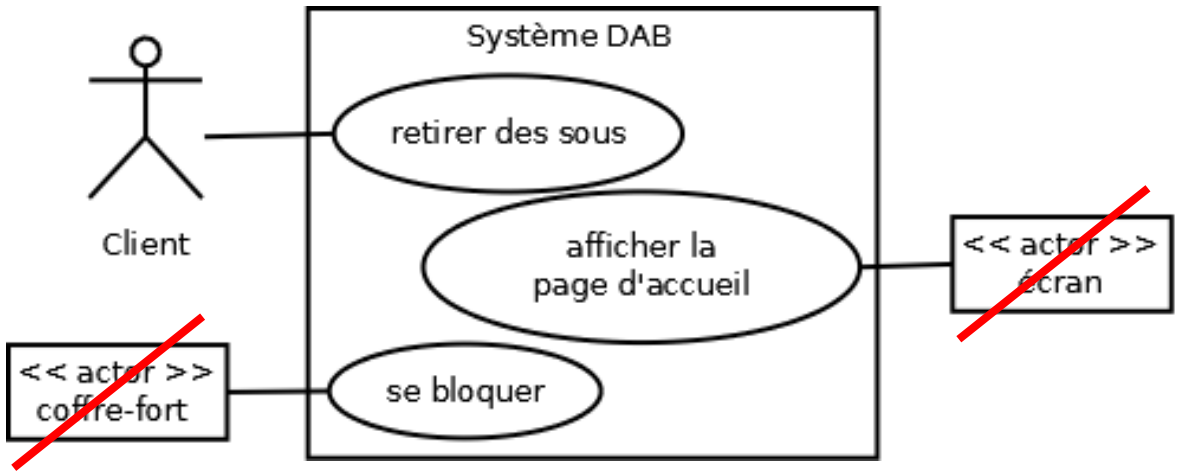
Acteurs (3/3)

Exemples d'erreurs courantes dans la modélisation des acteurs :

- confusion entre rôle et entité physique

- acteur sans cas d'utilisation

- acteur interne au système, ou dispositif d'entrée-sortie

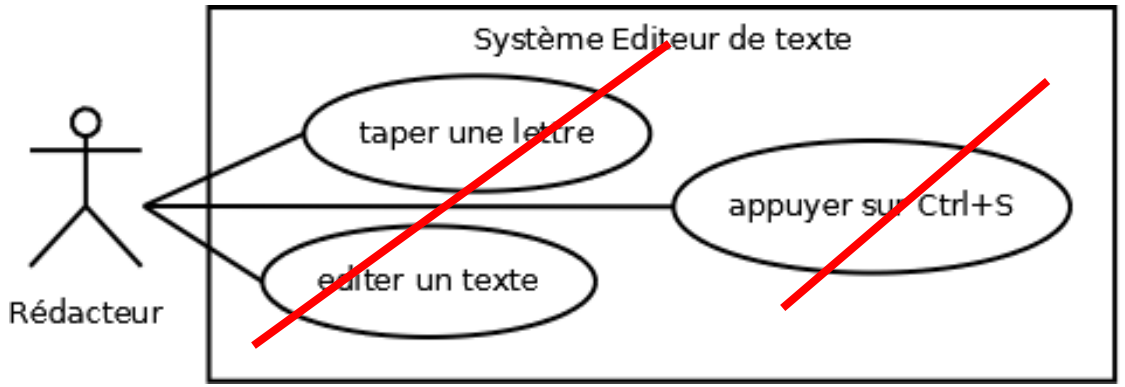


Cas d'utilisation (1/6)

Un cas d'utilisation est une collection de scénarios représentatifs d'une activité ayant du sens pour l'utilisateur.

Exemples d'erreurs courantes dans le choix des cas d'utilisation :

- *cas d'utilisation trop vague ou trop précis ou n'ayant pas de sens pour l'utilisateur*

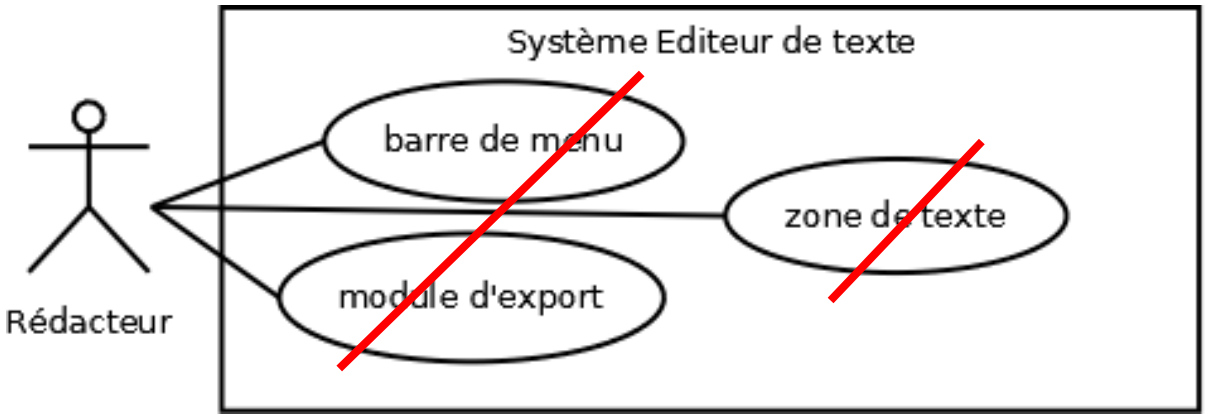


=> *remplacer par des cas d'utilisation « mettre du texte en forme », « sauver le texte dans un fichier », etc*

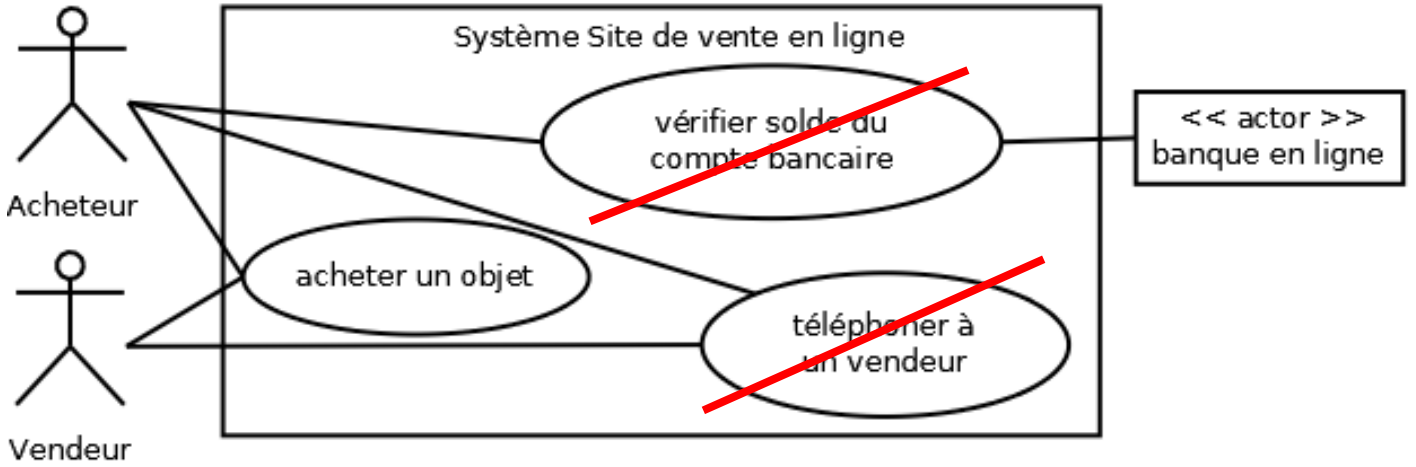
Cas d'utilisation (2/6)

