

Démarche d'analyse et de conception avec le langage UML

Je tiens à remercier mon chat et mon poisson rouge (jusqu'à ce que mon chat ne le mange) pour l'aide et le soutien moral qu'ils m'ont apportés.

Index

<i>Démarche d'analyse et de conception avec le langage UML</i>	<i>1</i>
<i>Index</i>	<i>2</i>
<i>Préambule</i>	<i>3</i>
<i>Objectif global d'une démarche d'analyse et de conception objet</i>	<i>4</i>
<i>Les étapes de la démarche</i>	<i>5</i>
<i>Exemple traité</i>	<i>11</i>
<i>Définition des besoins</i>	<i>12</i>
première étape : les exigences et les acteurs	12
deuxième étape : les intentions d'acteurs	16
troisième étape : le diagramme de use case	17
quatrième étape : use case description de haut niveau	20
<i>L'analyse</i>	<i>21</i>
cinquième étape : use case description de bas niveau	21
septième étape : diagramme de classe d'analyse	29
huitième étape : Le glossaire	33
neuvième étape : les contrats d'opération	34
<i>La conception</i>	<i>40</i>
dixième étape : le diagramme d'état	40
onzième étape : le diagramme de collaboration	42
1) Syntaxe du diagramme de collaboration	42
2) Visibilité des objets	49
3) GRASP patterns	51
4) Diagramme de collaboration du magasin	57
douzième étape : le diagramme de classe de conception	61
1) Première étape	61
2) Deuxième étape	62
<i>Le processus unifié</i>	<i>69</i>
Notion de lot (ou de cycle)	69
Notion de phases	69
Notion d'itération	71
<i>conclusion</i>	<i>73</i>
<i>bibliographie</i>	<i>74</i>

Préambule

Cette démarche de conception est la démarche proposée par Craig LARMAN. Je l'ai connue à travers le cours "Analyse et conception avec UML et les Patterns " de la société Valtech. J'ai ici repris cette démarche, je l'ai légèrement simplifiée, et j'ai apporté les informations nécessaires pour que des stagiaires de l'AFPA puissent s'imprégner de cette démarche. Dans un premier temps j'ai repris l'exercice du cours Valtech. Souhaitons qu'il y ait un deuxième temps ...

Les formateurs qui désirent former leurs stagiaires à cette démarche devraient suivre le cours pré cité, afin de prendre du recul par rapport à la démarche.

Je recommande à tous la lecture des ouvrages de Craig LARMAN sur UML.

Objectif global d'une démarche d'analyse et de conception objet

Le but de cette démarche est d'analyser et de concevoir une application objet. L'objectif est d'avoir une application qui puisse vivre (évolution de l'application dans le temps), et qui enrichisse la base de savoir-faire de l'entreprise (développement de nouveaux applicatifs en s'appuyant sur des objets métiers déjà développés). La réutilisabilité est un des soucis principaux pour pouvoir réduire le coût des logiciels et augmenter la puissance de développement des programmeurs.

Nous nous appuierons sur le workflow, c'est à dire les règles des processus métier pour débusquer les uses cases, et les acteurs. Dans l'analyse et la conception objet nous cherchons à construire une architecture en couche de la forme suivante :

couche I.H.M.	présentation
couche application (workflow)	application
couche objets métiers	métier
couche accès aux données	accès aux données
couche base de données	base de données

Nous verrons que chaque couche définira un ou des contrôleurs pour la rendre indépendante des autres.

Une étude a montré que dans certaines entreprises les règles métier changeaient de 10% par mois !!! Il est facile de comprendre alors la raison de ce découplage des objets métiers. Les objets ont tendance à être stables dans le temps, par contre les IHM et base de données peuvent également être changées sans que l'on touche au métier ni aux règles de l'entreprise.

Cette architecture ne s'appliquera bien sûr pas à toutes les applications, c'est à prendre comme un exemple complet.

Les étapes de la démarche

Ce chapitre est un résumé du processus de développement d'une application. Il est difficile de comprendre ce résumé, sans avoir lu en détail et expérimenté chacun des chapitres auquel il fait référence. Par contre je vous conseille de revenir souvent à ce résumé (textuel et graphique) pour bien vous situer dans la démarche, avant l'étude de tout nouveau chapitre.

- ? Lister l'ensemble des **exigences** du client issues du cahier des charges, ou issu d'une démarche préalable de collecte d'information (documents électroniques, notes de réunions, notes d'interviews, ...). Chaque exigence sera numérotée et ainsi pourra être tracée.
- ? Deux solutions possibles :
 - ? Nous allons regrouper les exigences par **intentions d'acteur** complètes puis nous allons faire un diagramme de contexte (nous nous appuierons sur un diagramme de collaboration pour cela)
 - ? Si toutes les règles de processus métier sont définies, nous réaliserons un diagramme d'activité en colonnes ("swim lane") où les colonnes sont les acteurs. Cela permet de dégager les responsabilités de chaque acteur, et ainsi de connaître les intentions des acteurs.
- ? Définir les uses cases et construire le diagramme de **uses cases**.
- ? Faire la **description de haut niveau** de chaque use case : chercher des scénarios de base.
- ? Faire la **description détaillée** de chaque use case : donner les séquences nominales puis les séquences alternatives (Erreurs et exceptions, en limitant au maximum les exceptions). Cette description peut être complétée par un diagramme d'activité qui montre le séquençement des différents scénarios.

Cette description détaillée comprend les scénarios, les pré conditions, à partir de quoi se déroule le use case, comment se termine le use case, et enfin les post conditions (règles métier assurées quand le use case est terminé).
- ? Faire un diagramme de séquence par scénario (ici c'est un diagramme de séquence boîte noire, où la boîte noire correspond au système informatique à développer).
- ? Faire un diagramme de classe par use case. Le diagramme de classe final sera la superposition de ces diagrammes de classe.

Les classes obtenues sont des classes du monde réel.
Nous ne mettons pas les opérations car nous ne connaissons pas l'intérieur du système informatique.
- ? Un contrat est réalisé pour chaque opération du système. Ce contrat contient un nom, des responsabilités, les exigences auxquelles répond notre itération de cycle de vie, les pré conditions, et les post conditions (décrites sous la forme " has been ") et enfin les exceptions.

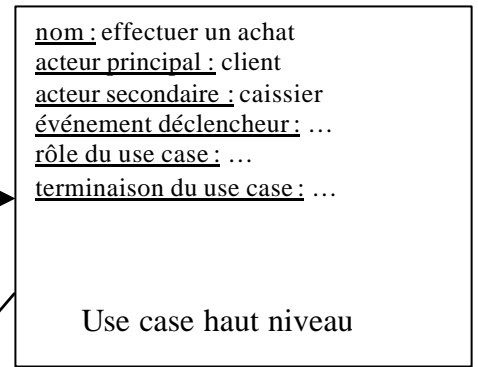
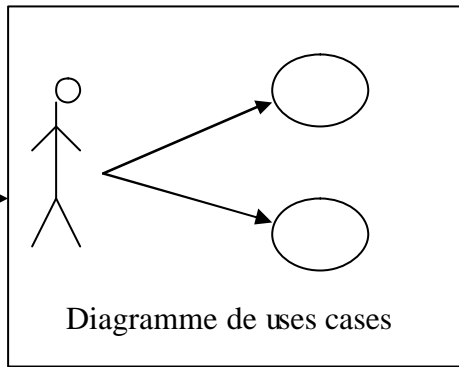
Ce contrat peut être réalisé sous forme textuelle, ou plus souvent sous forme d'un diagramme de séquence boîte blanche où les objets échangent des messages pour rendre le service demandé.
- ? A partir des diagrammes de classe et des contrats nous réaliserons les diagrammes de collaboration qui montrent comment les objets collaborent pour rendre le service demandé. Nous appliquerons les patterns de conception pour cela (GRASP patterns)
- ? En parallèle nous réaliserons les diagrammes d'état des objets les plus complexes, ainsi nous détecterons des méthodes internes à ces objets.

- ? Nous réaliserons enfin le diagramme de classe de conception, en tenant compte à nouveau des GRASP patterns. Ceci peut remettre en cause le diagramme de classe précédemment établi.

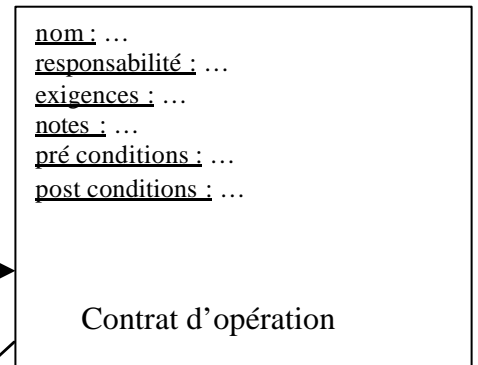
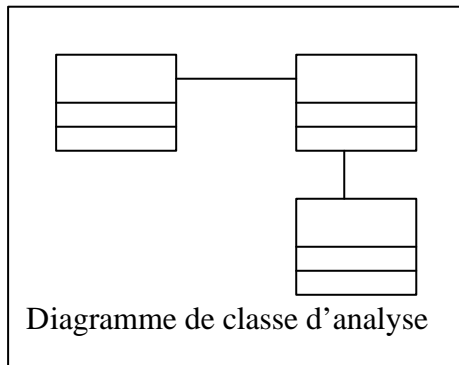
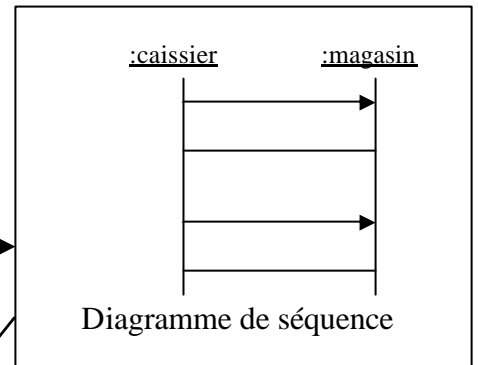
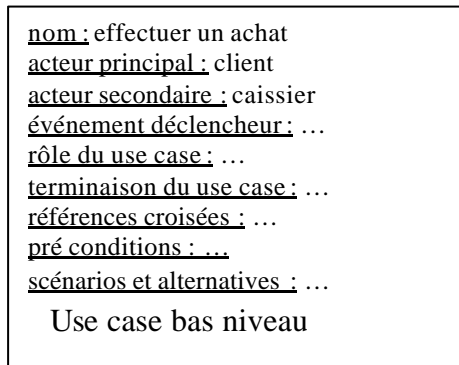
Processus simplifié

D
E
F
I
N
I
R

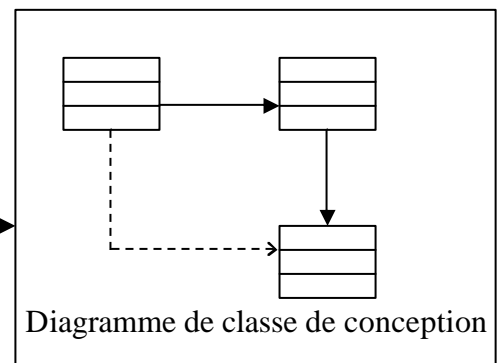
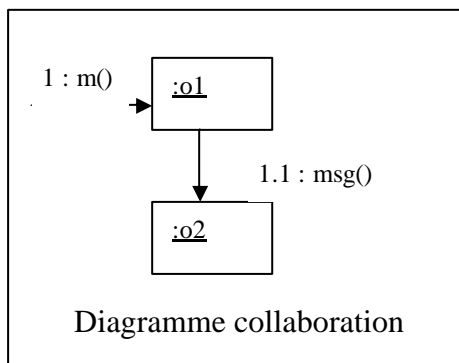
B
E
S
O
I
N
S



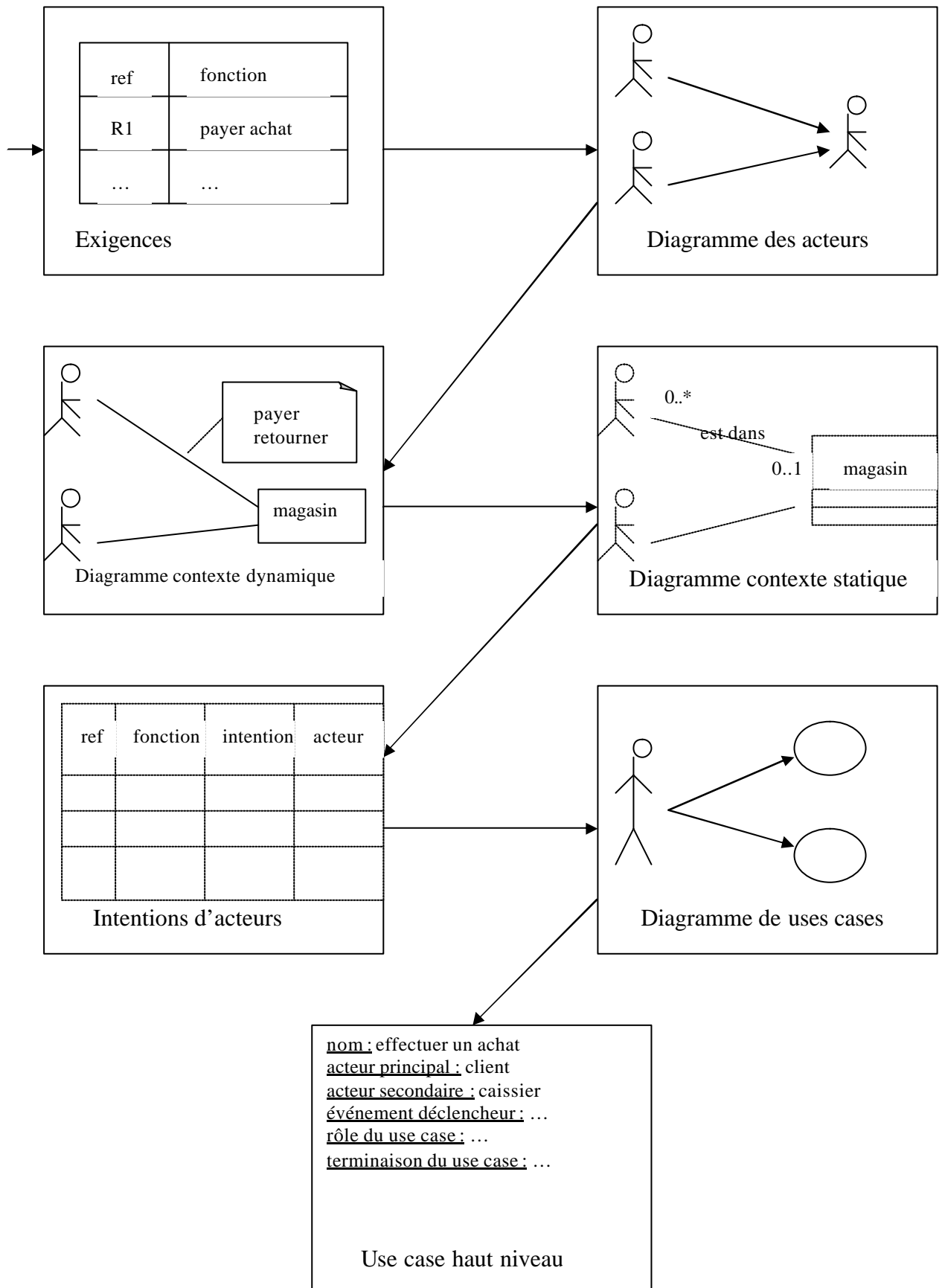
A
N
A
L
Y
S
E



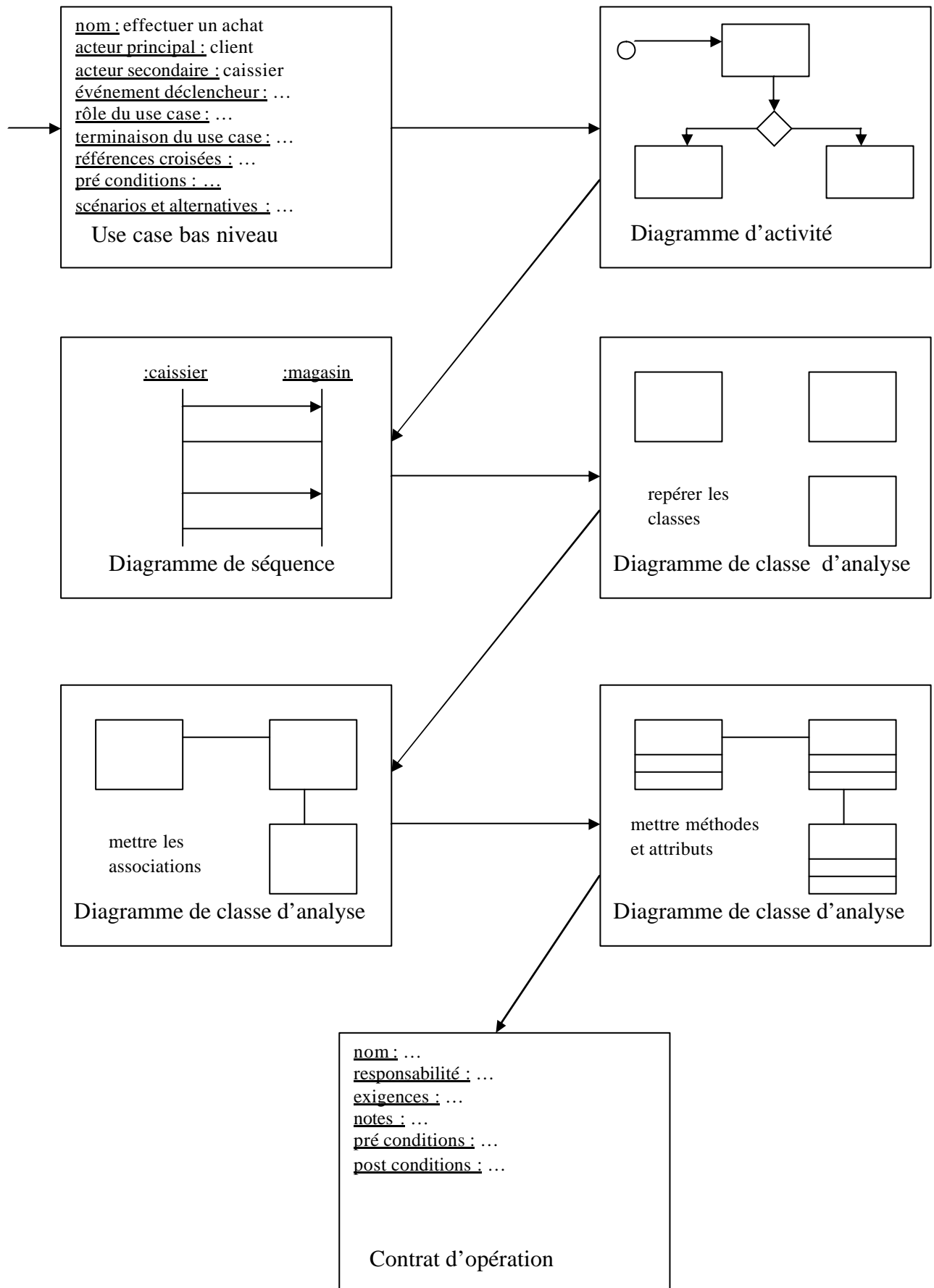
C
O
N
C
E
P
T
I
O
N



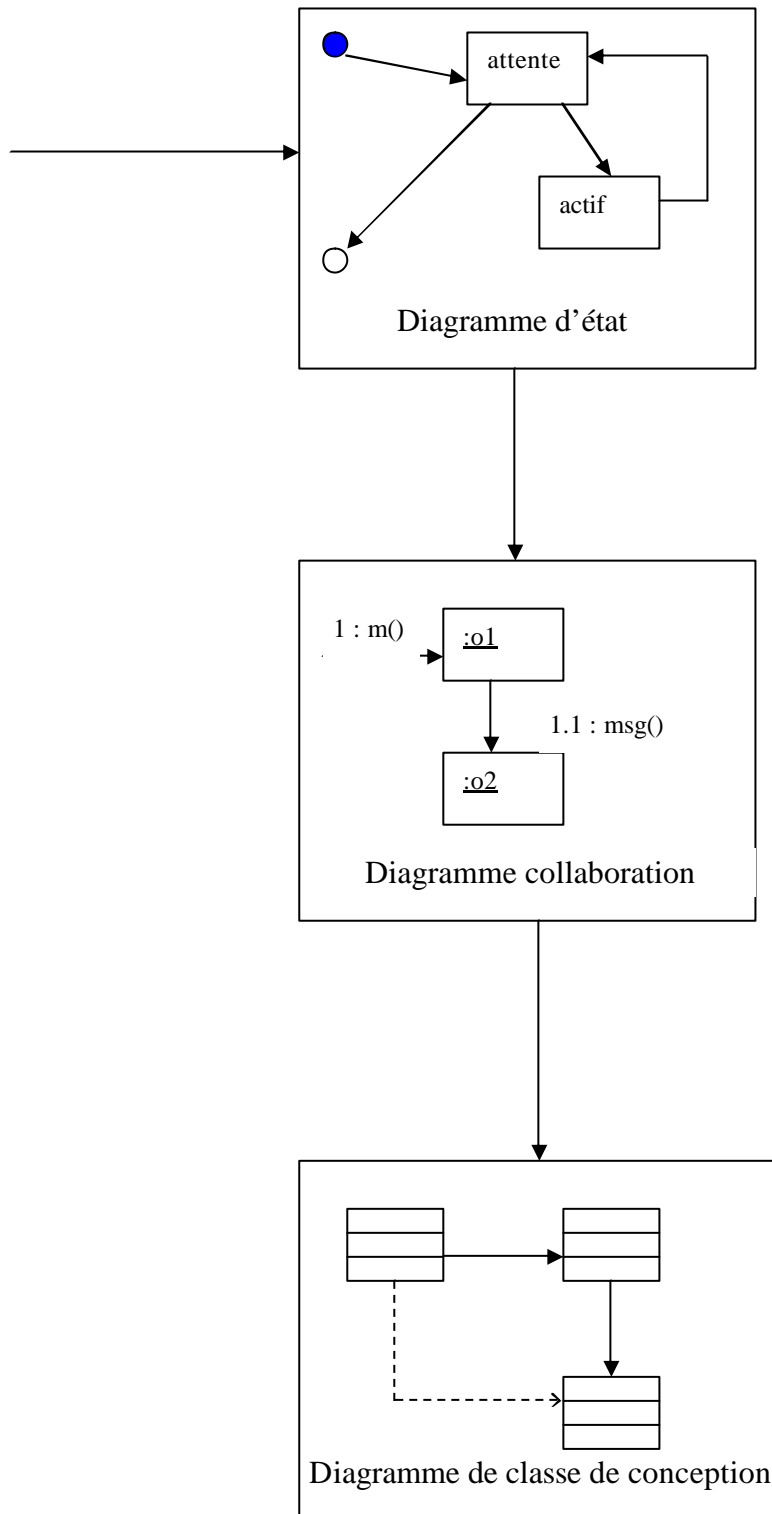
Processus de définition des besoins



Processus d'analyse



Processus de conception



Exemple traité

Nous allons traiter l'ensemble de la démarche d'analyse objet sur un même exemple. Cet exemple a été pris dans la littérature (voir la bibliographie). Nous ne traiterons pas l'ensemble de l'exemple, mais l'ensemble de la démarche.

Réalisation d'une caisse informatisée.

Un commerçant de produits touristiques (souvenirs, livres régionaux, ...) désire informatiser sa caisse. Chaque type de produit possède un code unique (étiquette à code à barres), et un même prix pour tous les produits de ce type. L'objectif est de faciliter la maintenance des prix des articles.

Chaque type de produit est référencé dans un catalogue, avec son prix associé. Quand le prix d'un produit doit être modifié, le manager modifie son prix dans le catalogue, puis sur l'étagère où il est rangé.

Le caissier s'identifie pour démarrer la caisse (avec mot de passe).

La caisse fera les fonctions habituelles d'une caisse : calcul du sous total, calcul du total, possibilité de rentrer plusieurs articles ayant un même code, retour d'une marchandise avec le ticket de caisse. Le paiement se fera en monnaie seulement.

La caisse permet d'éditer des rapports :

- ? Le reçu qui sera donné uniquement pour une vente effective. Il contient le nom du magasin, un message de bienvenue, la date et l'heure. Puis pour chaque vente il donne le code du produit, la description du produit, le prix unitaire, la quantité et le sous total. Enfin nous y trouvons le total TTC.
- ? Le rapport quotidien de l'ensemble des ventes (date, heure, total).
- ? Le rapport quotidien détaillé: liste de l'ensemble détaillé des ventes de la journée.

La caisse s'exécute sur un PC. Une douchette permettra de lire les codes à barres. Les informations peuvent être rentrées au clavier, ou à la souris.

Définition des besoins

Pour bien comprendre le fonctionnement du logiciel, et son interaction avec son environnement, nous avons intérêt à travailler non pas sur le logiciel demandé, mais sur le fonctionnement intégral du processus d'entreprise (workflow). Ainsi nous aurons une meilleure approche du logiciel. Ainsi nous considérerons l'ensemble des acteurs qui interagissent sur le processus d'entreprise, même s'ils n'agissent pas sur le logiciel lui-même. Cela permettra de déterminer les intentions d'acteurs qui nous donneront les uses cases.

La définition des besoins démarre avec le début du projet. Elle a pour but d'obtenir un premier jet des besoins fonctionnels et opérationnels du client. Dans cette phase nous collectons des informations pour définir le besoin du client.

A l'issue de cette étape, nous aurons mis textuellement ou à l'aide de schémas très simples, notre compréhension du problème à traiter sur le papier.

Le client doit être capable de valider l'étude ainsi faite. Elle lui est donc accessible.

première étape : les exigences et les acteurs

Les exigences que nous allons découvrir sont les exigences fonctionnelles. A partir du problème posé, c'est à dire de tout ce qui est écrit, plus tout ce qui ne l'est pas, nous allons lister l'ensemble des fonctions qui pourront être réalisées par le logiciel. Ces exigences seront numérotées pour pouvoir les tracer dans les intentions d'acteur puis dans les uses cases.

Référence	Fonction
R1	Modifier le prix d'un produit
R2	Calculer le total d'une vente
R3	Rentrer une quantité par article
R4	Calculer le sous total d'un produit
R5	Retourner une marchandise
R6	Payer l'achat
R7	Editer un ticket de vente
R8	Editer un rapport succinct
R9	Editer un rapport détaillé
R10	Se connecter à la caisse
R11	Se connecter en gérant
R12	Définir les droits des usagers
R13	Entrer un produit au catalogue
R14	Supprimer un produit du catalogue
R15	Enregistrer un produit à la caisse
R16	Initialiser la caisse

Dans la référence R veut dire Requirement (c'est à dire exigence en Anglais).

La modification du prix sur une étagère n'est pas traitée par le système. Donc ce n'est pas une exigence fonctionnelle pour notre logiciel.

Se connecter en gérant permet de modifier les prix des articles. Ce n'est pas permis pour un caissier.

Définir les droits des usagers n'est pas indiqué dans le texte, mais est nécessaire au bon fonctionnement du logiciel.

Le PC ainsi que la douchette sont des exigences d'architecture, elles ne seront pas prises en compte ici.

L'initialisation de la caisse est une fonctionnalité masquée.

