# Langages Orientés Objet

*Collaborations*
*Composition, Agrégation et Association*

Elisabetta Bevacqua

# Collaborations entre classes

- Un objet ne sait pas tout faire mais il peut être en relation avec d'autres objets et collaborer avec eux :
  - par des connexions entre instances,
  - par exécution de méthodes de ces instances.
- Nous verrons :
  - Association,
  - Agrégation,
  - Composition.

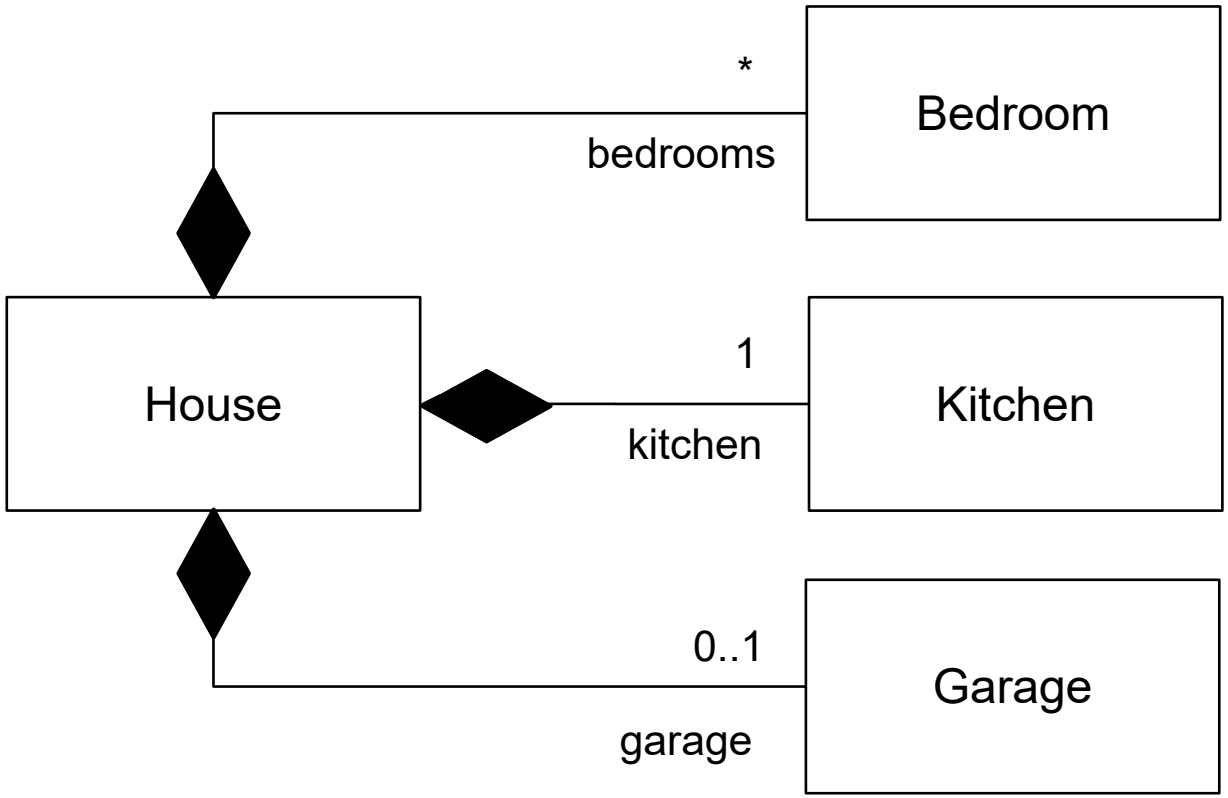
# Composition

- La composition est une relation de type « *possède un* » ou « *possède plusieurs* ». Elle est unidirectionnelle.
- Les « objets composites » sont des instances de classes composées.
- Les cycles de vies des éléments (les “composants”) et du composite sont liés : si le composite est détruit (ou copié), ses composants le sont aussi. (**Même cycle de vie**).
- A un même moment, une instance de composant ne peut appartenir qu’à un seul composite. (**Non-partageabilité**)