Exemples d'erreurs courantes dans le choix des cas d'utilisation :

- *cas d'utilisation correspondant à des éléments du logiciel*

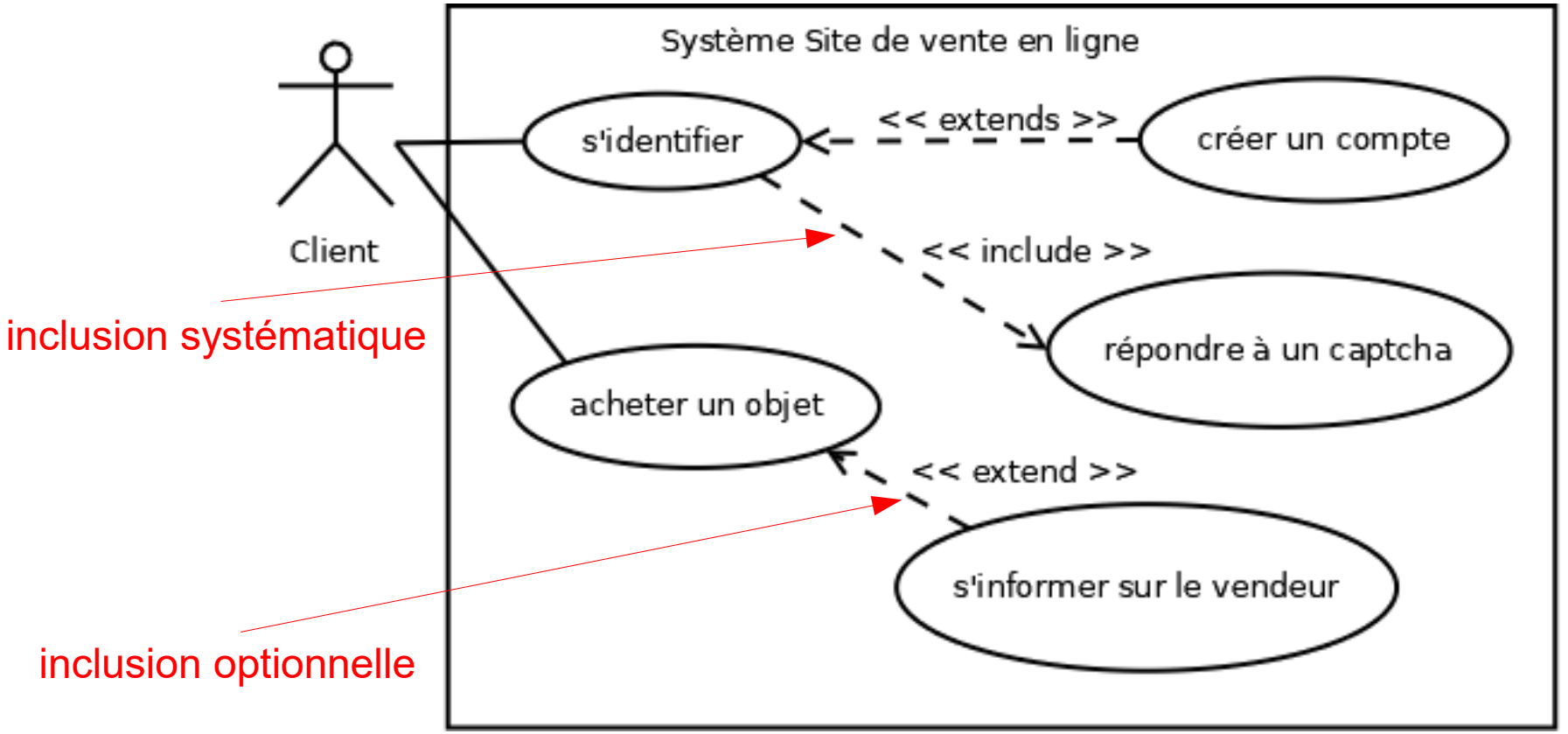


- *cas d'utilisation correspondant à une interaction entre acteurs*



Cas d'utilisation (3/6)

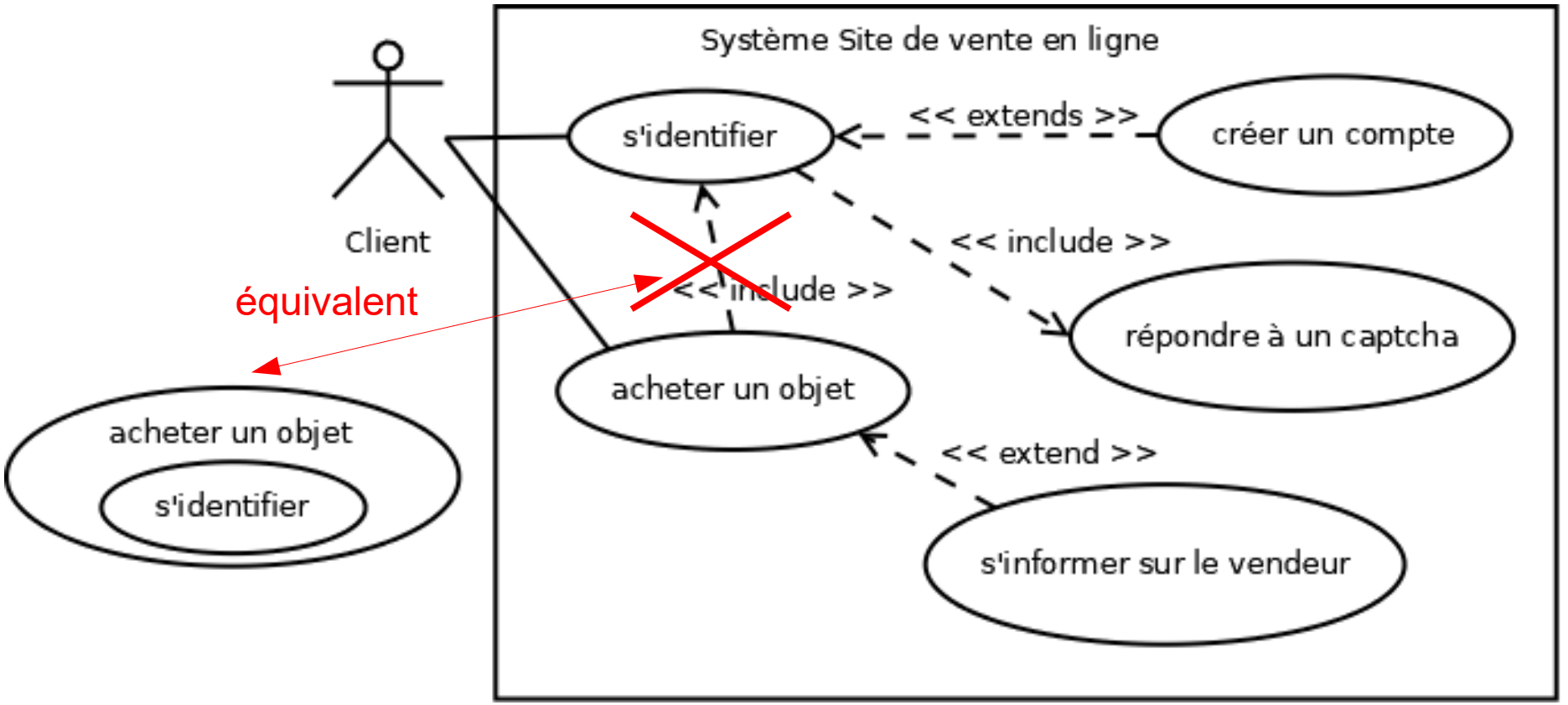
Des relations d'inclusion peuvent être spécifiées entre cas d'utilisation. Certains cas d'utilisation peuvent ne pas être liés directement à un acteur.



