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 – 5°édition ; Pascal Roques ; Ed. Eyrolles

Objectifs

Réaliser une application en suivant les étapes d'un processus de développement présentant une vue d'ensemble d'UML et de la modélisation graphique.

Activités du TP :

- utiliser un outil de génie logiciel et un environnement de développement graphique
- respecter une démarche de modélisation
- vérifier la cohérence des artefacts produits
- mettre en place une architecture à couches
- mettre en oeuvre un pattern GoF dans la conception détaillée
- coder un diagramme de séquence
- réaliser les tests de validation de l'application
- déployer une application

Travaux pratiques : le jeu du démineur

Expression du besoin

Avant d'entrer dans les détails du développement itératif, de l'analyse des besoins et de l'A/COO, on va décrire l'objectif de cette étude: **concevoir un jeu de démineur** (comme celui qui est livré avec le système d'exploitation *Microsoft Windows©*). Le but du jeu est de trouver le plus rapidement possible toutes les cases du plateau contenant des mines sans les toucher.

Mise en situation et contraintes

Dans cette manipulation on utilisera :

- le logiciel **bouml** comme outil pour produire les documents UML 2
- le langage **C++** comme langage objet pour produire le code de l'application
- l'environnement de développement *Builder C++* de *Borland©* sous *Microsoft Windows©*

Présentation du jeu

Le jeu est composé d'un plateau rectangulaire, d'un chronomètre et d'un compteur de mines. Les plateau est un quadrillage de cases. Au début du jeu, toutes les cases du plateau sont couvertes, le compteur de mines indiquant le nombre de mines restant à localiser. Le chronomètre compte le nombre de secondes écoulées depuis le début de la partie. La partie commence lorsque la première case est découverte.

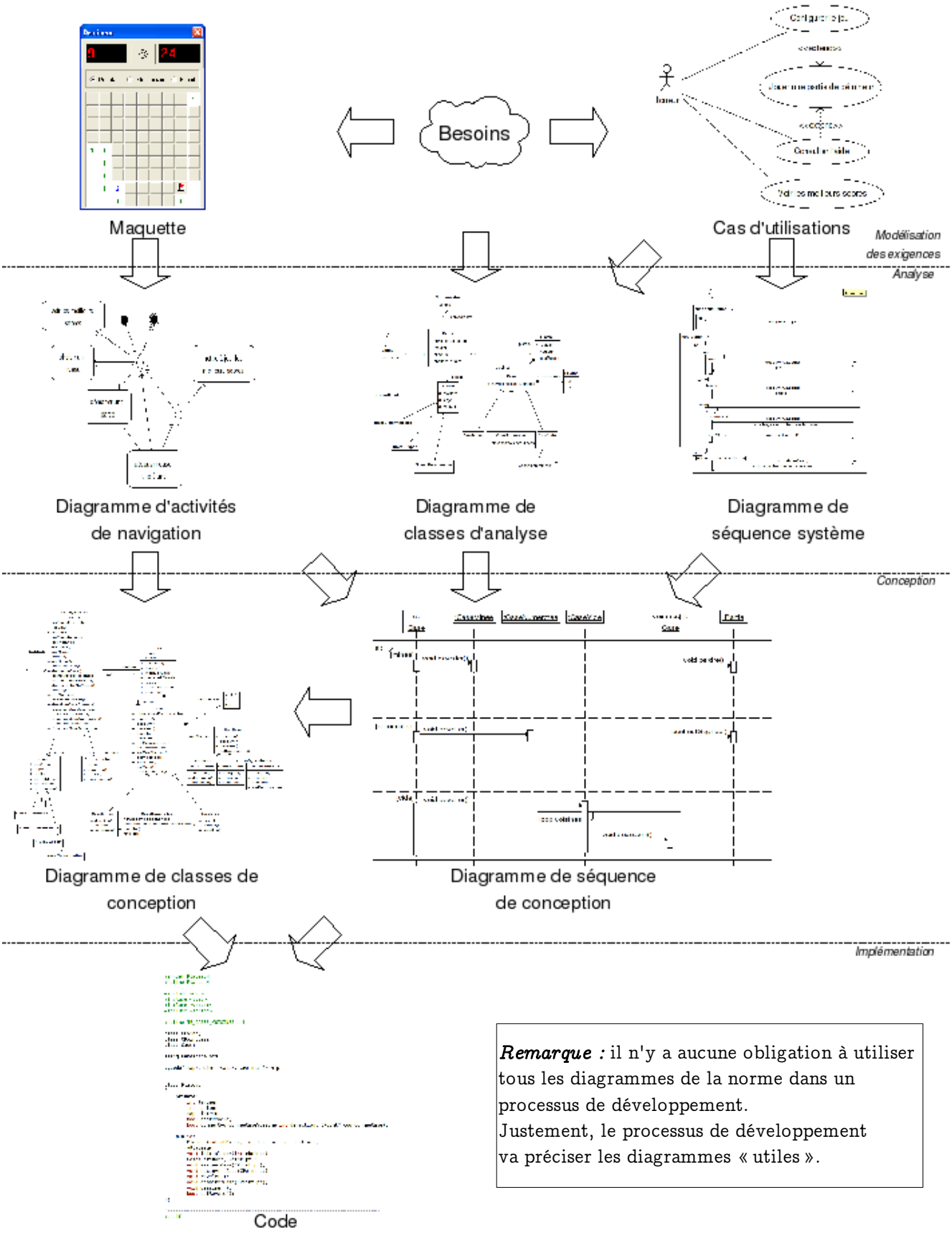
Quand une case est découverte, son contenu est affiché. Le contenu d'une case peut être rien, une mine ou un nombre indiquant le nombre de mines présentes dans les cases voisines. Les scénarios suivants peuvent se produire lorsqu'une case est découverte, en fonction de son contenu :

1. Un chiffre – Il ne se passe rien
2. Un blanc – Toutes les cases voisines sont dévoilées, à condition qu'elles ne soient pas signalées par un drapeau. Si l'une de ces cases voisines ne contient rien, le processus de découverte continue automatiquement à partir de cette case.
3. Une mine – Le jeu est terminé et le joueur a perdu.

Si elle est toujours couverte, une case peut être marquée en respectant les règles suivantes:

- Marquer une case qui n'est ni découverte ni marquée décrémente le compteur de mines restant à localiser et un drapeau apparaît sur la case. Il indique que cette case contient potentiellement une mine. Une case marquée d'un drapeau ne peut être découverte.
- Marquer une case déjà signalée d'un drapeau permet de la remettre dans son état initial, à savoir couverte et non marquée. Le compteur de mines est alors incrémenté de 1.

Démarche de modélisation



Remarque : il n'y a aucune obligation à utiliser tous les diagrammes de la norme dans un processus de développement.
Justement, le processus de développement va préciser les diagrammes « utiles ».

I . Modélisation des exigences

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.

Étape n°I.1 : définir les cas d'utilisation

L'analyse des besoins peut inclure des « histoires » ou des scénarios de la façon dont l'application sera utilisée. Pour cela, on rédige des **cas d'utilisations**. Un cas d'utilisation (*Use Case*) décrit un ensemble de séquence d'actions réalisées par le système et produisant un résultat observable intéressant pour un acteur.

Remarque : les cas d'utilisation ne sont pas « Orientés Objet », ce ne sont que des « histoires écrites » qui constituent un outil d'analyse des besoins.

L'acteur principal unique est le joueur. Son cas d'utilisation favori consiste à « Jouer une partie de démineur ». Néanmoins, il a la possibilité de configurer les dimensions du plateau, le nombre initial de mines, etc... On peut aussi ajouter les cas d'utilisations classiques pour ce type d'application : « Consulter l'aide » en ligne et « Voir les meilleurs scores ».

Remarque : ce n'est pas le bon moment !
De la même manière, au moment de rédiger les cas d'utilisations, on ne sait pas encore comment se fera l'enregistrement persistant des meilleurs scores : fichier, base de donnée, etc... Étant donné qu'il n'y a pas eu de contraintes formulées, ce choix se décidera au moment de la conception (« comment faire ? »).

1 . Rédiger un diagramme de cas d'utilisation pour le jeu de démineur.