# Exemple

Même cycle de vie

# Rappel : la composition en UML



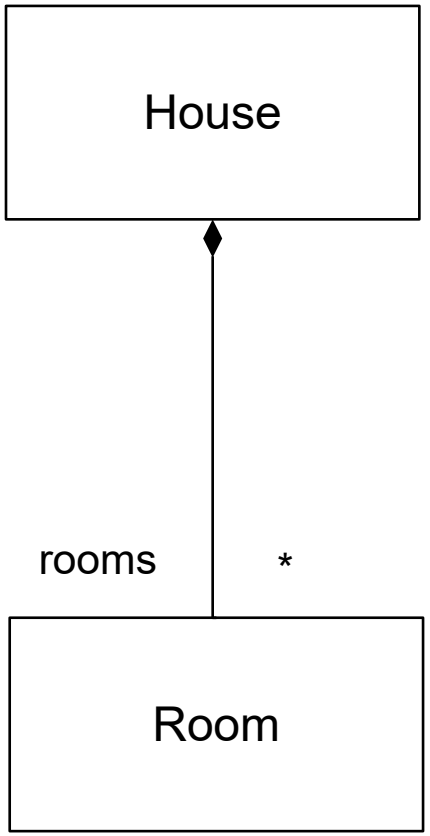
# La composition dans les langages OO

- Selon les langages, les propriétés de la composition sont plus ou moins simples à respecter et parfois cela est même impossible.
- Impossible dans les langages avec sémantique de référence et garbage collector (Java, Python…).
- Les langages avec sémantique de valeur assurent implicitement la possession de la donnée et le même cycle de vie (Rust, C++ moderne).
  - Possible en C++ ancien et sémantique de référence mais avec destructeur explicite.

# Un exemple
## d'abord en Python

*composition.py*

```
class Room:
    def __init__(self, kind, square_meters):
        self.kind = kind
        self.square_meters = square_meters

class House :
    def __init__(self):
        self.__rooms = []

    def __del__(self) :
        print('destroying house')

    def add(self, kind, square_meters):
        self.__rooms.append(Room(kind, square_meters))

    def get_bedroom(self) :
        return self.__rooms[0]
```



# Un exemple
## d'abord en Python

*composition.py*

```
def house_problem():
    hs = House()

    for i in range(4):
        hs.add('bedroom', 16+i)

    print(hs.show())
    br = hs.get_bedroom()
    print('returning {} {}'.
          format(br.kind, br.square_meters))
    return br


if __name__=='__main__':

    br = house_problem()

    print('obtained {} {}'.
          format(br.kind, br.square_meters))
```

*output*

```
In house:
• bedroom 16
• bedroom 17
• bedroom 18
• bedroom 19
returning bedroom 16
destroying house
obtained bedroom 16
```

- La variable *br* dans la fonction *main* ne devrait pas référencer un objet vu que la maison est détruite.
- Incohérence, pas facile à détecter !

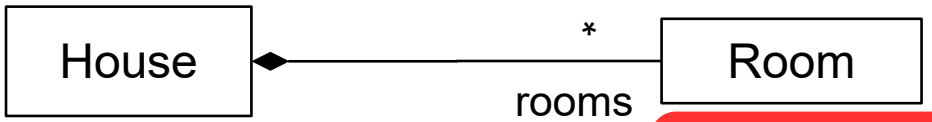
# Le même exemple
## mais en Rust

*composition.rs*

```rust
pub struct Room {
    pub kind: String,
    pub square_meters: usize,
}

pub struct House {
    rooms: Vec<Room>,
}

impl House {
    pub fn new() -> Self {
        Self { rooms: Vec::new() }
    }

    pub fn add(
        &mut self,
        kind: String,
        square_meters: usize,
    ) {
        self.rooms.push(Room {
            kind,
            square_meters,
        });
    }
```

*composition.rs*

```rust
    pub fn get_bedroom(&self)
    -> Option<&Room> {
        self.rooms.get(0)
    }
}

impl Drop for House {
    fn drop(&mut self) {
        println!("detroying house");
    }
}
```



# Le même exemple
## mais en Rust

*composition.rs*

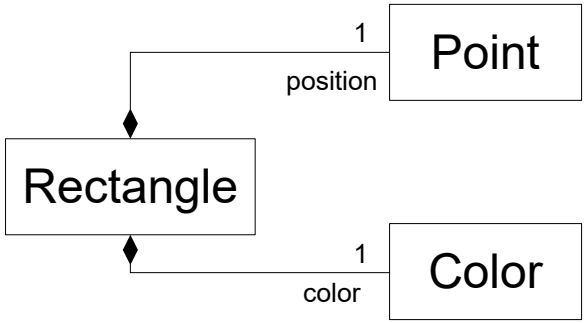
```
fn main() {
  fn house_problem() -> &Room {
      let mut hs = House::new();
      for i in 0..4 {
         hs.add("bedroom".to_owned(), 16+i);
      }

      println!("{:?}", hs);

      let br = hs.get_bedroom().unwrap();

      println!("returning {} {}",
            br.kind, br.square_meters);
      br
  }

  let br = house_problem();

  println!("obtained {} {}",
       br.kind, br.square_meters);
}
```

- L'erreur est détectée à la compilation.
- Impossible de renvoyer une référence sur une valeur que sera détruite, le compilateur est capable de le voir.
- Le C++ permet le même type de vérification mais avec des options de compilation.

