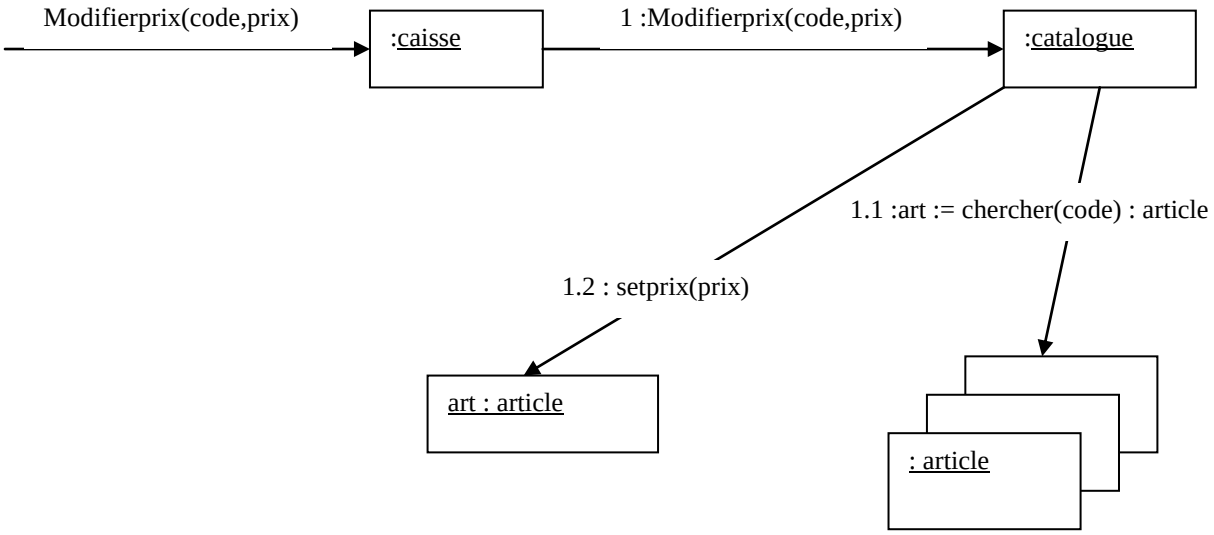
onzième étape : le diagramme de collaboration

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 permettrons 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.

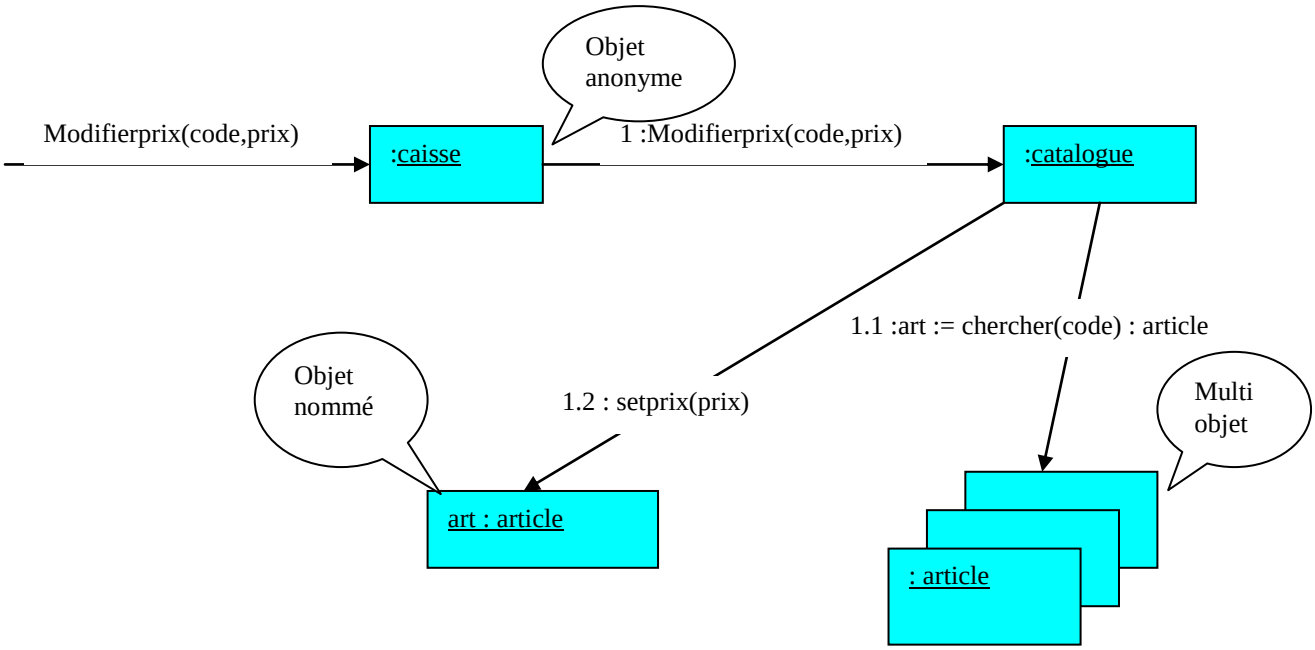


a) les objets

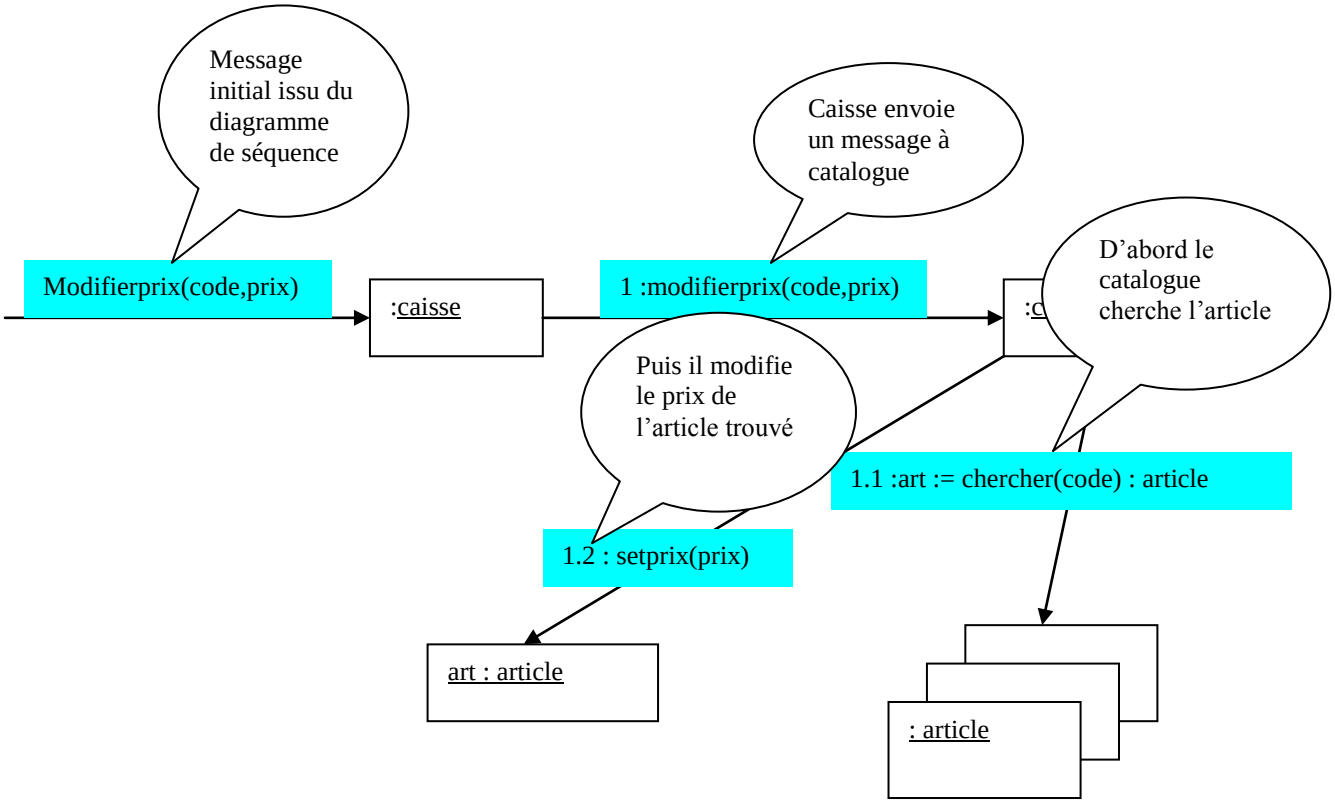
Ici nous représentons la coopération des objets pour rendre le service demandé. Il s'agit donc bien d'objets. Ces objets apparaissent sous trois formes :

