Table des matières

Bibliographie

Le guide de l'utilisateur UML ; Grady Booch, James Rumbaugh, Ivar Jacobson ; Ed. Eyrolles
UML 2 et les design patterns ; Craig Larman ; Ed. Pearson Education
UML ; Pascal Roques ; Ed. Eyrolles
UML 2 par la pratique; Pascal Roques ; Ed. Eyrolles

Objectifs

Réaliser un exemple simple permettant d'introduire l'utilisation de *pattern* en conception préliminaire.

Activités du TP :

- utiliser des *patterns* GRASP pour la conception préliminaire
- travailler en équipe
- réutiliser une classe validée
- choisir un conteneur pour une multiplicité *
- générer du code à partir d'un outil de génie logiciel
- faire évoluer un logiciel de manière itérative et incrémentale

Le jeu de l'oison (itération 1)

Expression du besoin

Le jeu de l'oison est une version simplifiée du jeu de l'oie avec un parcours ne comportant que 20 cases.

Les cases suivantes ont un comportement particulier : la case 7 permet d'avancer de trois cases, la case 13 fait reculer de trois cases et la case 18 (mort) force à revenir au départ.

Si le déplacement excède la distance à parcourir, le joueur doit rétrograder. Pour gagner, il faut arriver le plus loin possible en 3 tours. Le nombre de joueurs est limité à 2.

Pour tenir compte du temps consacré au développement dans ce TP, cette version ne permettra de faire qu'une partie et on ne pourra saisir le nom du joueur (mais un nom sera tout de même associé au joueur).

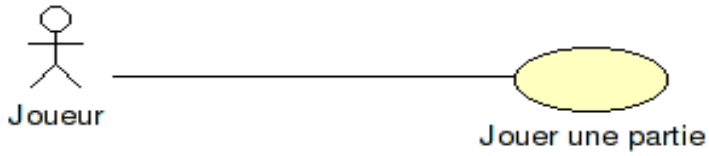
Remarque : dans l'itération 2, on complétera les règles d'où l'importance d'une « bonne » ACOO.

Important : On retrouve ici un jeu de dés. L 'ACOO doit donc permettre la réutilisation ce que l'on va faire en reprenant la classe Dé (validé unitairement si possible !) des TP précédents.

Modélisation des exigences et Analyse

On va formaliser les besoins utilisateurs par un diagramme des cas d'utilisation, une maquette IHM avec son diagramme d'activités et un modèle du domaine.

Diagramme des cas d'utilisations



Cas d'utilisation	*Description*
Jouer une partie	Chaque **joueur** dispose d'un pion. Il lance les 2 dés, les additionne et avance de la quantité indiquée par les dés. Le joueur suivant fait de même. Le vainqueur est celui qui arrive le plus loin en 3 tours de jeu.

Diagramme de classes du domaine

La décomposition du domaine implique l'identification des concepts, des attributs et des associations illustré par un diagramme de classe montrant les concepts et les objets du domaine.

Il faut être clair sur la **terminologie** utilisée (en accord avec l'expertise du client) pour éviter les confusions. Ici par exemple, le terme de **tour** peut être ambiguë. Il a une double signification suivant si on le se place au niveau du joueur (qui lance les dés et déplace un pion) ou au niveau de la partie (tous les joueurs ont joué à leur tour). Pour ce dernier, on peut par exemple utiliser le terme de ***round***.

Voir travail demandé.

