Travaux dirigés 1
Modélisation graphique

La programmation orienté objet consiste à concevoir un programme en termes de briques logicielles appelées *objets*. Concrètement, un objet est une structure de données (*attributs*) qui répond à un ensemble de messages (*méthodes*) dont la syntaxe est définie par un type (*classe*). Ils sont fondés sur le principe d'encapsulation : c'est seulement à travers l'échange de messages que les objets communiquent entre eux. Par conséquent, les fonctionnalités d'un programme émergent de l'interaction entre les objets qui le constituent.

La difficulté de l'approche orientée objet se trouve dans le travail conceptuel nécessaire pour définir les classes, les relations entre celles-ci et les interactions parmi les objets correspondants. C'est pourquoi il ne convient pas de se lancer tête baissée dans l'écriture du code pour réaliser un logiciel. Il faut d'abord organiser ses idées, les documenter et ensuite planifier le travail en plusieurs étapes. Cette démarche antérieure à la programmation est ce qu'on appelle *méthode de développement*.

Le langage de modélisation unifié (UML) est une notation graphique standard conçue pour représenter et communiquer les divers aspects du développement. Bien qu'il est impossible de donner une représentation graphique complète d'un logiciel, on peut construire des vues partielles sur un tel système, dont la conjonction donnera une vision d'ensemble. Par conséquent, UML comporte plusieurs diagrammes qui permettent de représenter les différents concepts d'un système. Ces diagrammes, d'une utilité variable selon les cas, ne sont pas nécessairement tous produits à l'occasion d'une modélisation. Les plus utiles sont :

1. le **diagramme de classes**, qui permet de représenter l'architecture interne du système en termes de classes et leurs relations ;
2. le **diagramme d'objets**, qui permet de représenter les objets existants dans le système à un instant précis ;
3. le **diagramme de séquence**, qui permet de représenter le comportement du système en termes d'interactions entre objets.

L'objectif du présent TD est d'approfondir les diagrammes susmentionnés, en se focalisant sur leur équivalence avec un langage orienté objet tel que C#. Il est organisé en trois sections, dont les deux premières proposent des petits exercices de rétro-ingénierie qui visent à faire comprendre l'équivalence entre UML et C#. En revanche, la troisième section porte sur un mini-projet de pro-ingénierie basé sur le jeu « démineur ».

1.1 Diagramme de classes

Le diagramme de classes est un schéma qui présente les classes d'un système ainsi que les différentes relations entre celles-ci. Il s'agit d'une vue statique, car les aspects temporels et dynamiques n'y sont pas pris en compte. Ce diagramme propose une représentation abstraite du paradigme objet, afin d'aboutir à une notation indépendante de toute langage de programmation. Il est donc essentiel de connaitre les règles pour passer d'une représentation à l'autre. Plus précisément, on parle de *pro-ingénierie* lorsqu'on traduit l'UML en codes, et de *rétro-ingénierie* lorsqu'on va dans le sens inverse.

1.1.1 Classes et objets

Definition de classe

Une *classe* est la description d'une collection d'objets ayant en commun le type, les attributs et les méthodes. Elle est représentée en UML par un rectangle contenant le nom de la classe, ainsi que la liste des attributs et des méthodes. Les noms de ces derniers sont précédés par un signe qui indique leur visibilité : publique (+), privée (-) ou protégée (#). Voici un exemple.

```
class Person {
   string name;
   string surname;
   DateTime date;

   public Person(string n,string s,DateTime d){
      name = n;
      surname = s;
      date = d;
   }

   public int Age() {
      return DateTime.Today.Year - date.Year;
   }
}
```

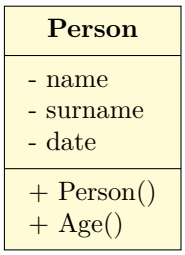
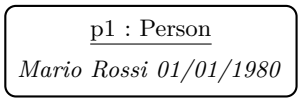


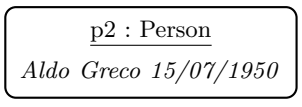
Diagramme d'objets

Un *objet* est l'instance d'une classe. Il est représenté en UML par un rectangle indiquant, tout souligné, le nom de l'objet suivi par « : » et le nom de la classe correspondante. Un *diagramme d'objets* est une photographie à un instant précis des objets existants dans le système. Il accompagne souvent un diagramme de classes à des fins de compréhension. Voici un exemple.

```
DateTime bday1 = new DateTime(1980,1,1);
Person p1 = new Person("Mario","Rossi",bday1);
```




```
DateTime bday2 = new DateTime(1950,7,15);
Person p2 = new Person("Aldo","Greco",bday2);
```



1.1.2 Relation de dépendance

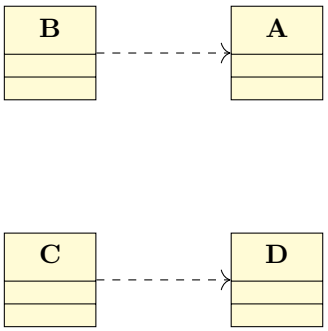
La philosophie du paradigme objet est que chaque classe doit être responsable de taches précises et limités. Pour remplir ses responsabilités, une classe n'agit pas toutes seules : elle entretient des relations avec d'autres classes afin de pouvoir utiliser les services de celles-ci. Les *relations entre classes* sont un aspect essentiel de l'approche objet, car elles définissent la manière dont les objets d'un système sont reliés. UML prévoit quatre types de relation : la dépendance, l'association, la généralisation et la réalisation.

Relations entre classes

La forme plus simple de relation est la *dépendance*, qui se manifeste lorsqu'une classe utilise une autre classe pour définir/créer des variables locales, déclarer des arguments de fonction, ou appeler ses méthodes statiques. En UML, la dépendance est notée par une flèche en pointillé (⇢).

