Génie logiciel 1

Introduction au cours

Informations sur le cours (1/2)

- **Contenu du cours**
  1. UML
  2. Développement piloté par les tests
  3. Conception dirigée par le domaine
  4. Persistance des données
  5. Projet final (6h en classe + travail personnel)

- **Organisation des activités (de 1 à 4)**
  - 0.5h de cours
  - 2.5h d'exercices et corrections + **travail personnel**
  - 3h de mini-projet + **travail personnel** → à rendre après une semaine pour évaluation

- **REFERENCES**

  - P. Roques, **UML 2 par la Pratique**, Eyerolles, 7e édition, 2009.
  - R. Martin, **Coder proprement**, Pearson, 2009.
  - E. Evans, **Domain-Driven Design**, Addison-Wesley, 2003.
  - J. Lonchamp, **Conception d'applications en Java/JEE**, Dunod, 2014.
  - R. Martin, **Agile Principles, Patterns, and Practices in C#**, Prentice Hall, 2006.
  - E. Freeman et al**., Design Patterns : tête la première**, O'Reilly, 2005.

Informations sur le cours (2/2)

- **NOTATION (travail en binôme)**
  - Note du mini-projet 1 : $N_1 \in [0,2]$ → Correction sur place
  - Note du mini-projet 2 : $N_2 \in [0,3]$ → Rendu du code et du rapport
  - Note du mini-projet 3 : $N_3 \in [0,3]$ → Rendu du code et du rapport
  - Note du mini-projet 4 : $N_4 \in [0,3]$ → Rendu du code et du rapport
  - Note du projet final : $N_5 \in [0,9]$ → Présentation et démo (5 min.)

- **Projet final (en binôme) : sujet à choix !**
  - Propositions détaillés pour la fin de la séance 2
  - Validation pour la fin de la séance 3
  - Cahier des charges prêt pour la fin de la séance 4
  - La présentation finale remplace le partiel

Qu’est-ce que le génie logiciel ?

Comment passer des besoins aux codes ?