Cas d'utilisation (4/6)

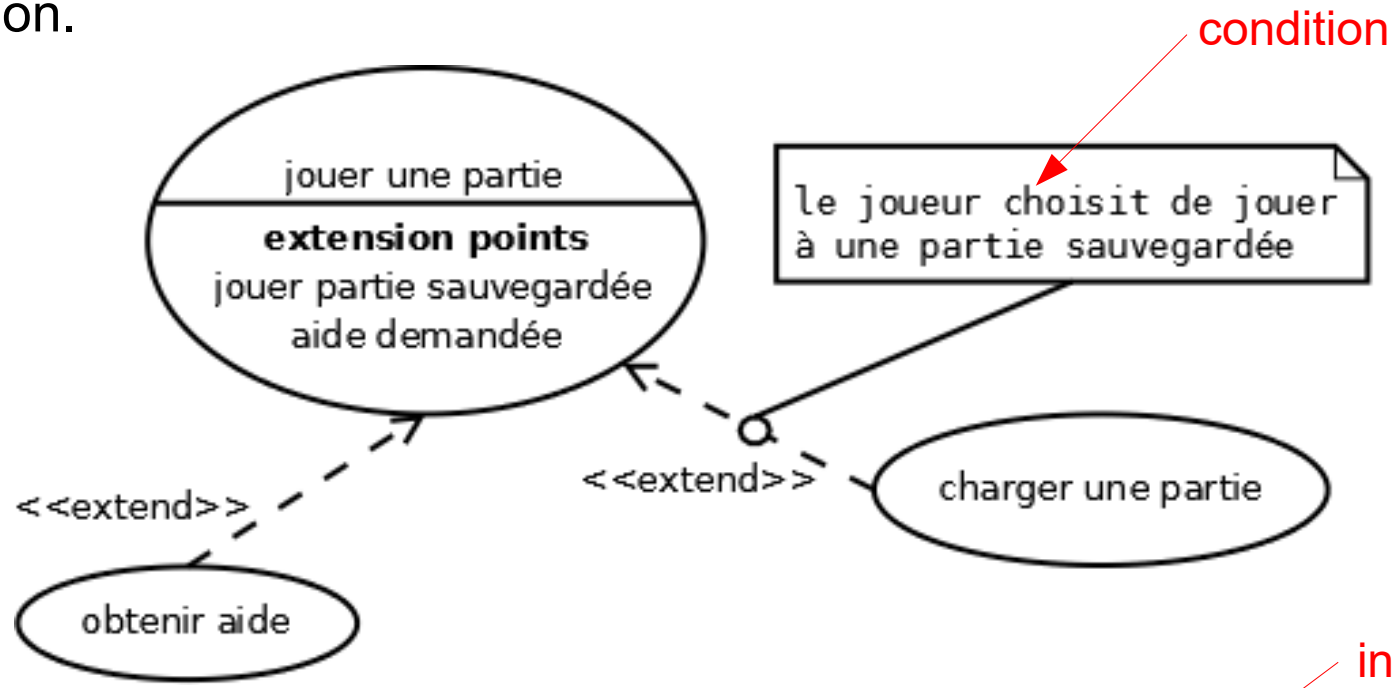
Erreur courante : vouloir absolument mettre des inclusions entre cas.

Erreur encore plus courante : mettre des inclusions correspondant en fait à des précédences temporelles

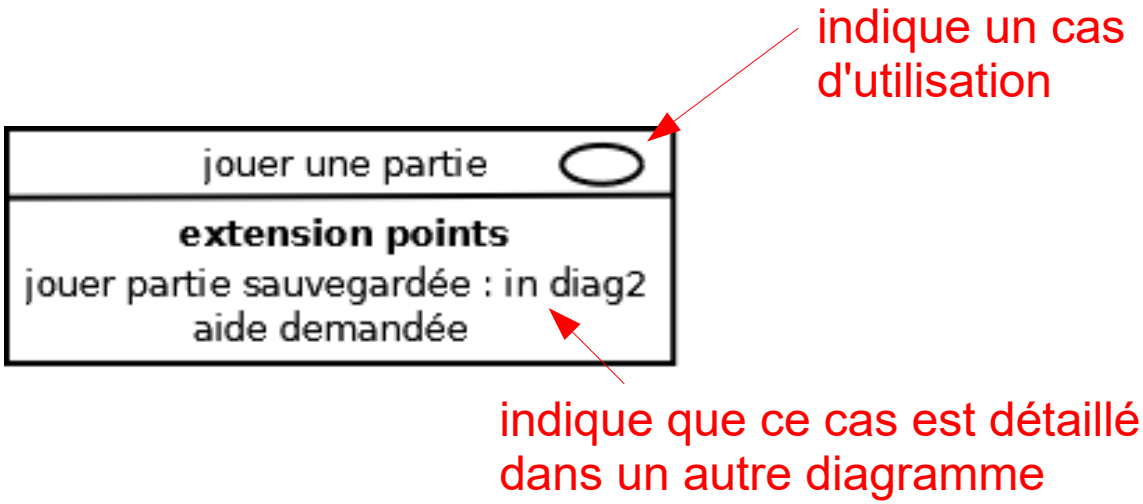


Cas d'utilisation (5/6)

Un point d'extension peut être utilisé pour spécifier les conditions d'extension.