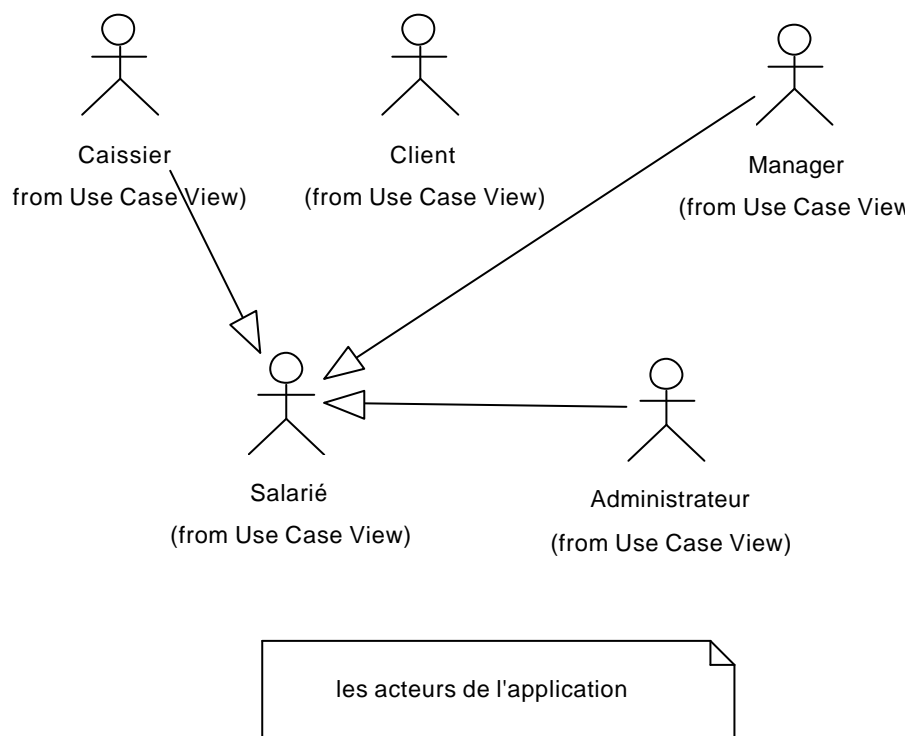
Entrer et supprimer un produit du catalogue sont des fonctions tellement évidentes que le client n'en parle pas. Elles doivent cependant être intégrées au logiciel.

Les informations rentrées au clavier ou à la souris sont des exigences d'interface qui seront prises en compte ultérieurement.

Notons que les exigences ne sont pas classées ici.

Regardons maintenant les acteurs qui gravitent autour de notre application.

Les acteurs seront vus dans le sens processus transversal de l'entreprise (workflow). Préférer un acteur humain plutôt que le dispositif électronique devant lequel il est. L'acteur humain a bien une intention quand il fait quelque chose. Un dispositif électronique beaucoup moins en général.



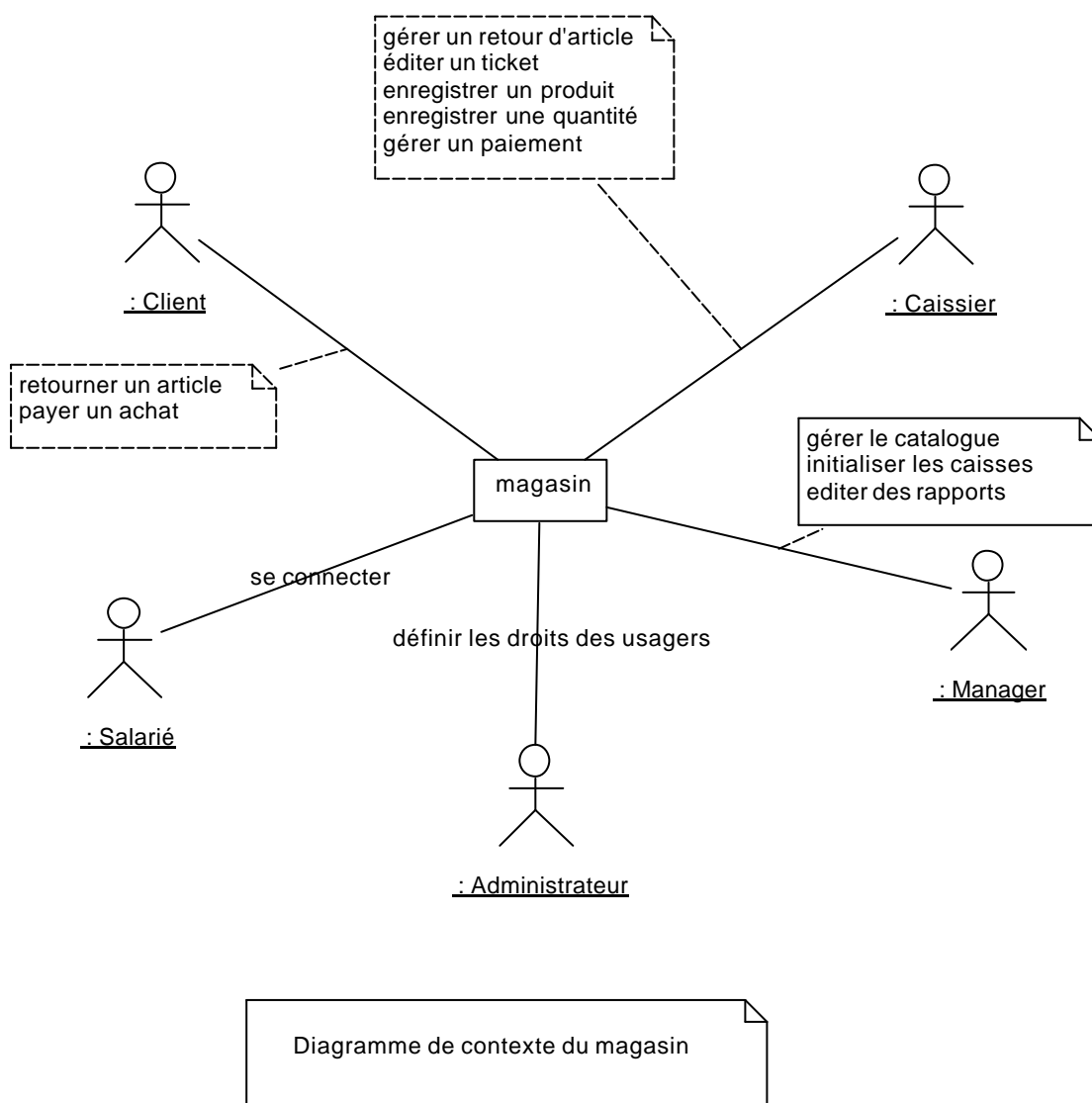
Ce diagramme est un diagramme de classe qui permet de lister les différents acteurs. Il se peut que notre liste ne soit pas complète, une itération suivante du processus nous permettra de compléter ce schéma.

Nous avons défini 5 acteurs. N'oublions pas qu'un acteur représente un rôle, et non pas une personne.

Le Manager peut être aussi l'Administrateur (notion de rôle).

Un Salarié est soit un Caissier, soit le Manager. Il se peut que dans la première itération l'analyste ne remarque pas ceci. C'est souvent dans une itération ultérieure que nous verrons les généralisations d'acteurs.

Nous connaissons les acteurs, et les exigences de l'application. Nous pouvons donc faire un diagramme de contexte dynamique. Ce diagramme de contexte qui définit les interactions entre les acteurs et le système (pas encore système informatique car nous en sommes au processus métier donc à découvrir les uses cases) sera réalisé en UML par un diagramme de collaboration. Il représente le fonctionnement de l'entreprise. Ici nous ne précisons pas forcément l'ordre et le séquençement des opérations.



Nous pouvons aussi en complément tracer un diagramme de classe des acteurs du système pour préciser les cardinalités des différents acteurs du système. Nous utiliserons alors un diagramme de classes de UML. Ce diagramme s'appelle un diagramme de contexte statique.

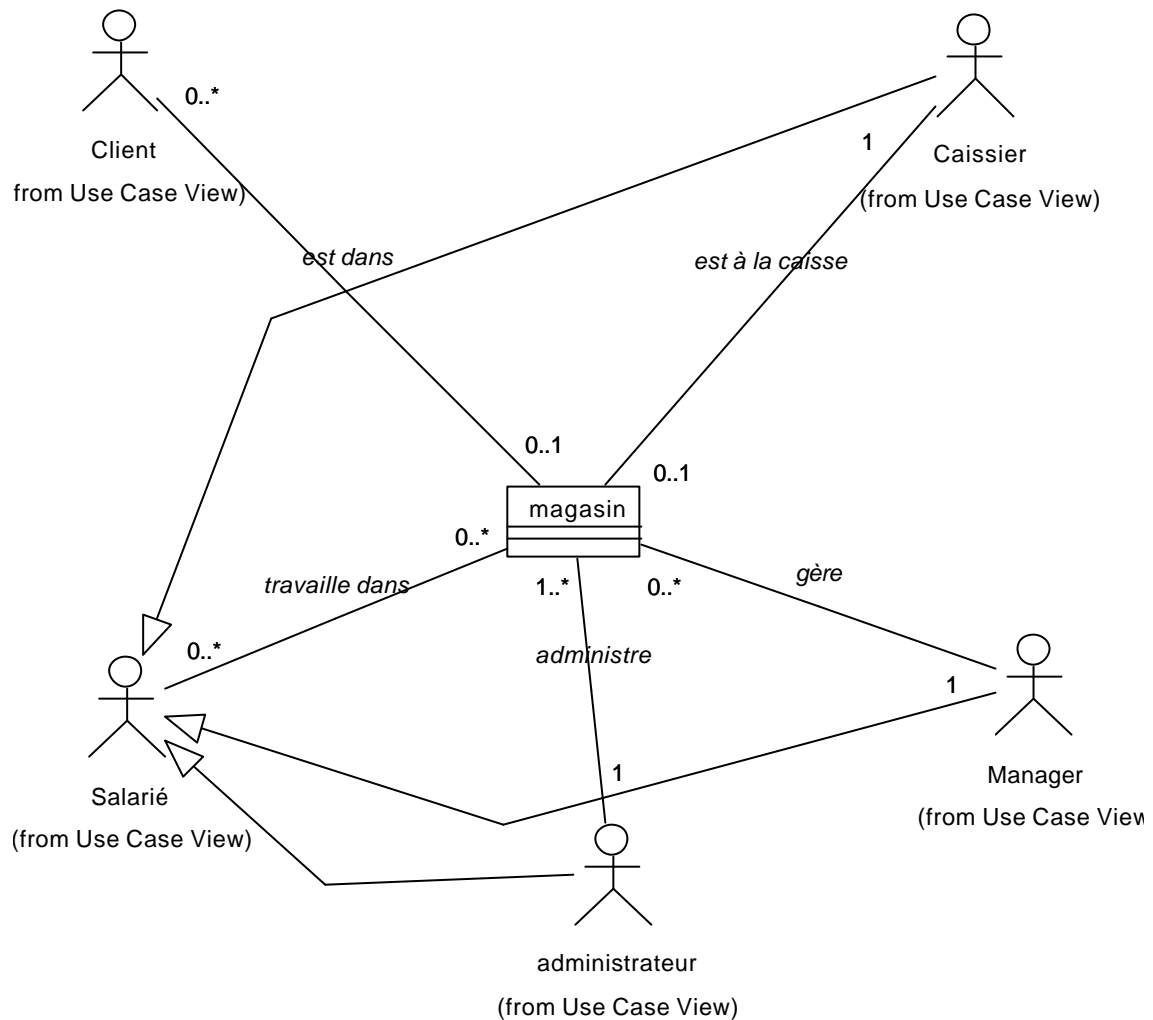


Diagramme de contexte statique (capture initiale des besoins)

deuxième étape : les intentions d'acteurs

Nous allons maintenant regrouper les exigences en intentions d'acteurs. Une intention d'acteur est un enchaînement de fonctions qui rend un service complet à un acteur (et pas une fraction de service).

Référence	Fonction	Intention d'acteur	Acteurs
R1	Modifier le prix d'un produit	Gérer le catalogue	<u>Manager</u>
R13	Entrer un produit au catalogue	Gérer le catalogue	
R14	Supprimer un produit du catalogue	Gérer le catalogue	
R16	Initialiser la caisse	Initialiser la caisse	<u>Manager</u>
R5	Retourner une marchandise	Retourner un article	<u>Client</u> , Caissier
R10	Se connecter à la caisse	Se connecter	<u>Salarié</u>
R11	Se connecter en gérant	Se connecter	
R8	Editer un rapport succinct	Editer un rapport	<u>Manager</u>
R9	Editer un rapport détaillé	Editer un rapport	
R12	Définir les droits des usagers	Définir les profils	<u>Administrateur</u>
R7	Editer un ticket de vente	Effectuer un achat	<u>Client</u> , Caissier
R6	Payer l'achat	Effectuer un achat	
R2	Calculer le total d'une vente	Effectuer un achat	
R3	Rentrer une quantité par article	Effectuer un achat	
R15	Enregistrer un produit à la caisse	Effectuer un achat	
R4	Calculer le sous total d'un produit	Effectuer un achat	

Ici dans le tableau la notion d'acteur repose sur l'intention d'acteur, pas sur la fonction. Ainsi toutes les lignes d'une même intention d'acteur ne sont-elles pas renseignées pour les acteurs.

Ici nous distinguerons l'acteur principal des acteurs secondaires en soulignant l'acteur qui est à l'origine de l'intention.

Nous allons bien sûr vérifier que les exigences définies précédemment sont toutes incluses dans les intentions d'acteur. La traçabilité est un facteur essentiel des phases de définition des besoins et de celle d'analyse.

Nous avons les intentions d'acteurs, nous pouvons donc faire un diagramme de Use Case.

Toute cette démarche peut être faite d'une autre manière:

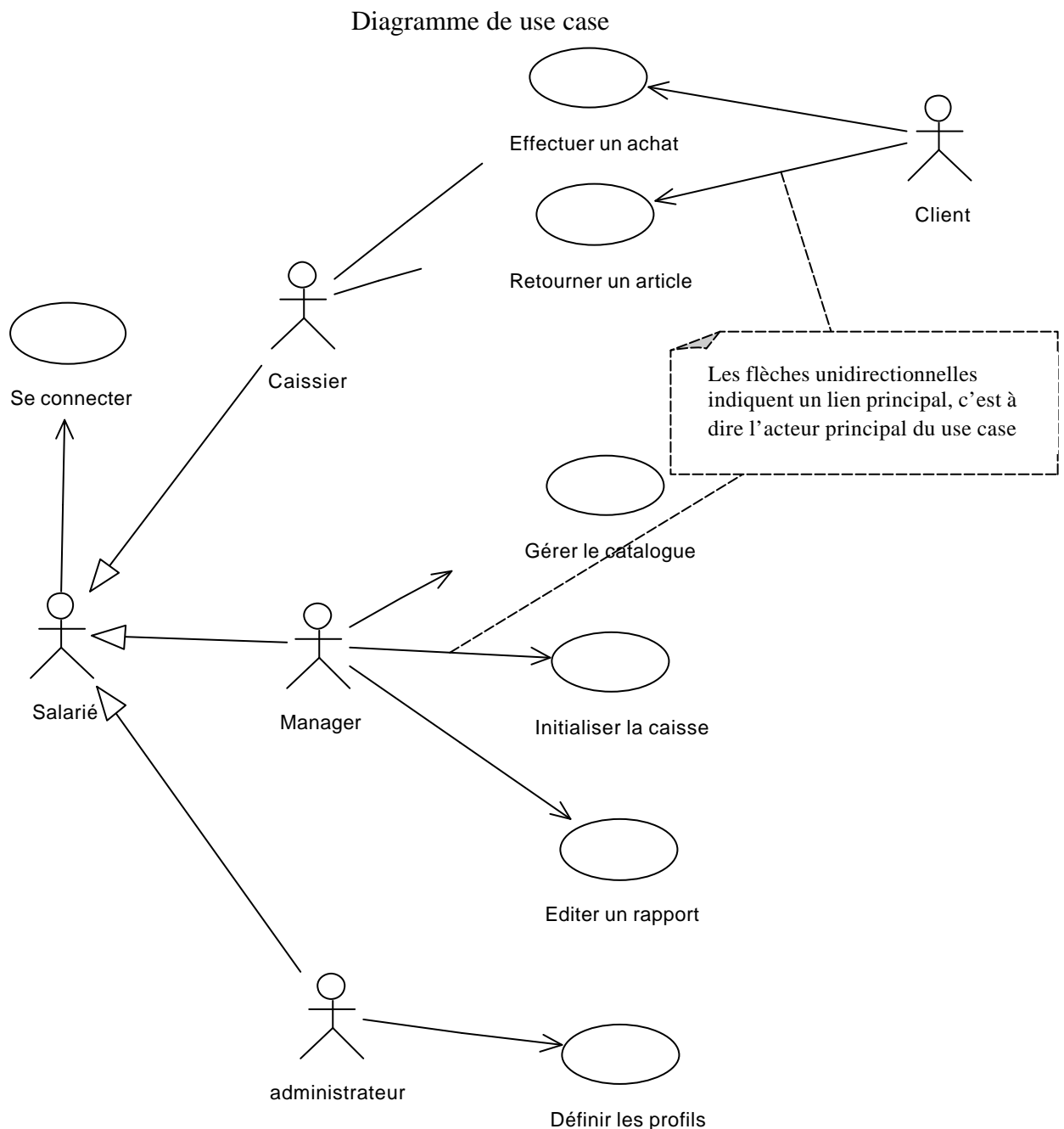
- ? Nous déterminons les acteurs et les exigences
- ? A partir des exigences nous faisons un diagramme d'activité (UML) en colonnes (swim lane). Chaque colonne correspond à un acteur. Cela nous permet de définir les responsabilités des acteurs, donc de définir les uses cases correspondant.

troisième étape : le diagramme de use case

Un use case est une intention d'acteur. Quand un acteur démarre un use case, il a une intention précise. Quelle est cette intention? Effectuer un achat est une intention qui recouvre un certain nombre d'exigences. C'est une unité d'intention complète d'un acteur: c'est donc un use case. Payer ses achats n'est pas une intention complète d'acteur. Nous n'allons pas dans un magasin pour payer, mais bien pour faire des achats. C'est donc une partie (et pas la plus agréable) de effectuer un achat. Ce n'est donc pas un use case.

L'acteur déclencheur de l'achat est le client. C'est lui qui est venu effectuer un achat.

Le diagramme de uses cases est la représentation graphique du tableau d'intention d'acteurs précédent.



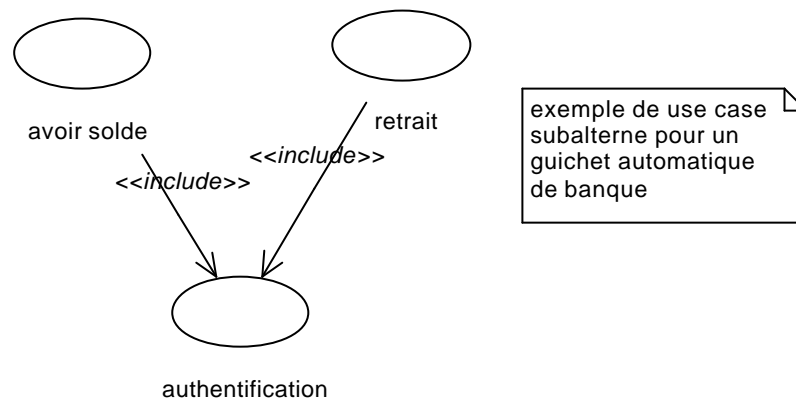
Quand nous définissons les uses cases, il faut faire extrêmement attention à la notion de point de vue. Pour le magasin le client est un acteur essentiel. Par rapport au système informatique ce n'est pas un acteur. L'acteur qui est en interface avec le système informatique est souvent le caissier, mais jamais le client.

Nous voulons définir le système informatique du magasin, pour cela il faut partir de l'ensemble des acteurs qui gravitent autour du magasin, sinon nous risquons d'oublier un certain nombre de fonctionnalités. Le fait de retourner un article est un souci du client, pas du caissier. Si nous ne prenons le point de vue que du système informatique nous risquons de ne pas penser à ce problème. Dans la phase d'analyse nous délimiterons le système informatique dans le processus global du magasin.

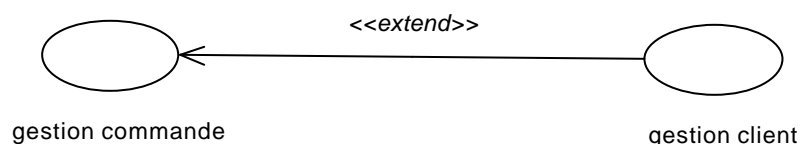
Le schéma de uses cases peut faire apparaître:

? Des fonctionnalités bien précises qui se retrouvent parmi plusieurs uses cases. Nous parlerons alors de use case included ou subalterne.

Un tel use case ne peut être lié qu'à un use case, pas à un acteur. (nous mettrons alors le stéréotype `include` sur le lien use case vers use case subalterne).

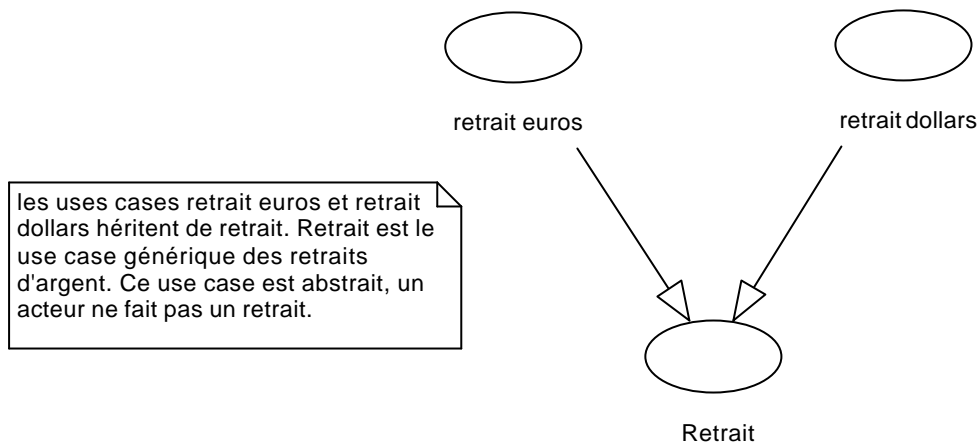


? Un use case peut nécessiter le service d'un autre use case. Le use case dont nous avons besoin du service étend le use case demandeur. Nous utiliserons le stéréotype `extend`. Ici les uses cases peuvent être sollicités par des acteurs.



exemple de use case étendu pour un système de gestion de commande. Quand nous rentrons une commande, il faut pouvoir créer le client s'il n'existe pas.

Nous pouvons avoir des cas de spécialisation de cas d'utilisation. Plusieurs uses cases peuvent avoir une trame commune et donc hériter d'un use case parent et abstrait.



Ces spécialisation, include et extend ne se mettent bien souvent pas dans le premier cycle de développement, car c'est l'étude de l'application qui mettra en évidence ces caractéristiques.

Nous verrons plus tard ces notions de cycle dans le déroulement d'une démarche comme celle-ci.

quatrième étape : use case description de haut niveau

Il nous faut maintenant définir les uses cases. Cette description est une description de haut niveau qui se fait avant l'analyse. Nous sommes toujours dans la définition du besoin.

Une description de haut niveau d'un use case est une description textuelle qui comprend les chapitre suivants:

Nom du Use Case:

Acteur principal:

Acteurs secondaires:

Événement déclencheur du Use Case:

Rôle du Use Case:

Terminaison du Use Case:

Cette description est assez succincte 3 ou 4 lignes maximum pour le rôle du use case. C'est une description narrative d'un processus métier. Cette description ne repose pas sur la technologie objet.

Les cas d'utilisation (autre nom pour un use case) vus dans l'analyse sont dits essentiels. Ici nous ferons donc la description de cas d'utilisation de haut niveau essentiels.

Essentiel signifie que l'on ne fait pas référence à la technologie, que le use case décrit le cœur du métier, ils sont appelés use case pour 100 ans. De haut niveau veut dire que l'on en fait une description brève, sans entrer dans le détail du séquençement du use case.

Nous verrons par la suite des uses cases détaillés (en phase d'analyse), et des uses cases réels (en phase de conception).

Description d'un use case essentiel de haut niveau:

Nom du Use Case:

Acteur principal:

Acteurs secondaires:

Événement déclencheur du Use Case:

Rôle du Use Case:

Terminaison du Use Case:

Effectuer un achat

Client

Caissier

Un client arrive à la caisse avec ses achats

Le caissier passe tous les articles, fait la note, et encaisse le paiement.

Le client a payé et a ramassé tous ses articles.

L'analyse

Après une première ébauche des besoins des clients, il va falloir approfondir ses besoins, examiner les différents scénarios des uses cases (éventuellement avec un diagramme d'activité pour exprimer ces différents scénarios), tenir compte des traitements exceptionnels et cas d'erreur. Il va être nécessaire de regarder le séquençement des opérations d'un point de vue fonctionnel, pour voir comment les différents acteurs interagissent avec le logiciel, à l'aide des diagrammes de séquence boîte noire (la boîte noire c'est le système informatique à développer). Il est alors nécessaire de distinguer les différents objets qui collaborent dans notre système informatique (avec un diagramme de classe). Enfin nous allons détailler le rôle des opérations en définissant des contrats d'opération (soit sous forme textuelle, soit sous forme de diagramme de séquence). Nous aurons alors tous les éléments pour passer à la conception.

Nous n'avons ici comme souci que de détailler les besoins du client pour s'en imprégner, afin de pouvoir le formaliser et le préciser.

cinquième étape : use case description de bas niveau

La description détaillée d'un use case permet de bien décrire les enchaînements des services rendus par le logiciel pour un usage précis. N'oublions pas que nous restons au niveau essentiel, c'est à dire que nous ne donnerons pas de solution au problème, nous ne faisons que préciser sa description. Nous sommes toujours dans l'espace du problème, pas dans celui de la solution.

Une description détaillée de use case comprend:

- ? Nom du Use Case:
- ? Acteur principal:
- ? Acteurs secondaires:
- ? Événement déclencheur du Use Case:
- ? Rôle du Use Case:
- ? Terminaison du Use Case:
- ? Les références croisées des exigences:
- ? Les pré conditions nécessaires au fonctionnement du use case:
- ? Description complète du use case, avec les différents scénarios: Cette description intègre les cas d'erreur (traités par le use case) et les exceptions (forçant la sortie du use case).
- ? Contraintes d'IHM (optionnel): Attention de ne pas décrire l'interface ici, mais bien de préciser des éléments à prendre en compte lors de la réalisation des interfaces.
- ? Contraintes non fonctionnelles (optionnel): Ici nous prendrons en compte les dimensionnements, les performances attendues, ... Ceci permettra de mieux évaluer les contraintes techniques.

La description complète du use case donne la priorité aux choses que les acteurs perçoivent. Nous décrirons les actions des acteurs en commençant par l'acteur principal qui initie le use case, puis nous donnerons les réponses du système (vu comme une boîte noire). Le premier travail se fait sur un cas nominal (c'est à dire idéal). Les cas d'erreur (avec correction et reprise) et les exceptions (avec abandon du use case) sont traités ensuite.

Le formalisme est textuel et prend la forme suivante:

Actions des acteurs	Réponses du système
1. L'acteur principal fait ...	2. Le système lui envoie ...
3. L'acteur principal calcule ...	
4. L'acteur secondaire traite ...	5. Le système envoie le compte rendu

Traitement des cas d'erreur et d'exception:

A l'étape 2 si le code de ... alors le système renvoie un message et l'acteur principal refait ...