# Composition : 1 composant

## composition.rs

```rust
pub struct Color {
    pub red: u8,
    pub green: u8,
    pub blue: u8,
}


pub struct Point {
    pub x: f64,
    pub y: f64,
}
```




UML

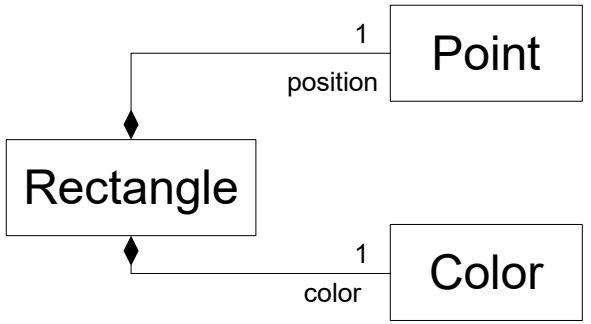
## composition.rs

```rust
pub struct Rectangle {
    position: Point,
    color: Color,
    width: f64,
    height: f64,
}

impl Rectangle {
    pub fn new(
        position: Point,
        color: Color,
        width: f64,
        height: f64,
    ) -> Self {
        Self {position, color, width, height,}
    }

    pub fn draw(&self) {
        //println!(...)
    }
}
```

# Composition : 1 composant

*main.rs*

```
fn main() {
    let red = Color {
        red: 255,
        green: 0,
        blue: 0,
    };

    let pos = Point {x: 10.0,
                     y: 9.0};

    let rect = Rectangle::new(pos,
                              red,
                              20.0,
                              30.0);

    println!("Rectangle rect");
    rect.draw();
}
```

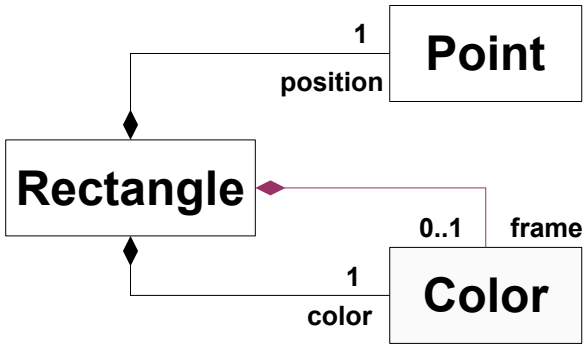



**UML**

# Composition : 0..1 composant

*composition.rs*

```
pub struct Rectangle {
    position: Point,
    color: Color,
    frame: Option<Color>,
    width: f64,
    height: f64,
}

impl Rectangle {
    pub fn new(
        position: Point,
        color: Color,
        frame: Option<Color>,
        width: f64,
        height: f64,
    ) -> Self {
        Self {position, color, frame,
              width, height,}
    }

    pub fn draw(&self) {
        //println!(...)
    }
}
```



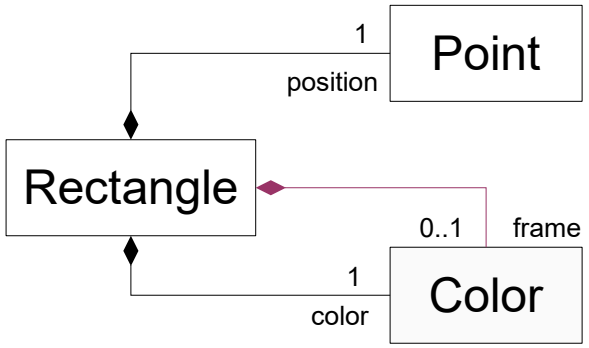

UML

# Composition : 0..1 composant

*main.rs*

```
fn main() {
  let red = Color {red: 255, green: 0, blue: 0};
  let green = Color {red: 0, green: 255, blue: 0};

  let rect1 = Rectangle::new(
                          Point{x: 10.0, y: 9.0},
                          red.clone(),
                          Some(green),
                          20.0, 30.0);

  let rect2 = Rectangle::new(
                          Point{x: 1.0, y: 9.0},
                          red,
                          None,
                          20.0, 30.0);

  println!("Rectangle rect1");
  rect1.draw();
  println!("Rectangle rect2");
  rect2.draw();
}
```

**UML**



# Rappel : Option<T>

- Rappel : en Rust il est possible de déclarer une donnée optionnelle à travers le type std::option::Option<T> ou simplement Option<T>.

  https://web.enib.fr/~harrouet/rust/rust_03_rules.html#option

- Son utilisation oblige à prendre en considération l'indisponibilité de la donnée.
- A partir d'une donnée optionnelle Option :
  - sa variante *Some* désigne la valeur qu'elle contient (si elle en contient une) ;
  - sa variante *None* indique que la donnée n'a pas de valeur.

