

Notion de type paramétré (exemple à compléter avec la série PoP)

La classe contient un attribut de type « catégorie » qui est ensuite utilisé au niveau des méthodes pour sélectionner le code exécuté pour chaque catégorie.

Exemple: cette classe **Form** contient un attribut **Category** défini avec **enum** et qui peut prendre l'une des valeurs (entière) des trois symboles : CIRCLE, SQUARE, RECTANGLE

```
enum Category {CIRCLE, SQUARE, RECTANGLE};
```

```
...
```

```
class Form
```

```
{
```

```
public:
```

```
    Form(Category c =CIRCLE, double x=0, double y=0, double p1=0);
```

```
    Form(Category c, double x, double y, double largeur, double hauteur );
```

```
...
```

```
    void affiche();
```

```
...
```

```
private:
```

```
    Category category;
```

```
    double x, y; // centre de la Form
```

```
    std::vector<double> param{std::vector<double>(1,0.)}; //un param par défaut
```

```
};
```

Constructeur pour CIRCLE ou SQUARE:

p1 est le rayon pour CIRCLE et le coté pour SQUARE

Ce constructeur sert aussi de constructeur par défaut



Constructeur pour RECTANGLE

Notion de type paramétré (suite)

Dans l'implémentation:

Ce constructeur de **CIRCLE** ou de **SQUARE** vérifie l'adéquation de la catégorie et de la valeur transmise pour le paramètre (rayon / coté)

```
#include "Form.h"
...
Form::Form(Category c, double x, double y, double p1)
    :category(c), x(x), y(y)
{
    switch(category)
    {
        case CIRCLE:
        case SQUARE: param[0] = p1;
            if(param[0] < 0.)
            {
                cout << "Form: incorrect parameter value" << endl;
                exit(EXIT_FAILURE);
            }
            break;
        default: cout << "Form: incorrect category" << endl;
            exit(EXIT_FAILURE);
    }
}
```

Notion de type paramétré (suite)

Dans l'implémentation (suite): la méthode **affiche**

```
#include "Form.h"
```

```
...
```

```
void Form::affiche()
```

```
{
```

```
    switch (category)
```

```
    {
```

```
        case CIRCLE: cout << "CIRCLE with center: (" << x << ", " << y  
                        << " ) and radius " << param[0] << endl;
```

```
            break;
```

```
        case SQUARE: cout << "SQUARE with center: (" << x << ", " << y  
                        << " ) and side " << param[0] << endl;
```

```
            break;
```

```
        case RECTANGLE: cout << "RECTANGLE with center: (" << x << ", " << y  
                        << " ), width: " << param[0]  
                        << " and height: " << param[1] << endl;
```

```
            break;
```

```
        default: cout << "Form: incorrect category" << endl;
```

```
            exit(EXIT_FAILURE);
```

```
    }
```

```
}
```

Une méthode typique devra toujours contenir un switch sur l'attribut category
il est recommandé de mettre en œuvre le principe d'Abstraction dès que le code
spécialisé par catégorie dépasse 2-3 lignes.

Notion de type paramétré : avantage principal

Des instances de différentes catégories peuvent être stockées dans un même conteneur

```
#include "Form.h"                                     main.cc
...
int main()
{
    vector<Form> tab;
    ...
    // ici du code qui ajoute des instances Form
    // de différentes categories dans le vector tab
    ...

    // unique boucle qui délègue à la classe Form
    // le détail des actions pour chaque méthode

    for(const auto& s : tab)
        s.affiche();
    ...
}
```

Notion de type paramétré : faiblesses

Une seule classe gère le code de toutes les catégories:

- mise au point plus globale/longue
- La classe n'est validée que lorsque toutes les catégories ont été validées

Plus grave, la maintenance du code peut ré-introduire des bugs dans du code validé

- si on veut ajouter une nouvelle catégorie, par exemple TRIANGLE, il faut éditer toutes les méthodes qui contiennent un switch sur l'attribut categorie
- Mauvais approche pour le développement robuste de code : perte de temps

Définir *une classe par catégorie* est-elle la solution ?

