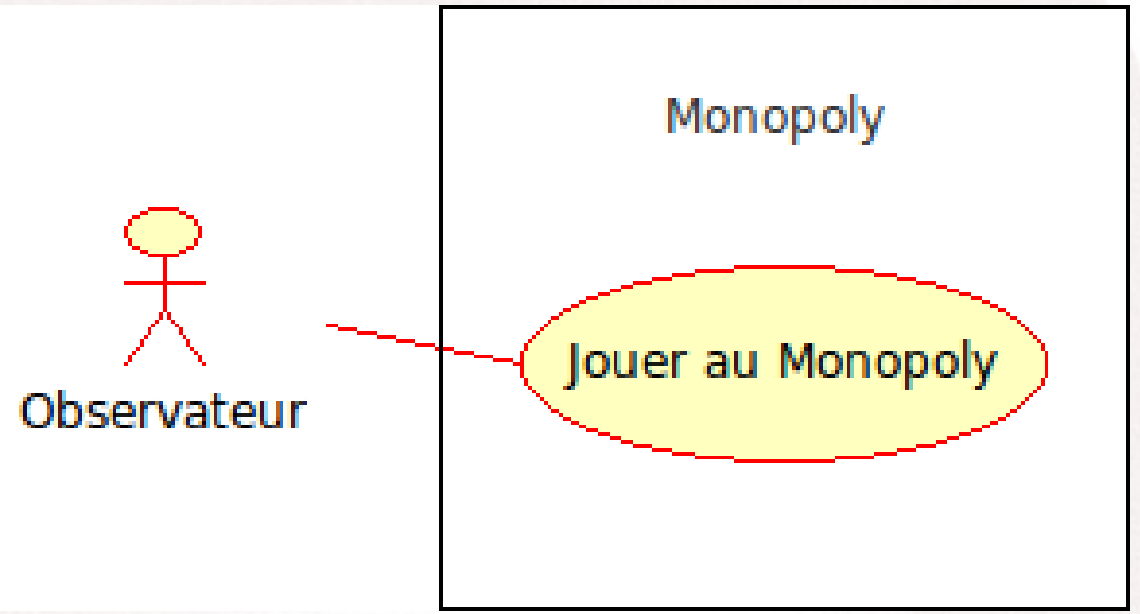
Monopoly (Craig Larman)

Premier use case :

Craig Larman se positionne dans le cadre suivant : Le jeu se déroule sous forme de *simulation*, donc, nous avons un **observateur**..



Use case Textuel

Application de Monopoly

Acteur principal : observateur

Intérêts :

L'observateur : il veut pouvoir observer facilement la simulation

Scénario principal (succès)

1) *l'Observateur* demande l'initialisation d'une nouvelle partie, il saisit le nombre de joueurs.

2)*l'Observateur* lance la partie.

3)Le *Système* affiche les coups joués pour le joueur suivant.

Répéter l'opération 3 jusqu'à avoir un joueur gagnant, ou arrêt par *l'Observateur*...

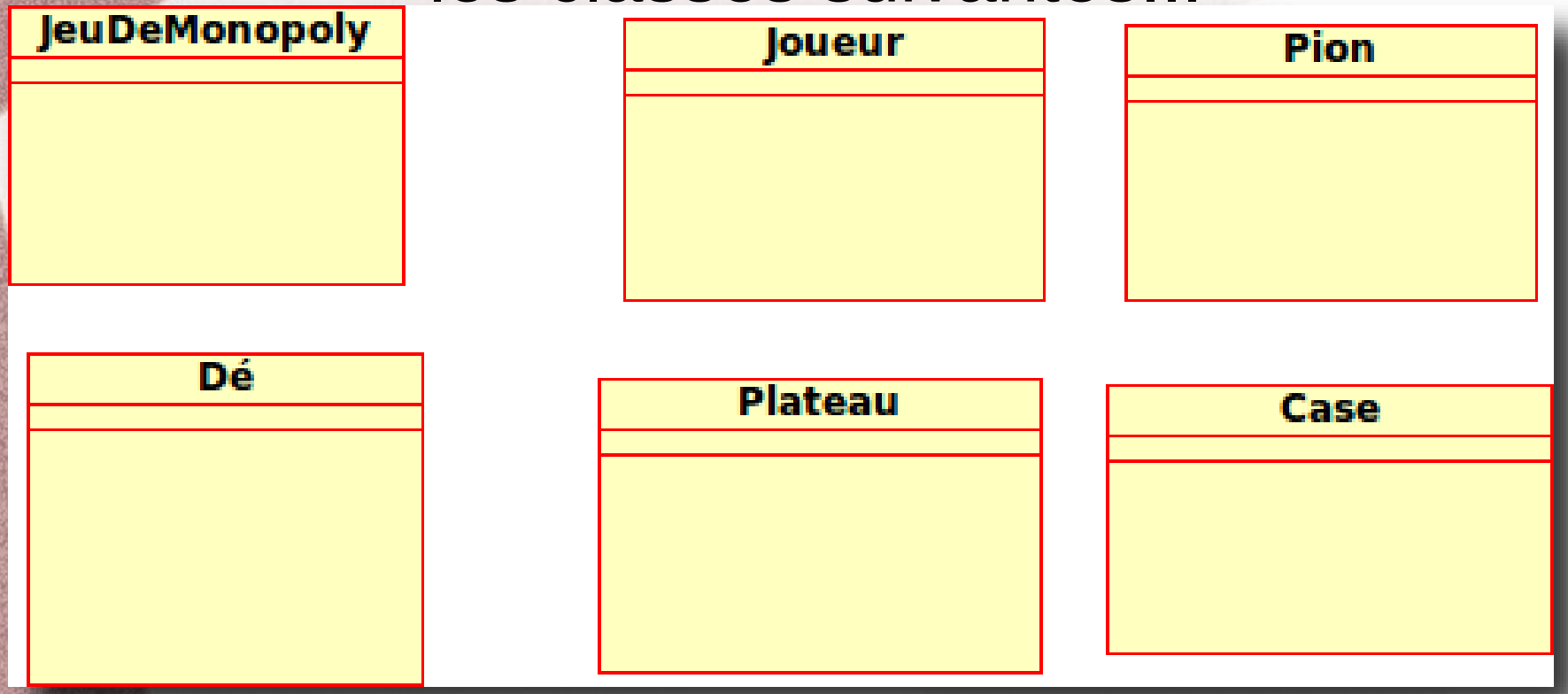
Inutilité...

Ce use case n'apporte pas grand chose.. Il est beaucoup trop générique, c'est vraiment une première approche.. Il s'agit d'une vision très globale, avec énormément de recul...

Mais dans le cadre dans lequel s'est positionné l'auteur, il est difficile de lister d'autres services offerts par le système... (on pourrait imaginer : arrêt de la partie, etc)...

Listes des classes conceptuelles

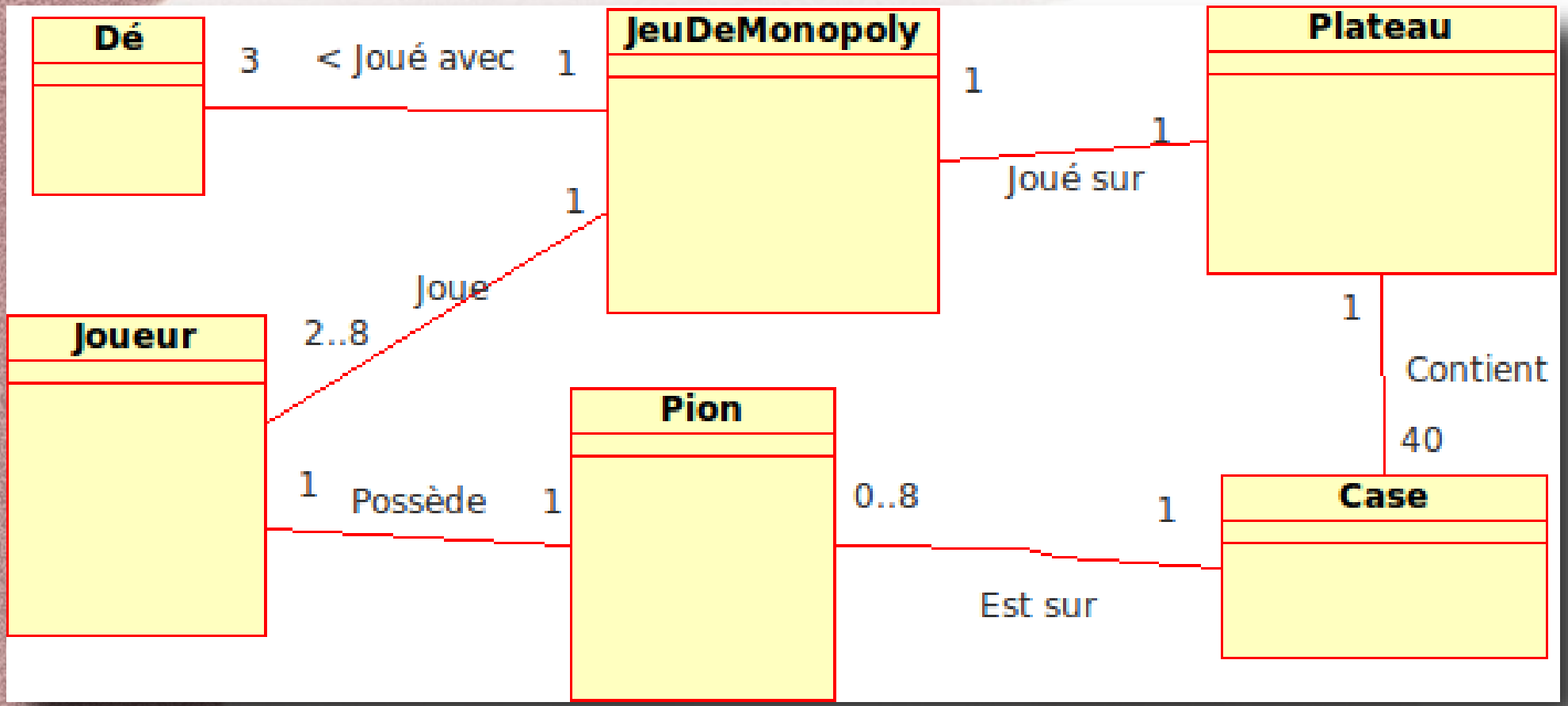
Une première approche permet de lister, de sentir, les classes suivantes...



