LE LANGAGE C++
MASTER 1
DU MODÈLE AUX CLASSES

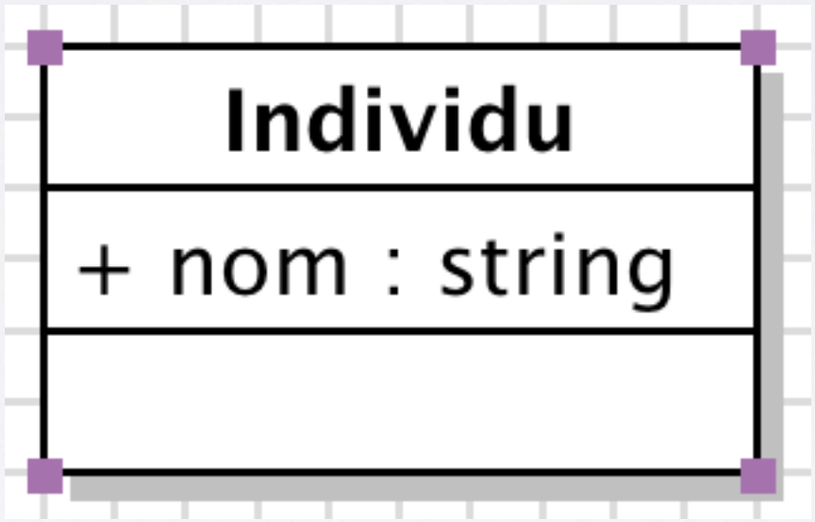
La classe comme réalisation du modèle

Jean-Baptiste.Yunes@u-paris.fr
U.F.R. d'Informatique
Université de Paris

11/2021

LES ATTRIBUTS

- Un Individu dont on imaginerait qu'il aurait comme attribut un nom (représenté sous forme de chaîne de caractères) serait décrit en notation UML :



- L'écriture C++ pourrait être :

```
class Individu {
  public:
    string nom;
};
```

```
Individu i;
i.nom = "Anselme";
cout << "Il s'appelle " << i.nom << endl;
```

- Inconvénients d'une telle écriture :
  - dépendance forte introduite avec le code utilisateur (de la classe)
  - pas d'uniformité syntaxique avec les autres services

- Une meilleure écriture C++ pourrait être :

```
class Individu {
  private:
    string nom;
  public:
    string getNom();
    void setNom(string n);
};
```

```
string Individu::getNom() { return nom; }
void Individu::setNom(string n) { nom = n; }
```

```
Individu i;
i.setNom(“Anselme“);
cout << “Il s’appelle “ << i.getNom() << endl;
```

- Pas de dépendance vis-à-vis de la représentation

- Une meilleure écriture C++ pourrait être :

```
class Individu {
  private:
    string leVraiNom;
  public:
    string getNom();
    void setNom(string n);
};
```

```
string Individu::getNom() { return leVraiNom; }
void Individu::setNom(string n) { leVraiNom = n; }
```

```
Individu i;
i.setNom("Anselme");
cout << "Il s'appelle " << i.getNom() << endl;
```

- Les modifications de la représentation n'affectent que le concepteur de la classe...

- Une meilleure écriture C++ pourrait être :

