Modélisation d'un objet

Cours pris en note par `Thomas Peugnet`.

Instanciation

Le processus de création d'un objet à partir de sa classe est appelée **l'instanciation**. A partir d'une classe, on peut instancier autant d'objet similaires.

Un objet n'est l'instance que d'une seule classe. L'héritage n'a rien à voir la dedans. La vrai classe d'un objet est "celle qui est tout en bas".

Exemples

```cpp
// Exemple de classe :
class Human
{
    std::string name_;
    float size_;
    unsigned int birth_year_;

    unsigned age();
    void hello();
};
// Ce qui va creer la classe Human
```

Ce qui, en UML, correspond à :

```
name : string
size : float
birth_year : unsigned
age() : unsgined
hello() : void
```

Il est indiqué de ne pas se préoccuper des subtilités du `C++` pour le moment.

Les fonctions `hello` et `age` permettent un comportement dynamique au sein de la classe.

Pour déclarer une nouvelle variable/objet de type `human`, il sera nécessaire d´écrire :

```cpp
Human h1 = Human ("Alain Terieur", 1.80, 1970)
```

Cycle de vie d'un objet

- Dynamique comme toute autre valeur: On créé, on utilise, puis on détruit.
- La Construction et la Destruction sont deux phases très importantes dans la vie d'un objet.

Construction

Elle repond a deux problematiques:

- Vision atomique de la phase de création. Bien fabriquer l'objet en entier.
- Assurer la cohérence interne (aspect agrégatifs). Verifier que l'objet ai du sens.

Le constructeur est une procédure spéciale. (Pas de type de retour).
La construction est uniquement constituée des effets de bord.

Ils sont fournis par défaut (mais modifiable). Les noms des constructeurs sont prédéfinis.

Exemple C++

Définition:

```cpp
Human::Human (const std::string& name, gender g, unsigned birth_year)
: name_ (name), gender_ (g), birth_year_ (birth_year)
{}
```

Instanciation:

```cpp
// Pour la pile
Human h1 = Human ("Alain Terieur", gender::male, 1970);
Human h2 ("Alain Terieur", gender::male, 1970);
Human h3 { "Alain Terieur", gender::male, 1970 };
Human h4 = Human { "Alain Terieur", gender::male, 1970 };

// Pour le tas
Human* h5 = new Human ("Corinne titgoutte", gender::female, 1990);
auto h6 = std::make_unique<Human>
  ("Justine Titgoutte", gender::female, 1995);
```

Exemple Java

Définition:

```
class Human
{
  String name;
  gender gender;
  int birthYear;
  Human (String _name, Gender _gender, int _birthYear)
  {
    name = _name;
    gender = _gender;
    birthYear = _birthYear;
  }
}
```

Instanciation:

```
Human h1 = new Human ("Alain Terieur", Gender.male, 1970);
```

Destruction

Un destructeur doit:

- Etre atomique, avoir lieu sans interuption
- Assurer le nettoyage (libération de la memoire)

La destruction est formalisée dans les langages de POO classique.
Le concept de destructeur n'a de sens uniquement lorsque l'on gère la mémoire à la main.

Un destructeur n'a pas de valeur de retour. Il est fourni par défaut (mais modifiable). Celui par défaut va juste nettoyer la mémoire.

Exemple C++

Définition:

```
Human::~Human()
{}
```

Appel:

```
delete h5;
```

Attributs de classe

Ce sont des variables propres à la classe et non à l'objet.

L'accès ne nécessite pas l'existence d'une instance.

On appelle ca `static` en c++ et java.

```
class Human
{
  static unsigned population_;
}
unsigned Human::population_ = 0;
```

Appel :

```
h1.population_();
```

Niveaux de protection

- **Private**: Accessible uniquement depuis la classe
- **Public**: Accessible depuis n'importe où.

Lorsqu'un attribut est **private** on peut utiliser un **getter/setter** pour **modifier celui-ci** depuis l'extérieur de la classe, tout en contrôlant la modification.

```
public class Human
{
  public String name ()
  {
    return name;
  }

  public void rename (String _name)
  {
    name = _name;
  }

  private String name;
  private final Gender gender;
  private final int birthYear;
}
```

Amitié

- Principe d'exeception au régime de privatisation.
- Accès privilégié depuis l'extérieur, autorisé au cas par cas.
- Le concept d'amitié est variable selon les langages. (Il n'existe pas en java)

```
class Human
{
  friend void birth_control (const Human& human);
}

void birth_control (const Human& human)
{
  std::cout << "The NSA knows about " << human.name_ << "...";
}

birth_control(h1);
```

Héritage

L'agrégation et l'héritage sont deux formes d'inclusions distinctes.

Exemple Java

```
public class Employee extends Human
{
        public Employee (String _name, float _size, int _birthYear, String
_company, int _salary)
        {
            super (_name, _size, _birthYear);
            company = _company;
            salary = _salary;
        }
    private String company;
    private int salary;
}
```

Un `Employee` est un sous-type de `Human`.

```
Human h = new Human ("Alain Terieur", 1.80f, 1970);
Employe e = new Employee ("Alain Terieur", 1.80f, 1970, "EPITA", 24000);
Human t = new Employee ("Alain Terieur", 1.80f, 1970, "EPITA", 24000);
```

Le dernier objet est de type `Human`, mais est en réalité un `Employee`. Il est caché au compilateur. Cela pourra être utile par la suite.

Une fonction déclarée `Human function(agr1, arg2)` pourra renvoyer un objet de type `Human` ou de type `Employee`.

Il existe aussi des protections au sein d'un héritage.

Polymorphisme

Cf. photo `Polymorphisme.png` pour comprendre le concept.