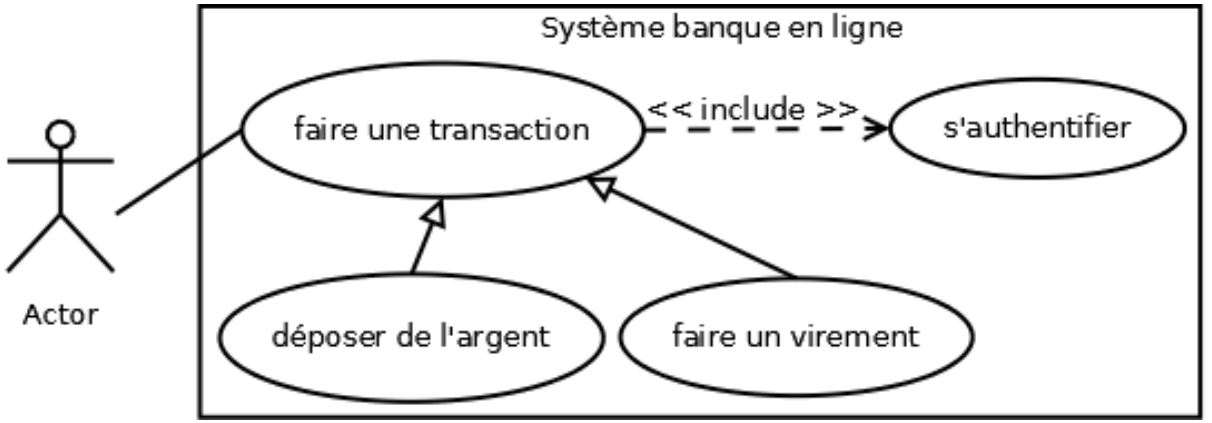


Une autre syntaxe est possible :



Cas d'utilisation (6/6)

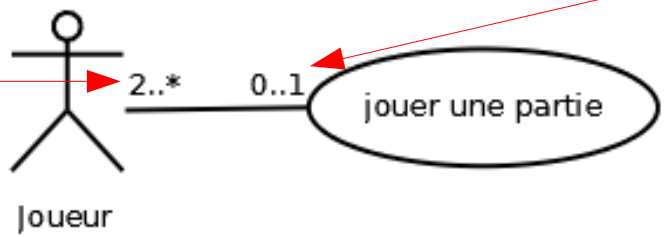
Une relation d'héritage entre cas peut être utilisée pour indiquer une spécialisation.



Des cardinalités peuvent être spécifiées entre acteurs et cas d'utilisation.

une partie est jouée par au moins 2 joueurs

un joueur ne peut jouer qu'à une partie en même temps



Diagrammes d'interaction

La description des cas d'utilisation peut être complétée par des diagrammes d'interaction, qui décrivent le comportement du système du point de vue temporel.

Un diagramme d'interaction décrit un scénario instancié :

- modèle de cas d'utilisation : interactions entre le système et les acteurs

- autres modèles : interactions entre des objets internes au système

Les deux diagrammes d'interaction principalement utilisés sont les diagrammes de séquences et les diagrammes de communication.