Maquette

Ce TP n'a pas pour objectif de développer une IHM conséquente pour cette application. On ne prévoit qu'un simple affichage pour suivre le déroulement de la partie. On se limitera à cette interface :

```
[tv@alias src]$ ./jeuDeLOison
Tour n° 1 :
Joueur toto a obtenu 8 points ->        |........x.....................|
Joueur titi a obtenu 4 points ->        |....x.........................|

Tour n° 2 :
Joueur toto a obtenu 5 points ->        |.............x................|
Joueur titi a obtenu 9 points ->        |.............x................|

Tour n° 3 :
Joueur toto a obtenu 6 points ->        |..................x..........|
Joueur titi a obtenu 8 points ->        |.....................x........|

Fin de partie : le joueur titi a gagné !
```

La conception préliminaire

Problème et Solution : les patterns GRASP

Une fois l'**analyse** terminée, on sait donc **« quoi faire »**. Maintenant en **conception**, on doit déterminer **« comment le faire »**.

Quelques exemples de questions que l'on se pose en conception :

- A quelle classe faut-il confier la responsabilité de certains attributs ?
- Qui est responsable du contrôle de la boucle ? Qui coordonne l'ensemble ?
- Doit-on créer une classe Pion ? une classe Joueur ?
- Qui instancie les objets Dés ? Qui lance les dés ?
- etc ...

On va utiliser les ***patterns*** **GRASP** pour nous aider à réaliser cette phase de **conception préliminaire**.

Les *patterns* GRASP sont des patrons créés par Craig Larman qui décrivent des règles pour affecter les responsabilités aux classes d'un programme orienté objet pendant la conception, en liaison avec la méthode de conception BCE (*Boundary Control Entity*), en français MVC (Modèle Vue Contrôleur). Il y a **9 *patterns* GRASP** :

- **Créateur**
- **Expert**
- **Faible couplage**
- **Contrôleur**
- **Forte cohésion**
- **Fabrication pure**
- **Polymorphisme**
- Protection des variations
- Indirection

Remarque : ces *patterns* ne sont qu'une aide pédagogique qui permet de structurer et de nommer des principes. Une fois que ces principes sont saisis, les termes spécifiques sont sans importance.

Remarque : dans cette étude, on n'abordera pas les *patterns* Indirection et Protection des variations. Une définition ci-dessous des ces deux *patterns*.

Indirection : Affecter des responsabilités à un objet qui sert d'intermédiaire entre d'autres composants ou services pour éviter de les coupler directement.

Protection des variations : Identifier les points de variations ou d'instabilité prévisibles. Affecter les responsabilités pour créer une interface stable autour d'eux.

Pattern Créateur

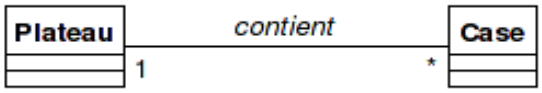
C'est l'un des premiers problèmes à prendre en considération en conception OO. C'est une responsabilité de faire quelque chose.

En fait n'importe quel objet pourrait créer un objet Dé ou un objet Case, mais lequel serait choisi par le plus grand nombre de développeurs OO expérimentés ? Et pourquoi ?
Pourquoi ne pas demander à un objet Chien d'être le créateur ! Non parce que cela ne cadre pas avec le modèle du domaine. De la même manière, on ne nommera pas les classes Joueur et Dé : AB123 et R2D2 !

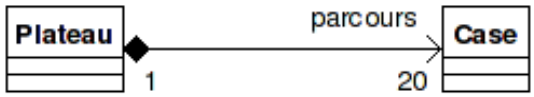
Le pattern Créateur se définit de la manière suivante :

Nom	**Créateur**
Problème	Qui crée A ?
Solution	Affecte à la classe B la responsabilité de créer une instance de la classe A si une ou plusieurs des conditions suivantes est vraie (plus il y en a mieux c'est) : • contient ou agrège des objets A • enregistre des objets A • utilise étroitement des objets A • possède les données d'initialisation des objets A

Problème : **qui crée l'objet Case ?**
A partir du modèle du domaine, on constate qu'un Plateau contient des Cases.



En cohérence avec le principe du *pattern* Créateur, le Plateau créera les Cases. De plus, les Cases feront toujours partie d'un Plateau et celui-ci gérera leur création et leur destruction : les Cases sont donc dans une relation de **composition** avec le Plateau.