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

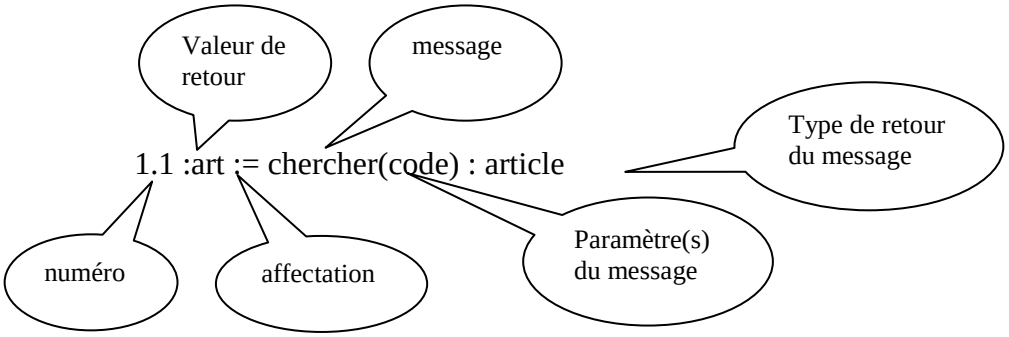


Le message initial est non numéroté par convention. Ce message est un des événements issu du diagramme de séquence boîte noire, dont nous avons fait un contrat d'opération.

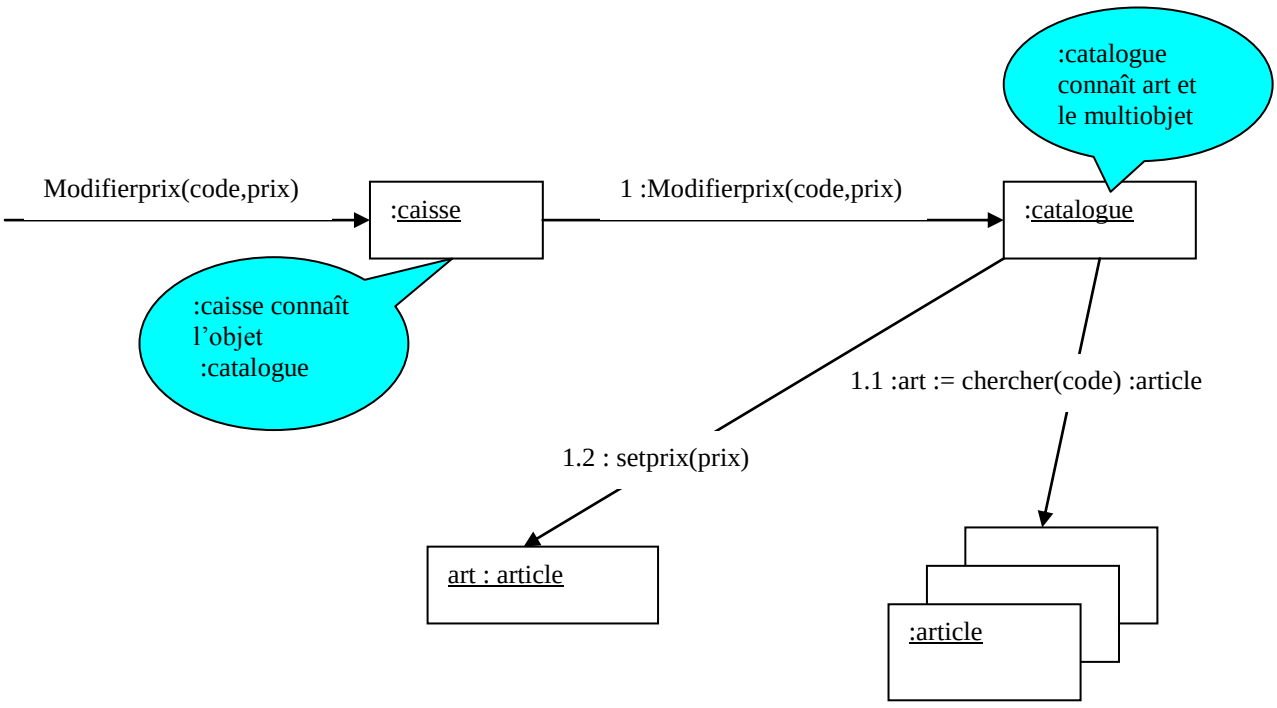
Les opérations effectuées en réponse à ce message seront numérotées de 1 à n (notons qu'en général n n'est pas très élevé, si la conception est bien faite).

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 :



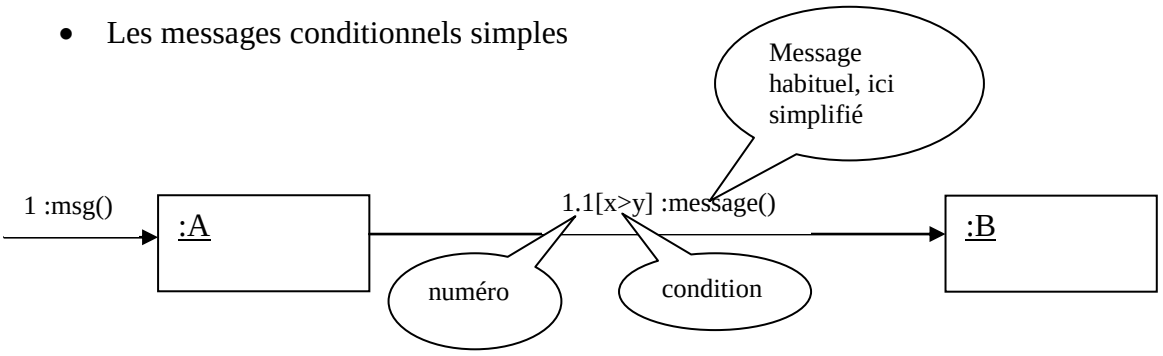
Un lien entre deux objets est directionnel. La flèche indique le receveur du message. Ce lien implique que l'objet qui envoie le message connaisse celui qui le reçoit. Un objet ne peut en effet envoyer un message qu'à un objet qu'il connaît (rappelez-vous que l'on envoie un message à un objet en lui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

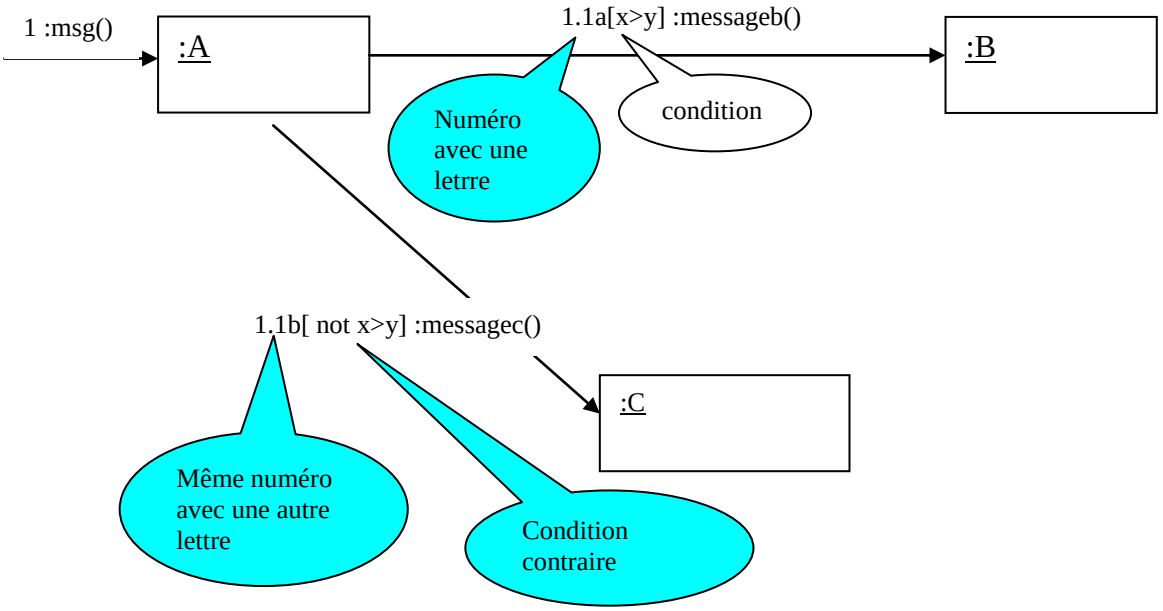
Nous avons vu la forme générale des messages. Il peut y avoir quelques formes particulières de messages, que nous allons lister maintenant.