Diagrammes de séquences

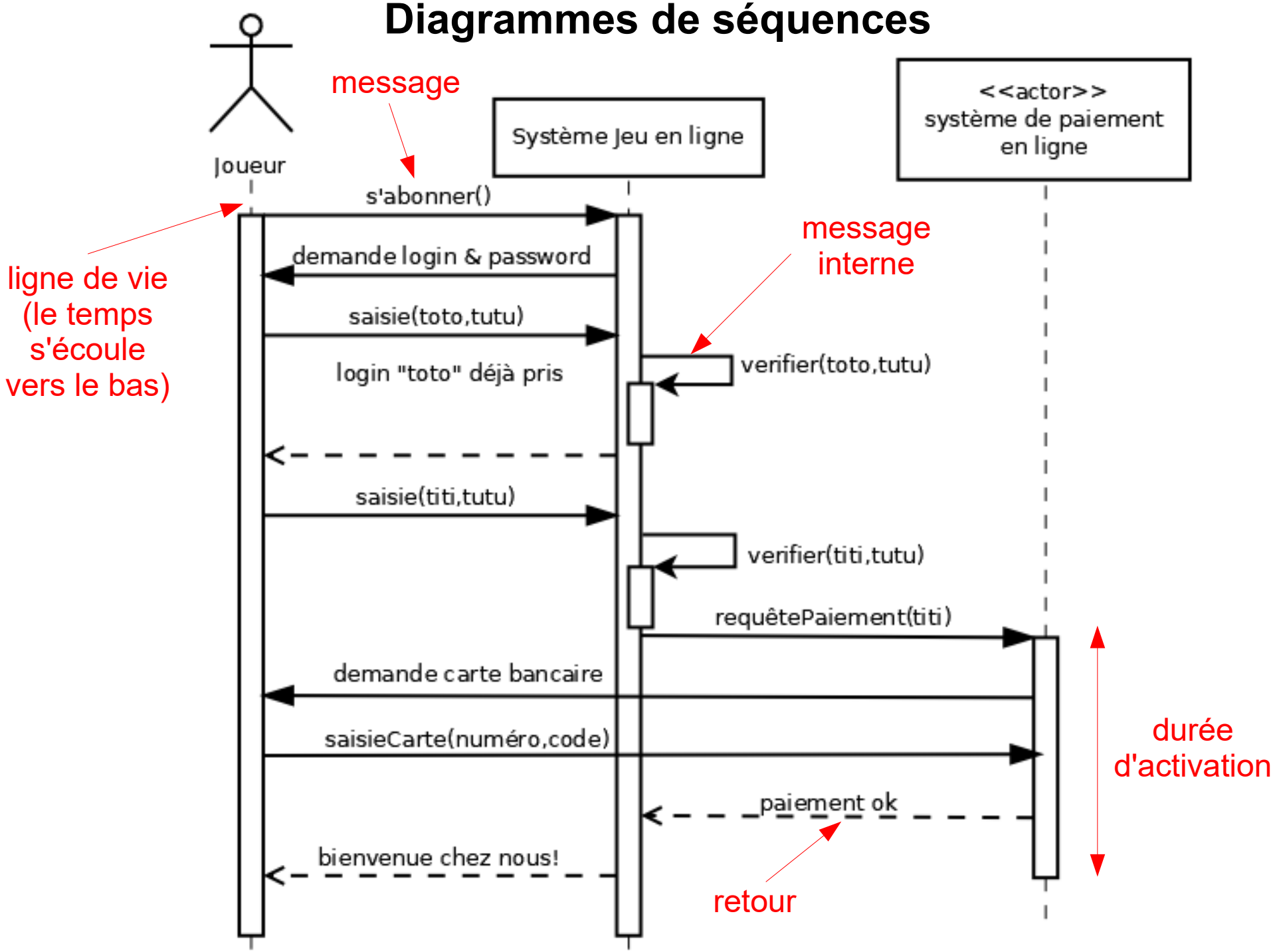
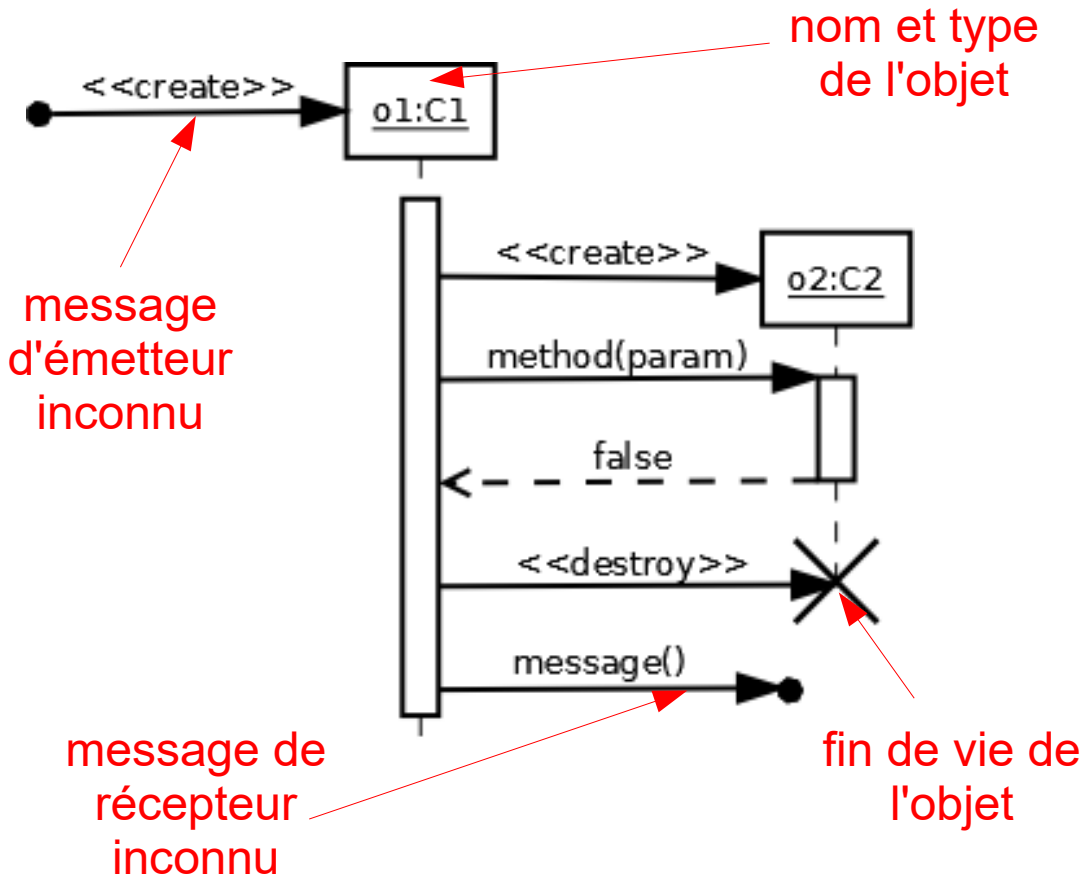


Diagramme de séquences : messages

Les messages échangés correspondent à des communications entre objets.

Création et destruction d'objet :



Message réflexif :

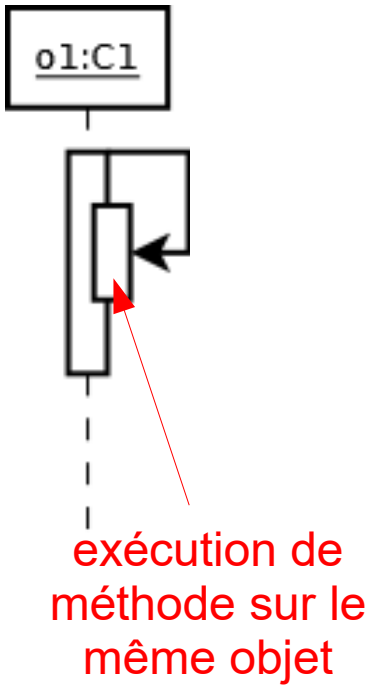
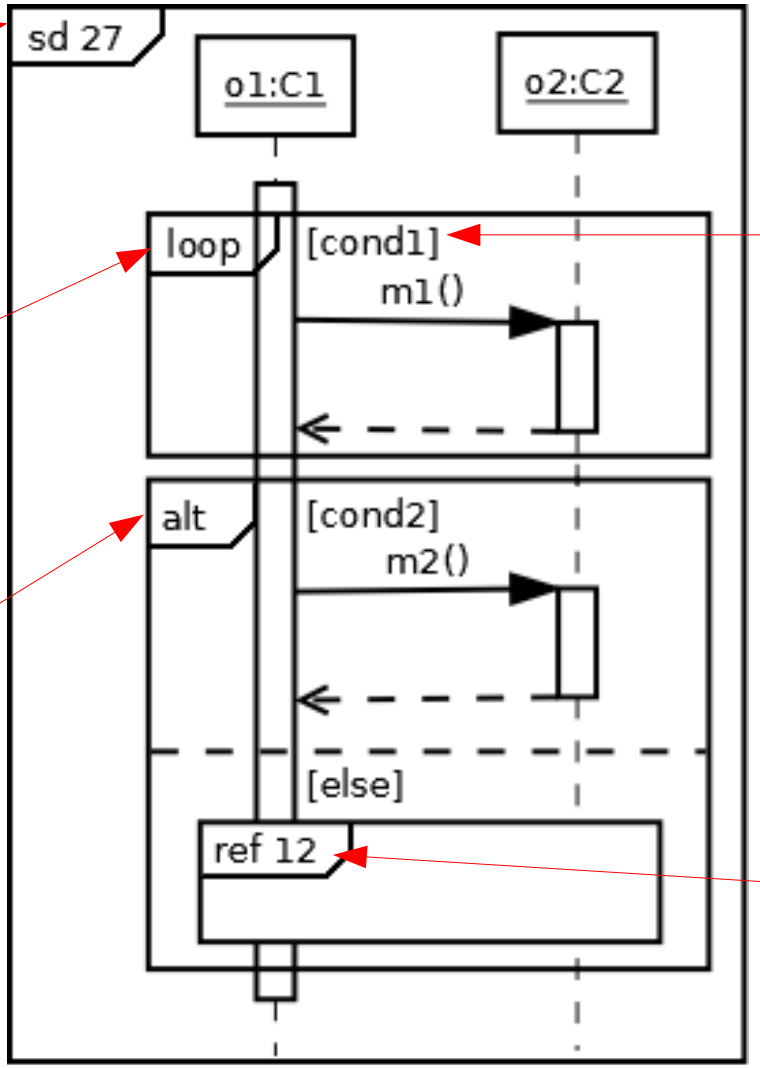