Arrêtons nous ici pour cette première itération... (pas de carte, pas de maison, etc...)

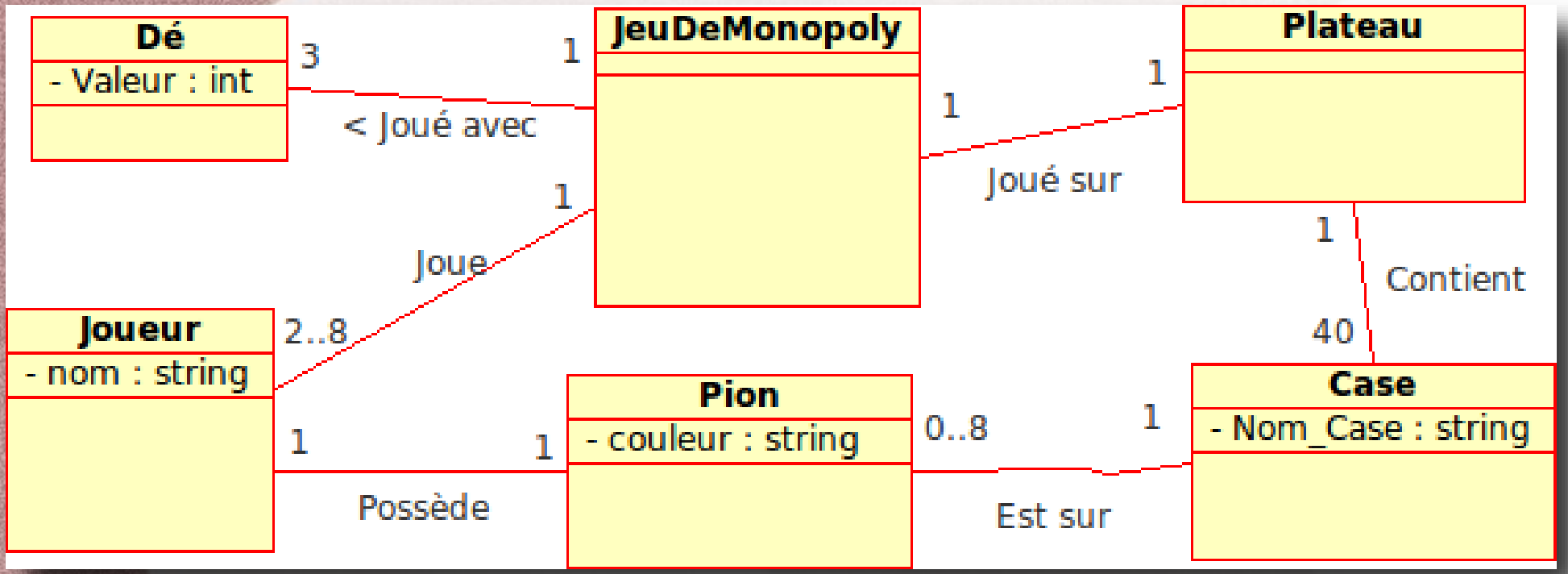
les premières associations...

C'est le premier ressenti, entre les premières classes détectées... Une première approximation...



Quelques attributs..

Dans cette itération, on ajoute les attributs majeurs.

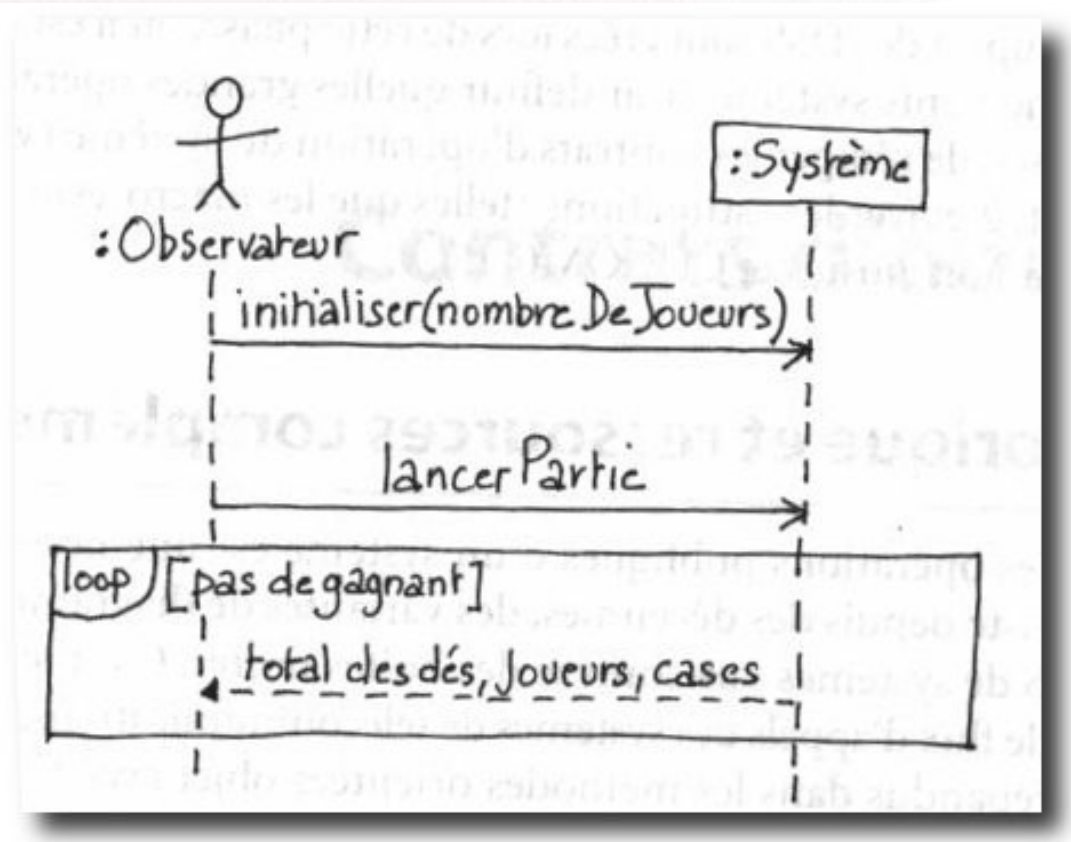


Il s'agit ici des attributs incontournables, ceux nécessaires à la première ébauche (le use-case définit que l'observateur veut voir la partie se dérouler). On appelle ce diagramme de classe le **modèle de Domaine...**

