

# Programmation Modulaire

## Initiation au génie logiciel : modularité, encapsulation

**Viviane Pons**

Mail : [viviane.pons@universite-paris-saclay.fr](mailto:viviane.pons@universite-paris-saclay.fr)

Page professionnelle : <http://www.lri.fr/~pons>

26 mars 2025

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
  - Notion d'interface
  - Principe d'encapsulation
  - Droits d'accès en C++
- 3 Exemples
  - Utilisations d'un type abstrait
  - Implémentation d'un type abstrait
  - Évolution d'un type abstrait
- 4 Résumé et compléments

# Plan

- 1** Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
- 3 Exemples
- 4 Résumé et compléments

## Petit historique de l'évolution de l'informatique

- Jusqu'en 1985, les ordinateurs appartenait à des sociétés ou des institutions. Les logiciels étaient développés par des membres des institutions pour leurs propres besoins.
- 1970 – Nouvelles notions : multi-utilisateur, les interfaces graphiques, la programmation concurrente, les bases de données et le temps réel. Conséquence : logiciels beaucoup plus sophistiqués
- 1980 – Ordinateur personnel, progiciels — des logiciels prêts-à-porter.
- 1985-2000 – Systèmes distribués, Internet, client-serveur, cloud computing. Le logiciel devient un élément d'un ensemble. Plusieurs ordinateurs et plusieurs logiciels travaillent en collectivité.

## Petit historique de l'évolution de l'informatique

- Les machines vont de plus en plus vite, elles sont donc capables d'effectuer des opérations de plus en plus complexes
- On écrit de plus en plus de logiciels, les logiciels sont de plus en plus gros

### Retenir

*Mais, on constate que le **nombre d'erreurs par lignes de code** est en gros **indépendant du langage**.*

## Petit historique de l'évolution de l'informatique

- Les machines vont de plus en plus vite, elles sont donc capables d'effectuer des opérations de plus en plus complexes
- On écrit de plus en plus de logiciels, les logiciels sont de plus en plus gros

### Retenir

Mais, on constate que le **nombre d'erreurs par lignes de code** est en gros *indépendant du langage*.

# La crise du logiciel

Jusqu'en 1970, on écrivait les logiciels de manière artisanale...

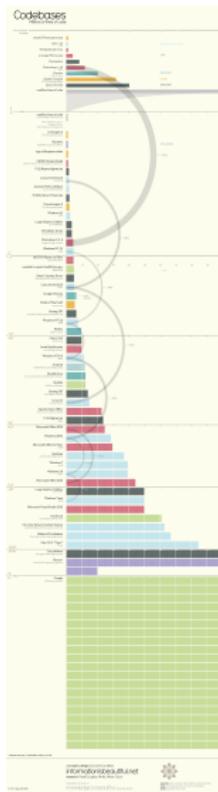
## Problème

Autour de 1970, crise du logiciel :

- **baisse significative de qualité des logiciels**
- début de l'**utilisation des circuits intégrés** dans les ordinateurs
- L'**augmentation de la puissance de calcul** des ordinateurs a permis de réaliser des logiciels **beaucoup plus complexes** qu'auparavant.

## Combien de lignes de code

- Voir combien de ligne de code
- Google Chrome, World of WarCraft = 5 millions de LoC
- Facebook = 50 millions de LoC
- Apple Mac OS X = 80 millions de LoC
- Google = 20 milliards de LoC



## Combien d'erreurs par lignes de code

Steve McConnell, auteur de «Code Complete» (1993) et «Software Estimation : Demystifying the Black Art» (2006) :

Unité :  $d/kLoC$  = défauts par 1000 lignes de code.

- Code industriel moyen :
  - de l'ordre 15 – 50  $d/kLoC$  dans le code produit.
- Microsoft :
  - Environ 10 – 20  $d/kLoC$  lors des tests maison
  - 0.5  $d/kLoC$  dans le code produit
- En 1990, la technique *cleanroom development* permet de produire un taux de
  - 3  $d/kLoC$  lors des tests maison
  - 0.1  $d/kLoC$  dans le code produit

## Combien d'erreur par lignes de code (2)

Selon [Steve McConnell](#) :

- «Quelques projets (par exemple la navette spatiale) ont atteint un taux de 0 défauts pour 500 000 lignes en utilisant des systèmes de **développement formel, de revue par les pairs et de tests statistiques**».
- Mais chaque programmeur s'occupait de **2 600 lignes de code pendant 10 ans**, pour s'assurer de leur correction.

