

Secteur Tertiaire Informatique
Filière étude - développement

Activité Développer une application X-Tiers

Démarche autour d'UML partie 2- ressource

Accueil

Apprentissage

**Période en
entreprise**

Évaluation



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.

SOMMAIRE

I.1	neuvième étape : les contrats d'opération	4
II	La conception	10
II.1	dixième étape : le diagramme d'état	10
II.2	onzième étape : le diagramme de collaboration	11
II.2	onzième étape : le diagramme de collaboration	12
1)	Syntaxe du diagramme de collaboration	12
2)	Visibilité des objets	21
3)	GRASP patterns	23
4)	Diagramme de collaboration du magasin	30
II.3	douzième étape : le diagramme de classe de conception	35
1)	Première étape	35
2)	Deuxième étape	36
III	Le processus unifié	40
III.1	Notion de lot (ou de cycle)	40
III.2	Notion de phases	40
III.3	Notion d'itération	42
IV	conclusion	44
V	bibliographie	45

I.1

NEUVIEME ETAPE : LES CONTRATS D'OPERATION

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:

- Nom: Nom de l'opération avec ses paramètres.
- Responsabilités: Description du rôle de l'opération.
- Exigences: Liste des exigences dont le use case tient compte dans cette itération.
- Notes: Remarques diverses.
- Pré conditions: Etat nécessaire du système pour que l'opération se fasse.
- Post conditions: 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 une description d'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 terminée. *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 terminée.
- 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.

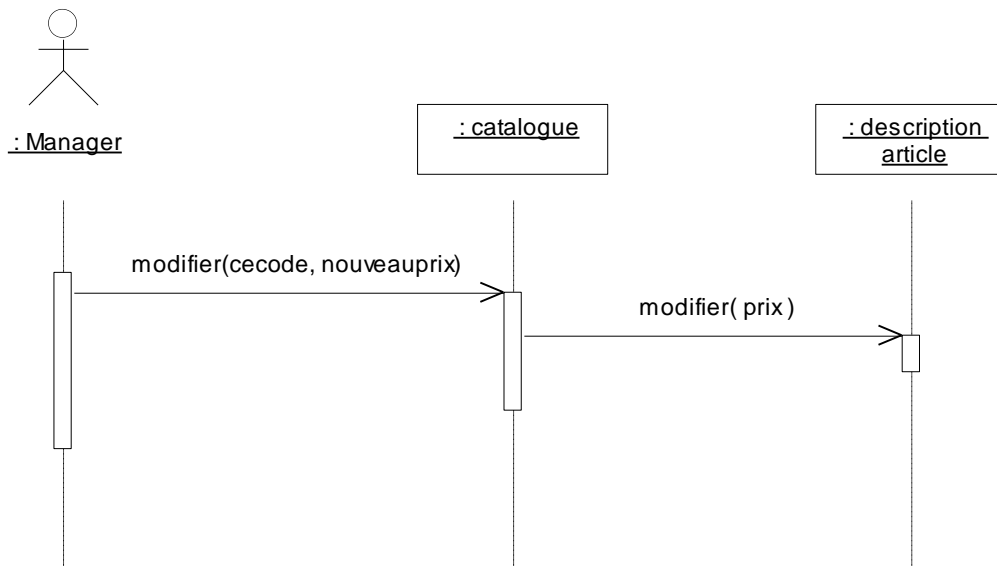
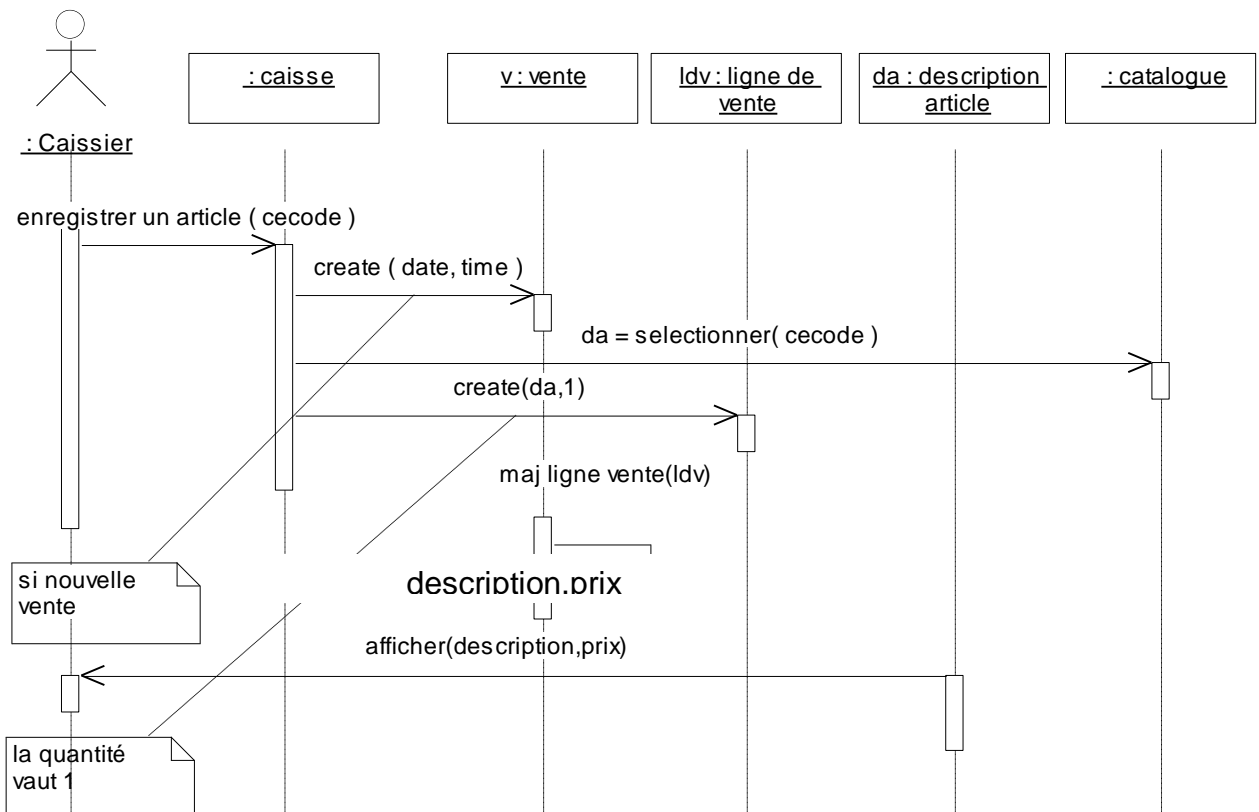


Diagramme de séquence boîte blanche de l'opération modifier le prix d'un article au catalogue dans le use case gérer le catalogue.

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.

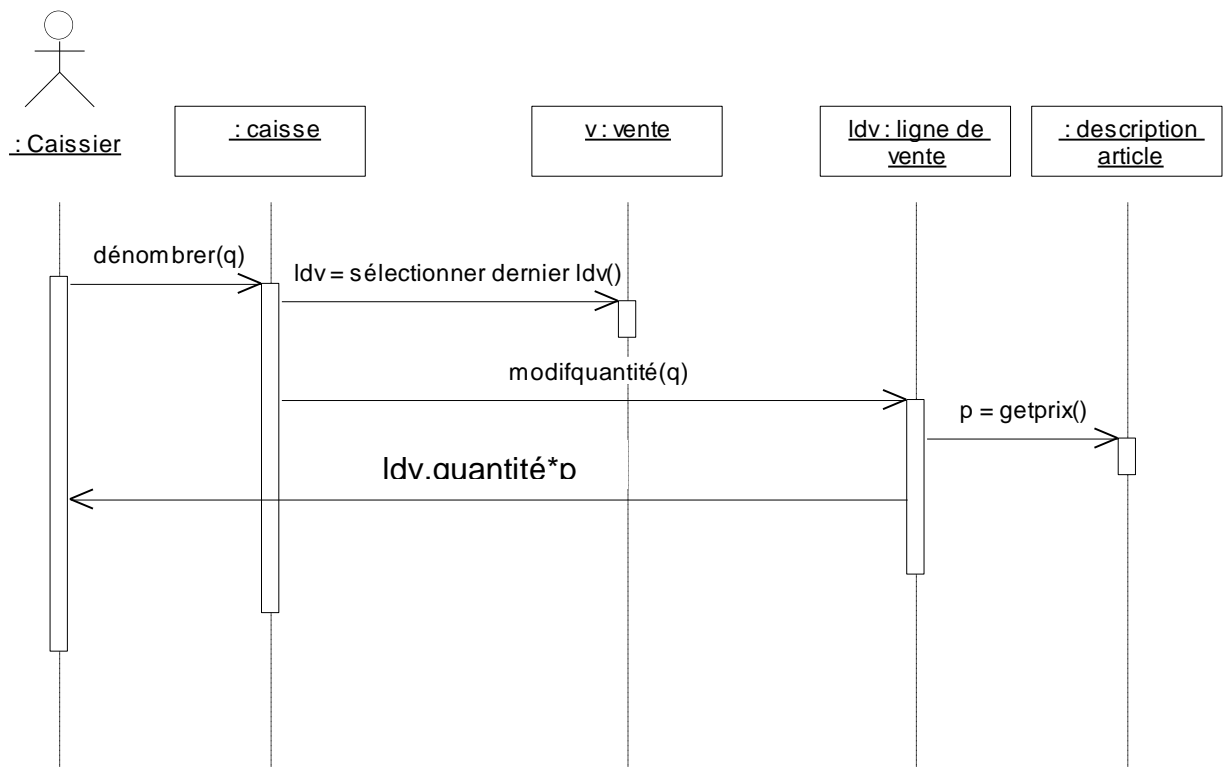


Diagramme de séquence boîte blanche de l'opération finir vente.

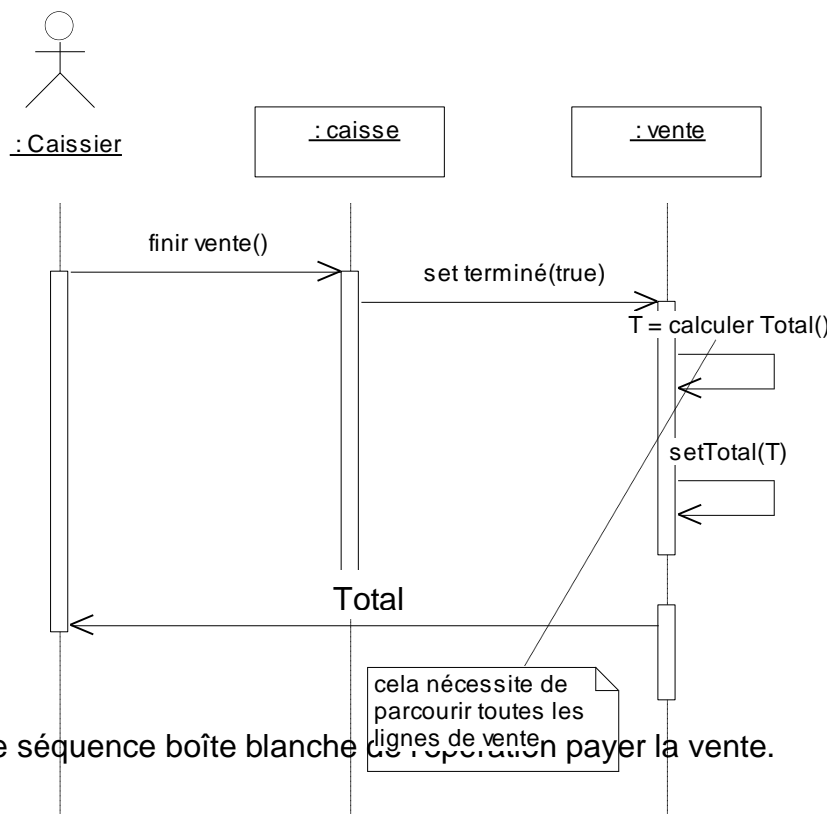
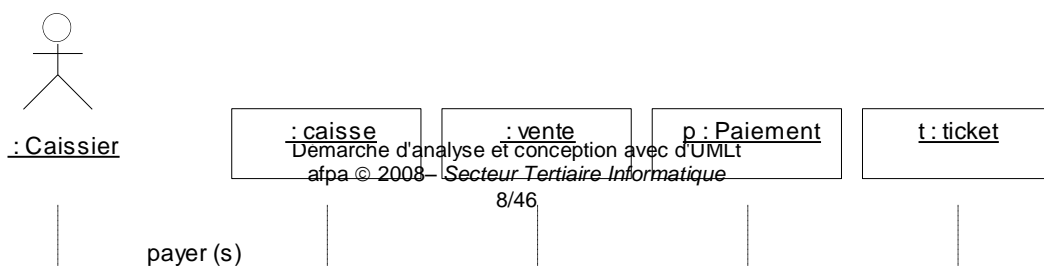


Diagramme de séquence boîte blanche de l'opération payer la vente.