- 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.

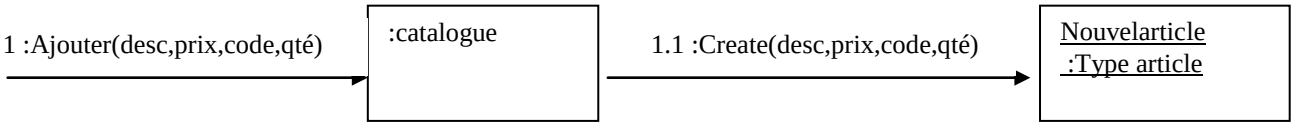
- Les messages conditionnels exclusifs (ou si sinon)



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

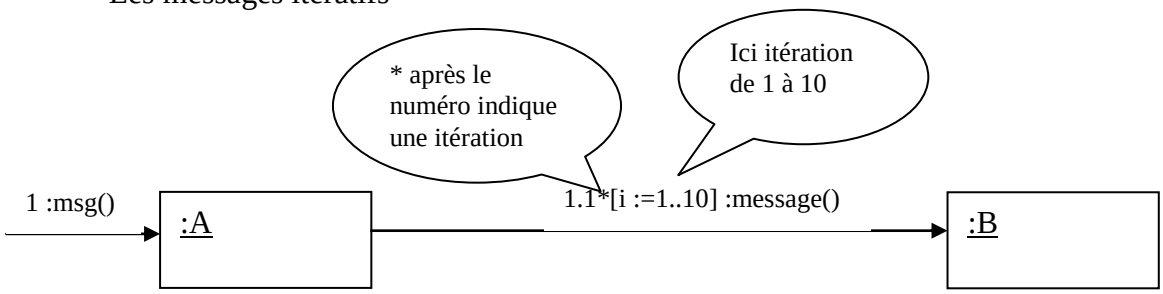
- Les messages d'instanciation

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 ?).

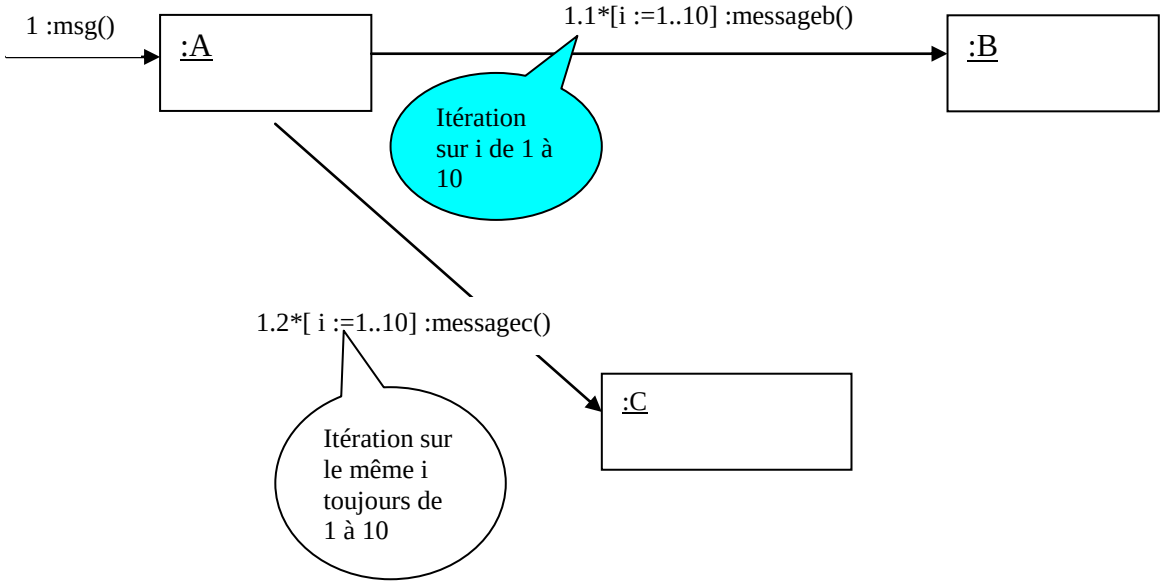
- Les messages itératifs



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

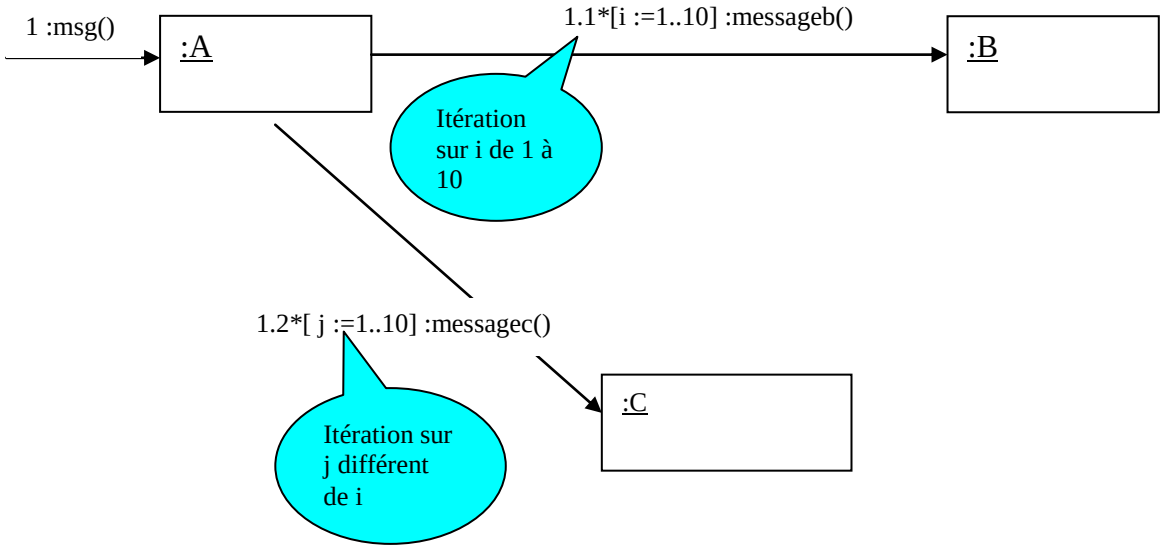
Ce sont des messages itératifs, où plusieurs messages sont envoyés dans l'itération.



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

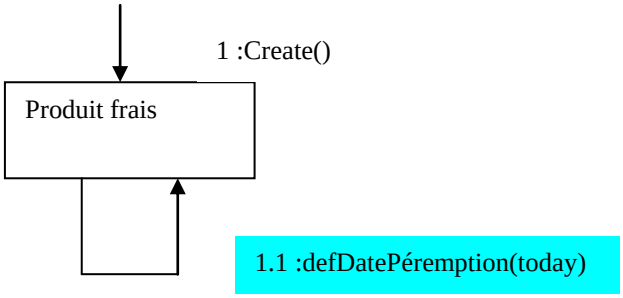
Ce sont des messages itératifs, où plusieurs itérations se suivent. Ici les boucles sont distinctes.