Dépendance

```
class B
{
    public void Method()
    {
        A a = new A();
        ...
    }
}
class C
{
    public void Method(D d) {
        ...
    }
}
```



Exercice 1.1

Les exemples ci-dessous montrent une relation entre les classes `Player` and `Die`. Dans quel cas, la relation entre celles-ci est une simple *dépendance* ?

a)

```
class Die
{
    Random rnd= new Random();

    public int Roll() {
        return rnd.Next(1,7);
    }
}

class Player
{
    public void Play(Die die)
    {
        int value= die.Roll();
        ...
    }
}
```

b)

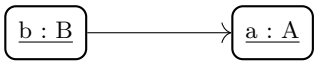
```
class Die {
    Random rnd= new Random();

    public int Roll() {
        return rnd.Next(1,7);
    }
}

class Player {
    public Die die;

    public void Play()
    {
        int value= die.Roll();
        ...
    }
}
```

1.1.3 Relation d'association

Association unidirectionnelle

L'*association unidirectionnelle* se matérialise quand une classe sert de type à un attribut d'une autre classe, entrainant un lien durable entre les objets de celles-ci. L'association unidirectionnelle est représentée en UML par une flèche pleine (→) orientée vers la classe qui sert de type à l'autre.

```
class B {
   A a;

   public void Link(A a) {
      this.a = a;
   }
}

static void Main() {
   A a = new A();
   B b = new B();
   b.Link(a);
}
```



Association bidirectionnelle

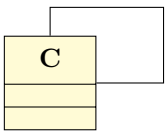
Une association est dite *bidirectionnelle* si chacune des deux classes sert de type à l'autre. Le lien entre leurs objets est alors navigable dans les deux sens, en raison du fait que chaque objet possède une référence vers l'autre. Cette relation est notée en UML par un trait plein (–) reliant les deux classes.

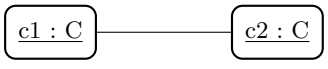
```
class A {
   B b;

   public void Link(B b) {
      this.b = b;
   }
}

static void Main() {
   A a = new A();
   B b = new B();
   b.Link(a);
   a.Link(b);
}
```

Association réflexive

Quand les deux extrémités d'une association (bidirectionnelle) pointent vers la même classe, la relation est dite *réflexive*. Elle indique qu'un objet peut être relié à d'autres objets de sa propre classe, y compris lui-même.

```
class C {
   C c;

   public void Link(C c) {
      this.c = c;
   }
}

static void Main() {
   C c1 = new C(), c2 = new C();
   c1.Link(c2);
   c2.Link(c1);
}
```





Exercice 1.2

La seule présence des attributs ne suffit pas à assurer une association bi-directionnelle valide. En effet, rien n'empêche qu'un objet X de la classe `A` soit relié à un objet Y de la classe `B` qui, au lieu d'être connecté à X, est branché avec un autre objet Z de la classe `A`. Un peu d'algorithmique est donc nécessaire pour assurer la cohérence de l'association. Ci-dessous une possible implémentation, où la méthode `Link` n'est que partiellement définie. Complétez-la de façon que les objets dans le `main` correspondent aux diagrammes donnés sur le coté (*hint* : analysez d'abord la méthode `Unlink`).

```
class B
{
   A a;

   public void Unlink()
   {
      if( this.a !=  null ) {
         A aa = this.a;
         this.a = null;
         aa.Unlink();
      }
   }

   public void Link(A a)
   {
      if( ____________ ) {
         Unlink();
         if( a != null ) {
            // relier 'this' avec 'a'
            ---------------
            ---------------
         }
      }
   }
}

class A
{
   B b;

   public void Unlink() {
      ... // comme avant
   }

   public void Link(B b) {
      ... // comme avant
   }
}

static void Main()
{
   A a  = new A();
   B b1 = new B()
   B b2 = new B();
   b1.Link(a);
   b2.Link(a);
}
```

Diagramme d'objets après `b1.Link(a)`

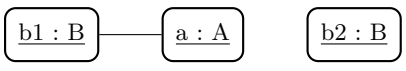
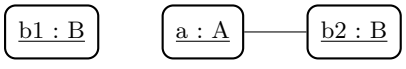


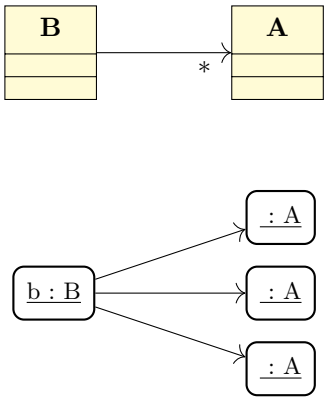
Diagramme d'objets après `b2.Link(a)`



Multiplicité

Normalement, une association indique que l'objet d'une classe est relié à un seul objet d'une autre classe. Il est toutefois possible de changer la *multiplicité* à chaque extrémité de l'association, afin de spécifier le nombre minimum et maximum d'objets participants à la relation. UML définit 4 types de multiplicité : *exactement un* (1..1), *au plus un* (0..1), *plusieurs* (0..*) et *au moins un* (1..*). Concrètement, la multiplicité « * » indique qu'une classe possède un attribut de type « tableau » pour stocker des références sur les objets de l'autre classe. En C#, cela peut se traduire par la classe `List` ou `Dyctionary` : la première est préférable quand il faut respecter un ordre ou récupérer les objets à partir d'un indice entier, alors que la deuxième est préférable quand il faut récupérer les objets à partir d'une clé arbitraire.

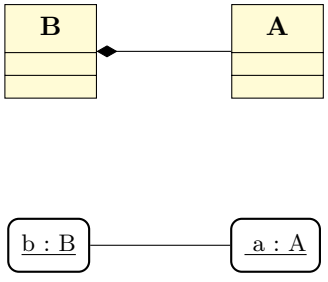
```
class B {
   List<A> ra = new List<A>();

   public void Link(A a) {
      ra.Add(a);
   }
   public void Unlink(A a) {
      ra.Remove(a);
   }
}

static void Main() {
   B b = new B();
   b.Link( new A() );
   b.Link( new A() );
   b.Link( new A() );
}
```



Composition

Une association représente une relation entre deux classes de même niveau conceptuel : aucune des deux n'est plus importante que l'autre. En revanche, pour exprimer que l'une des classes contient l'autre, on peut utiliser un type spécial d'association appelé *composition*. Cette relation indique qu'un objet de la classe « contenue » appartient toujours à un seul objet de la classe « composite », lequel est responsable de la création et destruction des objets qu'il contient. La composition est dénotée en UML par un losange noir (♦) à l'extrémité de la ligne d'association du coté de la classe composite.

```
class B {
   A a;
   public B() {
      a = new A(this);
   }
}
class A {
   B b;
   public A(B b) {
      this.b = b;
   }
}

static void Main() {
   B b = new B();
}
```



Il existe un autre type d'association, marqué par un losange blanc (◊), qui suggère vaguement une relation « tout-partie ». Bien qu'elle soit défini en UML avec le terme *agrégation*, cette relation ne possède pas de sémantique distincte de l'association. Pour citer l'un des créateurs d'UML : « *en dépit du peu de sémantique attaché à l'agrégation, tous pensent qu'elle est nécessaire pour différents raisons ; considerez-la comme un placebo.* » Suivez donc les conseils des créateurs d'UML et abstenez-vous d'utiliser l'agrégation : quand c'est nécessaire, utilisez plutôt la composition !