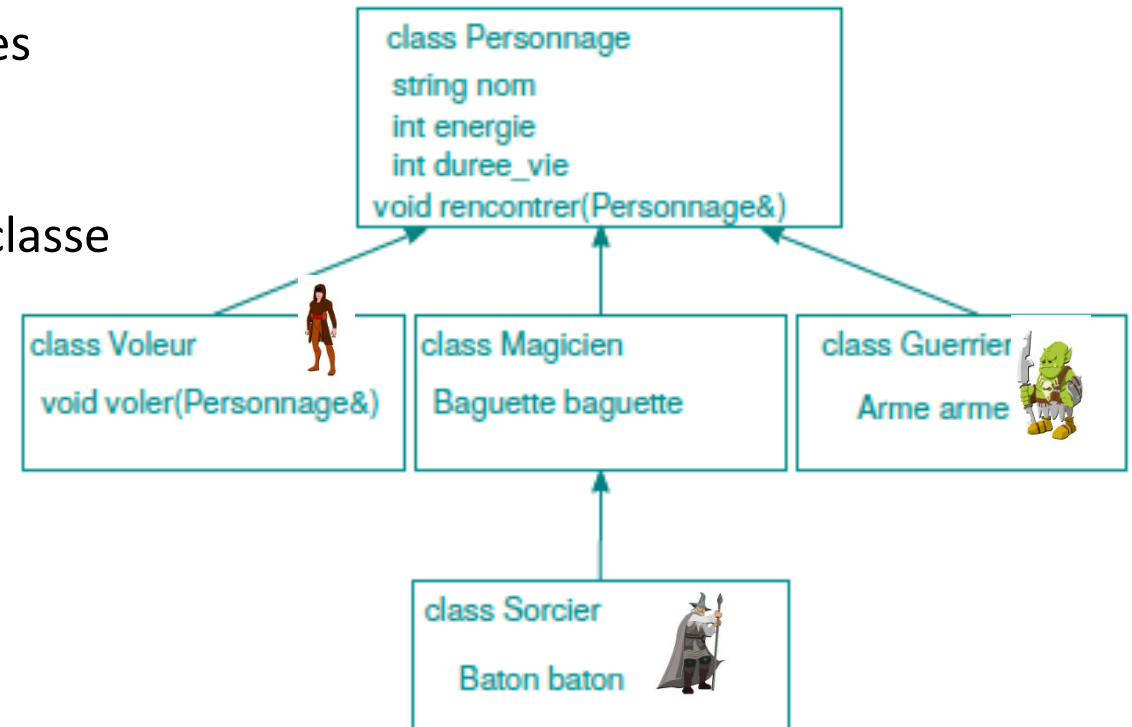
- Pas si les classes sont totalement indépendantes :
 - => beaucoup de code dupliqué dans les classes gérant des entités proches.
- De plus on ne peut plus stocker des instances de catégories différentes dans un même conteneur

La solution est de mettre au point une *hiérarchie de classe* (MOOC sem4)

Mise en œuvre du principe de ré-utilisation avec l'héritage

Motivation: lorsqu'un ensemble d'entités présente un forte cohérence sémantique

- Eviter de dupliquer du code manipulant des propriétés communes
- Partager un sous-ensemble de propriétés/méthodes communes dans une superclasse
- Définir des classes spécialisées à partir de la superclasse
- Environnement professionnel:
Offrir une base de code cohérente et **validée** pouvant être seulement **étendue** par l'**ajout de sous-classes**