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

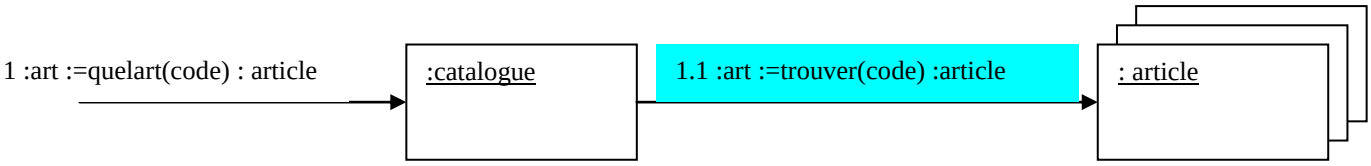
- Les messages vers this

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

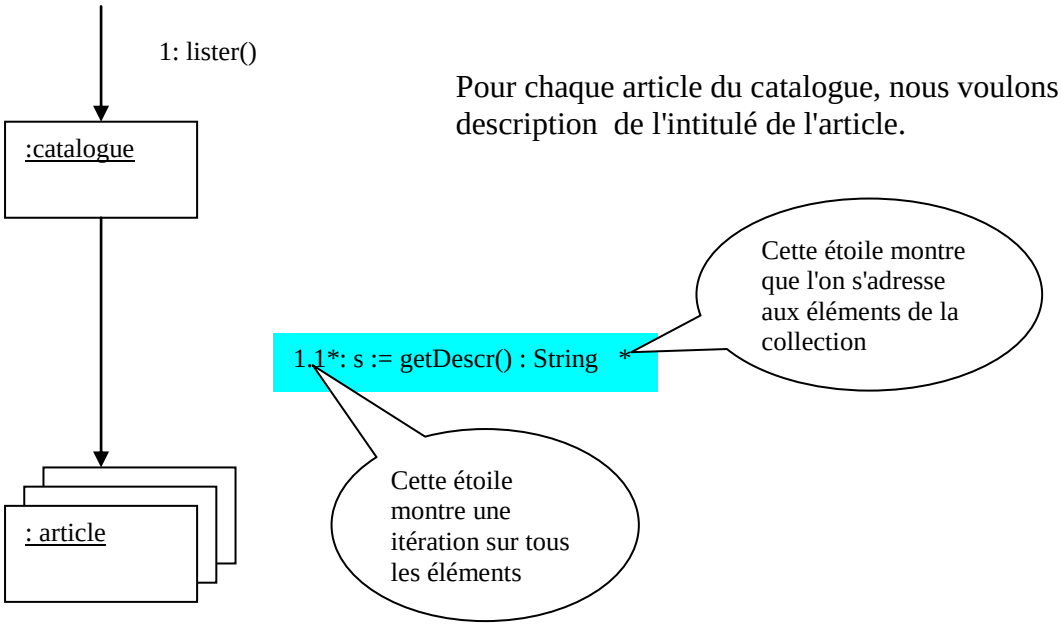


Le message trouver n'est pas envoyé à l'objet :Type art mais au multi objet, qui se traduira en programmation par une collection (tableau, vecteur, hashtable, etc).

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.

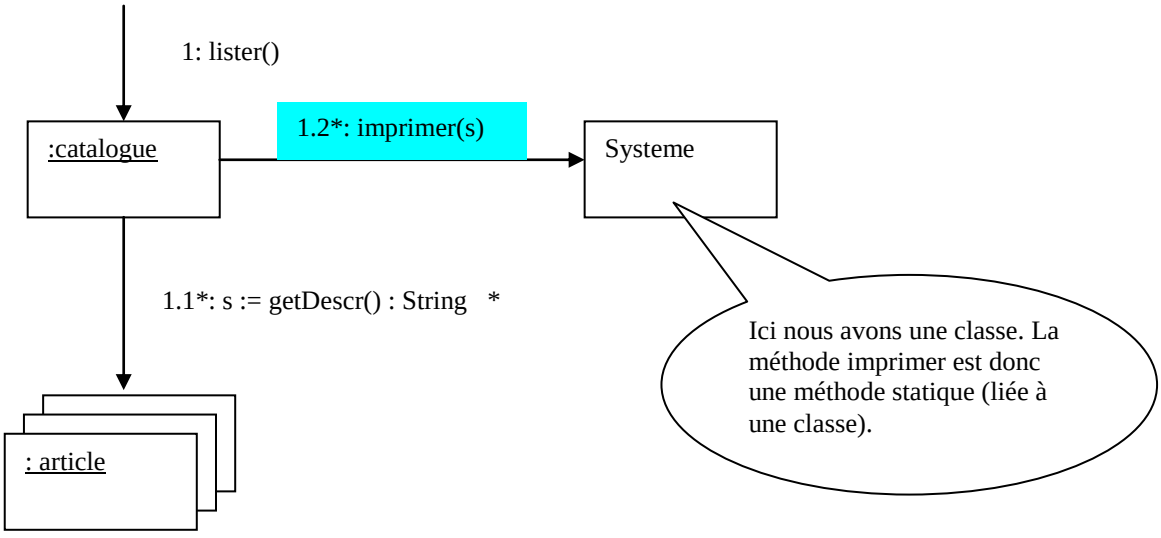
- Itération sur les éléments d'un multiobjet



Pour chaque article du catalogue, nous voulons la description de l'intitulé de l'article.

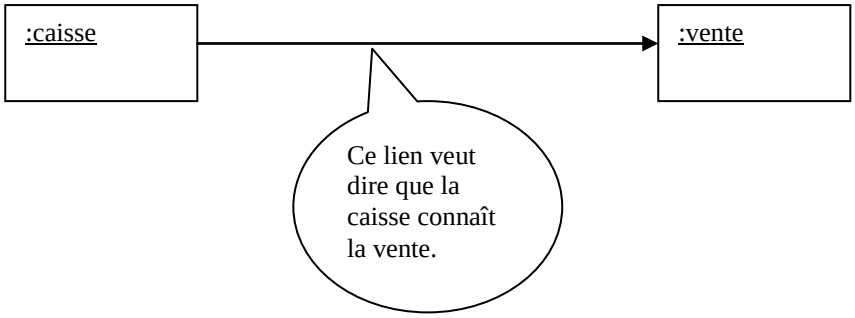
Il nous reste ici à imprimer la description obtenue. Pour cela il faut envoyer un message à une classe.

- 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.



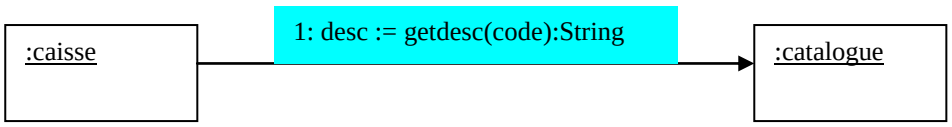
La classe caisse doit connaître la classe vente pour pouvoir lui parler:" vente , oh ma vente! Ajoute-toi un article de ce code.".

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.

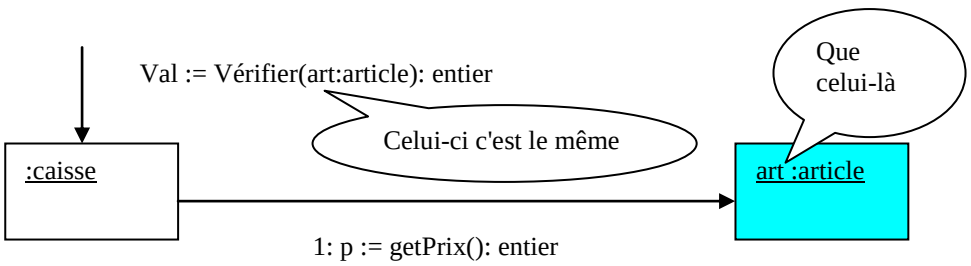
- La visibilité de type attribut.