Étape n°I.2 : définir une planification itérative

Le processus unifié encourage **une planification itérative pilotée à la fois par les risques et par le client**. Cela signifie que les objectifs des premières itérations sont choisis afin de :

- **identifier les risques les plus importants**
- **construire les fonctionnalités visibles qui comptent le plus pour le client**

Exemple : pour l'itération 1, on choisira **x% des cas d'utilisation** présentant ces trois qualités

- **significatifs du point de vue de l'architecture** (on sera donc forcer de concevoir, implémenter et tester l'architecture noyau)
- **de grande valeur pour le client** (les fonctionnalités qui comptent vraiment pour lui)
- **à haut risque** (par exemple « gérer 500 requêtes concurrentes »)

On vous demande de hiérarchiser les cas d'utilisations identifiés en tenant compte des facteurs suivants et on distinguera trois niveaux (**H**aut, **M**oyen et **B**as) :

- la **priorité fonctionnelle** (à déterminer avec le client)
- le **risque technique** (estimer par le chef de projet)

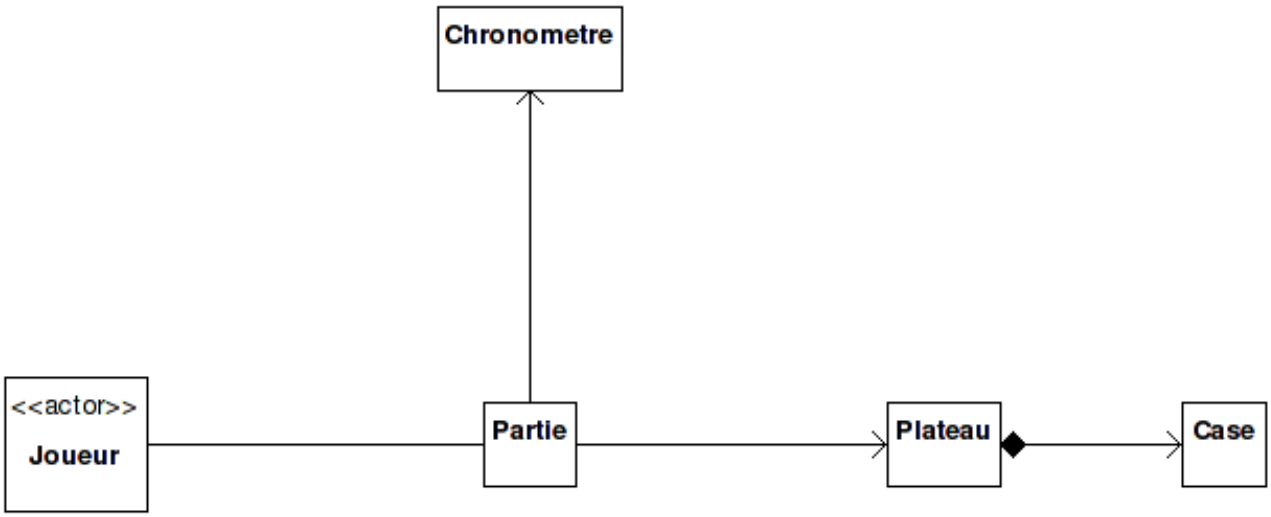
2 . Proposer une une planification itérative qui permet de construire les fonctionnalités visibles qui comptent le plus pour le client.

Cas d'utilisation	*Priorité*	*Risque*	*Itération #*

Par la suite, on réalisera une planification des tâches à réaliser dans chaque itération.

Étape n°I.3 : définir un modèle du domaine

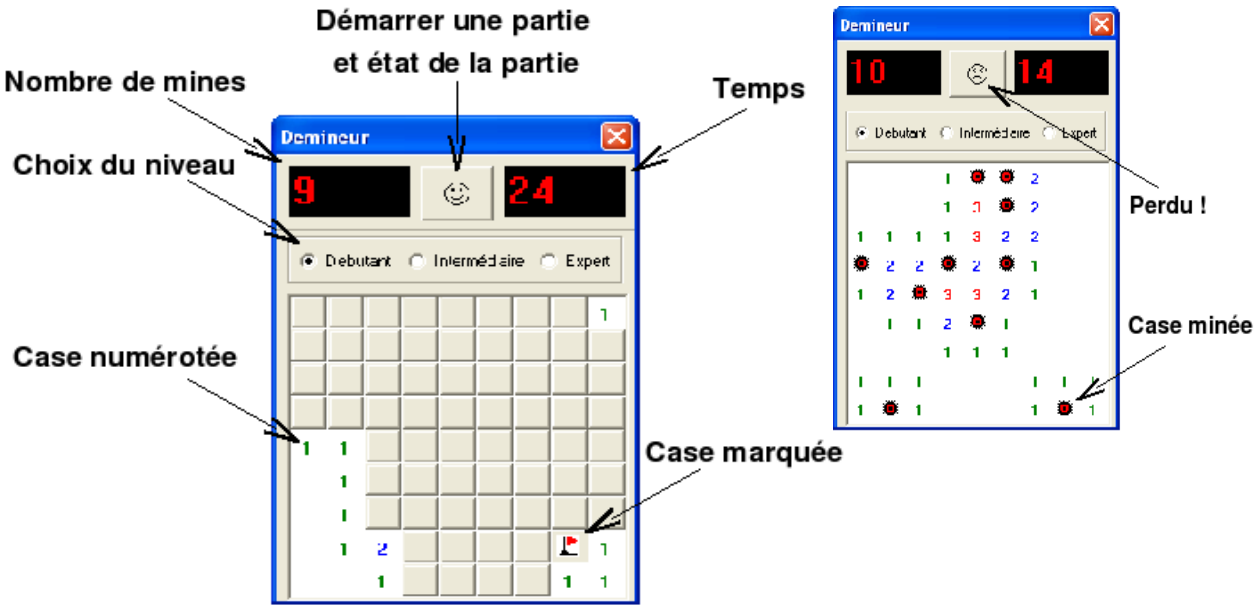
L'analyse orientée objet est centrée sur la création d'une description du domaine dans la perspective d'une classification des objets. La décomposition du domaine implique l'identification des concepts, des attributs et des associations (une association est une relation entre éléments UML), illustré par un ensemble de diagrammes montrant les concepts et les objets du domaine. On utilisera par exemple un **diagramme de classe** pour représenter le modèle du domaine.



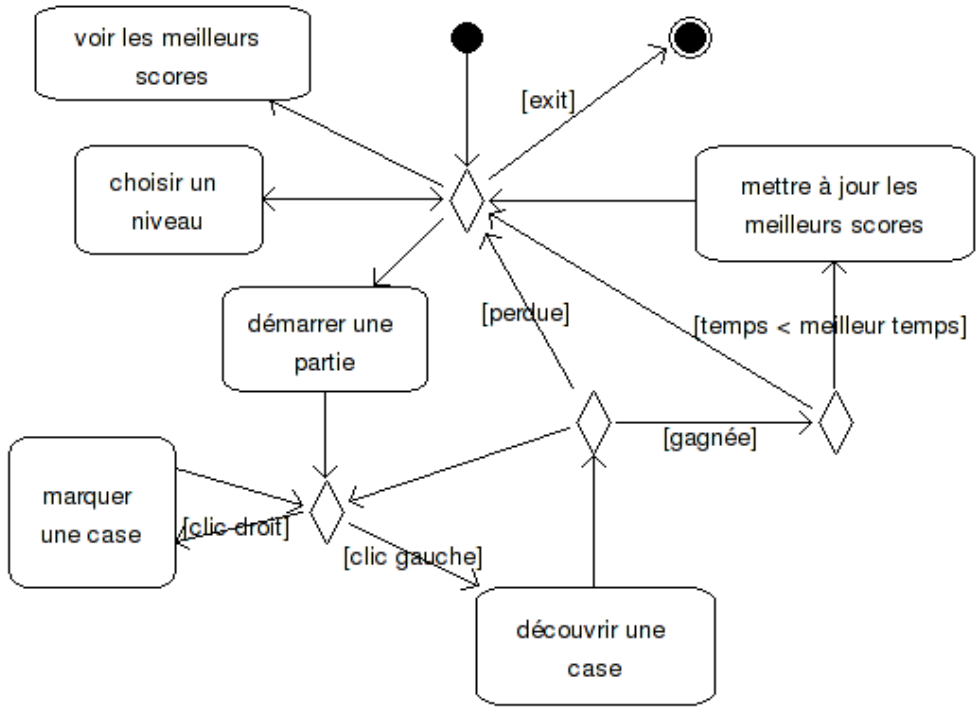
3 . Compléter ce diagramme de classe du domaine en indiquant le nom des associations, leurs multiplicités et les attributs.

Étape n°I.4 : élaborer une maquette et les diagrammes d'activités

La maquette s'inspire du **jeu de démineur** qui est livré avec le système d'exploitation *Microsoft Windows©* :



Le diagramme d'activités permet de faire apparaître les deux activités principales du cas d'utilisation Joueur une partie de démineur : **découvrir une case** et **marquer une case**.



II . Analyse

Étape n°II.1 : élaborer les diagrammes de séquence « système »

Il est possible de décrire le comportement du système par des **diagrammes de séquence système** où le système est vu comme une **« boîte noire »**. Le système est donc vu de l'extérieur (les acteurs) sans préjuger de comment il sera réalisé. La « boîte noire » sera ouverte (décrite) seulement en conception.

Un cas particulier : **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**.

Important : **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. 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.

Cas d'utilisation principal : **Jouer une partie de démineur**
Le joueur passe son temps à découvrir ou (***alt***) marquer des cases. Dans la boucle (***loop***) de jeu, on représente les trois possibilités (***alt***) : perte, gain ou cas normal. Les cas de perte ou de gain arrêtent la partie (***break***). En cas de gain, il pourra (***opt***) entrer son nom dans les meilleurs scores si (***[]***) il a le meilleur temps en fonction du niveau choisi.