Agrégation

Exercice 1.3

Vous trouverez ci-dessous l'implémentation des classes `Plateau` et `Case`. Elles représentent un quadrillage de cases, où chaque case est reliée à ses voisines, qui peuvent être 3, 5 ou 8 selon la position. Décrivez les codes ci-dessous en utilisant un diagramme de classes et un diagramme d'objets. (Pour le diagramme d'objets, concentrez-vous sur les objets de la classe `Case`.)

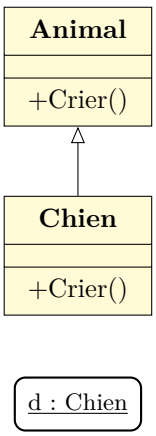
```
public class Plateau
{
   Case[,] cases;

   public Plateau(int largeur, int hauteur)
   {
      cases = new Case[largeur, hauteur];

      for (int x=0; x < largeur; x++) {
         for(int y=0; y < hauteur; y++)
         {
            cases[x,y] = new Case();

            int N = hauteur - 1;
            if (x>0 && y>0) Connecter( cases[x,y], cases[x-1,y-1] );
            if (x>0       ) Connecter( cases[x,y], cases[x-1,y  ] );
            if (       y>0) Connecter( cases[x,y], cases[x  ,y-1] );
            if (x>0 && y<N) Connecter( cases[x,y], cases[x-1,y+1] );
         }
      }
   }

   void Connecter(Case a, Case b) {
      a.Connecter(b);
      b.Connecter(a);
   }
}

public class Case
{
   List<Case> voisines = new List<Case>();

   public void Connecter(Case c) {
      voisines.Add(c);
   }
}

public void main() {
   Plateau p = new Plateau(3,3);
}
```

1.1.4 Relation de généralisation

Généralisation

La *généralisation* indique qu'une classe « spécialisée » possède toutes les caractéristiques de une classe « générale » : le type, les attributs et les méthodes. Par conséquent, un objet de la classe spécialisée est également un objet de la classe générale. La relation de généralisation se traduit par le concept d'héritage dans la plupart des langages objet. Elle est représentée en UML par une flèche avec un trait plein dont la pointe est un triangle fermé (—▷) orienté vers la classe générale. Voici un exemple.

```
class Animal
{
    public virtual void Crier() {
        Console.WriteLine("?");
    }
}

class Chien : Animal
{
    public override void Crier() {
        Console.WriteLine("Bau");
    }
}

static void Main()
{
    Animal d = new Chien();
    d.Crier(); // sortie : bau
}
```



La classe générale est parfois qualifiée de classe base, classe mère ou super-classe, tandis que la classe spécialisée peut être désignée par les termes de sous-classe, classe fille, classe dérivée ou encore de spécialisation. Pour rappel, les propriétés principales de l'héritage entre classes sont les suivantes.

1. **Héritage d'attributs et de méthodes**. La classe fille possède toutes les caractéristiques de la classe mère, mais elle ne peut pas accéder aux caractéristiques privées de cette dernière.
2. **Héritage d'associations**. Toutes les associations de la classe mère s'appliquent aux classes filles.
3. **Héritage de type**. Une instance de la classe fille peut être utilisée partout où une instance de la classe mère est attendue. Bien entendu, dans ce cas, seules les caractéristiques définies par l'interface de la classe mère sont accessibles.
4. **Polymorphisme**. La classe fille peut redéfinir (en gardant la même signature) les méthodes de la classe mère. Sauf indication contraire, un objet utilise les opérations les plus spécialisées dans la hiérarchie des classes, même quand l'objet est perçu comme une instance des classes parents : c'est le principe du polymorphisme d'héritage.
5. **Héritage multiple**. Une classe peut avoir plusieurs parents, bien que cela n'est pas permis dans les langages tels que Java ou C#.

Exercice 1.4

Vous trouvez ci-dessous l'implémentation des classes `Repertoire`, `Fichier` et `Raccourci`, qui représentent un système de gestion de fichiers. Décrivez les codes en utilisant un diagramme de classes et un diagramme d'objets.

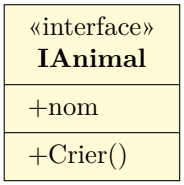
```
class Element
{
   public string nom;
   public Repertoire parent;

   public string NomAbsolu()
   {
      string path = "/" + nom;

      Element e = parent;
      while( e != null )
      {
         path = "/" + e.nom + path;
         e = e.parent;
      }
      return path;
   }
}

class Raccourci : Element {
   public Cible cible;
}

class Cible : Element { }

class Fichier : Cible { }

class Repertoire : Cible
{
   List<Element> enfants = new List<Element>();

   public void Ajouter(Element e) {
      e.parent = this;
      enfants.Add(e);
   }
}

static void Main()
{
   Repertoire root = new Repertoire { nom = "root" };
   Repertoire home = new Repertoire { nom = "home" };
   Fichier    f1   = new Fichier    { nom = "secret.pdf" };
   Fichier    f2   = new Fichier    { nom = "perso.txt" };
   Raccourci  r    = new Raccourci  { nom = "lien" };

   root.Ajouter(home);
   root.Ajouter(f1);
   home.Ajouter(f2);
   home.Ajouter(r);
   r.cible = f1;

   Console.WriteLine(f1.NomAbsolu()); //sortie: /root/secret.pdf
   Console.WriteLine(f2.NomAbsolu()); //sortie: /root/home/perso.txt
}
```

1.1.5 Relation de réalisation

Une classe est fondée sur le principe de l'*encapsulation*, consistant à masquer les détails d'implémentation derrière une interface faite d'attributs et de méthodes publiques. Cela implique qu'une classe définit en même temps l'interface et l'implémentation de ses objets. En revanche, un classeur de type *interface* ne fait que déclarer les attributs et les méthodes publiques d'un groupe d'objets, sans spécifier leur implémentation. C'est pourquoi une interface ne peut pas instancier des objets (notez qu'il est possible d'utiliser l'héritage, même multiple, entre interfaces). Elle est représentée en UML comme une classe avec le stéréotype « interface ». Voici un exemple.