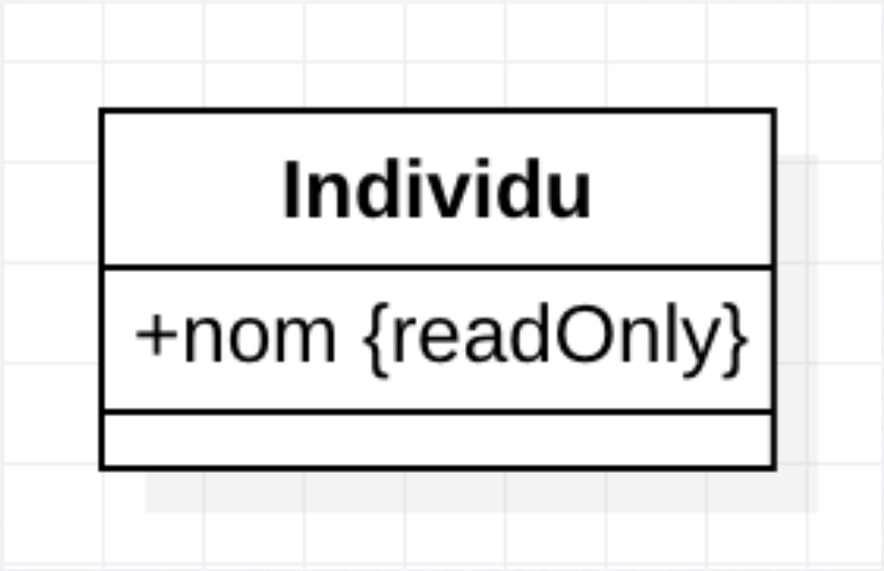
```
class Individu {
  private:
    char *nom;
  public:
    string getNom();
    void setNom(string n);
};
```

```
string Individu::getNom() { return string(nom); }
void Individu::setNom(string n) { nom = n.c_str(); }
```

```
Individu i;
i.setNom("Anselme");
cout << "Il s'appelle " << i.getNom() << endl;
```

- Les modifications de la représentation n'affectent que le concepteur de la classe...

- Une bonne habitude consiste donc à cacher la représentation de l'attribut et d'offrir des services permettant de le manipuler
- Les services `get`*Attribut*`()` et `set`*Attribut*`()` sont appelées des accesseurs (*accessors*) : `get`ters et `set`ters
- Un attribut en lecture seule :



- Une bonne écriture C++ pourrait être :

```
class Individu {
  private:
    string nom;
  public:
    string getNom() const;
};
```

```
string Individu::getNom() const { return nom; }
```

```
Individu i;
cout << “Il s’appelle “ << i.getNom() << endl;
```

- L’utilisateur n’a pas accès directement au service de modification de l’attribut : soit parce que ce service n’existe pas, soit parce qu’il est caché!

- Une (bien) meilleure écriture C++ pourrait être :

```
class Individu {
  private:
    const string nom;
  public:
    const string getNom() const;
};
```

```
const string Individu::getNom() const { return nom; }
```

```
Individu i;
cout << “Il s’appelle “ << i.getNom() << endl;
```

- Un service caché :

```
class Individu {
  private:
    string nom;
    void setNom(string n);
  public:
    string getNom() const;
};
```

```
string Individu::getNom() const { return string(nom); }
void Individu::setNom(string n) { nom = n; }
```

```
Individu i;
cout << "Il s'appelle " << i.getNom() << endl;
```

- Les services cachés sont très utiles! Si ce n'est pour l'utilisateur (qui ne les voit pas), c'est pour le concepteur de la classe qui bénéficie alors de leur existence, mais nécessite de la discipline...

- Un service caché :

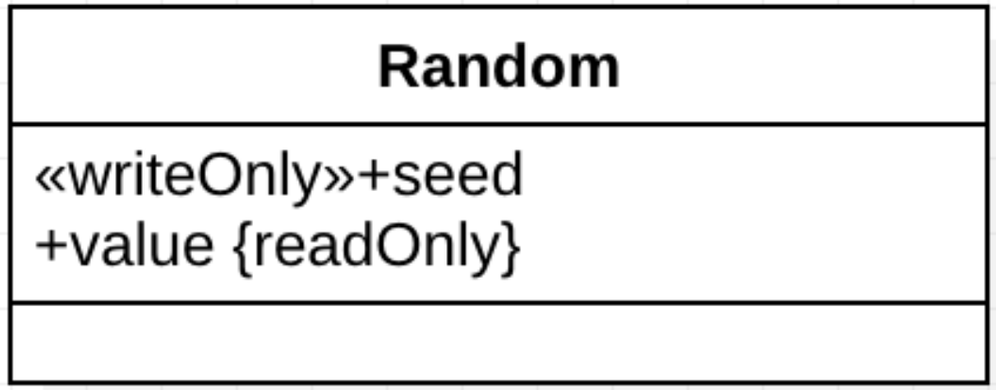
```
class Individu {
  private:
    string nom;
    void setNom(string n);
  public:
    Individu(string n);
    string getNom();
};
```

```
Individu::Individu(string n) { setNom(n); }
string Individu::getNom() { return string(nom); }
void Individu::setNom(string n) { nom = n; }
```

```
Individu i(“Alphonsine“);
cout << “Il s’appelle “ << i.getNom() << endl;
```