Besoins

Codes

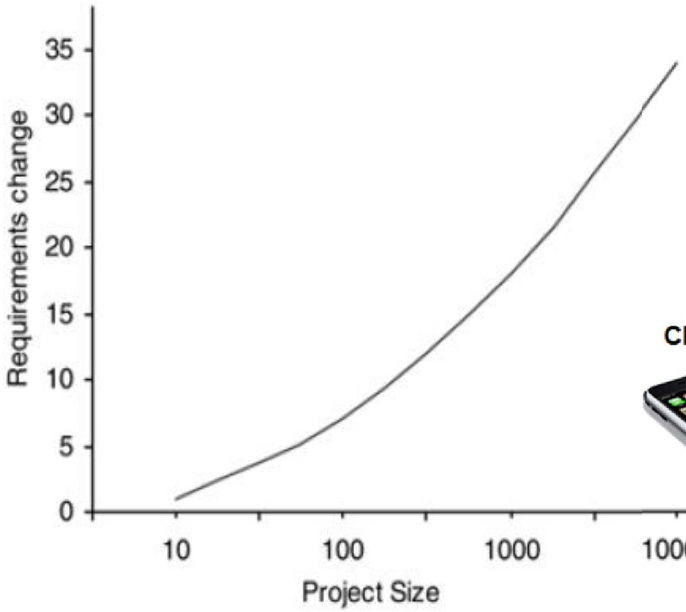
GÉNIE LOGICIEL

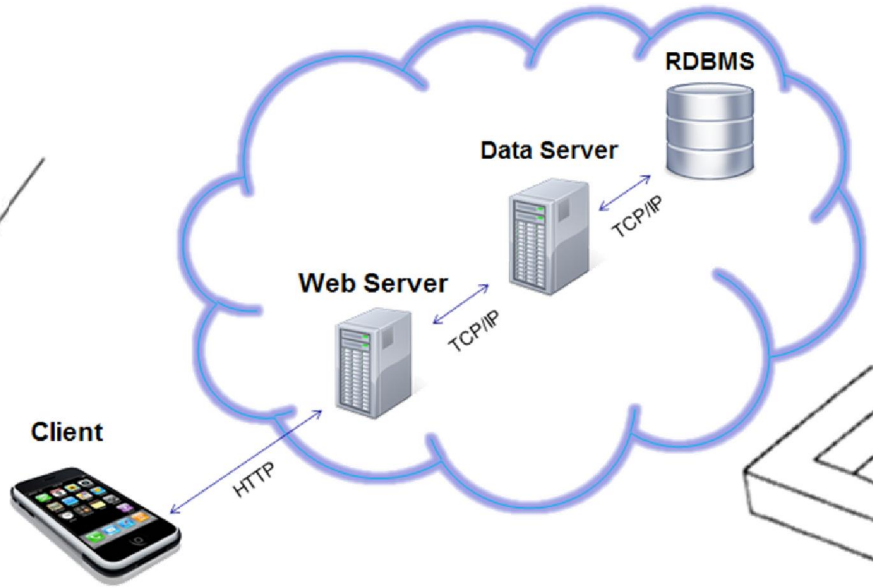
méthodes et pratiques de développement

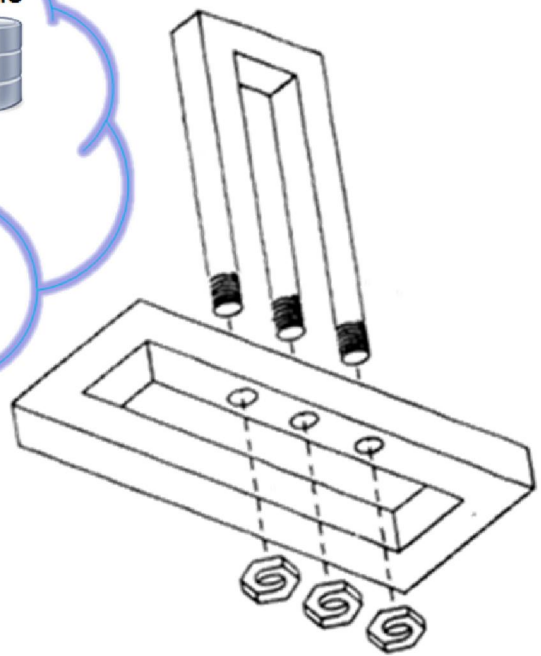
Vous êtes ici

Le développement est complexe !

- Difficultés rencontrées dans le développement logiciel
  - *ambigüité des besoins*
  - *inexpérience dans le domaine d'intérêt*
  - *intégration de nombreux outils, méthodes et technologies*
  - *changements & grande échelle*







Comment gérer la complexité ?

- La complexité est gérée de deux manières
  - ***pratique 1*** → *le développement est organisé en étapes*
  - ***pratique 2*** → *chaque étape est pilotée par l'**approche orientée objet***

Besoins

Analyse
(des besoins)

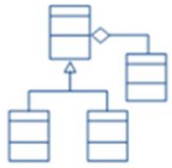
Conception
(d'une solution)

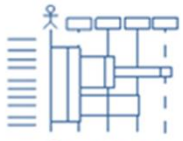
Implémentation
(de la solution)

Codes

<u>Comment concrétiser la pensée objet dans les étapes du développement ?</u>

Par la modélisation graphique :





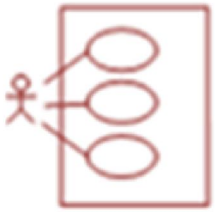
Génie logiciel 1

Modélisation graphique

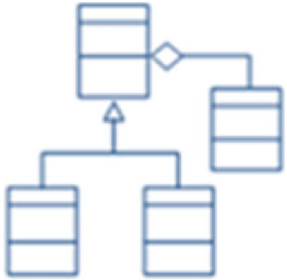
Langage de modélisation unifié

- UML est une **notation graphique** qui permet de spécifier un logiciel selon des points de vues distincts et complémentaires.
  - ***Vue fonctionnelle*** → *Description des besoins utilisateurs*
  - ***Vue statique*** → *Structure des classes*
  - ***Vue dynamique*** → *Interaction entre les objets*

Vue fonctionnelle
(diagramme de cas d'utilisation)



Vue statique
(diagramme de classes)



Vue dynamique
(diagramme de séquence)

Diagramme de classes

- C'est la description des **classes** et des **relations** entre celles-ci
  - ***Vue statique*** → *les aspects temporels et dynamiques sont absents*