# Exemple Option<Color> dans Rectangle

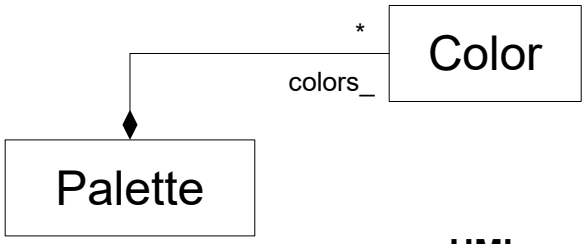
*composition.rs*

```
impl Rectangle {
    //...
    pub fn draw(&self) {
        //...
        if let Some(f) = &self.frame {
            println!("\tcolor: [{},{},{}]", f.red, f.green, f.blue);
        } else {
            println!("No frame");
        }

        //OU
        match &self.frame {
            Some(f) => {
                println!("\tcolor: [{},{},{}]", f.red, f.green, f.blue);
            }
            None => {
                println!("No frame");
            }
        }
    }
}
```

# Composition : *n* composants

### composition.rs

```
pub struct Palette {
    size: usize,
    next: usize,
    // composition à
    // plusieurs composants
    pixels: Vec<Color>,
}
```




**UML**

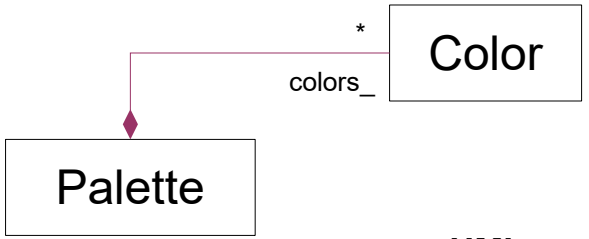
### composition.rs

```
impl Palette {
    pub fn new(size: usize) -> Self {
        Self {size, next: 0,
              pixels: vec![
                Color {red: 0,green: 0,blue: 0};
                size],
        }
    }

    pub fn add(&mut self, color: Color) {
        self.pixels[self.next] = color;
        self.next =
            (self.next + 1).rem_euclid(self.size);
    }

    pub fn draw(&self) {
        //println!(...)
    }
}
```

# Composition : *n* composants

**main.rs**

```
fn main() {
    let mut plt = Palette::new(3);
    for i in 0..4 {
        plt.add(Color {
            red: i,
            green: i,
            blue: i,
        });
    }

    println!("Palette plt");
    plt.draw();
}
```




UML

# Agrégation

- L'agrégation est une relation de type « *contient un* » ou « *contient plusieurs* ». Elle est unidirectionnelle.
- Un objet d'une classe agrégée peut être contenu par plusieurs instances de classes agrégats. (**Partageabilité**)
  - Souvent, l'agrégation est présentée comme une relation qui prétend la **non-partageabilité** (les agrégés n'appartiennent qu'un seul agrégat, c'est un débat ouvert).
  - Finalement, cela dépend de la sémantique qu'on a besoin de lui donner.
- Les cycles de vies des objets sont indépendants : un objet d'une classe continue à exister même si l'objet qui le contient est détruit et vice-versa. (**Cycle de vie indépendant**).