S-

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.

II La conception

II.1 DIXIEME ETAPE : LE DIAGRAMME D'ETAT

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.

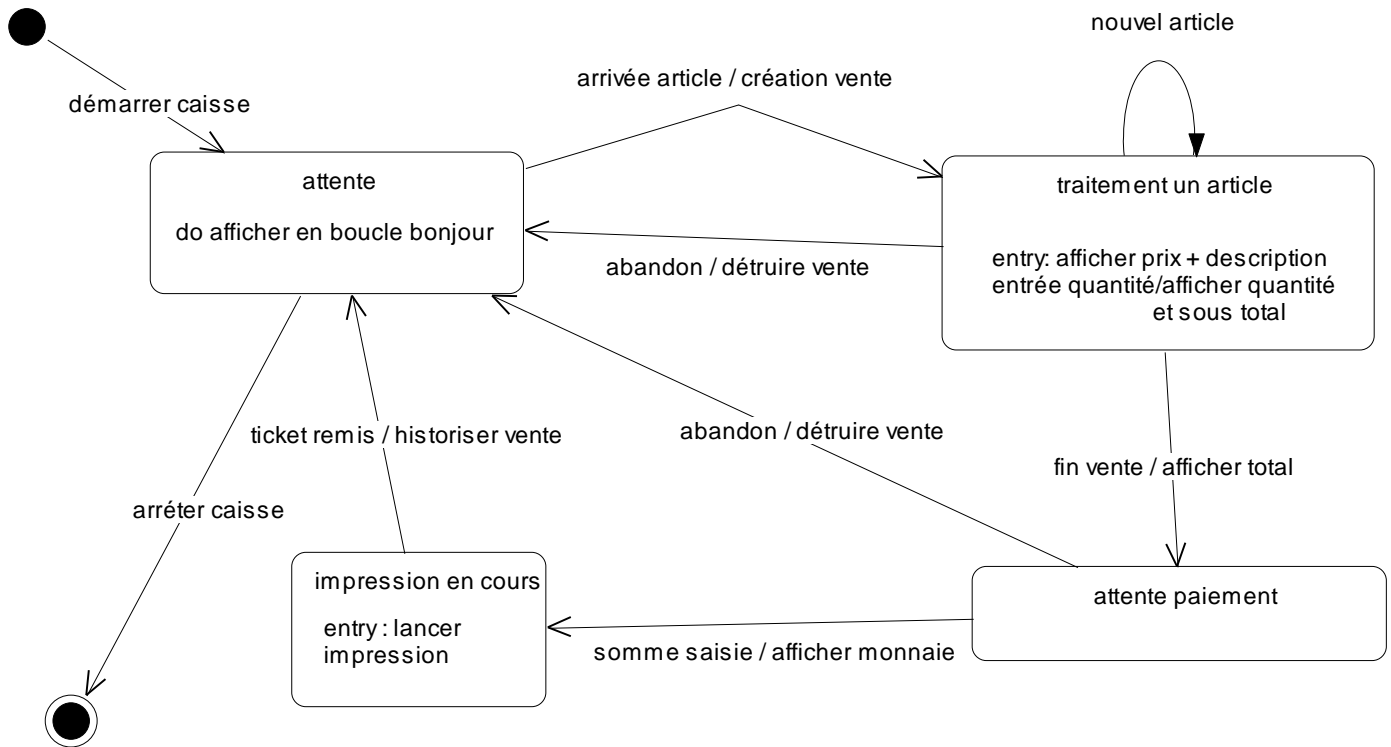
- Elle est considérée comme instantanée.
- Elle est généralement associée à une transition.

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.

II.2



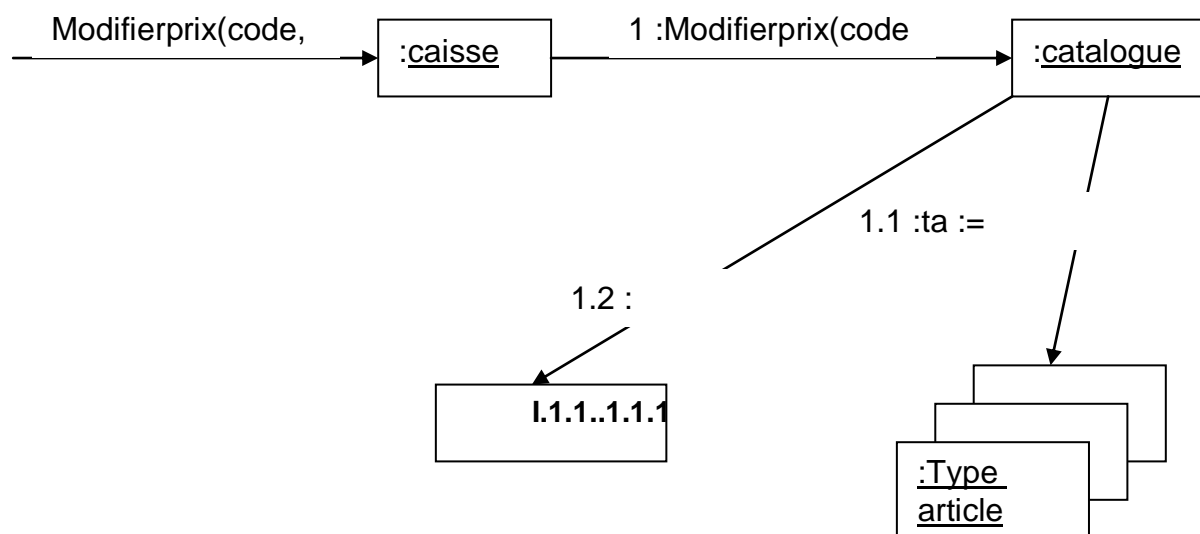
Etat de la caisse concernant le use case effectuer un achat

ONZIEME ETAPE : 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.



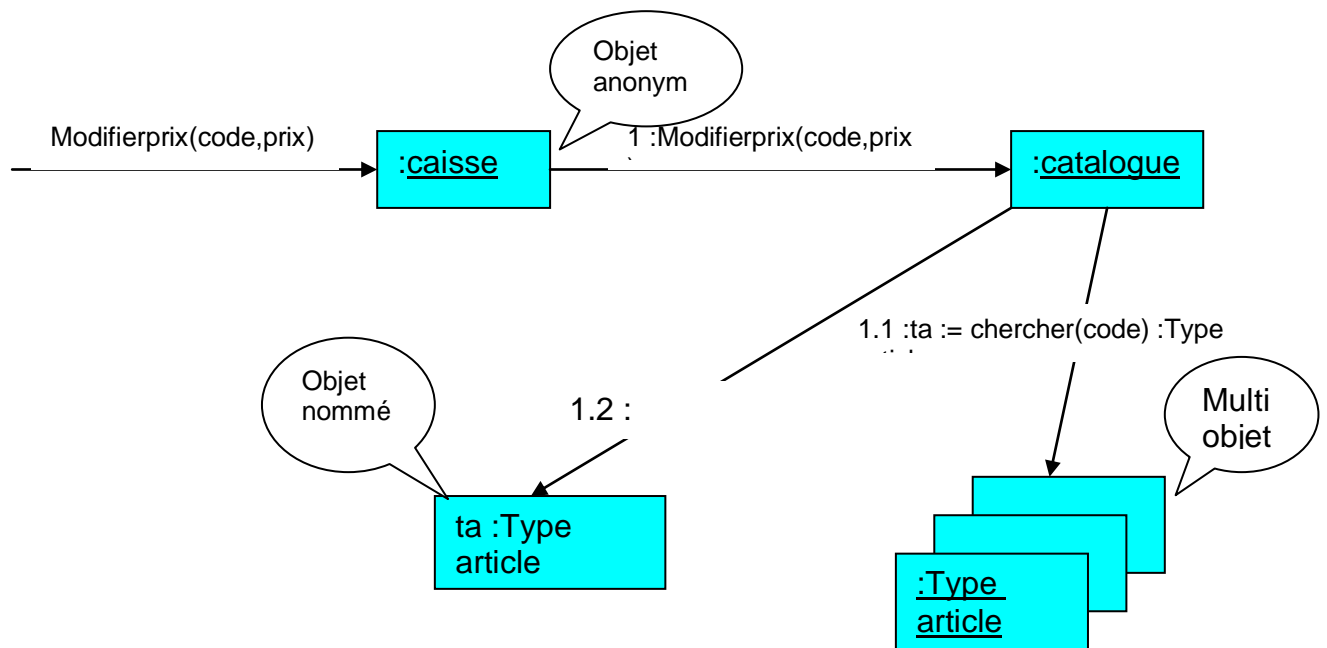
a) les objets

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 :

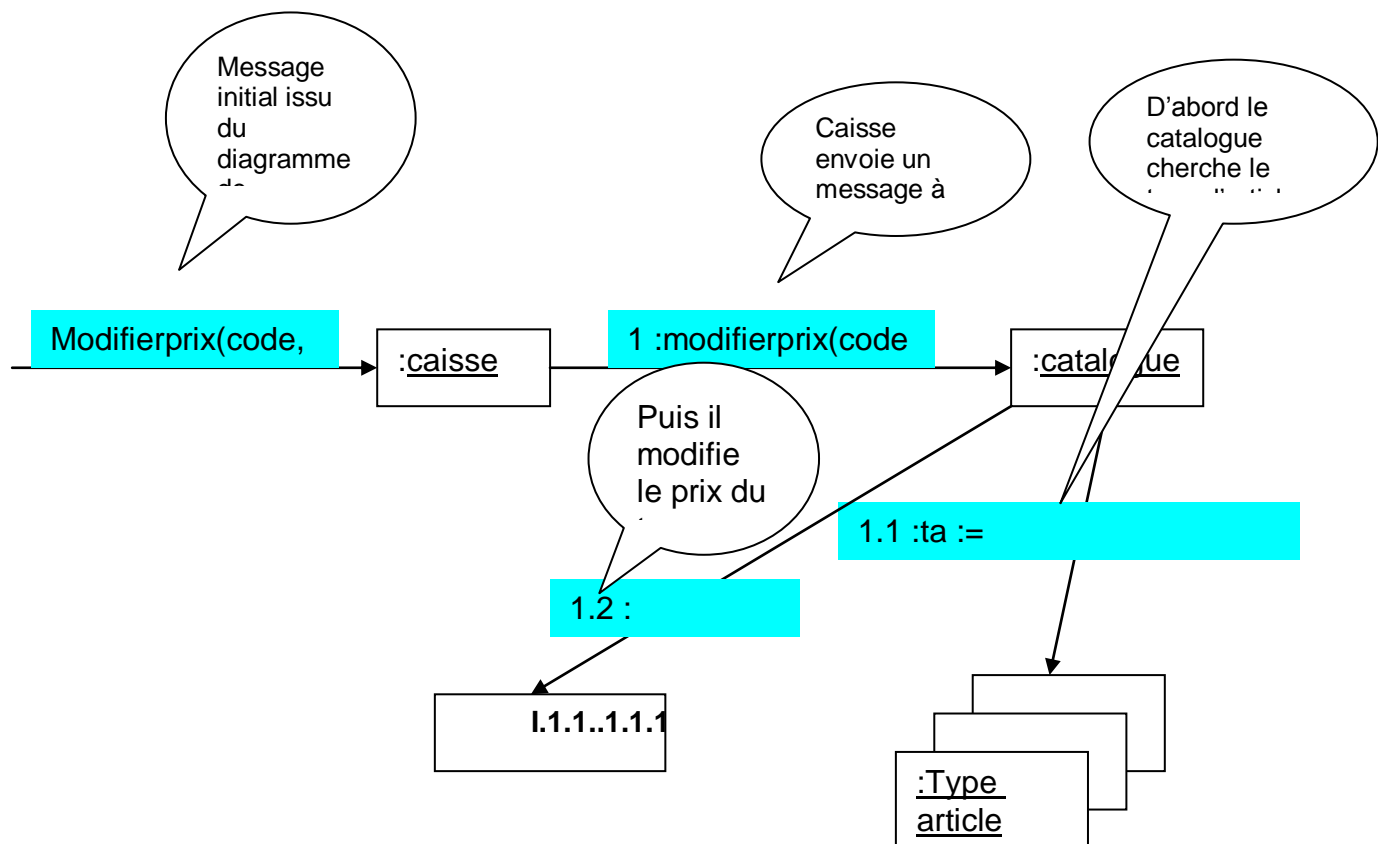
Les objets **anonymes** (:caisse, :catalogue). A comprendre la caisse courante, ou le catalogue courant...