Interface

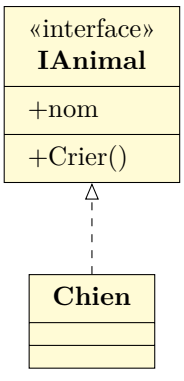
```
interface IAnimal
{
    string nom
    {
        get;
        set;
    }

    void Crier();
}
```



Comme une interface n'a qu'un objectif de déclaration, au moins une implémentation doit lui être associée. La relation de *réalisation* permet d'indiquer qu'une classe implémente les méthodes déclarées dans une interface. Graphiquement, cela est représenté par un trait discontinu terminé par une flèche triangulaire ($- - \triangleright$). Notez également qu'une classe peut réaliser plusieurs interfaces et une interface peut être réalisée par plusieurs classes.

Réalisation

```
class Chien : IAnimal
{
    string race = "Chien";

    public string name
    {
        get{ return race;  }
        set{ race = value; }
    }

    void Crier()
    {
        Console.WriteLine("Bau");
    }
}

static void Main()
{
    IAnimal a = new Chien();
    a.Crier();
}
```



Pour simplifier la notation, UML permet de noter la réalisation d'une interface par un lollipop ($-\circ$) et la dépendance d'une interface par un socket ($-\subset$).

Exercice 1.5

L'héritage est une des grandes qualités de la programmation orientée objet, mais il est souvent mal employé sous le prétexte de favoriser la réutilisation du code. Un exemple est l'omniprésent projet de cataloguer les animaux et leurs cris. Puisque tous les animaux crient, il serait stupide de réécrire la méthode `Crier` pour chaque catégorie d'animaux. Nous pouvons alors la factoriser dans une super-classe `Aminal` (cf. section 1.1.4), afin d'atteindre une glorieuse réutilisation de code qui révolutionnera le catalogage des animaux !

Malheureusement, l'héritage impose une contrainte d'immutabilité aux classes dérivées : elles doivent agir comme la classe de base. Dans l'exemple précédent, la contrainte est que tous les animaux doivent crier, ce qui rend impossible d'ajouter un animal qui ne fait que du bruit. Cela montre une limitation de l'héritage, mais pas nécessairement de la programmation orientée objet. En effet, il est largement adopté le principe de « *préférer la composition d'objets à l'héritage de classes.* » En règle générale, l'héritage est approprié seulement quand les deux conditions suivantes sont vérifiées :

1. la sous-classe étend mais ne nullifie pas les comportements hérités ;
2. un objet n'a jamais besoin de transmuter (changer de classe).

L'exemple des animaux viole la condition 1 au moment où nous ajoutons la classe `Grillon`, car la méthode `Crier` n'est pas définie pour ce type d'animaux. Une implémentation qui respecte le principe de *réutilisation par composition* est montrée ci-dessous. Décrivez le code en utilisant un diagramme de classes et comparez-le à la solution basée sur le simple héritage.

```
abstract class Animal
{
   protected ICrier a;
   protected IBruit b;

   public void Crier() {
      a.Crier();
    }
   public void Bruit() {
      b.Bruit();
   }
}

class Chien : Animal
{
   public Chien() {
      a = new CriDeChien();
      b = new PasDeBruit();
   }
}

class Grillon : Animal
{
   public Grillon() {
      a = new PasDeCri();
      b = new BruitDeGrillon();
   }
}
```

```
interface ICrier {
   public void Crier();
}
interface IBruit {
   public void Bruit();
}

class PasDeCri : ICrier {
   public void Crier() {
      Console.WriteLine("...");
   }
}
class CriDeChien : ICrier {
   public void Crier() {
      Console.WriteLine("Bau");
   }
}

class PasDeBruit : IBruit {
   public void Bruit() {
      Console.WriteLine("...");
   }
}
class BruitDeGrillon : IBruit {
   public void Bruit() {
      Console.WriteLine("Cri-Cri");
   }
}
```

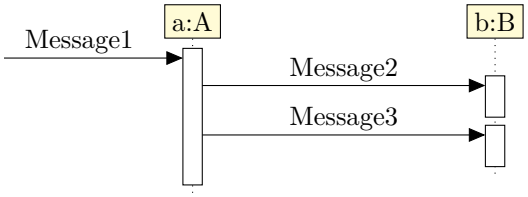
1.2 Diagramme de séquence

Le diagramme de séquence est un schéma qui représente la façon dont les objets interagissent via des messages. Il s'agit d'une vue dynamique, car elle met l'accent sur la chronologie des envois de messages. La plus parte des débutants en UML pensent généralement que les diagrammes de classes sont les éléments les plus importants de la modélisation objet. Ce n'est pas vrai ! En effet, ce n'est qu'au moment où nous devons penser concrètement aux messages à envoyer, à qui les envoyer et dans quel ordre, que notre attention est réellement focalisée sur les détails concrets de la conception objet.

1.2.1 Lignes de vie

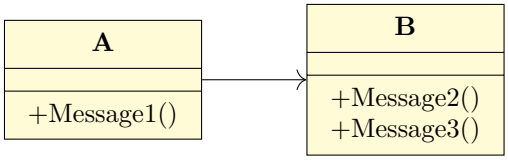
Ligne de vie

Une *ligne de vie* est l'un des participants à l'interaction modélisée par un diagramme de séquence. Bien que tout élément actif du système puisse participer à une interaction, sur le plan pratique et informel, une ligne de vie équivaut souvent à une instance d'une classe. Elle est dénotée en UML par un rectangle contenant l'étiquette « nom objet : nom class », auquel est accroché un trait vertical (souvent pointillée) indiquant l'écoulement du temps. Les principales informations contenues dans un diagramme de séquence sont les communications échangées entre les lignes de vie. Graphiquement, une communication est symbolisée par une flèche entre deux lignes de vie, orientée de l'expéditeur vers la cible. La chronologie des communications est organisée le long des linges de vie, du haut vers le bas. Voici un exemple.



