Études de cas complémentaires

Mots-clés

■ Opération système ■ Diagramme de communication ■ Diagramme de séquence ■ Méthode ■ Typage ■ Navigabilité des associations ■ Dépendances entre classes ■ Collection d'objets ■ Java ■ C# ■ Design pattern.

Ce chapitre va nous permettre de compléter au moyen d'autres études de cas notre panorama des techniques de modélisation UML, en insistant sur celles mises en œuvre durant l'activité de conception, en particulier :

- Les diagrammes d'interaction ;
- Les diagrammes de classes.

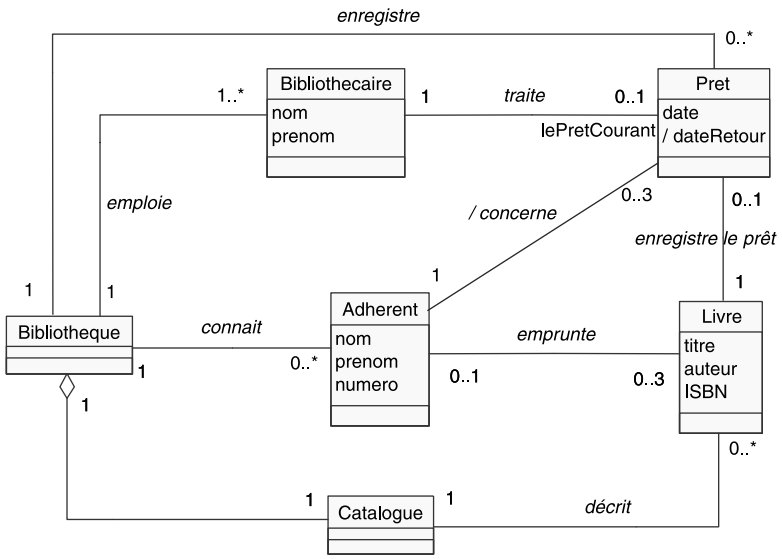
Nous apprendrons également à utiliser certains design patterns supplémentaires.

ÉTUDE DU SYSTÈME D'INFORMATION D'UNE BIBLIOTHÈQUE

Nous allons partir d'un modèle d'analyse d'un système informatique qui doit permettre de gérer une bibliothèque. Cette bibliothèque ne prête que des livres dans un premier temps.

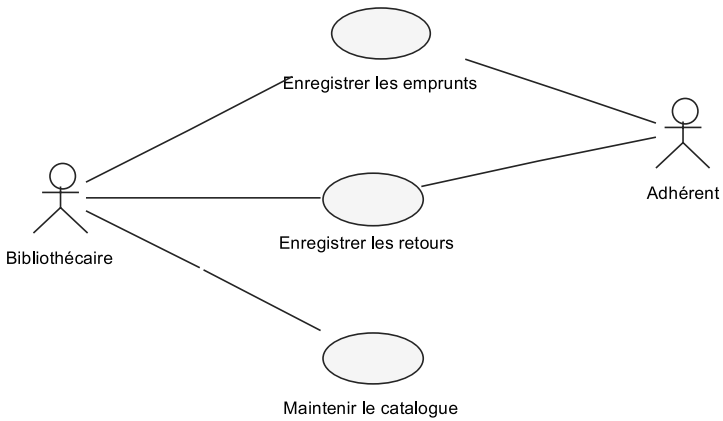
Le diagramme de classes d'analyse est montré sur la figure suivante.

Figure 8-1.
Diagramme de classes d'analyse du S.I. de la bibliothèque



Le diagramme de cas d'utilisation provisoire est présenté ci-après.

Figure 8-2.
Diagramme de cas d'utilisation préliminaire du S.I. de la bibliothèque



Poursuivons par les opérations système du cas d'utilisation *Enregistrer les emprunts* qui sont détaillées sur le diagramme de séquence système suivant.

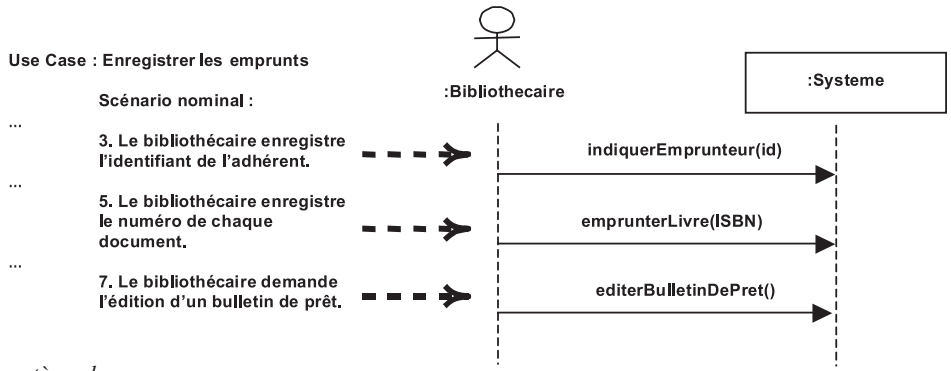


Figure 8-3.
Opérations système du cas d'utilisation Enregistrer les emprunts

Nous allons ensuite nous intéresser au contrat de l'opération *emprunterLivre*. Notez que nous passons sous silence la première opération *indiquerEmprunteur*, mais uniquement parce qu'elle est moins riche que la deuxième et n'apporterait rien à notre étude. Dans la réalité, les opérations système doivent bien sûr être conçues par ordre chronologique.

- Nom
  emprunterLivre (ISBN).
- Responsabilités
  Enregistrer l'emprunt d'un livre identifié par son numéro ISBN.
- Références
  Cas d'utilisation *Enregistrer les emprunts*.
- Pré-conditions
  - le catalogue de livres existe et n'est pas vide ;
  - l'adhérent a été reconnu par le système et n'a pas atteint le seuil maximal d'emprunts.
- Post-conditions
  - un prêt p a été créé ;
  - l'attribut *date* de p a été positionné à la date du jour ;
  - l'attribut *dateRetour* de p a été positionné à (la date du jour + deux semaines) ;
  - p a été lié au livre l dont l'attribut ISBN vaut l'ISBN passé en paramètre ;
  - p a été lié à l'adhérent concerné et à la bibliothèque.

CONCEPTION DU COMPORTEMENT (DIAGRAMMES D'INTERACTION)

EXERCICE 8-1.
Diagramme de communication de conception

Développez un diagramme de communication pour l'opération système *emprunterLivre* à partir des informations précédentes.

Détaillez chacune de vos décisions de conception.

Ne vous préoccupez pas des classes d'interface (<<boundary>>).

Développez un diagramme de communication pour EMPRUNTERLIVRE.

Solution

Notre diagramme de communications démarre par la réception d'un message système venant d'un acteur. Puisque la possibilité nous est donnée de ne pas nous préoccuper des objets d'interface, nous allons directement choisir un objet qui traitera cet événement système.

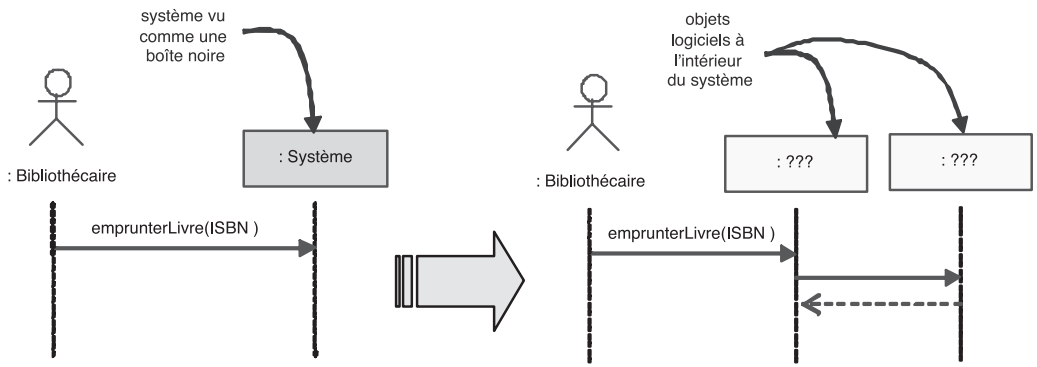


Figure 8-4.
Passage de l'analyse à la conception

La solution que nous avions mise en œuvre au chapitre précédent (étape 6) nous amenait à introduire un objet artificiel de conception de type « contrôleur » que nous aurions pu appeler ici *ControleurEmprunts*.

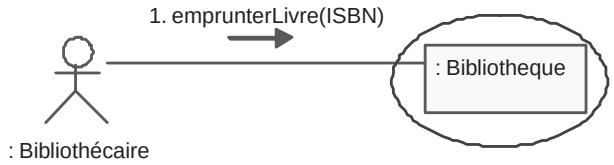
En fait, dans le cas d'un système comprenant un nombre restreint d'opérations système, une approche plus simple est possible, qui n'oblige pas à ajouter une nouvelle classe. Cette solution consiste à utiliser comme contrôleur une instance d'une classe d'analyse existante :

- soit un objet représentant le système entier ou l'organisation elle-même ;
- soit un objet représentant un rôle qui aurait réalisé l'opération système.

Le premier choix possible est le plus direct, mais son inconvénient majeur, c'est que l'on affecte le traitement de toutes les opérations système à un objet unique, qui risque rapidement d'être surchargé en matière de responsabilités. C'est une solution acceptable dans notre exemple, et la classe candidate est la classe *Bibliotheque*.

La seconde possibilité permet en général de répartir les opérations système sur plusieurs objets. Mais ici la seule classe candidate est la classe *Bibliothecaire*, ce qui n'apporte rien par rapport à la solution précédente. Nous conservons donc *Bibliotheque* comme objet contrôleur pour notre opération système. Le diagramme de communication peut donc démarrer de la manière suivante.

Figure 8-5.
Diagramme de communication de l'opération emprunterLivre *(début)*



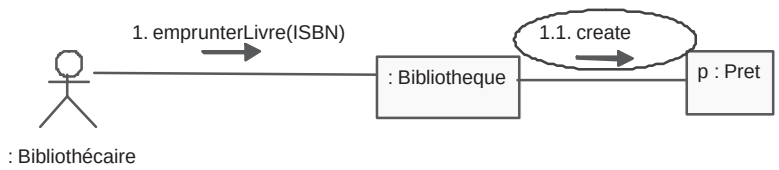
Comment devons-nous maintenant procéder ? Il faut tout simplement étudier le contrat d'opération. Rappelez-vous cependant que les post-conditions répertoriées dans le contrat ne sont pas forcément ordonnées. Dans notre cas, il semble quand même raisonnable de commencer par la création de l'objet prêt puisque les autres post-conditions s'appliquent à ses attributs ou à ses liens.