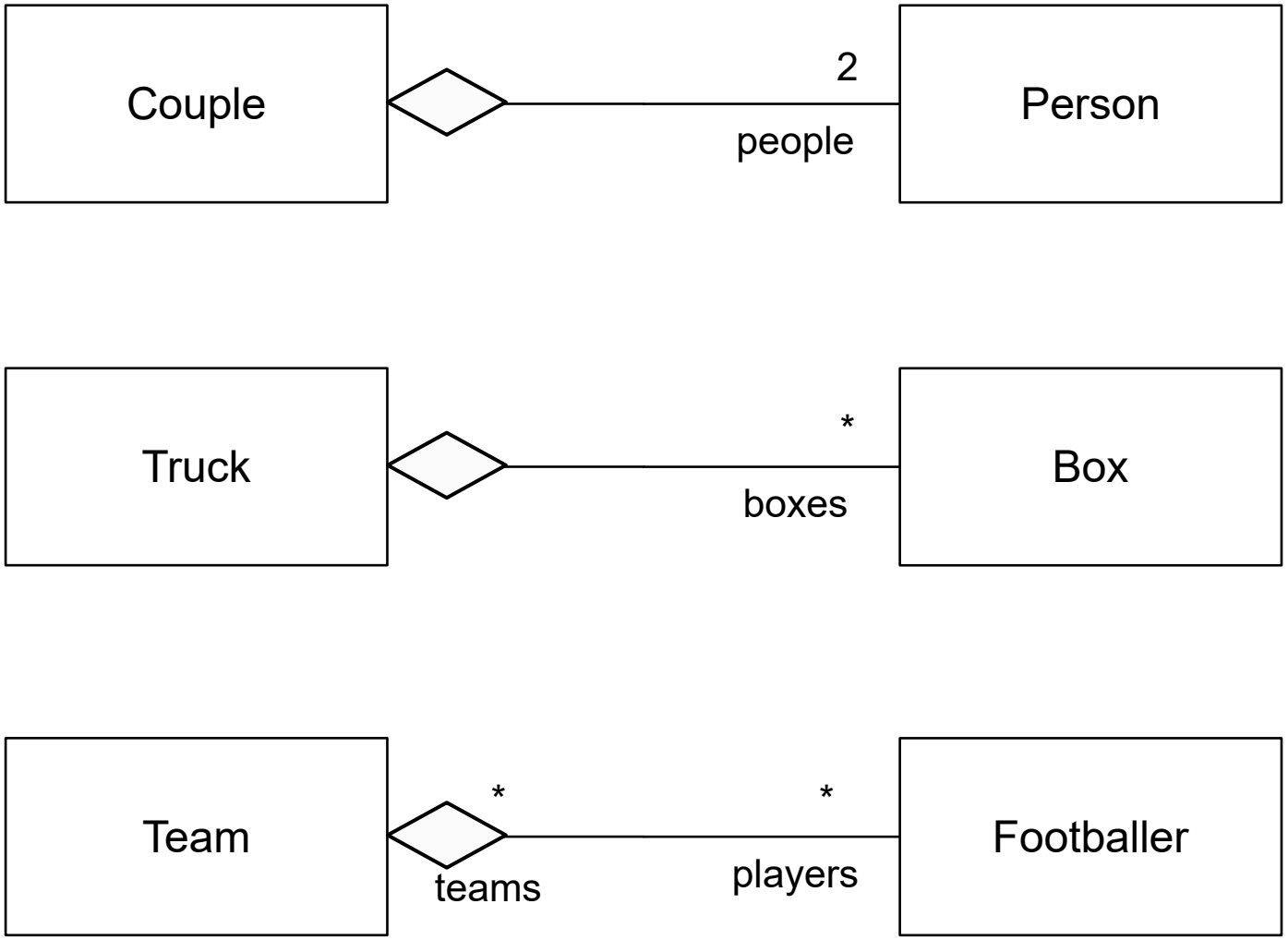
# Exemple

Un couple est une agrégation
de deux personnes


Un chariot est une agrégation
de marchandise


Une équipe de football est un agrégat de footballeur
et un footballeur peut jouer pour deux équipes