Quel est la représentation de ce diagramme dans le code ? Probablement que la classe `A` dispose d'une méthode nommé `Message1` et entretient une relation avec la classe `B`, laquelle possède les méthodes `Message2` et `Message3`. Si les classes sont reliées par une association, la définition partielle de `A` est :

```
class A {
   B b;
   void Message1() {
      b.Message2();
      b.Message3();
   }
}
```

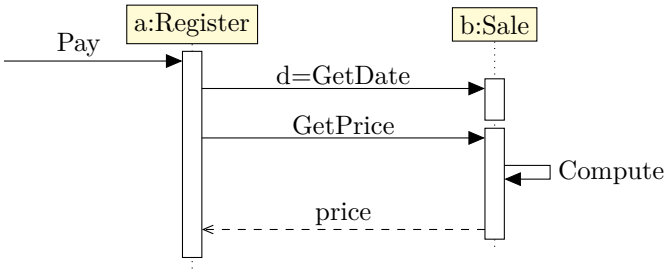


Notez qu'il n'existe pas un ordonnancement linéaire qui consisterait à créer les diagrammes de séquence avant les diagrammes de classe ou vice versa. Dans un processus de développement piloté par la modélisation, ces vues dynamiques et statiques complémentaires sont représentées simultanément !

1.2.2 Messages

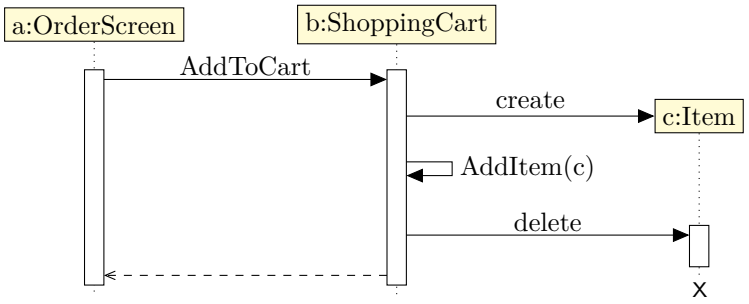
Un *message* est une communication entre deux lignes de vie, telles que l'invocation d'une méthode ou la création/destruction d'une instance. Il est représenté en UML par une flèche continue (—▶). Lors de l'invocation d'une méthode, le message est porteur de l'expression `retour = méthode( paramètres )`, dont le retour peut alternativement apparaitre sur un message de réponse indiqué par une flèche pointillée (⇠--). Il est également possible de représenter la pile d'appels d'une méthode à l'aide d'une barre d'activation, qui peut être imbriquée quand un objet s'envoie un message à lui-même.

Message d'invocation



Dans certaines circonstances, il est souhaitable de montrer explicitement la création ou la destruction des lignes de vie. Un message de création équivaut à invoquer l'opérateur `new` et appeler le constructeur de l'objet créé. En revanche, un message de destruction indique que un'objet n'est plus utilisable, mais sa réelle destruction n'est pas forcement consécutive à la réception du message (cela dépende du ramasse-miettes). Graphiquement, le message de création est représenté par une flèche noire pointillée (- - -▶) qui pointe sur le sommet d'une ligne de vie, alors que celui de destruction est matérialisé par une croix (—▶X) qui marque la fin d'une ligne de vie.

Messages de creation et destruction



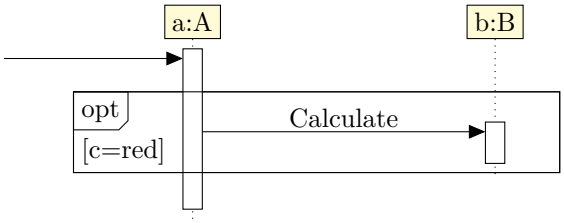
Le polymorphisme est un concept fondamentale de l'approche objet. Comment le représenter dans un diagramme de séquence ? C'est une question courante en UML. Une solution consiste à utiliser plusieurs diagrammes de séquence : l'un montrant le message polymorphe vers un objet abstrait de la super-classe ; puis plusieurs diagrammes séparés détaillant les cas spécialisés, chacun commençant par le message polymorphe.

Message polymorphe

1.2.3 Cadres d'interaction

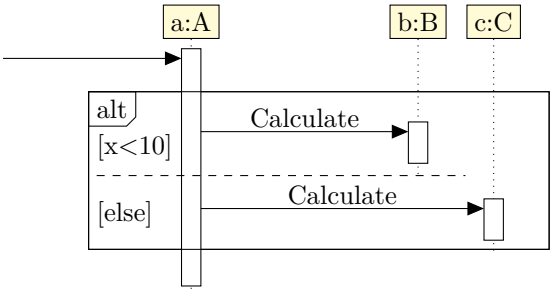
Pour prendre en charge les constructions conditionnelles ou itératives, un diagramme de séquence utilise les *cadres d'interaction*. Graphiquement, un cadre est un rectangle placé autour d'un ou plusieurs messages qui forment un fragment d'interaction. Parmi les 12 cadres UML, les plus utiles sont : le choix optionnel (cadre `opt`), l'alternative (cadre `alt`) et la boucle (cadre `loop`). Le cadre `opt` contient un fragment d'interaction qui sera exécuté seulement si une clause conditionnelle est vrai. La clause est indiquée, entre crochets, sur la ligne de vie à laquelle appartient. Voici un exemple.

Cadre OPT



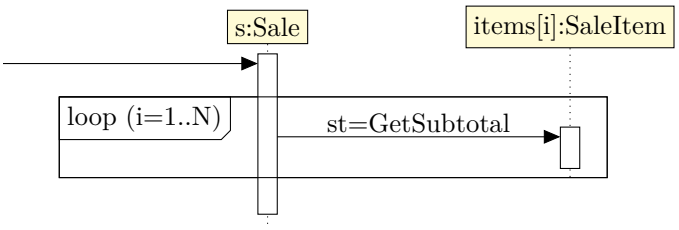
Le cadre `alt` contient deux ou plus fragments d'interaction séparés par des lignes pointillés. Chaque fragment possède une clause conditionnelle. Au plus un seul fragment est exécuté : celui dont la clause est vraie. La clause *else* est vraie si aucune autre condition n'est vraie. Si plusieurs clauses prennent la valeur vraie, le choix est non déterministe. Voici un exemple.