- On pourrait imaginer implémenter un attribut en écriture seule…

```
class Random {
private:
  uint32_t current;
public:
  Random();
  void setSeed(uint32_t s);
  uint32_t getValue();
};
```



- Un attribut en écriture seule

```
Random::Random() { current = 0; }
void Random::setSeed(uint32_t s) { current = s; }
uint32_t Random::getValue() {
  uint32_t c = current;
  current = (1103515245*current+12345)%(1<<31);
  return c;
}
```

```
int main() {
  Random r;
  for (int i=0; i<10; i++) std::cout << r.getValue() << std::endl;
  r.setSeed(11);
  for (int i=0; i<10; i++) std::cout << r.getValue() << std::endl;
}
```

L'ASSOCIATION

- L'association ordinaire…
  - Exemple : le cas de la représentation du *lien* entre une personne et un vol

  - une personne peut avoir réservé une place dans plusieurs vols, dans chaque cas, elle sera considérée comme un passager
  - à un vol donné peut être associé un nombre quelconque (voire nul) de passagers

- L'implémentation d'une association ordinaire...

```
class Personne {
  private:
    Vol **vols;
    int nVols;
  public:
    void sEnregistreDansLeVol(Vol &);
    bool estPassagerDuVol(const Vol &);
};
```

```
void Personne::sEnregistreDansLeVol(Vol &leVol) {
  if (this->estPassagerDuVol(leVol)) return;
  add(vols,nvols,leVol);
  leVol.prendUnPassagerEnPlus(*this);
}
```

- L'implémentation d'une association ordinaire…

```
class Vol {
  private:
    Personne **passagers;
    int nPassagers;
  public:
    void prendUnPassagerEnPlus(Personne &);
    bool aPourPassager(const Personne &);
};
```

```
void Vol::prendUnPassagerEnPlus(Personne &lePassager) {
  if (this->aPourPassager(lePassager)) return;
  add(passagers,nPassagers,lePassager);
  lePassager.sEnregistreDansLeVol(*this);
}
```

- En général les méthodes permettant de manipuler une relation portent des noms conventionnels :
  - `void addPassager(Personne lePassager);`
  - `void removePassager(Personne lePassager);`
  - `bool isPassager(Personne lePassager);`
  - `Personne getPassagerAt(int i);`
  - `int getNumberOfPassagers();`
  - `Personne *getPassagers();`
  - ...

- La vie de la relation est compliquée. Des contraintes fonctionnelles peuvent exister comme celle devant assurer la symétrie de la relation
  - comme dans notre exemple… où l'on observe que le moyen de s'en sortir est un peu étrange. Il est difficile à la lecture du code de l'une des classes de comprendre que la contrainte est assurée…
- Une façon de s'en sortir est de donner plus de chair à la relation

LA CLASSE D'ASSOCIATION

- L'implémentation d'une association ordinaire... Une relation qui prend forme...

```
class Personne {
  private:
    const Vol *vols;
    int nVols;
    void addVol(const Vol &);
  friend void addVolPassager(Personne &,Vol &);
};
class Vol {
  private:
    const Personne *passagers;
    int nPassagers;
    void addPassager(const Personne &);
    friend void addVolPassager(Personne &,Vol &);
};
```

```
void addVolPassager(Personne &unPassager,Vol &unVol) {
  unPassager.addVol(unVol);
  unVol.addPassager(unPassager);
}
```