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

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



b) la syntaxe des messages

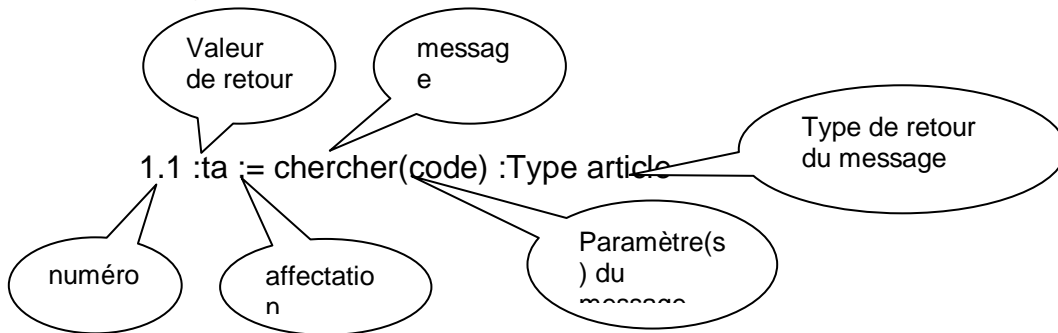


Le message initial est non numéroté par convention. Ce message est un des événements issu du diagramme de séquence boîte noire, dont nous avons fait un contrat d'opération.

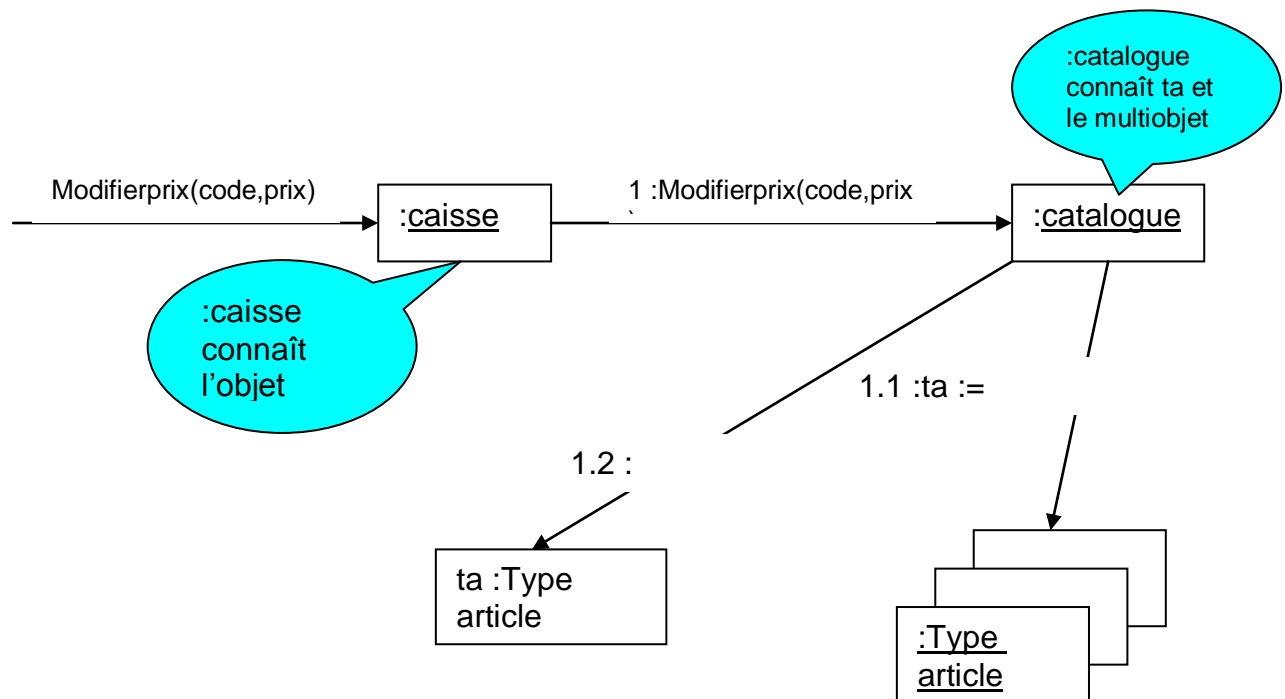
Les opérations effectuées en réponse à ce message seront numérotées de 1 à n (notons qu'en général n n'est pas très élevé, si la conception est bien faite).

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 :



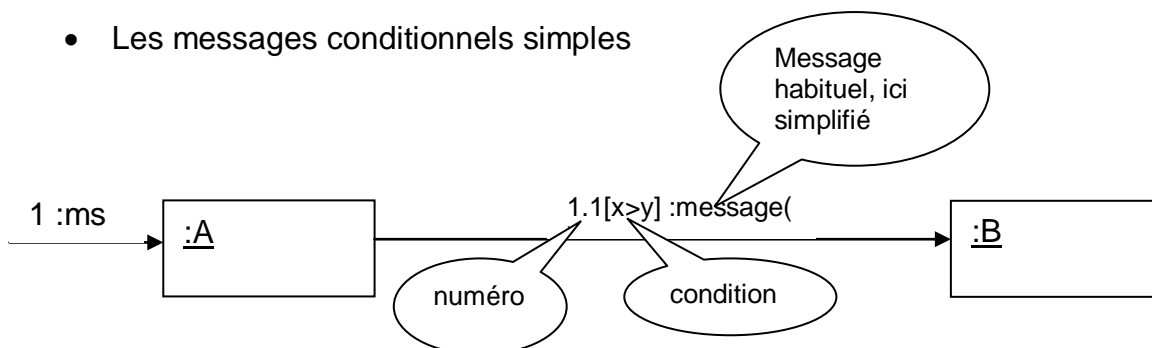
Un lien entre deux objets est directionnel. La flèche indique le receveur du message. Ce lien implique que l'objet qui envoie le message connaît celui qui le reçoit. Un objet ne peut en effet envoyer un message qu'à un objet qu'il connaît (rappelez-vous que l'on envoie un message à un objet en lui parlant : 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

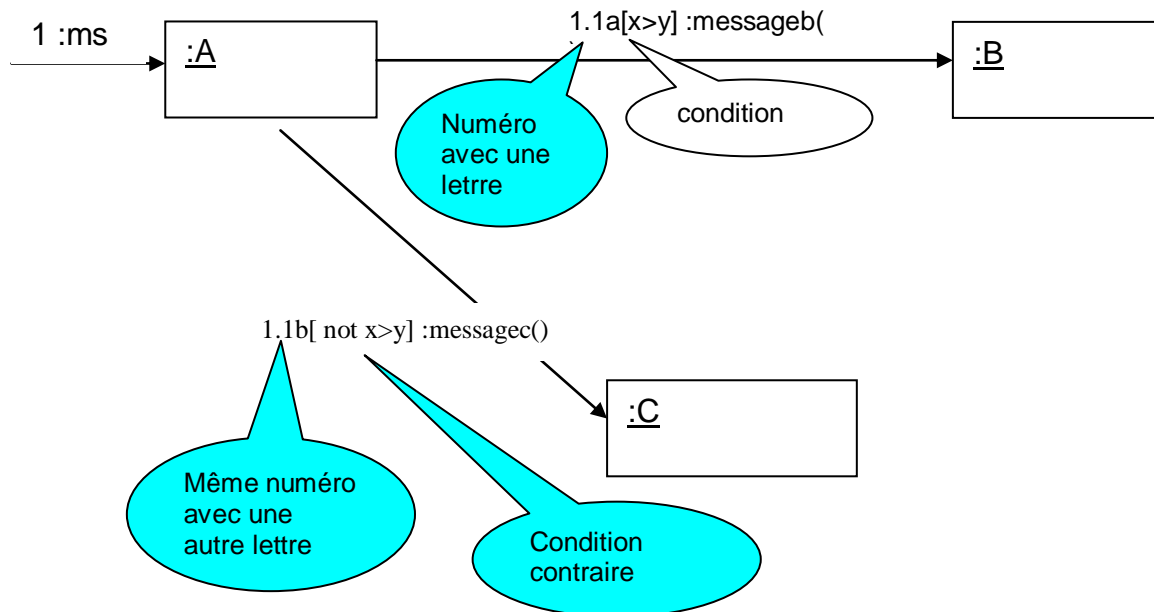
Nous avons vu la forme générale des messages. Il peut y avoir quelques formes particulières de messages, que nous allons lister maintenant.

- 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.

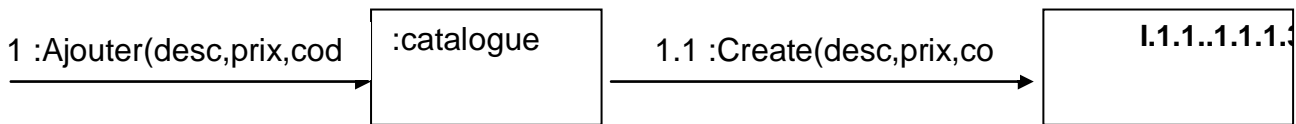
- Les messages conditionnels exclusifs (ou si sinon)



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

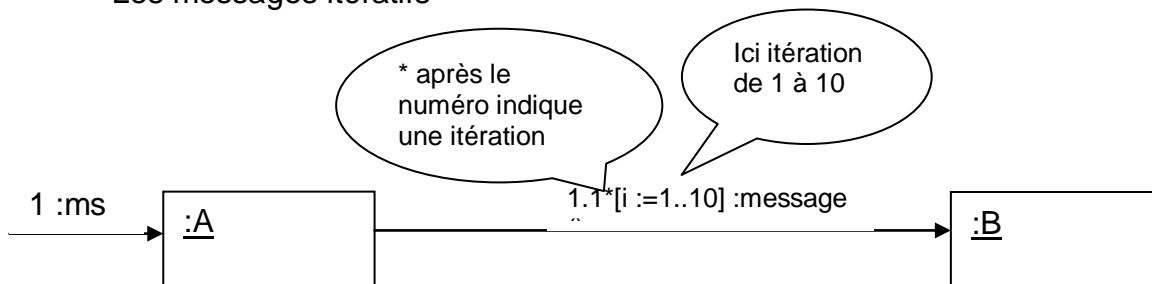
- Les messages d'instanciation

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 ?).