### Problème

Combien d'entreprises permettent à leurs programmeurs une productivité de 260 lignes de code correct par an ?

## Combien d'erreur par lignes de code (2)

Selon [Steve McConnell](#) :

- «Quelques projets (par exemple la navette spatiale) ont atteint un taux de 0 défauts pour 500 000 lignes en utilisant des systèmes de **développement formel, de revue par les pairs et de tests statistiques**».
- Mais chaque programmeur s'occupait de **2 600 lignes de code pendant 10 ans**, pour s'assurer de leur correction.

### Problème

Combien d'entreprises permettent à leurs programmeurs une productivité de 260 lignes de code correct par an ?

## Combien d'erreur par lignes de code (2)

Selon [Steve McConnell](#) :

- «Quelques projets (par exemple la navette spatiale) ont atteint un taux de 0 défauts pour 500 000 lignes en utilisant des systèmes de **développement formel, de revue par les pairs et de tests statistiques**».
- Mais chaque programmeur s'occupait de **2 600 lignes de code pendant 10 ans**, pour s'assurer de leur correction.

### Problème

Combien d'entreprises permettent à leurs programmeurs une productivité de 260 lignes de code correct par an ?

## Génie logiciel

Notion inventée par [Margaret Hamilton](#), conceptrice du système embarqué du Programme Apollo.

Parenthèse : je recommande la vidéo :

Brian Troutwine

*The Charming Genius of the Apollo Guidance Computer*

<https://www.youtube.com/watch?v=d1nz7vgyUh8>

Pour guider le vaisseau spatial, l'AGC est le premier ordinateur :

- avec une interface utilisateur (clavier / écran)
- multitache (plusieurs programme en même temps)
- temps réel (le système délivre les résultats exacts dans des délais imposés)

## Génie logiciel

Notion inventée par [Margaret Hamilton](#), conceptrice du système embarqué du Programme Apollo.

Parenthèse : je recommande la vidéo :

Brian Troutwine

*The Charming Genius of the Apollo Guidance Computer*

<https://www.youtube.com/watch?v=d1nz7vgyUh8>

Pour guider le vaisseau spatial, l'AGC est le premier ordinateur :

- avec une interface utilisateur (clavier / écran)
- multitache (plusieurs programme en même temps)
- temps réel (le système délivre les résultats exacts dans des délais imposés)

## Génie logiciel

Notion inventée par [Margaret Hamilton](#), conceptrice du système embarqué du Programme Apollo.

Parenthèse : je recommande la vidéo :

Brian Troutwine

*The Charming Genius of the Apollo Guidance Computer*

<https://www.youtube.com/watch?v=d1nz7vgyUh8>

Pour guider le vaisseau spatial, l'AGC est le premier ordinateur :

- avec une interface utilisateur (clavier / écran)
- multitache (plusieurs programme en même temps)
- temps réel (le système délivre les résultats exacts dans des délais imposés)

## Génie logiciel

Lecture recommandée :

[https://fr.wikipedia.org/wiki/Génie\\_logiciel](https://fr.wikipedia.org/wiki/Génie_logiciel)

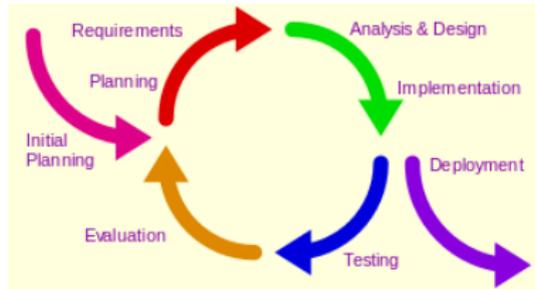
Retenir (arrêté ministériel du 30 décembre 1983)

*Le génie logiciel est l'ensemble des activités de conception et de mise en œuvre des produits et des procédures tendant à rationaliser la production du logiciel et son suivi.*

Mots clefs :

- code source, spécification, production
- cycle de vie des logiciels
- analyse du besoin, spécifications, développement, test, maintenance

# Éléments de génie logiciel



- Planification du travail : architecture logicielle, développement de cadriciel (*framework*)
- Technique de division du travail : langage objet, programmation modulaire, programmation générique, *etc*
- Contrôle de qualité : revue de code, test, vérification formelle
- Solutions standards pour les problèmes courants : bonnes pratiques, patrons de conception (*design pattern*)
- Gestion de code : documentation, système de gestion de version, traqueur de bugs

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire**
- 3 Exemples
- 4 Résumé et compléments

# Initiation au génie logiciel

Nous allons nous intéresser particulièrement à un point :