On peut ajouter la configuration du jeu comme étape optionnelle (***opt***) avant de commencer à jouer.

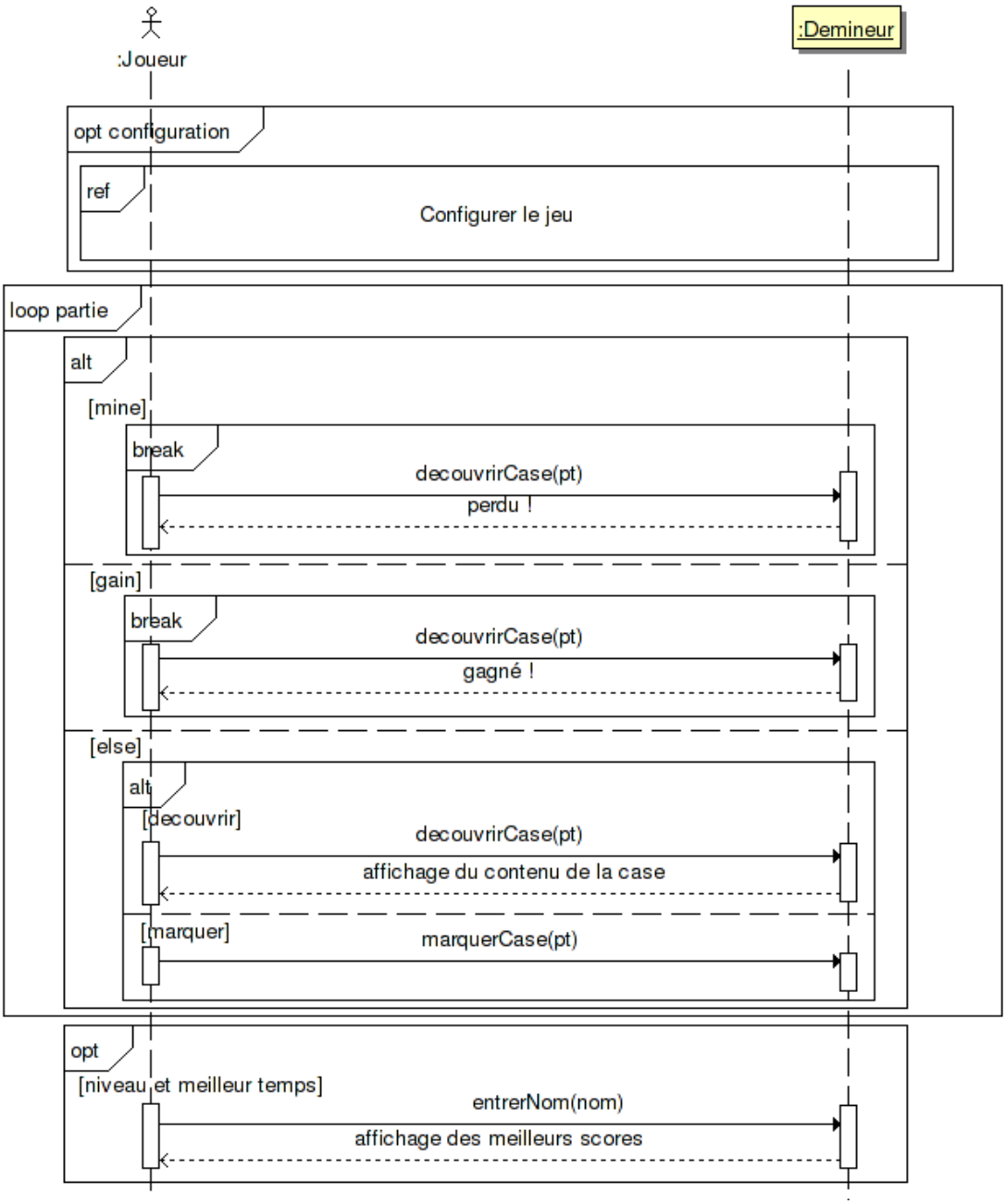
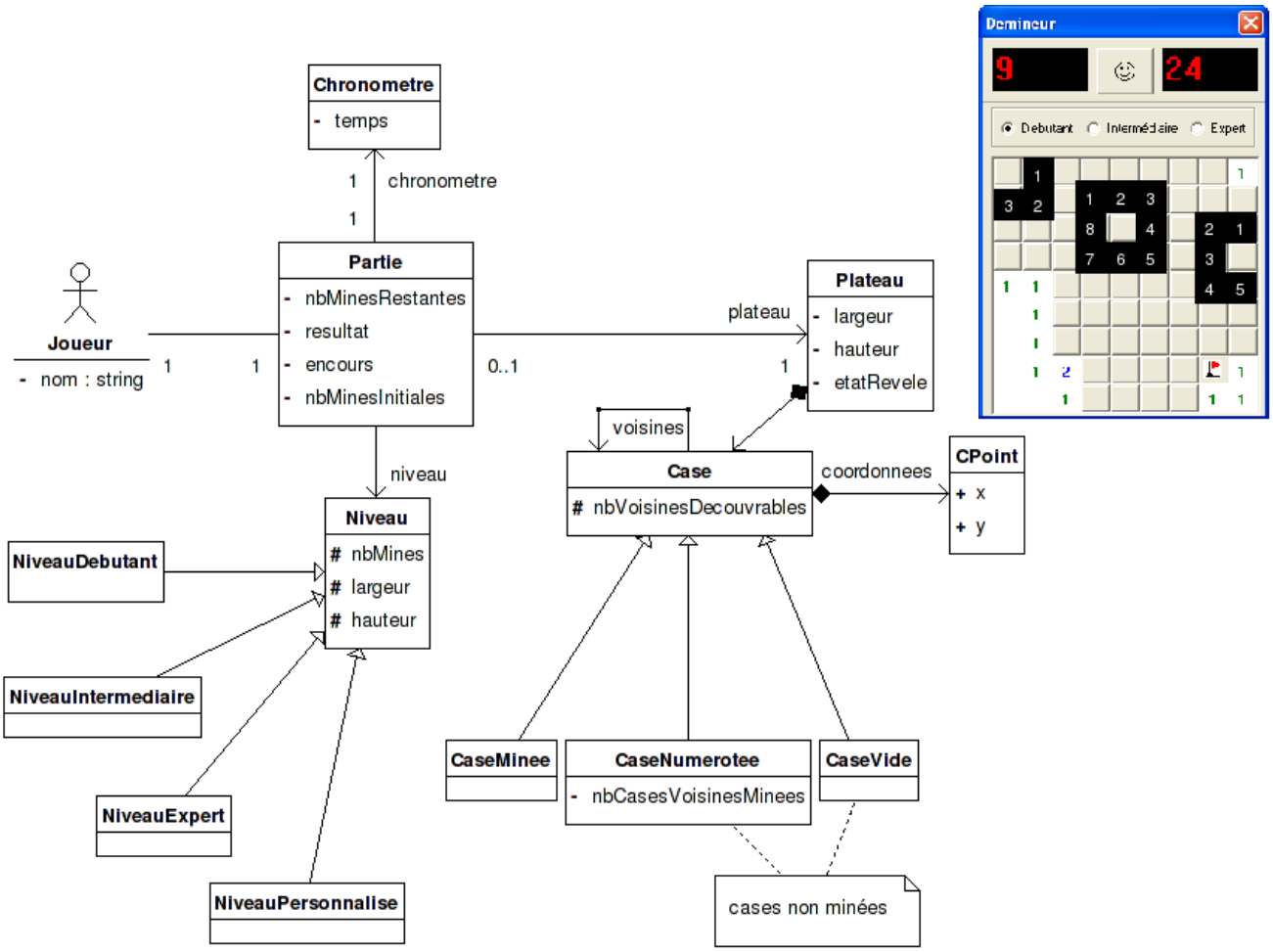


Diagramme de séquence « système » de Jouer une partie de démineur

Étape n°II.2 : élaborer un diagramme de classes d'analyse et un diagramme d'état si nécessaire

Une analyse plus profonde du domaine nous permet de spécialiser les types de **Case**. Le **Plateau** est défini pas une hauteur et une largeur (suivant le **Niveau** sélectionné par le joueur) et les cases se doivent d'être repérables par un système de coordonnées (x,y). Une case possède 3, 5 ou 8 voisines.



Dans cette application, la difficulté consiste à « bien » modéliser la problématique des cases. En effet :

- qu'une case soit **minée** (ou pas) est propre à la case et reste vrai du début à la fin de sa vie (une partie)
- qu'une case soit **marquée** (ou pas) varie durant sa vie : il s'agit donc d'une notion d'**état**.

Il faut donc compléter l'analyse par un **diagramme d'état** de la classe Case.

4 . Élaborer un diagramme d'état de la classe Case.

Étape n°II.3 : rédiger le plan de recette (test de validation)

Si les tests se déroulent parmi les activités finales du projet (mais pas de la programmation !), ils se préparent dès le début. Il existe différentes formes de tests : tests de fonctionnements (unitaires et intégrations), **tests de validation** et tests d'architecture.