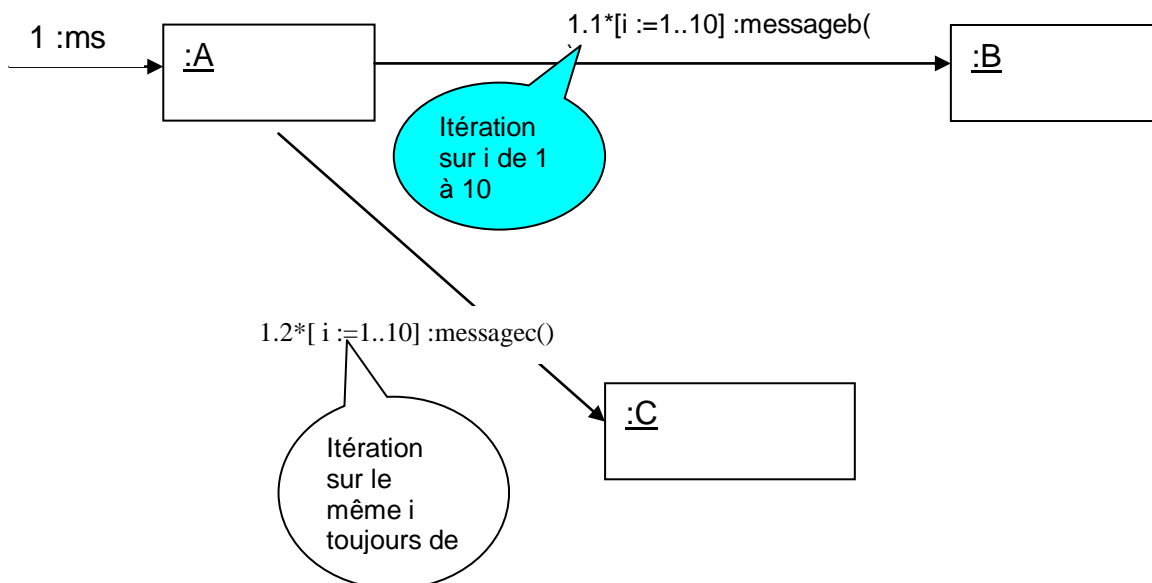
- Les messages itératifs



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

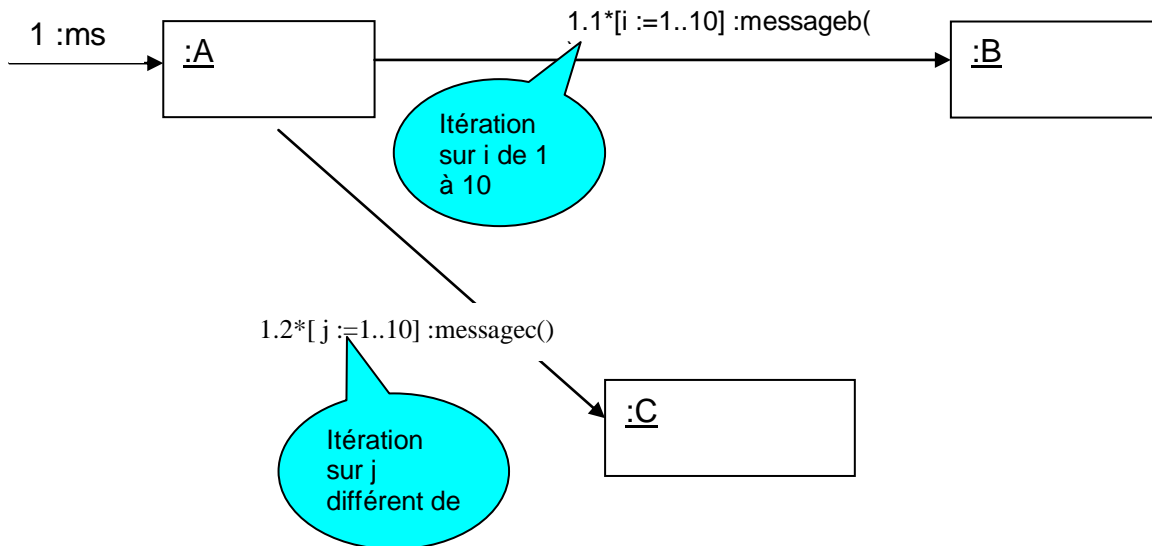
Ce sont des messages itératifs, où plusieurs messages sont envoyés dans l'itération.



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

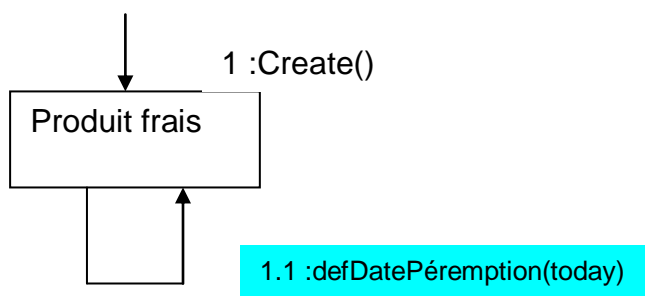
Ce sont des messages itératifs, où plusieurs itérations se suivent. Ici les boucles sont distinctes.



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

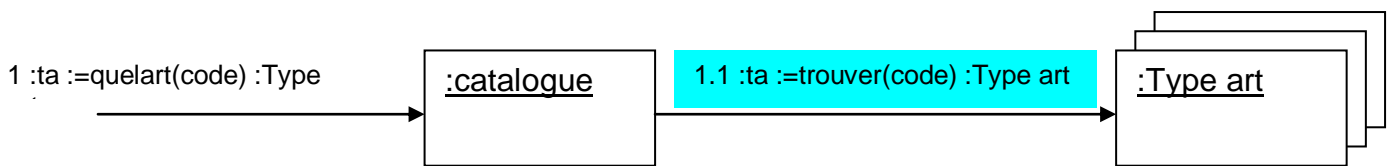
Les messages vers this

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

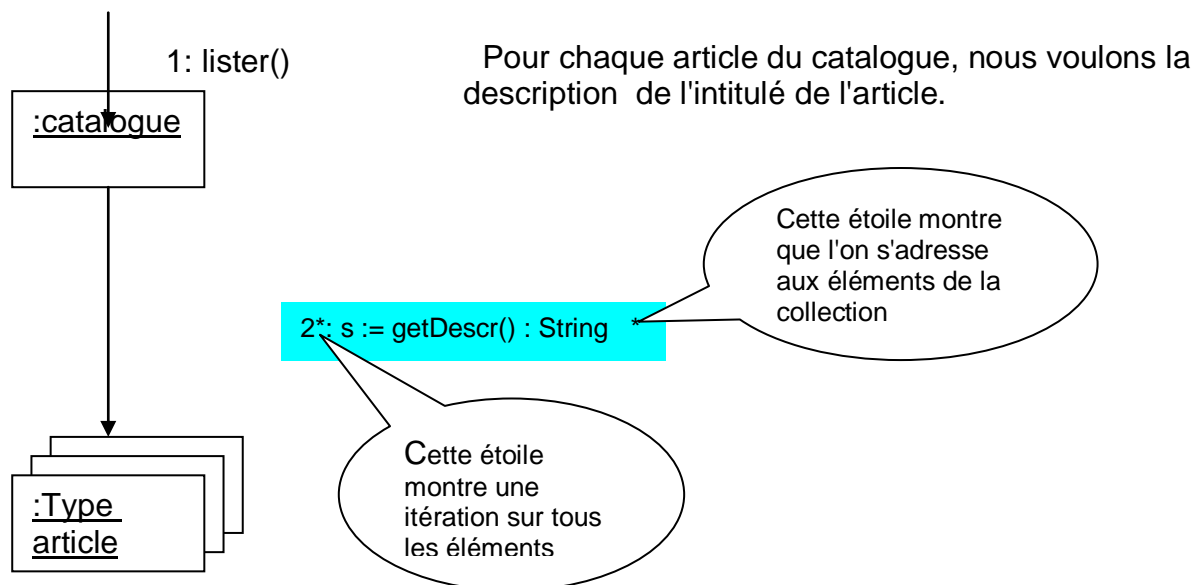


Le message trouver n'est pas envoyé à l'objet :Type art mais au multi objet, qui se traduira en programmation par une collection (tableau, vecteur, hashtable, etc).

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

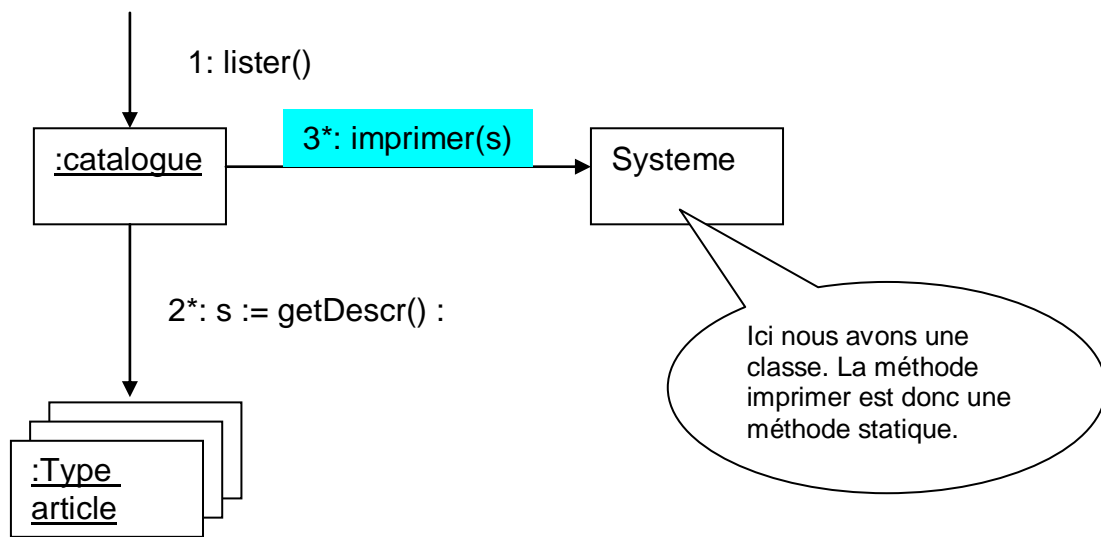
- Trouver : récupère un élément d'un multiobjet à partir d'une clé.
- Ajouter : ajoute un élément au multiobjet.
- Supprimer : supprime un élément du multiobjet.
- Taille : donne le nombre d'éléments du multiobjet.
- Suivant : permet de passer à l'élément suivant du multiobjet.
- Contient : permet de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



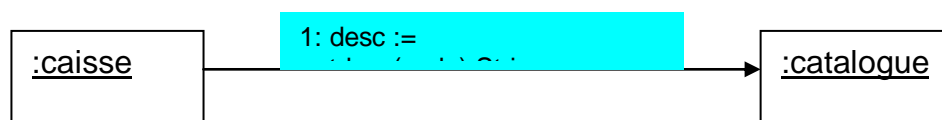
La classe caisse doit connaître la classe vente pour pouvoir lui parler: " vente , oh ma vente! Ajoute-toi un article de ce code."

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.

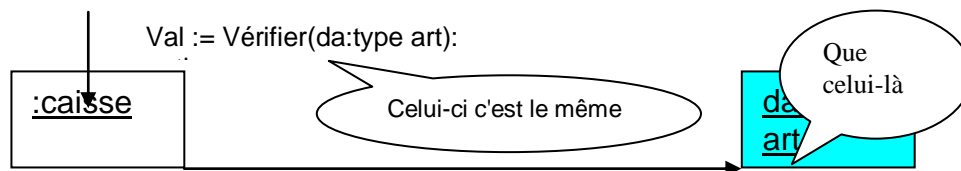
- La visibilité de type attribut.



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

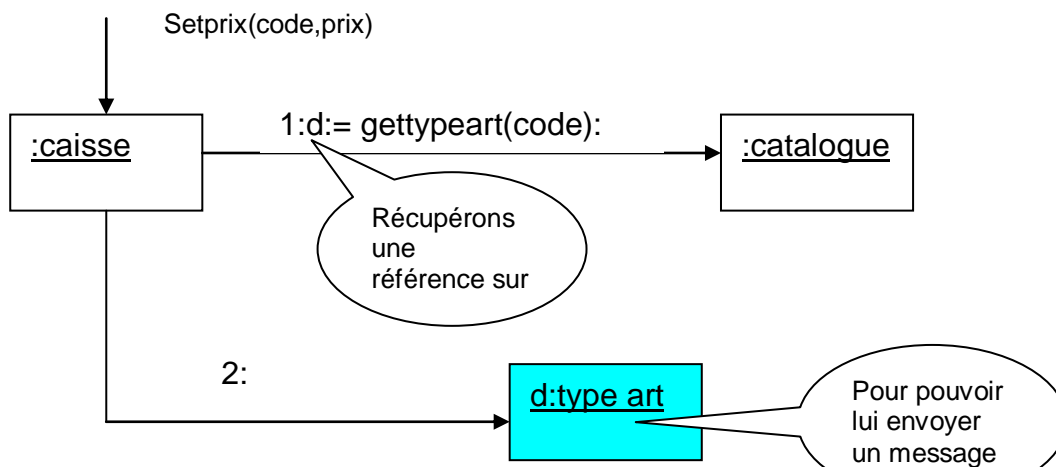
Supposons que le diagramme de séquence boîte noire mette en évidence un message de type vérifier avec en paramètre une description d'article. Supposons également que ce soit la caisse qui traite ce message. Nous obtenons le diagramme de collaboration suivant:



1: p := getPrix():

Ici la caisse ne connaît le type d'article da que temporairement. Le passage de paramètre lui fait connaître le type d'article da à qui la caisse va envoyer un message.