## Retenir (Programmation modulaire)

*Technique de programmation* qui permet de diviser un gros programme en composants (**les modules**). On peut ainsi

- *développer les composants indépendamment* (division du travail en équipes)
- *tester les composants indépendamment*
- *les améliorer* alors qu'ils sont déjà utilisés
- *les réutiliser* pour d'autres programmes

## Exemple : un sac de courses pour faire son marché !

### Problème

Objectif : gestion / simulation d'un sac pouvant contenir

- des pommes
- des oranges
- des salades
- des morceaux de viande
- autre

Bien sûr, il peut y avoir plusieurs objets de chaque sorte dans le sac.

## Division des tâches

On veut diviser le travail en trois équipes :

- 1 l'équipe qui va s'occuper de la **gestion du sac**
- 2 l'équipe qui va **tester** que la gestion du sac fonctionne bien
- 3 l'équipe qui va s'occuper de **l'interaction avec l'utilisateur**

### Remarque

Les programmeurs des équipes 2 et 3 n'ont **pas besoin de connaître** les détails de **la logique interne** du sac.

## Division des tâches

On veut diviser le travail en trois équipes :

- 1 l'équipe qui va s'occuper de la **gestion du sac**
- 2 l'équipe qui va **tester** que la gestion du sac fonctionne bien
- 3 l'équipe qui va s'occuper de **l'interaction avec l'utilisateur**

### Remarque

Les programmeurs des équipes 2 et 3 n'ont **pas besoin de connaître** les détails de **la logique interne** du sac.

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire**
  - Notion d'interface
  - Principe d'encapsulation
  - Droits d'accès en C++
- 3 Exemples
- 4 Résumé et compléments

# Programmation modulaire

Pour pouvoir travailler indépendamment, les trois équipes doivent se mettre d'accord sur **le mode d'emploi du composant sac**.

Définition (Notion d'**interface et de spécification**)

La **spécification fonctionnelle** est la *description des fonctions d'un logiciel en vue de son implémentation*.

décrit souvent une *interface de programmation applicative* (anglais API pour Application Programming Interface). C'est l'ensemble des fonctions qui servent de façade par laquelle un **logiciel offre des services à d'autres logiciels**.

## Exemple : interface pour le Sac

sac-interface.hpp

```
1
2  /** Constructeur par défaut
3  Sac ();
4
5  /** Ajoute un objet à un sac
6  /** @param[in] n : l'objet à ajouter
7  void ajoute(Objet n);
8
9  /** Retire un objet à un sac
10 /** @param[in] n : l'objet à retirer
11 /** si n n'est pas dans le sac, ne fait rien
12 /** @return si on a bien pris l'objet dans le sac
13 bool retire(Objet n);
14
15 /** Teste si le sac est vide
16 /** @return : un booléen selon le résultat du test
17 bool estVide() const;
18
19 /** Retourne un objet du sac
20 /** @return : un objet du sac s'il n'est pas vide
21 /**          le comportement est indéfini si e est vide
22 Objet objet() const;
```

## Remarque

### L'interface

sac-interface.hpp

```
Sac ();  
void ajoute(Objet n);  
bool retire(Objet n);  
bool estVide() const;  
Objet objet() const;
```

suppose définis deux types

- un type `Objet` qui **modélise les objets** à mettre dans le sac
- un type `Sac` qui **modélise le sac**

## Abstraction de type

Dans l'interface vue précédemment, on n'a pas décrit les types, ils sont **abstraits**.

### Définition

*En génie logiciel, un type abstrait est une **spécification d'un ensemble de données et de l'ensemble des opérations** qu'elles peuvent effectuer.*

*On qualifie d'**abstrait** ce type de données car il correspond à un cahier des charges qu'une structure de données doit ensuite implémenter, mais l'on ne **spécifie pas la structure de données**.*

## Notion de types abstraits (2)

Les types abstraits sont des représentations des données (au sens large) **indépendantes de la réalisation et a fortiori du codage.**

Pour l'implémentation, à chaque type abstrait donné, **on associera un type concret** (choisi parmi les différentes variantes possibles), **qui sera ensuite codé en un type du langage C++.**

La relation (type abstrait/concret) devra être analysée finement afin de choisir au mieux le type concret en fonction du type abstrait considéré, selon des critères liés à :

- la simplicité d'utilisation,
- le temps de calcul,
- l'espace mémoire requis.