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.

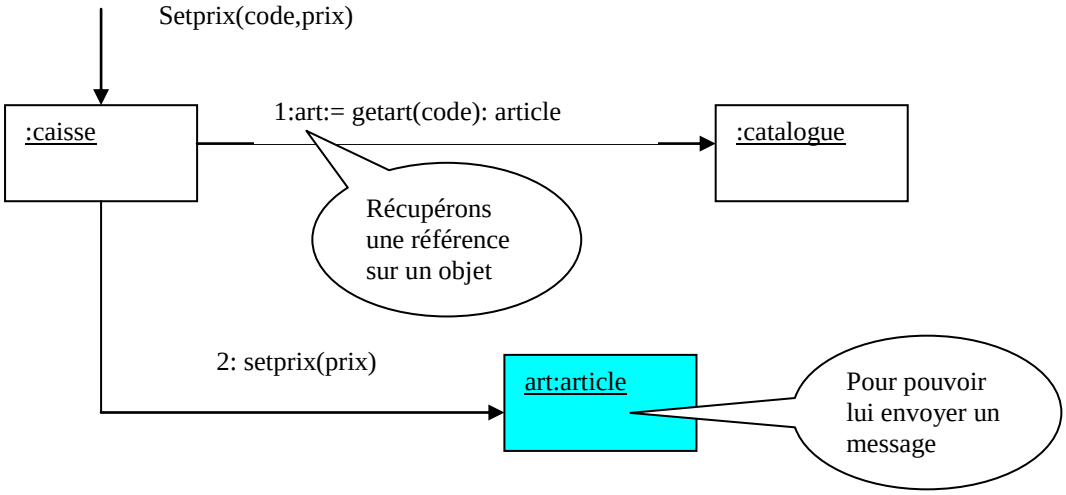
- La visibilité de type paramètre

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

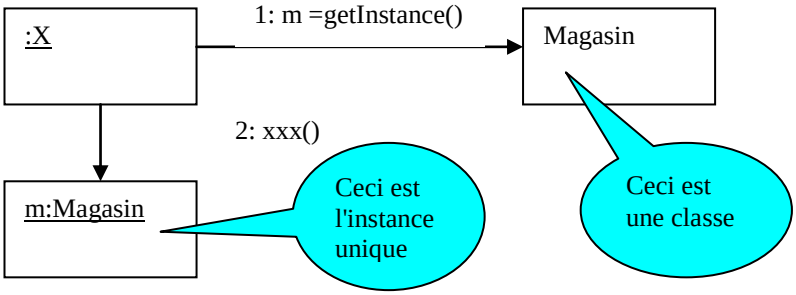
C'est l'utilisation par un objet d'une référence globale à un objet connu de tous. Cela peut aussi être le cas de variables globales dans le cas de certains langages. Nous comprenons bien que ce type de visibilité sera utilisé dans de rares cas, où un objet est omniprésent pour tous les autres objets,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

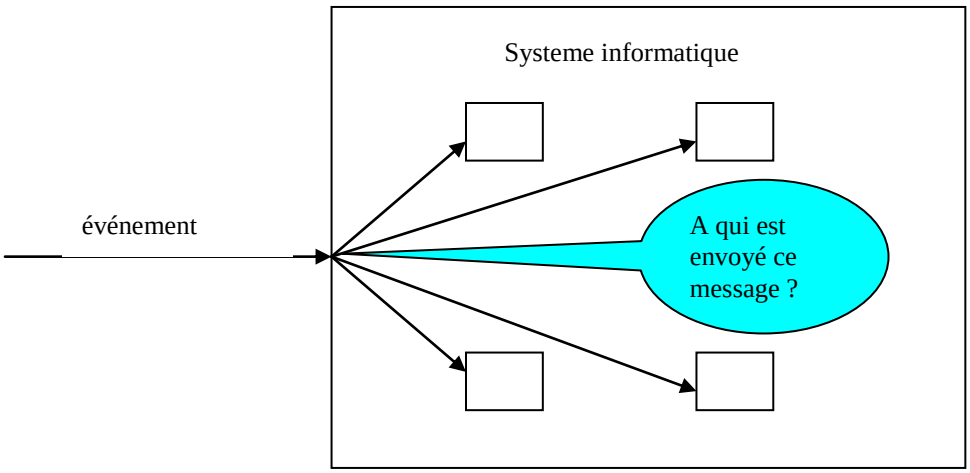
- La classe magasin aura une donnée membre statique instance.
- A la création (constructeur) du magasin la donnée membre sera initialisée à this (l'instance nouvellement créée du magasin).
- La classe magasin aura une fonction statique getInstance qui retournera l'instance du magasin. Ainsi n'importe 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.



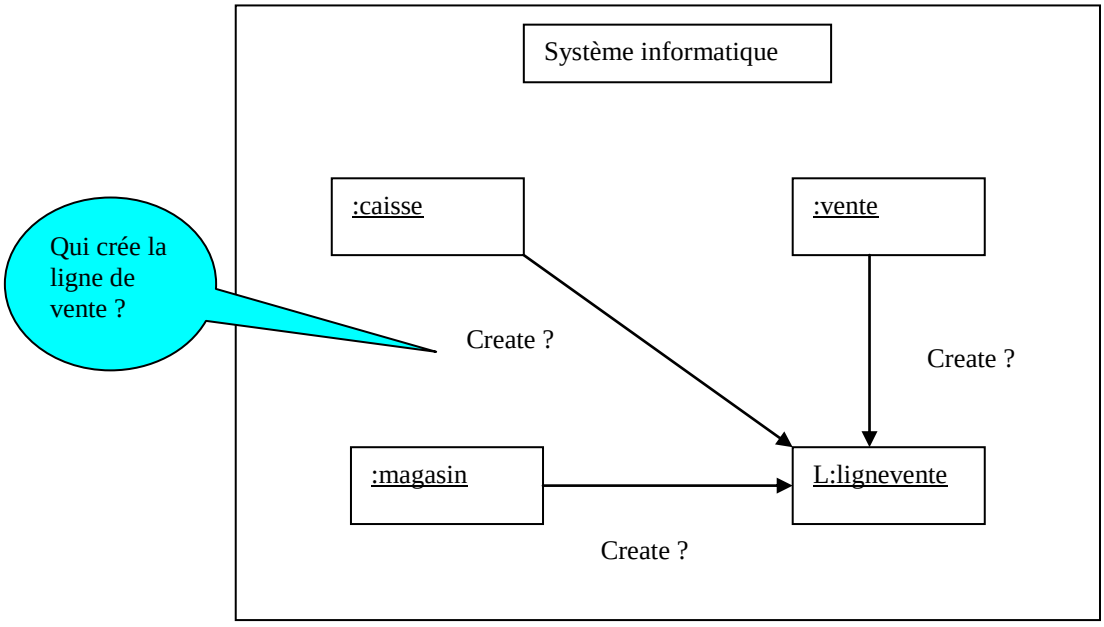
3) GRASP patterns

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).

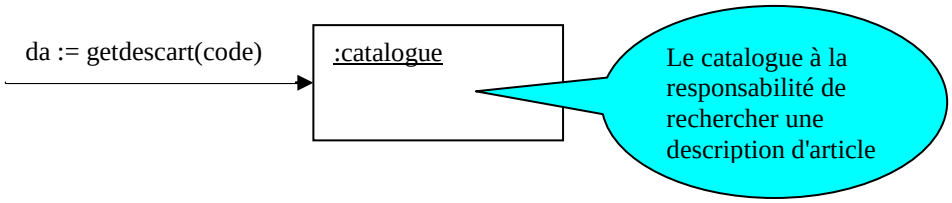
Les qualités du logiciel qui lui permettront de vivre, d'être corrigé, d'évoluer, de grandir dépendront complètement des choix que l'on va effectuer ici.

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 Responsability Assignement 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.