- L'étape suivante consiste alors simplement à donner une existence réelle à la relation...

```
class Personne {
  private:
    static VolPassager *volPassager;
  public:
    void addVol(const Vol &) const;
};

class Vol {
  private:
    static VolPassager *volPassager;
  public:
    void addPassager(const Personne &) const;
};
```

À partager!!!

```
void Vol::addPassager(const Personne &unPassager) const {
  volPassager->add(*this,unPassager);
}
void Personne::addVol(const Vol &unVol) const {
  volPassager->add(unVol,*this);
}
```

- L'étape suivante est de tout externaliser...

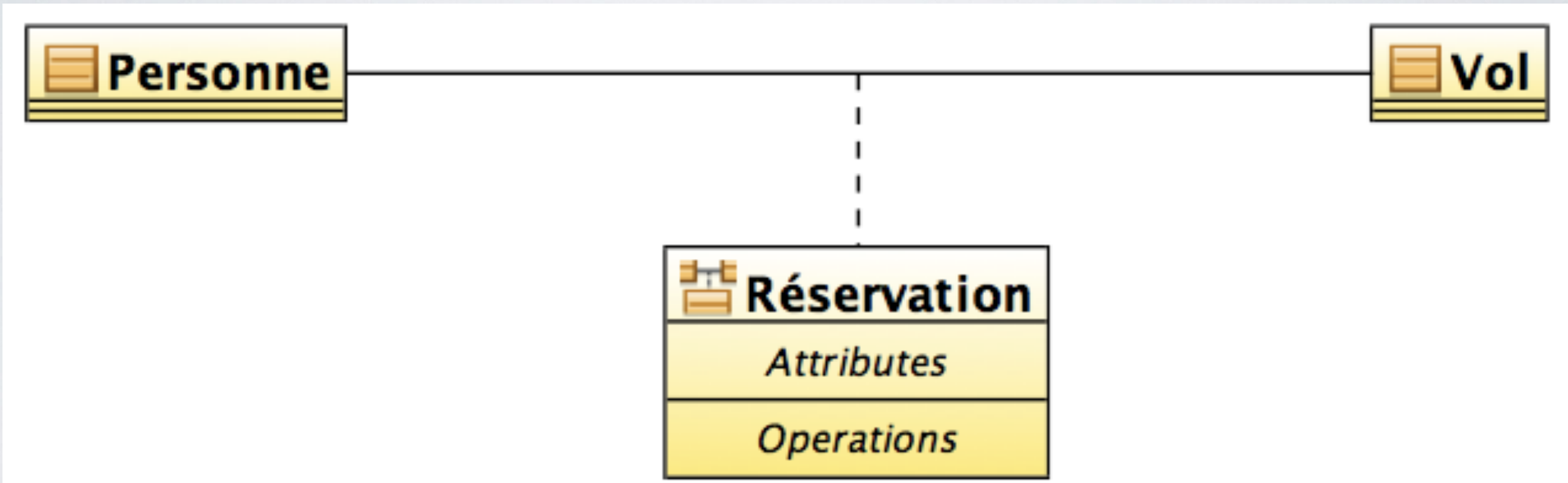
```
class Personne {
};
class Vol {
};
class RelationVolPassager {
  public:
    static void add(const Vol &,const Personne &);
};
```

- OU

```
class Personne {
  void addVol(const Vol &v) { RelationVolPassager::add(v,*this); }
};
class Vol {
  void addPassager(const Personne &p) { RelationVolPassager::add(*this,p); }
};
class RelationVolPassager {
  friend class Vol;
  friend class Personne;
  private:
    static void add(const Vol &,const Personne &);
};
```

- Il n'y a pas de raison pour que les objets concernés aient quoi que ce soit à connaître de la relation en ce qui concerne leur propre structure…
- C'est une vision extrémiste. Quoique très réaliste.