un DSS

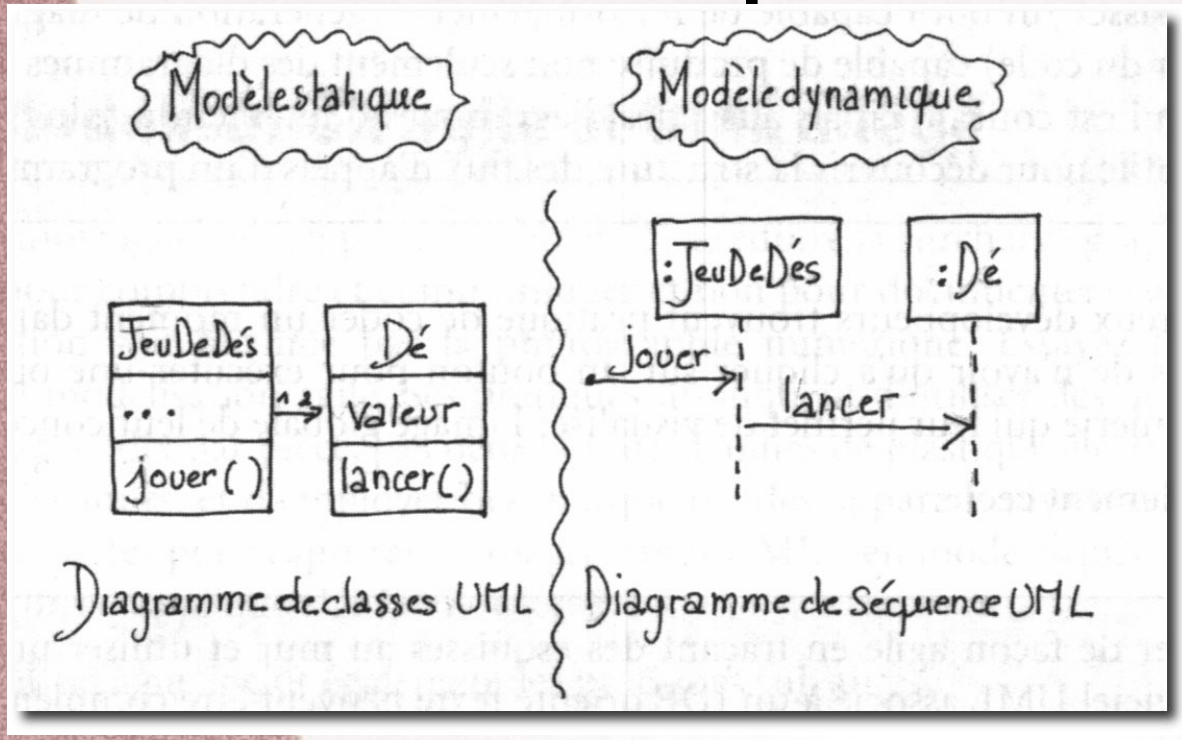
Trés pratique pour le coté dynamique de l'appli. Conseil de **Larman** (pour qui ce diagramme est le plus important) :

- *ne créez pas de DSS pour tous les scénarios.*
- *Ne passez pas trop de temps à les esquisser : quelques minutes à une demi-heure maxi*



Modélisation = dynamicité

Pour affiner notre modélisation, il faut utiliser ces *diagrammes d'interaction* (on va commencer à comprendre le fonctionnement lorsque le "*pneu va toucher le bitume*", c'est à dire qu'on va commencer à voir **qui travaille** avec qui...)



On affine notre diagramme de classe en observant les interactions entre objets, en regardant si le déroulement des actions détectées dans le use case est possible.

Application des Design Pattern

Premier pattern : **Créateur** : Qui crée les Cases ? Le Plateau ! Ce qui nous fait pencher en plus pour une composition (*les Cases meurent avec le Plateau)*. De plus, Plateau est **expert** en Case...

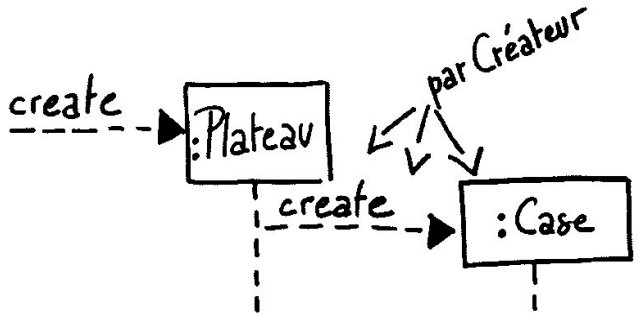


Figure 16.4 • Application du pattern Créateur dans un modèle dynamique.

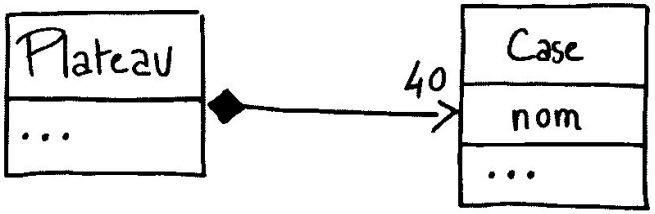
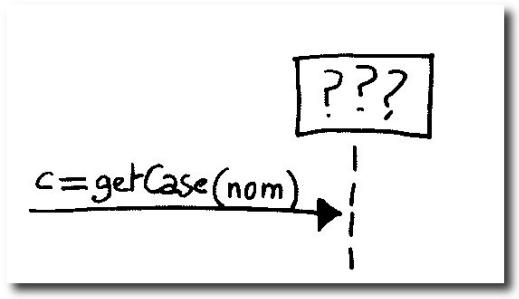
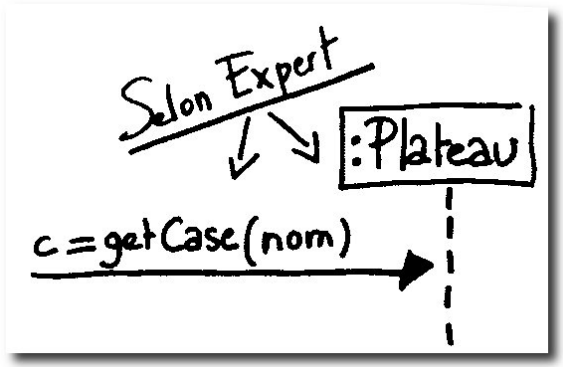


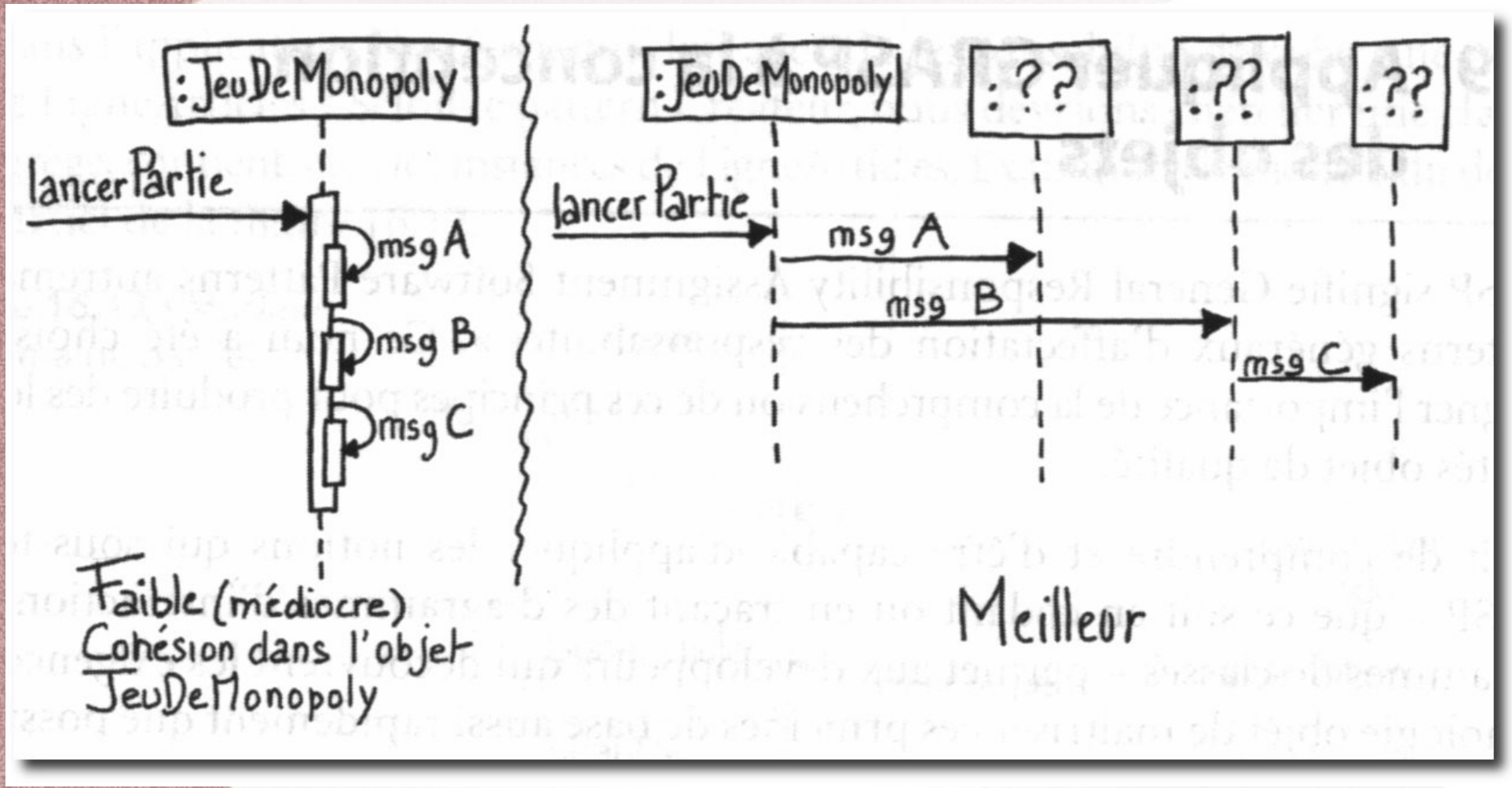
Figure 16.5 • Dans un DCC du Modèle de Conception, *Plateau* a une association d'agrégation composite avec *Case*. Nous appliquons Créateur dans un modèle statique.