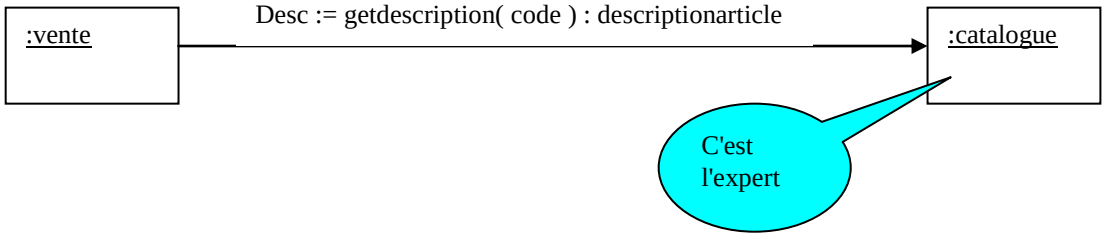
Ces classes à faible cohésion sont difficiles à comprendre, à réutiliser, à maintenir. Elles sont également fragiles car soumises aux moindres variations, elles sont donc instables.

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.

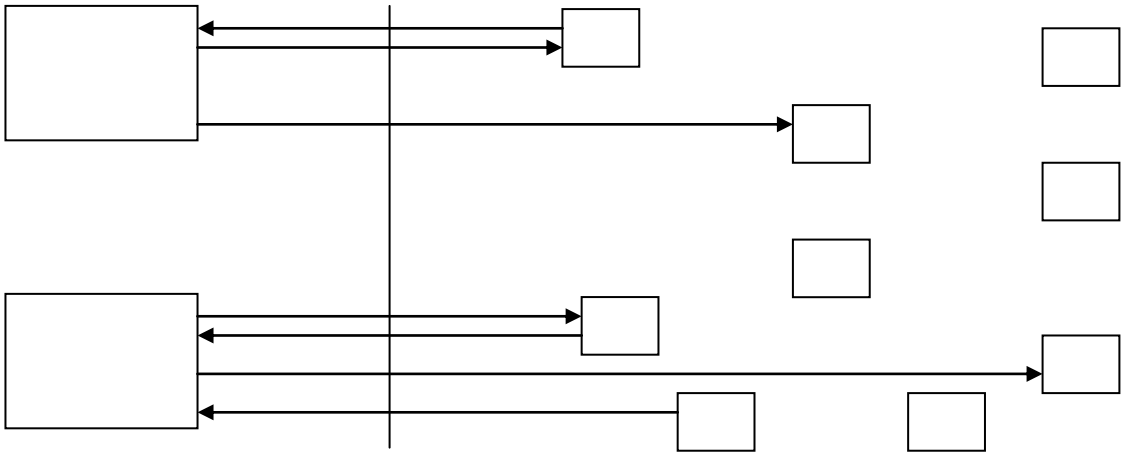
- A utilise souvent B.

  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.

Interface utilisateur Système informatique



Objets de l'interface utilisateur objets du sytème informatique
(Frame)

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.

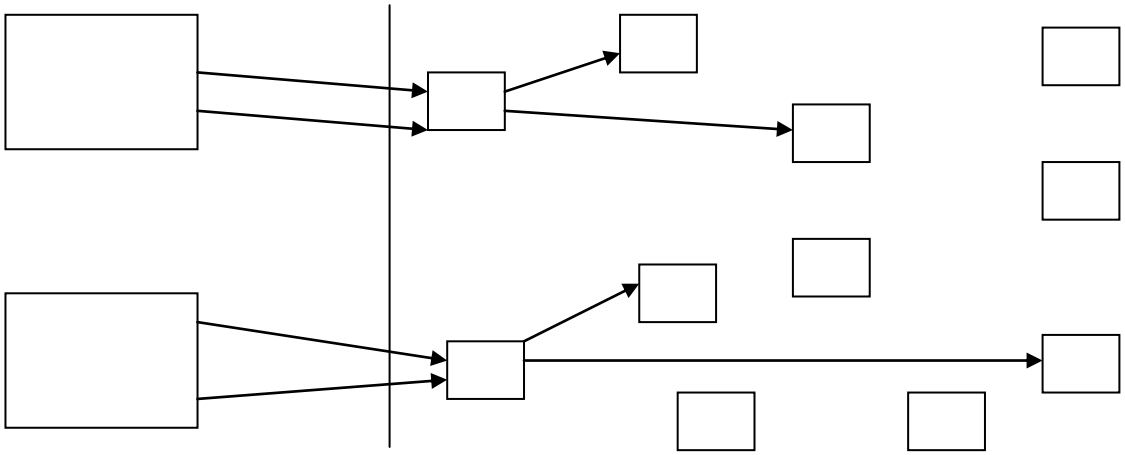
Les objets de l'interface utilisateur vont solliciter un objet d'interface, plutôt que de solliciter les objets métier eux-mêmes. Cet objet s'appelle un contrôleur.

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.

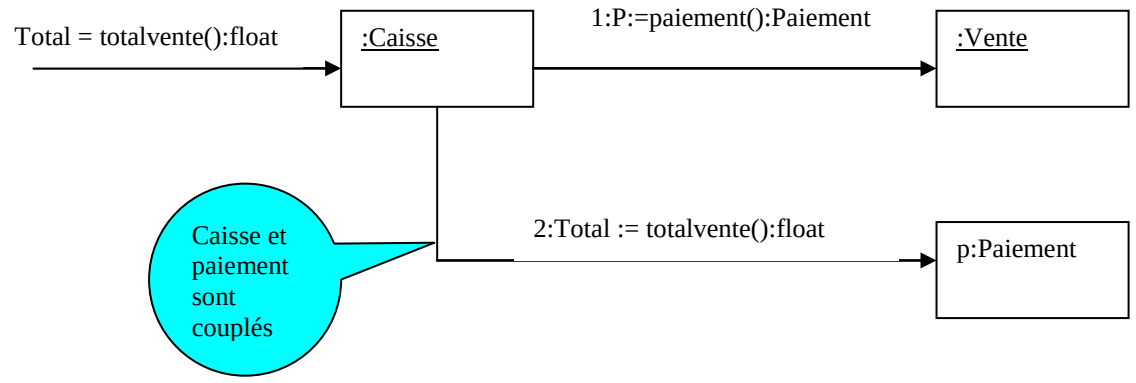
Quels sont les objets auxquels un objet à le droit de parler?

- 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.

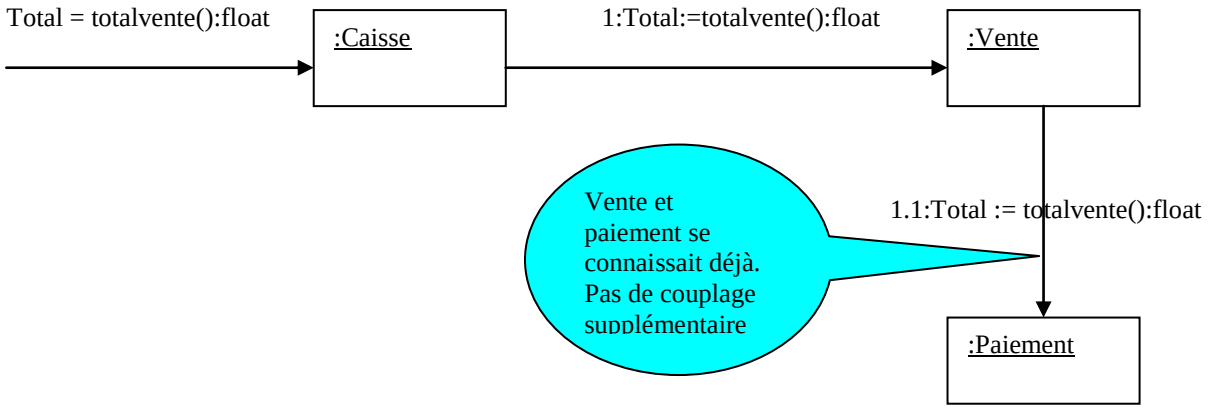
Les autres objets sont considérés comme des inconnus auxquels, nous le savons depuis la plus tendre enfance, il ne faut pas parler.

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.