Les tests de validations permettent de vérifier que le produit réalisé est le bon.

Ces tests, quels qu'ils soient, sont **planifiés à l'avance**, dans l'étape correspondante à leur niveau de détails. Cela signifie qu'il faut, dans chaque étape du processus, spécifier quels tests seront effectués, à quel moment, par qui, dans quelles conditions, avec quels moyens et surtout de quelle manière.

Les tests de validations sont conçus dès les spécifications : ils permettent de vérifier l'adéquation du produit aux exigences fonctionnelles du client (si les besoins captés à l'analyse sont bien couverts). Les points de départ sont les cas d'utilisation et les scénarios.

Exemple de test de validation pour un logiciel de dessin :

Désignation	*Démarche à suivre*	*Résultat attendu*	*oui/non*	*Remarques*
Déplacement d'un objet	- Sélectionner dans la barre d'outil l'icône correspondante à un objet - Cliquer sur le champ de travail pour créer l'objet - Déplacer l'objet sur le champ de travail - Cliquer sur Annuler	L'objet revient à sa position initiale		
...				

Puis, en fin de développement, il est complété afin de valider le produit (ici le produit ne sera pas validé !) :

Désignation	*Démarche à suivre*	*Résultat attendu*	*oui/non*	*Remarques*
Déplacement d'un objet	Sélectionner dans la barre d'outil l'icône correspondante à un objet Cliquer sur le champ de travail pour créer l'objet Déplacer l'objet sur le champ de travail Cliquer sur Annuler	L'objet revient à sa position initiale	**non**	l'objet disparaît du champ de travail
...				

5 . Rédiger un plan de test de validation de l'application.

III . Conception

Étape n°III.1 : définir une architecture logicielle

On fait le choix d'une **architecture très classique en 3 couches** (*pattern Architectural Layer*).

L'**architecture trois tiers** (« *3-Tier* » anglais *tier* signifiant étage ou niveau) ou architecture à trois niveaux est un **modèle logique d'architecture applicative** qui vise à séparer très nettement trois couches logicielles au sein d'une même application ou système, à modéliser et présenter cette application comme un empilement de trois couches dont le rôle est clairement défini :

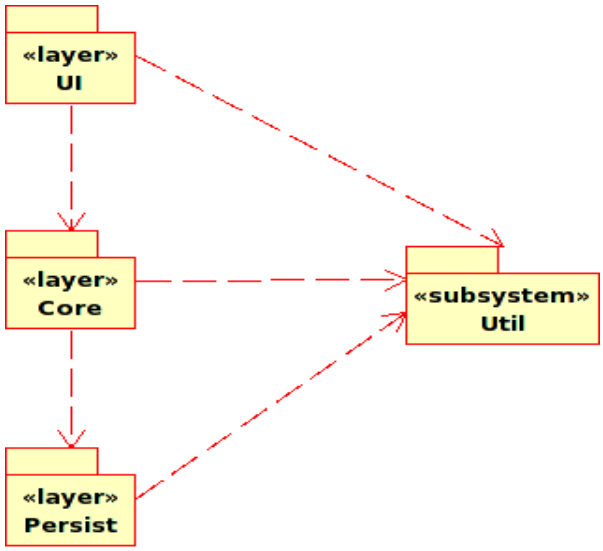
- la **présentation des données** (**couche présentation**) : correspondant à l'affichage, la restitution sur le poste de travail, le dialogue avec l'utilisateur (IHM) ;
- le **traitement métier des données** (**couche métier ou application**) : correspond à la partie fonctionnelle de l'application, celle qui implémente la « logique » applicative ;
- l' **accès aux données persistantes** (**couche accès aux données ou persistance**) : correspondant aux données qui sont destinées à être conservées sur la durée, voire de manière définitive.

Source Wikipedia : `http://fr.wikipedia.org/wiki/Architecture_trois_tiers`

Remarque : il existe aussi le **Modèle-Vue-Contrôleur** (**MVC**, *Model-View-Controller*) qui est une architecture et une méthode de conception qui organise l'interface Homme-Machine (IHM) d'une application logicielle. Il divise l'ihm en un **modèle** (modèle de données), une **vue** (présentation, interface utilisateur) et un **contrôleur** (logique de contrôle, gestion des évènements, synchronisation), chacun ayant un rôle précis dans l'interface.

Source Wikipedia : `http://fr.wikipedia.org/wiki/Mod%C3%A8le-Vue-Contr%C3%B4leur`

L'architecture ***n-tiers*** est utilisé le plus souvent en s'appuyant sur le *pattern Architectural Layer* :

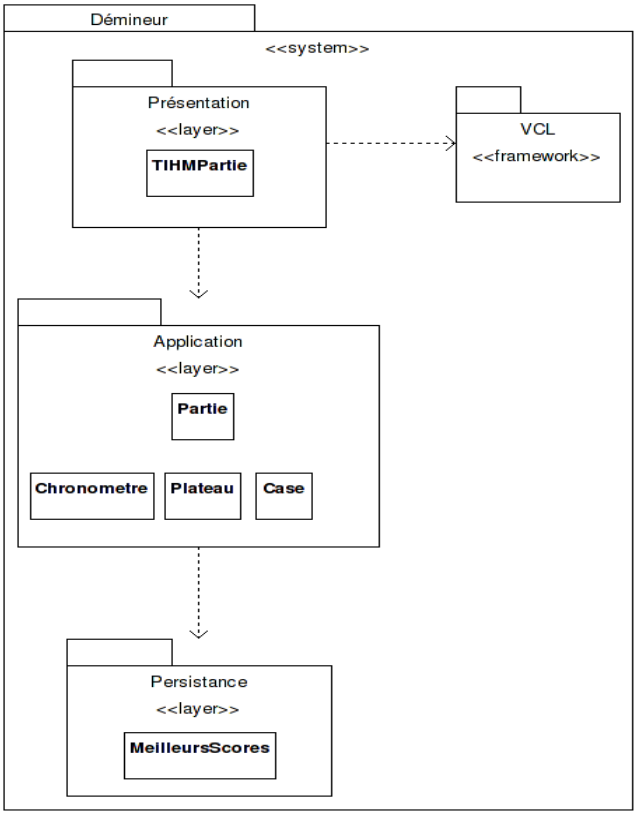


Cela donne le découpage suivant :

Présentation	**IHM** (la partie visible du système pour l'utilisateur)
Application	**Jouer une partie de démineur**, etc ...
Persistance	Fichier (et/ou Base de données) des meilleurs scores

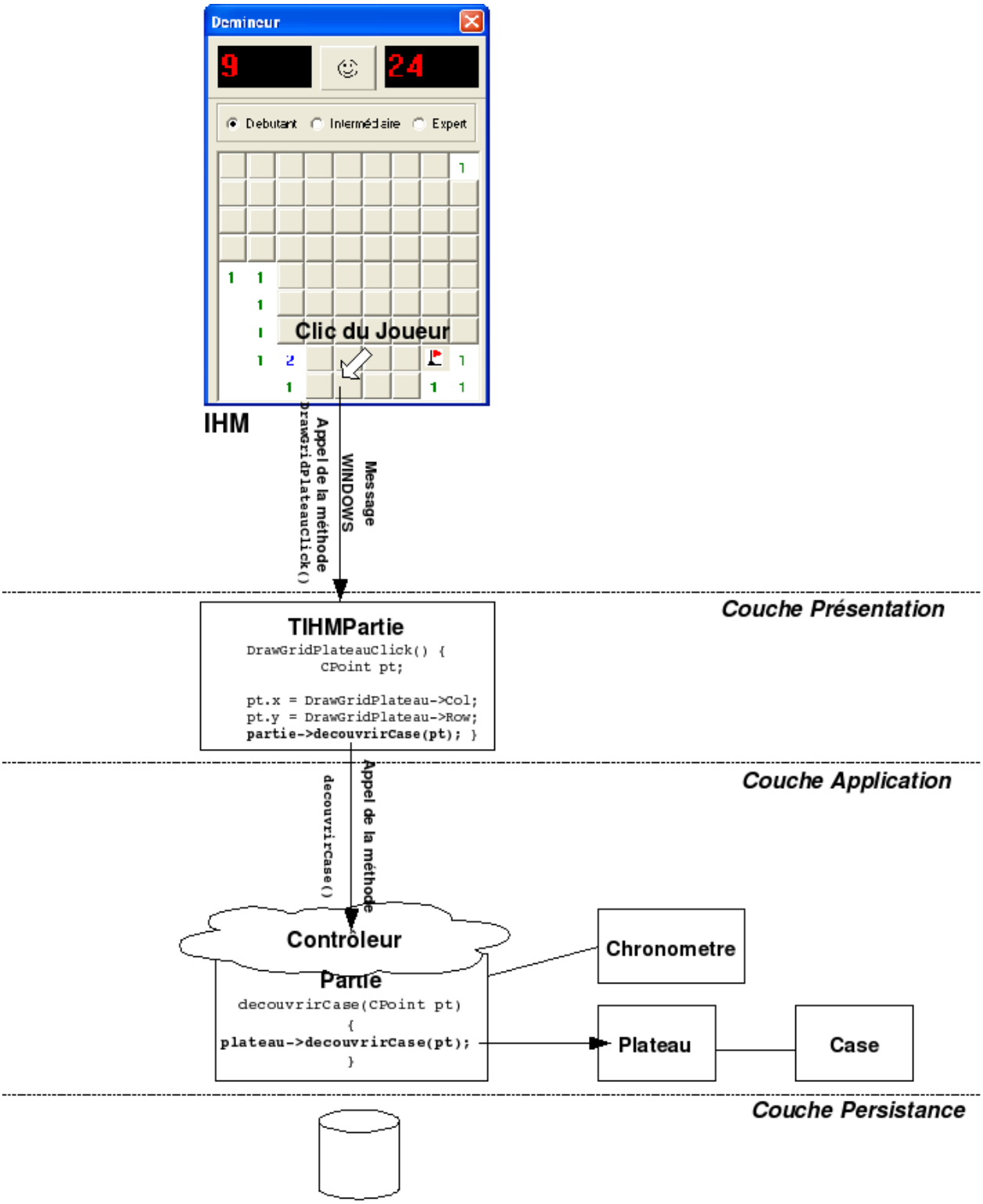
Remarque : dans cette itération, on ne réalise pas le cas d'utilisation « Voir les meilleurs scores ». Donc la couche Persistance ne sera pas implémentée.