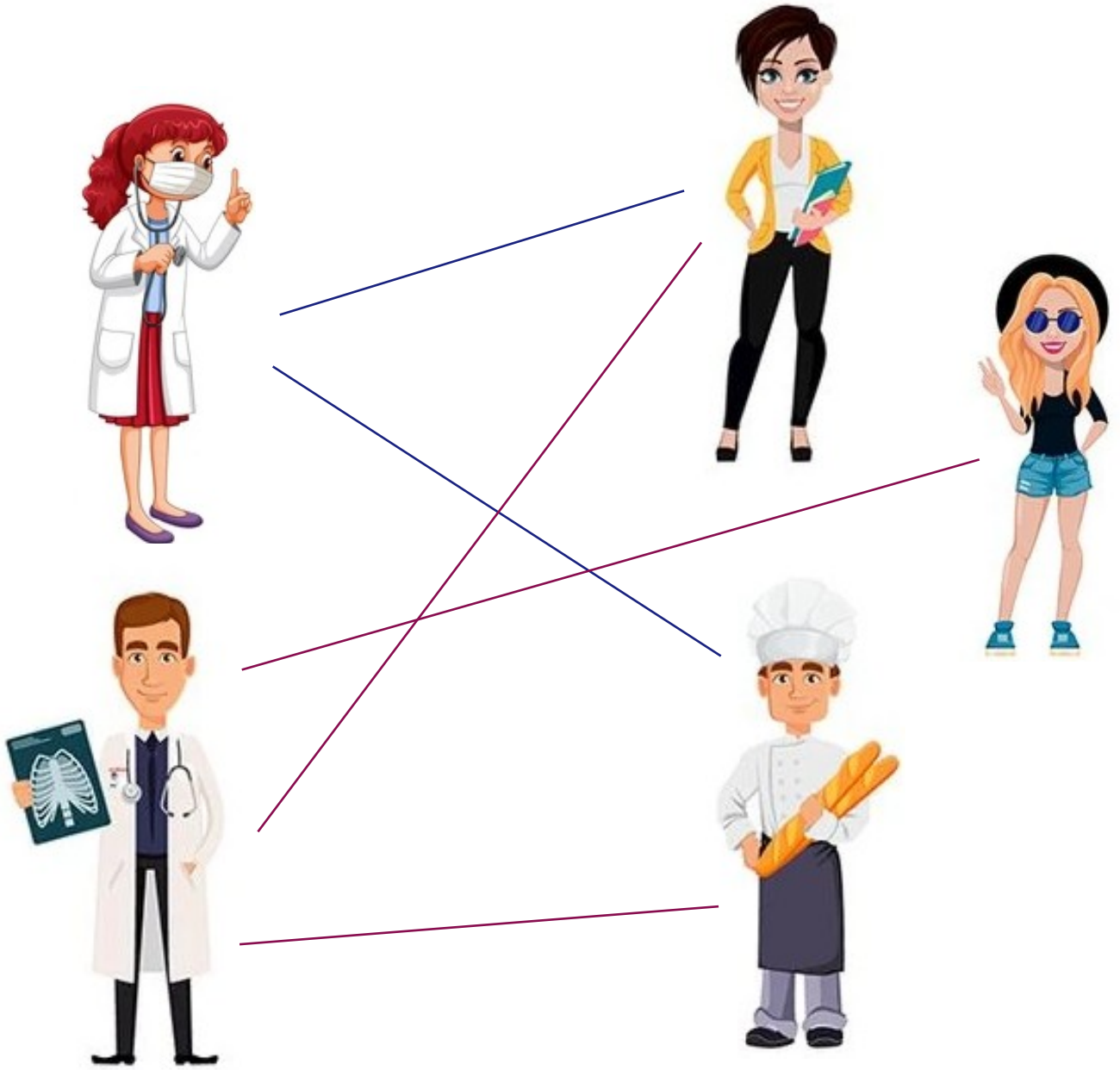
# Rappel : l'agrégation en UML



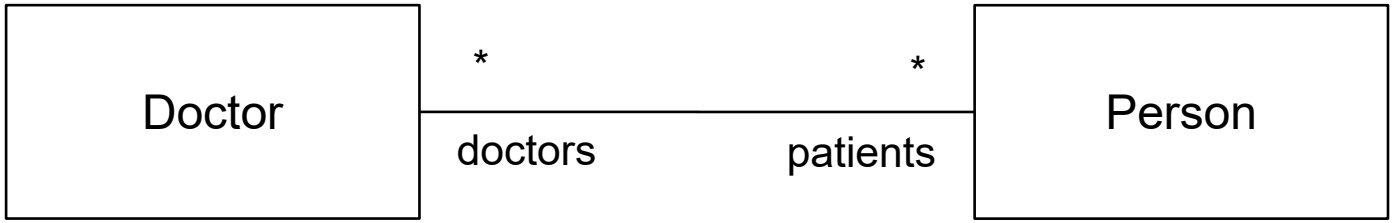
# Association

- L'association est une relation de type « *connaît un* » ou « *connaît plusieurs* ». Elle est bidirectionnelle.
- Un objet d'une classe peut être associé (et donc partagé) par plusieurs instances d'autres classes. (**Partageabilité**)
- Les cycles de vies des objets associés sont indépendants : un objet d'une classe continue à exister même si les objets à qui il est associés sont détruits et vice versa. (**Cycle de vie indépendant**).

# Exemple



# Rappel : association en UML



# Agrégation vs Association

- Propriétés de l'agrégation = propriétés de l'association.
- Et les différences alors ?
  - Sémantiques et purement conceptuelles.
  - L'agrégation implique une appartenance (faible).
  - L'agrégation est acyclique :
    - un couple peut contenir deux personnes mais une personne ne peut pas contenir un couple.
- La distinction entre les deux est souvent floue et cause de débats. La norme UML dit :

*La sémantique précise de l'agrégation partagée varie selon le domaine d'application et le modélisateur. L'ordre et la manière dont les instances de pièces sont créées ne sont pas définis (UML Superstructure version 2.1.1, pag. 41 (7.3.2)).*

# Association et Agrégation

- Association et agrégation expriment une connaissance et pas une possession et elles s'implémentent de la même façon.
- En langages de programmation, cette connaissance se concrétise par un pointeur ou une référence.
  - Celui qui associe ou agrège ne connaît que l'emplacement en mémoire de chaque donnée associée ou agrégée.
  - Il n'a aucun droit sur la création ou la destruction de la donnée.
  - Cette façon de faire ne marche correctement que si on peut assurer l'existence de la donnée pointée (référencée).

    Ceci, dans des problèmes complexes est **très difficile** et dépend fortement du langage utilisé !

# Exemple – Association
## Python

- Classe CountBytes associée à un fichier. Une instance écrit dans le fichier en comptant les caractères.