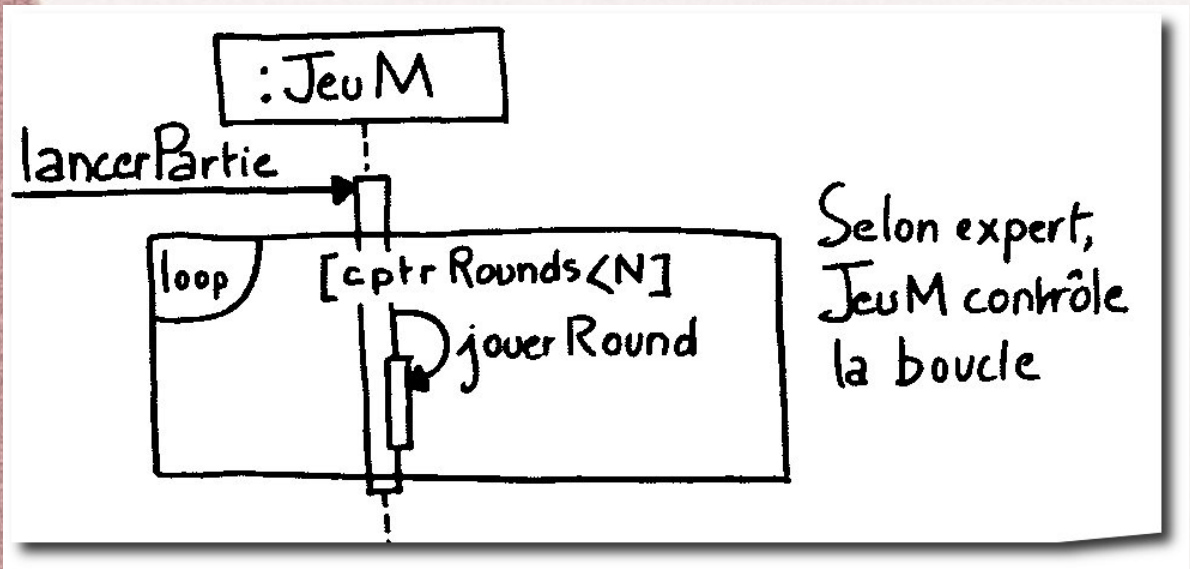
Design Pattern : Forte Cohésion...

... qui nous pousse à ne pas mettre toutes les **responsabilités** dans la classe JeuDeMonopoly



Design Pattern Contrôleur...

Qui pilote l'ensemble ? L'idée principale, c'est que c'est *JeuDeMonopoly* qui endossera cette responsabilité...



JeuDeMonopoly connait les joueurs. Il peut faire jouer leur tour à chacun, ce qui s'appelle un round.

JeuDeMonopoly pourra être le **singleton**, et comptera les rounds.

Réflexion sur le tour...

(itération 2 – on va un peu plus loin)

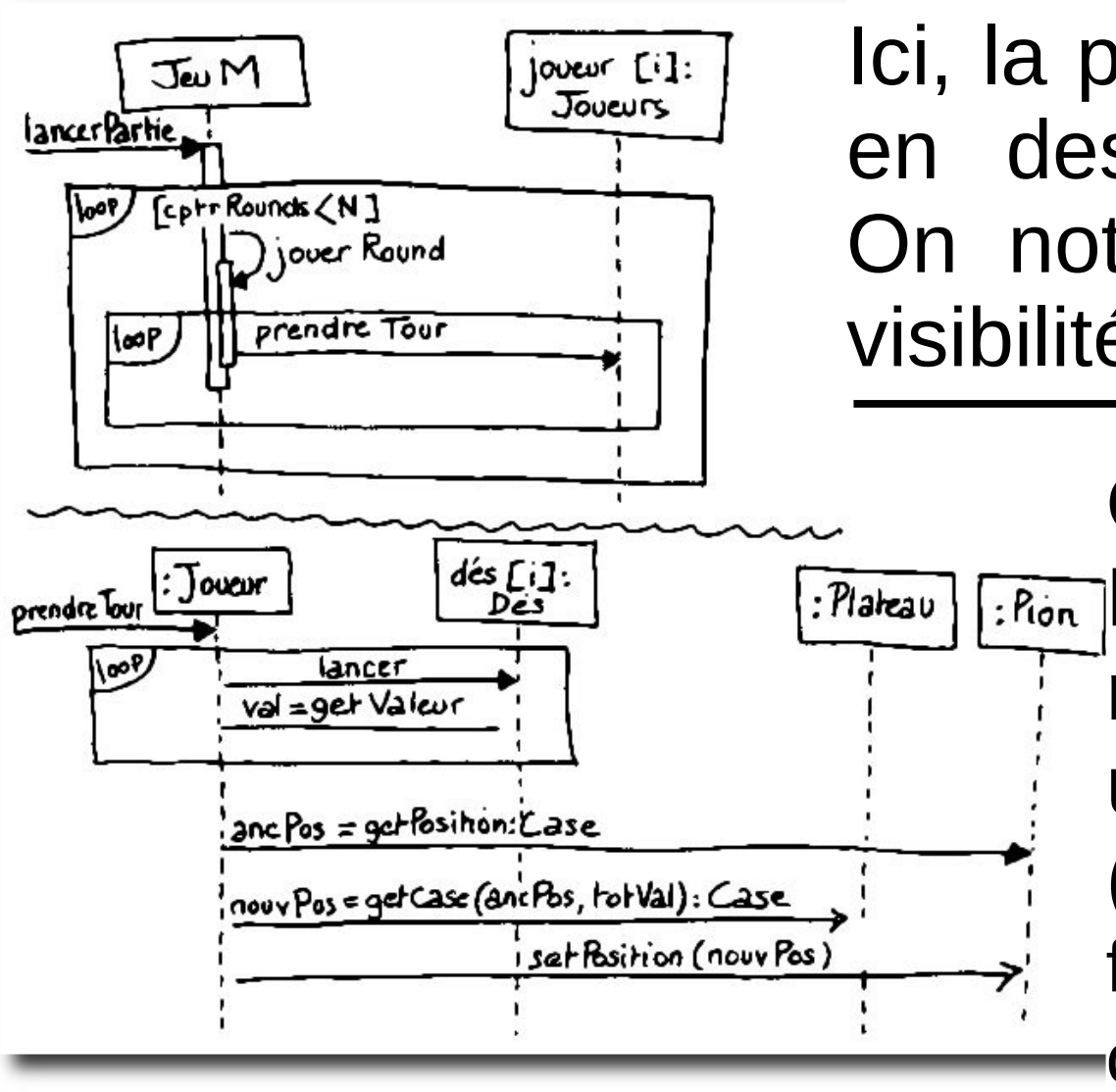
Chaque joueur :

- récupére sa valeur de dé
- calcule sa nouvelle case
- déplace le pion sur la nouvelle case

Est ce possible ? Le *Jeu* connait chaque *Joueur*. Chaque *Joueur* connaît son *Pion*, et qui lui sait sur quelle *Case* il est... L'ensemble semble cohérent.

Le *Dé* doit pouvoir être lancé, et sa valeur récupérée. Le *Joueur* peut récupérer sa *Case*, et demander au *Plateau*, qui connaît toutes les *Cases*, la nouvelle (en fonction du résultat des *Dés* et de l'ancienne)

Représentation Dynamique du tour



Ici, la première approche, en dessous, le détail... On note les besoins en visibilité de *Joueur*...