Problème : **comment traduire les associations navigables de multiplicté « * » ?**
La difficulté consiste à choisir le bon **conteneur** parmi les très nombreuses classes de base fournis avec les langages et leur environnement. Ce choix se fait en **conception détaillée** car le langage de développement doit être connu. En utilisant le document en annexe pour le choix du conteneur, la meilleure solution semble être le conteneur **Vector** de la STL (*Standard Template Library*).

Pattern Expert en Information

L'objectif de ce *pattern* est d'affecter au mieux une responsabilité à une classe logicielle. Afin de réaliser ces responsabilités, cette classe doit disposer des informations nécessaires.

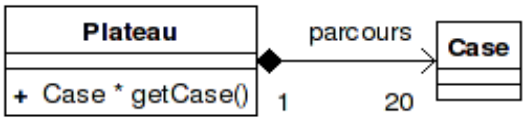
Nom	**Expert en Information** ou plus simplement **Expert**
Problème	Quel est le principe général d'affectation des responsabilités aux objets ?
Solution	Affecter la responsabilité à la classe qui possède les informations nécessaire pour le faire

Si les responsabilités sont mal réparties, les classes logicielles vont être difficilement maintenables, plus dure à comprendre, et avec une réutilisation des composants peu flexible.

Ainsi, comme d'autres *pattern* GRASP, le modèle Expert est un *pattern* relativement intuitif, qui en général, est respecté par le bon sens du concepteur. Il s'agit tout simplement d'affecter à une classe les responsabilités correspondants aux informations dont elle dispose intrinsèquement (qu'elle possède) ou non (objets de collaborations).

Problème : **qui connaît un objet Case à partir d'une clé ?**

Pour s'acquitter de cette responsabilité, il faut pouvoir récupérer une Case quelconque (à partir de son numéro ou de sa position) parmi toutes les Cases. En conséquence, l'objet **Plateau** (qui possède une relation de composition) dispose des informations nécessaires pour faire cela.



Pattern Faible Couplage

Le couplage est une mesure du degré auquel un élément est lié à un autre, en a connaissance ou en dépend. S'il y a couplage ou dépendance, l'objet dépendant peut être affecté par les modifications de celui dont il dépend. Un objet A qui fait appel aux opérations d'un objet B a un couplage aux service de B. Le faible couplage a pour objectif de faciliter la maintenance en minimisant les dépendances entre éléments.

Nom	**Faible Couplage**
Problème	Comment réduire l'impact des modifications ?
Solution	Affecter les responsabilités de sorte à éviter tout couplage inutile

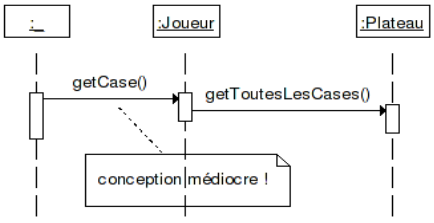
Bien entendu, il ne faut pas tomber dans le piège de décider de concevoir des éléments tous indépendants et faiblement couplés, car ceci irai à l'encontre du principe OO définissant un système objet comme un ensemble d'objets connectés les uns aux autres et communiquant entre eux.

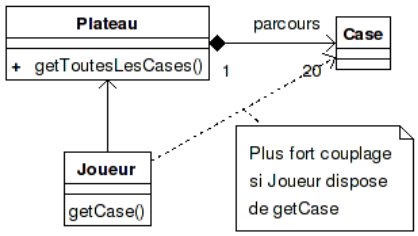
En général ce *pattern* est appliqué inconsciemment, c'est davantage un principe d'appréciation qu'une règle.