La question qui se pose alors est la suivante : quel objet doit avoir la responsabilité de la création du prêt p ?

Si nous revenons au diagramme de classes d'analyse, nous constatons que quatre classes possèdent déjà une association avec la classe *Prêt*, à savoir : *Bibliothecaire*, *Bibliotheque*, *Livre* et *Adherent*. Elles sont donc toutes de bonnes candidates initiales. Cependant, le meilleur choix est en général fourni par la classe qui possède une association de type composition, agrégation ou « enregistre ». Dans notre exemple, la classe *Bibliotheque* est justement liée à *Pret* par une association « enregistre ». Comme, en outre, la *Bibliotheque* est déjà le contrôleur, elle constitue le candidat idéal.

Le diagramme de communication devient alors :

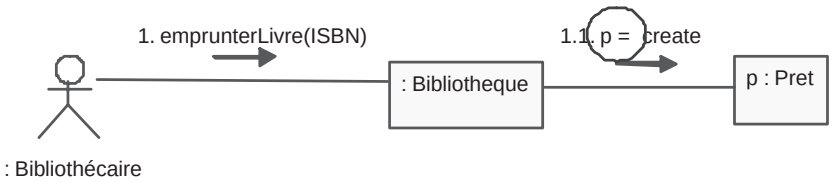
Figure 8-6.
Diagramme de communication de l'opération emprunterLivre *(suite)*



On notera l'utilisation déroutante au premier abord du message en anglais *create*. En effet, en toute rigueur, l'objet p ne peut pas recevoir le message *create*, puisqu'il n'existe pas encore ! Il s'agit là d'une bonne pratique qui évite d'entrer dans des considérations qui dépendent du langage de programmation cible. En Java (ou en C#), ce message de création se traduira probablement par le mot-clé `new` et l'appel du constructeur de la classe *Pret*, qui retournera une référence sur le nouvel objet.

Ainsi, la bibliothèque a maintenant une référence sur l'objet p nouvellement créé. Notez que, puisque le message de création renvoie toujours une référence sur le nouvel objet, nous ne montrons pas le retour explicitement, bien que cela soit légal. On obtiendrait une représentation un peu plus lourde telle que celle qui est illustrée par la figure suivante.

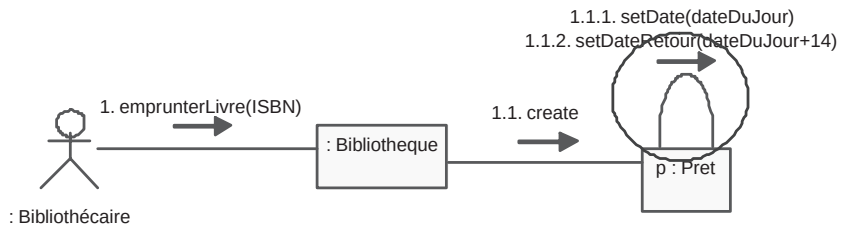
Figure 8-7.
Diagramme de communication de l'opération emprunterLivre *(suite alternative)*



Poursuivons notre réflexion. Les attributs *date* et *dateRetour* ne pourraient-ils pas être positionnés par l'objet p dès sa création, suite à la récupération de la date du jour par une méthode que nous ne détaillerons qu'en conception détaillée ?

C'est ce que nous représentons sur le diagramme de communication suivant.

Figure 8-8.
Diagramme de communication de l'opération emprunterLivre *(suite 2)*



Remarquez une fois de plus (voir chapitre 7, étape 8) l'utilisation de la notation décimale des numéros de messages, qui permet de représenter l'imbrication des messages. On notera également l'utilisation de la boucle au-dessus de l'objet qui symbolise un lien de l'objet vers lui-même comme support d'un message « à soi-même ».

Le diagramme de séquence correspondant (avec la notation des « focus of control ») est donné ci-après en comparaison.

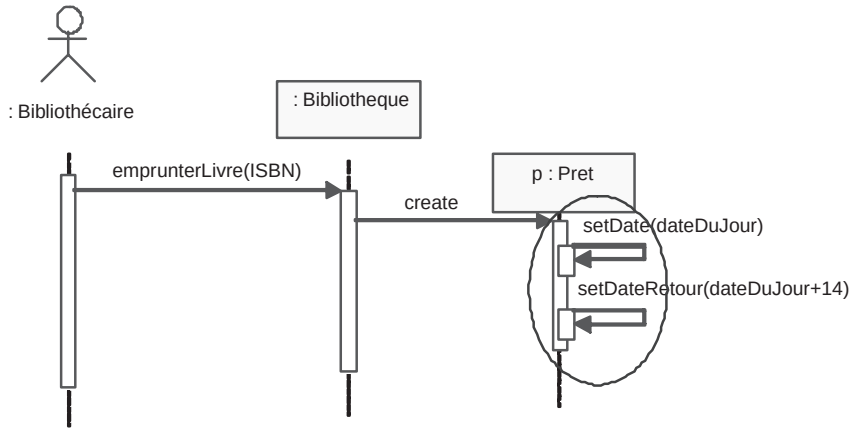


Figure 8-9. *Diagramme de séquence de l'opération* emprunterLivre *(suite 2)*

Revenons à l'établissement du contrat de l'opération. Que devons-nous faire maintenant ? Il nous faut un lien avec le livre dont l'attribut ISBN vaut l'ISBN passé en paramètre de l'opération *emprunterLivre*.

Quel objet est-il le mieux placé pour trouver un livre en fonction de son ISBN ?

Reprenons le diagramme de classes d'analyse (figure 8-1) : la classe *Catalogue* est la candidate idéale, puisque le catalogue connaît tous les livres. Mais quel objet doit-il lui formuler la demande ? Ce sera soit le prêt p, soit la bibliothèque, d'après notre diagramme de communication.

Pour respecter le principe de faible couplage, il vaut mieux que la bibliothèque s'en charge, car elle possède déjà une association avec le catalogue, contrairement au prêt. De plus, il est fort probable que la bibliothèque doive collaborer avec le catalogue dans le cadre d'autres opérations système, par exemple lors de l'ajout de nouveaux livres. Il faut donc que la bibliothèque ait un lien permanent avec le catalogue, ce qui n'est pas du tout le cas du prêt. Toutes ces raisons font clairement pencher la balance en faveur de la bibliothèque.

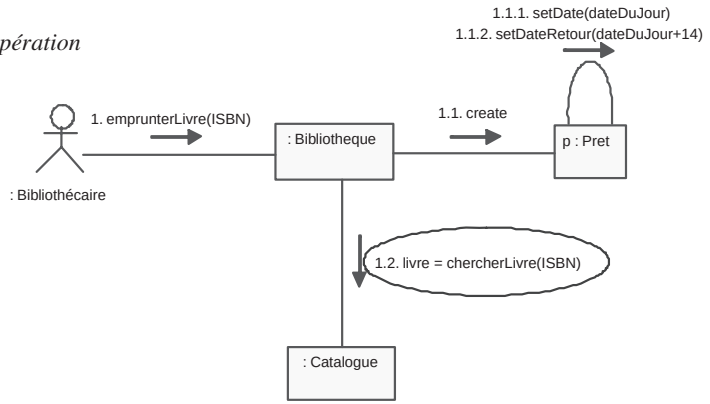
On notera que cela suppose que les objets catalogue et bibliothèque soient créés lors de l'initialisation du système et qu'un lien de visibilité soit établi entre eux. Il est courant en conception objet de travailler d'abord sur les collaborations entre objets « métier », puis de traiter dans un second temps le problème plus technique de l'initialisation du système informatique. Cela permet de s'assurer que les bonnes décisions d'affectation de responsabilités aux

objets dans le contexte des collaborations métier contraignent bien l'initialisation et pas le contraire.

Revenons à notre choix de communication entre la bibliothèque et le catalogue. Comment nommer le message ? Appelons-le *chercherLivre*, avec un numéro ISBN en paramètre, et comme retour une référence *livre* sur le bon objet livre.

Le diagramme de communication modifié est présenté ci-après.

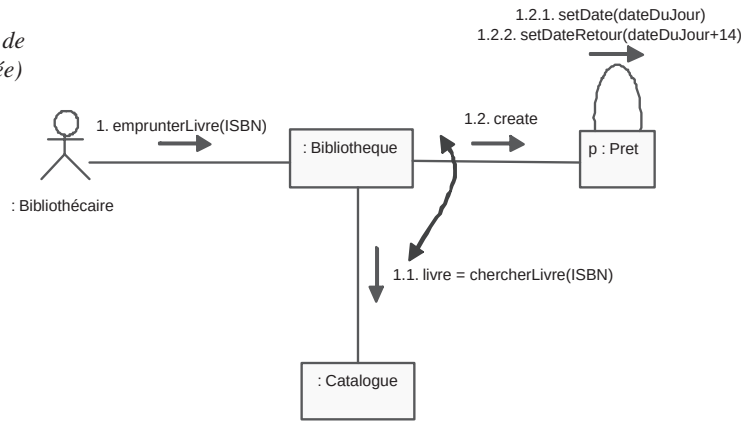
Figure 8-10.
Diagramme de comunication de l'opération emprunterLivre *(suite 3)*



Que se passe-t-il si le catalogue ne parvient pas à trouver un livre dont l'ISBN correspond à celui recherché ? Ne faut-il pas plutôt attendre d'avoir obtenu cette référence *livre* pour effectuer la création du prêt p ?

C'est tout à fait cela et, pour ce faire, il suffit simplement de permuter l'ordre des deux messages, comme cela est représenté sur la figure suivante. On notera que la numérotation décimale des messages d'affectation des attributs est mise à jour en conséquence.