▪ La visibilité de type locale



Ici caisse va chercher une référence sur un type d'article, pour pouvoir envoyer un message à ce type d'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.

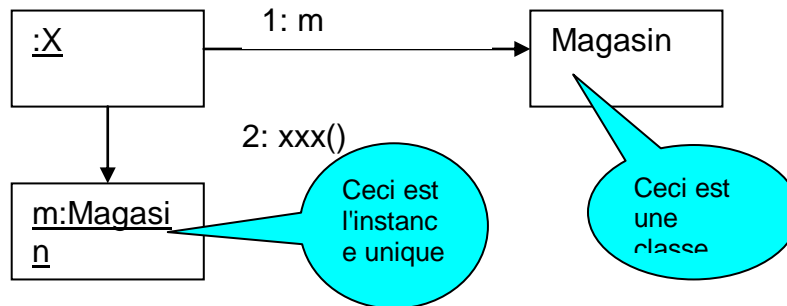
Le problème est que certaines, rares, classes ayant une instance unique doivent être connues de nombreux objets. Il est alors conseillé d'utiliser le GOF Pattern "Singleton".

Supposons qu'une classe nécessite d'en connaître une autre ayant une instance unique (le magasin dans notre exemple), 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

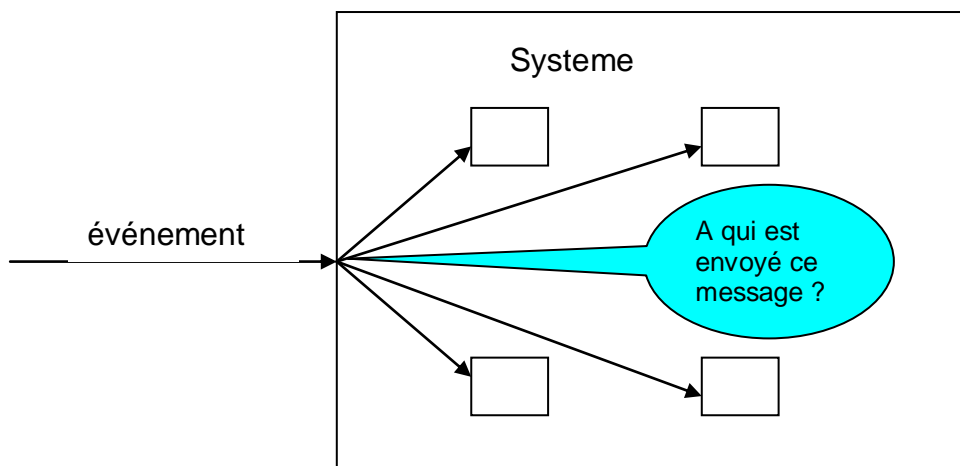
magasin existant (car la méthode étant statique, elle est envoyée à la classe elle même). Ainsi l'autre classe pourra envoyer sa requête à l'objet magasin.

Voici un diagramme de collaboration qui illustre cet exemple.



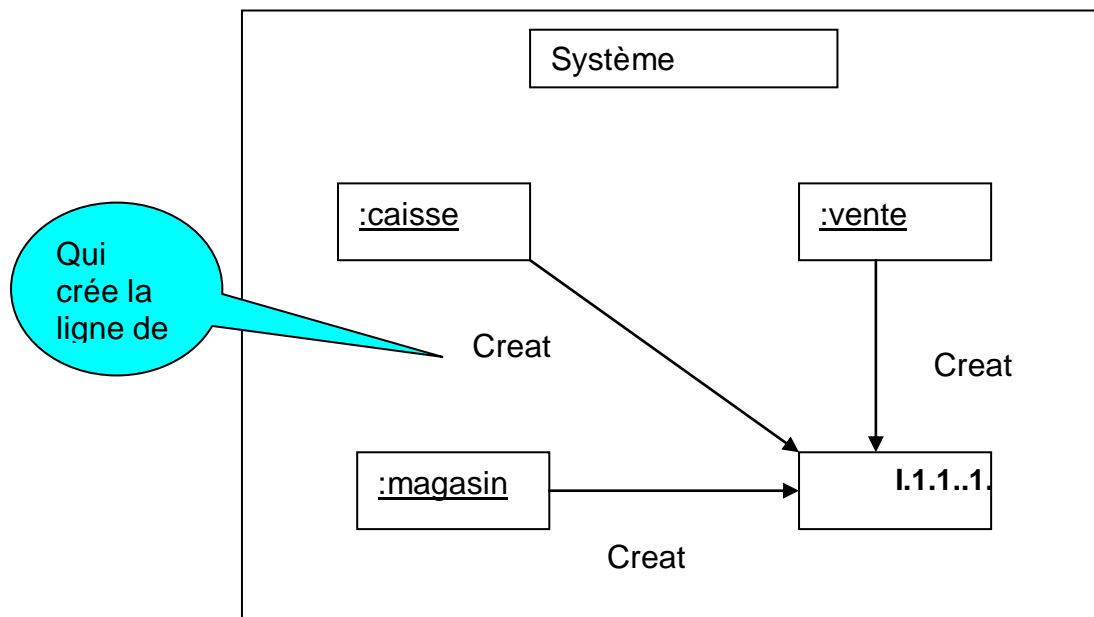
3) GRASP patterns

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).

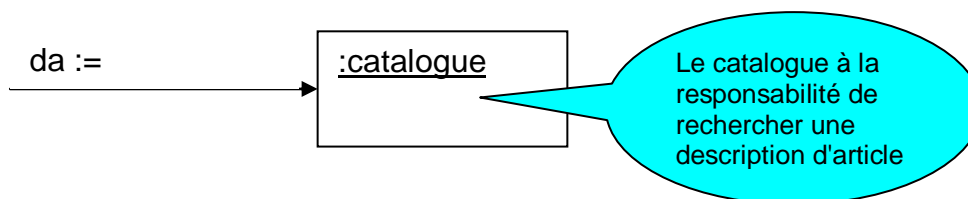
To grasp (en anglais) veut dire: saisir, comprendre, intégrer. Il est fondamental d'intégrer ces modèles avant de faire de la conception objet, pour obtenir des diagrammes de classe de conception, et des diagrammes de collaboration de qualité.

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.

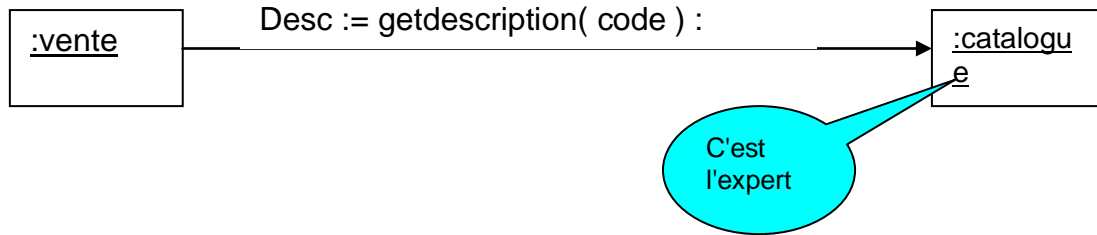
Ces classes à faible cohésion sont difficiles à comprendre, à réutiliser, à maintenir. Elles sont également fragiles car soumises aux moindres variations, elles sont donc instables.

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:

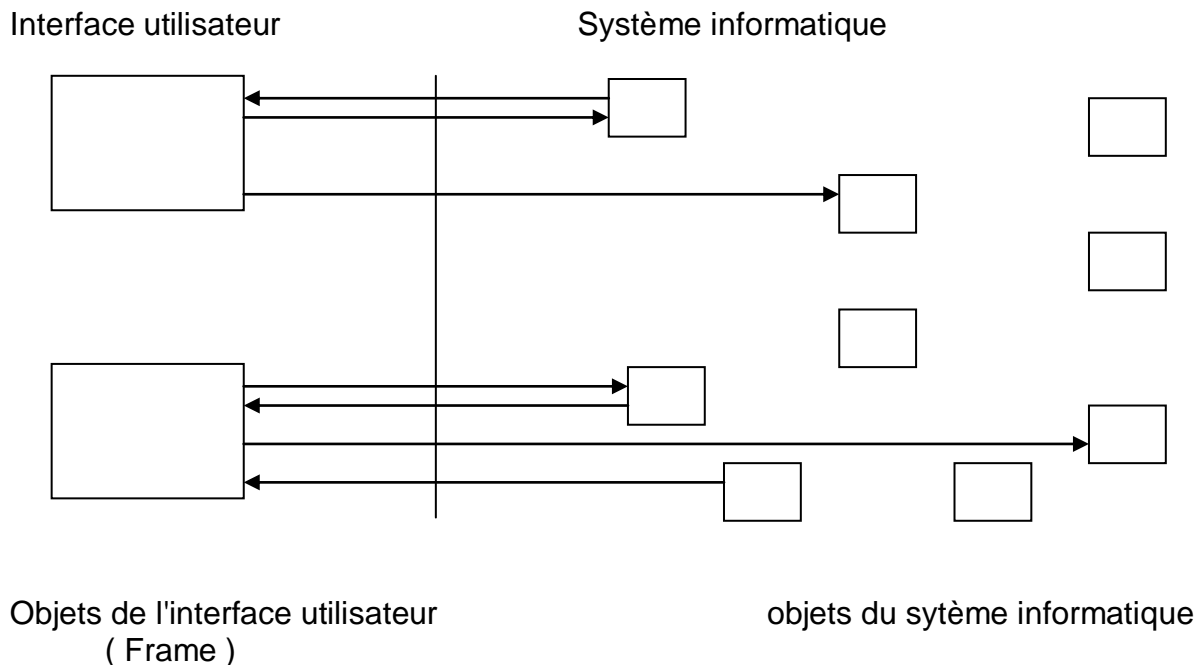
- A contient B.
- A est un agrégat de B.
- A enregistre B.
- A utilise souvent B.

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.

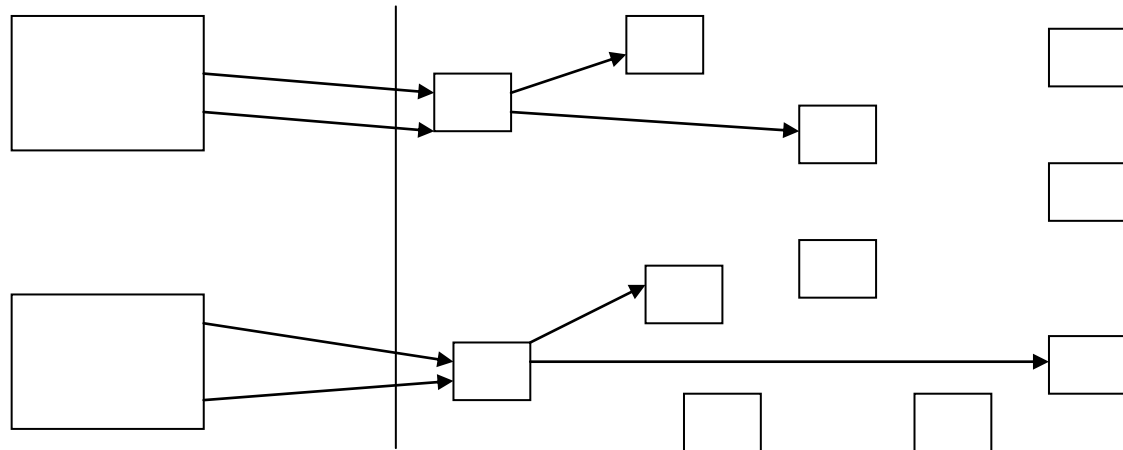
Il peut y avoir quatre sortes de contrôleurs:

- 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.

Interface utilisateur

Système informatique



Objets de l'interface utilisateur
(Frame)

objets du système informatique

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

L'événement `entrerunarticle` 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?