Cadre ALT



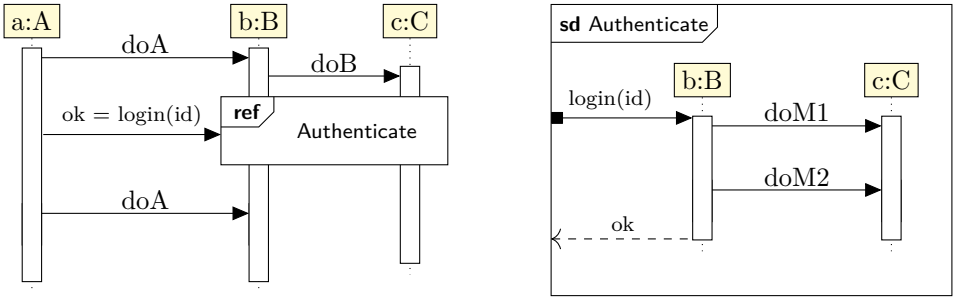
Le cadre `loop` contient un fragment qui sera exécuté plusieurs fois, en fonction d'une clause conditionnelle et/ou d'un intervalle indiquant le nombre minimum et maximum d'itérations. La boucle est répétée tant que la conjonction logique entre les deux conditions n'est pas fausse. Dans un cadre `loop`, une ligne de vie peut utiliser une expression « sélecteur » pour indiquer qu'elle dépende de l'itération courante de la boucle. Voici un exemple.

Cadre LOOP



Il est également possible de faire référence à un fragment d'interaction au sein d'un autre diagramme de séquence. Cela est utile quand on veut simplifier une interaction en factorisant une portion, ou lorsqu'il existe un fragment d'interaction réutilisable. Pour ce faire, il faut entourer le fragment dans un cadre portant l'étiquette `sd` suivie d'un nom, lequel pourra être invoqué dans un autre diagramme en utilisant le cadre `ref`. Voici un exemple.

Cadre REF



Exercice 1.6

Vous trouvez ci-dessous l'implémentation (partielle) de la classe `Player`. Décrivez la méthode `TakeTurn` en utilisant un diagramme de séquence.

```
class Player
{
   Piece  piece;
   Board  board;
   Die[]  dice;
   ...

   public void takeTurn()
   {
      int rollTotal = 0;
      for (int i = 0; i < dice.length; i++) {
         dice[i].roll();
         rollTotal += dice[i].getFaceValue();
      }

      Square newLoc =
         board.getSquare(piece.getLocation(), rollTotal);

      piece.setLocation(newLoc);
   }
}

public class Board
{
   static int SIZE = 40;
   List squares = new List(SIZE);
   ...

   public Square getSquare(Square start, int distance) {
      int endIndex = (start.getIndex() + distance) % SIZE;
      return squares.get(endIndex);
   }
}
```

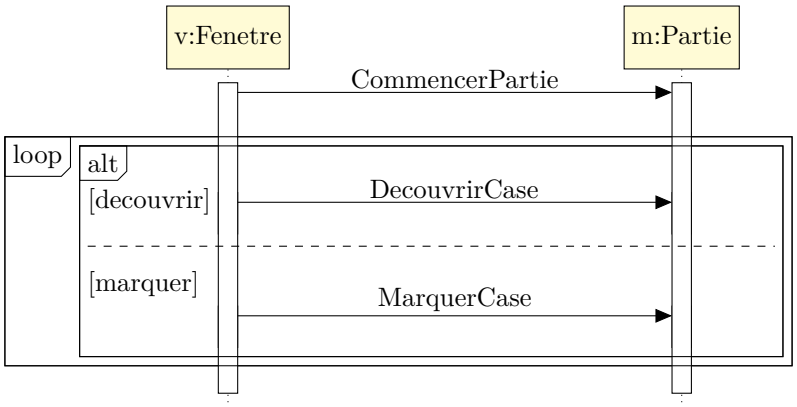
1.3 Mini-projet : démineur

L'objectif de cette étude est de développer un jeu de démineur. Le but du jeu est de trouver, sans les toucher, toutes les cases du plateau contenant des mines. Au début du jeu, les cases sont couvertes. Quand une case est découverte, son contenu est affiché : il peut être rien, une mine ou un nombre indiquant les mines présentes dans les cases voisines. En présence d'une mine, le jeu est terminé et le joueur a perdu. Si le contenu est un chiffre, il ne se passe rien. Enfin, si la case est vide (chiffre zéro), 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.

Une case couverte peut être marquée pour indiquer qu'elle contient potentiellement une mine. Le marquage fait apparaitre un drapeau sur la case et fait décrémenter le compteur de mines restant à localiser. 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.

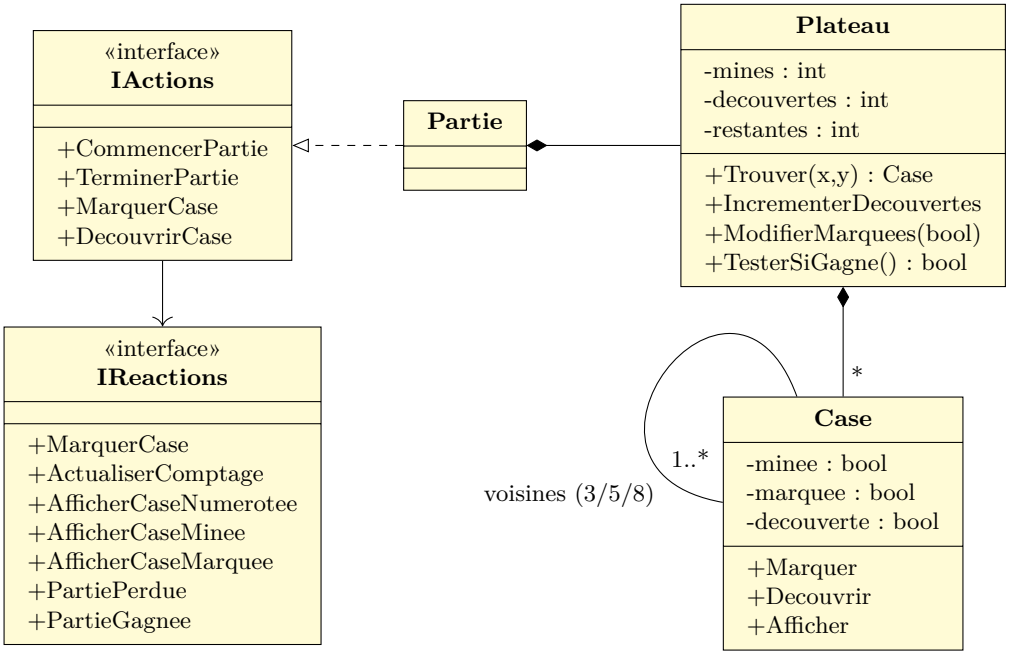
INSTRUCTIONS Pour réaliser le démineur, nous vous fournissons une solution Visual Studio organisée en deux dossiers. Le dossier « IHM » contient une simple interface homme-machine, alors que le dossier « Jeu » contient une implémentation fictive qui permet d'avoir une version exécutable du jeu (lancez-le pour tester). L'objectif du TP est de modifier le dossier « Jeu » afin de réaliser un démineur qui respecte le cahier des charges.