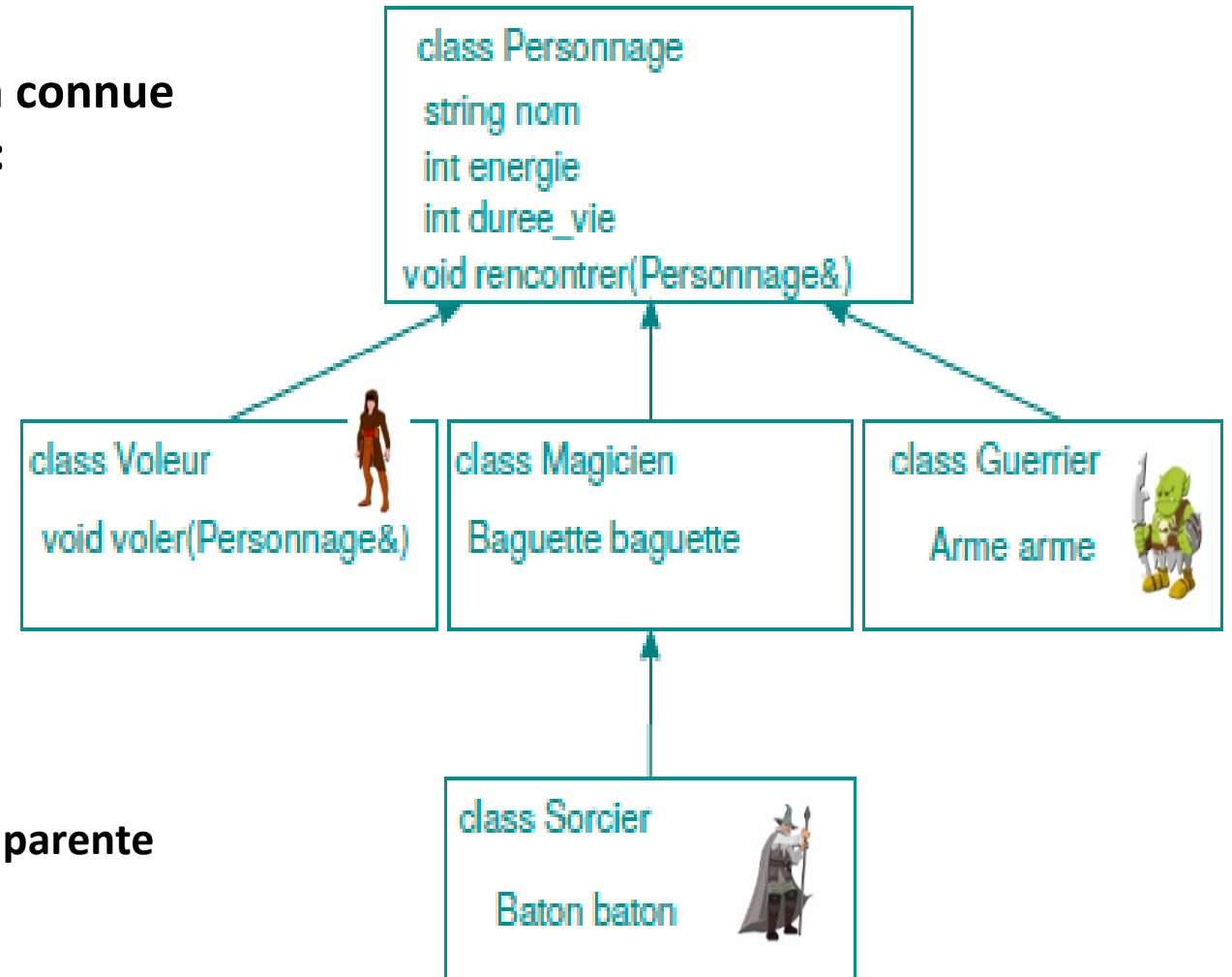
Héritage : Distinguer “Est un(e)” de “Possède un(e)”

La relation « Possède un(e) » est la relation déjà connue entre **une instance d’une classe** et **ses attributs** :
=> Un Personnage possède : nom, energie, duree_vie

La relation « Est un(e) » est celle qui existe entre **une sous-classe** et **sa classe parente** :
=> Un Sorcier est un Magicien
=> Un Magicien est un Personnage

Une sous-classe est aussi appelée une classe *dérivée*

Une classe dérivée est une *spécialisation* de sa classe parente



Résumé

- **La notion de type paramétré est utile temporairement (Rendu1 projet).**
- **Il présente l'avantage de pouvoir stocker des instances de différentes catégories dans un même conteneur.**
- **Les inconvénients sont supérieurs à cet avantage:**
 - **Mise au point plus longue car code plus lourd**
 - **Risque de ré-introduire des bugs dans un code validé quand on veut augmenter le nombre de catégories**
- **Le concept de hiérarchie de classe et d'héritage résout ces problèmes**
 - **Le code commun à toutes les classes est rassemblé dans une superclasse**
 - **On spécialise la superclasse en dérivant une sous-classe indépendante**
 - **Chaque classe dérivée peut être mise au point et validée séparément**