Diagramme de séquences : cadres



cadre contenant un diagramme de séquence

cadre contenant une boucle

cadre contenant une conditionnelle

condition de boucle

interaction décrite dans un autre diagramme

Diagramme de séquences : contraintes temporelles

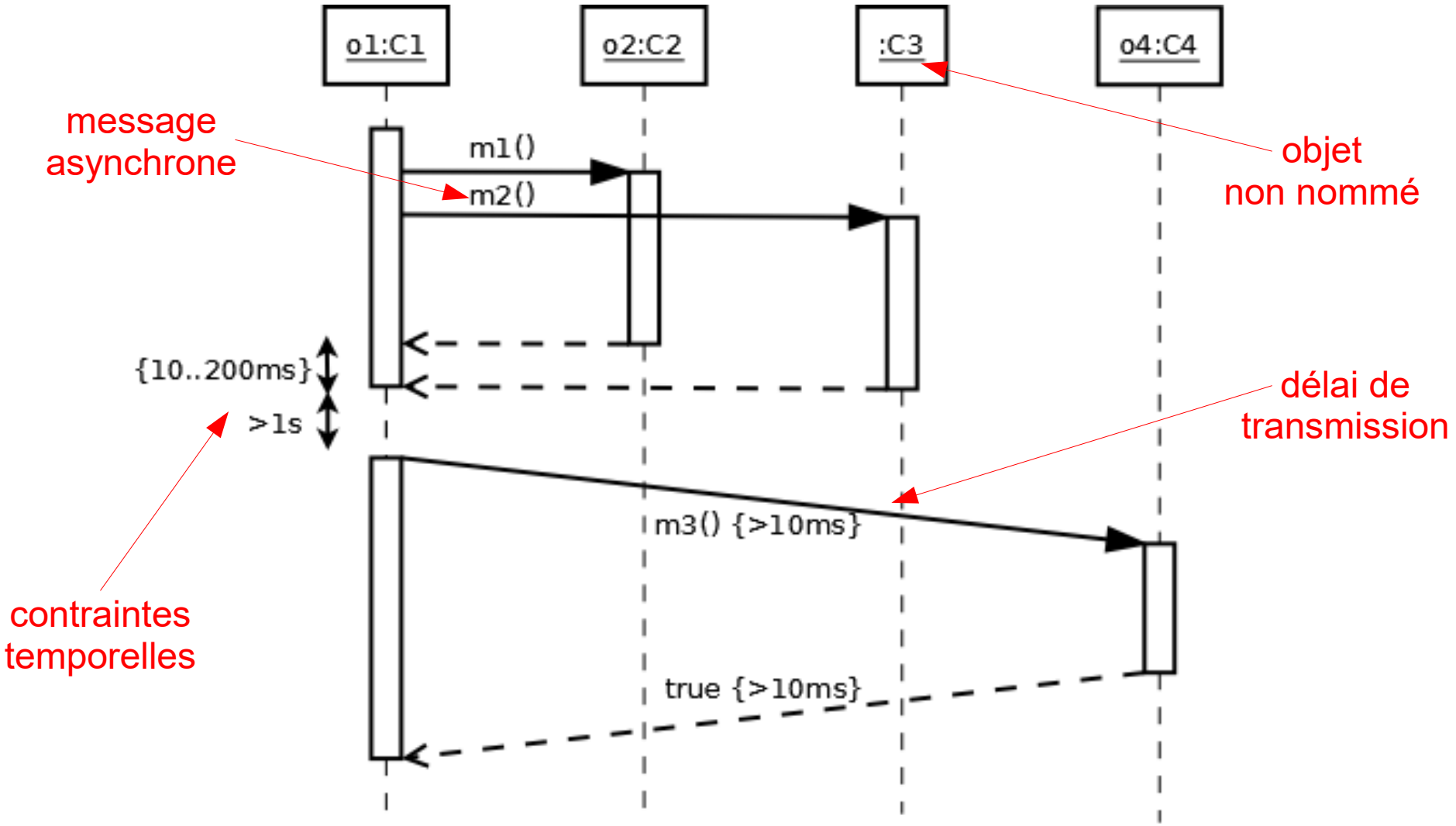


Diagramme de séquences : usages (1/2)

Dans un modèle d'analyse, détailler un cas d'utilisation dans un diagramme de séquence sert à mettre en évidence les éléments (classes, méthodes, etc) nécessaires à la réalisation des fonctionnalités.