Problème : **pourquoi un Plateau plutôt qu'un Chien ?**

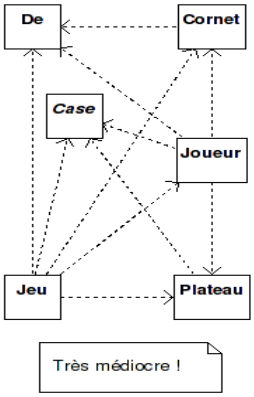
On a décidé d'affecter la responsabilité de connaître une Case particulière à l'objet Plateau parce que le Plateau sait tout sur les Cases (c'est lui l'Expert). Mais pourquoi Expert donne-t-il ce conseil ? Car dans ce cas, seul le Plateau sera couplé à Case.

Une autre (mauvaise) solution augmenterait le couplage :





Et alors on pourrait arriver à une solution très médiocre :



Pattern Contrôleur

Ce *pattern* permet d'affecter la responsabilité à une classe « façade » représentant l'ensemble d'un système ou un scénario de cas d'utilisation. Comment représenter le programme principal, le système lui-même ?

Il faut gérer l'ensemble des actions, coordonner les différents éléments. Il faut une classe Contrôleur, qui déléguera ensuite aux divers objets spécialisés, et récupérera les résultats de leurs actions.

Il ne s'agit pas de l'interface homme machine. Le *pattern* Contrôleur consiste en l'affectation de la responsabilité de la réception et/ou du traitement d'un message système à une classe.

Il reçoit les demandes, et le redirige vers la bonne classe, celle qui en a la responsabilité.

Nom	**Contrôleur**
Problème	Quel est le premier objet (au delà de la couche Présentation) qui reçoit et coordonne une opération système ?
Solution	Affecter la responsabilité à un objet représentant l'un des ces choix : • l'objet représente le système global, un objet racine, un équipement sur lequel s'exécute le logiciel ou un sous-système majeur (contrôleur de façade, c'est à dire l'interface d'accès à l'ensemble d'un système) • l'objet représente un scénario de cas d'utilisation dans lequel l'opération système a lieu (contrôleur de session, qui est chargé de traiter tous les événements systèmes contenus dans un scénario de cas d'utilisation)

Problème : **qui est le Contrôleur de l'opération système lancerPartie ?**

Le choix d'un objet racine tel que JeuDeLOison comme contrôleur de façade est satisfaisant :

- si les opérations système sont peu nombreuses (il n'y en a que deux pour ce cas d'utilisation)
- si le contrôleur de façade n'assume pas trop de responsabilité (voir Forte Cohésion)

Pattern Forte Cohésion

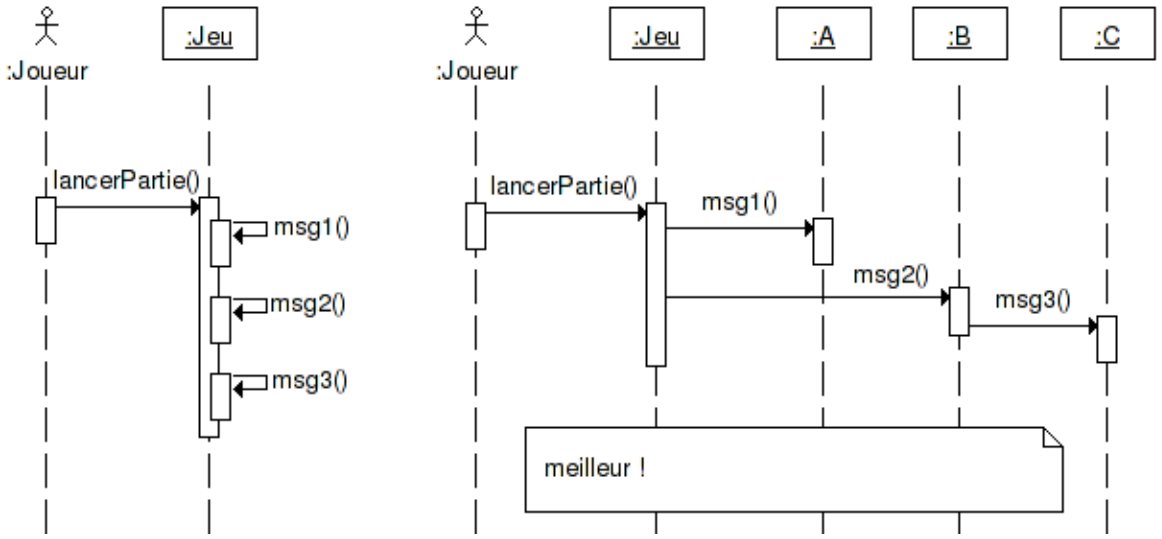
La cohésion mesure le degré de spécialisation des responsabilités d'un composant ou classe. Comme dans le *pattern* Faible couplage, la cohésion médiocre altère la compréhension, la réutilisation, la maintenabilité et subit toute sorte de changements.