VUE D'ENSEMBLE L'utilisateur du système est le joueur, qui va passer son temps à découvrir ou marquer les cases. Cette interaction, véhiculée par l'IHM, peut être décrite par un diagramme de séquence dont les participants sont les instances des classes `Fenetre` et `Partie`. Le jeu peut être représenté par une grande boucle, dans laquelle les opérations de découverte ou marquage d'une case peuvent arriver dans un ordre quelconque.



1.3.1 Diagramme de classes

Les classes `Fenetre` et `Partie` interagissent entre elles par le biais des interfaces `IActions` et `IReactions`. La première interface énumère toutes les opérations que le joueur peut invoquer durant une partie, alors que la deuxième interface spécifie toutes les événements que le jeu peut émettre durant une partie. Voici le diagramme de classes du système.



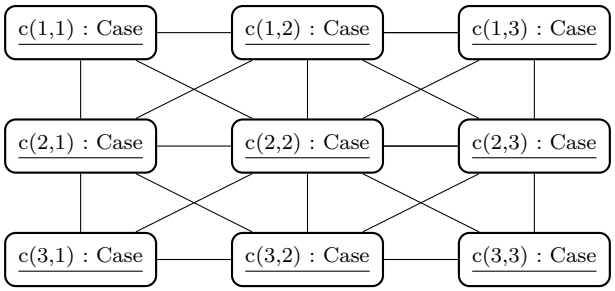
La logique du démineur est implémentée dans les classes `Partie`, `Plateau` et `Case`. La classe `Partie` contrôle le déroulement du jeu. Elle est composée de la classe `Plateau`, qui est instanciée dans la méthode `CommencerPartie` :

```
public class Partie : IActions
{
    Plateau plateau;

    public IReactions vue { get; set; }

    public void CommencerPartie(int largeur, int hauteur, int mines)
    {
        plateau = new Plateau(this, largeur, hauteur, mines);
    }

    ...
}
```

La classe `Plateau` est responsable de créer le quadrillage des cases et décider aléatoirement où placer les mines. Elle fournit des méthodes d'utilité telles que : (1) retrouver une case à partir de ses coordonnées spatiales, (2) compter les cases découvertes, (3) compter les cases marquées, (4) tester si toutes les cases ont été découvertes. Voici une implémentation partielle.

```
public class Plateau
{
   public Partie partie;

   public int largeur;
   public int hauteur;
   Case[,] cases;
   int mines, decouvertes, restantes;
   ...

   public Case Trouver(int x, int y) {
      return cases[x, y];
   }

   public void IncrementerDecouvertes() {
      decouvertes++;
   }

   public void ModifierMarquees(bool marquee)
   {
      if(marquee) restantes--; else restantes++;

      partie.vue.ActualiserComptage(restantes);
   }

   public bool TesterSiGagne() {
      return decouvertes + mines == largeur * hauteur;
   }
}
```

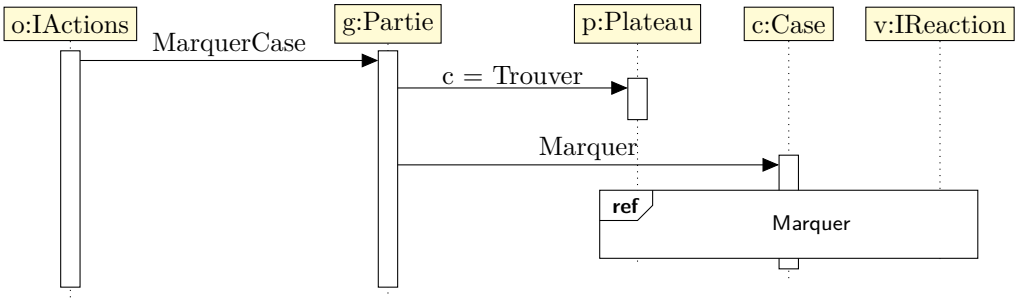
Les cases du quadrillage sont décrites par la classe `Case`. Elle possède une association réflexive qui permet de relier caque case à ses voisines, pouvant être 3, 5 ou 8 selon la position au sein de la grille. C'est dans le constructeur de la classe `Plateau` que les cases sont crées et connectées entre elles. Voici le diagramme d'objets correspondant à un plateau 3x3.



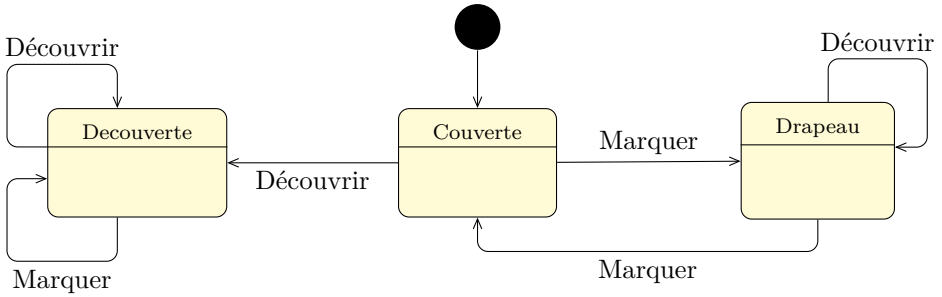
INSTRUCTIONS Votre première tache est de traduire le diagramme de classes (montré plus haut) en C#. Cela consiste à définir les classes `Plateau` et `Case` avec leurs attributs et méthodes. Dans un premier temps, laissez vide le corps des méthodes. Ensuite, implémentez les méthodes de la classe `Plateau`. La seule difficulté est posée par le constructeur, qui est responsable de créer les cases et de les connecter avec leurs voisines. Une fois implémenté et testé la classe `Plateau`, vous pouvez avancer à l'étape suivante, qui décrit comment implémenter les méthodes des classes `Partie` et `Case`.