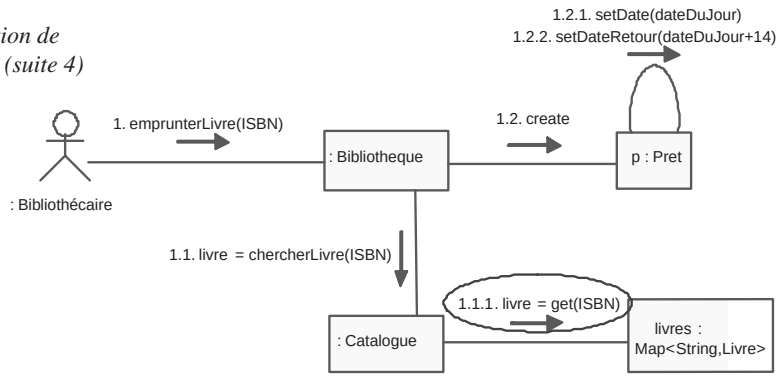
Figure 8-11.
Diagramme de communication de emprunterLivre *(suite 3 corrigée)*



Sur le diagramme de classes, il y a une association « 1 - * » entre *Catalogue* et *Livre*. Cela implique que le catalogue va utiliser une collection d'objets livres, probablement implémentée en Java sous la forme d'une *HashMap*. Nous avons déjà donné les règles de traduction des multiplicités d'associations en collections Java au chapitre précédent (figure 7-76). Nous y avons également expliqué l'utilisation du concept de « multi-objet » dans les diagrammes de communication pour éviter justement de prendre trop tôt des décisions de conception détaillée (voir figure 7-47). C'est la solution que nous allons aussi adopter ici, mais en utilisant la notation UML 2 qui remplace le multi-objet UML 1, à savoir une ligne de vie typée par une collection triée (notée livres : *Map*<String, Livre>). Le message devient plus générique que *chercherLivre*, puisque la collection représente un objet technique « sur étagère », et non pas un objet métier.

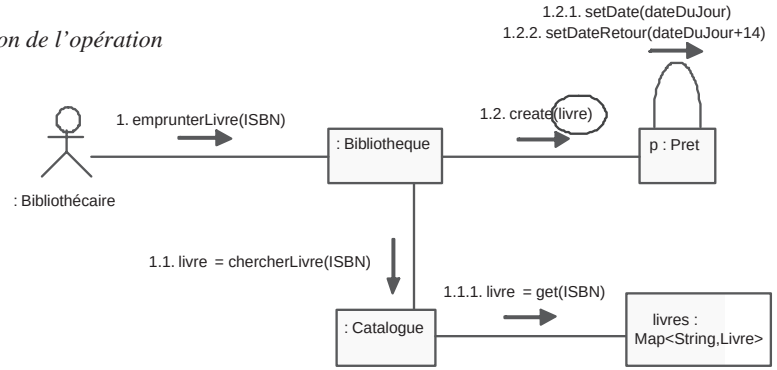
Le diagramme de communication est complété comme suit :

Figure 8-12.
Diagramme de communication de l'opération emprunterLivre *(suite 4)*



Maintenant que nous avons localisé le bon livre et que sa référence a été retournée à la bibliothèque, nous pouvons nous en servir pour établir le lien entre le livre et le prêt. La solution la plus directe consiste à passer la référence livre en paramètre au message de création, comme cela est représenté ci-après.

Figure 8-13.
Diagramme de communication de l'opération emprunterLivre *(suite 5)*



Reprenons la liste des post-conditions à vérifier, en indiquant le numéro correspondant du diagramme de communication précédent :

- un prêt p a été créé : 1.2 ;
- l'attribut *date* de p a été positionné à la date du jour : 1.2.1 ;
- l'attribut *dateRetour* de p a été positionné à (la date du jour + deux semaines) : 1.2.2 ;
- p a été lié au livre l dont l'attribut ISBN vaut l'ISBN passé en paramètre : 1.1 et 1.2 ;
- p a été lié à l'adhérent concerné et à la bibliothèque : cela reste à faire.

Il nous faut donc réaliser la dernière post-condition. Quel objet peut-il connaître l'adhérent concerné ? Et d'ailleurs, quand le système a-t-il identifié l'adhérent ?

Rappelons-nous que nous traitons actuellement l'opération système *emprunterLivre*, mais elle a été précédée par *indiquerEmprunteur*, comme le schéma suivant le rappelle.

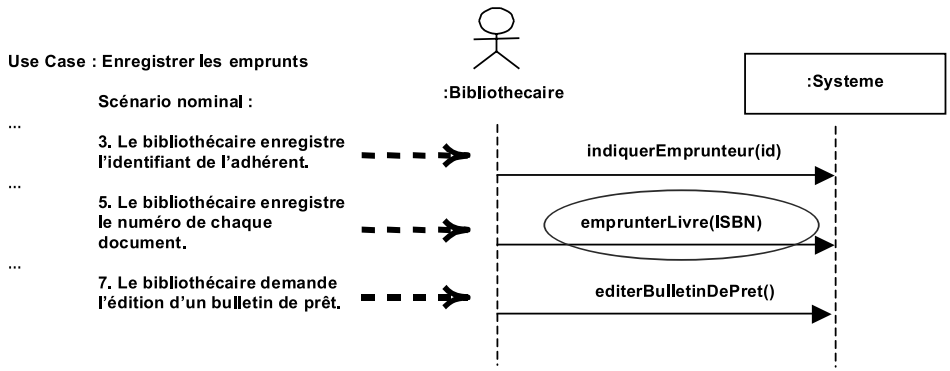
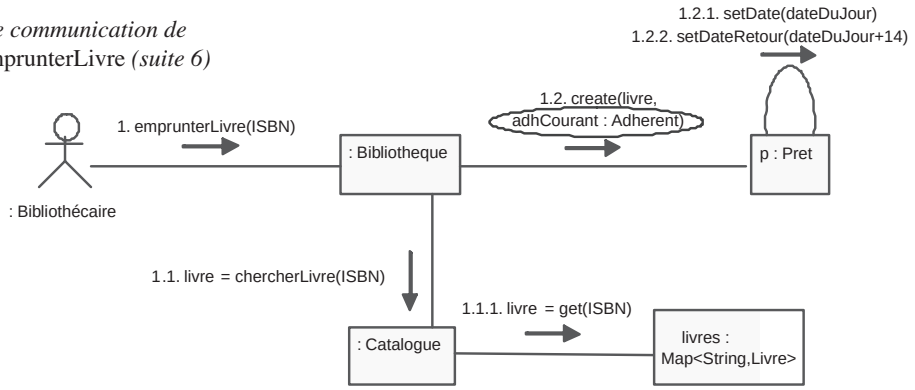


Figure 8-14.
Opérations système du cas d'utilisation Enregistrer les emprunts

Il est donc impératif de penser que, lors de l'opération système *indiquer Emprunteur*, la bibliothèque conserve une référence sur l'adhérent en c ours de traitement. Elle peut donc également passer une référence sur l'adhérent au message de création du prêt p.

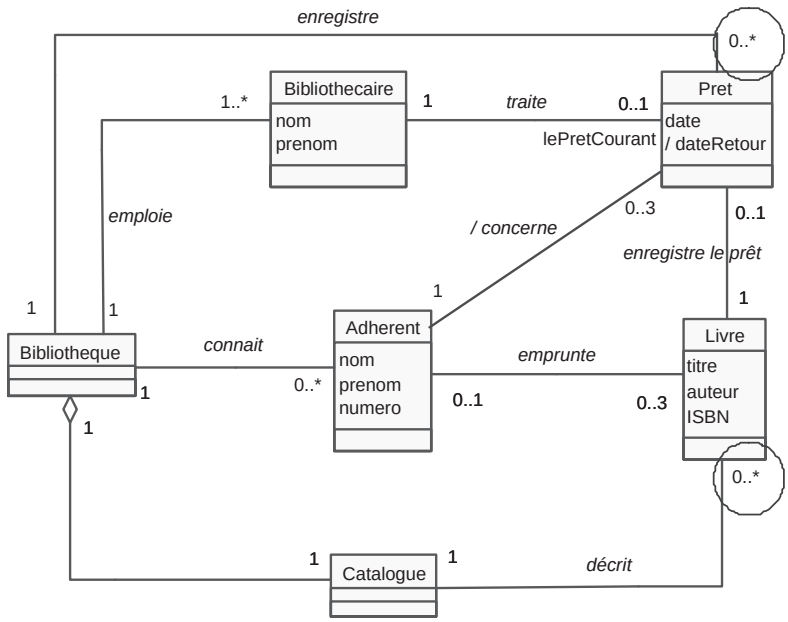
Le diagramme de communication devient maintenant :

Figure 8-15.
Diagramme de communication de l'opération emprunterLivre *(suite 6)*



La dernière post-condition stipule également qu'un lien doit exister entre la bibliothèque et le nouveau prêt p. Comme c'est la bibliothèque qui crée p, le lien existe déjà, au moins de façon transitoire. Cependant, nous pouvons remarquer qu'il existe une association « 1 - * » entre les classes *Bibliotheque* et *Pret*, comme cela est le cas également entre *Catalogue* et *Livre*.

Figure 8-16.
Diagramme de classes d'analyse du S.I. de la bibliothèque



Ainsi, par analogie, si la bibliothèque veut conserver une trace durable des prêts qui ont été créés, il lui faut également gérer une collection, à laquelle elle doit ajouter le nouveau prêt p. Comme au chapitre 7 (voir figure 7-47), nous allons utiliser un message générique *add*() auquel nous passerons en paramètre la référence sur le prêt p.

Les diagrammes de communication et de séquence complets sont montrés aux figures suivantes.