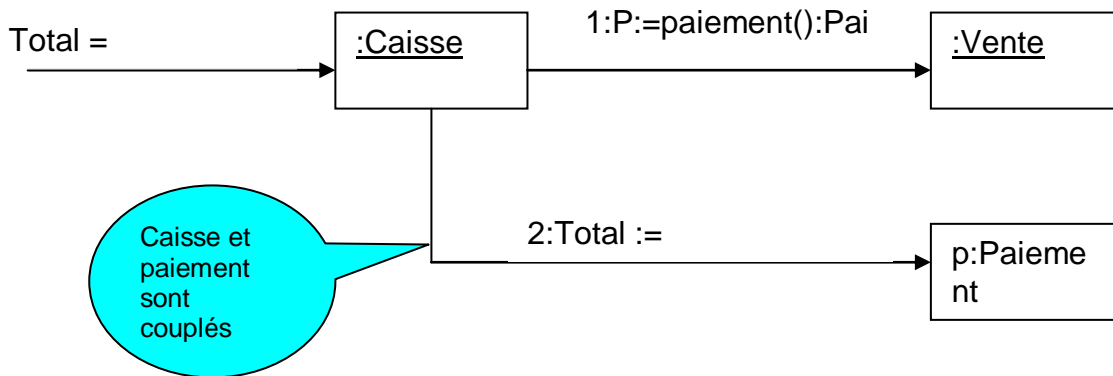
- Lui-même.
- Un objet paramètre de la méthode appelée.
- Un objet attribut de l'objet lui-même.
- Un objet élément d'une collection attribut de l'objet lui-même.
- Un objet créé par la méthode.

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

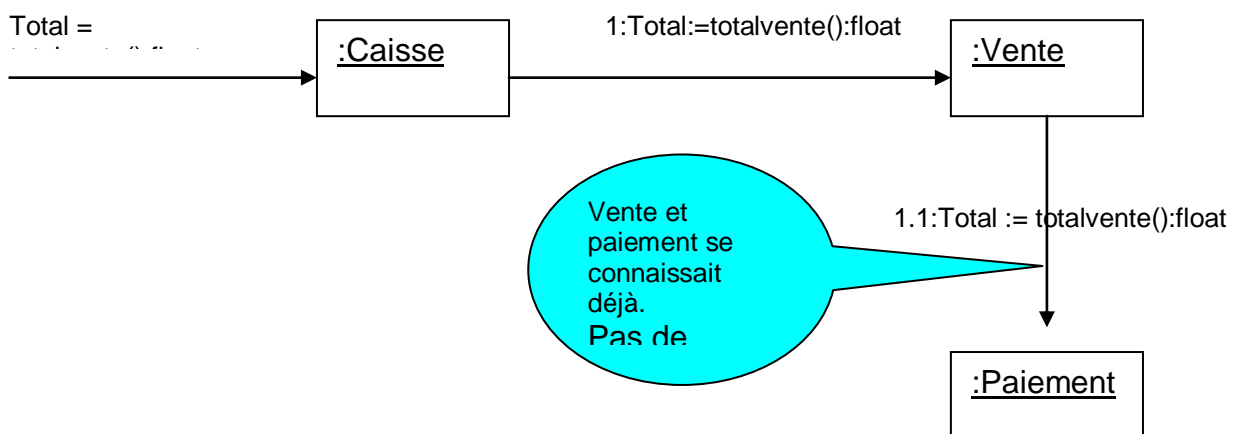
Prenons un exemple:

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 changent rien pour la caisse.

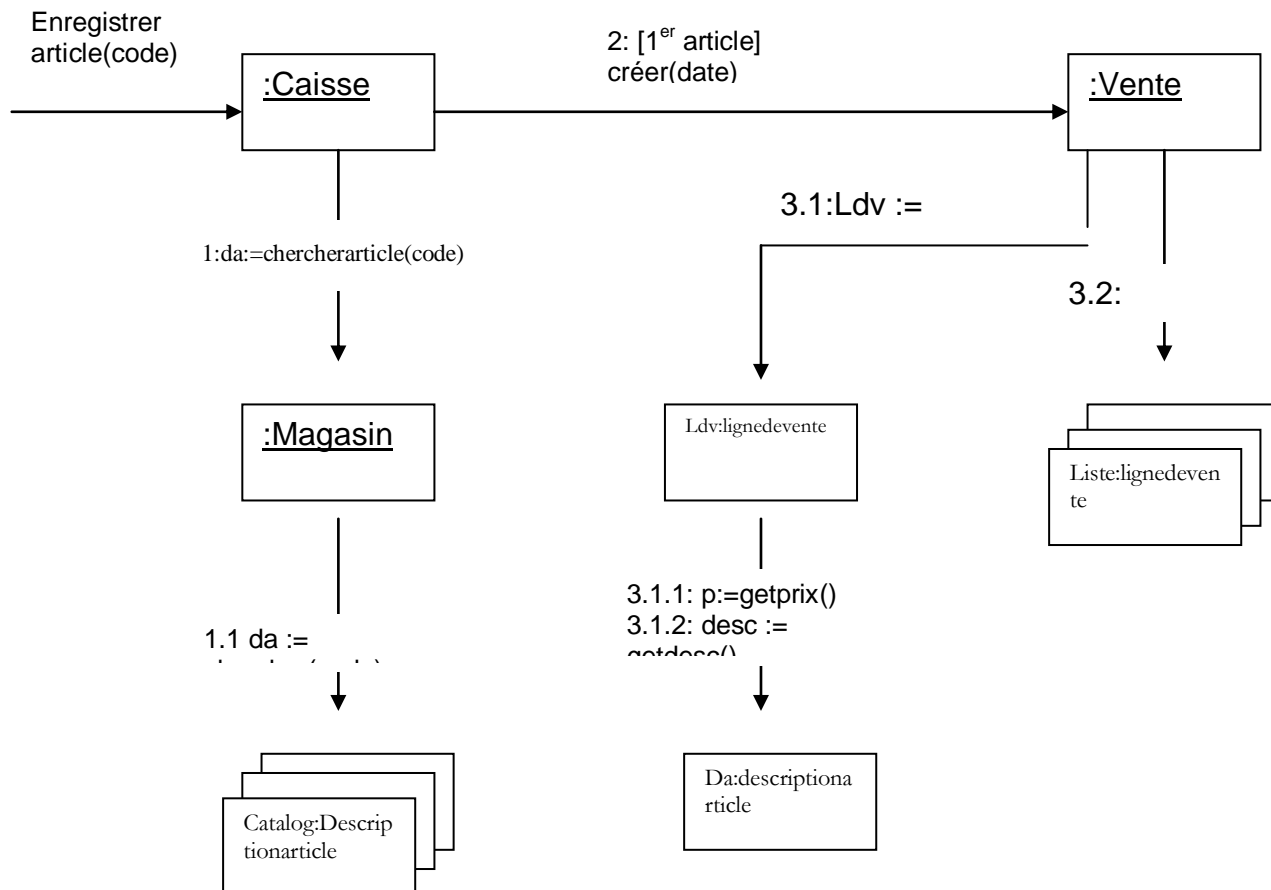
4) Diagramme de collaboration du magasin

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.

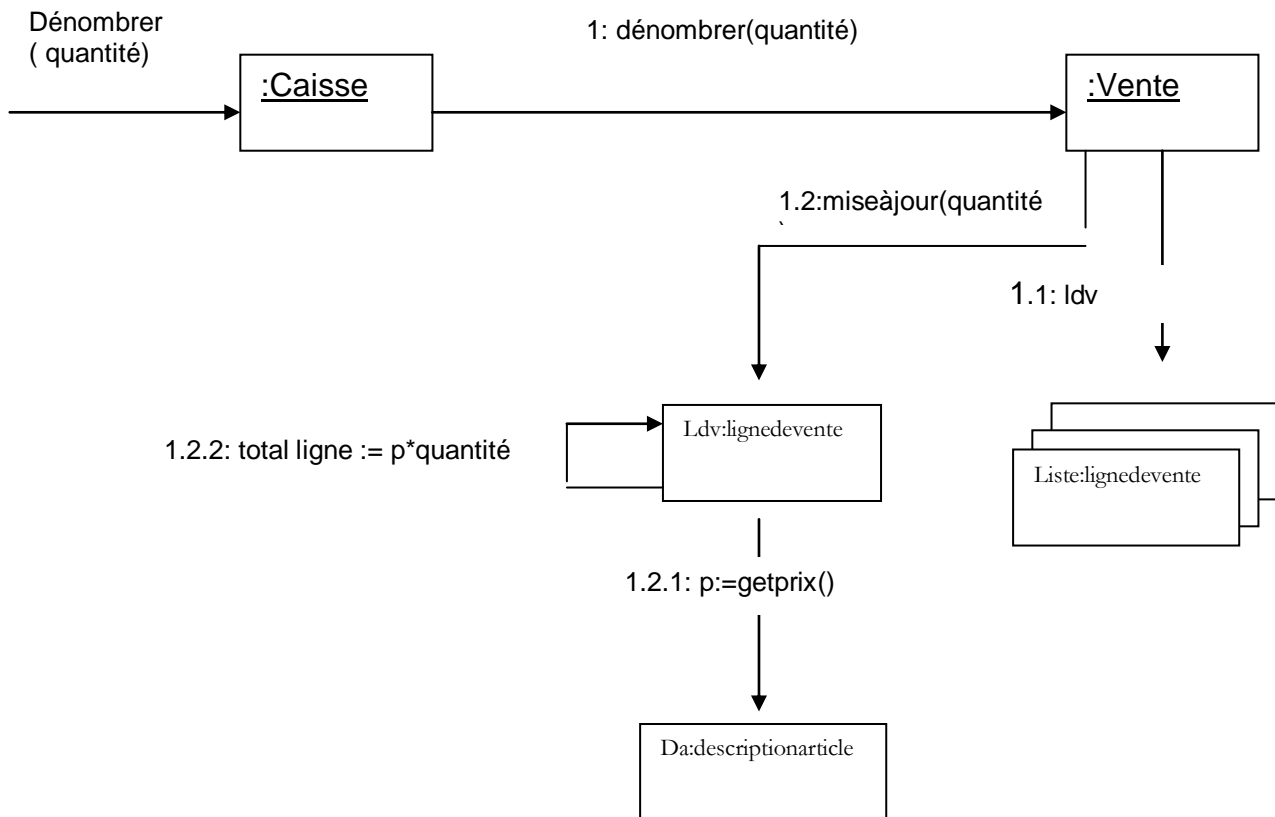
Nous allons faire autant de diagrammes de collaboration que nous avons fait de contrats d'opérations. Pour chaque contrat d'opération nous allons prendre l'événement du système comme message d'attaque de notre diagramme de collaboration, puis nous allons créer les interactions entre les objets qui, à 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 construise au hasard.