A l'étape 4 si ... alors le système affiche ... et le use case s'arrête

En premier lieu nous voyons les échanges entre le système et les acteurs, ainsi que la chronologie et l'enchaînement des actions, dans le cas nominal.

Dans un deuxième temps nous listons les cas d'erreur (avec traitement de récupération) et les traitements d'exception avec arrêt du use case.

Nous restons bien fonctionnel car nous ne décrivons pas comment est réalisé le système, mais comment sont réalisés les services qu'il rend.

Notion de scénario: un use case est un regroupement d'actions plus élémentaires qui permet de traiter une intention d'acteur assez générale. Un scénario est une réalisation du use case, donc une "intention élémentaire". Par exemple pour une gestion de compte client dans une banque, nous avons un use case gérer les comptes. Nous avons plusieurs scénarios: créer un compte, consulter un compte, modifier un compte, ...

Pour chacun des scénarios il va falloir faire une description détaillée de use case (avec traitement d'erreur et d'exception pour chacun). Un diagramme d'activité va permettre de montrer l'ensemble d'un use case, en regroupant l'ensemble des scénarios de ce use case.

Exemple de description d'un use case (effectuer un achat):

Nom du Use Case: **Effectuer un achat**

Acteur principal: Client

Acteurs secondaires: Caissier

Événement déclencheur du Use Case: Un client arrive à la caisse avec ses achats

Rôle du Use Case: Le caissier passe tous les articles, fait la note, et encaisse le paiement.

Terminaison du Use Case: Le client a payé et a ramassé tous ses articles.

Les références croisées des exigences: R2, R3, R4, R6, R7, R15

Les pré conditions nécessaires au fonctionnement du use case: La caisse est allumée et initialisée. Un caissier est connecté à la caisse.

Description complète du use case, avec les différents scénarios:

Ici il n'y a qu'un scénario possible. Nous verrons dans l'exemple suivant un use case avec plusieurs scénarios pour mettre en évidence le diagramme d'activité.

Actions des acteurs	Réponses du système
1) Le <u>client</u> arrive à la caisse avec les articles qu'il désire acheter.	
2) Le caissier enregistre chaque article.	3) Le système détermine le prix de l'article, les informations sur l'article et ajoute ces informations à la transaction en cours. Il affiche ces informations sur l'écran du client et du caissier.
4) Le caissier indique la fin de vente quand il n'y a plus d'article.	5) Le système calcule et affiche le montant total de l'achat.
6) Le caissier annonce au client le montant total.	
7) Le client donne au caissier une somme d'argent supérieure ou égale à la somme demandée.	
8) Le caissier enregistre la somme donnée par le client.	9) Le système calcule la somme à rendre au client.
10) Le caissier encaisse la somme remise par le client et rend la monnaie au client.	11) Le système enregistre la vente effectuée
	12) Le système édite le ticket de caisse
13) Le caissier donne le ticket de caisse au client	
14) le client s'en va avec les articles qu'il a achetés.	

Variantes du scénario nominal (celui qui se déroule quand tout se passe bien).

Paragraphe 2: il peut y avoir plusieurs articles d'un même produit. Le caissier enregistre le produit ainsi que le nombre d'articles.

Paragraphe 3: s'il y a plusieurs articles d'un même produit le système calcule le sous total.

Paragraphe 2: Le code produit peut être invalide. Le système affiche un message d'erreur.

Paragraphe 7: Le client n'a pas la somme d'argent suffisante. La vente est annulée. Remarque: Ceci est une exception qui termine anormalement le use case.

Paragraphe 8: Le caissier n'a pas la monnaie pour rendre au client. Il va faire la monnaie à la caisse principale. Remarque: ici nous avons fait apparaître un nouvel acteur: le caissier principal (ce sera probablement la même personne que le manager mais un acteur différent).

Paragraphe 12: le système ne peut pas éditer le ticket de caisse. Un message est affiché au caissier, et celui-ci change le rouleau de papier.

Paragraphe 14: remarquons que si le client repart sans ses articles, il est probable que le caissier les lui mettra de côté. Cet événement ne change rien à notre système futur. Ce genre d'incident ne sera pas pris en considération.

En terme de représentation nous pouvons préférer mettre une colonne par acteur pour bien voir les responsabilités des acteurs vis à vis du système. Ici cela donnerait:

Actions du client	Actions du caissier	Réponses du système
1) Le <u>client</u> arrive à la caisse avec les articles qu'il désire acheter.	2) Le caissier enregistre chaque article.	3) Le système détermine le prix de l'article, les informations sur l'article et ajoute ces informations à la transaction en cours. Il affiche ces informations sur l'écran du client et du caissier.
	4) Le caissier indique la fin de vente quand il n'y a plus d'article.	5) Le système calcule et affiche le montant total de l'achat.
	6) Le caissier annonce au client le montant total.	
7) Le client donne au caissier une somme d'argent supérieure ou égale à la somme demandée.	8) Le caissier enregistre la somme donnée par le client.	9) Le système calcule la somme à rendre au client.
	10) Le caissier encaisse la somme remise par le client et rend la monnaie au client.	11) Le système enregistre la vente effectuée
		12) Le système édite le ticket de caisse
	13) Le caissier donne le ticket de caisse au client	
14) le client s'en va avec les articles qu'il a achetés.		

Cette représentation met en évidence que le client n'a pas accès au système informatique.

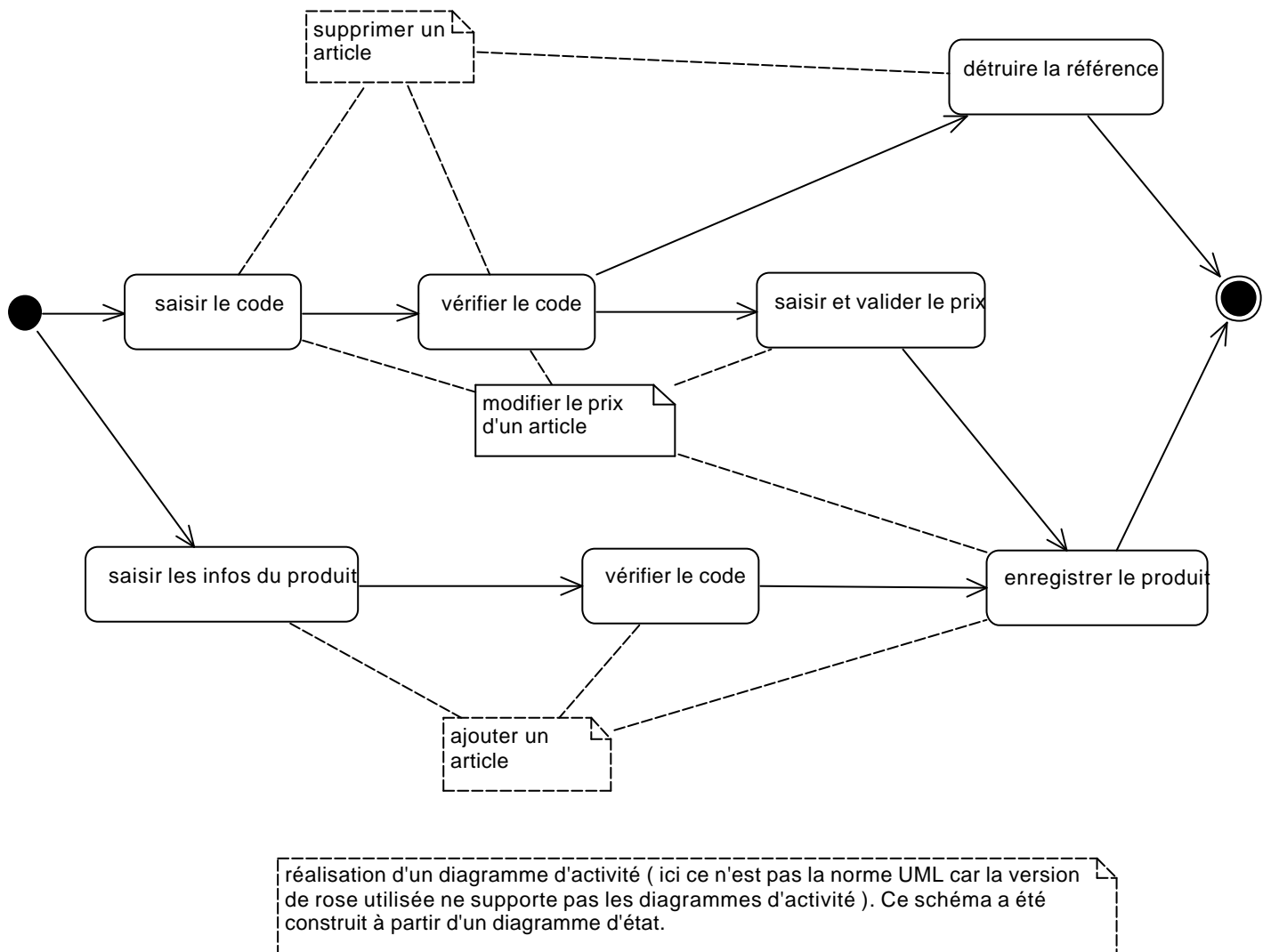
Contraintes d'IHM (optionnel): La désignation de l'article et son prix (éventuellement le sous total si plusieurs occurrences du même article) sont affichés à chaque article au caissier et au client. Le total est affiché également aux deux.

Contraintes non fonctionnelles (optionnel): Le magasin reçoit en moyenne 200 clients par jour. Chaque client achète en moyenne 10 articles.

Voici un exemple de diagramme d'activité représentant la réunion de scénarios modélisant un use case.

Nous allons gérer le catalogue en utilisant un diagramme d'activité pour représenter l'ensemble des scénarios (un scénario est un déroulement d'un use case de bout en bout, en commençant par le début et se terminant par la fin.)

Un scénario supporte aussi des variantes et des exceptions. Chaque scénario peut être décrit comme le use case ci-dessus. Mais il est bon de regrouper l'ensemble de ces scénarios dans un diagramme d'activité pour avoir une vue complète du use case.



Note :



Ceci représente un commentaire dans tout schéma UML.

Les notes nous montrent les trois scénarios possibles pour ce use case. Pour chacun de ces scénarios il est nécessaire de faire le tableau précédent pour mieux détailler le use case dans le scénario particulier. En particulier il faut penser aux cas d'erreur et aux exceptions.

Remarquons qu'un certain nombre de symboles du diagramme d'activité ne sont pas disponibles comme l'alternative, et les conditions de cette alternative.

sixième étape : diagramme de séquence boîte noire

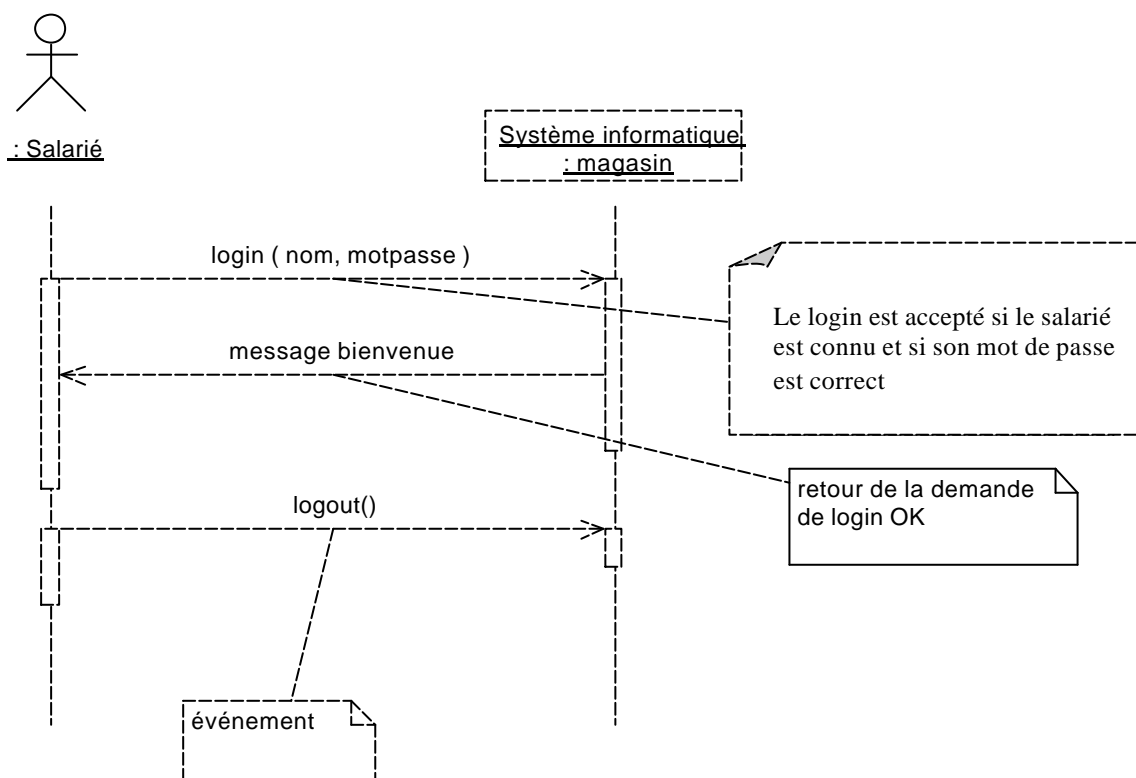
Les uses cases ont été validés par le client. Nous allons donc changer notre point de vue (workflow c'est à dire processus métier) pour adopter un point de vue système informatique. Nous abandonnons donc les événements métier pour nous intéresser aux événements informatiques.

Nous allons développer un diagramme de séquence par scénario de use case détaillé. Ces diagrammes de séquence sont dits boîte noire car nous ne savons toujours pas comment sera fait le système informatique. Ici nous précisons les échanges entre les acteurs utilisateurs du système informatique et le système proprement dit.

Les interactions des acteurs directement sur le système sont appelées des événements. Les actions engendrées par le système suite à ces événements sont appelées des opérations. A un événement correspondra une opération. Dans la terminologie message et événement sont équivalents entre eux, ainsi que méthode et opération.

Les diagrammes de séquence seront agrémentés de notes et commentaires qui permettront d'illustrer le diagramme: explication de contrôles, itérations, logique, choix, ...

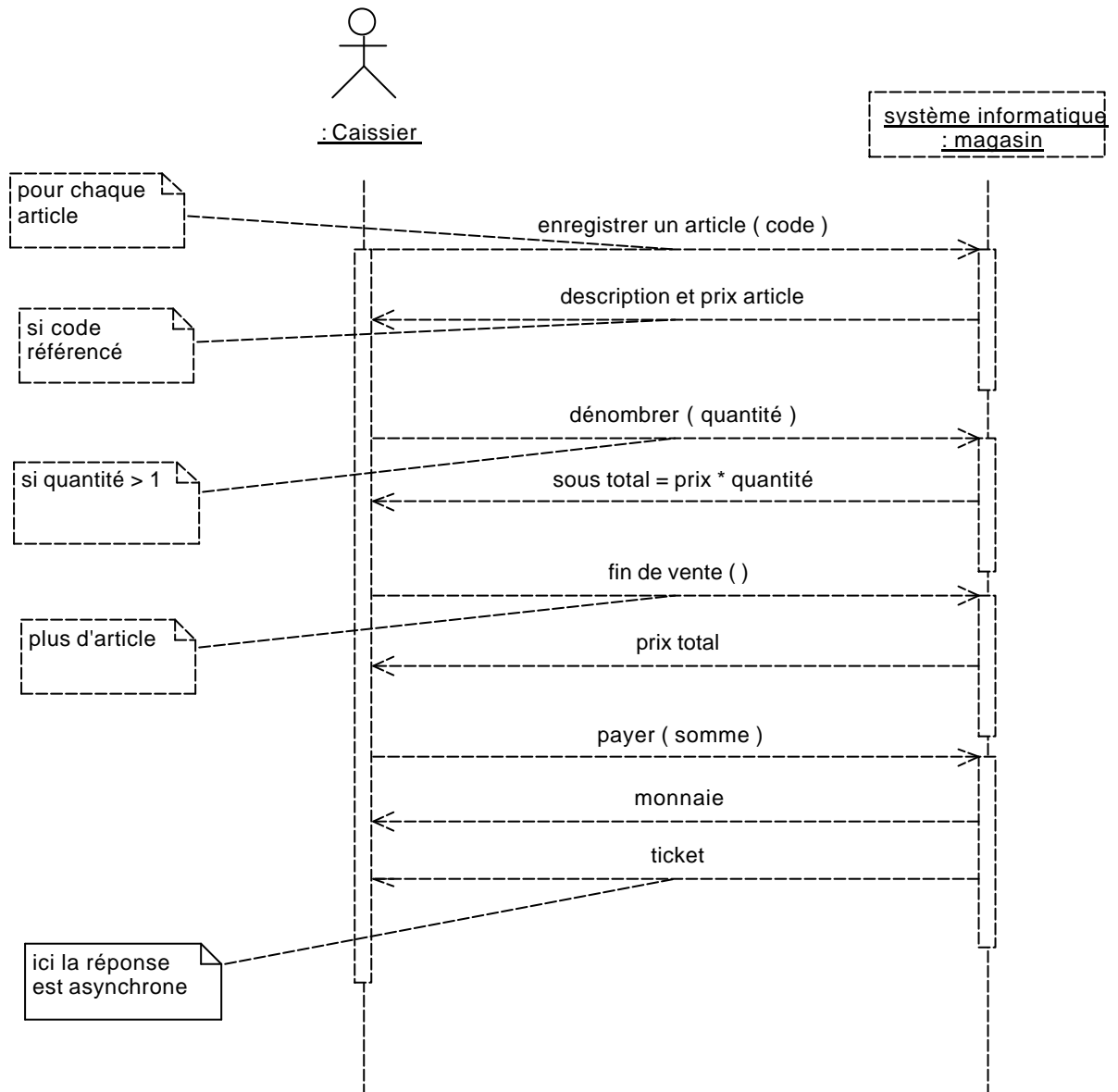
Rappelons qu'un diagramme de séquence boîte noire est de la forme:



Les valeurs de retour se formalisent de différentes manières suivant la version d'UML choisie. Ici j'ai pris un exemple de notation la plus simple possible. Attention à ne pas confondre avec un événement (sens acteur vers système informatique).

En fin de phase nous connaissons les échanges entre les utilisateurs et le système informatique. Nous pourrions construire une maquette des écrans pour les acteurs. Formellement nous le ferons en phase de conception. Notre maquette d'écran serait une maquette "fonctionnelle" (c'est à dire que le dessin de l'écran n'est pas figé).

Diagramme de séquence boîte noire du use case détaillé "effectuer des achats":



septième étape : diagramme de classe d'analyse

Le diagramme de classes est un des documents les plus importants, voire le plus important de l'analyse d'un logiciel par une méthode objet. C'est le premier document qui est dans le monde des objets (les acteurs n'étant pas forcément représentés dans le système informatique). C'est donc l'entrée dans le monde des objets.

Ce diagramme est un diagramme de classe d'analyse. Il ne représente pas les classes Java ou C++ que nous développerons par la suite.

Ici nous nous intéressons aux classes du domaine métier, c'est à dire des objets dont on parle dans le cahier des charges et dans les uses cases principalement. Nous ne nous intéressons pas aux objets informatiques dans ce diagramme (sauf bien entendu si le métier est l'informatique !!). Cela viendra en son temps, dans le diagramme de classe de conception. Quand nous passerons à la conception des classes disparaîtront, d'autres apparaîtront, dont les classes informatiques (collections, ...). C'est normal. En phase d'analyse, nous voulons simplement faire un diagramme de classe d'objets manipulés dans le domaine métier considéré.

Dans ce diagramme de classe les opérations ne sont pas représentées (ce sera lors de la conception).

Ce diagramme montrera les concepts du domaine métier, les liens (associations) entre ces concepts (avec leur cardinalité ou multiplicité), et les attributs de ces concepts (souvent sous forme de données simples).

Le but de ce diagramme est de clarifier le domaine métier du client, et de se familiariser avec ce domaine.

Dans une analyse structurée nous décomposons l'analyse suivant les tâches ou les fonctions. Dans une application à dominante gestion, nous décomposerons notre application suivant les données, et les traitements. Ici nous analyserons la complexité du domaine en regardant les objets métier et leurs interactions entre eux.

Comment identifier les bons objets métier pour construire notre diagramme de classe d'analyse?

Il va falloir recenser dans les documents de définition des besoins et dans les documents d'analyse l'ensemble des objets métier.

Les noms ou groupes nominaux vont être de bons candidats pour être élus au titre de classe, objet ou attribut. Cependant, il faut être prudent car il y aura aussi un certain nombre de faux amis et de doublons.

Voici une démarche de sélection des classes candidates:

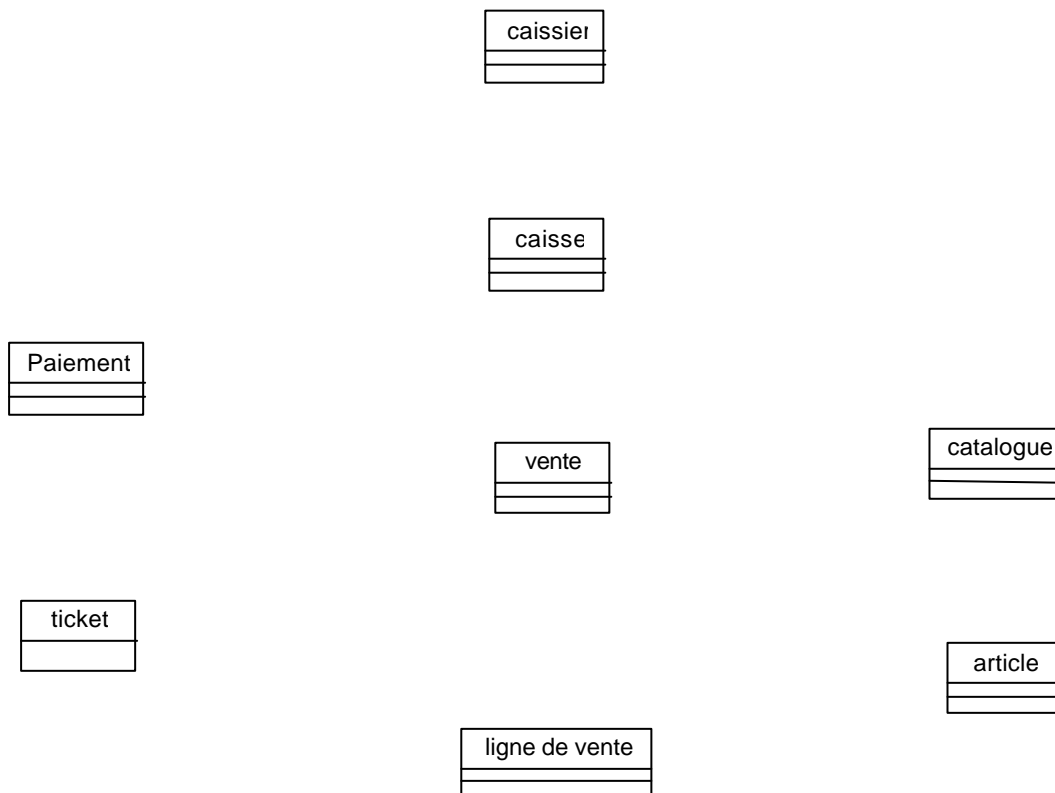
Par use case, nous réaliserons un diagramme de classe, le diagramme final étant la superposition de tous ces diagrammes.

Pour un use case nous recenserons les groupes nominaux rencontrés. Nous identifierons alors:

- ? Les classes (noms communs).
- ? Les objets (noms propres ou nom référencés).
- ? Les attributs (données simples qui qualifient une classe ou un objet).
- ? Les éléments qui sont étrangers à notre problème ou qui n'y ont pas de rôle réel.

- ?
- ? Les " classes fonctionnelles " qui ne sont porteuses d'aucune information, mais seulement de traitement, qui ne sont souvent pas des classes de notre diagramme. Par exemple gestionnaire du planning, décodeur de message, ...
 - ? Il est parfois difficile de savoir si une information est un attribut d'une classe ou une classe (avec une association avec la classe). Dans le doute il vaut mieux faire une classe, quitte à revenir en arrière dans une itération ultérieure. Par exemple pour la classe personne l'âge est un attribut (donnée simple) l'adresse étant à priori une donnée complexe sera plutôt une autre classe associée à la personne.
 - ? Les entités suivantes peuvent prétendre à devenir des classes :
 - les objets physiques (produit...),
 - les transactions (réservation, commande,...),
 - les lignes de transaction (ligne de commande...),
 - les conteneurs (catalogues, lots,...),
 - les organisations (services, départements,...),
 - les acteurs ou les rôles (client, fournisseur....),
 - les regroupements en abstraction (modèle, genre, type, catégorie....),
 - les évènements (vol,...)

Pour le use case " effectuer un achat " voici la liste des classes sélectionnées:



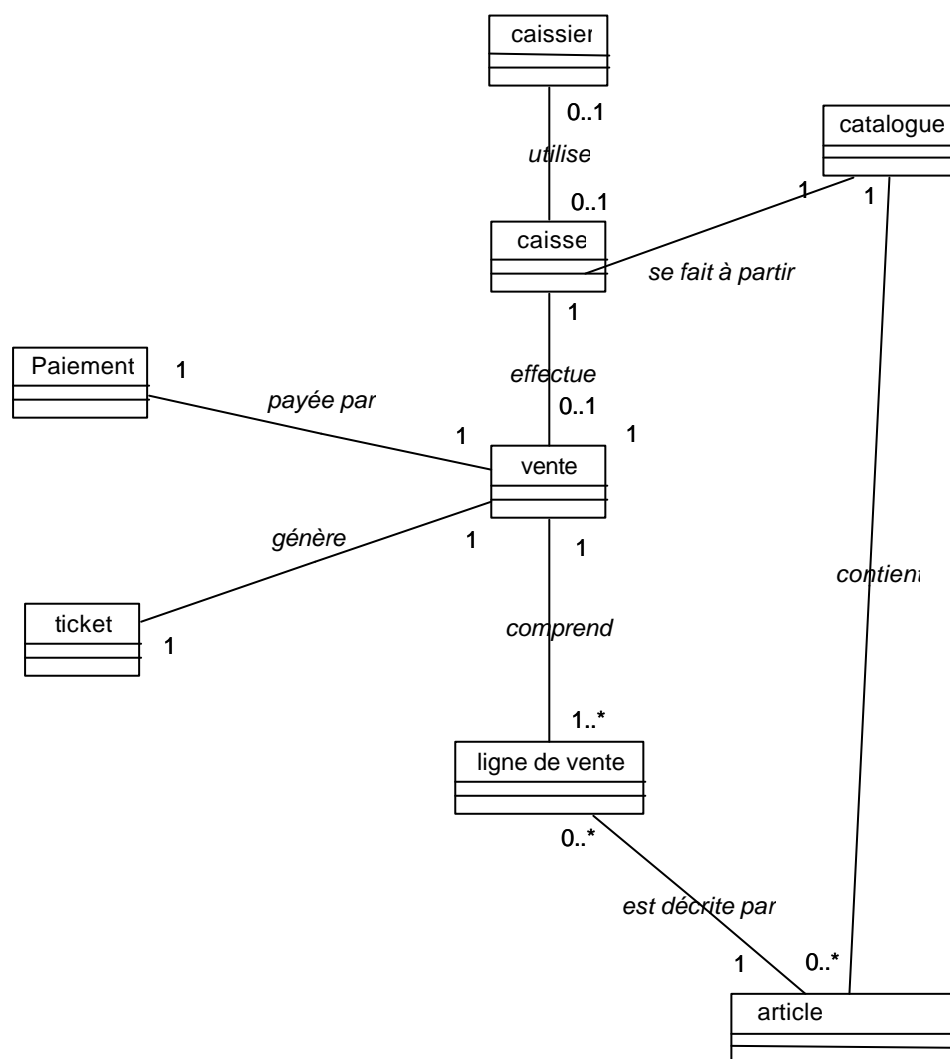
Nous ne gérerons pas les exemplaires des articles. Nous appellerons article un ensemble d'exemplaires identifiés par le même code barre, et comportant des informations identiques (prix ...).