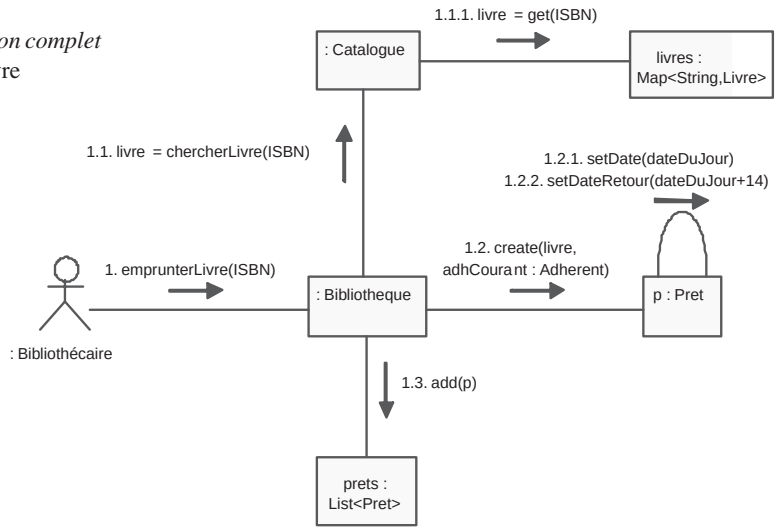


Figure 8-17.
Diagramme de communication complet de l'opération emprunterLivre

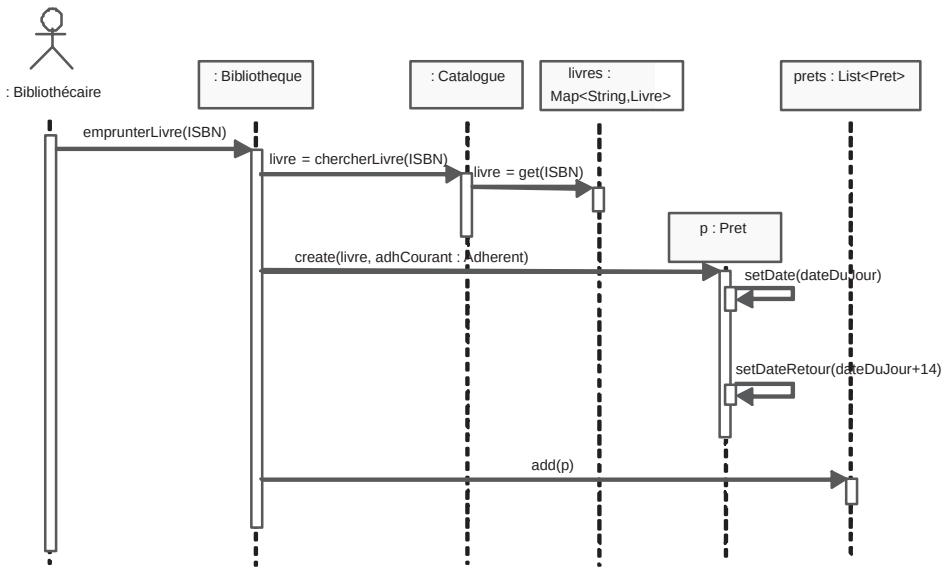


Figure 8-18.
Diagramme de séquence complet de l'opération emprunterLivre

CONCEPTION STATIQUE (DIAGRAMMES DE CLASSES ET DE PACKAGES)

EXERCICE 8-2.
Diagramme de classes de conception

Proposez un diagramme de classes de conception qui prenne en compte les résultats de la question précédente.

Développez le diagramme de classes de conception correspondant…

solution

Par rapport au diagramme de classes d'analyse (figure 8-1), nous pouvons :

- ajouter des méthodes : les opérations système traitées par la classe *Bibliotheque*, mais aussi *chercherLivre* de la classe *Catalogue* ;
- typer les attributs, ainsi que les paramètres et le retour des méthodes ;
- restreindre la navigabilité des associations d'après le sens des messages sur les liens entre objets du diagramme de communication ;
- préciser les noms des rôles du côté navigable des associations et ajouter des qualificatifs ;
- supprimer les classes et associations inutiles d'après les diagrammes de communication ;
- décider de transformer les attributs dérivés en méthodes ou les garder en tant que données membres (ex. : l'attribut dérivé */dateRetour* de *Pret* est devenu la méthode *getDateRetour*) ;
- ajouter les dépendances entre classes suite aux liens temporaires entre objets : *Bibliotheque* dépend de *Livre* car elle récupère une référence sur un objet livre d'après le message 1.1 du diagramme de communication précédent.

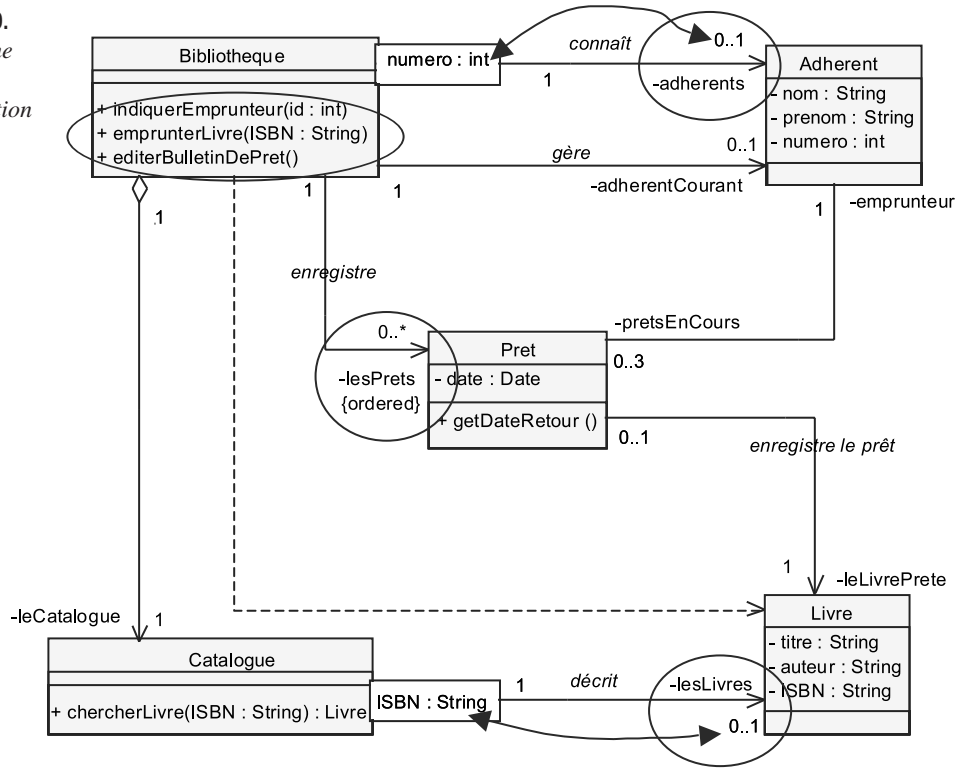


Figure 8-19.
Diagramme de classes de conception

EXERCICE 8-3.
Structuration en packages

Proposez un découpage du diagramme de classes de conception qui minimise les dépendances.

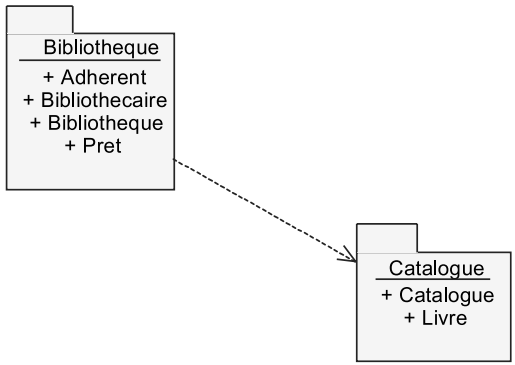
Réalisez un diagramme de packages d'architecture logique...

solution

De manière similaire à l'étude de cas précédente, le catalogue de livres est un candidat naturel à la réutilisation, d'autant qu'aucune relation ne part de l'ensemble de classes *Catalogue* + *Livre*.

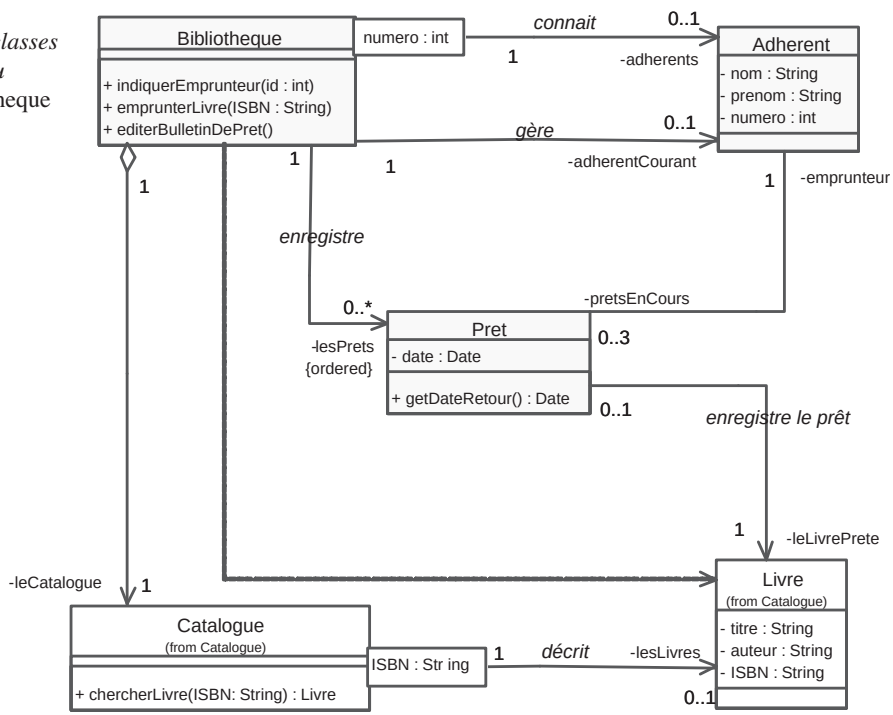
Le diagramme de packages décrivant l'architecture logique est donc donné par le schéma suivant.

Figure 8-20.
Diagramme de packages de conception



Le diagramme de classes du package Bibliothèque est très similaire à celui de la figure 8-19, mais avec l'indication de localisation des classes sous la forme (from XX).

Figure 8-21.
Diagramme de classes de conception du package Bibliotheque



EXERCICE 8-4.
Passage au code Java

Proposez un squelette de code Java pour la classe *Bibliotheque*.

Remplissez le corps de la méthode *emprunterLivre*.

Codez la classe Bibliotheque en Java...

solution

Comme on l'a déjà expliqué à l'étape 15 du chapitre 7, le passage du diagramme de classes au squelette de code Java et du diagramme de communication au corps des méthodes est assez direct (notez juste l'utilisation des « *generics* » 1.5). Ainsi, on obtient le fragment suivant du fichier « Bibliotheque.java » :

Figure 8-22.
Squelette de code Java 1.5 de la classe Bibliothèque

```
package bibliotheque ;

import java.util.*;
import catalogue.*;

public class Bibliotheque

{
   private Catalogue leCatalogue;
   private Map<Integer,Adherent> adherents =
       new HashMap<Integer,Adherent>();
   private List<Pret> lesPrets = new ArrayList<Pret>();
   private Adherent adherentCourant;

   public Bibliotheque ()
   {
     …
   }

   public void indiquerEmprunteur(int id)
   {
     …
   }

   public void emprunterLivre(String ISBN)
   {
     Livre livre = leCatalogue.chercherLivre(ISBN);
     Pret p = new Pret(livre, adherentCourant);
     lesPrets.add(p);
   }

   public void editerBulletinDePret()
   {
     …
   }
}
```

EXERCICE 8-5.
Passage au code C#

Proposez un squelette de code C# pour la classe *Bibliotheque*.

Remplissez le corps de la méthode *emprunterLivre*.