***countBytes.py***

```
class CountBytes:
    def __init__(self, file):
        self.file = file
        self.count = 0

    def write(self, txt):
        self.file.write(txt)
        self.count += len(txt)

f = open('output_py.txt', 'w')
c1 = CountBytes(f)
c1.write('ceci\n')
c1.write('cela\n')
print(c1.count)
```

- On fait le choix fort que le fichier existera toujours et la référence référencera toujours une donnée valide.


**UML**

# Validité des références

- L'interpréteur Python n'a pas moyen de vérifier que la référence est toujours valide (même problème en Java et C++).

***countBytes.py***

```
#...

f = open('output_py.txt', 'w')
c1 = CountBytes(f)
c1.write('ceci\n')
c1.write('cela\n')
print(c1.count)

f.close()      #plantage à l'exécution !

c1.write('encore\n')
print(c1.count)
```

- L'effort d'attention est tout sur le programmeur.

# Partageabilité

- La partageabilité est implicite en Python (en Java et C++ aussi), toute référence peut être partagée sans aucun contrôle.

***countBytes.py***

```
#...

f = open('output_py.txt', 'w')
c1 = CountBytes(f)
c1.write('ceci\n')
c1.write('cela\n')
print(c1.count)

c2 = CountBytes(f)
c2.write('ceci\n')

c1.write('encore\n')
print(c1.count)
```

- L'effort pour assurer la sémantique du code est tout sur le programmeur.
- Dans cet exemple, deux CountBytes référencent le même fichier :
  - Si le but est que chaque CountBytes compte ce qu'il a écrit, alors ok
  - Si le but est de compter les octets écrits dans le fichier, c'est raté.

# Et en Rust ?

*association.rs*

```
pub struct ExclusiveCountBytes<'a> {
    file: &'a mut std::fs::File,
    count: usize,
}

impl<'a> ExclusiveCountBytes<'a> {
    pub fn new(file: &'a mut std::fs::File)
    -> Self {
        Self { file, count: 0 }
    }

    pub fn write(
        &mut self,
        txt: &str,
    ) {
        self.file.write_all(
                    txt.as_bytes()).unwrap();
        self.count += txt.len();
    }

    pub fn count(&self) -> usize {
        self.count
    }
}
```

- 'a → *lifetime* : explicite au compilateur que la durée de vie de la donnée référencée doit être au moins la même que celle de l'objet qui la référence.
- Ce n'est pas optionnel, c'est obligatoire : le prix à payer pour que notre idée soit respectée.
- Cela est vérifié à la compilation, le code ne compile pas si ce n'est pas respecté.

# Durée de vie d'une donnée référencée

*association.rs*

```
fn main() {
    let mut f = std::fs::File::create(
        "output_rs_e.txt").unwrap();
    let mut c1 = ExclusiveCountBytes::new(&mut f);
    c1.write("ceci\n");
    c1.write("cela\n");
    println!("{}", c1.count());

    drop(f);    //erreur à la compilation !

    c1.write("encore\n");
    println!("{}", c1.count());
}
```

- La fonction drop() détruit la valeur de f.
- c1 est encore utilisé après l'appel à drop() donc il garde toujours la référence sur le fichier
- À la compilation, le compilateur nous informe que f ne peut pas être détruit vu qu'il est encore référencé par c1 après.

# Rust joue la prudence !

## Par défaut : non-partageabilité

*association.rs*

```
fn main() {
    let mut f = std::fs::File::create(
         "output_rs_e.txt").unwrap();
    let mut c1 = ExclusiveCountBytes::new(&mut f);
    c1.write("ceci\n");
    c1.write("cela\n");
    println!("{}", c1.count());

    //erreur à la compilation !
    let mut c2 = ExclusiveCountBytes::new(&mut f);
    c2.write("autre chose\n");

    c1.write("encore\n");
    println!("{}", c1.count());
}
```

- Pour assurer l'intégrité d'une donnée, Rust n'y autorise pas l'accès tant qu'elle est déjà référencée par une référence exclusive (*mutable*).
- Le compilateur nous informe que la variable c2 ne peut pas contenir une référence sur f parce qu'elle est déjà contenue par c1.

# Et si on voulait la partageabilité ?

- En Rust, il est interdit d'avoir une référence sur une donnée si celle-ci est déjà référencé par une autre référence exclusive (mutable).
  - Dans ExclusiveCountByte, file est une référence exclusive (mutable).
- En effet, le compilateur est incapable de prouver qu'il n'y aura pas d'accès à une donnée qui est déjà en train d'être modifiée.
- La partageabilité s'obtient seulement en s'appuyant explicitement sur la notion d'exclusivité dynamique (*interior-mutability*). https://web.enib.fr/~harrouet/rust/rust_04_values.html#int_mut
  - On n'accède aux données que par références partagées (non-mutables).
  - Une vérification est faite au *runtime* lorsque la donnée est effectivement visitée (en lecture ou en modification)
    - **panic!()** si une référence exclusive (mutable) coexiste avec une autre référence.