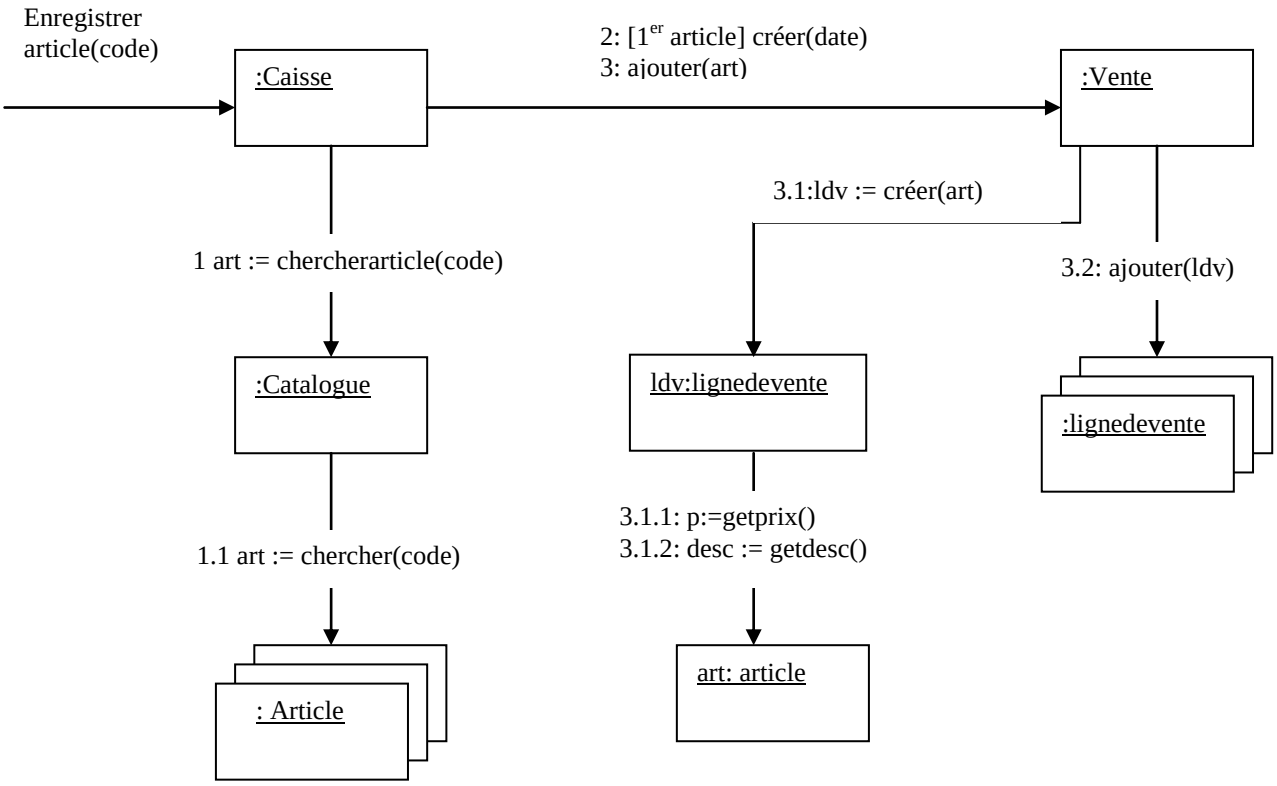
4) Diagramme de collaboration du magasin

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

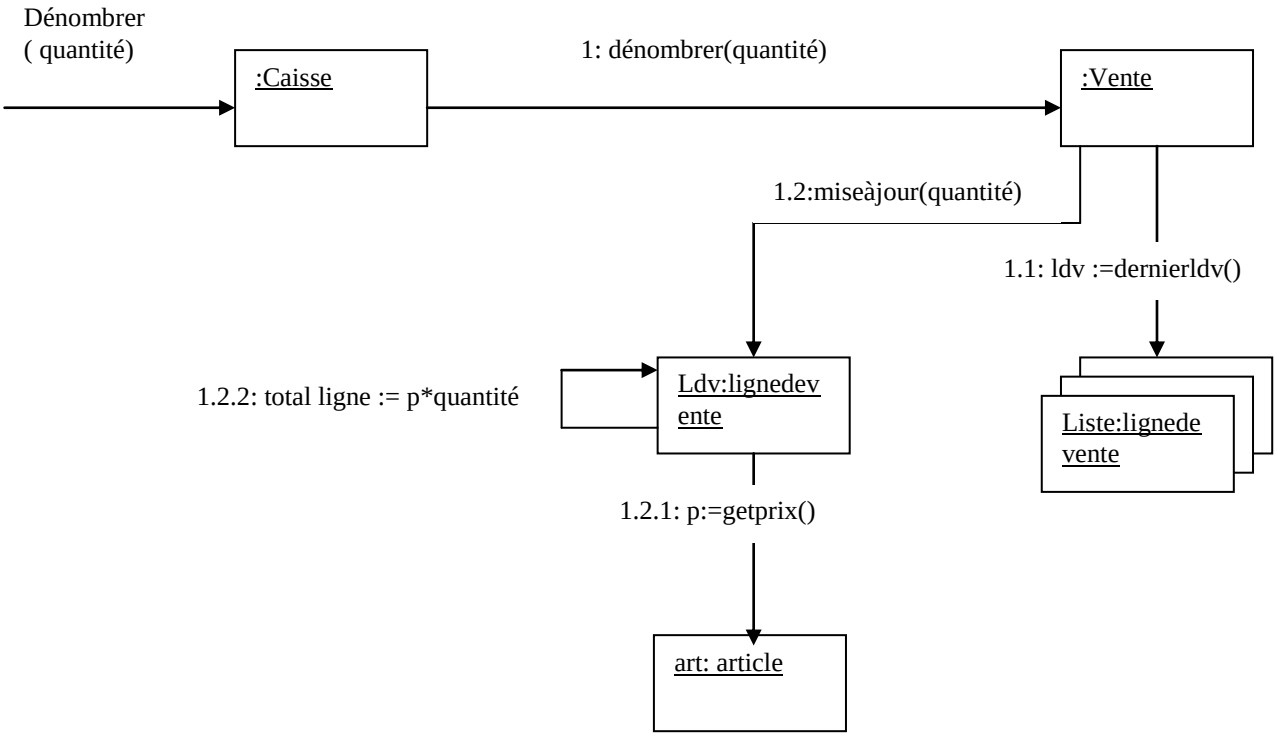
Nous allons faire autant de diagrammes de collaboration que nous avons fait de contrats d'opérations. Pour chaque contrat d'opération nous allons prendre l'événement du système comme message d'attaque de notre diagramme de collaboration, puis nous allons créer les interactions entre les objets qui, à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

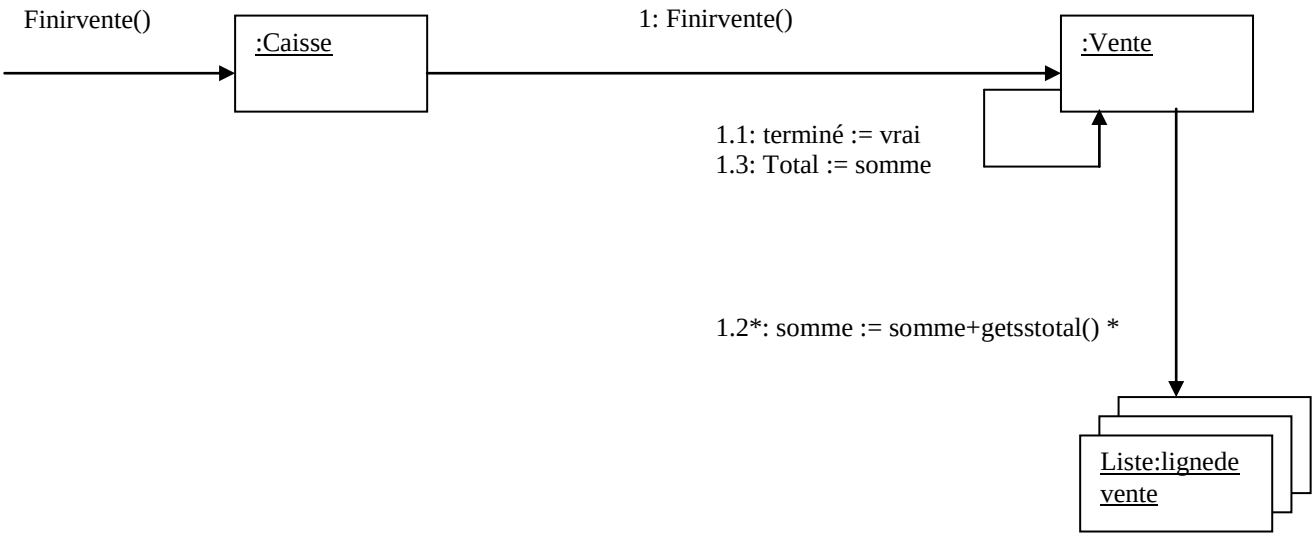
4.1) Diagramme de collaboration de Enregistrer un article