Nous allons maintenant rajouter les associations entre les classes, en donnant un intitulé et des cardinalités à ces associations.

Notre but n'est pas de mettre à jour toutes les associations entre les différentes classes, mais de noter les associations pertinentes pour comprendre le problème. Il faut privilégier les associations qui durent dans le temps, et dont la connaissance est nécessaire à la compréhension du problème.

Il faut éviter les associations redondantes ou dérivables d'autres associations.

L'établissement de ces associations nous amène à poser des questions au client. C'est un des buts recherchés. Voici une possibilité de diagramme de classe avec ses associations.



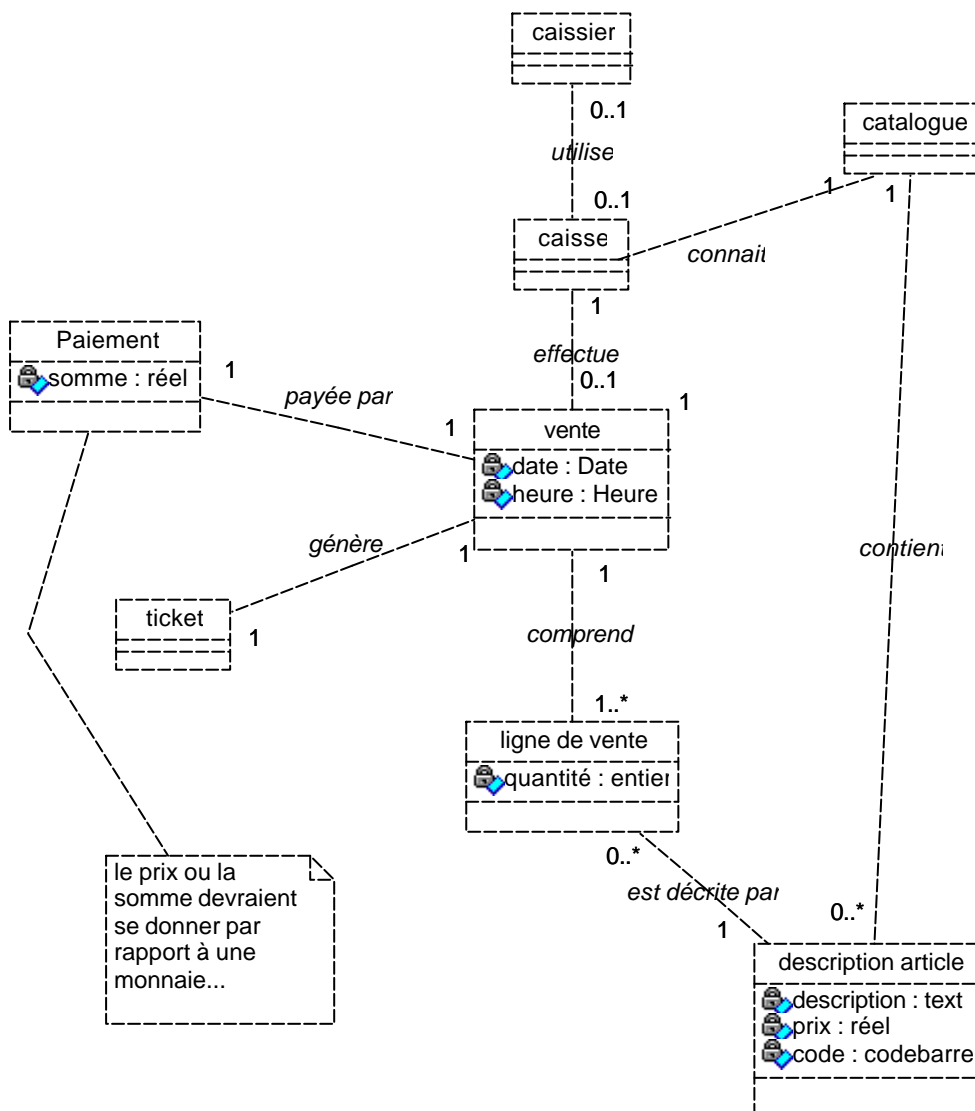
Sur ce diagramme de classe d'analyse il faut maintenant mettre les attributs des classes qui ont été repérés dans le cahier des charges, le diagramme de use case, ou dans le diagramme de séquence boîte noire du use case.

A priori, les attributs présents dans notre diagramme de classe d'analyse sont des attributs simples. On pourra faire exception des données pures qui sont des regroupements de données simples (ici code, adresse, prix) .

Les attributs qui ont un comportement sont des classes associées à notre classe. Ceci ne présage en rien du diagramme de classe de conception. Nous verrons ultérieurement ce que deviennent les associations dans ce schéma...

Dans les attributs, il ne doit pas y avoir de clef sur un autre objet. Cela se représente par une association.

Voici notre diagramme de classe enrichi des attributs d'analyse.



huitième étape : Le glossaire

Le glossaire répertorie tous les termes et leur définition. Il n'y a pas de graphisme particulier pour ce document. Il peut être fait sous la forme suivante:

terme	catégorie	description
Effectuer un achat	Use case	C'est le processus que suit un client pour effectuer ses achats dans le magasin.
Quantité	Attribut	C'est un attribut de la classe ligne de vente qui donne le nombre d'occurrences d'un même article lors d'un achat.
Vente	Classe	C'est la classe qui représente une vente en cours ou quand elle est finie.
...		

neuvième étape : les contrats d'opération

Nous allons maintenant étudier ce qui est fait dans le système, pour chaque flèche qui attaque le système dans les diagrammes de séquence.

Notre but est de comprendre la logique de fonctionnement du système. Les flèches représentées dans les diagrammes de séquence boîte noire déclenchent des opérations sur le système. Nous allons donc définir précisément le rôle de ces opérations en nous appuyant sur le diagramme de classe (appelé aussi modèle de domaine). C'est ce qui s'appelle des contrats d'opération.

Le système informatique a été vu comme une boîte noire (en particulier à travers les diagrammes de séquence). Il serait, d'un point de vue fonctionnel, intéressant de décrire ce que fait le système, en se basant sur le diagramme de classe, sans pour autant décrire comment il le fait. Les contrats d'opération vont nous permettre de décrire les changements d'états du système (c'est à dire les changements sur les objets et leurs associations) quand les opérations (issues des diagrammes de séquence) sont invoquées par les acteurs.

Un contrat d'opération est un document textuel qui décrit les changements provoqués par une opération sur le système. Le contrat d'opération est écrit sous forme textuelle et comporte des pré conditions et des post conditions. Les pré conditions décrivent l'état du système nécessaire pour que l'opération puisse s'effectuer. Les post conditions décrivent l'état du système lorsque l'opération s'est achevée.

Contrat d'opération type:

? <u>Nom:</u>	Nom de l'opération avec ses paramètres.
? <u>Responsabilités:</u>	Description du rôle de l'opération.
? <u>Exigences:</u>	Liste des exigences dont le use case tient compte dans cette itération.
? <u>Notes:</u>	Remarques diverses.
? <u>Pré conditions:</u>	Etat nécessaire du système pour que l'opération se fasse.
? <u>Post conditions:</u>	Etat du système lorsque l'opération s'est déroulée entièrement.

Les Pré conditions décrivent l'état du système, c'est-à-dire l'état des objets du système comme décrit dans le diagramme des classes d'analyse.

Nous jouons l'opération sur le système, en aveugle, et jusqu'à sa fin.

Notons les changements de l'état du système (des objets et de leurs associations) et nous obtenons les post conditions. Ces post conditions sont décrites sous la forme "has been", c'est-à-dire l'objet X **a été** modifié, son attribut Y **a été** mis à vrai.

Les post conditions portent sur 6 clauses:

- ? Les objets créés.
- ? Les objets détruits.
- ? Les associations créées.
- ? Les associations détruites.
- ? Les attributs modifiés.
- ? Les événements d'affichage (fonctionnels bien sûr !!!).

Prenons un exemple simple pour traiter ces contrats d'opération.

Modifier le prix d'un article au catalogue.

- ? Nom: modifier (cecode, nouveauprix).
- ? Responsabilités: modifier le prix d'un article de code donné, présent dans le catalogue.
- ? Exigences: R1, R13, R14 pour le use case gérer le catalogue.
- ? Notes: Si le code ne correspond pas a un article du catalogue, un message d'erreur sera envoyé au manager et l'opération s'arrête.
- ? Pré conditions: Il y a un article correspondant au code donné.
- ? Post conditions: l'attribut prix de l'objet description article, dont le code est cecode, a été changé par nouveauprix.

Nous allons faire les contrats d'opération des opérations du use case effectuer un achat. Pour cela il faut partir du diagramme de séquence boîte noire du use case effectuer un achat et du diagramme de classe d'analyse, et pour chaque flèche dirigée vers le système, rédiger un contrat d'opération. Par exemple :

Enregistrer un article.

- ? Nom: enregistrer un article (cecode).
- ? Responsabilités: enregistrer un article lors d'une vente, et l'ajoute à la vente en cours.
- ? Exigences: R15 pour le use case effectuer un achat.
- ? Notes: Si l'article est le premier de la vente, il débute la vente.
Si le code cecode n'est pas référencé dans le catalogue, un message d'erreur est envoyé au caissier.
- ? Pré conditions: Il y a un article correspondant au code donné.
Il y a un caissier à la caisse.
- ? Post conditions: Si c'est le premier article de la vente, il faut qu'un objet vente ait été créé et associé à la caisse.
Une ligne de vente (ldv) a été créée. Elle a été associée à un article correspondant au code cecode.
La ligne de vente ldv a été associée à la vente.
Le prix et la description de l'article ont été affichés.
L'attribut quantité a été mis à 1.

Dénombrer les articles identiques.

- ? Nom: dénombrer (quantité).
- ? Responsabilités: donne le nombre d'articles identiques à l'article enregistré, et calcule le sous total.
- ? Exigences: R3, R4 pour le use case effectuer un achat.
- ? Notes:
- ? Pré conditions: Il y a une ligne de vente (ldv) correspondant au dernier article enregistré.
- ? Post conditions: L'attribut quantité de ldv est affecté.
Le sous total (ldv.quantité*prix) est affiché.

Finir la vente.

- ? Nom: finir vente ().
- ? Responsabilités: finit une vente et calcule le prix total de la vente.
- ? Exigences: R2 pour le use case effectuer un achat.
- ? Notes:
- ? Pré conditions: Il existe une vente et au moins une ligne de vente.
- ? Post conditions: La vente est marquée à terminé. *Notons ici que nous avons besoin d'enregistrer le fait que la vente soit finie. Cela nous amène à définir un attribut booléen vente terminée.*
Le prix total de la vente est affiché. *Ici aussi nous allons enregistrer le prix total de la vente dans l'objet vente. Cela nous amène à définir un attribut réel Total.*

Payer la vente.

- ? Nom: payer la vente (somme).
- ? Responsabilités: calcule la monnaie à rendre et imprime le ticket de vente.
- ? Exigences: R7 pour le use case effectuer un achat.
- ? Notes: Si la somme n'est pas suffisante un message est envoyé au caissier.
- ? Pré conditions: Il y a une vente v en cours marquée à terminé.
- ? Post conditions: Un objet de type Paiement p a été créé.
P a été associé à la vente v.
V. total a été affecté à p.somme.
La monnaie à rendre (somme – p.somme) a été affichée.
Un ticket t a été créé.
La vente v a été associée au ticket t.
Le ticket de vente a été imprimé.

Nous allons voir une autre manière de représenter ces contrats d'opération à l'aide de diagramme de séquence boîte blanche. Cette manière de représenter les contrats d'opération est beaucoup plus utilisée que la manière textuelle. Elle n'est pas forcément mieux d'ailleurs...

Voici le diagramme de séquence boîte blanche (on regarde ici l'intérieur du système) de l'opération modifierprix. Cela correspond au contrat de l'opération.

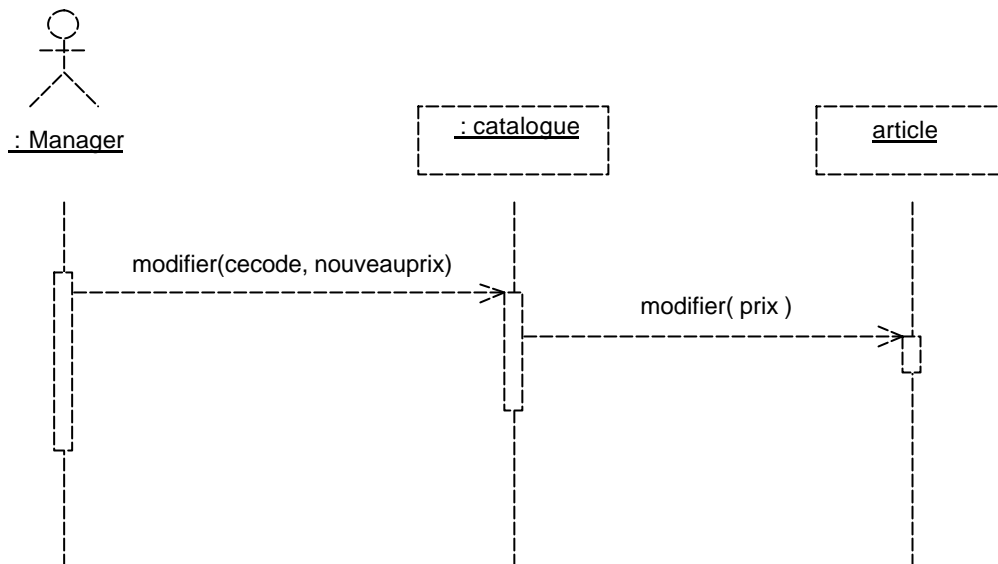
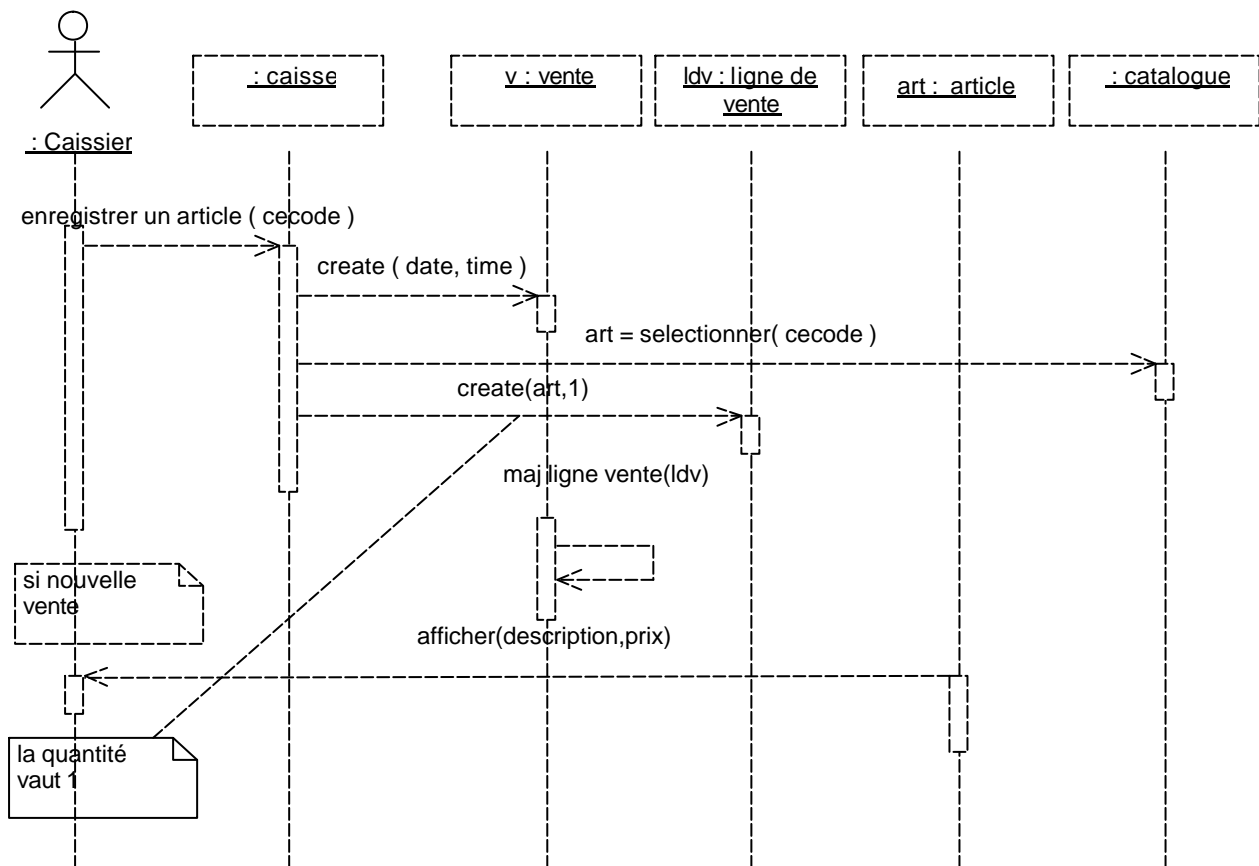


Diagramme de séquence boîte blanche de l'opération modifier le prix d'un article au catalogue dans le use case gérer le catalogue.

Maintenant nous allons faire les diagrammes de séquence boîte blanche des opérations du use case effectuer un achat.

Diagramme de séquence boîte blanche de enregistrer un article.



On notera que sur cette représentation nous ne voyons pas bien les associations qui ont été créées. Par contre nous voyons mieux les objets qui coopèrent pour arriver au résultat. Ces deux représentations du contrat sont assez complémentaires, et mériteraient d'être présentes toutes les deux.

Diagramme de séquence boîte blanche de l'opération dénombrer les articles identiques.

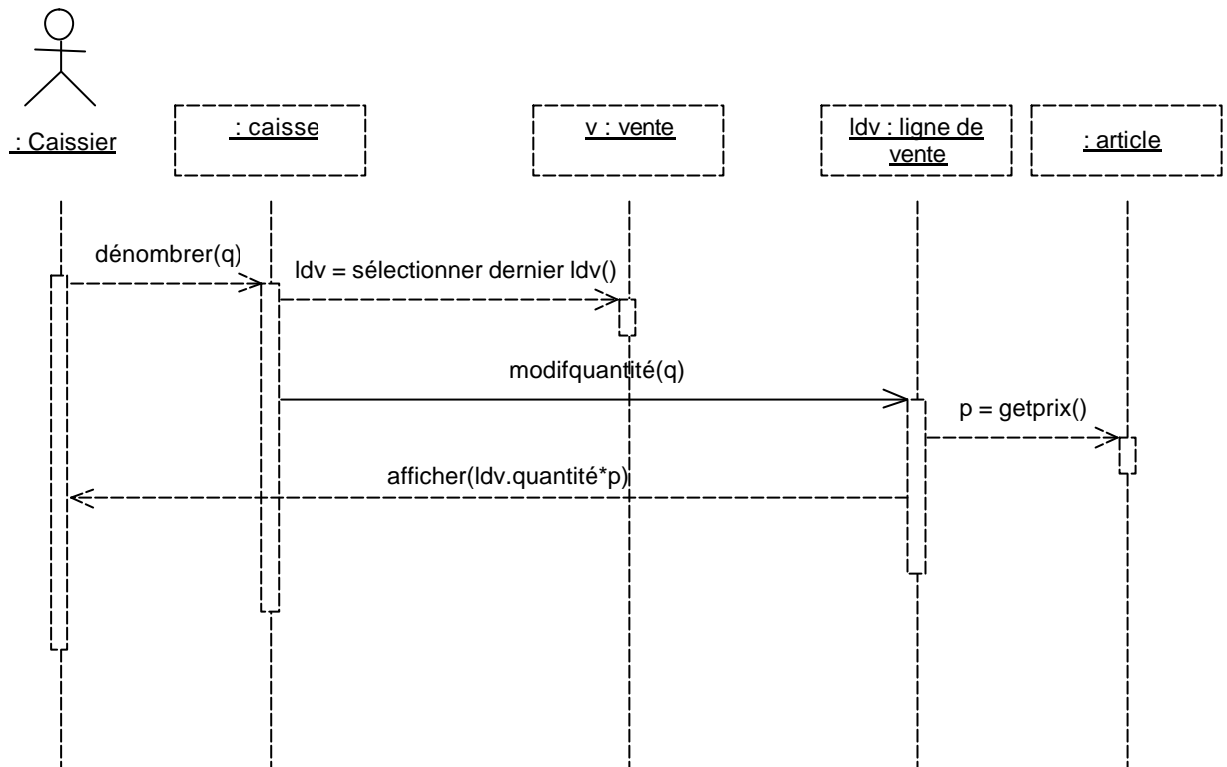


Diagramme de séquence boîte blanche de l'opération finir vente.

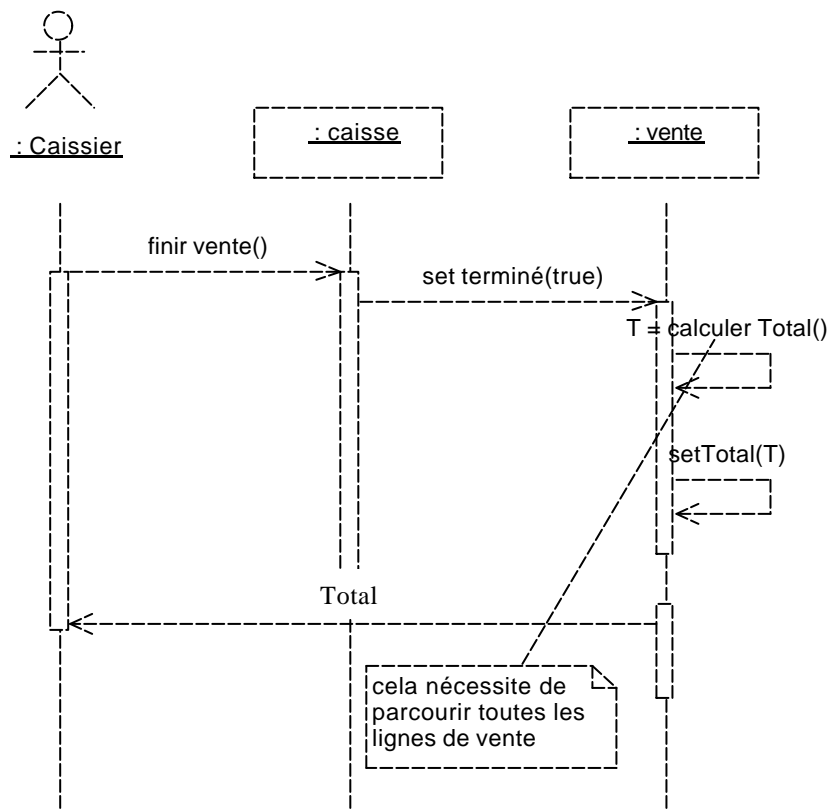
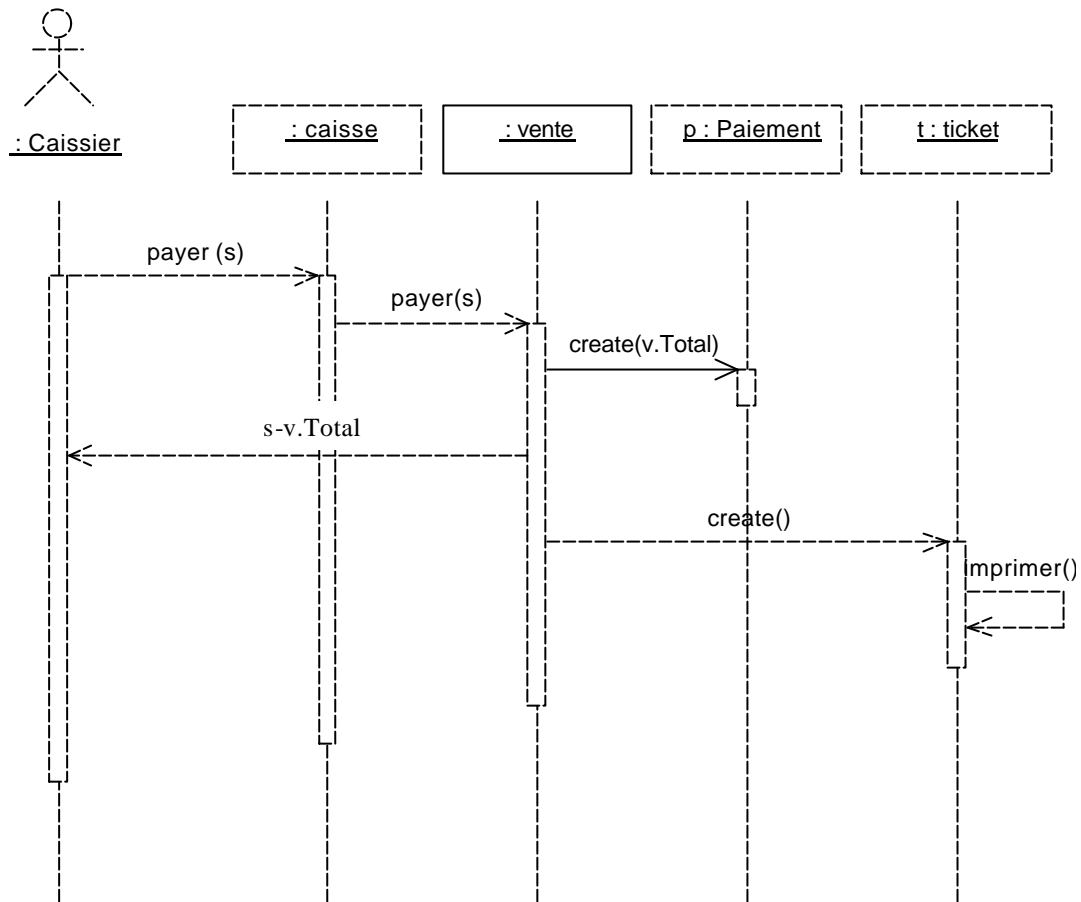


Diagramme de séquence boîte blanche de l'opération payer la vente.



Ces schémas sont moins précis que le contrat d'opération sous forme textuelle. Nous n'y retrouvons pas les associations. Les schémas de diagramme de séquence boîte blanche utilisés pour décrire un contrat d'opération (bien que souvent utilisés) vont introduire des choix de conception (qui déclenche les opérations?). Si ces choix ne sont pas faits le diagramme induit une erreur d'interprétation. Nous préférons donc faire ces choix de conception dans le diagramme de collaboration, et garder le contrat d'opération sous forme textuelle.

La conception

dixième étape : le diagramme d'état

Le diagramme d'état est à la frontière de l'analyse (par la compréhension des cycles de vie de certains objets) et de la conception (par les méthodes qu'il permet de définir dans les classes étudiées).