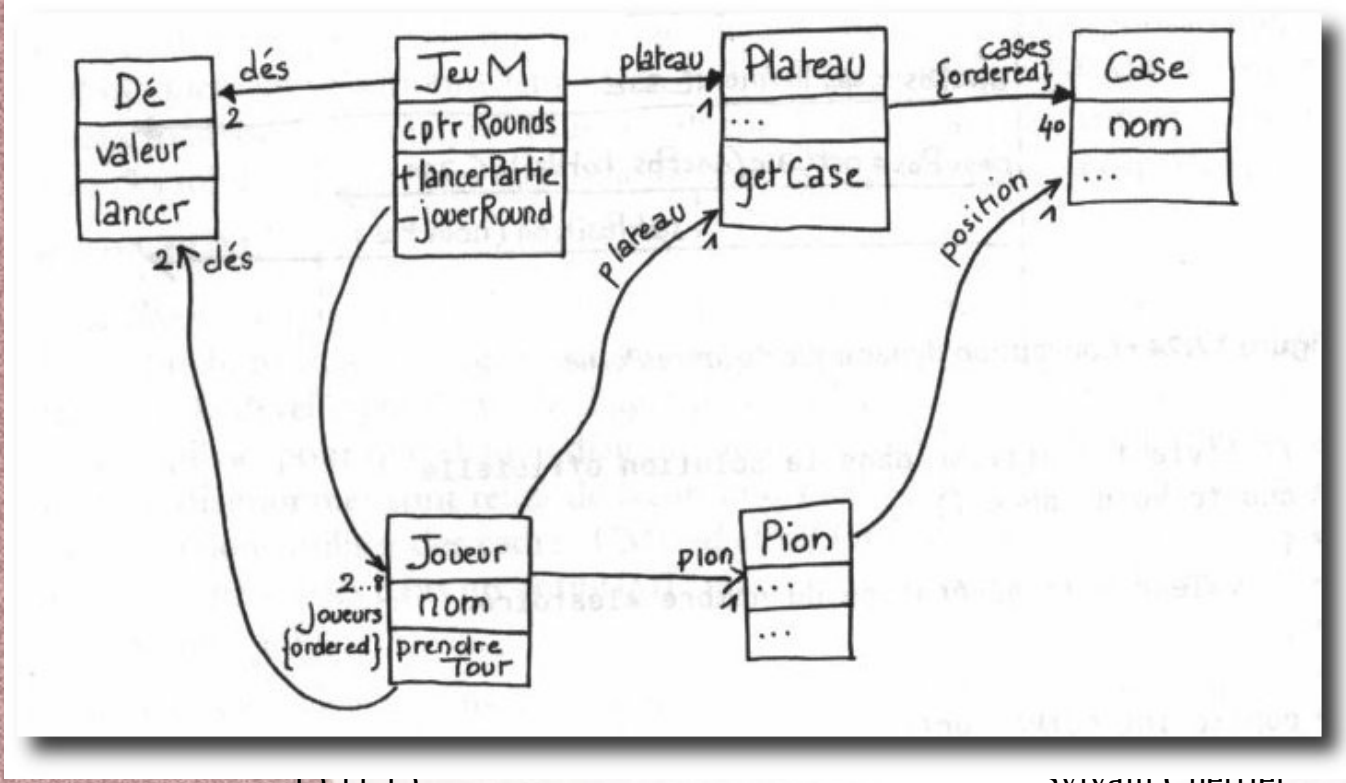
Question sur *Dé* : Pourquoi y a t'il une méthode *lancer*() et une autre *getValeur*() (et non une méthode qui fait les deux d'un coup) ?

Représentation Dynamique suite

Sur le graphique précédent, on remarquera la façon de découper les actions.
On donne le détail de 'prendre Tour()' en faisant un autre diagramme, qui contient l'action appellante, à gauche (la flèche), pour mémoire.
On peut observer les lignes de vie de chaque objet, et identifier l'ensemble des participants de l'action. Et **vérifier** que les messages qu'ils s'échangent sont **cohérents**, **plausibles** (Que celui qui émet le message dispose des arguments, et que celui qui le reçoit peut répondre)...

Les modifications sur les Classes

L'analyse du déroulement du tour nous montre des modifications à réaliser sur le diagramme de Classe. Il s'agit ici d'une vue plus proche du programmeur, qu'on appelle *Diagramme des Classe de Conception (DCC)*...



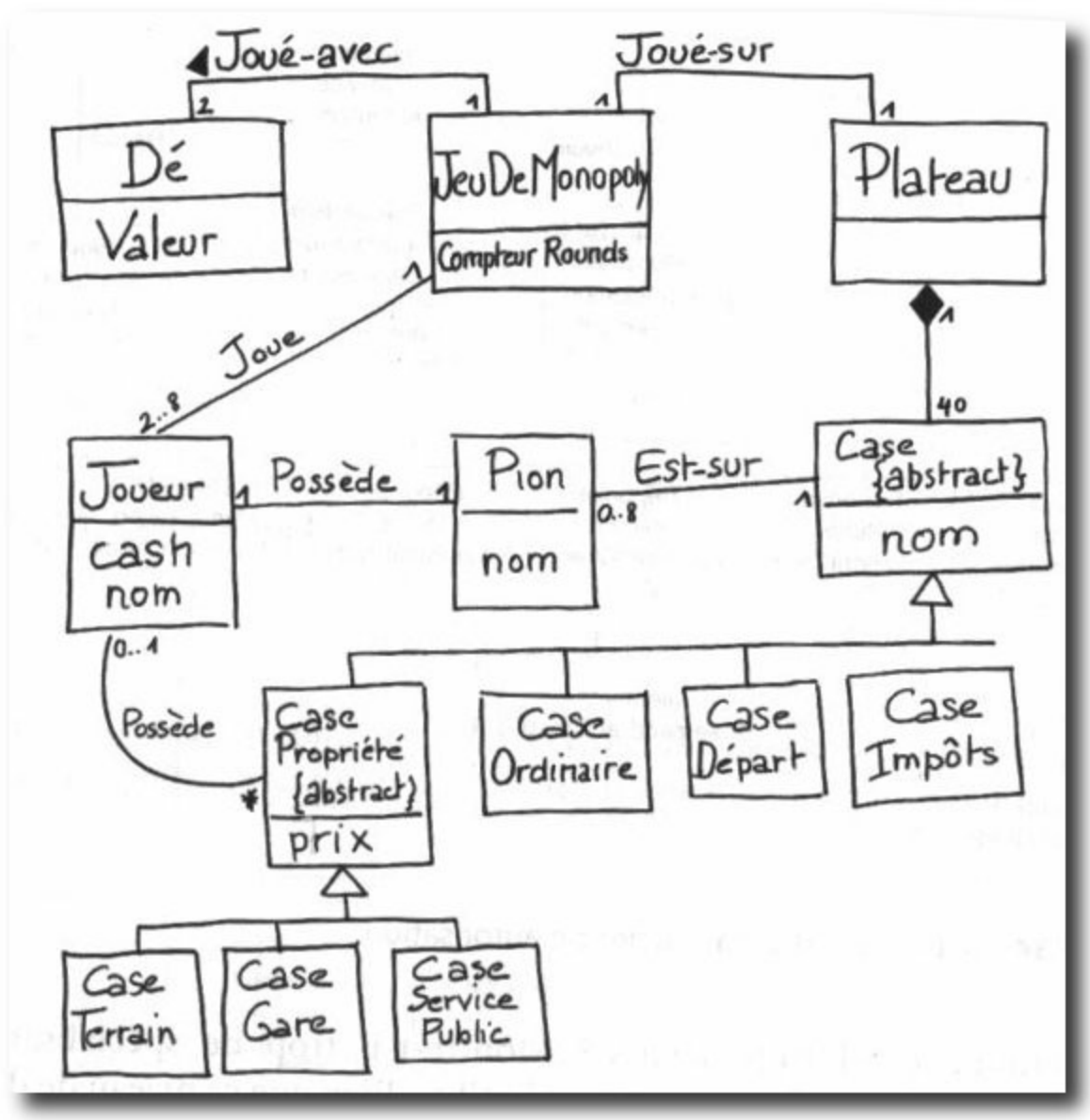
Ce *diagramme* ne remet pas en cause le premier. Il est plus orienté logiciel, car il résoud les problèmes du programmeur.

Itération suivante... Les cases

On affine maintenant notre solution pour les Cases...

Le **polymorphisme** semble assez évident.. et à 2 niveaux..

Nous sommes ici dans un *diagramme* de classe de niveau *Domaine*, générique.



On progresse...

On en profite pour faire apparaitre d'autres idées.. le tirage des dés, en un seul objet (un cornet de *Dés*)...
Et la gestion de l'arrivée sur un case, qui est assez riche... Devient une méthode **atterrir Sur(*Joueur* j)**...