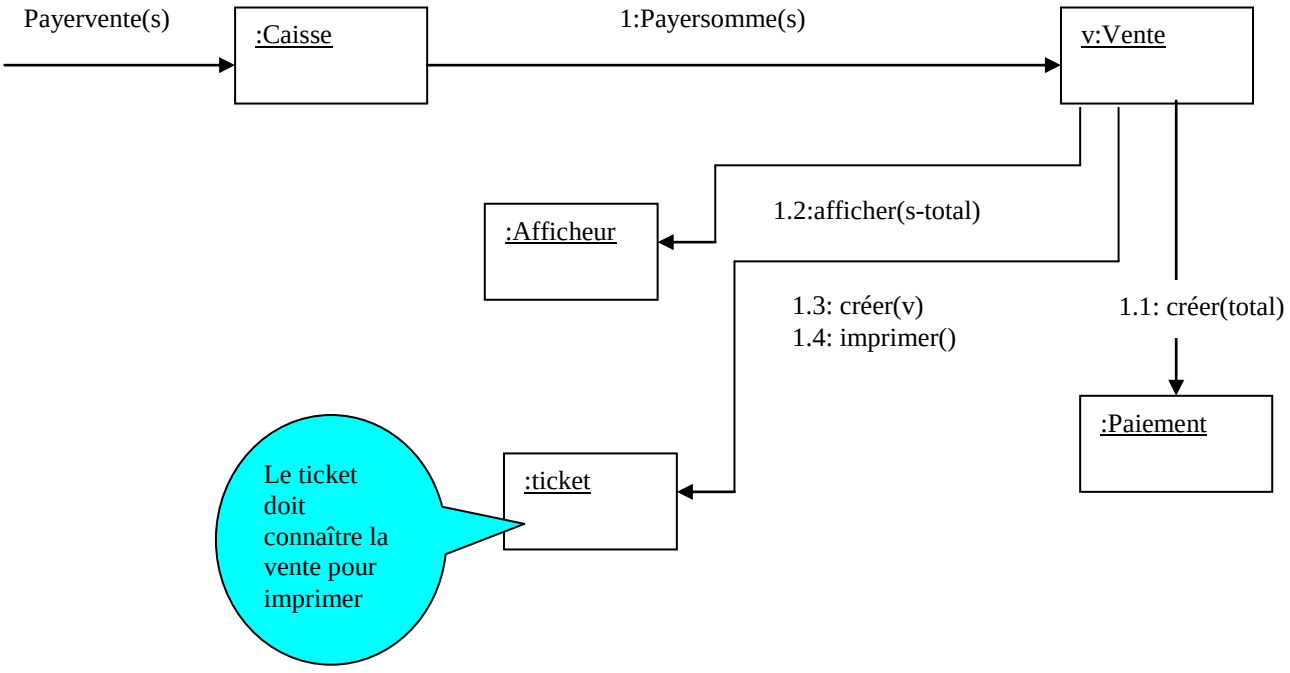
4.2) Diagramme de collaboration de dénombrer les articles identiques



4.3) Diagramme de collaboration de Finir la vente

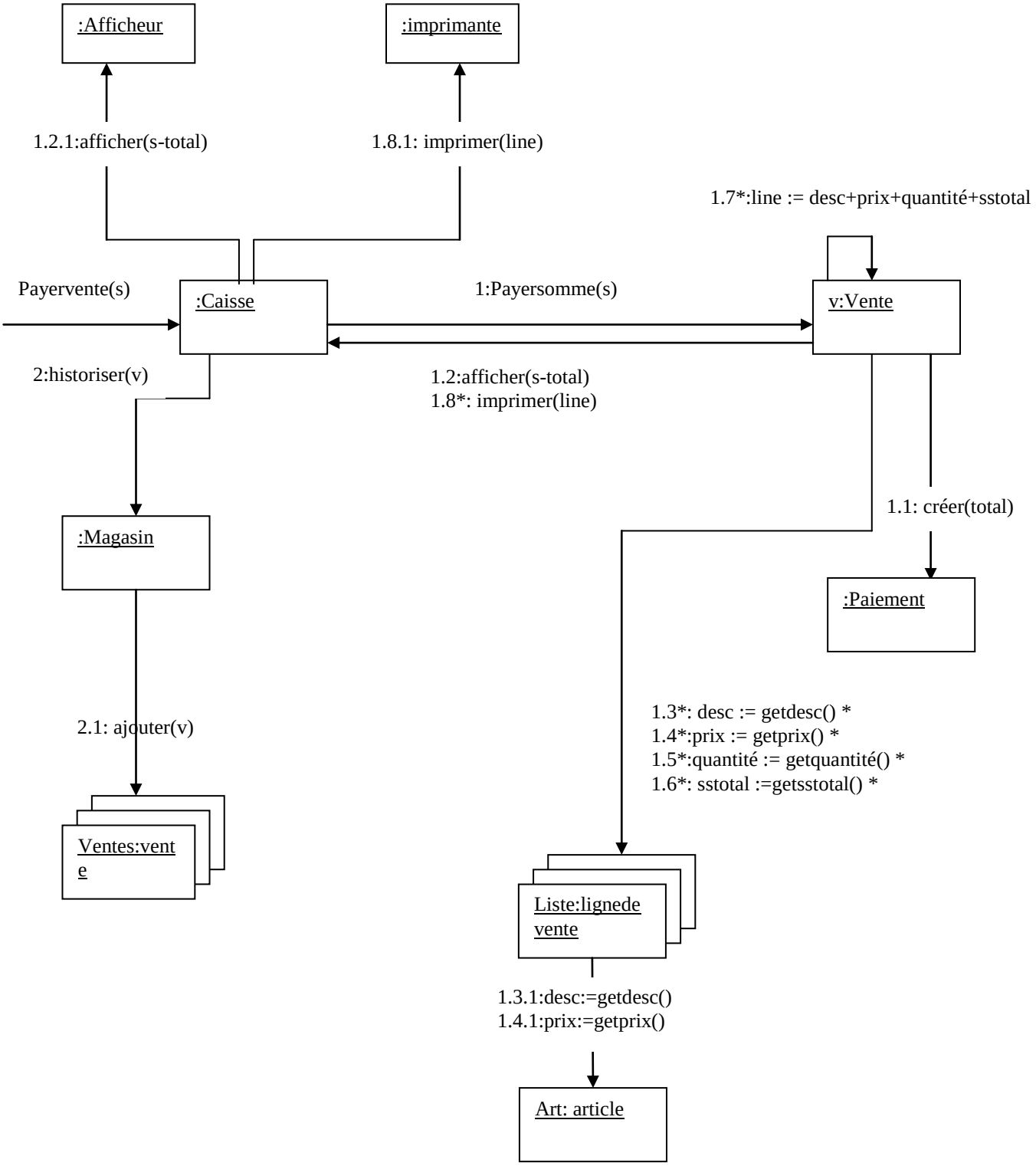


4.4) Diagramme de collaboration de Payer la vente



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.
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 modèliser 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.