"Le diagramme se réalise pour les classes d'objets ayant un comportement significativement complexe. Toutes les classes d'un système n'ont pas besoin d'avoir un diagramme d'état." (James RUMBAUGH)

Si, dans l'analyse de notre système, des classes paraissent complexes, par exemple si elles sont souvent sollicitées dans les diagrammes de séquence boîte blanche, il est bon de faire un diagramme d'état de ces classes: cela permet d'avoir une vue orthogonale de la classe par rapport à ses liens avec les autres classes. Ainsi, nous regardons le comportement des objets d'une classe, et leur évolution interne au fil des événements. Cela nous donne le cycle de vie des objets d'une classe.

Ainsi, ayant perçu le fonctionnement des objets d'une classe, il est plus facile de vérifier que les objets des autres classes respectent bien ce comportement. Cela améliore notre perception du problème.

Cela implique que les diagrammes d'état doivent être confrontés aux diagrammes d'interaction afin de vérifier la cohérence de ces diagrammes.

Les diagrammes d'état permettront également de compléter les diagrammes de classes de conception en particulier en mettant en évidence des méthodes publiques (correspondant aux actions du diagramme d'état) et des méthodes privées (correspondant aux activités du diagramme d'état).

Quelques rappels:

Un état représente un objet dans une situation où:

- ? Il a une réaction déterminée par rapport à un événement.
- ? Il exécute une activité.
- ? Il satisfait une condition.

Un état a une durée finie.

Un objet peut avoir une activité dans un état. Par exemple quand la caisse est en attente d'un client, elle peut faire défiler un message de bienvenue sur l'écran client.

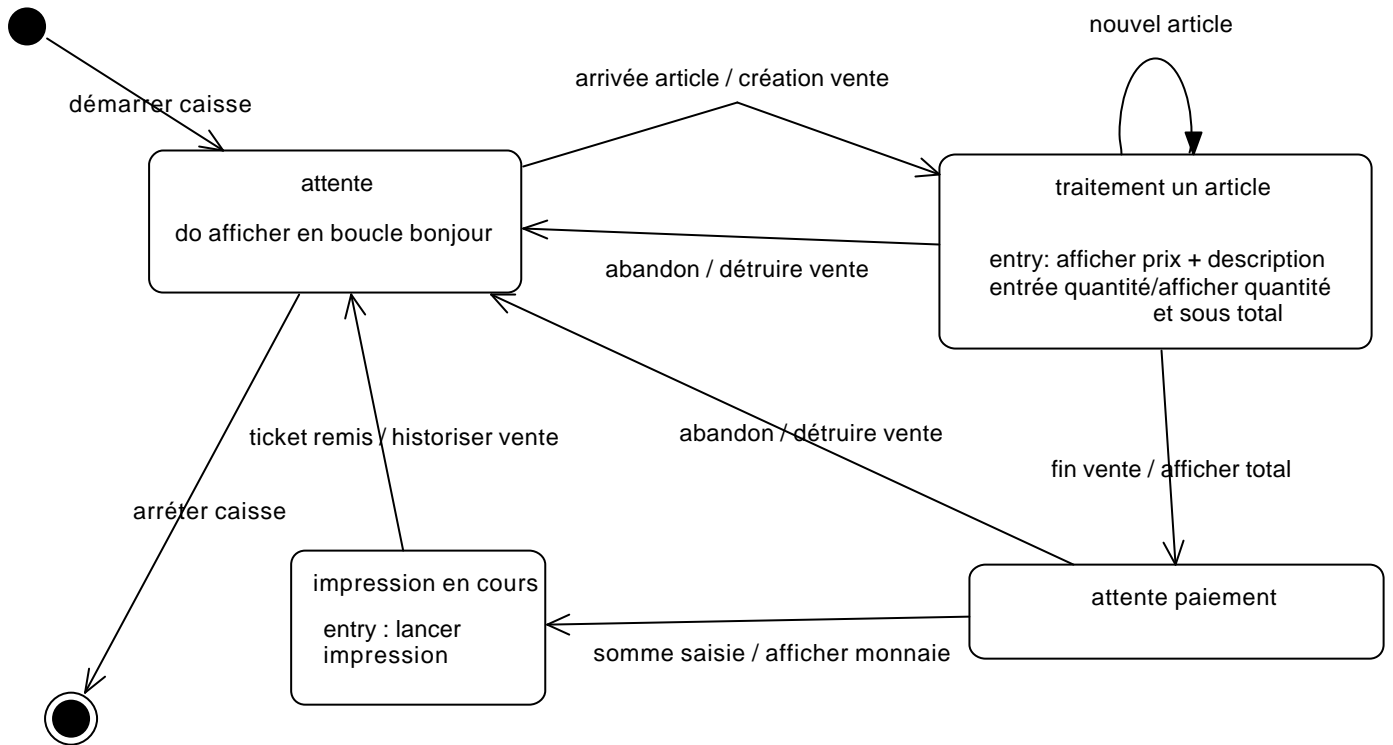
Un objet peut avoir des actions qui seront déclenchées sur des événements. Par exemple sur l'événement fin de vente, la caisse va afficher le total à payer.

Une action est une opération atomique qui ne peut être interrompue.

- ? Elle est considérée comme instantanée.
- ? Elle est généralement associée à une transition.

Une activité est une opération qui dure et qui peut être interrompue.

- ? Elle est associée à un état.
- ? Elle peut être continue et interrompue par un événement.
- ? Elle peut être finie et déclencher une transition.



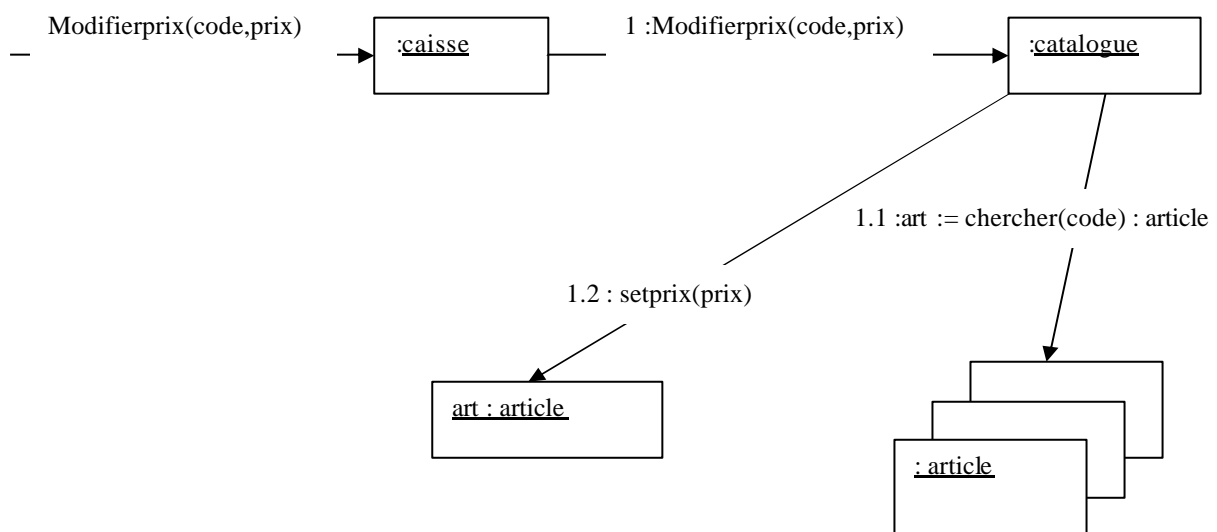
Etat de la caisse concernant le use case effectuer un achat

onzième étape : le diagramme de collaboration

Le diagramme de collaboration va nous montrer comment les objets interagissent entre eux pour rendre le service demandé par le contrat d'opération. Nous allons d'abord observer la syntaxe, et la forme que prend ce diagramme d'opération. Les objets doivent demander des services à d'autres objets. Il leur faut donc connaître ces objets pour pouvoir leur adresser des messages. Nous allons donc regarder la visibilité des objets (c'est à dire comment un objet peut connaître d'autres objets). Enfin quand une ligne du contrat d'opération est réalisée il faut se poser la question de savoir quel objet agit pour réaliser cette ligne (qui crée tel objet, qui reçoit l'événement initial). En un mot il est nécessaire de définir les responsabilités des objets. L'expérience et le savoir faire des professionnels nous ont permis de définir des règles, des modèles de pensée, qui nous permettront de nous guider pour définir les responsabilités des objets. Ce sont les GRASP patterns que nous verrons enfin, avant de traiter notre caisse.

1) Syntaxe du diagramme de collaboration

Nous allons prendre un exemple de diagramme de collaboration pour introduire toutes les notions véhiculées dans ce diagramme. Rappelons nous que ce diagramme, dans le contexte de la conception, nous montre comment les objets coopèrent entre eux pour réaliser les opérations définies par les contrats d'opération.



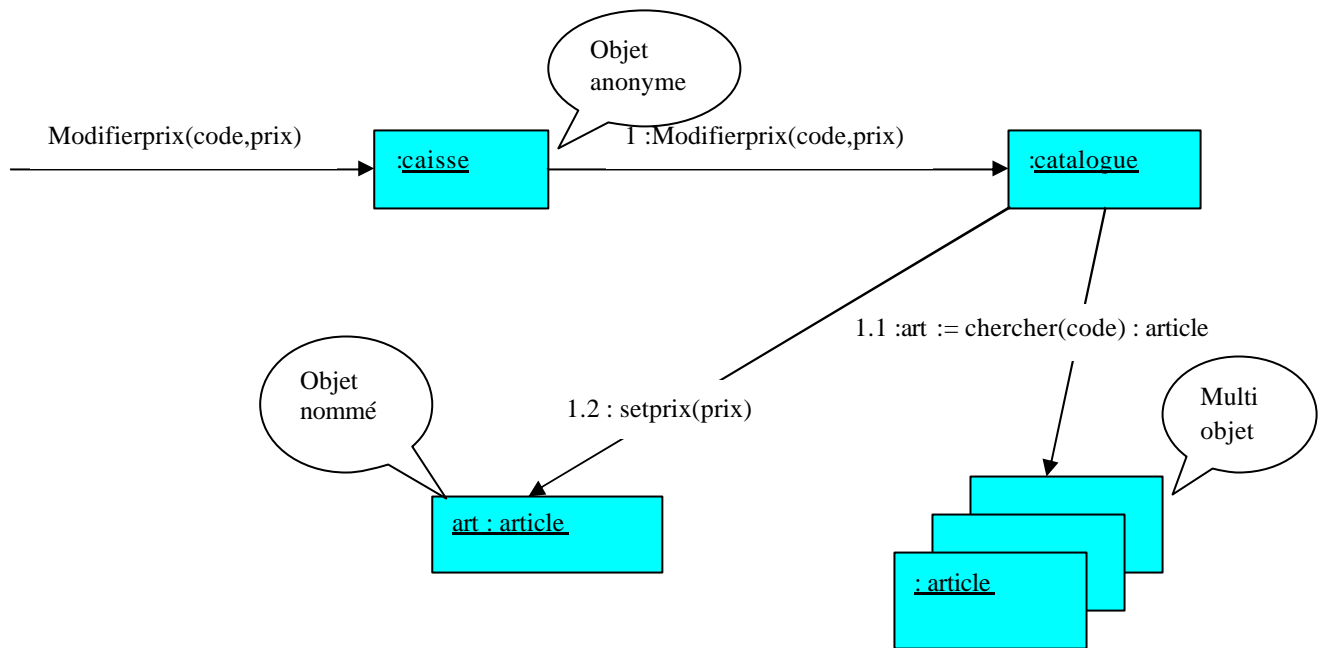
a) les objets

Ici nous représentons la coopération des objets pour rendre le service demandé. Il s'agit donc bien d'objets. Ces objets apparaissent sous trois formes :

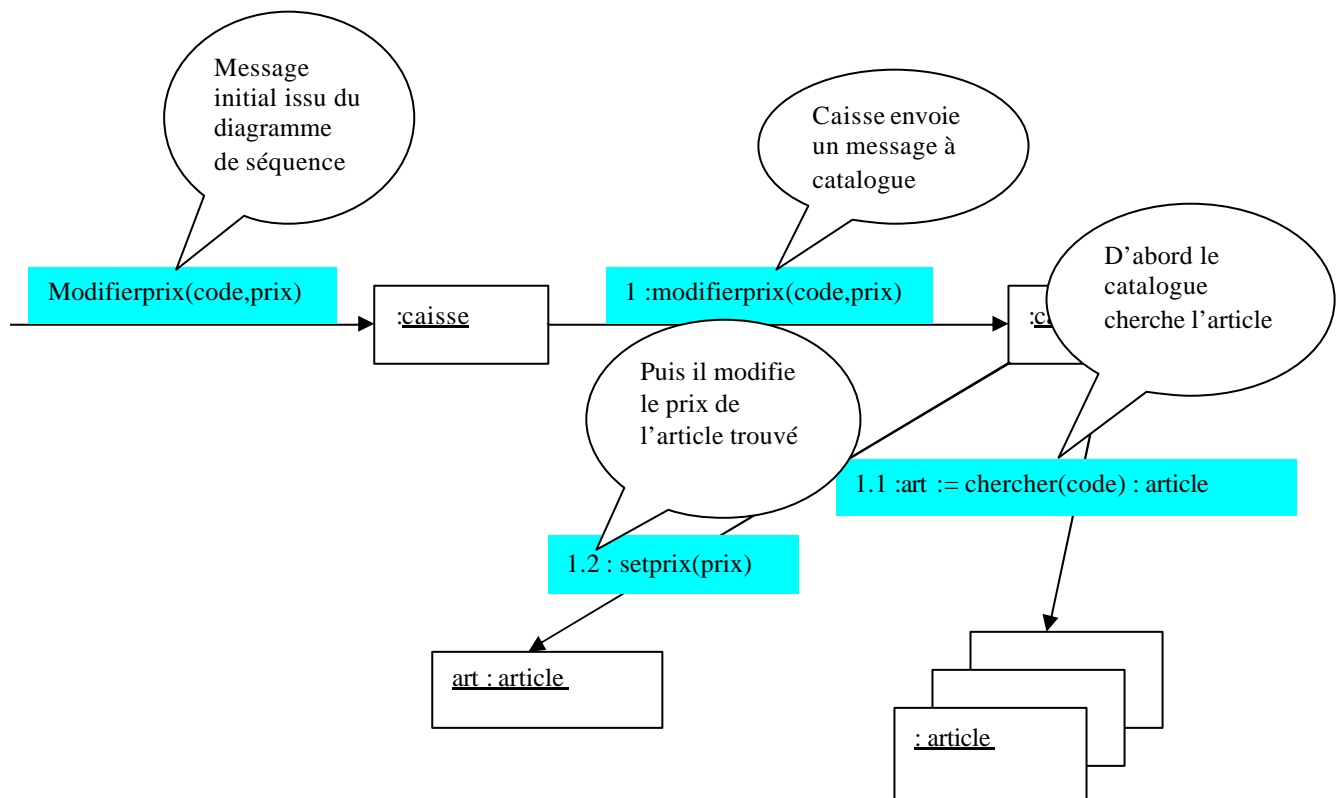
Les objets **anonymes** (:caisse, :catalogue). A comprendre la caisse courante, ou le catalogue courant...

Les objets **nommés** (art : article). A comprendre que c'est bien le type d'article qui correspond au code cherché.

Les multiobjets (: article). A comprendre comme la collection des types d'article associée au catalogue.



b) la syntaxe des messages

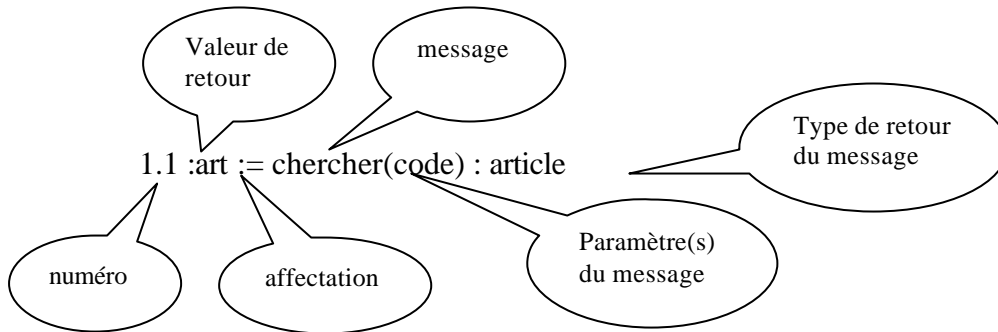


Le message initial est non numéroté par convention. Ce message est un des événements issu du diagramme de séquence boîte noire, dont nous avons fait un contrat d'opération.

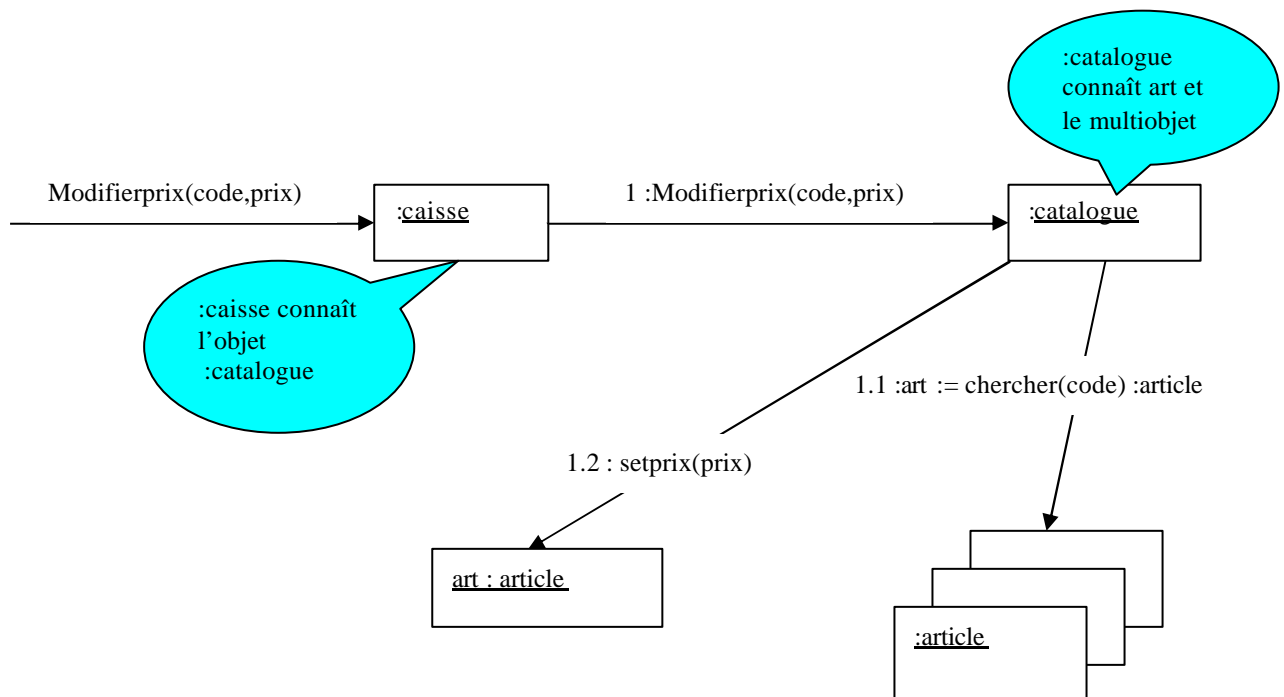
Les opérations effectuées en réponse à ce message seront numérotées de 1 à n (notons qu'en général n n'est pas très élevé, si la conception est bien faite).

Les opérations effectuées en réaction à un message numéro x seront numérotées de x.1 à x.n. Et ainsi de suite pour les opérations effectuées en réaction à un message numéro x.y (x.y.1 à x.y.n).

Un message a la forme suivante :



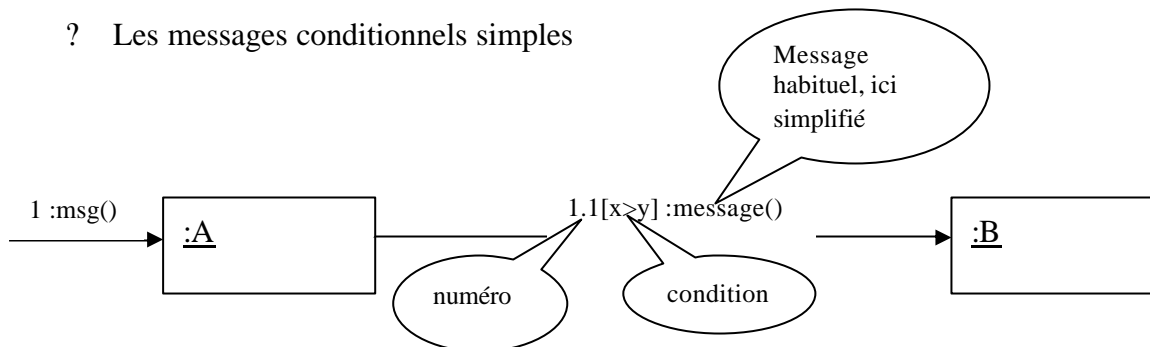
Un lien entre deux objets est directionnel. La flèche indique le receveur du message. Ce lien implique que l'objet qui envoie le message connaît celui qui le reçoit. Un objet ne peut en effet envoyer un message qu'à un objet qu'il connaît (rappelez-vous que l'on envoie un message à un objet en lui parlant : moncatalogue , modifie le prix de l'article de code moncode à monprix.). Chaque flèche implique une visibilité orientée entre les objets.



c) les types de messages

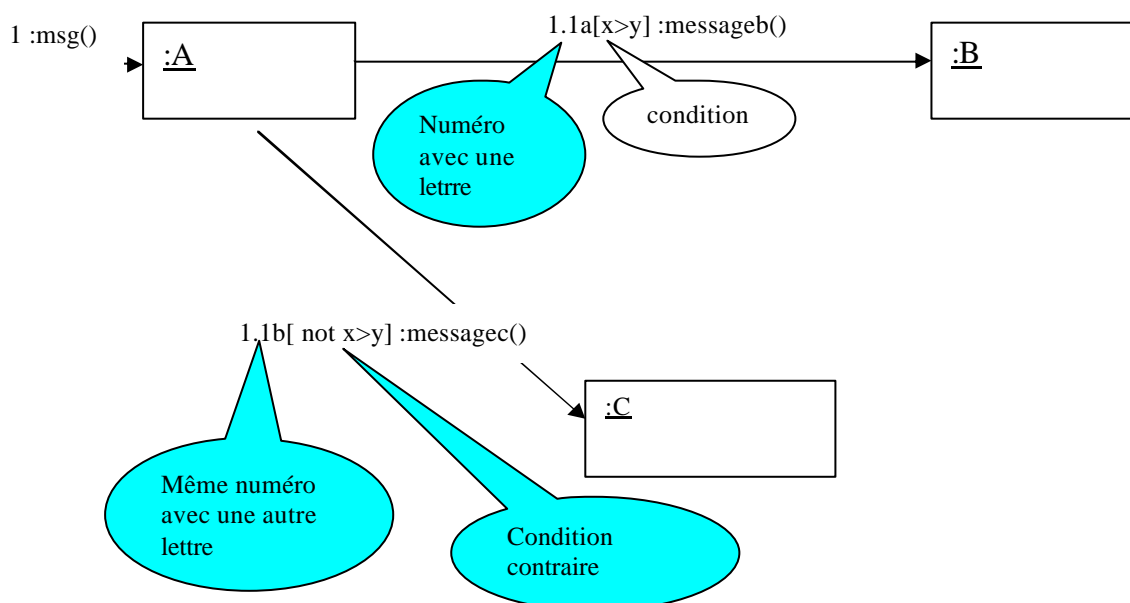
Nous avons vu la forme générale des messages. Il peut y avoir quelques formes particulières de messages, que nous allons lister maintenant.

? Les messages conditionnels simples



Le message n'est envoyé à **:B** que si la condition $x > y$ est remplie lors du déroulement du code de **msg()** dans la classe **A**.

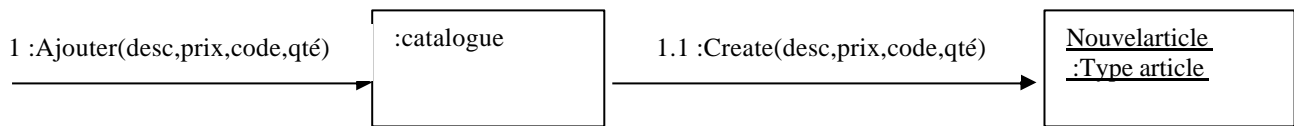
? Les messages conditionnels exclusifs (ou si sinon)



Si la condition $x > y$ est vérifiée, un message est envoyé à l'objet **:B**, sinon un message est envoyé à l'objet **:C**

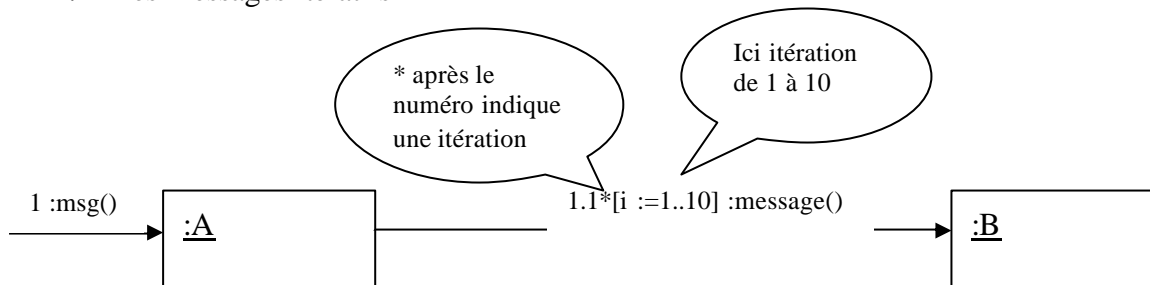
? Les messages d'instanciation

Ce sont les messages de création d'objet. Ce sont des messages **create**, avec ou sans paramètres, qui créeront de nouvelles instances d'objets.



Ici le message create crée un nouvel article en l'initialisant. Les vérifications d'usage seraient bien sûr à effectuer (le code de l'article n'existe t'il pas déjà au catalogue ?).

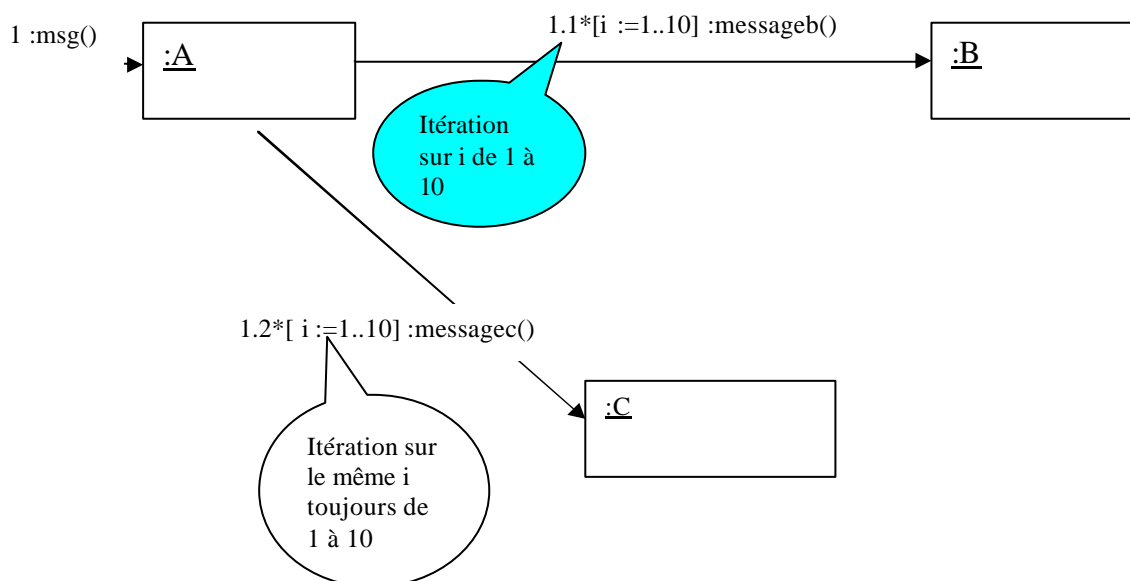
? Les messages itératifs



Le message sera envoyé dix fois à l'objet :B. De manière générale l'étoile placée après le numéro désigne une itération. Nous trouverons soit une énumération de l'itération, comme ici, une condition de continuation également (1.1*[not condition] :message()). Nous trouverons ultérieurement une itération sur tous les éléments.