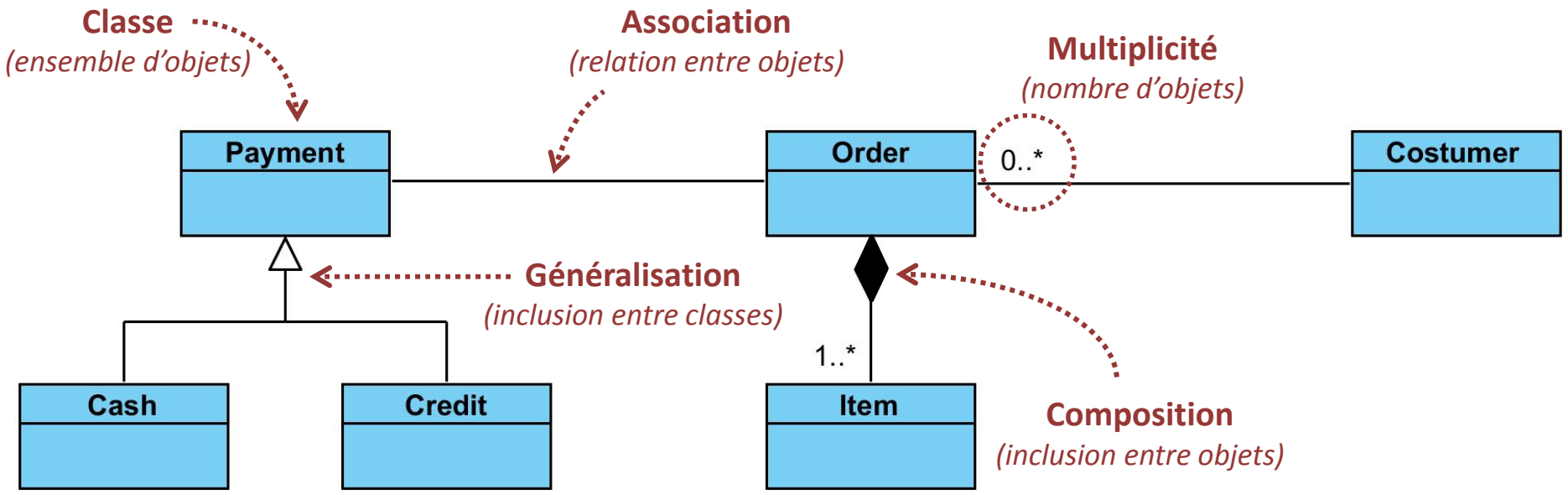
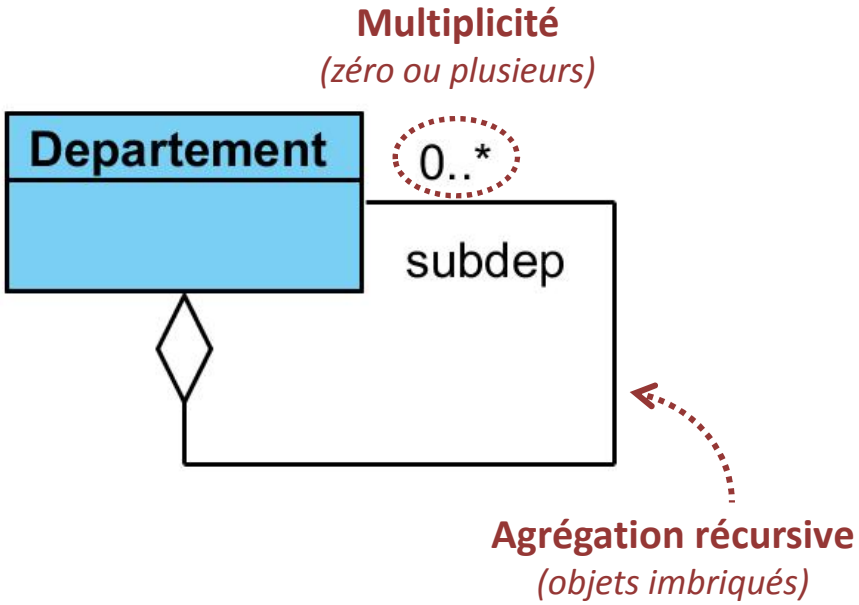


Diagramme d'objets

- C'est le graphe des **objets instanciés** et de leurs **connexions**
  - ***Vue statique*** → *il décrit un digramme de classes en termes d'objets*
  - ***Vue statique*** → *c'est une photographie à un instant précis des objets*