On représente l'architecture de l'application en utilisant un **diagramme de *packages*** d'architecture :



Le principal objectif de cette séparation en trois couches est d'isoler la logique de l'application de(s) classe(s) de présentation (IHM) ainsi que d'interdire un accès direct aux données stockées par ces classes. L'intérêt est de pouvoir modifier l'interface de l'application sans devoir modifier la couche Application et de pouvoir changer de stockage sans avoir à retoucher l'interface et la couche Application.

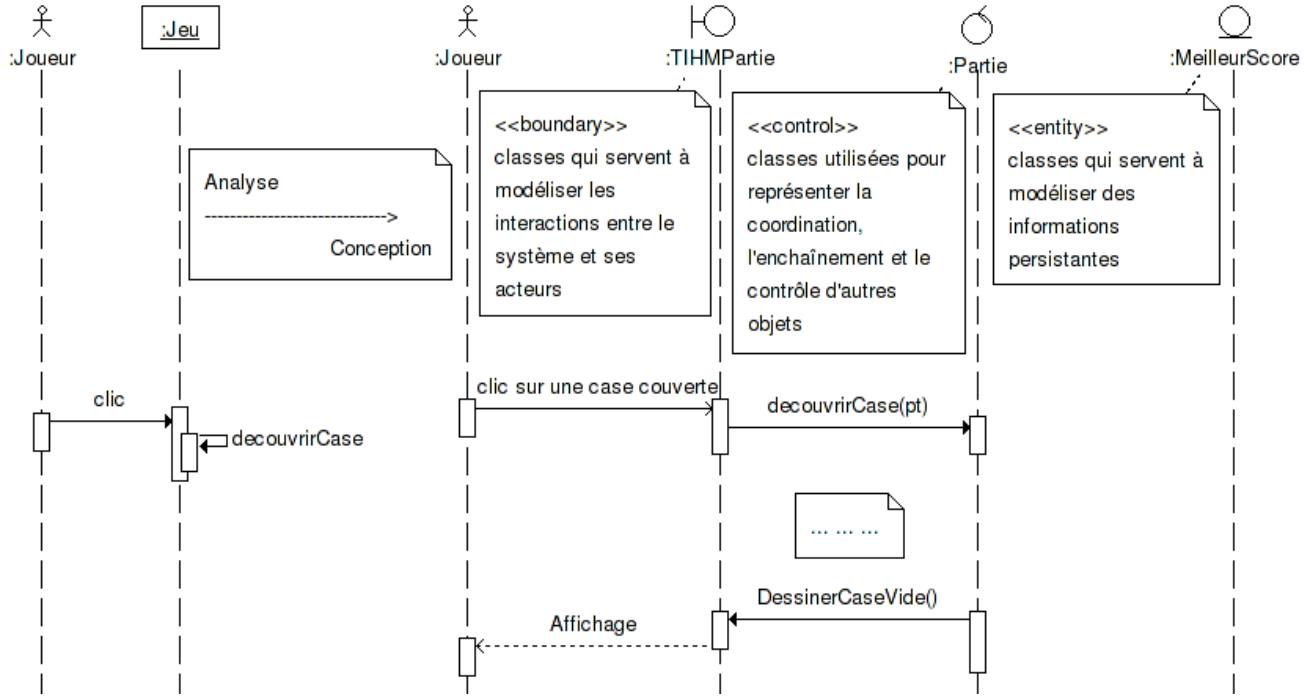
Pour éviter que les objets de l'IHM ne soient obligés de connaître les détails de la couche Application, on introduit un objet, souvent appelé **contrôleur**, qui joue le rôle de façade vis-à-vis de la couche Présentation :



Voir pattern GRASP Contrôleur

Étape n°III.2 : passer de l'analyse à la conception

On va remplacer le système vu comme une « boîte noire » (du point de vue de l'analyse) par des objets logiciels (du point de vue de la conception) :



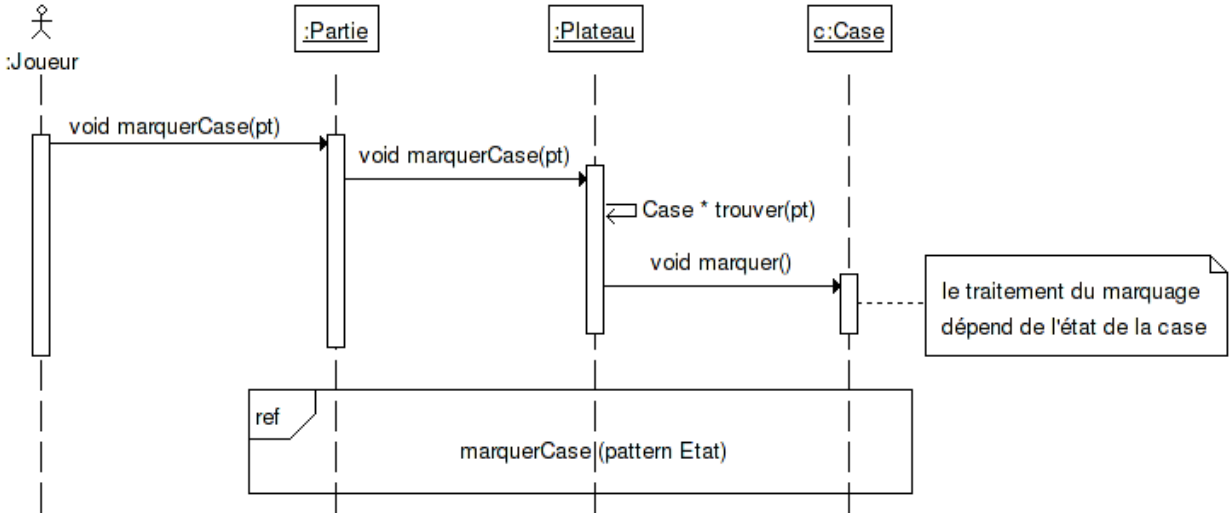
Le diagramme de séquence système (DSS) a fait apparaître deux opérations systèmes : **découvrirCase()** et **marquerCase()**.

On va élaborer les diagrammes de séquences de conception pour ces deux opérations.

L'opération marquerCase()

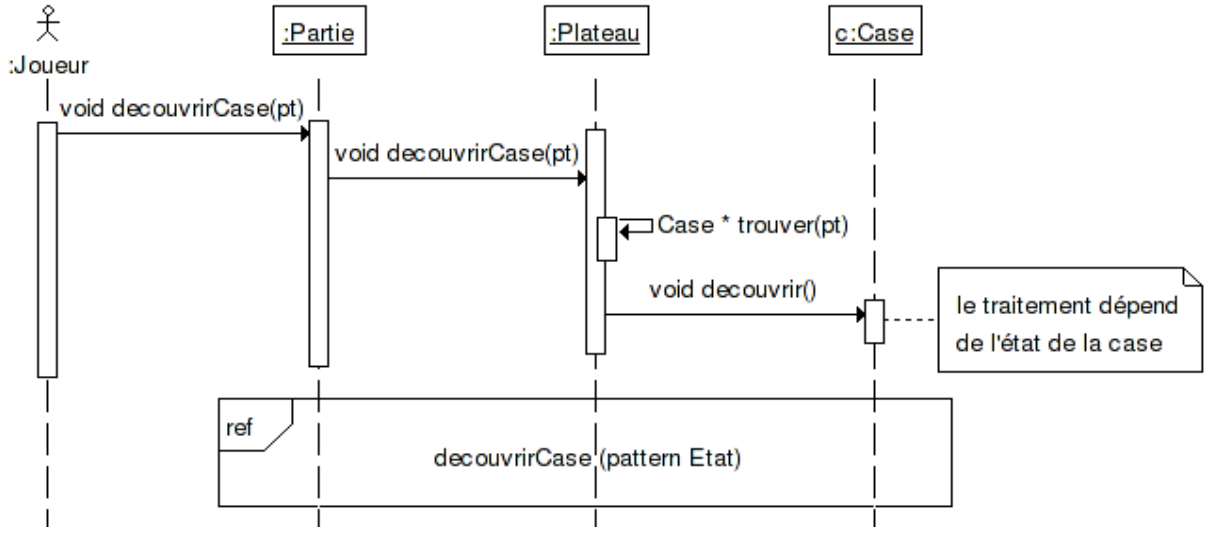
L'objet contrôleur le plus naturel semble être **Partie**. Par contre, **Partie** n'est pas l'expert des cases ni de leurs coordonnées. Il va donc déléguer le travail à l'expert des cases, soit le **Plateau**. Celui-ci doit d'abord retrouver la case à partir des coordonnées fournies en paramètre puis lui déléguer le traitement du marquage proprement dit (***ref***).