# Définition des types abstraits

## Retenir

*Un type abstrait est défini par :*

- *un **nom***
- *un ensemble de **constructeurs** (constantes ou fonctions) qui permettent de **produire des données de ce type** à partir d'autres données*
- *des **opérateurs qui permettent de manipuler les données** de ce type abstrait.*

## Liste des opérations

### Retenir

*Les opérations des types abstraits (autres que les constructeurs) peuvent être regroupées en 3 catégories :*

- *opérations d'accès* : **permettent d'accéder aux composants internes des données du type abstrait**
- *opérations de test* : **permettent de tester les caractéristiques des données**
- *autres opérations* : on considère ici, généralement, différentes **opérations de base** (souvent utilisées) qu'on souhaite rendre plus **efficaces** en les optimisant pour le type concret choisi.

## Dans notre exemple

- Constructeur

```
Sac ();
```

sac-interface.hpp

- Opérations de test

```
bool estVide() const;
```

sac-interface.hpp

- Opérations d'accès

```
void ajoute(Objet n);
bool retire(Objet n);
Objet objet() const;
```

sac-interface.hpp

- Autres opérations

```
int nbObjet() const;
int nbObjet(Objet n) const;
```

sac-interface.hpp

## Boîte noire / boîte blanche

### Retenir

*Un composant est vu comme une **boîte noire** lorsqu'on ne s'intéresse qu'à son **usage et son comportement**, définis par exemple par des spécifications : c'est le point de vue de l'**utilisateur**.*

### Retenir

*Un composant est vu comme une **boîte blanche** lorsqu'on s'intéresse à **son organisation** et à son fonctionnement : c'est le point de vue du **concepteur, du réparateur**.*

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire**
  - Notion d'interface
  - Principe d'encapsulation
  - Droits d'accès en C++
- 3 Exemples
- 4 Résumé et compléments

# Principe d'encapsulation

## Retenir

*Pour les **utilisateurs** du type abstrait, **toutes les manipulations des donn es du type abstrait doivent se faire au travers de l'interface** du type abstrait.*

- permet aux **concepteurs** de **modifier les structures de donn es internes sans modifier l'interface** du type abstrait et donc sans affecter **les utilisateurs**.
- fr quent lorsque l'on veut am liorer l'efficacit 
- permet d'**assurer l'int grit  (invariants)** de la structure de donn es
-  vite l'**effet plat de spaghettis**, o  l'on ne sait plus par qui, par quoi et comment les donn es sont modifi es

# Encapsulation en C++

## Remarque

Dans beaucoup de **langages objets, le principe d'encapsulation est pris en compte dans le langage lui-même**. Le C++ permet de contrôler le respect de l'encapsulation à l'aide des mots clés

```
struct      class  
public     private    protected
```

Le mots clé `protected` n'est utile que si l'on fait de la programmation objet avancée (héritage). Nous ne l'utiliserons pas.

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire**
  - Notion d'interface
  - Principe d'encapsulation
  - Droits d'accès en C++
- 3 Exemples
- 4 Résumé et compléments

## Droits d'accès en C++

### Retenir

*Le C++ distingue deux accessibilités :*

- **public** : accessible à tout le monde ;
- **private** : accessible uniquement à l'intérieur des méthodes de la classe.

*Ces restrictions peuvent être appliquées à tous les membres d'une classes : attributs, opérateurs, constantes, méthodes. . .*

## Droits d'accès en C++ (2)

### Retenir

*Les mots `class` et `struct` diffèrent seulement par l'accessibilité par défaut :*

- `struct` accessibilité `public`
- `class` accessibilité `private`

Donc par exemple, les deux écritures suivantes sont équivalentes :

<pre> 1  struct Toto { 2      private: 3          int val; 4      public: 5          int get_val() const; 6          void set_val(); 7  }; </pre>	<pre> 1  class Toto { 2      int val; 3      public: 4          int get_val() const; 5          void set_val(); 6  }; </pre>
---	--

## Droits d'accès en C++ (3)

L'usage le plus courant est le suivant :

- On utilise le mot `struct` si tout est public (le plus souvent sans constructeurs) ;
- Sinon on utilise `class` et l'on précise les accès.

## Droits d'accès en C++ : exemples

Pour le sac, on va donc déclarer les méthodes de l'interface avec l'accès `public` et les attributs seront d'accès `private`.