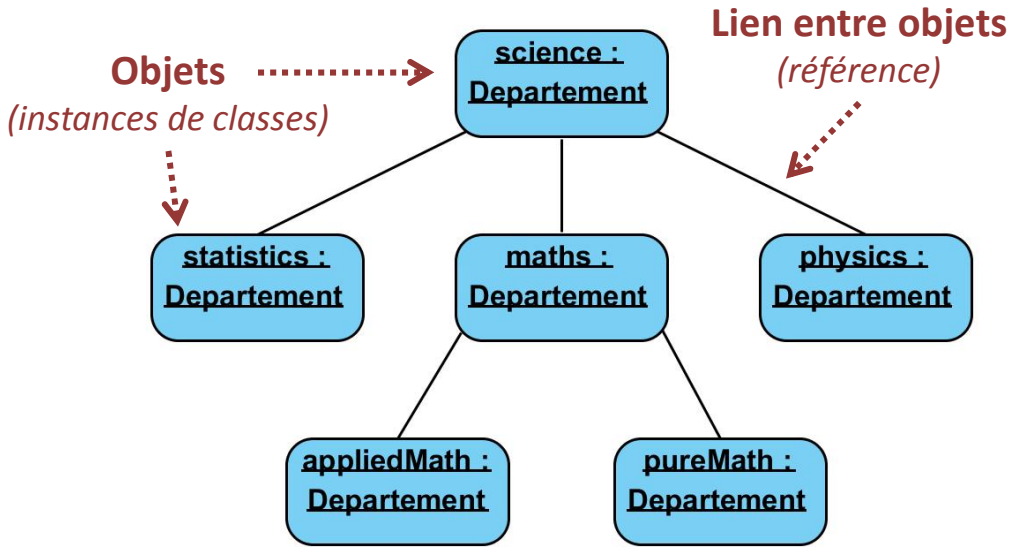
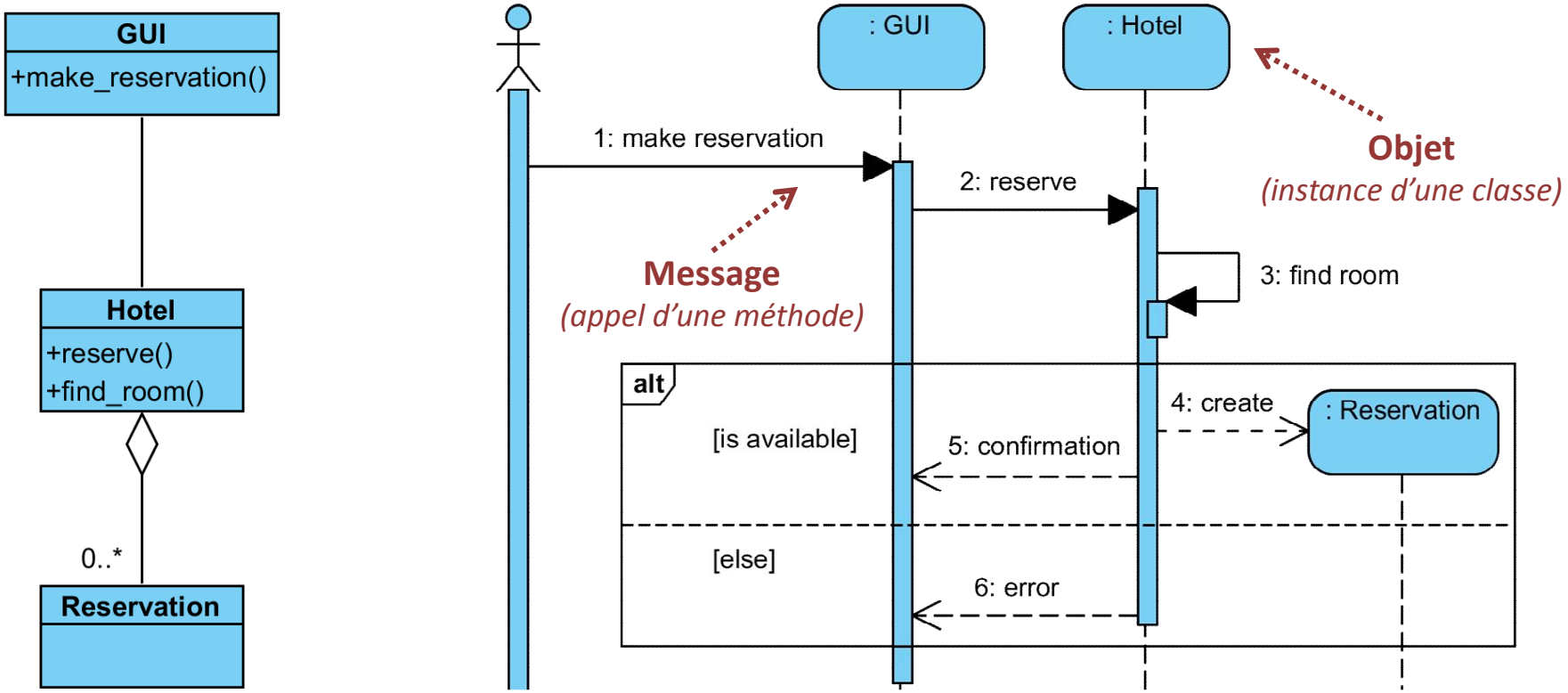