Nom	**Forte Cohésion**
Problème	Comment s'assurer que les objets restent compréhensibles et faciles à gérer (et qu'il contribue au Faible Couplage) ?
Solution	Affecter les responsabilités pour que la cohésion demeure élevée.

C'est souvent l'erreur commise par les développeurs OO débutants : confier tout le travail à une seule classe et on obtient un objet trop « gonflé » !

Par exemple on ne peut exiger d'un réfrigérateur qu'il fasse radiateur et range-disques en même temps (en tout cas dans sa modélisation logicielle). La multiplication des disciplines à responsabilité accroît exponentiellement le risque d'erreurs intrinsèques à cette classe.

Problème : **l'opération système lancerPartie est appelée, et après ?**



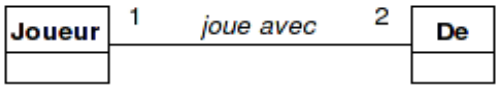
Pattern Fabrication Pure

Ce *pattern* permet d'affecter un ensemble de responsabilités à une classe artificielle ou de commodité qui ne représente pas un concept du modèle de domaine. La fabrication pure est une réalisation qui ne se déduit pas du domaine. Elle est totalement artificielle. Ce *pattern* est utilisé lorsque les *patterns* ne parviennent pas à assurer les principes de forte cohésion et de faible couplage.

Le concepteur doit alors inventer de nouveaux concepts abstraits sans relation directe avec le domaine.

Problème : **comment gérer les dés ?**

Selon le modèle du domaine, le **Joueur joue avec des Dés**.



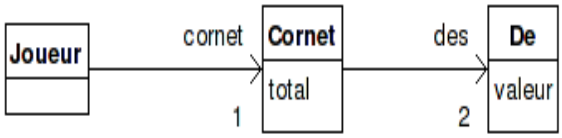
On a déjà vu que les dés sont des objets génériques utilisables dans toutes sortes de jeu. L'affectation de la responsabilité au Joueur de les lancer et de les additionner empêche de réutiliser ce service dans d'autres jeux. Il y a un autre problème : il n'est pas possible de demander simplement le total actuel des dés sans devoir les lancer de nouveau.

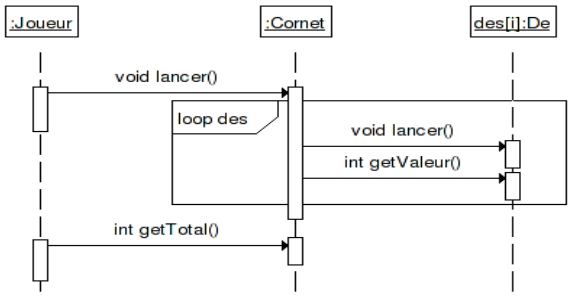
Qui choisir ?

Le choix de toute autre objet du domaine pose le même problème ! Cela nous amène à une **Fabrication Pure** : un objet inventé qui fournira le service dont on a besoin.

Quel nom ?