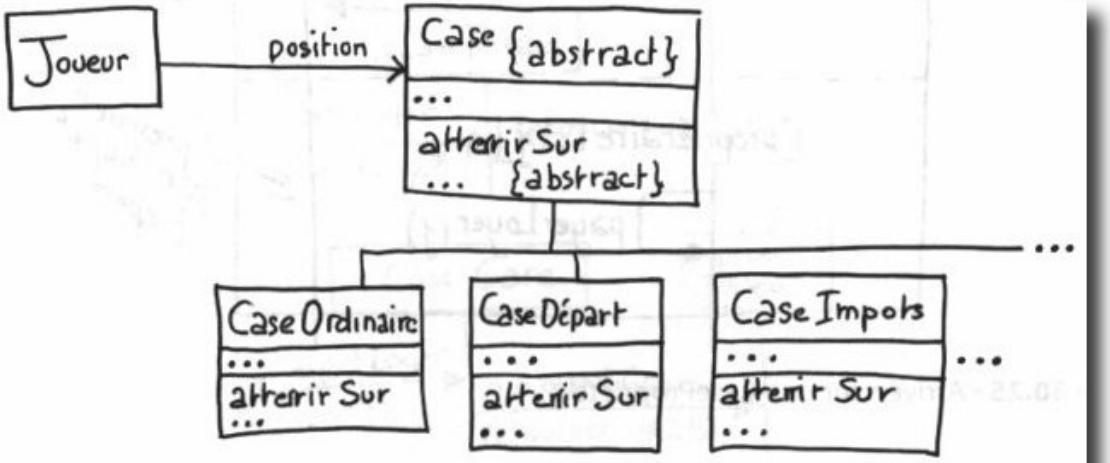


Figure 30.23 • DCC de la stratégie de conception polymorphe *atterrirSur*.

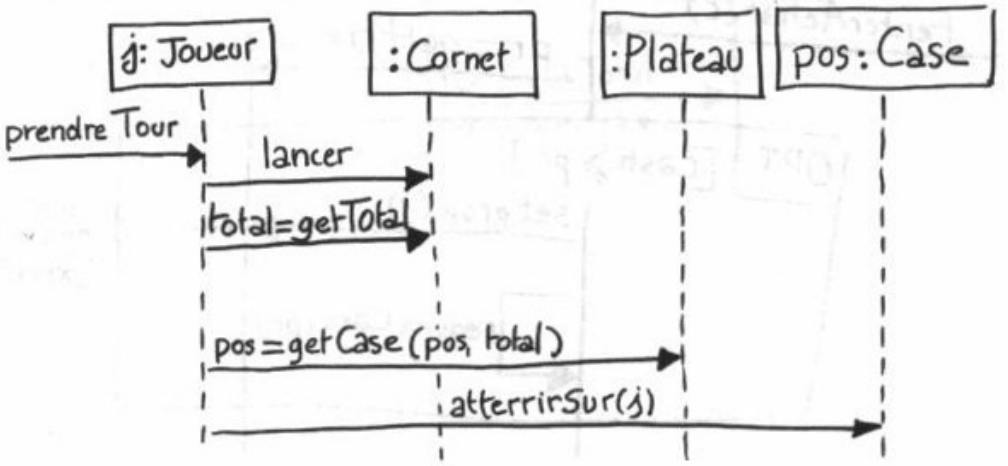


Figure 30.24 • Collaborations dynamiques pour la stratégie de conception *atterrirSur*.

Les échanges

Cette partie **dynamique, l'étude des échanges lorsque l'on tombe sur une case** est certainement la plus intéressante. .
Si c'est une *propriété*, il faut savoir si elle est *libre* (on peut alors *l'acheter*), sinon, il faut payer un *loyer*.. A qui ? Au *propriétaire* de cette **Case**...
Ici, le *diagramme dynamique* va nous permettre de nous projeter dans le fonctionnement de l'application, et de valider la possibilité de gestion des différents cas de figure...
Avez-vous remarqué qu'un **Joueur** se passe en tant qu'argument dans **atterrirSur(*Joueur* j)** ??

Détail dynamique d'AtterrirSur()

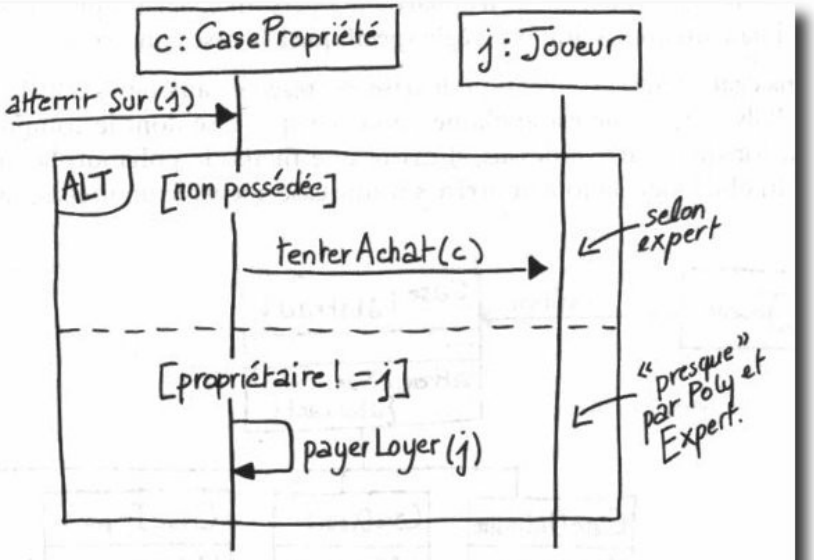


Figure 30.25 • Arrivée sur une CasePropriété.

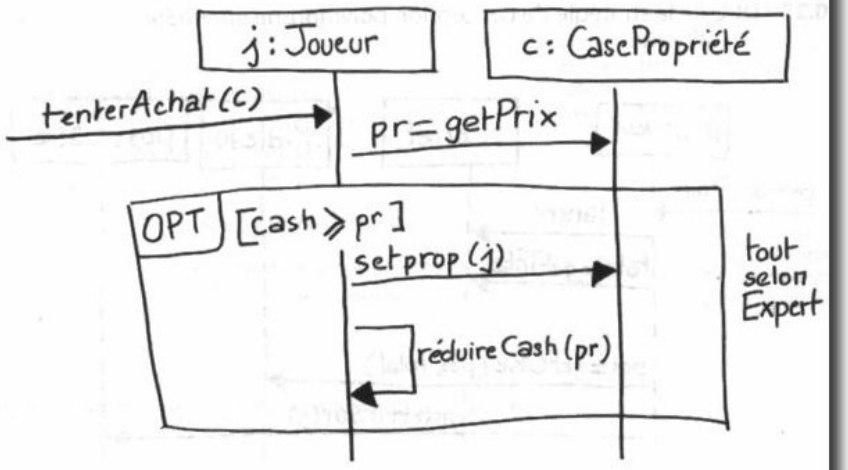


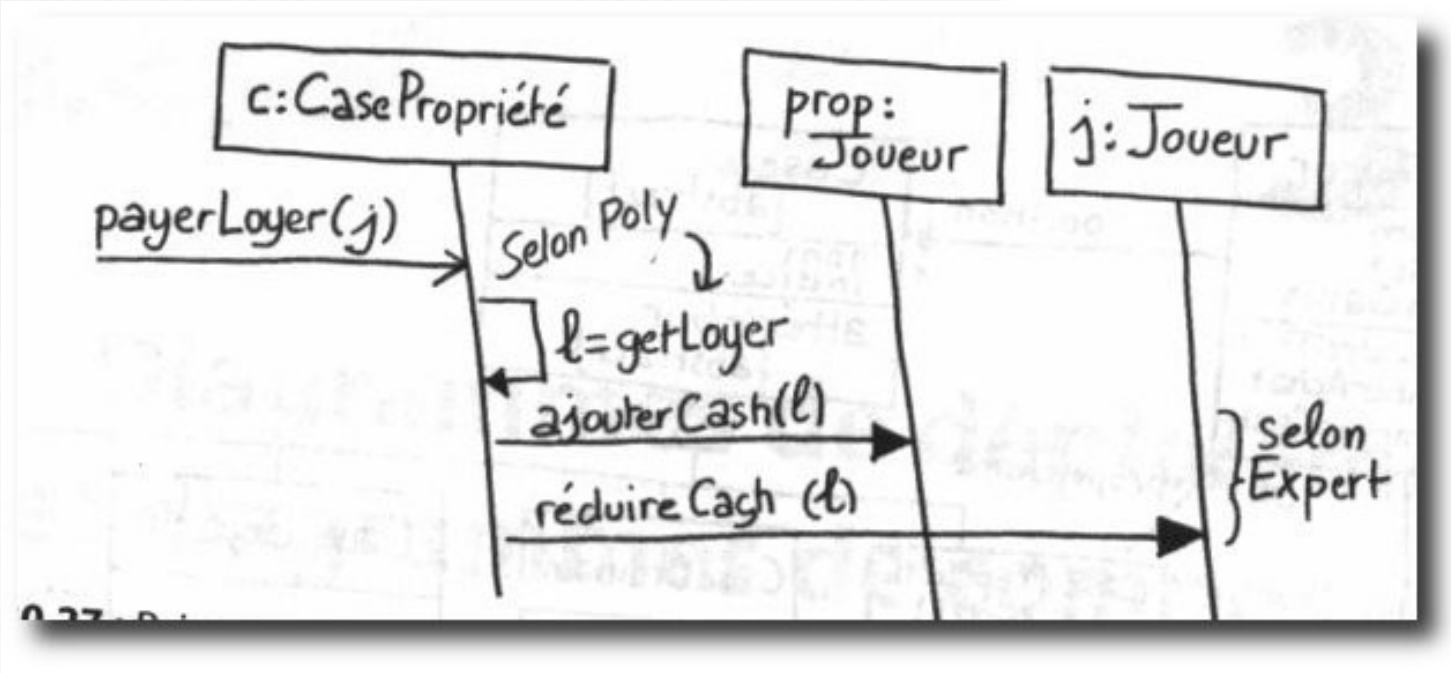
Figure 30.26 • Tentative d'achat d'une propriété.