Diagramme de séquence

- C'est la description des **messages échangés** entre les objets
  - ***Vue dynamique*** → *il présent les interactions selon un ordre chronologique*



Génie logiciel 1

Exercices

Qu'est-ce qu'une classe ?

- Une **classe** est la description d'objets du même type
  - ***attributs*** → *variables qui mémorisent l'**état** d'un objet*
  - ***méthodes*** → *fonctions qui définissent le **comportement** d'un objet*
  - ***classe*** → *nom qui identifie le **type** d'un objet*

```
class Point
{
  float x, y;

  public Point(float a=0, float b=0){
    x = a; y = b;
  }

  public void Add(Point d){
    x += d.x;
    y += d.y;
  }
}
```

Classe

Point
- x - y
+ Add

Objets

p1 : Point

p2 : Point

p3 : Point

Ex 1.1 – Dépendance

- Dans quels cas, la relation entre les classes Player et Die désigne une simple **dépendance** ?

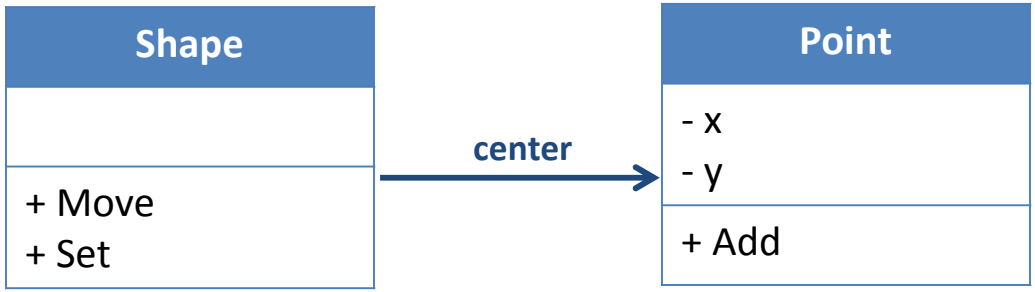
```
class Die
{
  Random rnd = new Random ();

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

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

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

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

class Player {
  Die die;

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

Qu'est-ce que une association ?

- L'**association** est une connexion durable entre deux **objets**
  - *la classe A contient un attribut dont le type est la classe B*
  - *un objet de A possède une référence à un objet de B*

```
class Shape {
  Point center;

  public void Move(Point p) {
    center.Add(p);
  }

  public void Set(Point p){
    center = p;
  }
}
```

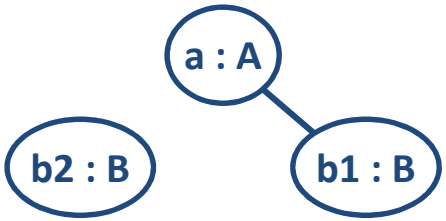


Ex. 1.2 - Association

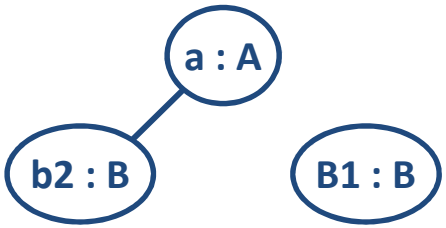
- Complétez le code ci-dessous afin d'assurer que l'association bidirectionnelle soit valide.