Codez la classe BIBLIOTHEQUE en C#...

solution

Comme pour l'exercice 8-4, on obtient facilement le fragment suivant du fichier « Bibliotheque.cs » :

Figure 8-23.
Squelette de code C# de la classe Bibliothèque

```
namespace Bibliotheque  ;

{
  using System;
  using Catalogue.*;

  public class Bibliotheque

  {
     private Catalogue LeCatalogue;
     private HashTable Adherents = new HashTable();
     private ArrayList LesPrets = new ArrayList();
     private Adherent AdherentCourant;

     public Bibliotheque ()
     {
       …
     }

     public void IndiquerEmprunteur(int Id)
     {
       …
     }

     public void EmprunterLivre(string ISBN )
     {
       Livre livre = LeCatalogue.ChercherLivre(ISBN);
       Pret p = new Pret(Livre, AdherentCourant);
       LesPrets.add(p);
     }

     public void EditerBulletinDePret()
     {
       …
     }

} }
```

ANALYSE ET CONCEPTION DU JEU DE DÉMINEUR

L'objectif de cette étude de cas est de 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.

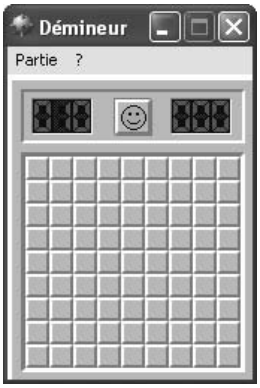


Figure 8-24.
Copie d'écran du jeu de démineur

Le jeu est composé d'un plateau rectangulaire, d'un chronomètre et d'un compteur de mines. Le 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 pas 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.

MODÉLISATION DES EXIGENCES

EXERCICE 8-6.
Diagramme de cas d'utilisation

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

solution

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. N'oublions pas également le cas d'utilisation « Consulter l'aide en ligne », qui est présent dans la majorité des applications informatiques interactives !

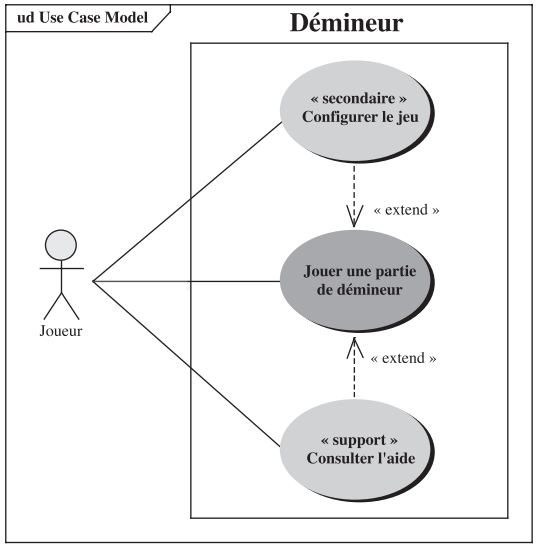


Figure 8-25.
Diagramme de cas d'utilisation du démineur

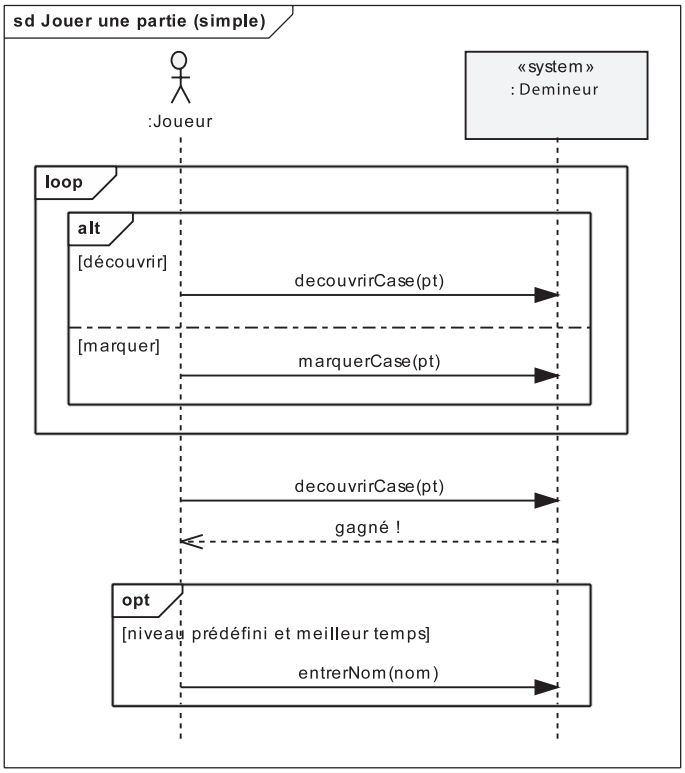
EXERCICE 8-7.
Diagramme de séquence système

Développez un diagramme de séquence système pour le cas d'utilisation « Jouer une partie de démineur ».

solution

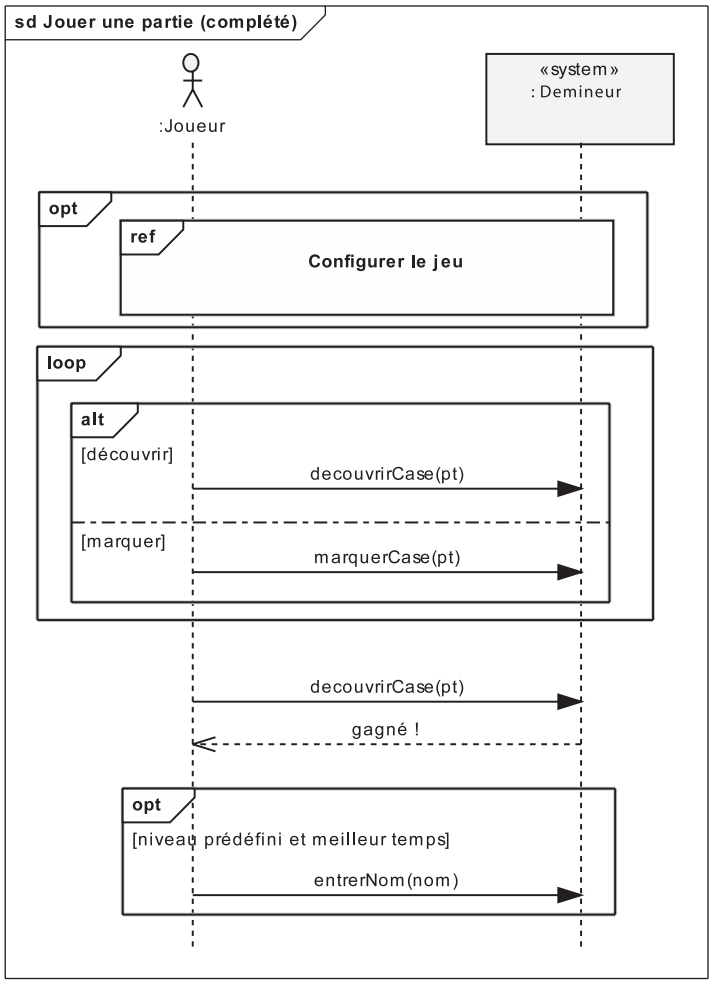
Le joueur va passer son temps à découvrir ou marquer des cases. Dans le cas nominal, il va finir par gagner et entrer dans les meilleurs scores (s'il jouait à un niveau prédéfini). On peut donc représenter tout cela par une grande boucle (fragment `loop`) dans laquelle les opérations système « marquerCase » et « découvrirCase » peuvent arriver dans un ordre quelconque (d'où l'opérateur `alt`).

Figure 8-26.
Diagramme de séquence système préliminaire



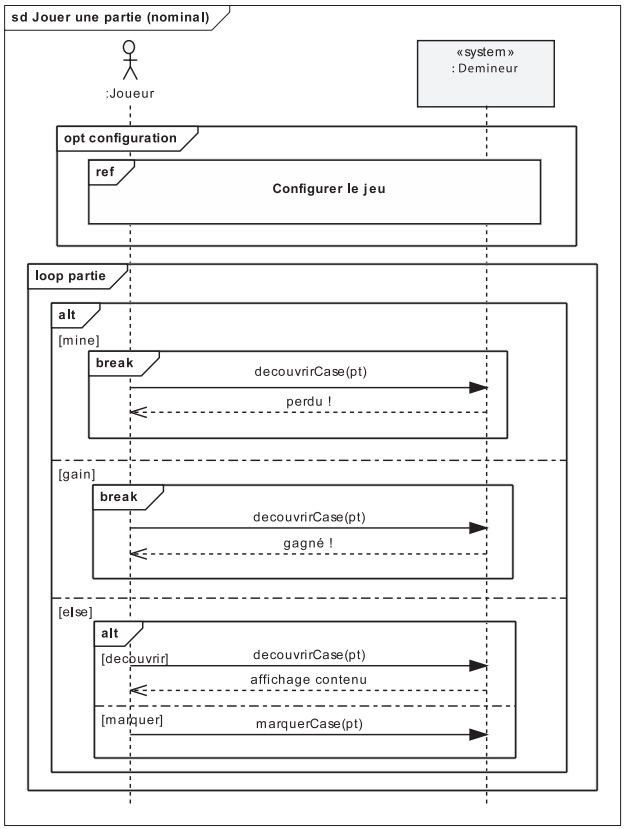
Nous pouvons ajouter la configuration du jeu comme une étape optionnelle avant de commencer à jouer. La notion de référence vers une occurrence d'interaction est tout à fait applicable.

Figure 8-27.
Diagramme de séquence système complété



Une solution plus sophistiquée consiste à représenter dans la boucle de jeu les trois possibilités : perte, gain ou cas normal. Les cas de perte ou gain arrêtent la partie, ce qui peut se représenter par l'opérateur prédéfini `break`.

Figure 8-28.
Diagramme de séquence système plus sophistiqué



ANALYSE ORIENTÉE OBJET

EXERCICE 8-8.
Modèle d'analyse

Réalisez un diagramme de classes d'analyse ainsi qu'un diagramme d'états si nécessaire.

solution

La principale difficulté consiste à bien représenter la double variabilité au niveau des cases :

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

Le diagramme de classes présenté sur le schéma suivant doit donc être complété par un diagramme d'états de la classe *Case*.

Notez également qu'une case possède exactement 3 (coin), 5 (bord) ou 8 voisines[1]. Certains attributs sont calculables donc dérivés, et les coordonnées d'une case jouent le rôle de qualifieur (ou qualificatif).