# Et si on voulait la partageabilité ?

*association.rs*

```
pub struct SharedCountBytes<'a> {
  file: &'a std::cell::RefCell<std::fs::File>,
  count: usize,
}

impl<'a> SharedCountBytes<'a> {
  pub fn new(file: &'a std::cell::
          RefCell<std::fs::File>) -> Self {
      Self { file, count: 0 }
  }

  pub fn write(
      &mut self,
      txt: &str,
  ) {
      self.file.borrow_mut().write_all(
             txt.as_bytes()).unwrap();
      self.count += txt.len();
  }

  pub fn count(&self) -> usize {
      self.count
  }
}
```

- En Rust, l'exclusivité dynamique s'obtient à travers le type **std::cell::RefCell<T>**.
- L'accès à la donnée dans la cellule est possible avec les fonctions :
  - **.borrow()** : accès partagé (non-mutable), pour consulter ;
  - **.borrow_mut()** : accès exclusif (mutable), pour modifier.

# Voilà la partageabilité !

*association.rs*

```
fn main() {
    let f = std::cell::RefCell::new(
        std::fs::File::create(
                "output_rs_s.txt").unwrap(),
    );
    let mut c1 = SharedCountBytes::new(&f);
    c1.write("ceci\n");
    c1.write("cela\n");
    println!("{}", c1.count());

   let mut c2 = SharedCountBytes::new(&f);
    c2.write("autre chose\n");

    c1.write("encore\n");
    println!("{}", c1.count());
}
```

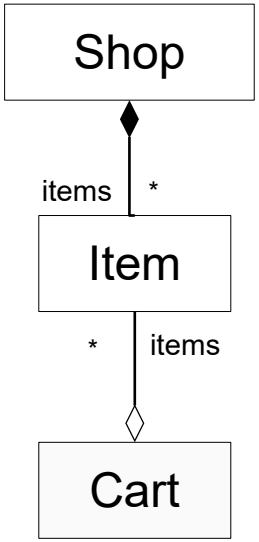
- Bref : en Rust, la partageabilité ne s'obtient pas par hasard.
- C'est un choix fort voulu par le programmeur :
  - il a dû **explicitement** la coder en gardant bien en tête la sémantique recherchée.
- Ceci permet d'éviter des erreurs de conception.

# Avec des problèmes plus complexes ?

- L'exemple montré, CountBytes, est extrêmement trivial : deux classes, une association unidirectionnelle avec multiplicité 1.
- Avec des codes plus complexes, garantir la cohérence des données (notamment des références) devient impossible.
- Dans des langages peu regardants, l'association telle qu'elle est décrite dans les livres et utilisant des références/pointeurs, est implémentable mais reste source d'erreurs (souvent difficiles à détecter parce qu'elles ne surviennent qu'au *runtime*).
- En Rust, qui doit prouver l'intégrité des données à la compilation, l'écriture d'une telle ambiguïté est impossible.

# Exemples de la complexité de l'association

- Un autre exemple un peu moins trivial, mais pas difficile non plus…
- Dans un langage à sémantique de valeur (tel que le C++), le vecteur d'items d'un Shop est déplacé dans la mémoire dès qu'il grandit.
  - Toutes les pointeurs aux mêmes items dans un Cart pointent dans le vide.
  - Solution : faire comme dans les langages à sémantique de références (Python, Java...), le vecteur d'items de Shop contient des pointeurs aux items alloués dynamiquement.
  - Conséquence : les données ne bougent plus **mais** sont éparpillées dans la mémoire.
  - **Et on réintroduit le risque d'incohérence** (acheter deux fois le même objet) et toute l'attention repose à nouveau sur le programmeur.



# Alors, quoi faire ?

- Éviter à tout prix les associations/agrégations implémentées avec des attributs de type référence/pointeur.
- Identifier plutôt l'associé à l'aide d'un code et le rechercher quand nécessaire. Par exemple :
  - Les items de Shop ont un code (penser aux dictionnaires).
  - Les items de Cart ne sont que des codes et quand on a besoin de récupérer l'item correspondant (pour le supprimer après achat, par exemple), on le recherche.
    - La recherche dans les dictionnaires est optimisée.
    - C'est comme ça que marchent les bases de données.
  - Exemple dans shop.rs.

# Langages Orientés Objet

*Collaborations*
*Composition, Agrégation et Association*

Elisabetta Bevacqua