On obtient le diagramme de séquence simplifié suivant :



L'opération découvrirCase()

Ce diagramme est très similaire au précédent :



Avant de terminer ces diagrammes, on va introduire des éléments de conception détaillée essentiels dans l'ACOO.

Étape n°III.3 : utiliser des éléments de conception détaillée

La relation Plateau-Case

On va tout d'abord concevoir la relation Plateau-Case.

Les cases feront toujours partie du Plateau et celui-ci gérera leur création et leur destruction : les cases sont donc dans une relation de **composition** avec le Plateau.

Puis, il faut traduire la **navigabilité *** entre Plateau et Case. Cela revient à choisir un conteneur pour la collection de cases. 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. Étant donné notre architecture à 3 niveaux séparés, on s'interdit un choix parmi les conteneurs proposés par l'EDI (Environnement de Développement Intégré ou IDE en anglais) *Borland Builder* pour ne pas « infecter » notre modèle. On se replie sur les conteneurs de la **STL** (*Standard Template Library*) qui assureront une portabilité de l'application.

En suivant le document en annexe pour le choix d'un conteneur STL, on obtient :

Tests	*Oui/Non*
L'ordre est important	**oui** (on a une système de coordonnées x,y)
Trié par clé	**oui** (une coordonnée)
Doublons autorisés	**non** (unicité de la case sur le plateau)
Clés et valeurs séparées	**oui** (une coordonnée donne une case)
Conteneur choisi	▷*Map*

Les patterns GoF en conception détaillée

Il apparaît souvent dans le développement d'application des problèmes récurrents, qui peuvent être généralisé, même s'ils apparaissent dans un contexte particulier. Les *design patterns* apportent des solutions connues à ces problèmes. Les développeurs Orientés Objet expérimentés construisent des « catalogues » de principes généraux et de solutions qui les guident lorsqu'ils créent des logiciels. Ces principes nommés et présentés sous forme structurée qui décrit le problème et la solution, peuvent être qualifiés de ***patterns***.

Plus simplement, un **pattern est un couple problème/solution** nommé et bien connu qu'on peut appliquer à de nouveaux contextes. Il est accompagné de conseils sur la façon de l'appliquer, d'une présentation des inconvénients, implémentations, variantes, ...

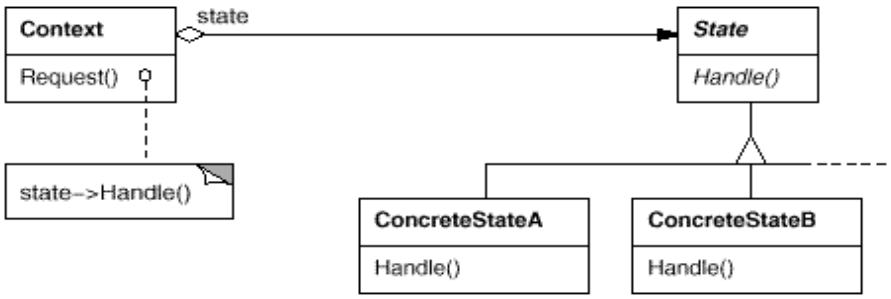
L'idée des patterns est venue de **Kent Beck** (également célèbre pour avoir créer ***eXtremme Programming***) au milieu des années 80. C'est surtout 1994 qui marqua l'histoire de la programmation OO avec la publication du *best-seller* ***« Design Patterns »*** de Gamma, Helm, Johnson et Vlissides. Considéré comme la bible du domaine, il décrit 23 *patterns* pour la conception OO. Les auteurs sont surnommés le ***« Gang Of Four »*** (la bande des quatre) et leurs *patterns* seront connus comme les ***patterns GoF***.

Ici, notre problème est : **le comportement de l'objet Case dépend de son état**. Le réflexe d'un développeur peu expérimenté en programmation OO serait d'implémenter une logique conditionnelle complexe difficile à maintenir et à faire évoluer. Une **médiocre conception** donnerait une solution semblable à cela :

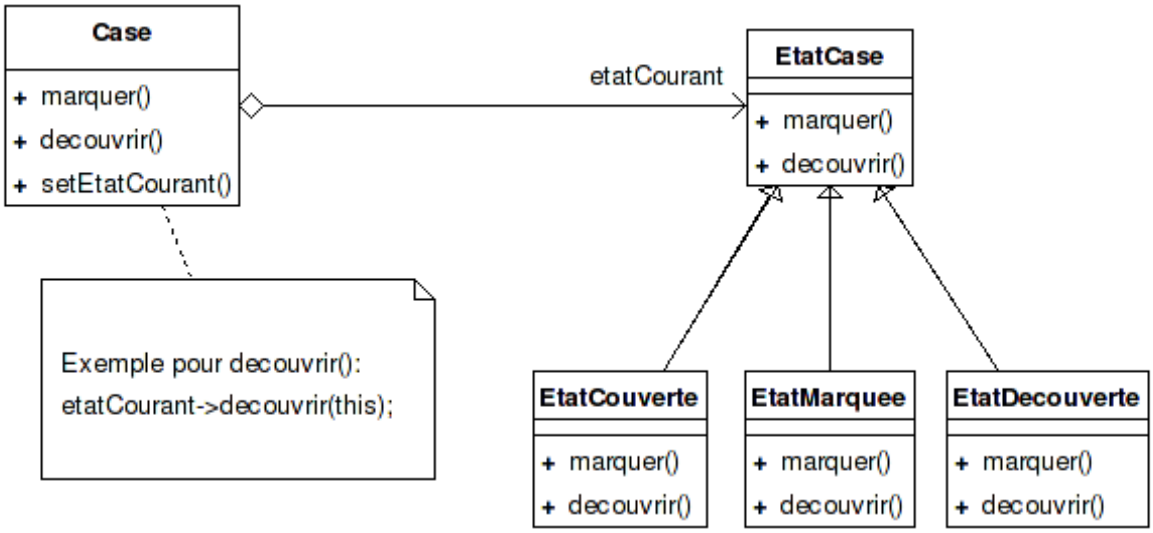
```
// Un démineur : http://chgi.developpez.com/drawgrid/
void Case::Decouvre(int x,int y) {
   if(y<=14)    {
      if (TabVal[x][y+1]>0 || TabCase[x][y+1]==1) TabCase[x][y+1]=1;
      else TabCase[x][y+1]=2;
   }
   if(y>0)     {
      if (TabVal[x][y-1]>0 || TabCase[x][y-1]==1) TabCase[x][y-1]=1;
      else TabCase[x][y-1]=2;
   }
   if(x<=14)   {
      if (TabVal[x+1][y]>0 || TabCase[x+1][y]==1) TabCase[x+1][y]=1;
      else TabCase[x+1][y]=2;
   }
   if(x<=14 && y<=14)    {
      if (TabVal[x+1][y+1]>0 || TabCase[x+1][y+1]==1) TabCase[x+1][y+1]=1;
      else TabCase[x+1][y+1]=2;
   }
   if(x<=14 && y>0)    {
      if (TabVal[x+1][y-1]>0 || TabCase[x+1][y-1]==1) TabCase[x+1][y-1]=1;
      else TabCase[x+1][y-1]=2;
   }
   if(x>0)    {
      if (TabVal[x-1][y]>0 || TabCase[x-1][y]==1) TabCase[x-1][y]=1;
      else TabCase[x-1][y]=2;
   }
   if(x>0 && y<=14)    {
      if (TabVal[x-1][y+1]>0 || TabCase[x-1][y+1]==1) TabCase[x-1][y+1]=1;
      else TabCase[x-1][y+1]=2;
   }
   if(x>0 && y>0)    {
      if (TabVal[x-1][y-1]>0 || TabCase[x-1][y-1]==1) TabCase[x-1][y-1]=1;
      else TabCase[x-1][y-1]=2;
   }
}
```