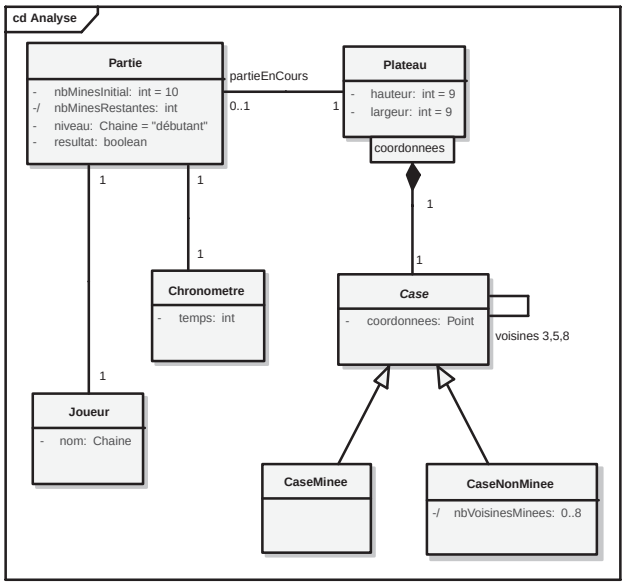


Figure 8-29.
Diagramme de classes d'analyse

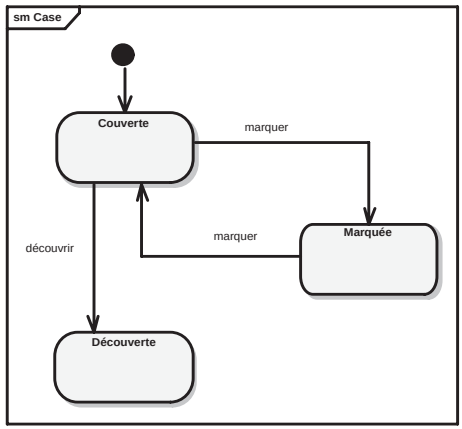


Figure 8-30.
Diagramme d'états de la classe Case

1. Nous avons continué à utiliser la notation 3,5,8 pour indiquer les multiplicités discrètes des nombres de voisines d'une case, bien que cette notation ait été abandonnée dans UML 2, car jugée trop complexe…

CONCEPTION ORIENTÉE OBJET AVEC LES DESIGN PATTERNS

EXERCICE 8-9.
Conception de l'opération « marquerCase(pt) »

Réalisez les diagrammes de séquence de conception objet pour l'opération système « marquerCase (pt) ».

Expliquez vos choix d'utilisation de design patterns si nécessaire.

solution

Commençons par décider quel objet va traiter les opérations système (voir exercice 8-1). Dans notre exemple, l'objet contrôleur le plus naturel est « Partie ». L'objet « 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 va tout d'abord récupérer une référence sur la case dont les coordonnées correspondent au paramètre « pt » de l'opération système. Pour cela, il fait appel à une collection triée de cases (une *Map*) pour trouver l'instance « c » correspondant au paramètre « pt ». Il va ensuite déléguer à celle-ci le traitement du marquage proprement dit.

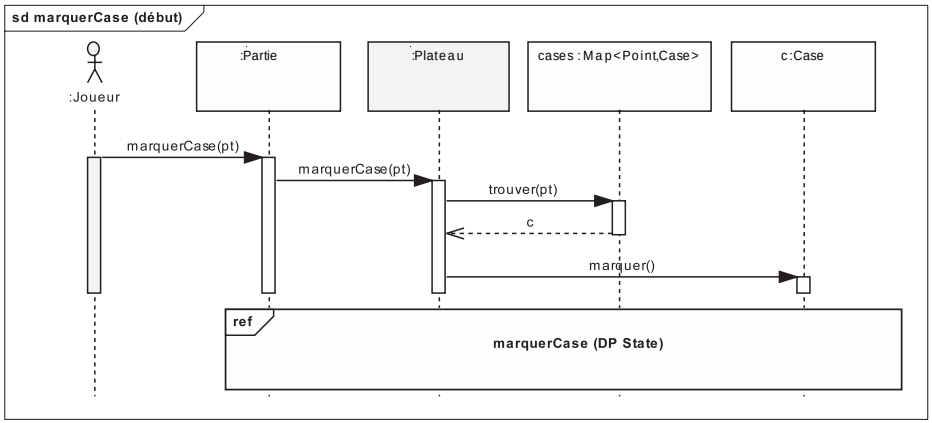


Figure 8-31.
Début du diagramme de séquence de conception pour l'opération système marquerCase(pt)

Néanmoins, le traitement du marquage dépend de l'état de la case. Si l'on veut éviter une logique conditionnelle complexe et peu évolutive, il est intéressant de déporter le comportement variable dans de nouveaux objets et d'utiliser pour cela le design pattern *State*.

À retenir

DESIGN PATTERN STATE (ÉTAT)

Problème à résoudre : le comportement d'un objet dépend de son état. Cependant, nous n'acceptons pas de câbler en dur cette dynamique dans sa classe…

Solution : utiliser le polymorphisme, mais pas directement par des sous-classes de la classe concernée. En effet, un objet ne change pas de classe au cours de sa vie. Il faut donc utiliser le principe de délégation, en créant une classe artificielle (pure fabrication) qui récupère le comportement variable. Cette classe artificielle sera elle-même spécialisée en autant de sous-classes que l'objet possède d'états différents.

La copie d'écran suivante montre comment un outil de modélisation UML peut intégrer les design patterns et permettre de créer facilement les classes, attributs, associations et opérations concernées. Dans cet exemple, l'outil Borland/Together a ainsi créé les éléments UML génériques, mais aussi le code Java correspondant au design pattern *State* (une interface *Etat* et des classes concrètes par sous-état, en recopiant toutes les méthodes de la classe de départ appelée *Context*, sans oublier de lui ajouter une méthode setEtatCourant pour gérer la référence au bon état).

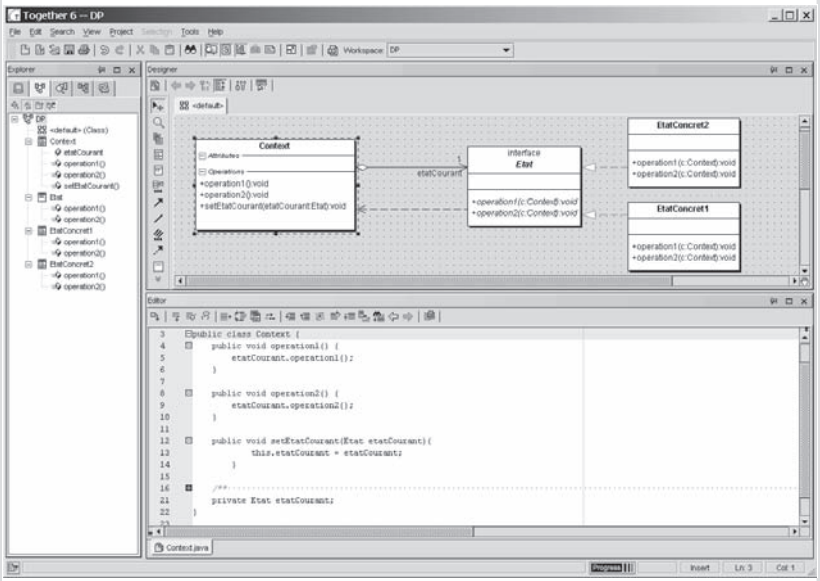


Figure 8-32.
Copie d'écran de l'outil Borland/Together en pleine action sur le DP State

La case va donc déléguer à son objet « étatCourant » le comportement variable de marquage, conformément au schéma suivant.

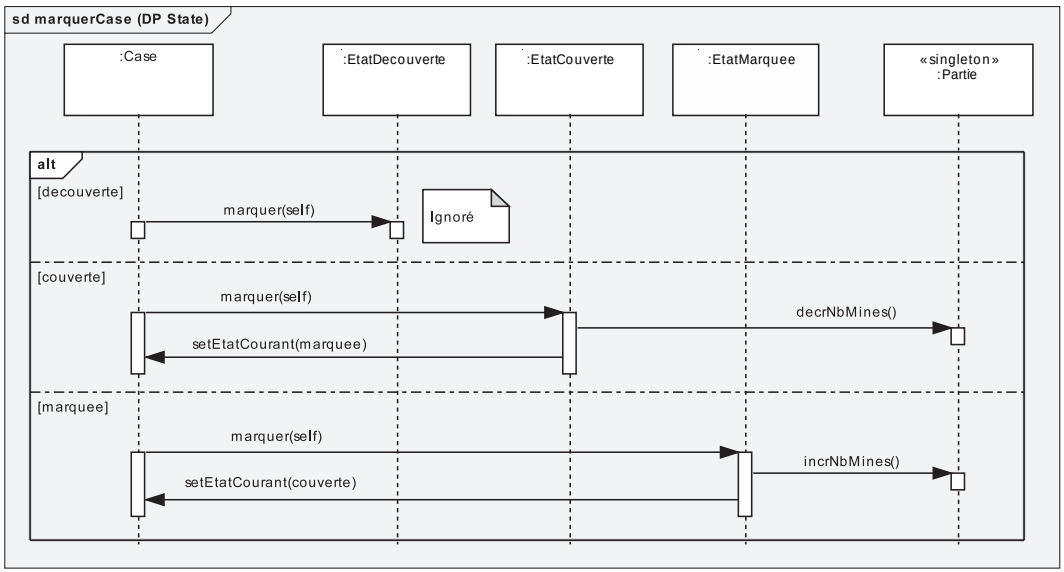


Figure 8-33.
Suite du diagramme de séquence de conception pour l'opération système marquerCase(pt) *: utilisation du DP* State

Remarquons que de nombreuses classes différentes vont avoir besoin d'un accès au contrôleur *Partie*, pour décrémenter ou incrémenter le compteur de mines comme indiqué sur le schéma précédent, mais aussi pour signifier la fin de la partie (dans le cas de la découverte d'une case minée). Or, la classe *Partie* ne doit avoir qu'une seule instance à la fois. Assurer qu'une classe ne sera instanciée qu'une seule fois et donner un accès global à cette instance unique, tel est l'objectif du design pattern *Singleton*.

À retenir

DESIGN PATTERN *SINGLETON*

Problème à résoudre : comment garantir l'existence d'une seule instance d'une classe donnée et fournir un point d'accès global à cette instance ?

Solution : sauvegarder l'objet singleton dans une variable de classe (statique) et fournir une méthode de classe retournant l'instance unique, en la créant à la première requête. Le constructeur est alors déclaré privé (ou mieux : protégé).