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

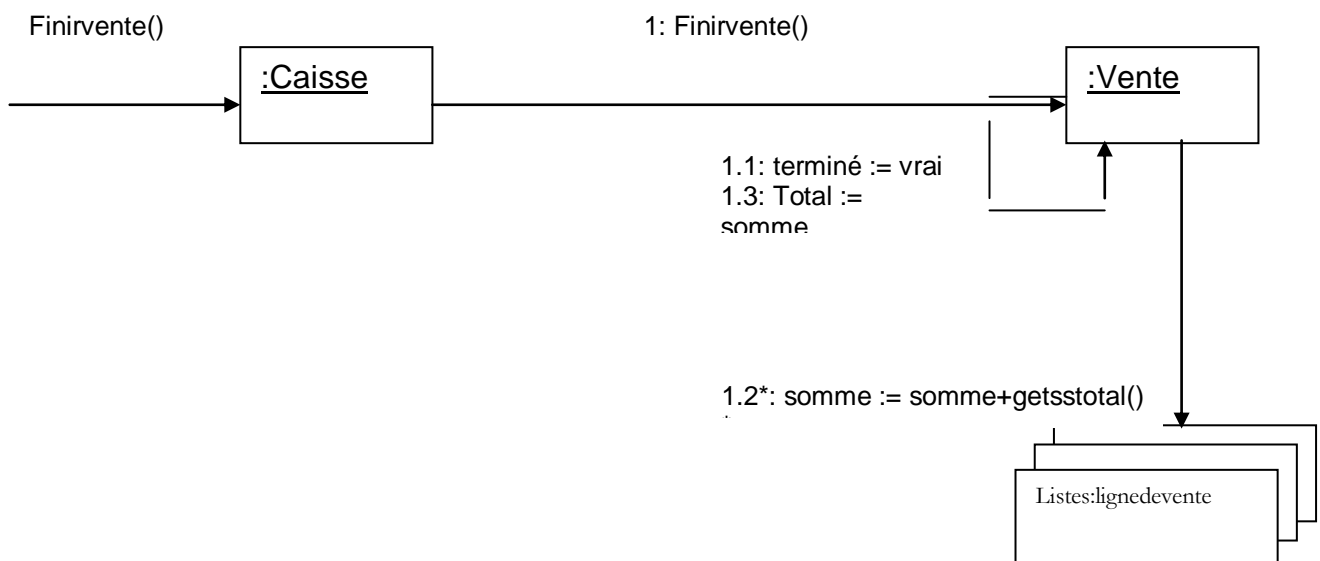
4.1) Diagramme de collaboration de Enregistrer un article



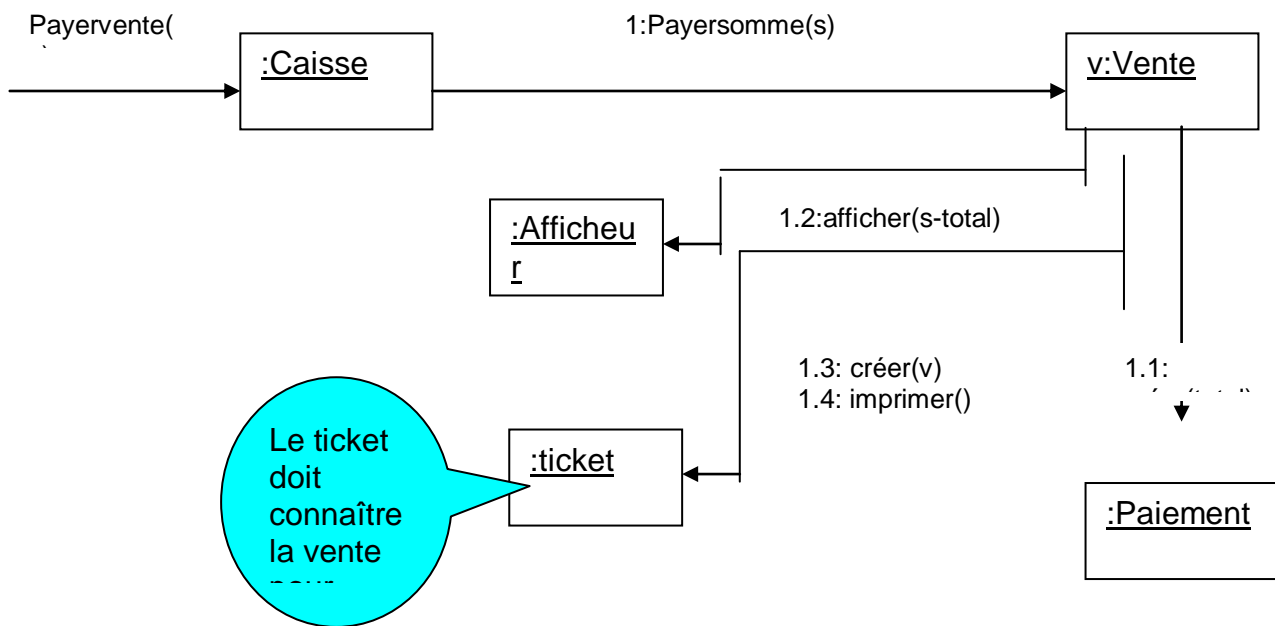
4.2) Diagramme de collaboration de dénombrer les articles identiques



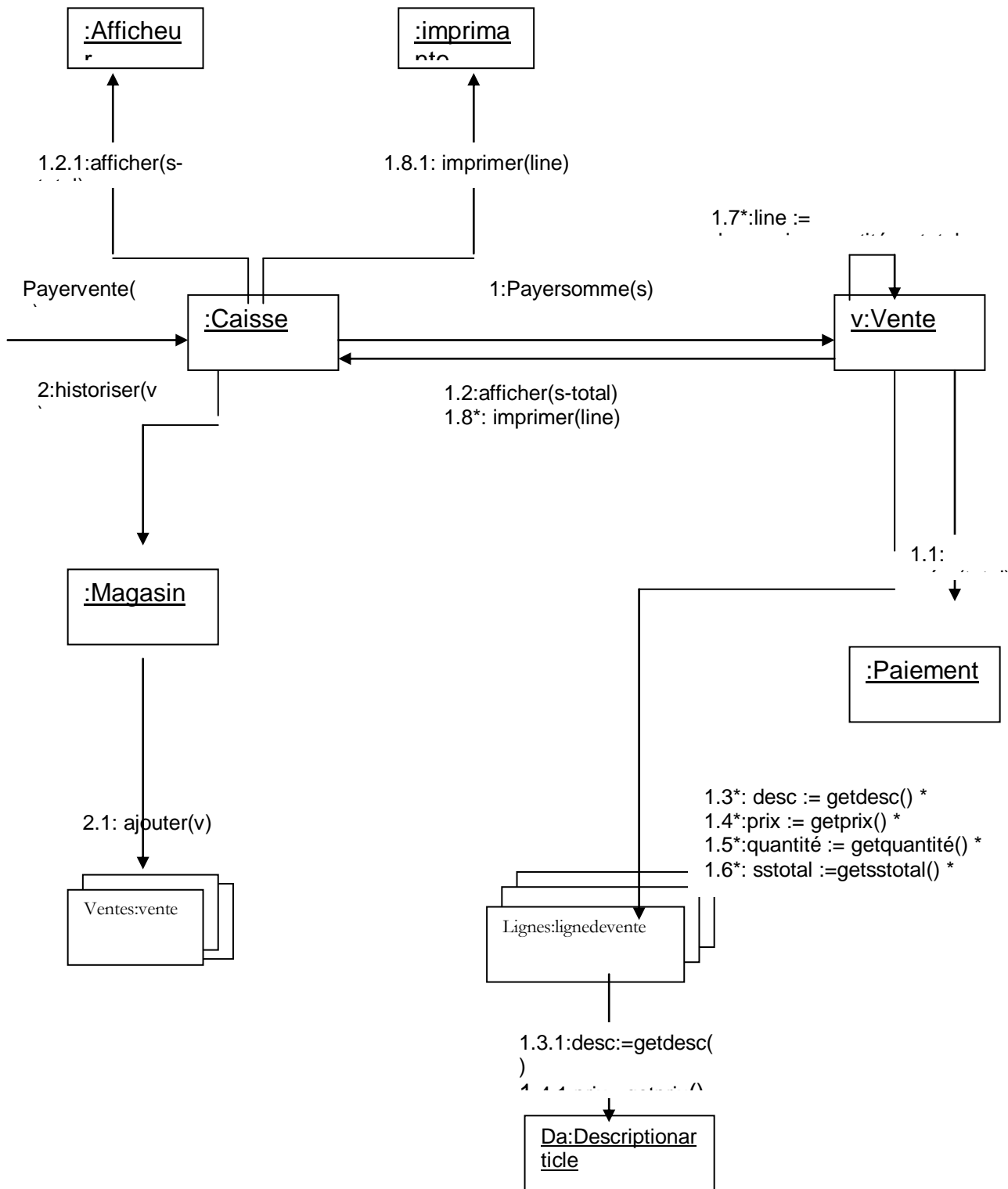
4.3) Diagramme de collaboration de Finir la vente



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.



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.

II.3 DOUZIEME ETAPE : 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

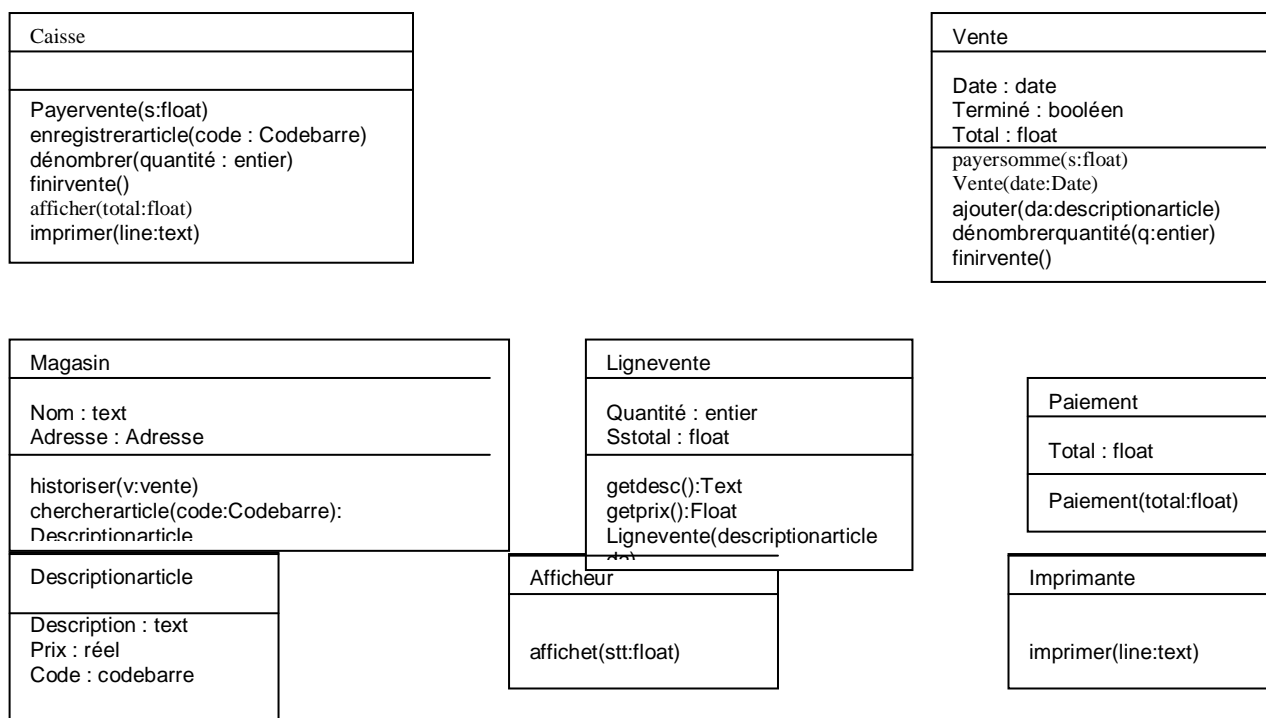
Nous allons référencer toutes les classes rencontrées dans les diagrammes de collaboration. Nous allons les dessiner dans un diagramme de classes. Nous allons y ajouter les attributs venant du diagramme de classe d'analyse, et les méthodes venant du diagramme de collaboration.

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.

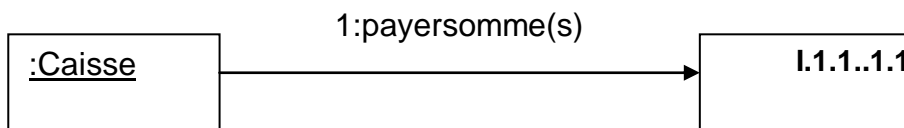


2) Deuxième étape

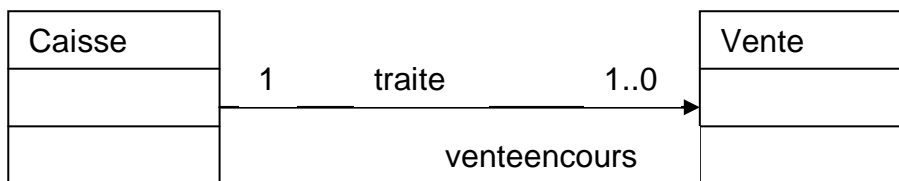
Il faut maintenant ajouter les associations nécessaires pour montrer la visibilité par attribut des objets. Cette visibilité par attribut va permettre de construire les attributs qui référencent les objets que l'on doit connaître. Cette association porte une flèche de navigation qui nous permet de connaître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

Prenons trois exemples :

1) issu du diagramme de collaboration de payervente:

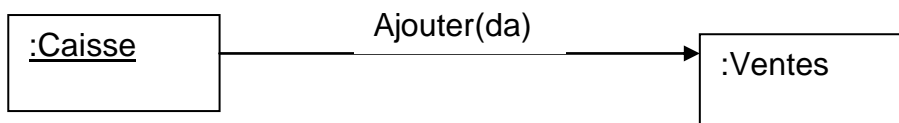


Ici la caisse doit connaître en permanence la vente en cours: nous aurons une association de type attribut.