Il faut tout de même essayer d'employer un vocabulaire en rapport avec le domaine. De nombreux jeux de plateau proposent d'utiliser un cornet pour secouer les dés et les lancer sur la table. On propose donc de créer une classe **Cornet** qui contiendra les dés, les lancera et connaîtra leur total.





Pattern Polymorphisme

Il faut utiliser le polymorphisme pour implémenter les variations de comportement en fonction de la classe. Le polymorphisme est le concept idéal pour réaliser une variation du comportement des objets en fonction de leur type. Il est utile dans le cadre de prévisions de remplacement ou d'ajout de composants.
Les programmes utilisant des (nombreuses) instructions *if/elseif* ou *switch/case* nécessitent une intervention au niveau du client lorsqu'on souhaite ajouter un cas. Cela rend le programme plus difficile à maintenir et le module difficile à réutiliser.

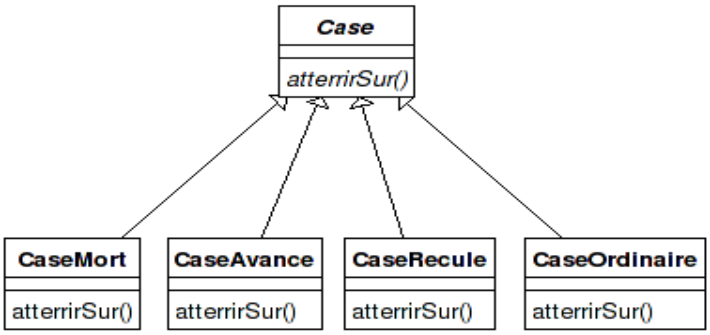
Problème : **comment concevoir les actions en fonction des types de cases ?**
Pour résumer, lorsqu'un joueur atterrit sur une case celle-ci influe sur le comportement du déplacement du pion : il existe une case qui fait revenir au départ, une autre qui fait faire un saut de trois cases, etc ... Il existe donc une action selon le type de cases.

Une mauvaise conception (trop souvent réalisée) aboutirait à ce genre de code :

```
#define CASE_MORT 1
//etc ...

switch(case.type)
{
case CASE_MORT :      revenirAuDepart(); break;
case CASE_AVANCE :    avancer(3); break;
case CASE_RECULE :    reculer(3); break;
// etc ...
case CASE_NORMALE :           break; //ne rien faire !
}
```

En appliquant Polymorphisme, on va créer une classe pour chaque sorte de Case dont la responsabilité atterrirSur diffère. On implémentera donc une méthode atterrirSur() dans chacune d'elles.



Bonne pratique : dans ce genre de situation, la super-classe Case se doit d'être **abstraite** (on ne peut instancier ce type de classe). En C++, on rend une classe abstraite de deux manières possibles :
- il suffit que la classe possède au moins une méthode virtuelle pure (*abstract* en UML)
- en rendant impossible l'accès au constructeur (en le plaçant en *protected* ou en *private*)

Dans un diagramme de classe UML, le nom de la classe abstraite apparaît en ***italique*** ainsi que les méthodes virtuelles pures.

Le cas d'utilisation Démarrer

Les systèmes possèdent explicitement ou implicitement un cas d'utilisation que l'on nomme souvent Démarrer qui correspond à l'**initialisation de l'application**.