? Les messages itératifs groupés

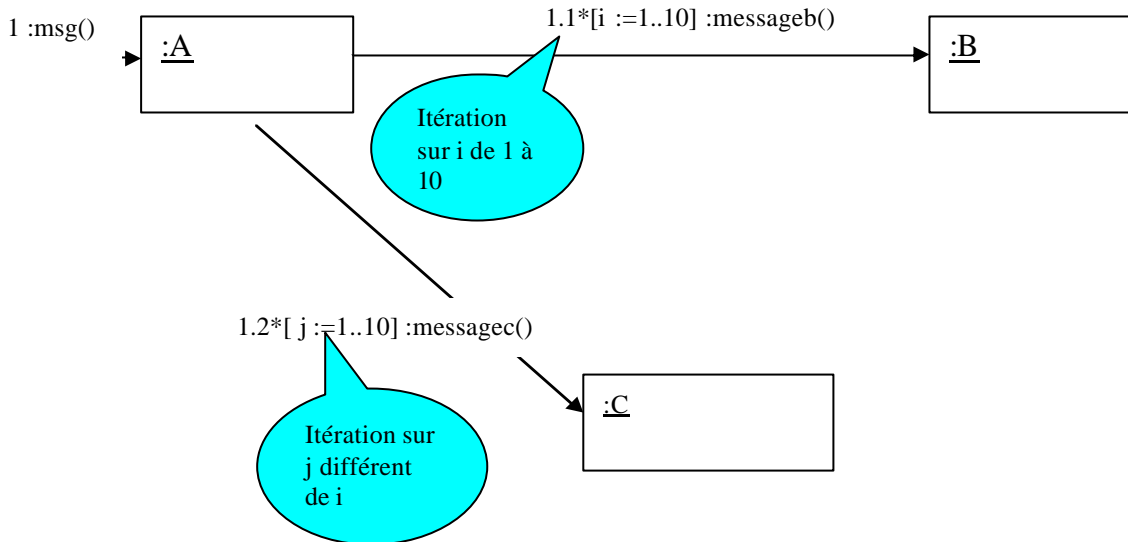
Ce sont des messages itératifs, où plusieurs messages sont envoyés dans l'itération.



Ici nous envoyons successivement un message à :B, puis un message à :C, le tout 10 fois. Il n'y a qu'une seule boucle.

? Les messages itératifs distincts

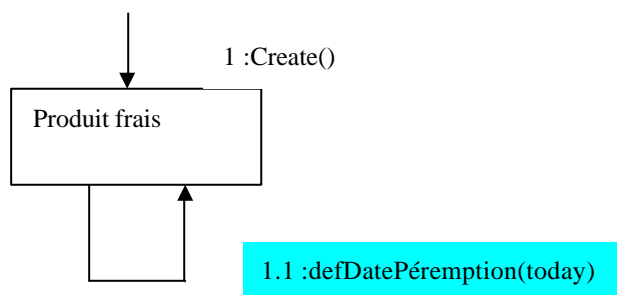
Ce sont des messages itératifs, où plusieurs itérations se suivent. Ici les boucles sont distinctes.



Ici nous envoyons successivement dix messages à :B, puis dix messages à :C. Il y a deux boucles distinctes.

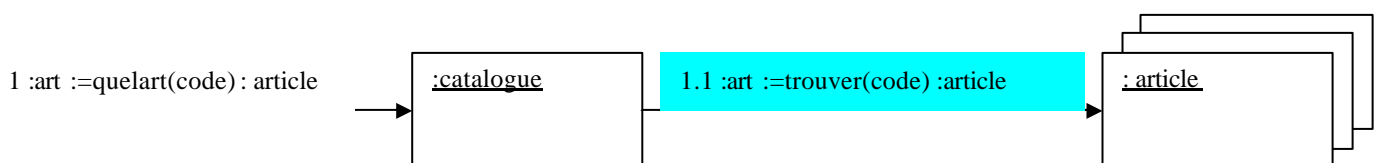
? Les messages vers this

Ici un objet s'envoie un message à lui-même.



La définition de la date de péremption du produit est faite par lui-même. Il sait combien de temps il peut se conserver dans des conditions de températures normales. Donc il s'envoie le message à lui-même.

? Les messages vers un multiobjet

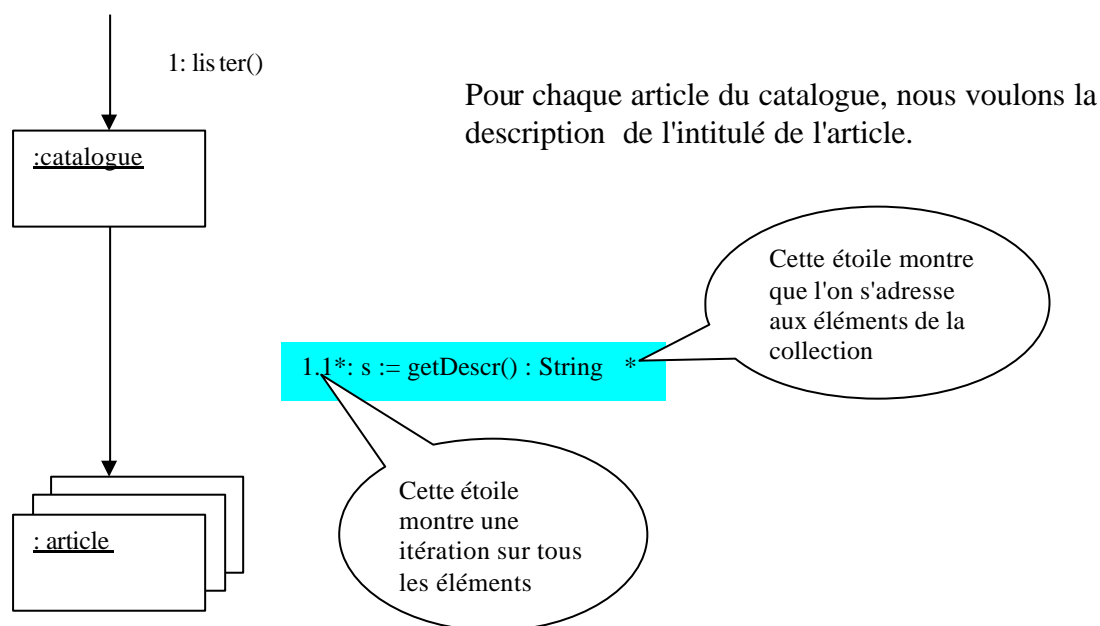


Le message trouver n'est pas envoyé à l'objet `:Type art` mais au multi objet, qui se traduira en programmation par une collection (tableau, vecteur, hashtable, etc).

Les multiobjets acceptent de manière générale un certain nombre de messages:

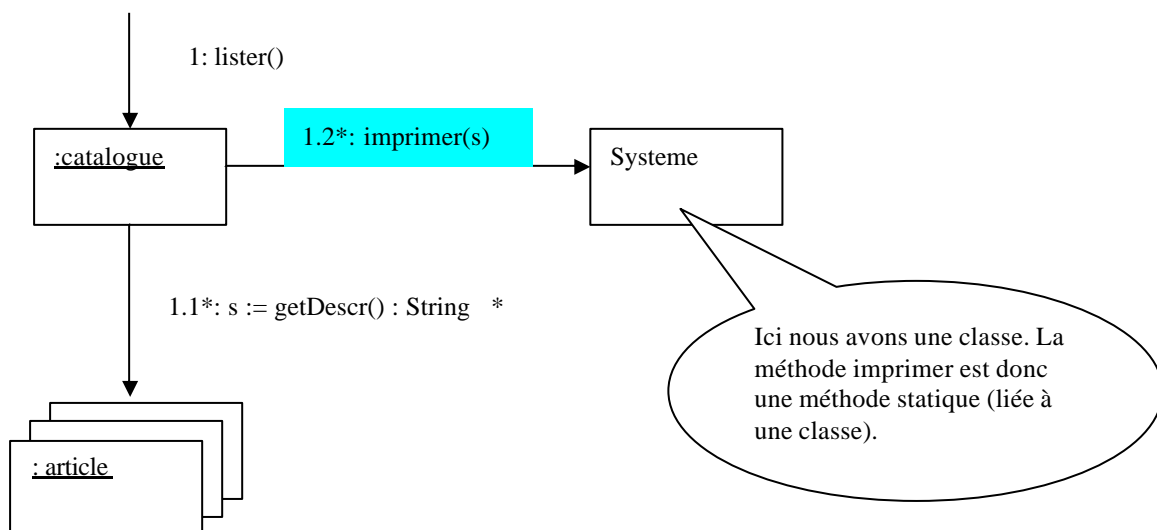
- ? Trouver : récupère un élément d'un multiobjet à partir d'une clé.
- ? Ajouter : ajoute un élément au multiobjet.
- ? Supprimer : supprime un élément du multiobjet.
- ? Taille : donne le nombre d'éléments du multiobjet.
- ? Suivant : permet de passer à l'élément suivant du multiobjet.
- ? Contient : permet de savoir si un élément est présent dans le multiobjet à partir d'une clé.

? Itération sur les éléments d'un multiobjet



Il nous reste ici à imprimer la description obtenue. Pour cela il faut envoyer un message à une classe.

? Envoi d'un message à une classe (appel d'une méthode statique)



2) Visibilité des objets

Pour pouvoir s'échanger des messages, les objets doivent se connaître.



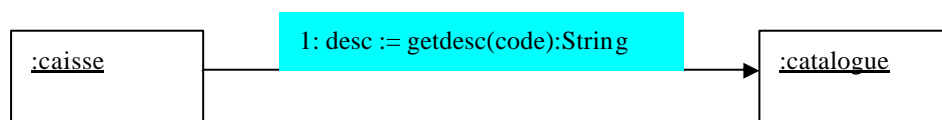
La classe caisse doit connaître la classe vente pour pouvoir lui parler : "vente , oh ma vente! Ajoute-toi un article de ce code."

La visibilité entre les objets n'est pas automatique. Il y a quatre manières différentes pour un objet d'en connaître un autre.

- ✍ La visibilité de type attribut.
- ✍ La visibilité de type paramètre
- ✍ La visibilité de type locale
- ✍ La visibilité de type globale

Nous allons regarder chacun de ces différents types de visibilité.

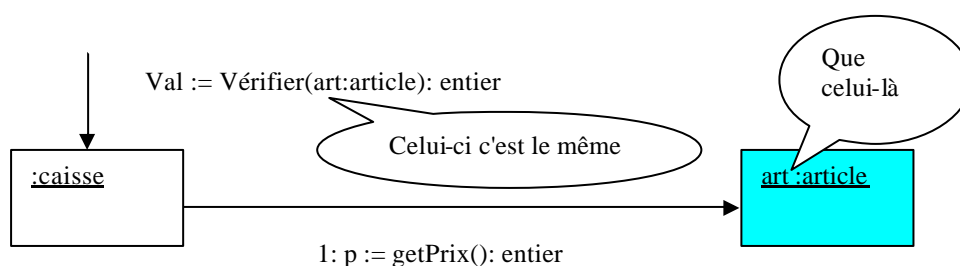
- ✍ La visibilité de type attribut.



La caisse doit connaître de manière permanente le catalogue. Elle a donc un attribut qui référence le catalogue. Le lien montre cet attribut, car c'est la seule manière ici de connaître la classe catalogue.

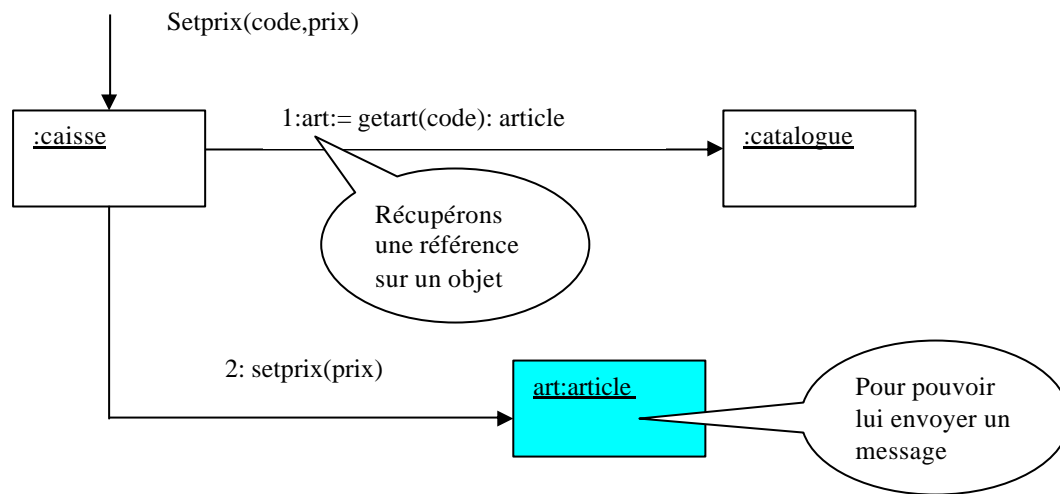
- ✍ La visibilité de type paramètre

Supposons que le diagramme de séquence boîte noire mette en évidence un message de type vérifier avec en paramètre une description d'article. Supposons également que ce soit la caisse qui traite ce message. Nous obtenons le diagramme de collaboration suivant:



Ici la caisse ne connaît l'article art que temporairement. Le passage de paramètre lui fait connaître l'article art à qui la caisse va envoyer un message.

✍ La visibilité de type locale



Ici caisse va chercher une référence sur un article, pour pouvoir envoyer un message à cet article. Ici aussi, la connaissance de l'objet est temporaire, mais elle se fait par une variable locale.

✍ La visibilité de type globale

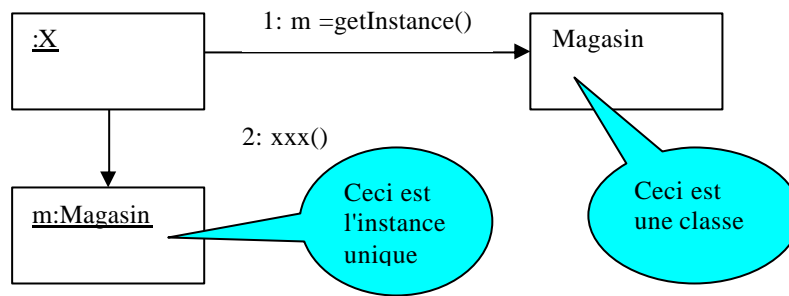
C'est l'utilisation par un objet d'une référence globale à un objet connu de tous. Cela peut aussi être le cas de variables globales dans le cas de certains langages. Nous comprenons bien que ce type de visibilité sera utilisé dans de rares cas, où un objet est omniprésent pour tous les autres objets, et sera considéré comme le contexte de vie de ces objets.

Le problème est que certaines, rares, classes ayant une instance unique doivent être connues de nombreux objets. Il est alors conseillé d'utiliser le GOF Pattern "Singleton".

Supposons qu'une classe nécessite d'en connaître une autre ayant une instance unique (Par exemple, le magasin, si l'on avait une classe magasin...), et que les autres techniques de visibilité soient notoirement malcommodes voici ce que vous pouvez faire:

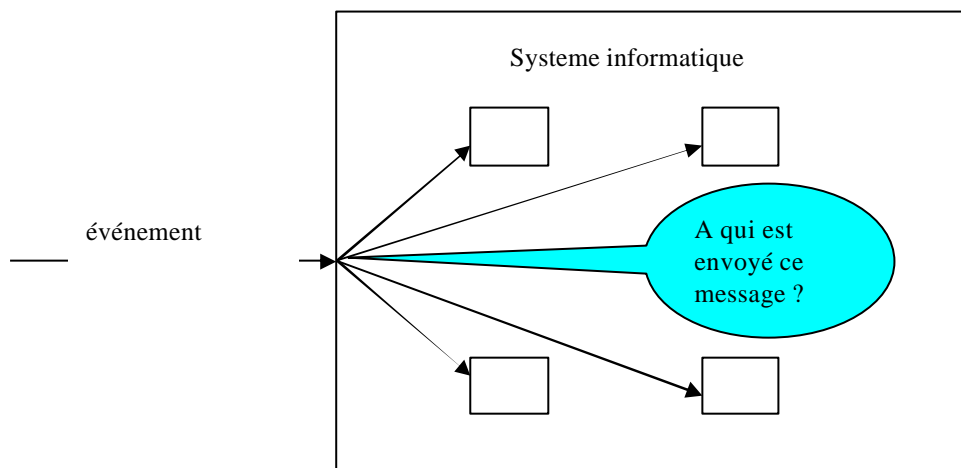
- ✍ La classe magasin aura une donnée membre statique instance.
- ✍ A la création (constructeur) du magasin la donnée membre sera initialisée à this (l'instance nouvellement créée du magasin).
- ✍ La classe magasin aura une fonction statique getInstance qui retournera l'instance du magasin. Ainsi n'importe quelle classe pourra connaître l'objet magasin existant (car la méthode étant statique, elle est envoyée à la classe elle même). Ainsi l'autre classe pourra envoyer sa requête à l'objet magasin.

Voici un diagramme de collaboration qui illustre cet exemple.



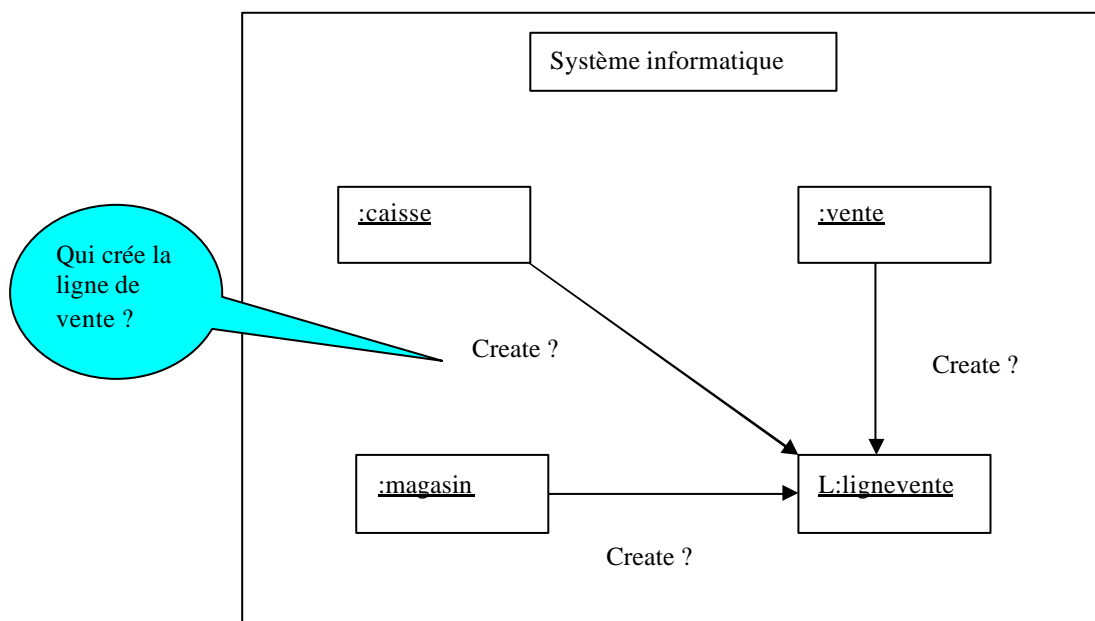
3) GRASP patterns

Quand un événement est envoyé au système informatique, rien n'indique quelle classe prend en charge l'événement et le traite en déroulant les opérations associées.



De même pour chaque opération permettant de mettre en œuvre le contrat d'opération, il faudra déterminer qui crée tel objet, qui envoie tel message à tel autre objet.

Contrat d'opération: une ligne de vente a été créée.



La réponse à ces questions va influencer énormément la conception de notre application. C'est ici que l'on va définir concrètement la responsabilité des objets, leur rôle. C'est aussi ici que l'on va mettre en place la structure du logiciel par couches (entre autre en implémentant les passerelles étanches entre les couches).

Les qualités du logiciel qui lui permettront de vivre, d'être corrigé, d'évoluer, de grandir dépendront complètement des choix que l'on va effectuer ici.

Les spécialistes de la conception objet, après avoir vécu quelques années de développement anarchique, puis après avoir testé l'apport de quelques règles de construction, ont fini par définir un certain nombre de règles pour aider le concepteur dans cette phase capitale et difficile. Ces règles sont issues de l'expérience d'une communauté de développeurs. Ce sont des conseils, ou des règles qui permettent de définir les responsabilités des objets. Ce sont les modèles d'assignation des responsabilités ou GRASP Patterns (General Responsibility Assignment Software Patterns).

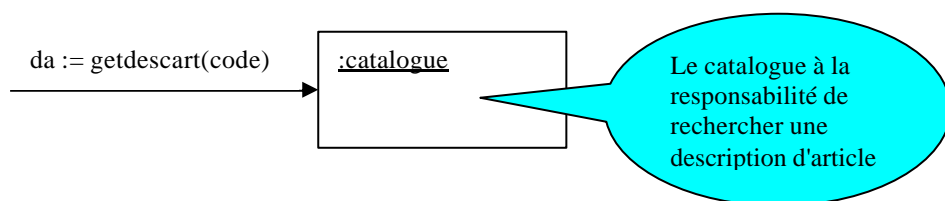
To grasp (en anglais) veut dire: saisir, comprendre, intégrer. Il est fondamental d'intégrer ces modèles avant de faire de la conception objet, pour obtenir des diagrammes de classe de conception, et des diagrammes de collaboration de qualité.

Il y a neuf grands principes pour attribuer les responsabilités aux objets, et modéliser les interactions entre les objets. Cela permet de répondre aux questions:

- ✍ Quelles méthodes dans quelles classes?
- ✍ Comment interagissent les objets pour remplir leur contrat?

De mauvaises réponses conduisent à réaliser des programmes fragiles, à faible réutilisation et à maintenance élevée.

Quand, dans le diagramme de collaboration, un objet envoie un message à un autre objet, cela signifie que l'objet recevant le message a la responsabilité de faire ce qui est demandé.



Un message implique une responsabilité.

Les patterns sont là pour nous aider lors de la conception. C'est un couple problème solution issu de la pratique des experts.

Nous allons voir les neuf GRASP patterns. Il y a d'autres patterns qui existent (par exemple GOF (Gang Of Four) patterns) ceux-là sont plus dédiés à des solutions à des problèmes particuliers (par exemple le modèle de traitement des événements (GOF patterns) qui a inspiré le modèle java de traitement des événements).

Les GRASP patterns sont des modèles généraux de conception, et doivent être considérés par le concepteur objet comme la base de son travail de conception.

Nous allons étudier chacun de ces patterns.

3.1) Faible couplage

Le couplage entre les classes se mesure : c'est la quantité de classes qu'une classe doit connaître, auxquelles elle est connectée, ou dont elle dépend.

Plus il y a de couplage, moins les classes s'adaptent aux évolutions. Il faut donc garder en permanence à l'esprit que des liens entre les classes ne seront rajoutés que si nous ne pouvons les éviter.

Un couplage faible mène à des systèmes évolutifs et maintenables. Il existe forcément un couplage pour permettre aux objets de communiquer.

3.2) Forte cohésion

Des classes de faible cohésion font des choses diverses (classes poubelles où sont rangées les différentes méthodes que l'on ne sait pas classer), ou tout simplement font trop de choses.

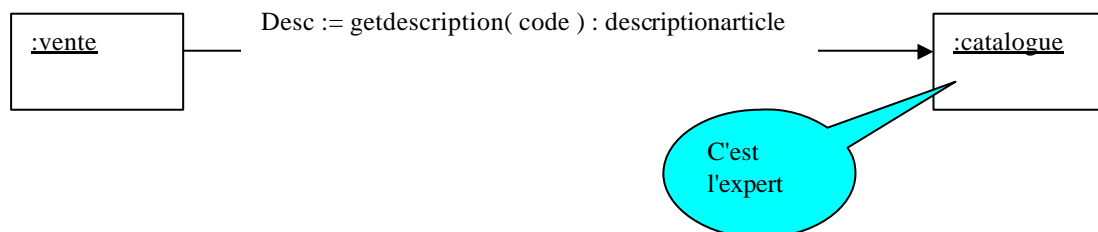
Ces classes à faible cohésion sont difficiles à comprendre, à réutiliser, à maintenir. Elles sont également fragiles car soumises aux moindres variations, elles sont donc instables.

Le programmeur doit toujours avoir ce principe de forte cohésion en tête. Il réalisera alors des classes qui ont un rôle clairement établi, avec un contour simple et clair.

3.3) Expert

Ici nous allons établir qui rend un service. Le principe est que la responsabilité revient à l'expert, celui qui sait car il détient l'information.

Quelle classe fournira le service `getdescription` (code) qui retourne la description d'un article dont nous avons le code?



C'est le catalogue qui possède les descriptions d'article, c'est lui l'expert. C'est donc lui qui nous fournira le service `getdescription`.

Ce principe conduit à placer les services avec les attributs. Nous pouvons aussi garder en tête le principe: "celui qui sait, fait".

3.4) Créateur

Quand une instance de classe doit être créée, il faut se poser la question: " Quelle classe doit créer cet objet?".

Une classe A crée peut créer une instance de la classe B si:

✍ A contient B.

✍ A est un agrégat de B.

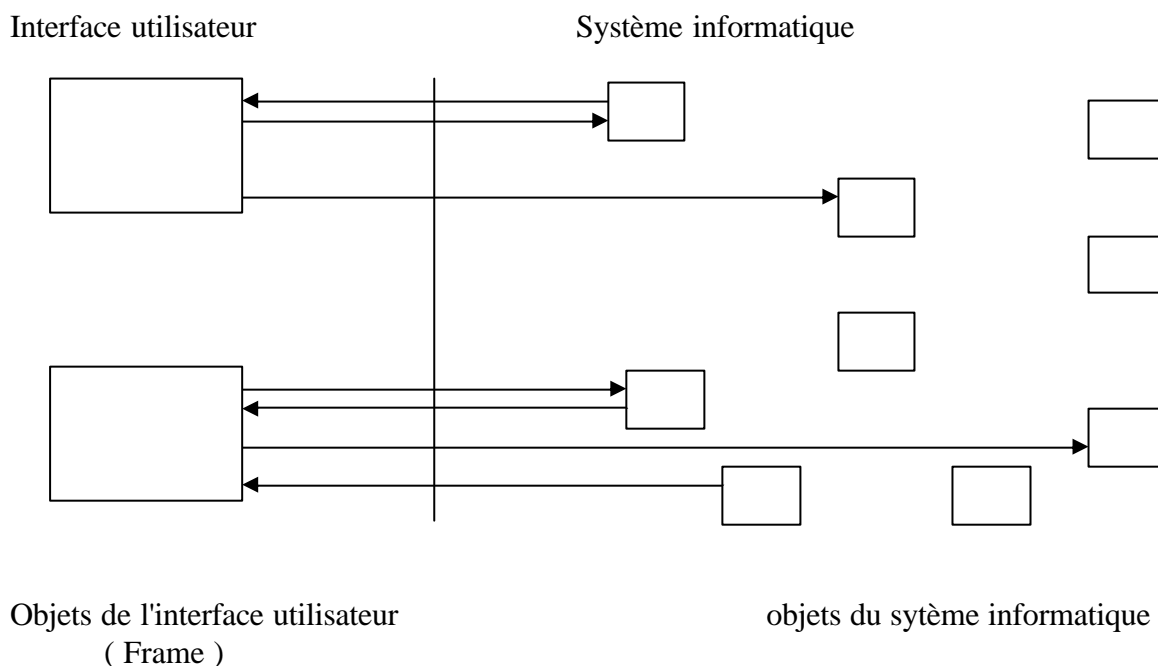
- ✍ A enregistre B.
- ✍ A utilise souvent B.