1.3.2 Opération « marquer case »

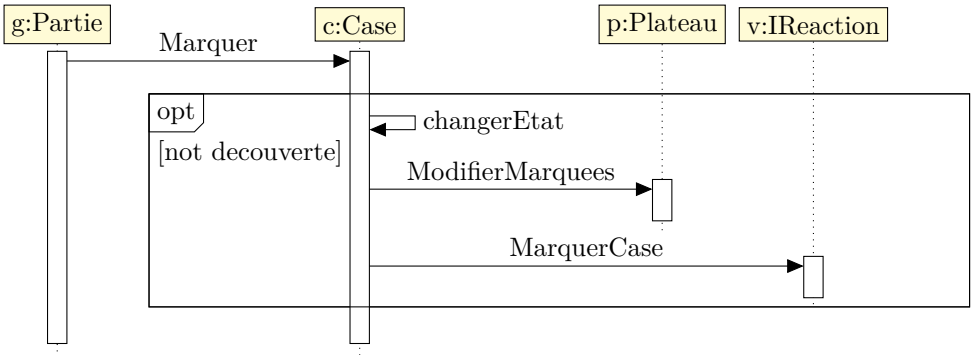
Les opérations principales du démineur se trouvent dans la classe `Partie`. Lorsque la méthode `MarquerCase` est invoquée, l'objet « partie » délègue le travail à l'expert des cases, soit le plateau. Celui-ci va tout d'abord récupérer une référence sur la case correspondant aux coordonnées spécifiées. Il va ensuite déléguer le marquage à la case en appelant la méthode `Marquer`.



Néanmoins, le traitement du marquage dépend de l'état de la case. Attention à bien représenter la double variabilité des cases. Le fait 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). Le fait qu'une case soit marquée (ou pas) varie durant sa vie : il s'agit donc d'une notion d'état. Un simple diagramme d'état permet d'indiquer les effets des méthodes `Decouvrir` et `Marquer` de la classe `Case`.



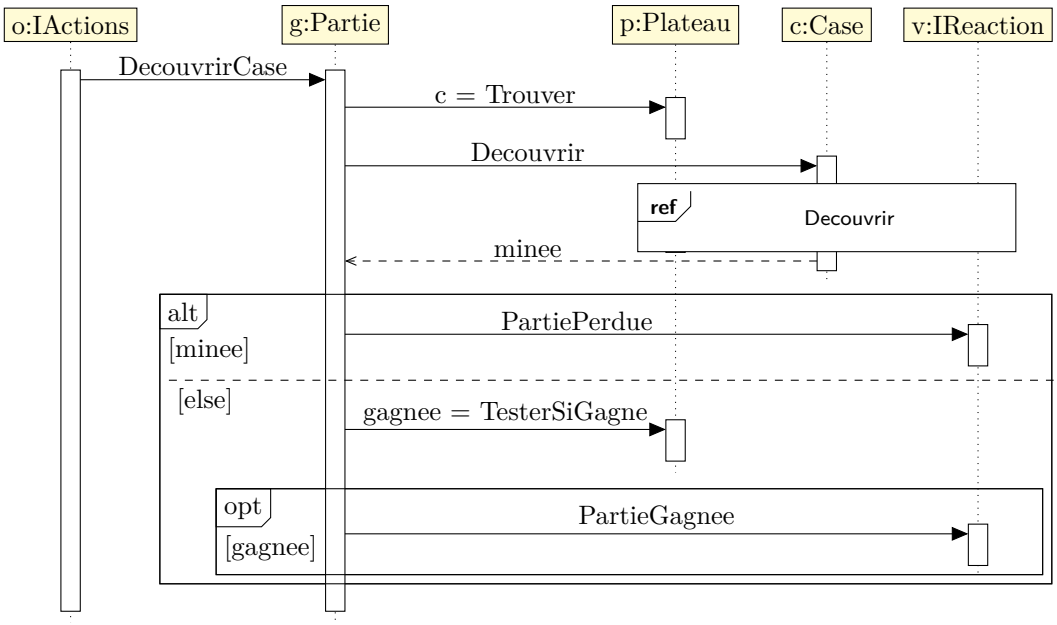
La méthode `Marquer` est alors décrite par le diagramme suivant.



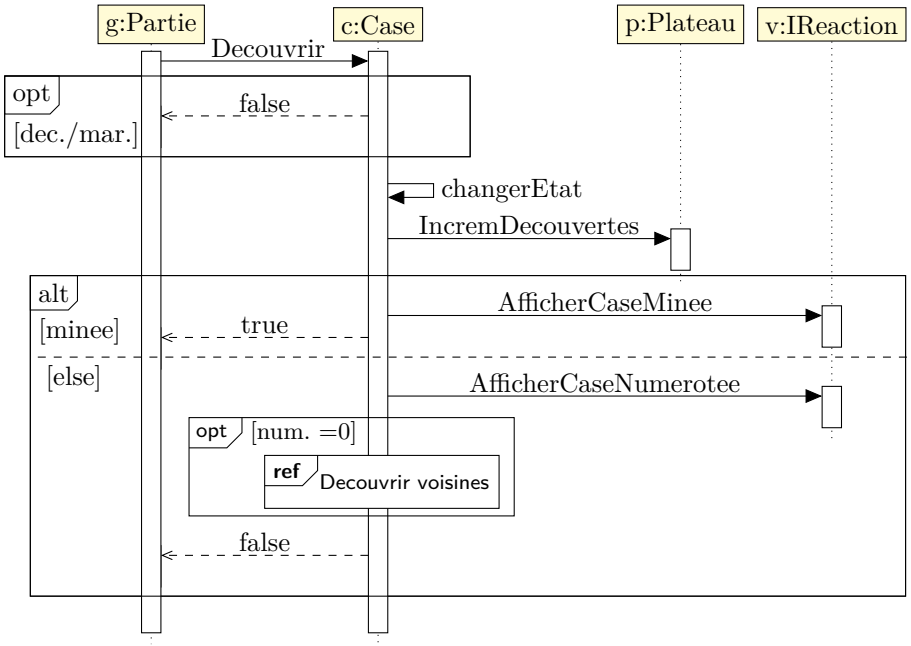
Istructions Implémentez les méthodes `MarquerCase` et `Marquer`.

1.3.3 Opération « découvrir case »

Quand la méthode `DecouvrirCase` est invoquée, l'objet partie récupère la case, appelle la méthode `Decouvrir` et vérifie si le joueur a perdu ou gagné.



La méthode `Decouvrir` vérifie l'état courant et termine de suite si la case est marquée ou découverte. Autrement, elle change son état en découverte et si la case n'est ni minée ni numérotée, elle découvre les cases voisines.



Istructions Implémentez les méthodes `DecouvrirCase` et `Decouvrir`.