- L'écriture UML correspondante d'une classe d'association est :



La ville comme agrégat

PAR ANOUCHKA KACZMAREK, 2011/03/13

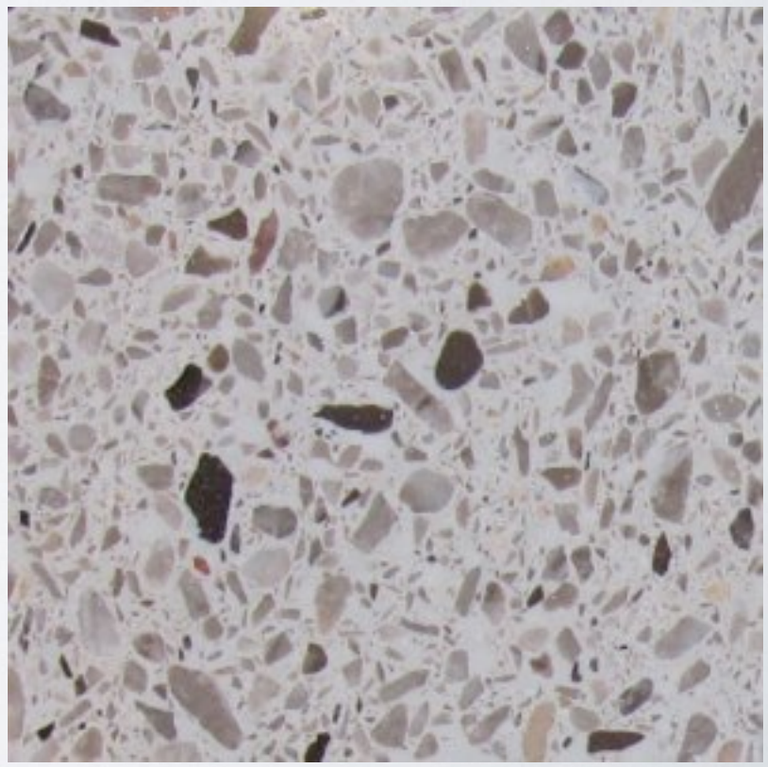
Agrégat: objet résultant de la réunion d'un ensemble d'éléments

Agréger: réunir de manère à créer un **ensemble**

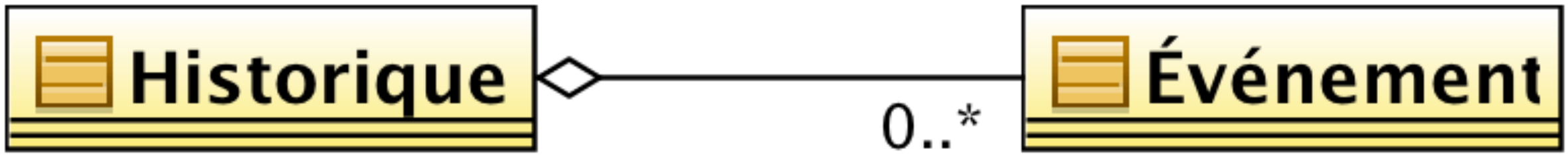
(chimie) Agrégat atomique: ensemble d'atomes liés de façon suffisamment étroite pour avoir des **propriétés spécifiques**

-> la ville comme agrégat, comme juxtaposition d'éléments formant un tout avec des caractéristiques spécifiques

L'AGREGATION



- L'agrégation est une relation particulière, une certaine asymétrie y est présente sous la forme d'une relation de type maître/esclave...