Le **Joueur** émet le message *atterrirSur*(Joueur j) vers une **Case** en se passant en argument. La **Case** peut alors comparer si cette personne est *propriétaire* ou pas. Et si la **Case** n'est pas possédée, elle envoie le message *tenterAchat*() au **Joueur** j (qu'elle connaît), en se passant elle-même en argument !

Ce **Joueur** J pourra lui même se repasser en argument de *setProp*() à destination de la **Case** ..

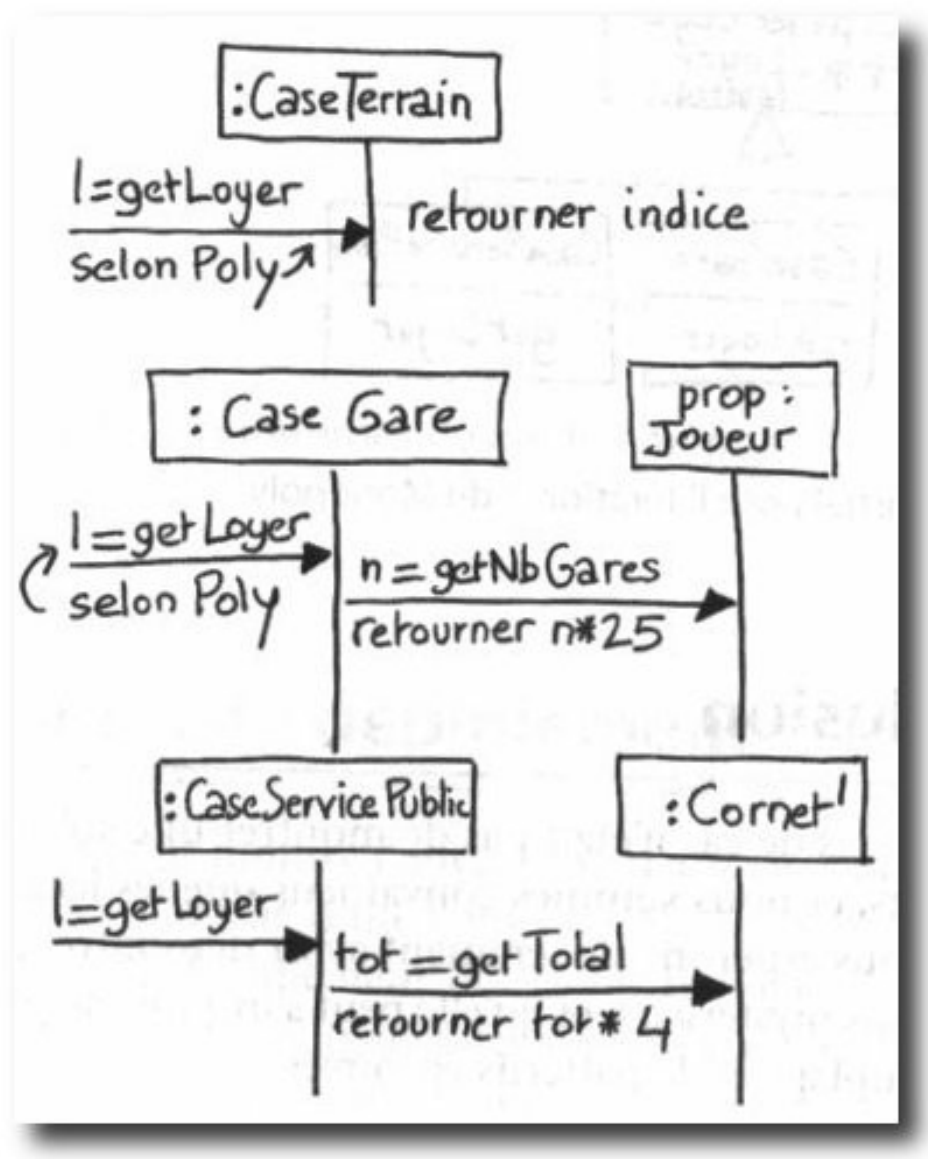
la méthode payerLoyer(Joueur j)

Séquence **payerLoyer(*Joueur* j)**. La **Case** connaît le **Joueur**, car il s'est lui-même passé en argument. La **Case** peut donc lancer sa propre méthode *payerLoyer*(**Joueur** j). Cette méthode créditera le **Joueur** *prop* (connue d'elle) et débitera le **Joueur** j (connu)...



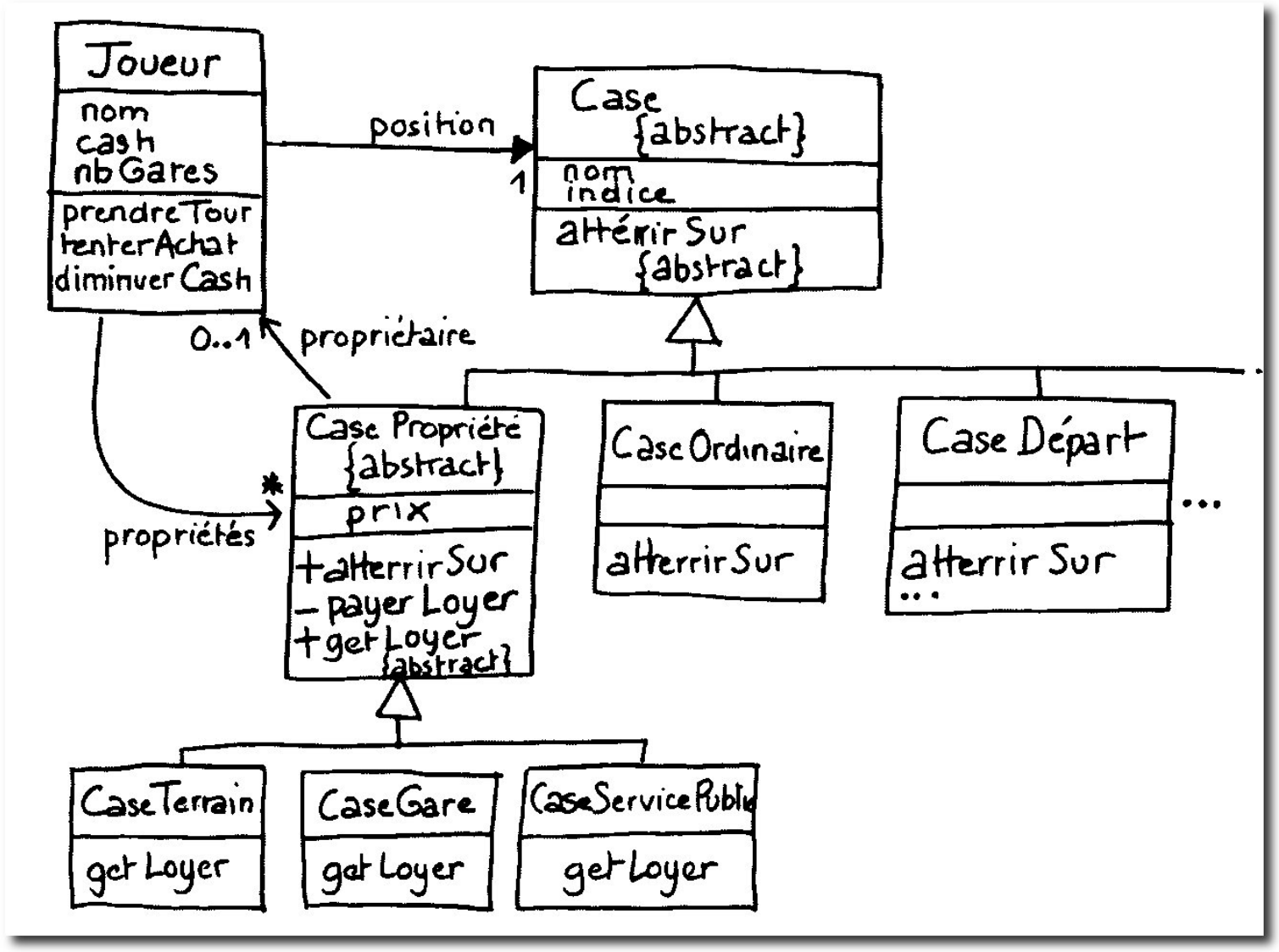
la méthode getLoyer()

Les loyers dépendent en général du nombre de maisons, d'hôtels, ou alors de règles particulières, implémentées grâce au Design Pattern **Polymorphisme**.