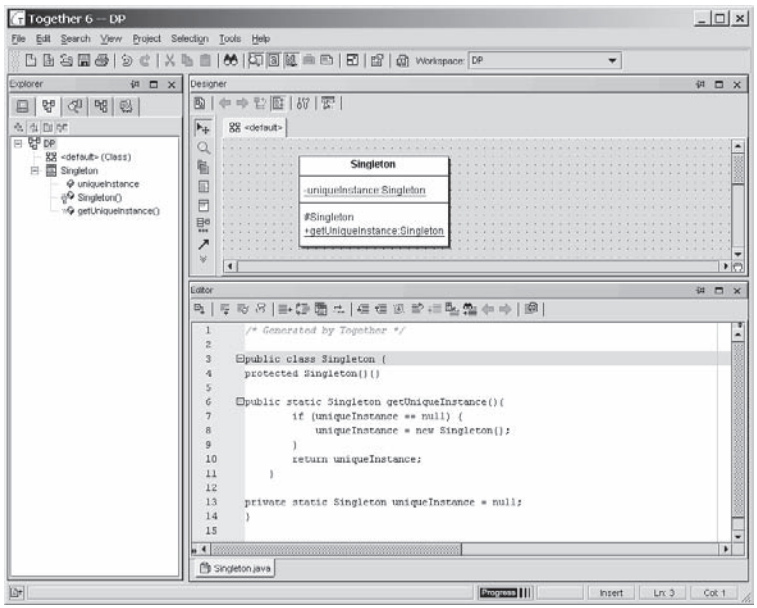


Figure 8-34.
Copie d'écran de l'outil Borland/Together en pleine action sur le DP Singleton

EXERCICE 8-10.
Conception de l'opération « decouvrirCase(pt) »

Réalisez les diagrammes de séquence de conception objet pour l'opération système « decouvrirCase (pt) ».

solution

Le début de l'interaction est très similaire à celle de l'opération système « marquerCase ». L'objet « Partie » va déléguer le travail à l'expert des cases,

soit le « Plateau ». Celui-ci va tout d'abord récupérer une référence sur la case dont les coordonnées correspondent au paramètre « pt » de l'opération système. Il va ensuite déléguer à celle-ci le traitement du message « decouvrir », qui dépend de son état.

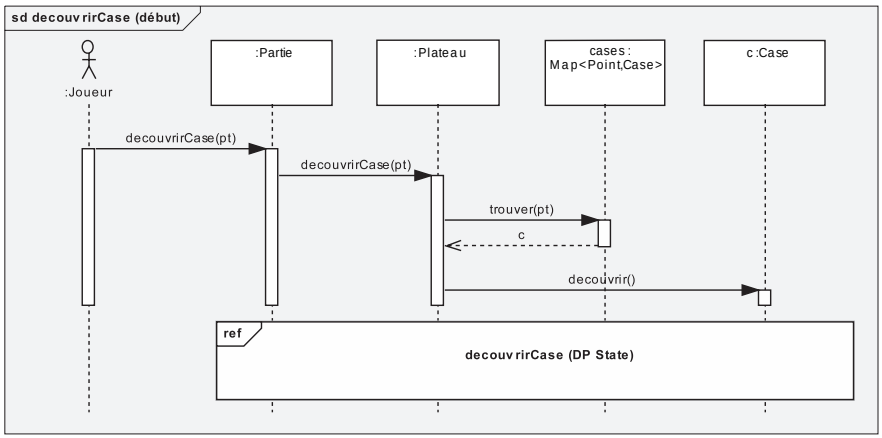


Figure 8-35. *Début du diagramme de séquence de conception pour l'opération système* decouvrirCase(pt)

Nous allons utiliser également ici le design pattern *State*. Toutefois cette fois-ci, la situation se complique : si la case était dans l'état « Couverte », elle passe à l'état « Découverte », mais le traitement effectué dépend maintenant du type de case : minée ou pas. La méthode « dévoiler », appelée par l'état « Couverte » sur la case est elle-même polymorphe !

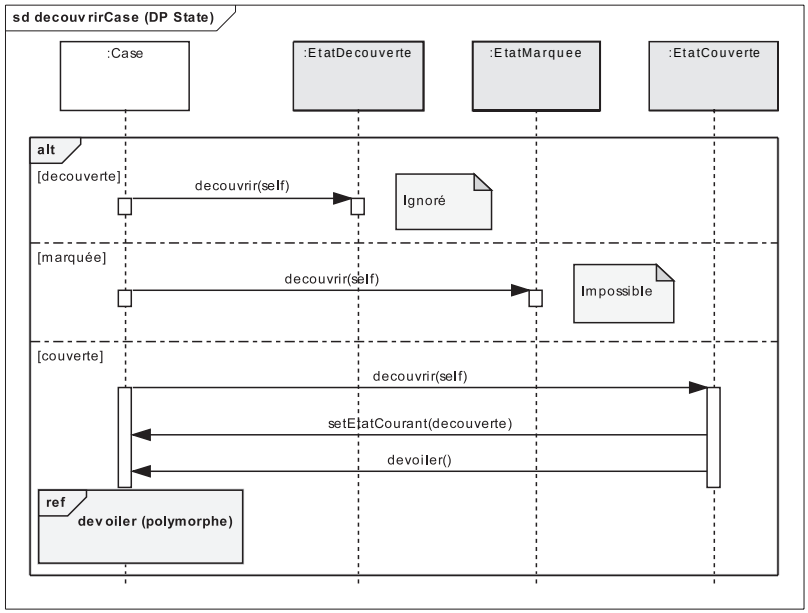


Figure 8-36. *Suite du diagramme de séquence de conception pour l'opération système* decouvrirCase(pt) *: application du DP* State

En effet, dévoiler interrompt brutalement la partie sur une case minée. Il faut alors arrêter le chronomètre, découvrir toutes les cases minées et signaler toutes les cases marquées à tort avec un X. S'il s'agit d'une case numérotée, il faut vérifier si la partie n'est pas gagnée (toutes les cases minées sont découvertes). Enfin, s'il s'agit d'une case vide, il faut propager le message « découvrir » à toutes les voisines[2].

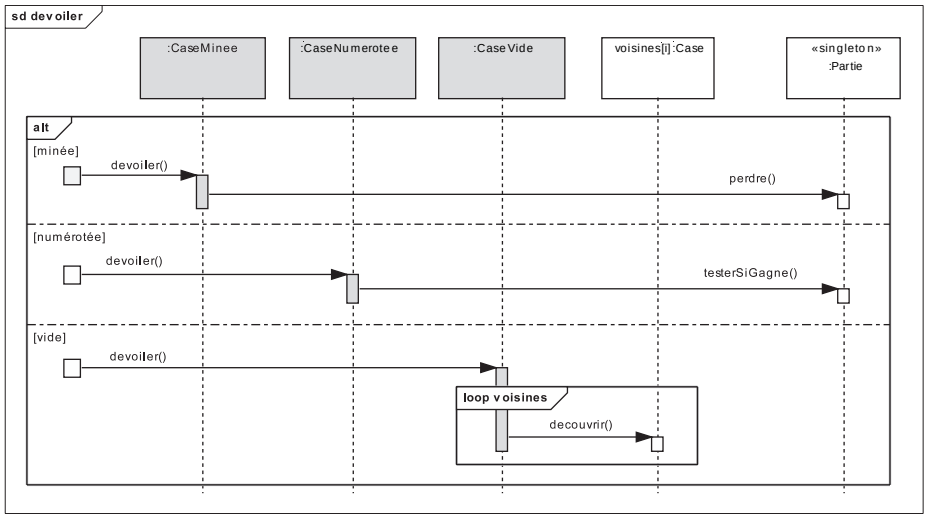


Figure 8-37.
Fin du diagramme de séquence[3] *de conception pour l'opération système* decouvrirCase(pt) : *polymorphisme de* dévoiler()

EXERCICE 8-11.
Diagramme de classes de conception

Dessinez le diagramme de classes de conception objet en indiquant les design patterns utilisés.

solution

En ajoutant les opérations dans les bonnes classes d'après les diagrammes de séquence précédents, on obtient assez facilement le schéma statique suivant.

2. Notez le nom de ligne de vie que nous avons employé pour les cases voisines : « voisines[i] : Case ». Il s'agit d'une instance de la classe *Case*, sélectionnée dans la collection *voisines*.
3. Tous les diagrammes de cette étude de cas (sauf les copies d'écran sur les design patterns) ont été réalisés avec l'excellent outil Enterprise Architect de Sparx Systems (http://sparxsystems.com.au).

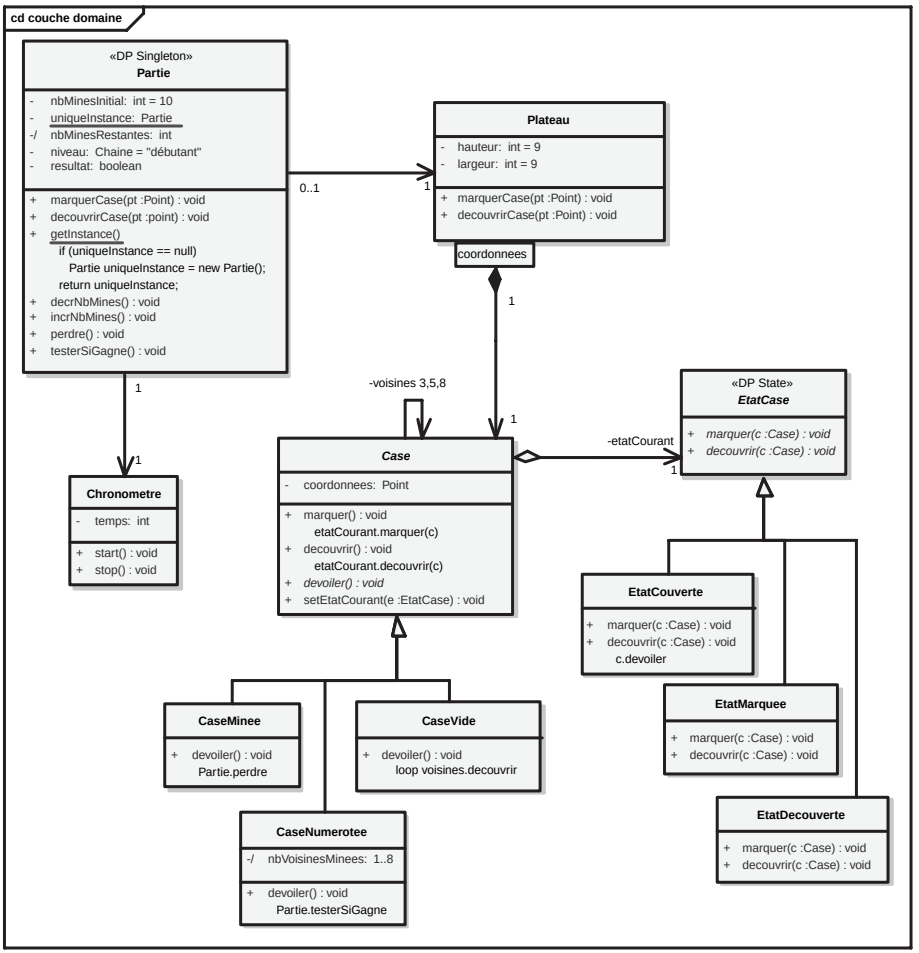


Figure 8-38. *Diagramme de classes de conception du jeu de démineur*

On notera le double arbre d'héritage que le DP *State* permet d'implémenter de façon modulaire et évolutive.