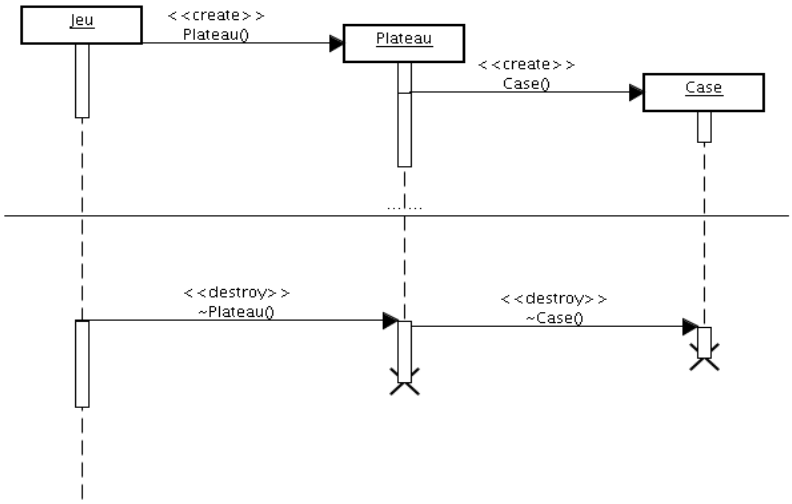
Remarque : **concevoir l'initialisation en dernier**
Lors de l'implémentation, il faudra coder en premier au moins un cas d'utilisation Démarrer (l'opération système d'initialisation est la première à s'exécuter au lancement de l'application). Mais, pendant la modélisation, il faut le considérer en dernier après avoir découvert ce qui doit être créé et initialisé.

Comment les applications démarrent-elles ?
L'opération système démarrer ou initialiser représente la phase initiale de lancement d'une application. Le lancement d'une application dépend du langage de programmation et du système d'exploitation.

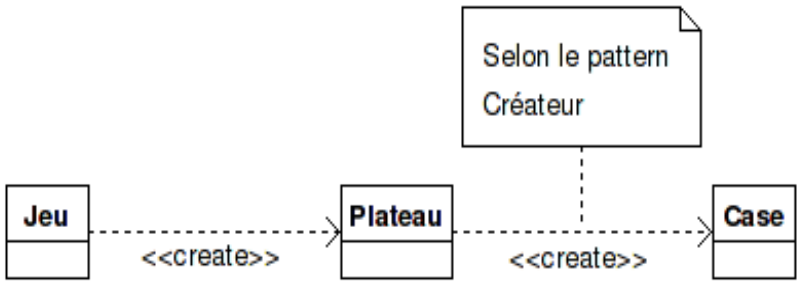
La règle : elle consiste à créer un objet du domaine initial qui sera le premier objet logiciel du « domaine » à être créé. Cette création peut avoir lieu dans la méthode **main()**.

Le choix : on choisi un objet racine approprié qui sera le créateur de certains autres objets.

La représentation :
On peut représenter les détails de cette conception **dynamique** au moyen de diagramme d'interaction (séquence et/ou collaboration). Par exemple :



Une autre approche consiste à utiliser un diagramme (statique) de classes avec des dépendances utilisant le stéréotype « create ». Cette solution donne une bonne vue de l'ensemble des créations.
Par exemple :



Le codage :
Cela dépend du langage de programmation et du système d'exploitation. Par exemple en C++ sous Linux :

```
#include "JeuDeLOison.h"

int main( int argc, char* argv[] )
{
  JeuDeLOison jeuDeLOison;

  jeuDeLOison.lancerPartie();

  return 0;
}
```

Dans le **constructeur** de la classe JeuDeLOison :

```
JeuDeLOison::JeuDeLOison():nombreDeJoueurs(2)
{
    nombreDeTours = 3;
    initialiser();

}
```

Et la méthode initialiser() :

```
void JeuDeLOison::initialiser()
{
    plateau = new Plateau;
    plateau->construire();

    joueurs[0] = new Joueur("toto", *plateau);
    joueurs[1] = new Joueur("titi", *plateau);
}
```

Travail demandé

Étape n°1 : Modélisation des exigences et Analyse (durée : 10 mn)

Contrainte : travail en équipe de 3 ou de 4

1 . Proposer un diagramme de classe d'analyse pour le modèle du domaine (durée : 10 mn).