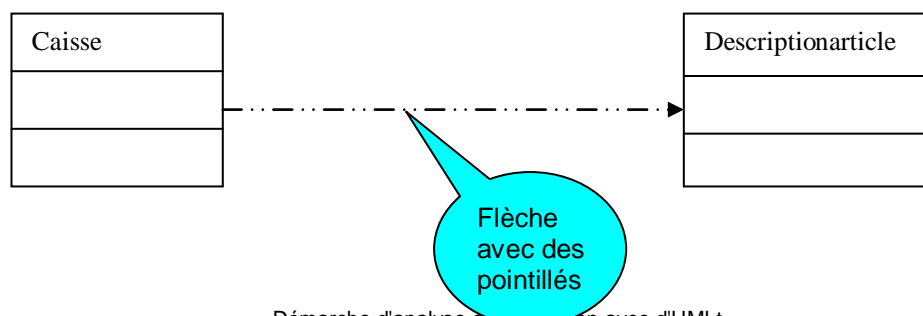


Cela indique que la classe Caisse a un attribut qui référence l'objet vente en cours. Le nom de rôle venteencours donnera, pour une génération automatique de code, le nom de l'attribut qui référence la vente en cours.

2) issu du diagramme de collaboration de enregistrerarticle :

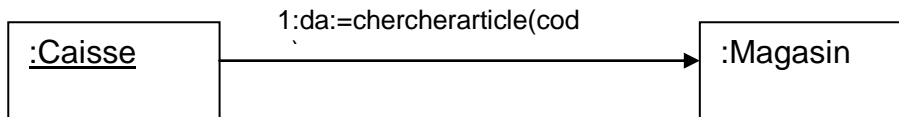


Ici la caisse a une visibilité de type variable locale sur la descriptionarticle (cela serait le même traitement si il avait une visibilité de type paramètre comme pour la Vente). Nous ajouterons des dépendances de type relation entre les deux classes.



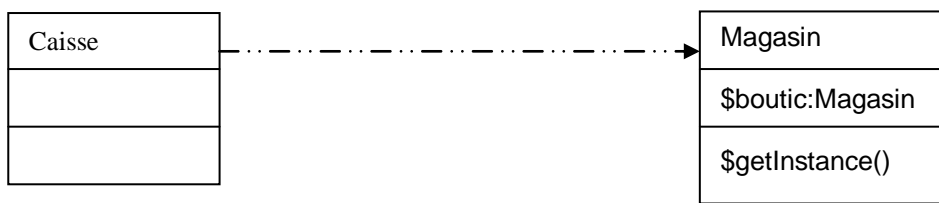
Cette association montre que le code des méthodes de Caisse doit manipuler la classe Descriptionarticle. Il sera donc nécessaire, à la génération de code, d'inclure un accès à la classe Descriptionarticle dans la classe Caisse (import java, ou include c++).

3) issu du diagramme de collaboration de payervente :



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;
}
  
```

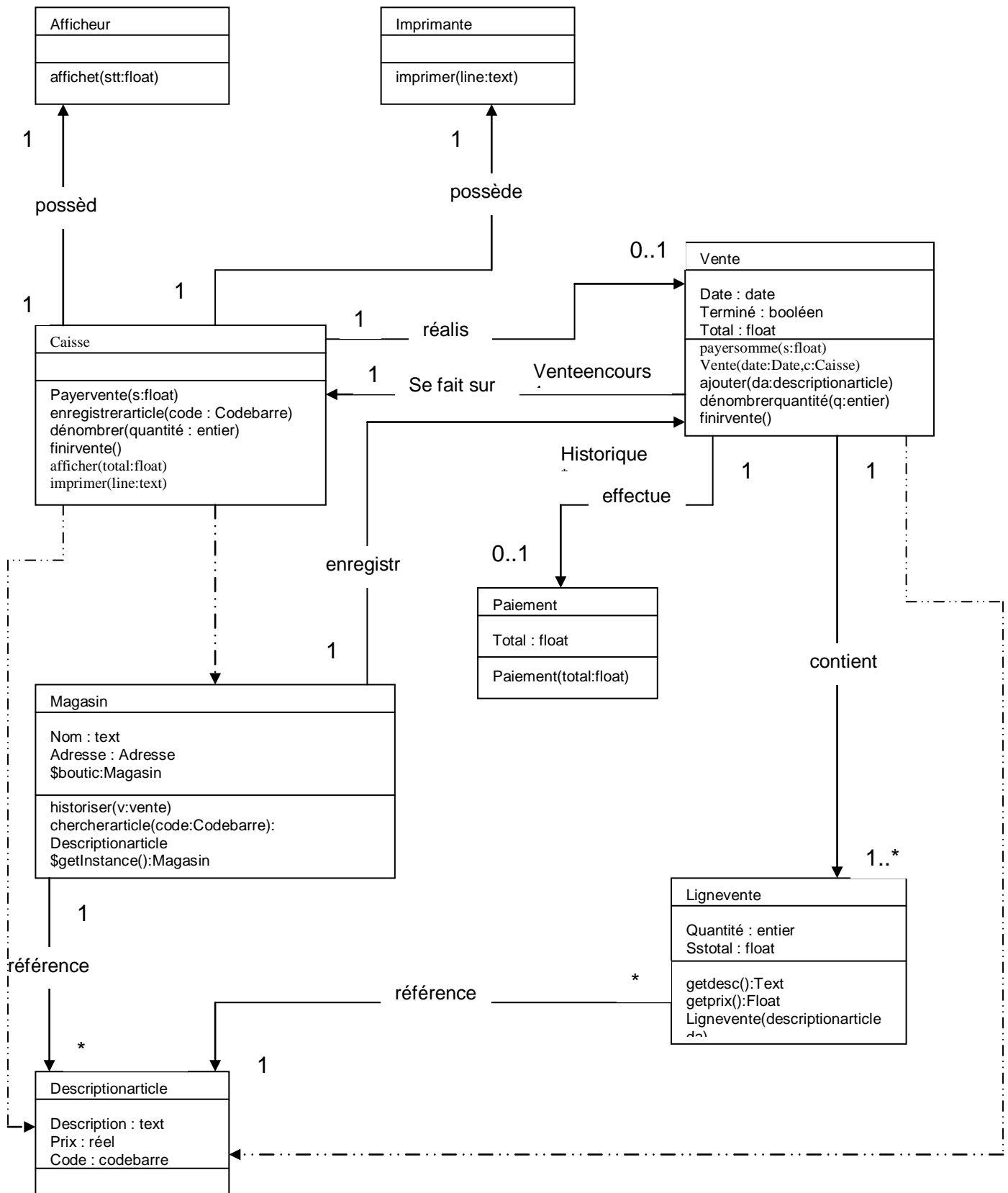
Ainsi nous aurons une instance unique du Magasin. Cette instance peut être appelée partout en utilisant le formalisme suivant:

(exemple en java)


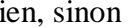
```

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 de descriptionarticle (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)

III 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.

III.1 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.

III.2 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.

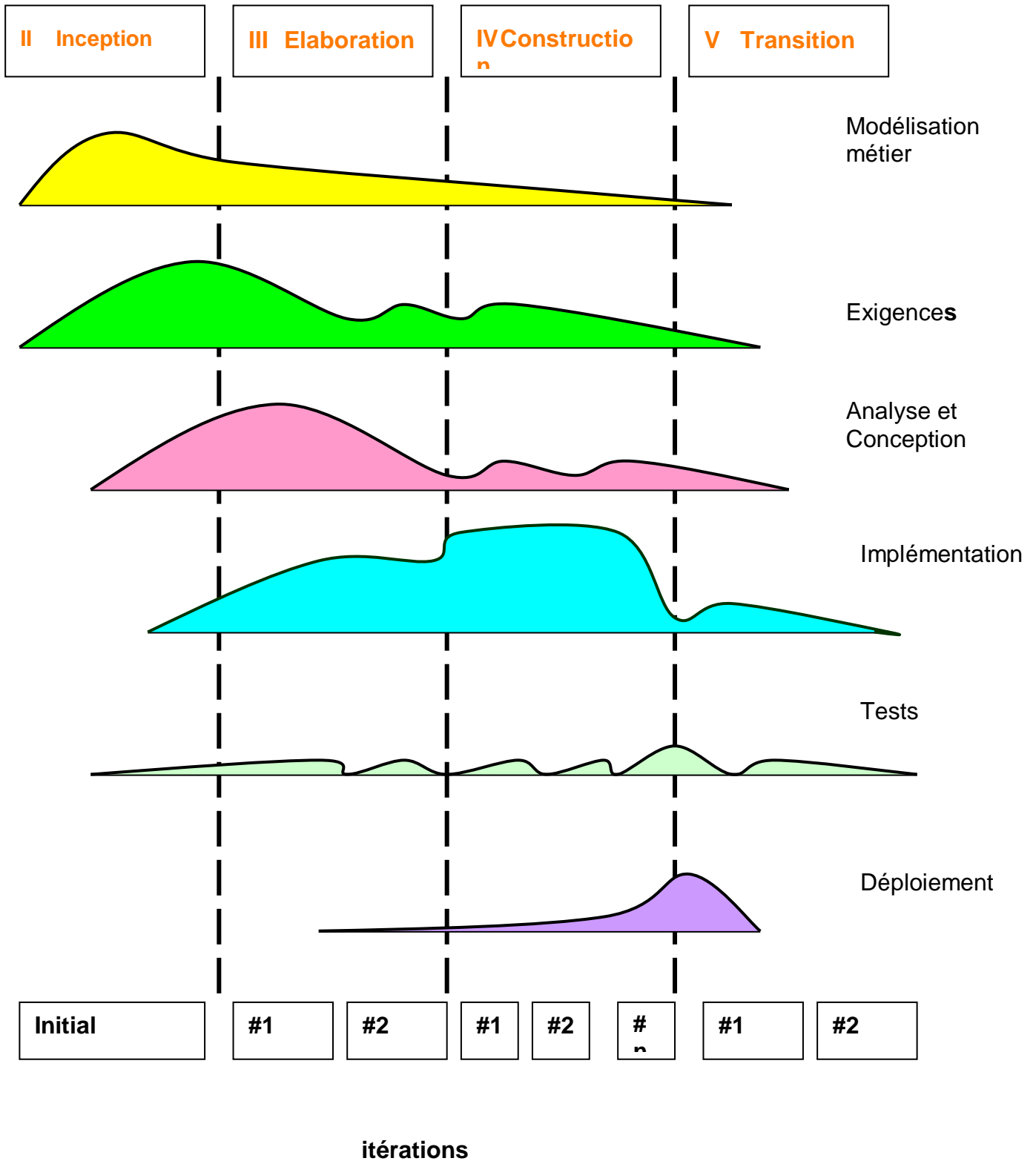
- Effort : 10% du temps total.
- Objectif :
 - faire les bêta tests.
 - convertir les données antérieures.
 - former les utilisateurs.
 - finalisation des manuels utilisateurs.
 - corriger les derniers bugs.
- Documents produits :
 - Diagrammes de composants.
 - Diagramme de déploiement.

III.3 NOTION D'ITERATION

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



IV 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.

V 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.

Etablissement référent
DI NEUILLY

Equipe de conception
Jean-Christophe CORRE

Remerciements :

Reproduction interdite

Article L 122-4 du code de la propriété intellectuelle.
« toute représentation ou reproduction intégrale ou partielle faite sans le
consentement de l'auteur ou de ses ayants droits ou ayants cause est
illicite. Il en est de même pour la traduction, l'adaptation ou la reproduction
par un art ou un procédé quelconques. »

Date de mise à jour jj/mm/aa
afpa © Date de dépôt légal mois année



**afpa / Direction de l'Ingénierie 13 place du Générale de Gaulle / 93108 Montreuil
Cedex
association nationale pour la formation professionnelle des
adultes
Ministère des Affaires sociales du Travail et de la
Solidarité**