```
class B
{
  A a;

  public void Unlink(){...}

  public void Link(A a)
  {
    if( _____________ )
    {
      Unlink();
      if(a != null) {

        ________________
        ________________
      }
    }
  }
}
```

b1.Link(a)



b2.Link(a)

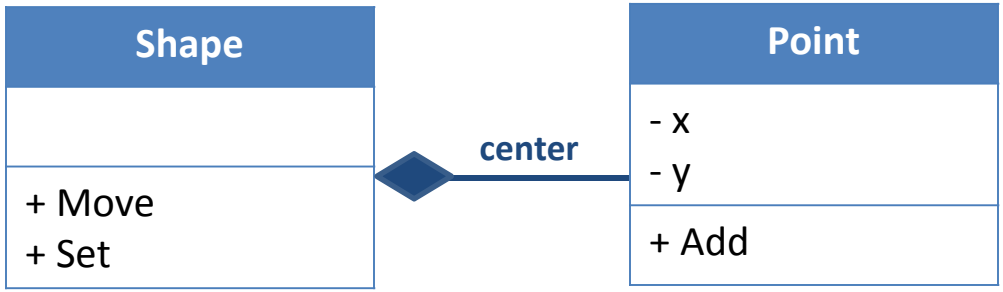



```
class A
{
  B b;

  public void Unlink(){...}

  public void Link(B b){...}
}

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

Qu'est-ce que une composition ?

- La **composition** est une relation d'inclusion entre deux **objets**
  - *la classe A contient un attribut dont le type est la classe B*
  - *un objet de A possède une référence à un objet de B*
  - ***à noter*** → *quand l'objet de A est détruit, l'objet de B est détruit aussi*

```
class Shape {
  Point center;

  public Shape() {
    center = new Point();
  }
  public void Move(Point p) {
    center.Add(p);
  }
  public void Set(int x,int y){
    center = new Point(x,y);
  }
}
```



Ex. 1.3 - Composition

- Décrivez les codes ci-dessous en utilisant un diagramme de classes et un diagramme d'objets.

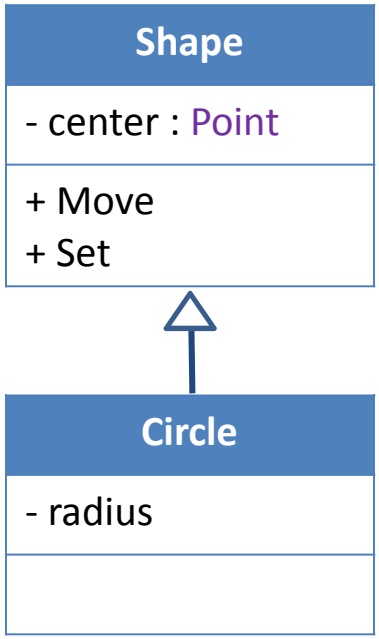
```
public class Plateau {
  Case[,] cases;
  public Plateau(int l, int h) {
    cases = new Case[l,h];
    for (int x =0; x < largeur; x++)
    for (int y =0; y < hauteur; y++) {
      cases[x,y] = new Case();
      if (x > 0 && y > 0)
        Connecter(cases[x,y],cases[x-1,y-1]);
      if (x > 0)
        Connecter(cases[x,y],cases[x-1,y]);
      if (y > 0)
        Connecter(cases[x,y],cases[x,y-1]);
      if (x > 0 && y < h-1)
        Connecter(cases[x,y],cases[x-1,y+1]);
    }
  }
  void Connecter(Case a, Case b) {
    a.Connecter(b);
    b.Connecter(a);
  }
}
```

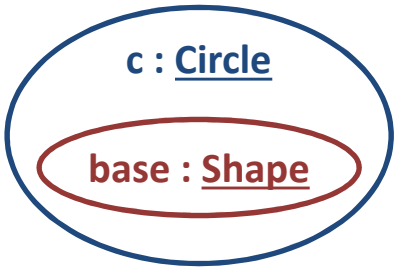
```
public class Case
{
  List<Case> voisines = new List <Case>();

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

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

Qu'est-ce que l'héritage ?

- L'**héritage** est une relation d'inclusion entre deux **classes**
  - ***héritage d'état*** → *la classe A acquiert les attributs de la classe B*
  - ***hér. de comportement*** → *la classe A acquiert les méthodes de la classe B*
  - ***héritage de type*** → *les objets de A sont également des objets de B*