CONSEILS MÉTHODOLOGIQUES

ARCHITECTURE EN COUCHES

- Séparez votre application en couches. Le principal intérêt que revêt la séparation en trois couches (3-tiers) est d'isoler la logique métier des classes de présentation (IHM), ainsi que d'interdire un accès direct aux données stockées par ces classes de présentation. Le souci premier est de répondre au critère d'évolutivité : pouvoir modifier l'interface de l'application sans devoir modifier les règles métier, et pouvoir changer de mécanisme de stockage sans avoir à retoucher l'interface, ni les règles métier.

- Pour améliorer la modularité, introduisez un objet artificiel, appelé « contrôleur », entre les objets graphiques et les objets métier. Cet objet de conception connaît l'interface des objets de la couche métier et joue le rôle de « façade » vis-à-vis de la couche présentation.
- Dans les cas simples, le contrôleur peut être un objet d'une classe d'analyse existante :
  - soit un objet représentant le système entier ou l'organisation elle-même ;
  - soit un objet représentant un rôle qui aurait réalisé l'opération système.
- Décrivez votre architecture en couches par un diagramme statique qui ne montre que des packages et leurs dépendances. Vous pouvez utiliser le stéréotype « `layer` » pour distinguer les packages qui représentent les couches.
- N'oubliez pas que le processus d'analyse/conception est fondamentalement itératif. L'architecture préliminaire pourra être affinée ou modifiée (principalement, au niveau des partitions à l'intérieur de chaque couche) par le travail de conception qui va suivre la première découpe d'analyse.

CONTRAT D'OPÉRATION

- Utilisez les contrats d'opérations : ils permettent ainsi de faire le lien entre le point de vue fonctionnel/dynamique des cas d'utilisation et le point de vue statique d'analyse. Un contrat d'opérations décrit les changements d'état du système quand une opération système est effectuée. Ces modifications sont exprimées en termes de « post-conditions » qui détaillent le nouvel état du système après l'exécution de l'opération. Les principales post-conditions concernent la création (ou la destruction) d'objets et de liens issus du modèle statique d'analyse, ainsi que la modification de valeurs d'attributs.
- Utilisez le plan type de description textuelle de contrat d'opérations donné ci-après :
  - nom
  - responsabilités
  - références
  - pré-conditions
  - post-conditions
  - exceptions (optionnel)
  - notes (optionnel)
- Concevez les opérations système en respectant leur chronologie.
- N'oubliez pas que les post-conditions ne représentent que le nouvel état du système à la fin de l'exécution de l'opération système. Elles ne sont absolument pas ordonnées : c'est le rôle du concepteur de choisir quel objet doit réaliser chaque action, et dans quel ordre.

- Pour détailler visuellement une architecture logique, il suffit de recenser toutes les classes utilisées dans les différents diagrammes, et de les représenter graphiquement à l'intérieur du package adéquat dans un diagramme de packages.

DE L'ANALYSE À LA CONCEPTION

- Pour passer de l'analyse à la conception, utilisez les trois stéréotypes de Jacobson qui permettent de montrer graphiquement comment un message émis par un acteur traverse les couches présentation, application et métier :
  - `<<boundary>>` : classes qui servent à modéliser les interactions entre le système et ses acteurs ;
  - `<<control>>` : classes utilisées pour représenter la coordination, l'enchaînement et le contrôle d'autres objets – elles sont en général reliées à un cas d'utilisation particulier ;
  - `<<entity>>` : classes qui servent à modéliser des informations durables et souvent persistantes.
- Partez des opérations système pour initialiser votre étude dynamique sous la forme de diagrammes de communication ou de séquence.

DIAGRAMMES D'INTERACTION DE CONCEPTION

- Sur les diagrammes de communication, utilisez la numérotation décimale qui permet de montrer l'imbrication des messages, d'une façon comparable à la représentation des « focus of control » sur le diagramme de séquence.
- Le diagramme de séquence devient de moins en moins lisible au fur et à mesure que l'on ajoute des objets. C'est pour cette raison simple que le diagramme de communication est utile en conception : il permet de disposer les objets dans les deux dimensions afin d'améliorer la lisibilité du schéma. Le diagramme de communication présente un autre avantage sur le diagramme de séquence : il permet aussi de représenter les relations structurelles entre les objets. Par contre, UML2 permet de représenter plus facilement des boucles et des alternatives sur le diagramme de séquence (fragments *loop*, *alt*, *opt*, etc.)
- Une idée intéressante pour améliorer la lisibilité du diagramme de communication consiste à le découper en deux en prenant l'objet contrôleur comme charnière :
  - une première partie afin de spécifier la cinématique de l'interface homme-machine avec les acteurs, les objets `<<boundary>>` et l'objet `<<control>>` ;
  - une seconde partie afin de spécifier la dynamique des couches applicatives et métier avec l'objet `<<control>>` et les objets `<<entity>>`.
- Dans un premier temps, travaillez sur les interactions entre objets « métier », ensuite traitez le problème plus technique de l'initialisation du système informatique. Cela permet de s'assurer que les bonnes décisions d'affectation de responsabilités aux

objets dans le contexte des interactions métier contraignent bien l'initialisation, et pas le contraire.

DIAGRAMMES DE CLASSES DE CONCEPTION

- Les diagrammes d'interaction vont permettre d'élaborer des diagrammes de classes de conception, et ce en ajoutant principalement les informations suivantes aux classes issues du modèle d'analyse :
  - les opérations : un message ne peut être reçu par un objet que si sa classe a déclaré l'opération publique correspondante ;
  - la navigabilité des associations ou les dépendances entre classes, suivant que les liens entre objets sont durables ou temporaires, et en fonction du sens de circulation des messages.
- Attention : un lien durable entre objets va donner lieu à une association navigable entre les classes correspondantes ; un lien temporaire (par paramètre : « `parameter` », ou variable locale : « `local` ») va donner lieu à une simple relation de dépendance. N'ajoutez pas les classes qui correspondent aux collections dans le diagramme de classes de conception, de façon à rester le plus longtemps possible indépendant du langage de programmation cible.
- Par rapport aux messages des diagrammes d'interaction, ne faites pas apparaître dans les diagrammes de classes de conception :
  - les opérations de création (message *create*) ;
  - les opérations génériques sur les classes conteneurs (*add*(), etc.) ;
  - les opérations d'accès aux attributs.
- Vous pouvez utiliser les stéréotypes « `parameter` » et « `local` »[4] sur les dépendances entre classes, pour refléter le type de lien temporaire qui existe entre les objets correspondants dans le diagramme d'interaction.
- Conserver un couplage faible est un principe qu'il faut bien avoir à l'esprit pour toutes les décisions de conception ; c'est un objectif sous-jacent à évaluer d'une façon continue. En effet, en y pourvoyant, on obtient en général une application plus évolutive et plus facile à maintenir.
- N'oubliez pas de faire aussi figurer dans le diagramme de classes celles qui n'appartiennent pas au package courant. En effet, il est important de montrer leurs relations avec des classes du package courant pour justifier ensuite le sens des dépendances entre les packages englobants. Précisément, il ne faut représenter que les associations navigables, les dépendances ou les généralisations qui pointent vers les classes externes à celles du package concerné.

4. Bien que ces stéréotypes semblent avoir disparu du standard dans UML 2, nous continuerons à en préconiser l'utilisation.

PASSAGE AU CODE OBJET

- Les modèles UML de conception permettent de produire aisément du code dans un langage de programmation objet tel que Java, C# ou autre :
  - les diagrammes de classes permettent de décrire le squelette du code, à savoir toutes les déclarations ;
  - les diagrammes d'interaction permettent d'écrire le corps des méthodes, en particulier la séquence d'appels de méthodes sur les objets qui interagissent.
- En première approche :
  - la classe UML devient une classe Java ou C# ;
  - les attributs UML deviennent des variables d'instances Java ou C# ;
  - les méthodes qui permettent l'accès en lecture (*get*) et en écriture (*set*) aux attributs, pour respecter le principe d'encapsulation, sont implicites (en C#, on utilisera la notion de *property*) ;
  - les opérations UML deviennent des méthodes Java ou C# ;
  - les rôles navigables produisent des variables d'instances, tout comme les attributs, mais avec un type utilisateur au lieu d'un type simple ;
  - le constructeur par défaut est implicite.
- N'oubliez pas la directive d'importation (ou d'utilisation) pour les relations avec les classes qui appartiennent à d'autres packages, ainsi que pour les classes de base Java ou C#.
- Comment traduire les associations navigables de multiplicité « * » ? Utilisez un attribut de type collection de références d'objets au lieu d'une simple référence sur un objet. La difficulté consiste à choisir la bonne collection parmi les très nombreuses classes de base que proposent Java et C#. Bien qu'il soit possible de créer des tableaux d'objets, ce n'est pas forcément la bonne solution. En la matière, on préfère plutôt recourir à des collections, parmi lesquelles les plus utilisées sont :
  - En Java : *ArrayList* (anciennement *Vector*) et *HashMap* (anciennement *HashTable*). Utilisez *ArrayList* si vous devez respecter un ordre et récupérer les objets à partir d'un indice entier ; utilisez *HashMap* si vous souhaitez récupérer les objets à partir d'une clé arbitraire.
  - En C# : *ArrayList, SortedList* et *HashTable*. Utilisez *ArrayList* si vous devez respecter un ordre et récupérer les objets à partir d'un indice entier ; utilisez *HashTable* ou *SortedList* si vous souhaitez récupérer les objets à partir d'une clé arbitraire.
  - N'hésitez pas à utiliser les collections typées (*generics*) si votre langage le permet (Java 1.5, C# 2.0, etc.).

DIAGRAMME DE DÉPLOIEMENT

- Décrivez l'implantation physique de votre application grâce au diagramme de déploiement.