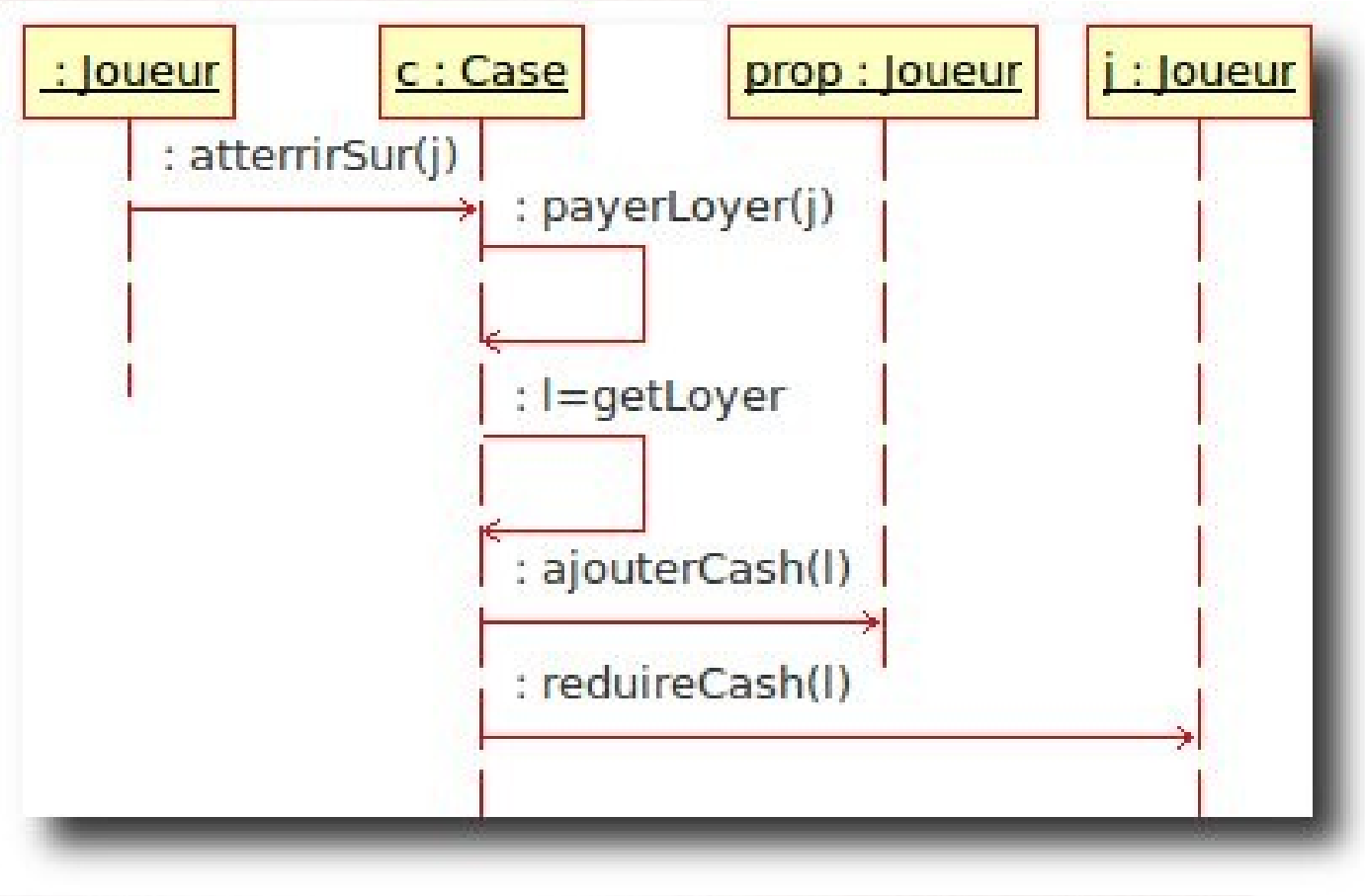
Le Diagramme Classe Conceptuel

Ce qui nous amène à ces modifications statiques....



Une séquence complète..

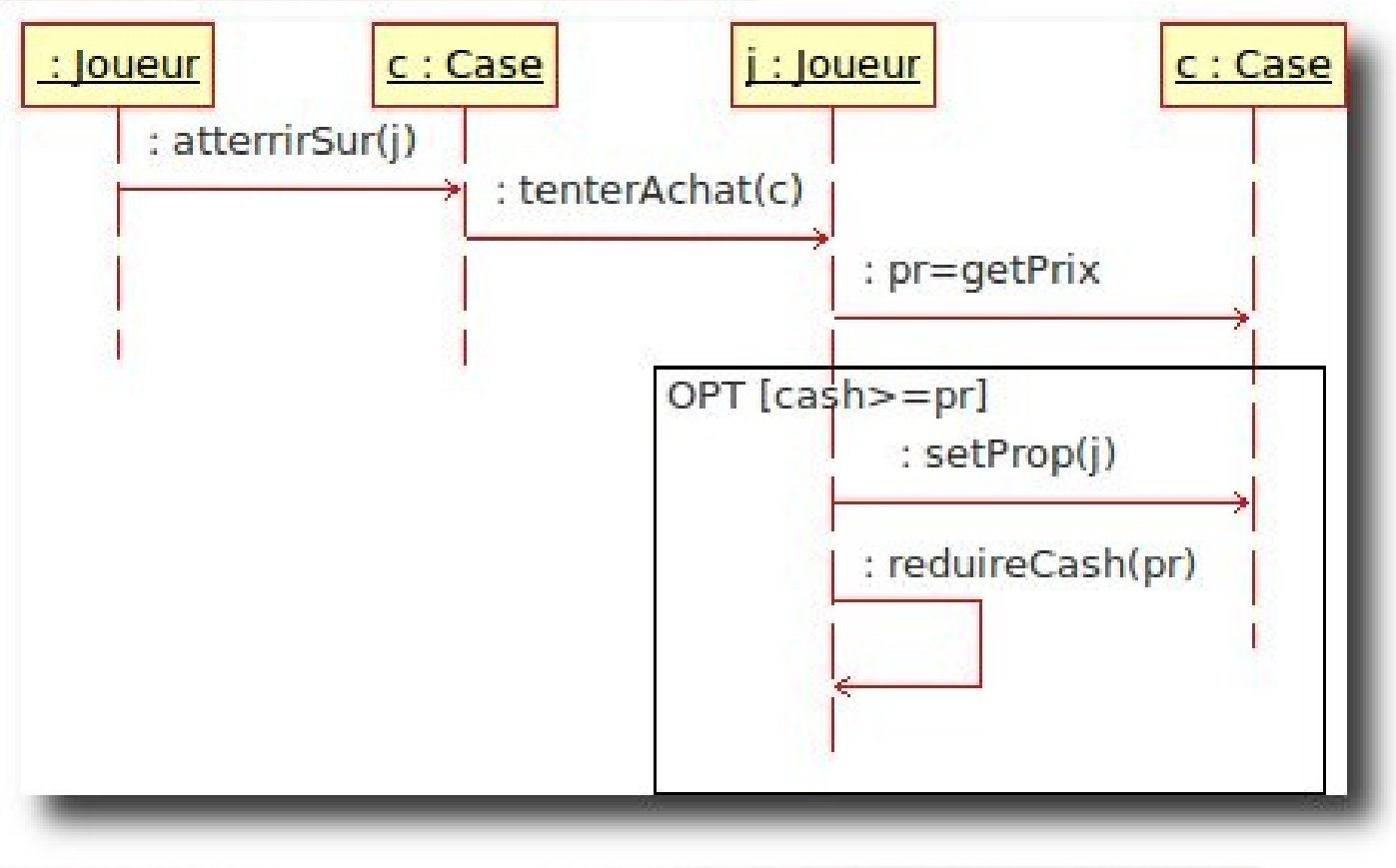
Reprenons la séquence complète suivante : Un *Joueur* tombe sur une *Case* qui appartient à un autre *Joueur*...



Séquence Achat..

Même chose pour la séquence d'achat, complète... On atterrit sur la *Case*, elle est libre, elle demande de *Joueur* de tenter l'achat...

Le *Joueur* demande à la *Case* son prix, vérifie si il peut payer, puis se déclare propriétaire et diminue son cash...



Bilan...

Grâce au diagramme de séquence, nous avons vérifié la cohérence de notre solution. Le *Plateau* connait les *Joueurs*, chaque *Joueur* connaît sa *Case* actuelle (aviez-vous remarqué que le *Pion* avait disparu ? Il n'apportait aucune information, il entrainait un couplage inutile entre classes), chaque *Joueur* connait ses *Propriétés*, et chaque *Propriété* connait son *Propriétaire*. Il s'agit de références, de pointeurs, qui ne consomment donc pas d'espace mémoire. Ils permettent une bonne navigabilité entre Classes.