Le pattern Etat

On va donc mettre en oeuvre le ***pattern* État** pour déporter le comportement variable dans de nouvelles classes. Le modèle est le suivant :



Si on l'applique à notre problème et on obtient le diagramme de classes suivant :



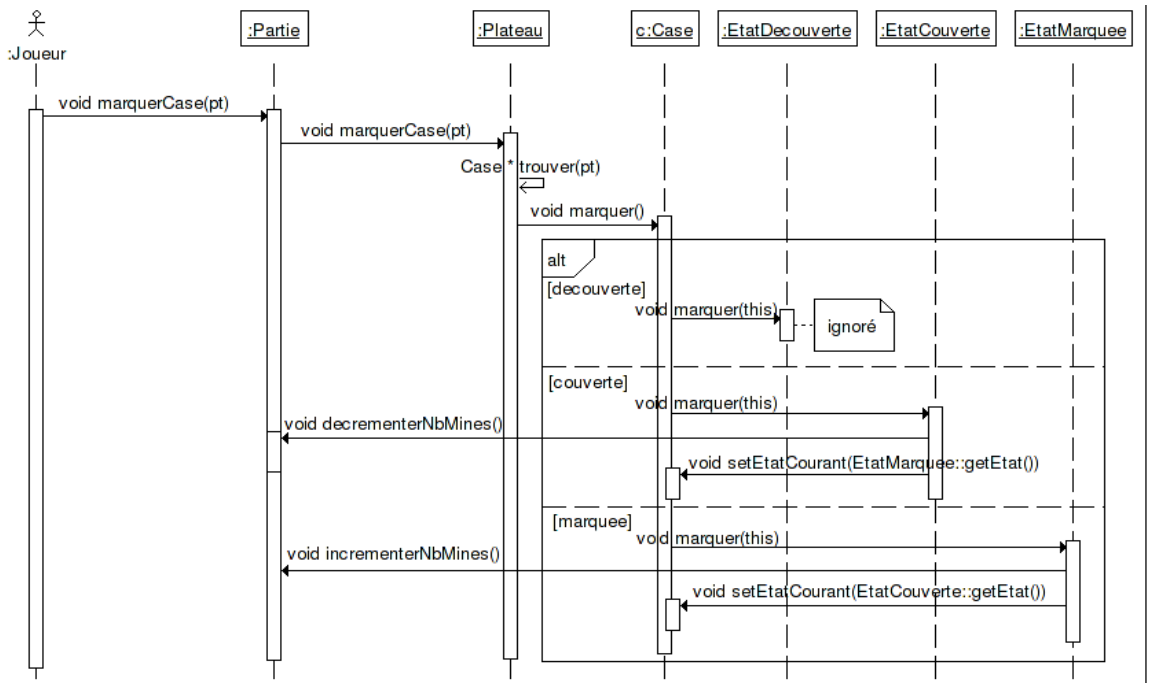
Chaque fois que la case change d'état, l'objet Case change l'objet état qu'il utilise. Si on découvre une case, celle-ci passe de couverte à découverte, alors Case remplacera l'instance EtatCouverte par l'instance EtatDecouverte :

```
void EtatCouverte::decouvrir(Case* c)
{
   c->setEtatCourant(EtatDecouverte::getEtat());
   // ...
}
```

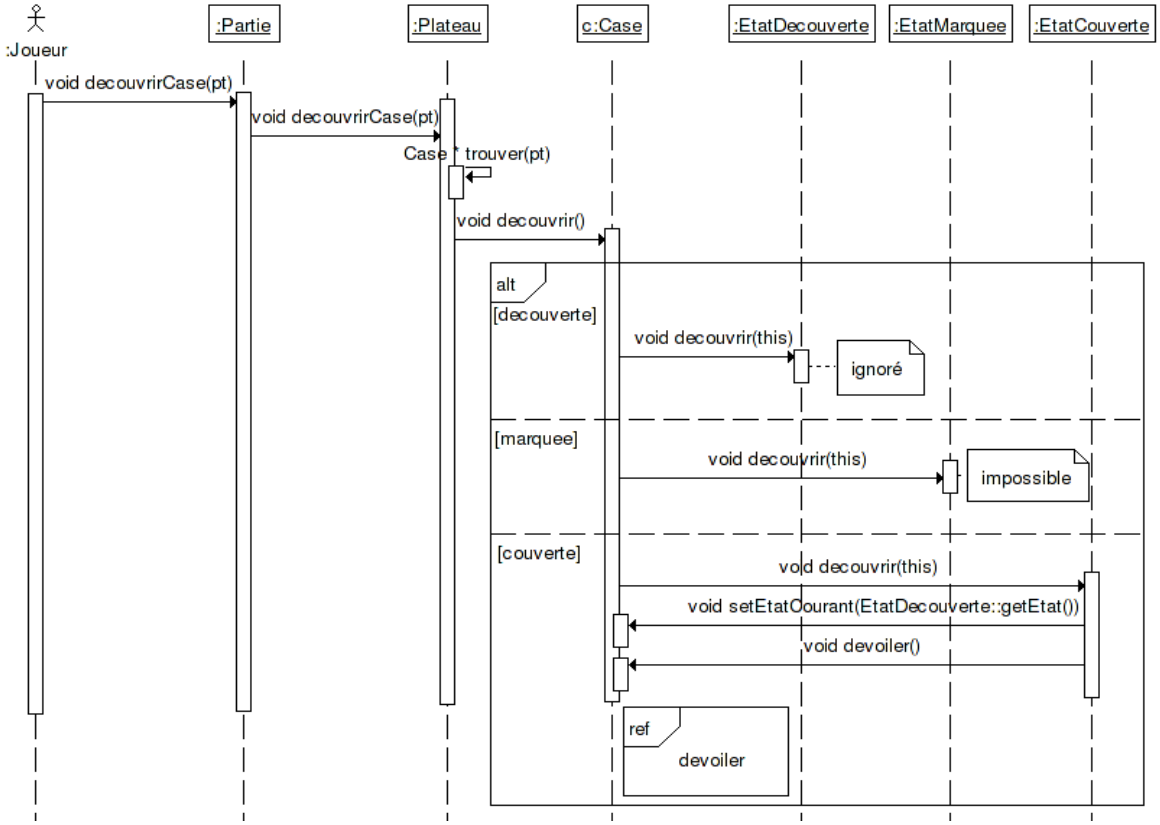
En résumé, la Case va donc déléguer à son instance **etatCourant** le comportement variable du marquage et de la découverte des cases.

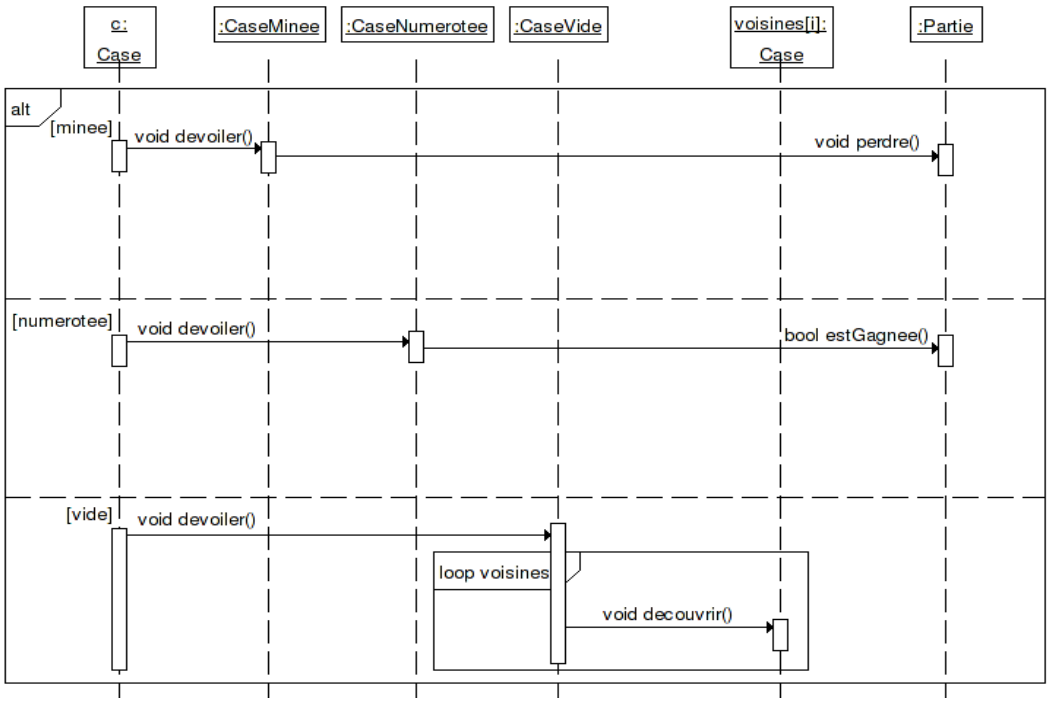
On peut maintenant terminer les diagrammes de séquence de conception en tenant compte de la mise en oeuvre du *pattern* Etat.