Alors A est la classe créatrice de B. Il arrive souvent que deux classes soient de bons candidats pour créer une instance. Alors il faut évaluer le meilleur candidat. Cela va dans le sens du faible couplage.

Ici c'est le catalogue qui crée les descriptions d'articles.

3.5) Contrôleur

En début de ce document, il était évoqué l'indépendance entre les différentes couches logicielles. Regardons ici l'interface entre la couche présentation et la couche métier.



Ici nous voyons de fortes dépendances entre les objets du Système informatique et les objets de l'interface. Si l'interface doit être modifiée, il y a de fortes chances qu'il faille également modifier les objets du système informatique.

Notons sur cet exemple un autre problème: les classes du système informatique, ici, connaissent les classes de l'interface utilisateur. Or, ces classes sont liées à un usage particulier (une application) alors que les classes métier sont transverses à toutes les applications. Elles ne peuvent donc pas connaître les interfaces utilisateur. Ce sont les interfaces utilisateurs qui vont chercher les informations des objets métier, et non les objets métier qui affichent les informations.

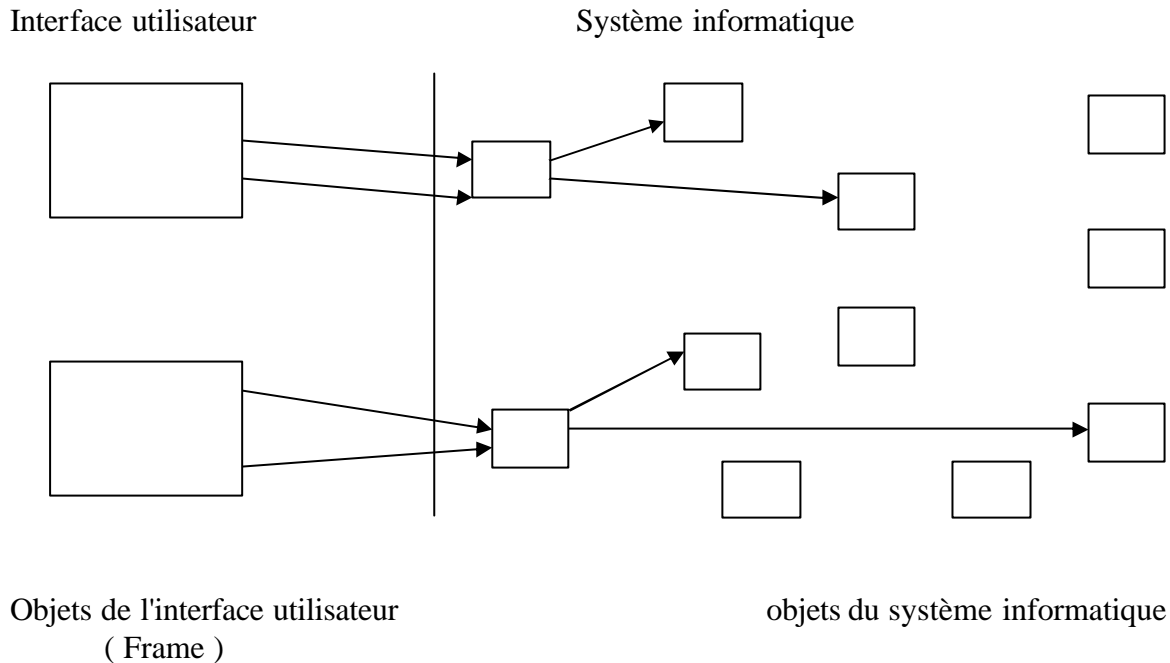
Les objets de l'interface utilisateur vont solliciter un objet d'interface, plutôt que de solliciter les objets métier eux-mêmes. Cet objet s'appelle un contrôleur.

Il peut y avoir quatre sortes de contrôleurs:

- Quelque chose qui représente entièrement le système (caisse).
- Quelque chose qui représente l'organisation ou le métier dans son ensemble (magasin).

- Quelque chose du monde réel qui est actif dans le processus: rôle d'une personne impliqué dans le processus (caissier).
- Handler artificiel pour traiter tous les événements d'un use case. (AchatHandler)

Regardons notre schéma des échanges entre les couches UI et métier.



Le problème est maintenant de savoir comment nous allons choisir notre meilleur contrôleur (il peut y en avoir plusieurs).

L'événement entrerenarticle arrive donc sur une des quatre classes: Caisse, Magasin, Caissier ou AchatHandler.

L'expérience montre que la troisième proposition, appelée contrôleur de rôle, est à utiliser avec parcimonie, car elle conduit souvent à construire un objet trop complexe qui ne délègue pas.

Les deux premières solutions, que l'on appelle contrôleurs de façade sont bien utilisées quand il y a peu d'événements système. La quatrième proposition (contrôleur de use case) est à utiliser quand il y a beaucoup d'événements à gérer dans le système. Il y aura alors autant de contrôleurs que de use cases. Cela permet de mieux maîtriser chaque use case, tout en ne compliquant pas notre modèle objet.

Nous avons peu d'événements à gérer. Nous prendrons donc la solution 1 ou 2. Le choix entre ces deux propositions va se faire en appliquant les patterns précédemment établis.

3.6) Polymorphisme

Quand vous travaillez avec des objets dont les comportements varient lorsque les objets évoluent, ces comportements doivent être définis dans les classes des objets, et les classes doivent être hiérarchisées pour marquer cette évolution.

Les comportements seront définis par des fonctions polymorphes, c'est à dire ayant même forme (même signature ou interface), mais avec des comportements mutants.

Ainsi lorsque l'on sollicite un objet de cette hiérarchie, il n'est pas besoin de savoir quelle est la nature exacte de l'objet, il suffit de lui envoyer le message adéquat (le message polymorphe), et lui réagira avec son savoir faire propre.

Cela permet de faire évoluer plus facilement les logiciels. Un objet mutant (avec un comportement polymorphe) est immédiatement pris en compte par les logiciels utilisant l'objet initial. Un programme ne teste donc pas un objet pour connaître sa nature et savoir comment l'utiliser: il lui envoie un message et l'objet sait se comporter. Cela va dans le sens de l'éradication de l'instruction switch (de JAVA ou de C++).

3.7) Pure fabrication

L'utilisation des différents grasp patterns nous conduit quelque fois à des impasses. Par exemple la sauvegarde d'un objet en base de données devrait être fait par l'objet lui-même (expert) mais alors l'objet est lié (couplé) à son environnement, et doit faire appel à un certain nombre d'outils de base de données, il devient donc peu cohérent.

La solution préconisée, dans un tel cas, est de créer de toute pièce un objet qui traite la sauvegarde en base de données. Notre objet reste alors cohérent, réutilisable, et un nouvel objet, dit de pure fabrication, s'occupe de la sauvegarde en base de données.

Cette solution n'est à employer que dans des cas bien particuliers, car elle conduit à réaliser des objets bibliothèque de fonctions.

3.8) Indirection

L'indirection est le fait de découpler deux objets, ou un objet et un service. La pure fabrication est un exemple d'indirection, mais aussi l'interfaçage avec un composant physique. Un objet ne s'adresse pas directement à un modem, mais à un objet qui dialogue avec le modem.

3.9) Ne parle pas aux inconnus

Pour éviter le couplage, chaque objet n'a le droit de parler qu'à ses proches. Ainsi, nous limitons les interactions entre les différents objets.

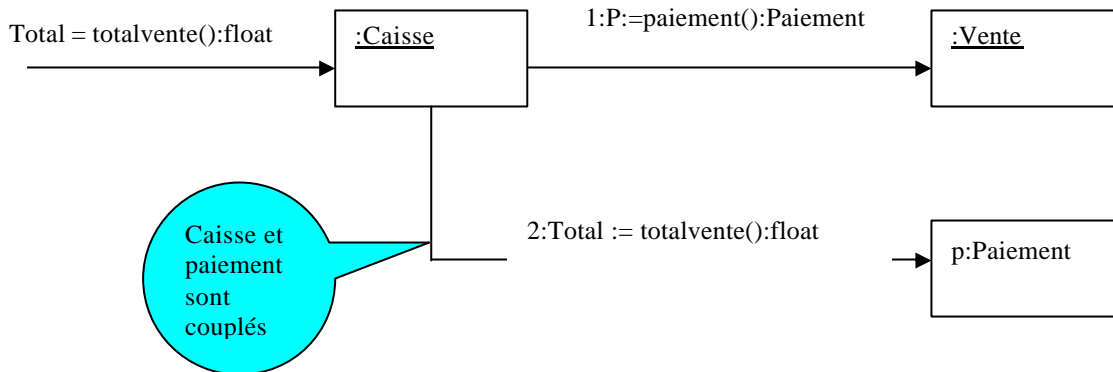
Quels sont les objets auxquels un objet à le droit de parler?

- ✍ Lui-même.
- ✍ Un objet paramètre de la méthode appelée.
- ✍ Un objet attribut de l'objet lui-même.
- ✍ Un objet élément d'une collection attribut de l'objet lui-même.
- ✍ Un objet créé par la méthode.

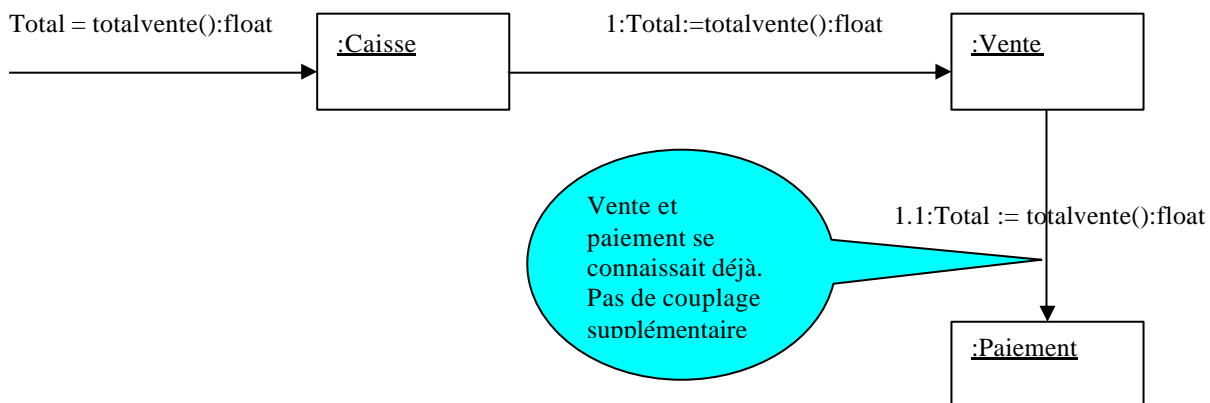
Les autres objets sont considérés comme des inconnus auxquels, nous le savons depuis la plus tendre enfance, il ne faut pas parler.

Prenons un exemple:

L'objet caisse connaît la vente en cours (c'est un attribut de la caisse). Cette vente connaît le paiement (c'est un attribut de la vente). Nous voulons réaliser une méthode de la caisse qui nous donne la valeur de la vente en cours. Voici une première solution:



Cette solution implique que l'objet caisse dialogue avec l'objet paiement. Hors a priori il ne connaît pas cet objet paiement. Pour limiter le couplage entre les objets, il est préférable d'utiliser la solution suivante:



Ici, la caisse ne sait pas comment la vente récupère le total. Des modifications de la structure des objets vente et paiement ainsi que de leurs relations ne change nt rien pour la caisse.

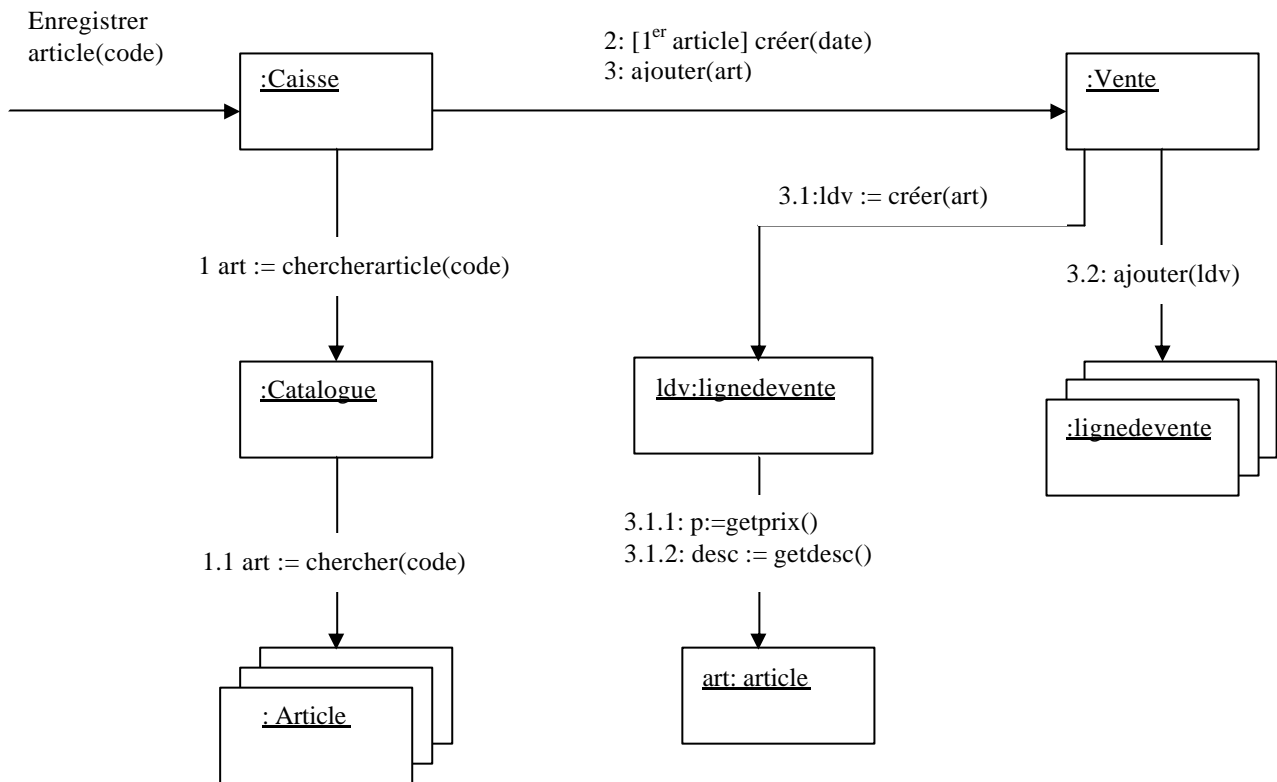
4) Diagramme de collaboration du magasin

Nous allons maintenant construire le diagramme de collaboration pour chacun des contrats d'opération que nous avons détaillés, en tenant compte des modèles de conception.

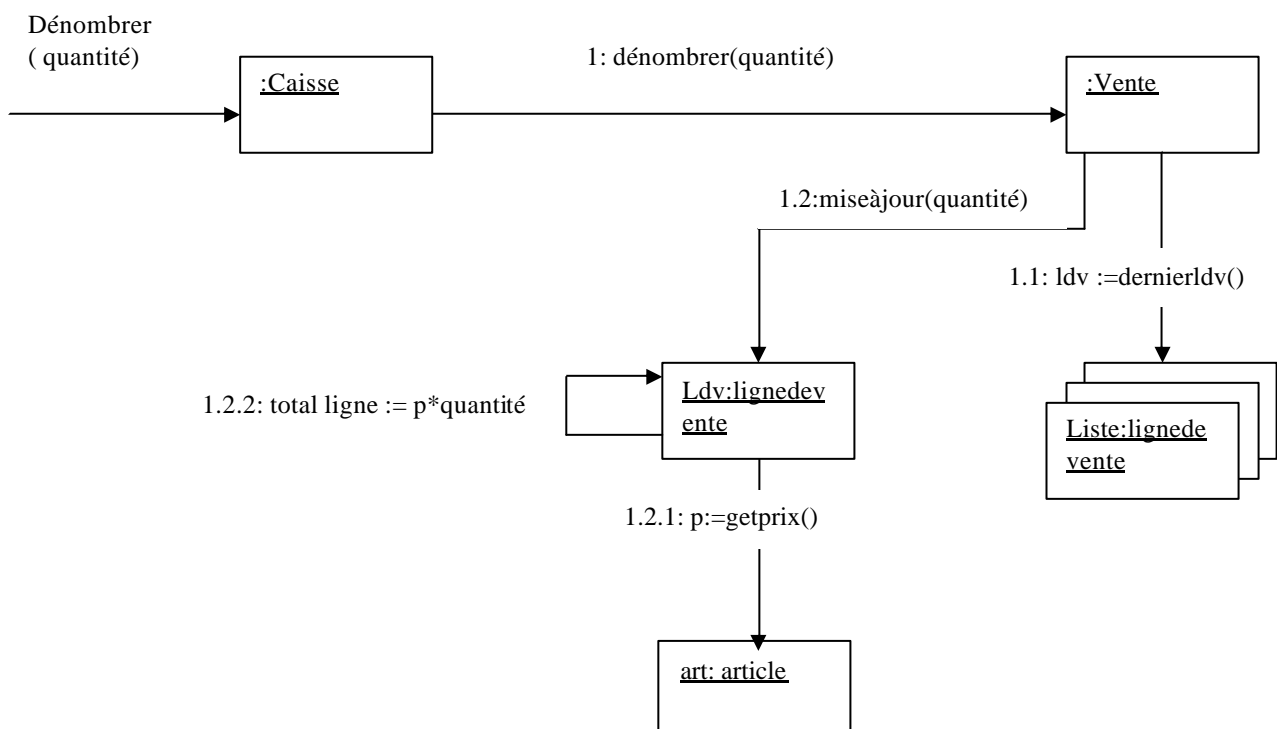
Nous allons faire autant de diagrammes de collaboration que nous avons fait de contrats d'opérations. Pour chaque contrat d'opération nous allons prendre l'événement du système comme message d'attaque de notre diagramme de collaboration, puis nous allons créer les interactions entre les objets qui, à partir de là, permettent de remplir le service demandé. Nous serons vigilant à bien respecter les modèles de conception. Il n'est pas un

message qui se construit au hasard. Chaque message envoyé d'un objet à un autre se justifie par un pattern de conception (au moins) .

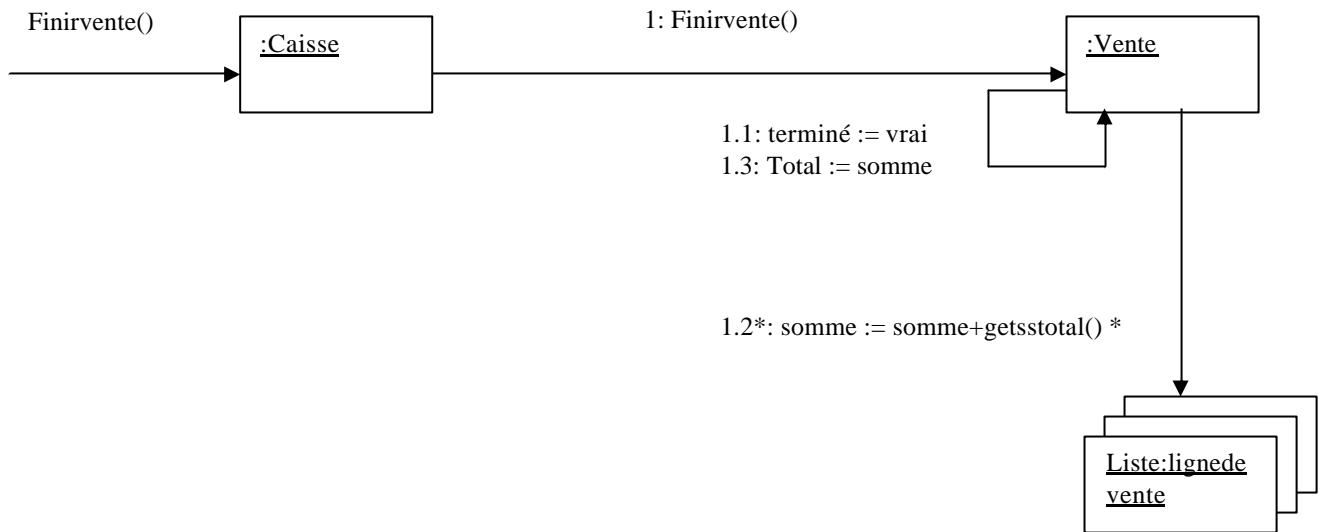
4.1) Diagramme de collaboration de Enregistrer un article



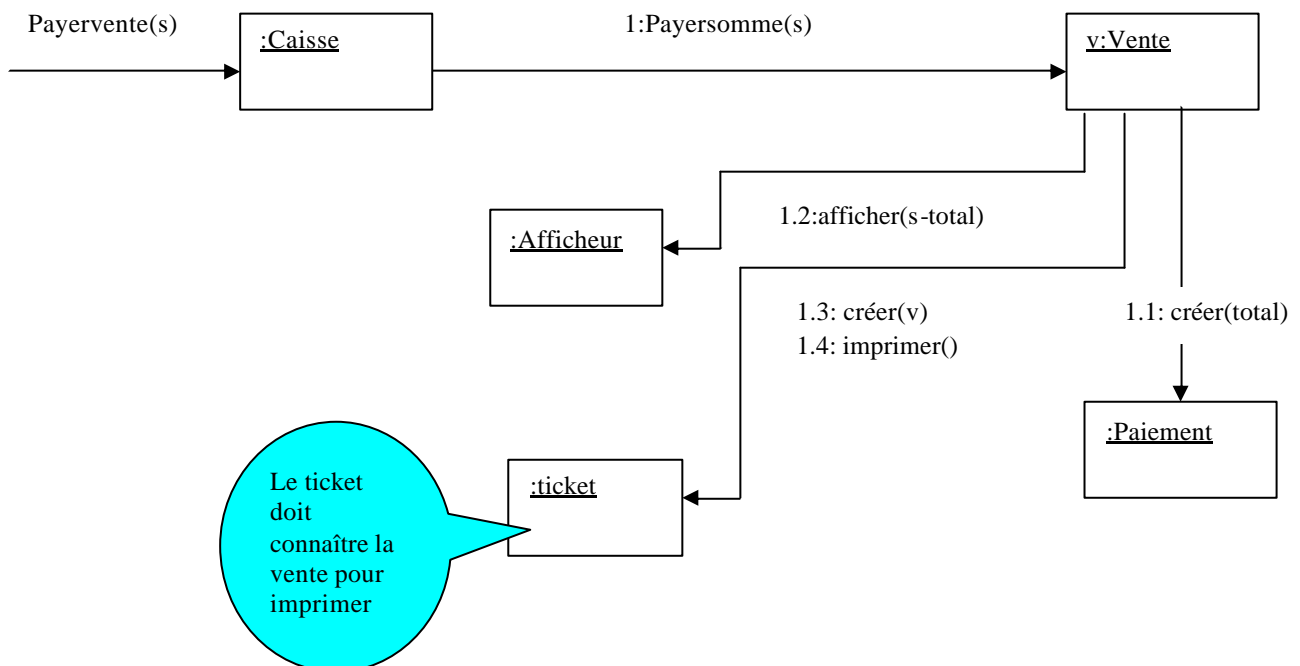
4.2) Diagramme de collaboration de dénombrer les articles identiques



4.3) Diagramme de collaboration de Finir la vente



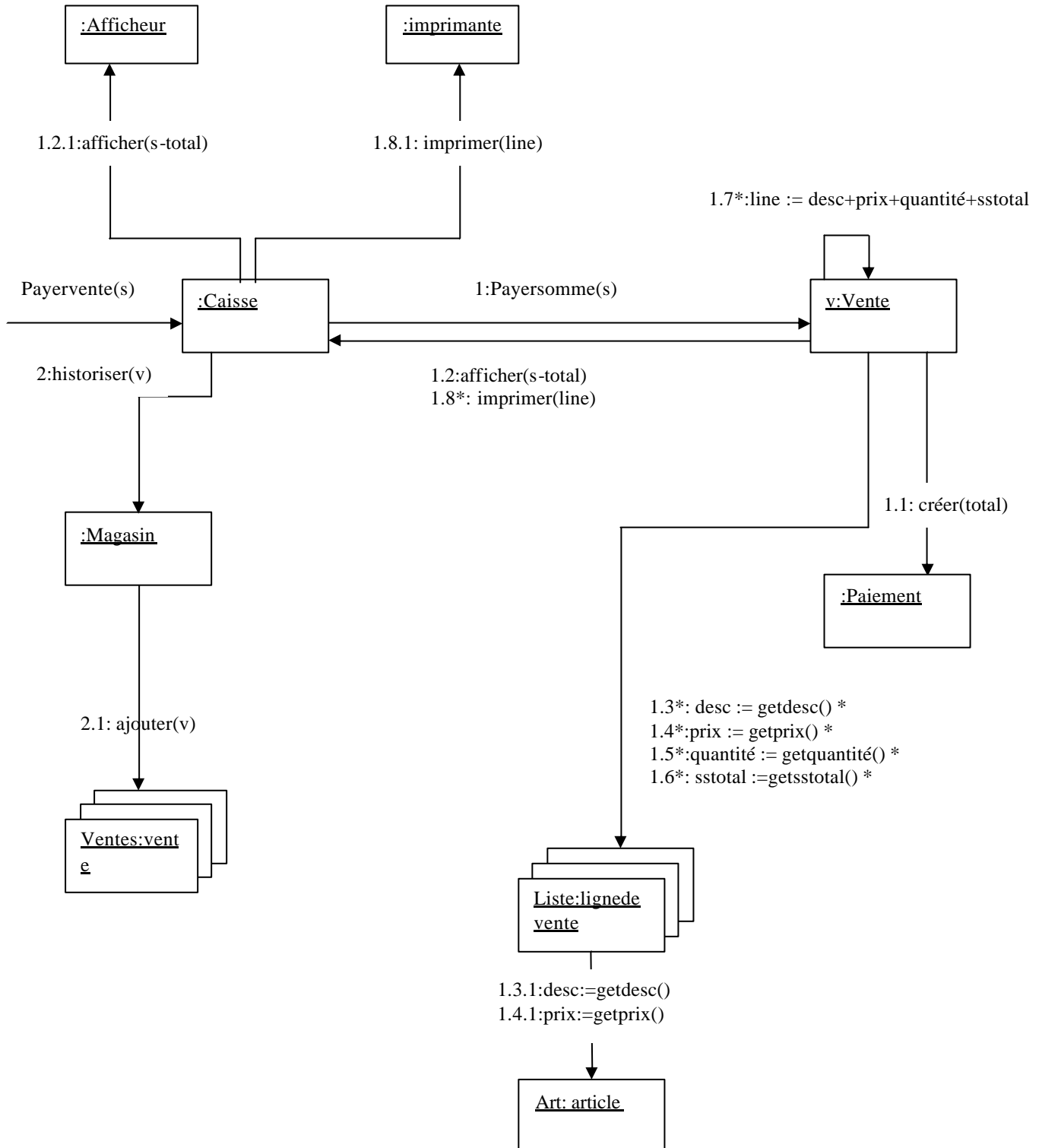
4.4) Diagramme de collaboration de Payer la vente



Les classes ticket et vente sont étroitement couplées. Nous pouvons nous poser la question de savoir si cette classe ticket a un intérêt. Il serait bon de faire l'impression par la vente (l'expert), mais en s'appuyant sur une indirection (imprimante) comme pour l'affichage. Enfin une analyse de tous les uses cases mettrait en évidence le besoin d'archiver les ventes, pour pouvoir faire les statistiques. Nous allons donc proposer une solution plus avancée de ce diagramme de collaboration.