```
class Circle : Shape
{
  float radius;

  public Circle(Point p, int r)
    : base(p)
  {
    radius = r;
  }

  // Point center;
  // void Move(Point p){...}
  // void  Set(Point p){...}
}
```





Qu'est-ce que le polymorphisme ?

- Le **polymorphisme** est la redéfinition d'une méthode par l'héritage
  - *on garde l'interface de la méthode déclarée dans la classe mère*
  - *on change le comportement de la méthode dans la classe dérivée*
  - ***à noter*** → *le but est d'avoir un comportement variable selon le type*

```
class Shape {
   ...
   public virtual float Area(){
     return -1;
   }
}

class Circle : Shape {
   ...
   public override float Area(){
     return 3.14*radius*radius;
   }
}
```

Comportement à l'exécution

```
void print(Shape s) {
   Console.Write( s.Area() );
}

Point p = new Point(0,0);
Shape a = new Shape(p);
Shape b = new Circle(p,1) );

print(a); // résultat: -1
print(b); // résultat: 3.14
```

Ex. 1.4 - Généralisation

- Décrivez les codes ci-dessous en utilisant un diagramme de classes et un diagramme d'objets.

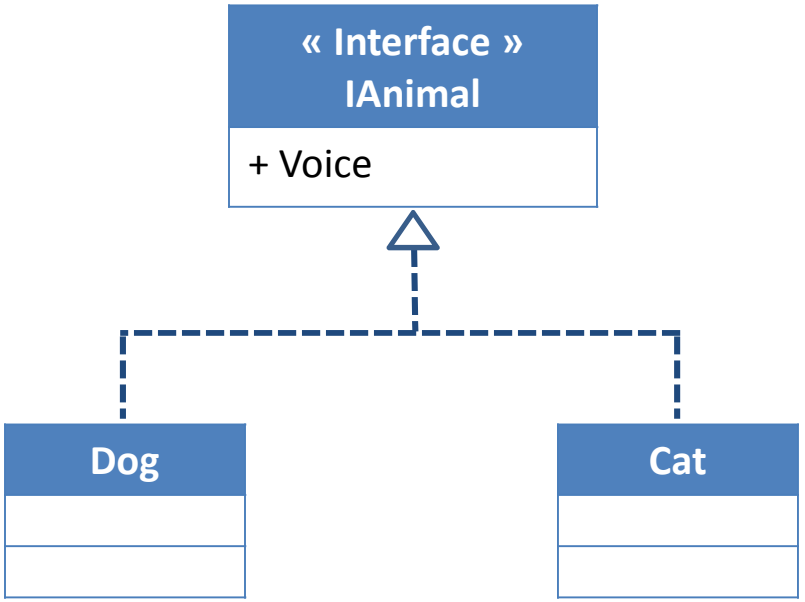
```
class Element {
  public string nom;
  public Repertoire parent;

  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());
  Console.WriteLine(f2.NomAbsolu());
}
```

Qu'est-ce qu'une interface ?

- Une **interface** est la déclaration d'un groupe de méthodes
  - *c'est un **type abstrait** à la racine d'une hiérarchie de classes*
  - *les classes filles fournissent une implémentation des méthodes déclarées*
  - ***à noter*** → *une classe peut hériter de **plusieurs** interfaces*

```
interface IAnimal {
  void Voice();
}

class Dog : IAnimal {
  public void Voice() {
    Console.Write("Bau");
  }
}

class Cat : IAnimal {
  public void Voice() {
    Console.Write("Miao");
  }
}
```



Ex. 1.5 - Réalisation

- Décrivez les codes ci-dessous en utilisant un diagramme de classes et un diagramme d'objets.

```
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();
}

class PasDeCri : ICrier
{
  public void Crier() {
    Console.Write("...");
  }
}

class CriDeChien : Icrier
{
  public void Crier() {
    Console.Write("Bau");
  }
}
```

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

class PasDeBruit : IBruit
{
  public void Bruit() {
    Console.Write("...");
  }
}

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

Ex. 1.6 – Vue dynamique

- Décrivez le code ci-dessous par 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);
  }
}
```