L'opération marquerCase()



L'opération découvrirCase()

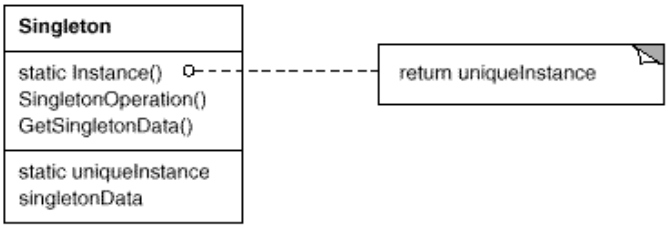




Plusieurs classes vont avoir besoin d'un accès au contrôleur Partie, pour incrémenter et décrémenter le compteur de mines, pour signifier la fin de la partie, ... La classe Partie ne doit avoir qu'une seule instance à la fois : on joue une partie unique. C'est un problème connu de la programmation OO et la solution est d'utiliser un *pattern* Singleton. C'est le plus connu et le plus simple des ***patterns*** **GoF**.

Le pattern Singleton

L'objectif du *pattern Singleton* est **de garantir qu'une classe n'a qu'une seule instance**. La meilleure solution consiste à confier à la classe elle-même la responsabilité d'assurer l'unicité de son instance.



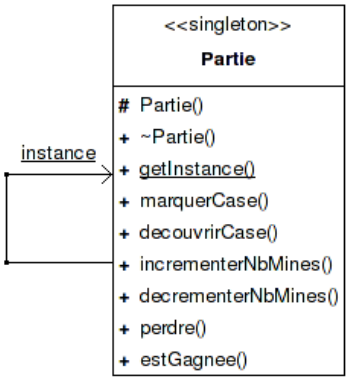
On va appliquer ce modèle à la classe **Partie** :

```
class Partie
{
private:
    static Partie* instance; //membre static pour exister indépendamment
protected:
    Partie(); //pour interdire l'accès au constructeur de l'extérieur de la classe
public:
    ~Partie();
    static Partie* getInstance(); //pour accéder au membre static instance
    //...
};
```

```
Partie* Partie::instance = NULL; //initialisation de l'instance

Partie* Partie::getInstance()

{
   if(instance == NULL) //aucune instance ?
      instance = new Partie(); //alors on instancie
   return instance; //on retourne l'instance
}
```



Et pour les classes qui auront besoin d'accéder au contrôleur Partie :

```
Partie *partie = Partie::getInstance(); //récupère l'instance unique
```

IV . Réalisation

Étape n°IV.1 : coder un diagramme de séquence

Une fois le diagramme de classes de conception établi, on peut passer à l'implémentation en codant dans un langage les classes.

Remarque : certains outils de génie logiciel sont capables de générer du code à partir des diagrammes déjà réalisés (le plus souvent à partir du diagramme de classes mais parfois aussi à partir du diagramme de séquence comme *Together* de *Borland*).

6 . Coder les diagrammes de séquence decouvrirCase et marquerCase.

IV . Validation

Étape n°IV.1 : réaliser les tests de validation

7 . Rédiger un compte-rendu de tests de validation et valider l'application.

V . Déploiement

Étape n°V.1 : réaliser une procédure d'installation de l'application

Cette étape se décompose en deux parties :

- fabriquer une version finale de l'application (*release*)
- déployer cette application sur une machine

Ces deux parties sont très souvent dépendantes de la plateforme et des outils utilisés.

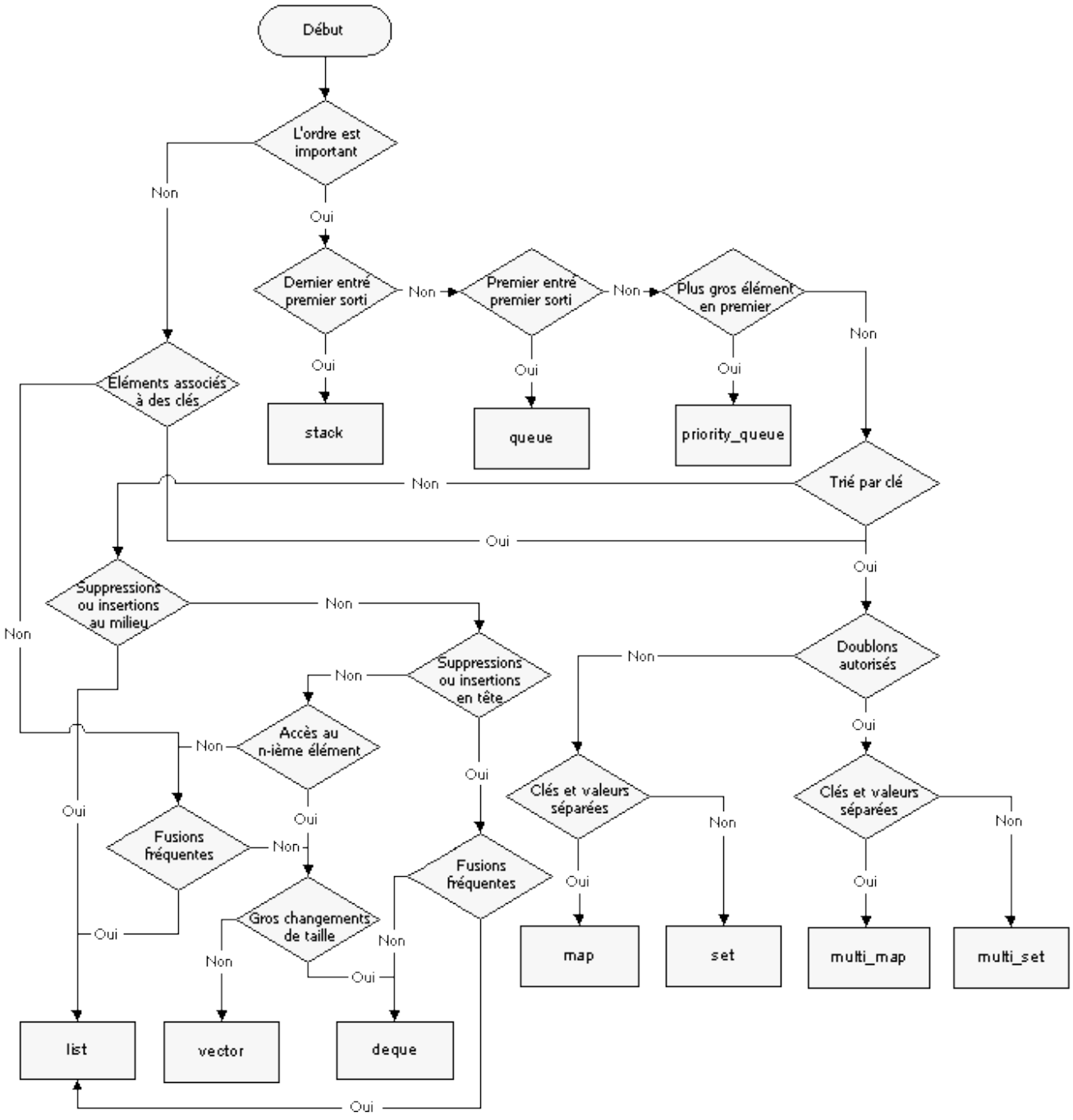
Dans cette manipulation, on peut évaluer l'importante des contraintes fixées : l'environnement de développement *Builder C++* de *Borland*© sous *Microsoft Windows*©.

8 . En utilisant le document en Annexe, fabriquer une version finale de l'application.

Pour le déploiement de l'application, notre choix se porte sur **Inno Setup** (http://www.jrsoftware.org/isinfo.php). L'installeur Inno Setup est aussi performant (si ce n'est plus) que *InstallShield* et autres. Il est surtout gratuit et fourni avec les codes sources (pour Delphi 2 à 5). On pourra aussi utiliser **ISTool** (http://www.istool.org/) qui est un assistant d'aide à la création de scripts pour Inno Setup, rendant leur création beaucoup plus simple. C'est le complément idéal à Inno Setup.

9 . En utilisant le tutoriel fourni pour cette application, créer l'installateur et déployer l'application sur une machine *vierge*.

Annexe 1 : choix d'un conteneur



Annexe 2 : Borland C++ Builder

Sur le site `cpp.developpez.com`, vous trouverez une FAQ pour la création d'exécutable sous *Borland C++ Builder* : `http://cpp.developpez.com/faq/bcb/?page=compilation` .

De manière générale il faut résoudre deux problèmes :

- les droits et les fichiers à déployer : un fichier a été installé avec *Borland C++ Builder* dans le répertoire d'installation qui se nomme "`deploy.txt`". Il vous faut consulter ce fichier pour savoir quels fichiers vous pouvez déployer avec quelle version.
- fabriquer un exécutable indépendant (statique) ou non (dynamique). Sous *Windows*, la dépendance est le plus souvent assurée par des DLLs.

Exemple pour créer un exécutable indépendant :