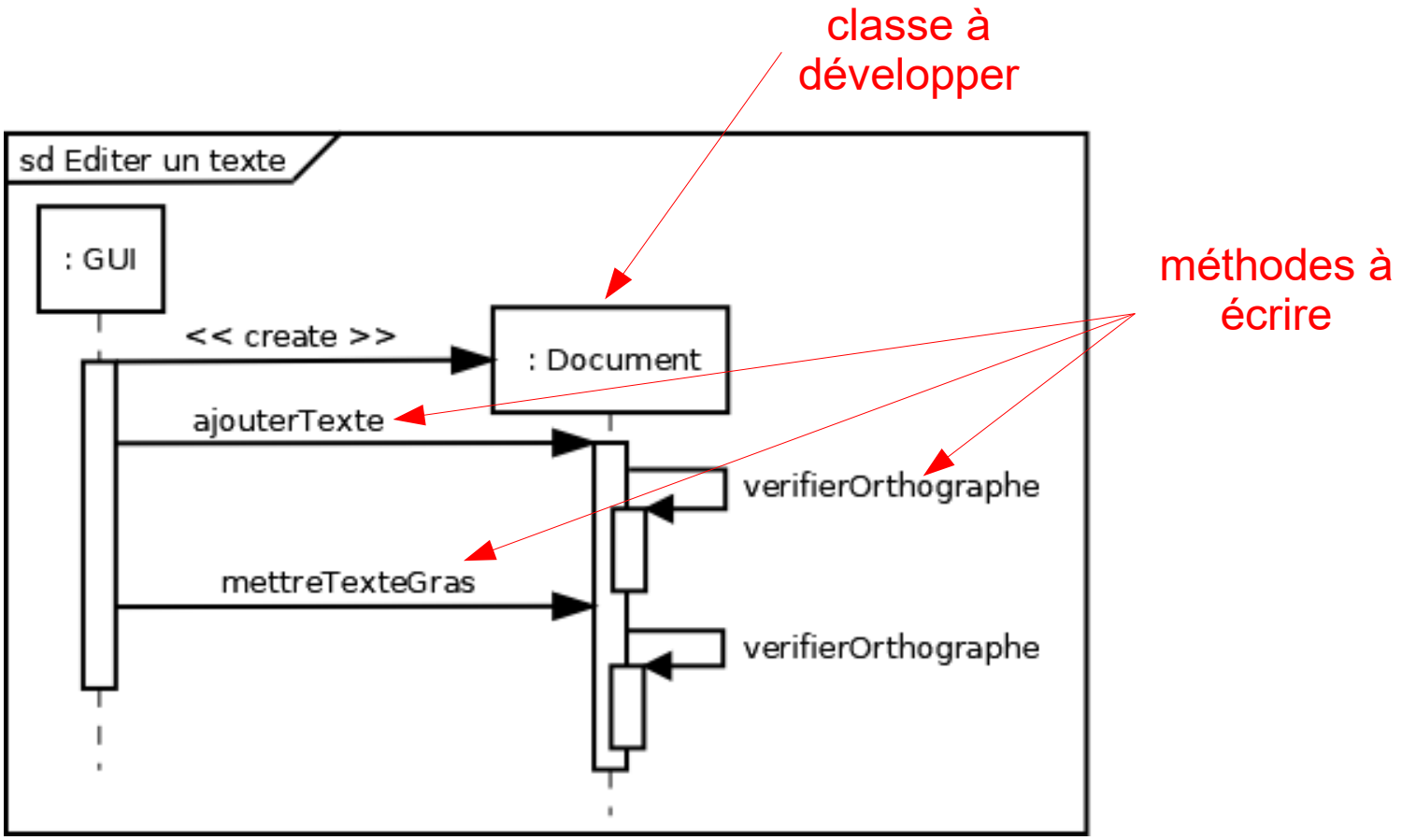
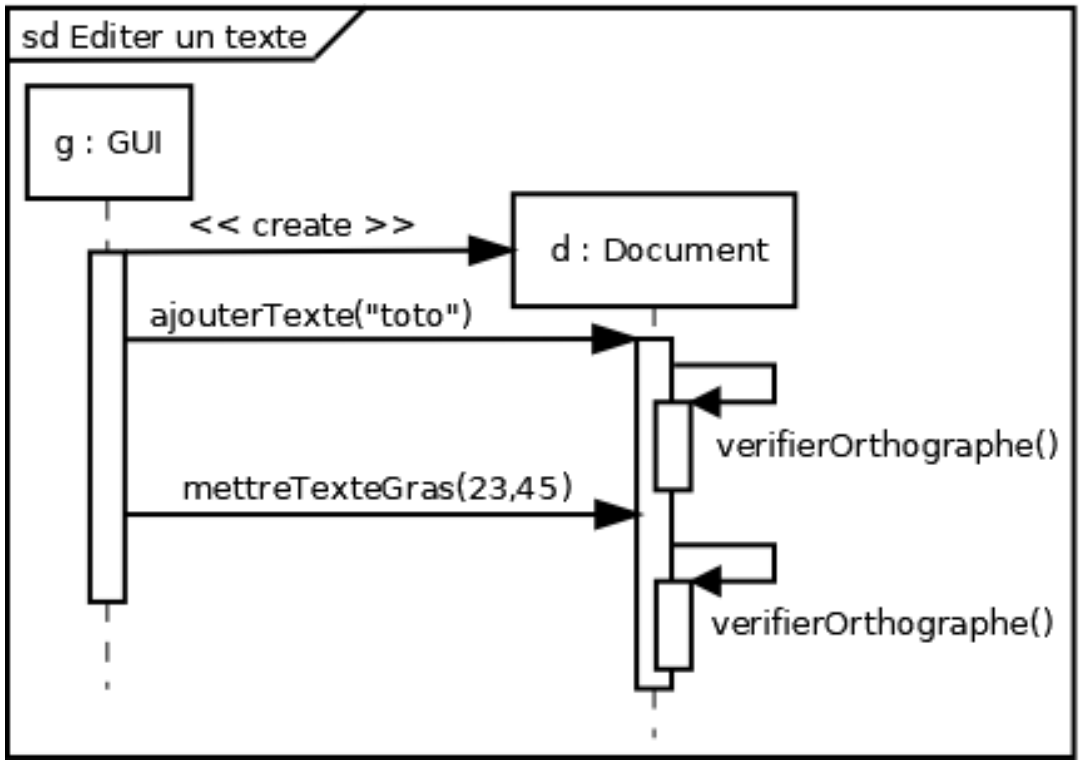


Diagramme de séquences : usages (1/2)

Dans un modèle de conception, un diagramme de séquence doit être suffisamment détaillé pour permettre le codage voire la génération automatique de code.



Dans la classe GUI :

```
d = new Document();
d.ajouterTexte(''toto'');
```

Dans la classe Document :

```
… ajouterTexte(String s){
… this.vérifierOrthographe(); …
}

… mettreTexteGras(int debut,
int fin){
… this.vérifierOrthographe(); …
}
```

Diagramme de communication (1/2)

Un diagramme de communication décrit un scénario de façon plus structurelle qu'un diagramme de séquence : les acteurs échangent des messages numérotés selon l'ordre du scénario.