Étape n°2 : La conception (durée : 50 mn)

Contrainte : travail en équipe de 3 ou de 4

2 . Élaborer un modèle de conception partiel en tenant compte de l'étude préalable (durée: 10mn)

3 . Appliquer le *pattern* Expert pour compléter le modèle de conception (durée : 10 mn).

3 . a . Qui est responsable du contrôle de la boucle ?
La première responsabilité est le contrôle de la boucle : exécuter n *rounds* et jouer un tour pour chaque joueur. Il faut appliquer le *pattern* **Expert**. Pour cela, compléter le tableau suivant :

Quelles sont les informations nécessaires ?	*Qui détient les informations ?*
le nombre actuel de *rounds* (pour connaître le point d'arrêt de la boucle)	
tous les joueurs (pour les faire jouer chacun leur tour)	

Choix de l'objet : _____

3 . b . Quel est l'objet qui doit être responsable de prendre un tour d'un joueur ?
Prendre un tour consiste à lancer les dés et à déplacer un pion sur la case indiquée par le total des valeurs de leurs faces. Il faut appliquer le *pattern* **Expert**. Pour cela, compléter le tableau suivant :

Quelles sont les informations nécessaires ?	*Qui détient les informations ?*
l'emplacement actuel du Joueur (pour connaître le point de départ du déplacement)	
les deux objets Dés (pour les lancer et calculer leur total)	
toutes les cases et leur organisation (pour pouvoir se déplacer sur la bonne case)	

Remarques : parfois le choix est difficile ...

- Lorsqu'il faut choisir entre plusieurs experts, il faut affecter la responsabilité à l'expert dominant (celui qui détient la majorité des informations).

- Lorsqu'il y a plusieurs choix de conception possibles, il faut tenir compte du couplage et de la cohésion.
- Si aucun choix n'est encore fait, il faut alors tenir compte de l'évolution (future) des objets logiciels notamment en tenant compte des itérations planifiées.

Choix de l'objet : _____

4 . Proposer un diagramme de classes de conception (durée : 10 mn).

5 . Concevoir l'initialisation de l'application. Proposer un diagramme de classes avec des dépendances utilisant le stéréotype « create » (durée : 5 mn).

6 . Proposer un diagramme de séquence pour l'opération système lancerPartie (durée : 15 mn).

Étape n°3 : La réalisation (durée : 2h30)

Contraintes :

- outil de génie logiciel : **bouml**
- environnement de développement : C++ avec le compilateur gcc/**g++** sous Linux
- si en équipe de 3 ou de 4 : **subversion** pour la gestion de la configuration

Tous les documents nécessaires à la réalisation de ce TP sont sur le serveur de la section.

Remarque : vous disposez d'un tutoriel pour la génération de code avec **bouml** (http://bouml.free.fr/documentation.html)

7 . Fabriquer un système partiel exécutable résultat de cette itération.

Le jeu de l'oison : itération 2 (durée : 30 mn)

On complète les règles. Le parcours comporte finalement **30 cases**. Les cases suivantes auront un comportement particulier :

- les cases 7 et **14** permettent d'avancer de trois cases.
- les cases 13 et **26** font reculer de trois cases.
- la **case 20 renvoie à sa cas d'origine**.
- la case **28** (mort) force à revenir au départ.

Pour gagner, il faut arriver le plus loin possible en **4 tours**.

Rappels :
La méthode de développement appliquée est un **développement itératif et incrémental** :

- **itératif** → une itération est un cycle de développement complet
- **incrémental** → chaque développement s'ajoute et enrichit l'existant

Le résultat de chaque itération est un **système partiel exécutable**.

Contrainte : travail individuel

8 . Certaines de ces modifications n'ont une influence que sur le code. Lesquelles ? Indiquer à quel endroit du code auront lieu les modifications.

9 . Quelle(s) modification(s) a une influence sur la conception ? Indiquer la modification(s) à apporter au modèle de conception.

10 . Fabriquer le résultat de cette itération.

Annexe : choix d'un conteneur