- Un historique est une collection d'événements, et c'est même sa seule raison d'être

- Il n'y a pas de nécessité à ce qu'un événement sache qu'il appartient à un historique

- Une réalisation C++ possible serait :

```
class Evenement {
};

class Historique {
  private:
    const Evenement **lesEvenements;
    int nEvenement;
  public:
    void addEvenement(const Evenement &);
    void removeEvenement(const Evenement &);
    Evenement *getEvenementAt(int);
    ...
};
```

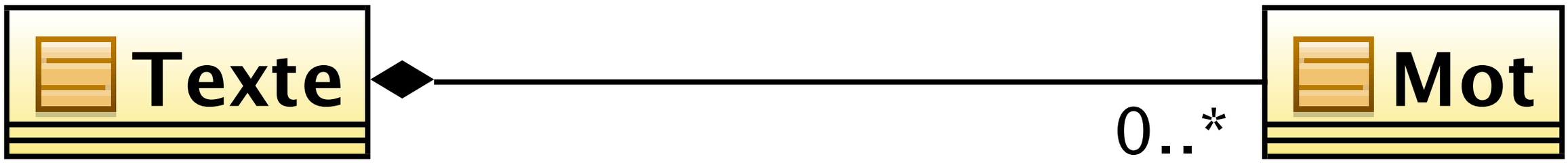
- où l'on constate que la représentation est similaire à celle d'une simple association... excepté l'asymétrie...

- La raison principale est que le langage est trop pauvre pour exprimer de façon particulière ce type de relation
- Ou alors, la distinction est si subtile qu'il n'y a pas nécessité d'introduire une construction particulière dans le langage
- Note : Il est bien difficile de faire de l'ingénierie inverse (reverse engineering), *i*.e. remonter du code jusqu'au modèle (en descendant on y perd)...
    - les langages de programmation sont en général plus pauvres sémantiquement qu'UML
        - besoin de documentation

LA COMPOSITION

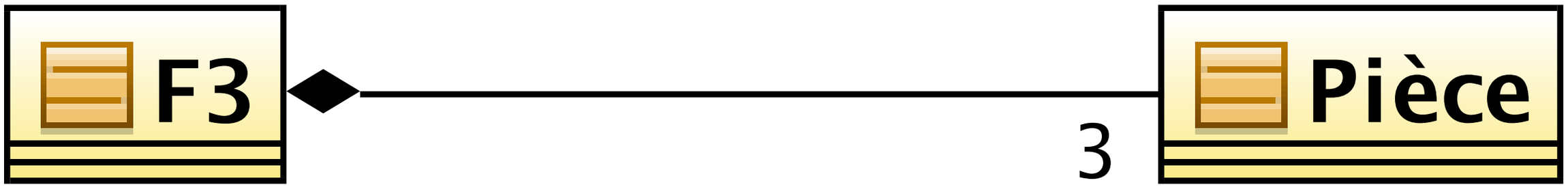
- Lorsque l'agrégat contrôle le cycle de vie des objets agrégés, on a sans doute affaire à une composition. Le symbole côté agrégat est alors rempli de noir.

- Un exemple type de composition est :



- Un texte n'est rien d'autre qu'un ensemble de mots
- Il n'est pas nécessaire qu'un mot connaisse le texte auquel il appartient
- le texte disparaît ? les mots aussi… (c'est sûr ?)

- Lorsque la composition n'est pas une structure dynamique, comme dans :



- on peut envisager une véritable composition, surtout si les composants ne sont qu'affaire de structuration interne...

- Son implémentation C++ pourrait être :

```
class Piece {};

class F3 {
 private:
   Piece pieces[3];
};
```

- Il est clair que l'on ne souhaite pas créer de pièce sans maison, ni l'inverse… Leurs vies sont liées…

- Il s'agit d'une structuration interne, on peut donc utiliser la construction adéquate du langage

- Soit

```
class F3 {
  private:
    class Piece {};
    Piece pieces[3];
};
```

- Et la généralisation/spécialisation ?
  - On y reviendra…