28

Classes à sémantique d'entité

Précédemment, nous avons appris à créer des objets respectant la sémantique de valeur. Nous allons voir maintenant une autre sémantique majeure du C++ qu'on appelle la *sémantique d'entité*. Elle est à contre-pied de la sémantique de valeur, mais comme cette dernière permet de se simplifier la vie et d'éviter des erreurs. Pour cela, nous allons voir ce qu'elle signifie, les conséquences sur le code, dans quels cas l'utiliser et les réflexes à avoir.

28.1. Des objets uniques !

Dire qu'une classe est à sémantique d'entité veut dire que les instances de cette classe sont des entités.

Une entité est un objet tel qu'on le conçoit dans la vie réelle. Deux entités ont beau avoir exactement les mêmes caractéristiques, elles ne sont pas les mêmes. Par exemple, si vous avez un stylo identique à celui de votre voisin, cela reste bien deux stylos distincts. En termes de programmation, si on implémente une classe `Stylo`, deux instances dans le même état (c'est-à-dire ayant exactement les mêmes attributs, comme la couleur ou le niveau d'encre) seront chacune uniques et ne représenteront en aucun cas le même objet.

En particulier, chaque instance va vivre sa vie indépendamment. C'est logique, la quantité d'encre dans votre stylo peut changer, ça n'altérera pas le stylo du voisin. La sémantique d'entité est donc radicalement différente de la sémantique de valeur, où deux instances identiques représentent forcément la même valeur.

Conséquences sur l'interface

Quand on écrit une classe à sémantique d'entité, il y a trois choses à prendre en compte :

- on ne surcharge plus les opérateurs arithmétiques (`+` , `-` , etc.) ;
- on ne surcharge plus les opérateurs de comparaison ;
- on n'utilise plus la copie.

La première règle est assez évidente. Ça n'a pas de sens de vouloir soustraire ou multiplier des chaises, des stylos ou des comptes en banque.

La deuxième provient de ce que chaque instance est unique. Dès lors, à quoi bon les comparer dans leur ensemble puisqu'elles ne seront jamais égales. En revanche, on peut très bien les comparer en utilisant certains attributs à sémantique de valeur. Par exemple, on peut comparer des comptes en banque suivant leur montant, des chaises en fonction de leur poids, etc.

La troisième règle est aussi liée à l'unicité de chaque instance. On ne peut pas créer une copie à partir d'un autre objet. On ne peut pas copier un personnage vers un autre, ou bien copier une lampe vers une autre. Cette raison est philosophique, mais nous allons voir en fin de chapitre que cela a des *conséquences pratiques* et que vouloir implémenter la copie d'entité est une source sans fin de bugs vicieux.

Pour être en mesure de désactiver la copie, on utilise le mot clé `delete` sur le constructeur de copie et l'opérateur d'affectation par copie, comme nous l'avons déjà vu pour le constructeur par défaut. On se protège ainsi de la copie, qu'elle soit explicite (`Entite b { a };`) ou implicite (appel à `push_back` par exemple).

```
class NonCopiable
{
public:
  NonCopiable(NonCopiable const & copie) = delete;
  NonCopiable& operator=(NonCopiableconst & copie) = delete;
};
```

La visibilité n'a dans ce cas aucune importance. Que le constructeur et l'opérateur de copie soient `public` ou `private`, la copie est désactivée dans les deux cas.

Un exemple

Supposons que vous écriviez un programme qui émule le fonctionnement d'une entreprise. Un besoin possible est de représenter les personnes qui y travaillent. Des personnes humaines sont un cas évident de sémantique d'entité. Un début de classe pourrait donc ressembler à cela.

```
#include <cassert>
#include <iostream>
#include <string>

class Personne
{
public:
```

```
  Personne() = delete;
  Personne(std::string const& nom, std::string const& prenom, int
 salaire);

  Personne(Personne const& copie) = delete;
  Personne& operator=(Personne const& copie) = delete;

  void travailler() const;

private:
  std::string m_nom;
  std::string m_prenom;
  int m_salaire;
};

Personne::Personne(std::string const& nom, std::string const&
 prenom, int salaire)
  : m_nom(nom), m_prenom(prenom), m_salaire(salaire)
{
  assert(!std::empty(m_nom) && "Le nom de famille est obligatoire.");
  assert(!std::empty(m_prenom) && "Le prénom est obligatoire.");
  assert(m_salaire >= 0 && "C'est vous qui payez la personne, pas
 l'inverse !");
}

void Personne::travailler() const
{
  std::cout << "Je suis " << m_prenom << " "
    << m_nom << " et j'effectue différentes tâches.\n";
}

using namespace std::literals;

int main()
{
  Personne const patron { "Ron"s, "Pat"s, 2500 };
  patron.travailler();

  // Pauvre Bob...
  Personne const stagiaire { "Le Stagiaire"s, "Bob"s, 0 };
  stagiaire.travailler();
  return 0;
}
```

```
Je suis Pat Ron et j'effectue différentes tâches.
Je suis Bob Le Stagiaire et j'effectue différentes tâches.
```

Cet exemple de personne et d'entreprise va servir de fil rouge pour ce chapitre et illustrera très bien ce à quoi servent les entités et comment les manipuler.

Note › *Les entités en C++ ne correspondent pas nécessairement à des entités de la vraie vie. Dans ce premier exemple, c'est le cas, mais nous verrons plus bas un cas où nos entités ne sont pas des objets réels.*

28.2. Des sous-types…

Dans une entreprise, tout le monde n'occupe pas le même poste. Par conséquent, il y a plusieurs types de travailleurs. Bien qu'ils soient différents, ils n'en restent pas moins tous des personnes. Il serait intéressant de pouvoir représenter cette hiérarchie. C++ offre justement une façon de représenter une relation EST-UN et qu'on appelle *l'héritage*. Grâce à cette technique, nous pouvons créer des sous-types plus spécialisés de nos entités de base.

Exemple

Nous pourrions commencer en définissant un type `Directeur`, qui sera une spécialisation (un sous-type) de `Personne`.

```
class Personne
{
//...
};

// Directeur hérite de Personne.
// Directeur est un sous-type de Personne.
class Directeur : public Personne
{
//...
};
```

En plus des propriétés que le sous-type a en commun avec sa classe parent, on peut aussi lui *ajouter des services propres*. Dans le cas d'une chaise, s'asseoir est un service commun à tous les types de chaises, mais rouler devient un service spécifique aux chaises à roulettes, se balancer aux chaises à bascule, etc. L'exemple suivant montre par exemple que notre nouveau type `Directeur`, en plus de travailler, pourra assister à des réunions. En effet, un directeur EST-UNE personne, mais pas que.

```
class Directeur : public Personne
{
public:
  void assister_aux_reunions() const;
};

void Directeur::assister_aux_reunions() const
{
  std::cout << "Des décisions importantes doivent être prises
 aujourd'hui !\n";
}
```