On a ici rajouté les notions d'historisation des ventes (qui découle d'autres use-case), ce qui nous permet de faire apparaître la classe magasin (singleton). Cette classe sera fusionnée avec la classe catalogue.

En ce qui concerne l’affichage, on a choisi d’utiliser un afficheur relié directement à la caisse. Il aurait pu être judicieux de lier l’afficheur à la ligne de vente (pour l’affichage des descriptions, prix, quantité et sous-total associés à la ligne de vente) et à la vente (pour l’affichage du total et de la monnaie à rendre).. Dans ce cas, on aurait pu aussi utiliser le pattern singleton pour modéliser cet afficheur (un objet unique, accessible depuis partout).



Tous les détails sur l’impression du ticket n’y sont pas (entête, total, somme rendue ...), mais ce schéma donne une bonne vue des relations entre les objets.

douzième étape : le diagramme de classe de conception

Nous allons enfin construire le diagramme de classes de conception. C'est le diagramme qui nous montre les classes qui seront développées. C'est donc l'aboutissement de ce travail d'analyse.

Pour l'ensemble des uses cases qui composent notre cycle de développement, nous allons réaliser le diagramme de classes de conception. Nous allons partir des diagrammes de collaboration, qui nous donnent les classes à développer, leurs relations ainsi que les attributs par référence. Nous compléterons ce diagramme par les attributs venant du diagramme de classe d'analyse.

Nous allons partir uniquement du use case effectuer un achat, donc des quatre diagrammes de collaboration du chapitre précédent. Etape par étape, nous allons construire le diagramme de classe de conception.

1) Première étape

Nous allons référencer toutes les classes rencontrées dans les diagrammes de collaboration. Nous allons les dessiner dans un diagramme de classes. Nous allons y ajouter les attributs venant du diagramme de classe d'analyse, et les méthodes venant du diagramme de collaboration.

Nous ne référençons que les classes participant à nos diagrammes de collaboration. Les collections ne sont pas référencées en tant que telles, cela n'apporte pas de plus value.

Les messages vers les collections n'ont pas lieu d'être, les collections supportant ces messages par défaut.

Les messages de création par défaut ne sont pas référencés, s'il n'existe pas de constructeur par initialisation (car alors ils existent forcément).

Les sélecteurs et les modifieurs n'ont pas de raison de figurer dans ce schéma pour ne pas le surcharger. En effet dans la majorité des cas, ces messages existent et sont développés à la construction de la classe.

Caisse
Payervente(s:float) enregistrerarticle(code : Codebarre) dénombrer(quantité : entier) finirvente() afficher(total:float) imprimer(line:text)

Vente
Date : date Terminé : booléen Total : float
payersomme(s:float) Vente(date:Date) ajouter(art: article) dénombrerquantité(q:entier) finirvente()

Magasin
Nom : text Adresse : Adresse
historiser(v:vente) chercherarticle(code:Codebarre): article

Lignevente
Quantité : entier Sstotal : float
getdesc():Text getprix():Float Lignevente(art :article)

Paiement
Total : float
Paiement(total:float)

Descriptionarticle
Description : text Prix : réel Code : codebarre

Afficheur
afficher(stt:float)

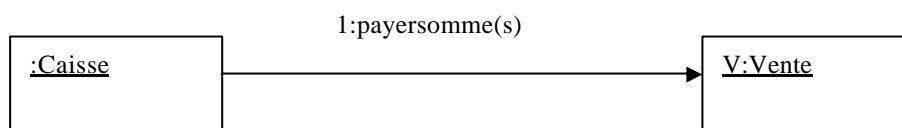
Imprimante
imprimer(line:text)

2) Deuxième étape

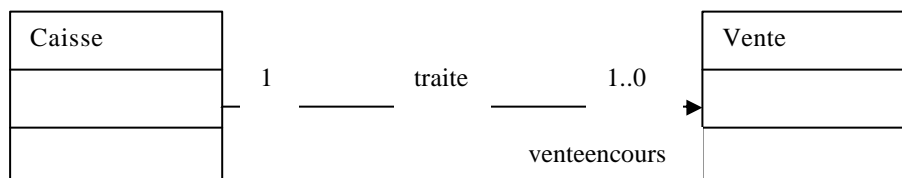
Il faut maintenant ajouter les associations nécessaires pour montrer la visibilité par attribut des objets. Cette visibilité par attribut va permettre de construire les attributs qui référencent les objets que l'on doit connaître. Cette association porte une flèche de navigation qui nous permet de connaître l'objet qui doit connaître l'autre.

Prenons trois exemples :

1) issu du diagramme de collaboration de payervente:

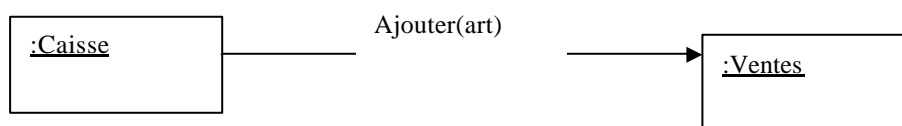


Ici la caisse doit connaître en permanence la vente en cours: nous aurons une association de type attribut.

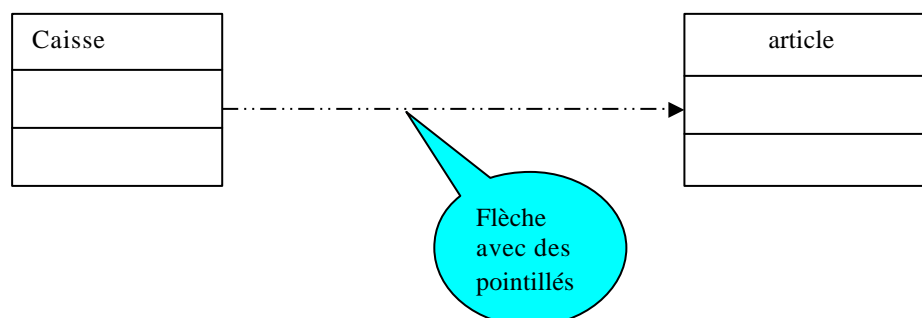


Cela indique que la classe Caisse a un attribut qui référence l'objet vente en cours. Le nom de rôle venteencours donnera, pour une génération automatique de code, le nom de l'attribut qui référence la vente en cours.

2) issu du diagramme de collaboration de enregistrerarticle :

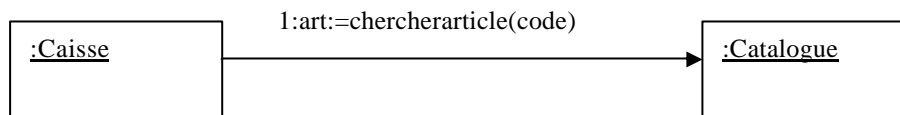


Ici la caisse a une visibilité de type variable locale sur l'article (cela serait le même traitement si il avait une visibilité de type paramètre comme pour la Vente). Nous ajouterons des dépendances de type relation entre les deux classes.



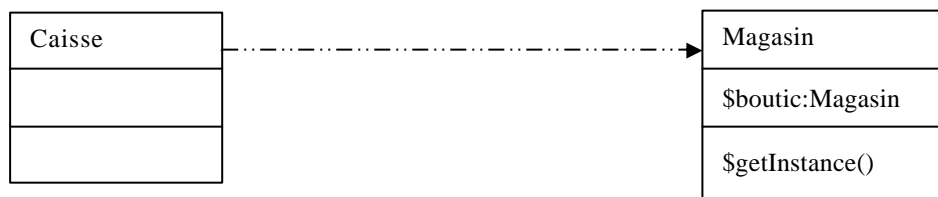
Cette association montre que le code des méthodes de Caisse doit manipuler la classe article. Il sera donc nécessaire, à la génération de code, d'inclure un accès à la classe article dans la classe Caisse (import java, ou include c++).

3) issu du diagramme de collaboration de payerverte :



Ici le magasin est une instance unique pour l'application. Un certain nombre de classes doivent connaître le magasin. Nous pouvons résoudre le problème comme au 1°. Mais, si nous ne voulons pas multiplier les références au magasin, nous pouvons appliquer le pattern Singleton (Singleton pour instance unique).

Cela revient effectivement à travailler avec une instance globale, mais fait proprement, dans des cas rares, et répertoriés. Nous retrouverons la notation de relation entre les classes, ici pour un accès à une variable globale.



Comment est traité le pattern Singleton?

La classe Magasin possède une instance de classe (statique) d'un objet d'elle-même, et une fonction de classe (statique). Le code de la fonction est le suivant:

(exemple en java)

```

public static Magasin getInstance()
{
    if (boutic == null)
    {
        boutic = new Magasin();
    }
    return boutic;
}
  
```

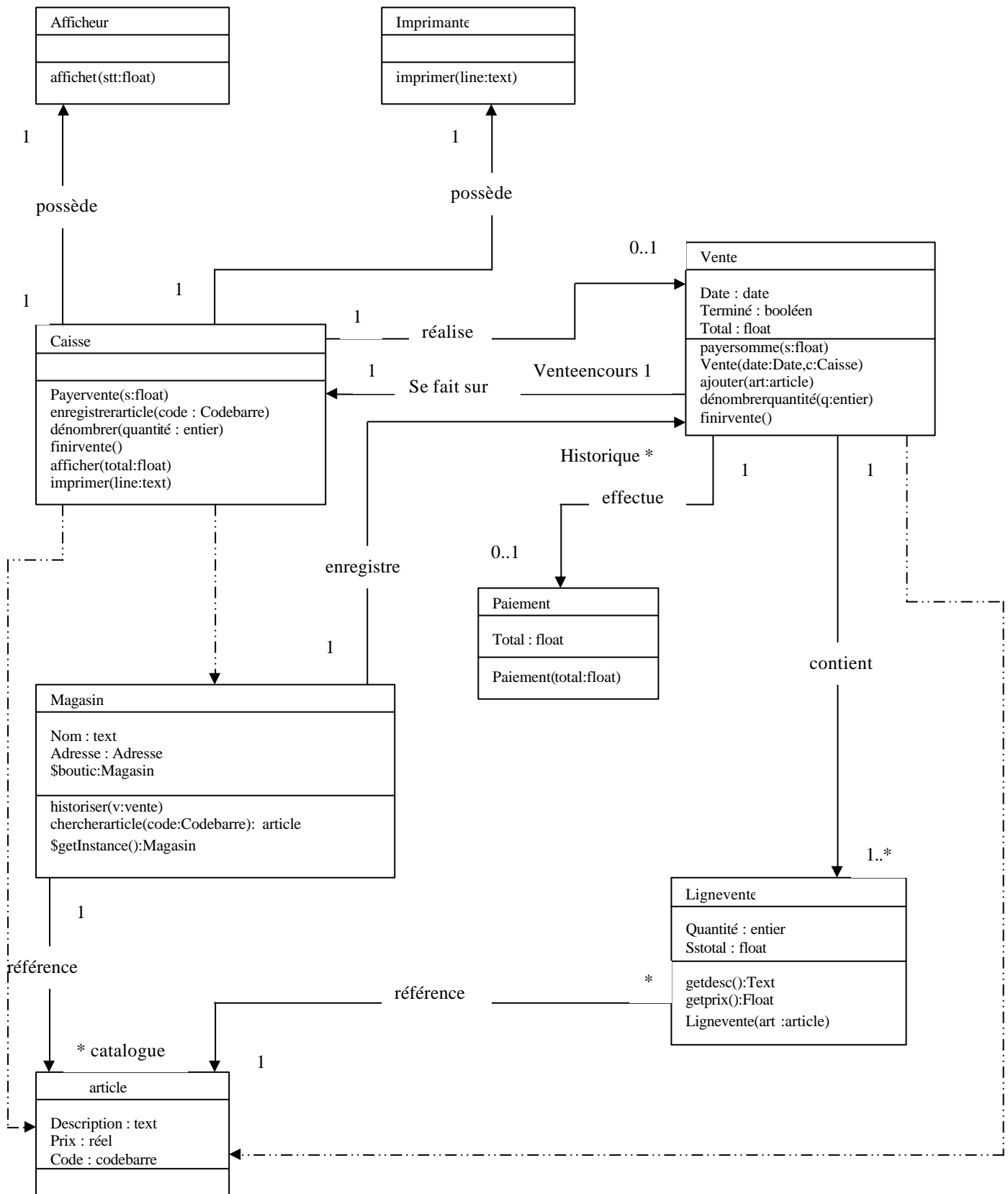
Ainsi nous aurons une instance unique du Magasin. Cette instance peut être appelée partout en utilisant le formalisme suivant:

(exemple en java)

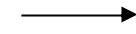
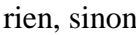
```

Magasin monbazar = Magasin.getInstance();
  
```

Voici maintenant le diagramme de classe de conception de notre exercice:



Notes sur le diagramme de conception:

- ✍ Quand la vente en cours est créée, il faut lui associer la caisse sur laquelle s'effectue la vente. Nous avons fait le choix que la vente ne connaisse que la caisse, plutôt que de connaître chacun de ses éléments (afficheur, imprimante,...)
- ✍ La caisse joue un double rôle: celui d'interface pour le use case effectuer un achat, et l'indirection vers les composants physiques de la caisse. Si la caisse devient trop complexe, il vaut mieux couper la classe en deux, d'un côté l'interface du use case, et de l'autre côté l'interface vers les composants de la caisse elle-même. Nous ne l'avons pas fait dans le cadre de cet exercice.
- ✍ Quand entre deux classes, il existe déjà une association (), il n'est pas utile d'y rajouter une relation (). Cela n'apporte rien, sinon du bruit.
- ✍ Ici, nous n'avons pas rajouté les indicateurs de visibilité (public +, privé -, ...), étant bien entendu que les méthodes sont publiques, et les attributs sont privés. Les fonctions d'accès aux attributs sont sous-entendues.
- ✍ Les noms de rôle mis sur certaines associations donnent le nom de l'attribut dans la classe concernée. Les associations sont unidirectionnelles et donne le sens de la visibilité. Le diagramme nous montre que la classe Caisse a un attribut qui s'appelle venteencours qui référence la vente en cours. De même la classe Magasin a un attribut qui s'appelle catalogue et qui est une collection d'articles (c'est la cardinalité qui nous montre que dans ce cas c'est une collection).
Les trois relations présentées sont des trois types possibles (global, paramètre et local)

treizième étape : le codage

A partir des diagrammes de collaboration et du diagramme de classe de conception, on en déduit simplement des éléments de codage associés. En voici quelques exemples :

```
class Caisse
{
    private Catalogue cat ;
    private Vente v;
    public void enregistrerarticle(int code)
    {
        Article art=Cat.chercherarticle(code);
        if (v==null)
        {
            v=new Vente();
        }
        v.ajouter(art);
    }
    public void finvente()
    {
        v.finvente();
    }
    ....
}
```

```
class Catalogue
{
    private Hashtable articles=new Hashtable(); ;

    public Article chercherarticle(int code)
    {
        return articles.get (code);
    }
    ....
}
```

```
class Vente
{
    private boolean venteterminée = false ;
    private Vector lignes = new Vector();
    public void ajouter(Article art)
    {
        lignes.addElement(new Ldv(art));
    }
    public void finvente()
    {
        venteterminée= true ;
    }
}
```

```

    public float total()
    {
        float total=0 ;
        Enumeration e=lignes.elements() ;
        while (e.hasMoreElements())
        {
            total+=((Ldv)e.nextElement()).soustotal();
        }
        return total;
    }

    ....
}

```

```

class Ldv
{
    private int qte ;
    private Article art ;
    ....
    public Ldv(Article a)
    {
        this.art=a ;
    }
    public void soustotal()
    {
        return qte*art.getprix() ;
    }
    public denommer(int newqte)
    {
        setqte(newqte) ;
    }
    ....
}

```

```

class Article
{
    private int code=0 ;
    private float prix=0 ;
    private String desc= "" ;
    public Article(int code, float prix, String desc)
    {
        this.code=code ;
        this.prix=prix;
        this.desc=desc;
    }
    public int getcode()
    {
        return code;
    }
}

```

```
    }  
    public int getprix()  
    {  
        return prix;  
    }  
    public int getdesc()  
    {  
        return desc;  
    }  
  
    ....  
}
```

Le processus unifié

Le processus unifié est la démarche d'analyse et de conception qui s'appuie sur le langage UML pour sa modélisation. Ce que nous avons vu jusqu'à présent, c'est une démarche linéaire pour partir d'un besoin utilisateur jusqu'à sa réalisation. Ceci n'est pas représentatif de la réalité des développements. Un développement d'un projet informatique va être incrémental et itératif. Cela permet d'éviter l'effet tunnel des projets classiques (le client perd de vue son logiciel entre le cahier des charges et la livraison). Ici le projet est développé en concertation avec le client, le dialogue doit être permanent, les livraisons se font régulièrement, permettant d'intervenir au plus tôt si il y a un écart entre le besoin du client et le logiciel réalisé.

Nous allons donc mettre en 3 dimensions le processus que nous avons étudié.

Notion de lot (ou de cycle)

Un lot, ou un cycle de développement, permet de livrer au client une version de logiciel. Il est important de faire des livraisons régulières du logiciel, pour que le client garde le contrôle du logiciel que nous lui développons. Typiquement un cycle peut durer trois semaines à deux mois. Les exceptions viendront des gros logiciels, où une nouvelle version sort tous les ans.

Comment s'y prendre pour découper un logiciel en lot ? Nous allons lister l'ensemble des uses cases, en les évaluant suivant deux critères : leur importance pour le client, et le risque technique de réalisation, en notant par exemple de un à cinq chacun des critères.

Nous comprenons bien qu'il vaut mieux réaliser d'abord un module de facturation qu'un module d'aide en ligne pour un commerçant. Il est aussi important de réaliser d'abord les exigences à fort risque technique, afin de vérifier la faisabilité technique, et la validité de la solution.

Nous classerons les uses cases par leur pondération (risque + importance). Puis nous donnerons une logique à ce classement. Cela peut nous amener à ne vérifier qu'une faisabilité pour un use case particulier, ou à traiter une partie des exigences d'un use case dans un premier lot, et les autres dans un lot suivant.

Ce découpage sera validé par le client, et nous donnera les différentes livraisons qui seront faites au client.

Notion de phases

Un lot est composé de plusieurs phases. Il y a 4 phases dans le développement d'un lot :

- Inception ou création
- Elaboration
- Construction
- Transition

Nous allons voir en quoi consiste chaque phase, puis nous verrons que chaque phase est constituée d'une ou plusieurs itérations.

Inception : Cette phase sert à prouver la faisabilité.

- ? Effort : 5% du temps total.
- ? Objectif :
 - ? appréhender le contexte du développement.
 - ? Connaître, dans les grandes lignes, les services rendus aux utilisateurs principaux.
 - ? Mettre en évidence les risques, et vérifier qu'ils sont maîtrisés.
 - ? Choisir une architecture.
 - ? Planifier la phase suivante
 - ? Estimer assez précisément les coûts, délais, ressources nécessaires.
- ? Documents produits :
 - ? Principaux cas d'utilisation
 - ? Descriptions de haut niveau de ces uses cases

Elaboration : Cette phase sert à construire l'architecture de base, et à définir la plupart des exigences.

- ? Effort : 20% du temps total.
- ? Objectif :
 - ? Recueillir les exigences.
 - ? Mettre en œuvre les exigences à haut risque.
 - ? Faire une description détaillée des uses cases.
 - ? Créer l'architecture du logiciel.
 - ? Définir les niveaux de qualité requis pour le système (fiabilité, temps de réponse, ...)
 - ? Planifier et budgéter le développement
- ? Documents produits :
 - ? Uses cases détaillés
 - ? Diagrammes de séquences boîte noire.
 - ? Diagramme de classes d'analyse.
 - ? Diagramme de classe de conception, diagrammes de collaboration, et contrats d'opérations sur quelques classes qui permettent de rendre le risque maîtrisable.

Construction : Cette phase sert à bâtir le système.

- ? Effort : 65% du temps total.
- ? Objectif :
 - ? Etendre l'analyse et la conception à l'ensemble des classes non critiques du système.
 - ? Coder ces classes.
 - ? Tester les classes unitairement.
 - ? Préparer les tests du système.
- ? Documents produits :
 - ? Modèle du domaine.

- ? Diagramme de classe de conception, diagrammes de collaboration, et contrats d'opérations sur quelques classes qui permettent de rendre le risque maîtrisable.
- ? Code.
- ? Tests et procédures de test du système.
- ? Surveiller les risques détectés en phases amont.

Transition : Cette phase sert à tester et déployer le produit chez le ou les utilisateurs.

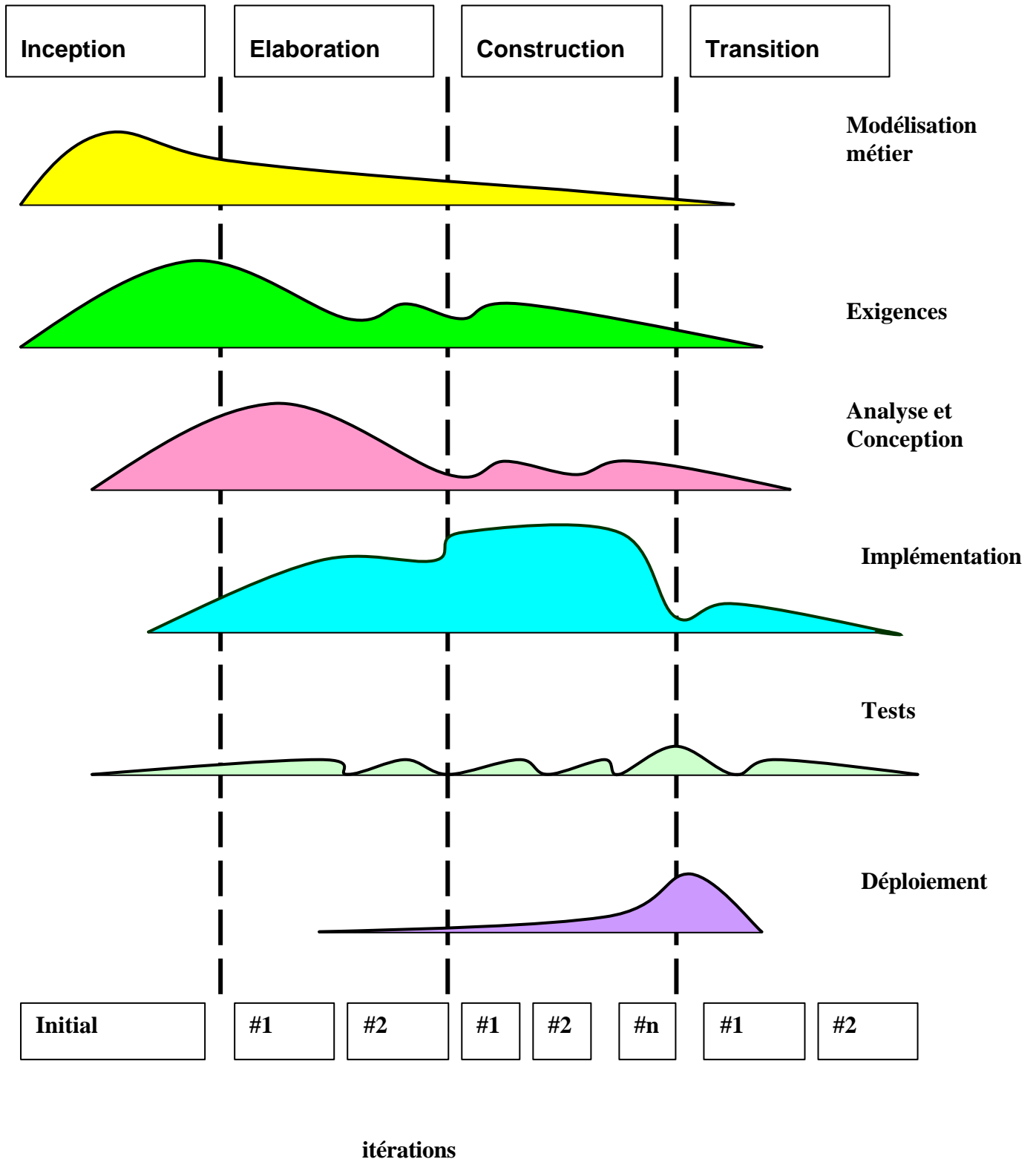
- ? Effort : 10% du temps total.
- ? Objectif :
 - ? faire les bêta tests.
 - ? convertir les données antérieures.
 - ? former les utilisateurs.
 - ? finalisation des manuels utilisateurs.
 - ? corriger les derniers bugs.
- ? Documents produits :
 - ? Diagrammes de composants.
 - ? Diagramme de déploiement.

Notion d'itération

Chacune des phases décrites, peut être effectuée en plusieurs itérations. Par exemple dans la phase de conception, les itérations peuvent être très brèves (de l'ordre de la demi journée, jusqu'à deux jours), et consiste à implémenter une classe par exemple.

Dans la phase d'élaboration, une itération pourra par exemple traiter d'un risque particulier.

phases



conclusion

Nous avons vu qu'une méthode (de type Unified Process) qui s'appuie sur UML, permettait d'avoir une démarche d'analyse et de conception qui nous amène de manière continue du problème du client, vers le système qu'il désire.

Cette méthode est incrémentale et itérative. Le client voit se construire son logiciel petit à petit, peut donc se l'approprier, se former, et mieux adapter le logiciel par rapport à son besoin. Cette méthode permet au client de participer à l'élaboration du logiciel, et permet aux développeurs de prendre en compte les remarques du client sans avoir à remettre en cause tout ce qui a déjà été réalisé.

Elle permet également de se construire des bibliothèques d'objets métiers, qui permettront de réduire les coûts des prochains logiciels à réaliser. Les informaticiens vont enfin pouvoir capitaliser leurs compétences au sein de bibliothèques d'objets.

Il ne vous reste plus qu'à acquérir de la pratique dans cette méthode. Le développeur commence à devenir réellement autonome sur l'ensemble de cette démarche au bout d'un an de pratique, avec un tuteur.

Bon courage.

bibliographie

Applying UML And Patterns Craig LARMAN Prentice Hall

Pour approfondir la pratique d'UML, je conseille de suivre le cours "Analyse et conception avec UML et les Patterns " de la société Valtech.