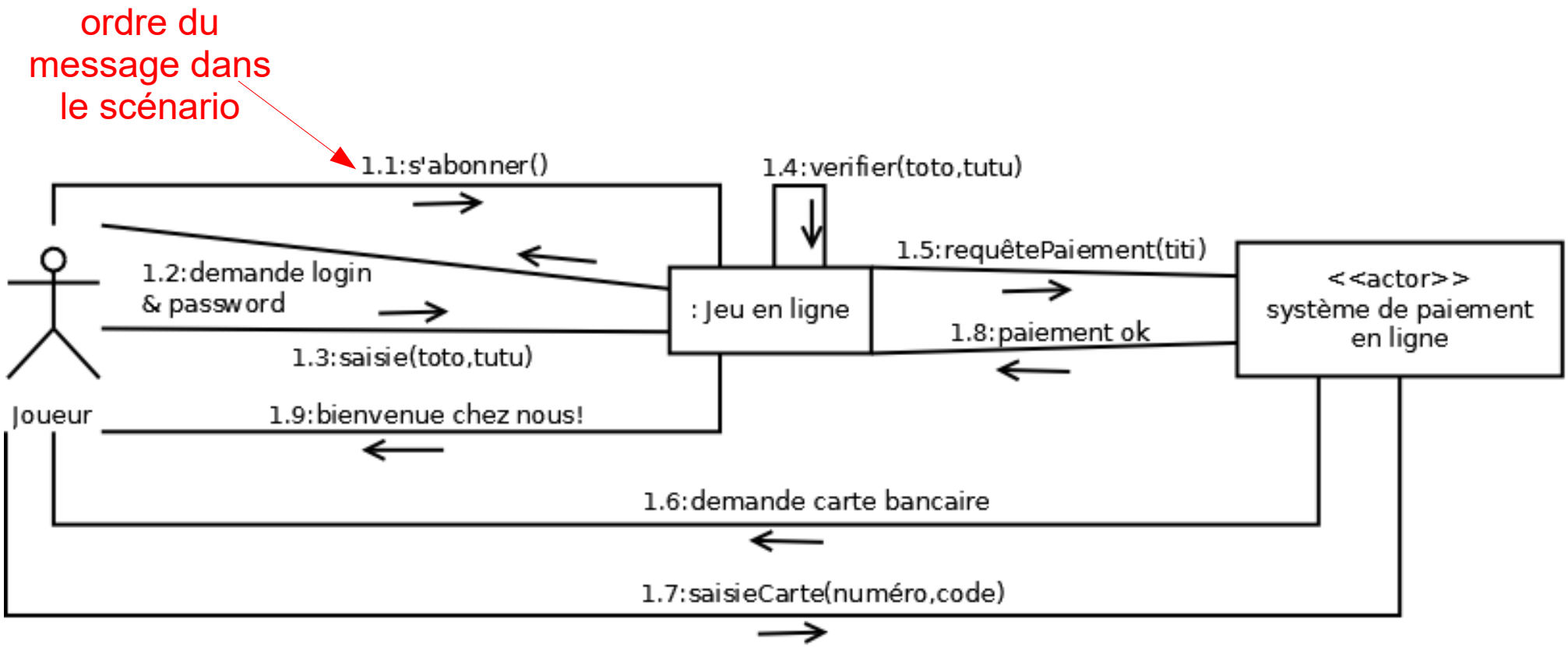


Diagramme de communication (2/2)

L'envoi de message peut se faire de façon conditionnelle ou répétitive :

1.2 [c>2] : m() signifie que lors de l'étape 1, le message m() est envoyé si c>2

1.2 *:m() signifie que lors de l'étape 1.2, le message m() est envoyé un nombre indéterminé de fois

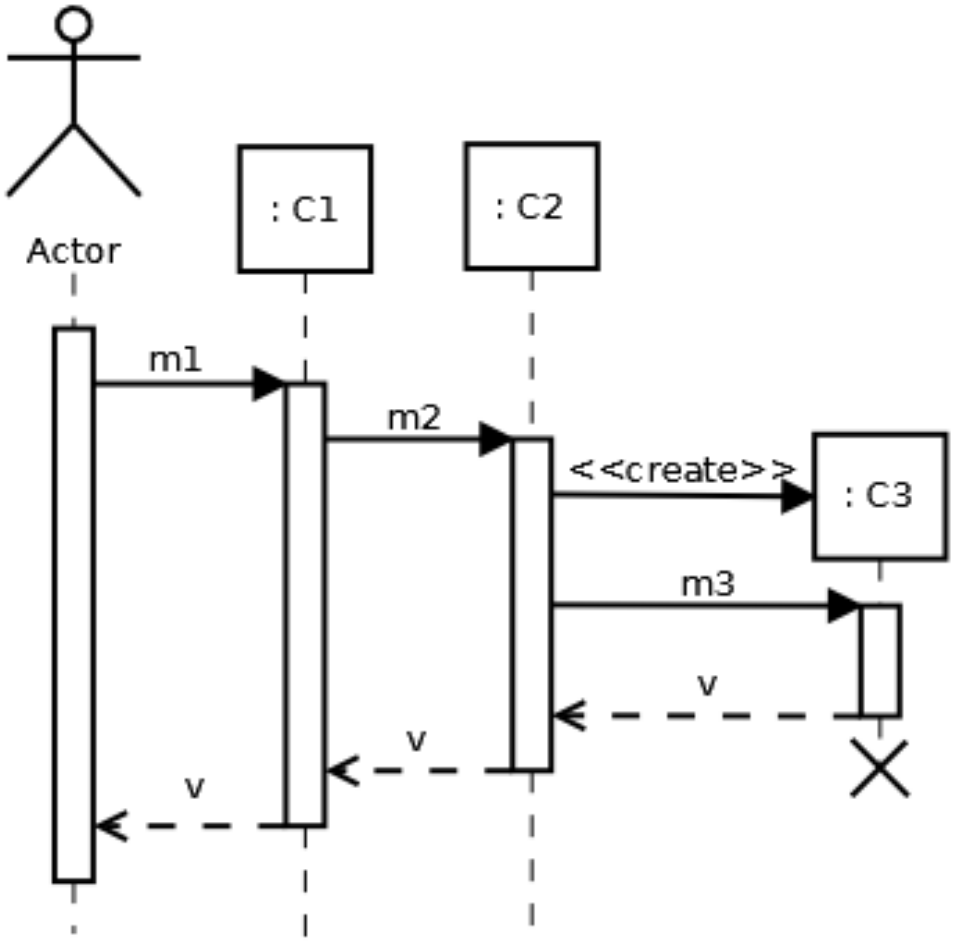
1.2 *[1..5]: m() signifie que lors de l'étape 1.2, le message m() est envoyé 5 fois

1.2 *[i=1..12]: m(i) signifie que lors de l'étape 1.2, le message m() est envoyé 12 fois avec les valeurs 1 à 12

1.2 m() [:27] signifie que lors de l'étape 1.2, le message m() renverra la valeur 27

Diagramme de communication : usages (1/2)

Un diagramme de communication donne une vue spatiale d'un scénario, là où un diagramme de séquence en donne une vue temporelle.



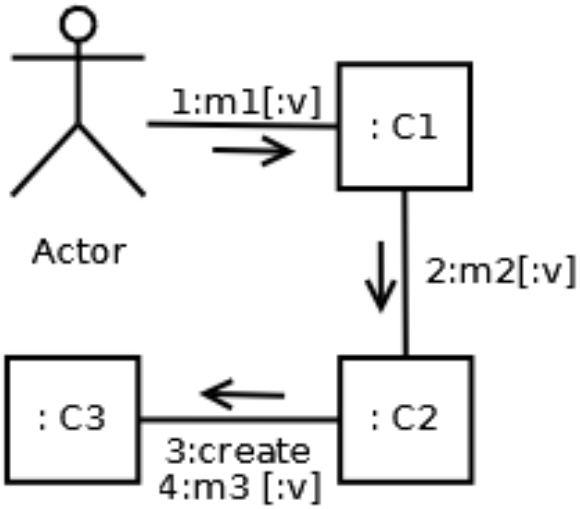


Diagramme de communication : usages (2/2)

Par rapport à un diagramme de séquences, un diagramme de communication :

- est plus simple à réaliser, surtout s'il y a beaucoup d'objets impliqués

- est bien plus simple à maintenir

- supporte facilement le parallélisme

Le choix du type de diagramme dépend du scénario à représenter et doit privilégier la lisibilité.