Note : il faut quand même déclarer les attributs dans le fichier `.hpp`

	sac-inttab.hpp		sac-inttab.hpp
8	<code>class Sac {</code>	38	<code>/** Le nombre d'occurrence d'un objet dans le</code>
9		39	<code>/** @param[in] : n un objet</code>
10	<code>public:</code>	40	<code>/** @return le nombre de fois où l'objet n es</code>
11		41	<code>int nbObjet(Objet n) const;</code>
12	<code>/** Constructeur par défaut</code>	42	
13	<code>Sac ();</code>	43	<code>private:</code>
14		44	
15	<code>/** Ajoute un objet à un sac</code>	45	<code>std::array&lt;int, MaxSac&gt; tab_;</code>
16	<code>/** @param[in] n : l'objet à ajouter</code>	46	<code>};</code>
17	<code>void ajoute(Objet n);</code>		
18			
19	<code>/** Retire un objet à un sac</code>		

# Getter et Setter (1)

## Retenir

*Dans une classe où les attributs sont privés, on peut avoir besoin de*

- *lire un attribut. Pour ceci, on crée une méthode appelée «getter» :*

```
typeAttr get_nomAttr() const { return attr; }
```

- *modifier un attribut. Pour ceci, on crée une méthode appelée «setter» :*

```
void set_nomAttr(typeAttr val) {  
    verifications éventuelles...  
    attr = val;  
}
```

## Getter et Setter (2)

Attention ! Il faut bien réfléchir avant de mettre des getter et des setter. Il ne sont que rarement nécessaires.

### Attention

En particulier, il ne faut pas rajouter un getter pour permettre à l'utilisateur d'implémenter lui-même une fonctionnalité qu'il faudrait avoir mise dans l'interface.

Exemple : affichage.

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
- 3 Exemples**
- 4 Résumé et compléments

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
- 3 Exemples**
  - Utilisations d'un type abstrait
  - Implémentation d'un type abstrait
  - Évolution d'un type abstrait
- 4 Résumé et compléments

## Utilisations du type abstrait Sac

marche.cpp

```
1  Sac s{};
2  int action;
3  switch (action) {
4  case 1:
5      if (s.estVide()) cout << "Le sac est vide";
6      else cout << "Le sac n'est pas vide";
7      break;
8  case 2: cout << "Quoi ?";
9      cin >> c;
10     s.ajoute(c);
11     break;
12 case 3: cout << "Quoi ?";
13     cin >> c;
14     if (not s.retire(c))
15         cout << "Il n'y a pas de " << c << " dans le sac";
16     break;
17 case 4:
18     if (s.estVide()) cout << "Le sac est vide";
19     else
20         cout << "Dans le sac, il y a : " << s.objet();
21     break;
22 case 5:
23     cout << endl << endl;
```

test-sac.cpp

```
TEST_CASE("Constructeur Sac") {  
    Sac s{};  
    CHECK(s.estVide());  
}  
  
TEST_CASE("Méthode ajoute") {  
    Sac s{};  
    CHECK(s.estVide());  
    s.ajoute(Course::pomme);  
    CHECK_FALSE(s.estVide());  
}
```

test-sac.cpp

```
TEST_CASE("Méthode objet") {  
    Sac s{};  
    Objet o;  
    s.ajoute(Course::pomme);  
    CHECK(s.objet() == Course::pomme);  
    s.ajoute(Course::orange);  
    o = s.objet();  
    CHECK((o == Course::pomme or o == C  
    s.retire(Course::pomme);  
    o = s.objet();  
    CHECK(o == Course::orange);  
}
```

```
test-sac.cpp // p2 o2 s1
TEST_CASE("Méthode retire") {
    Sac s{};
    s.ajoute(Course::pomme);
    CHECK_FALSE(s.estVide());
    CHECK_FALSE(s.retire(Course::viande));
    CHECK(s.retire(Course::pomme));
    CHECK(s.estVide());
    CHECK_FALSE(s.retire(Course::pomme));

    s.ajoute(Course::pomme);
    s.ajoute(Course::pomme);
    CHECK(s.retire(Course::pomme));
    CHECK(s.retire(Course::pomme));
    CHECK(s.estVide());
    CHECK_FALSE(s.retire(Course::pomme));

    s.ajoute(Course::pomme);
    s.ajoute(Course::orange);
    s.ajoute(Course::pomme);
    s.ajoute(Course::orange);
    s.ajoute(Course::salade);
    CHECK_FALSE(s.estVide());

    CHECK_FALSE(s.retire(Course::poire));
    CHECK(s.retire(Course::orange));
    // p2 o1 s1
    CHECK_FALSE(s.estVide());
    CHECK(s.retire(Course::salade));
    // p2 o1 s0
    CHECK_FALSE(s.estVide());
    CHECK_FALSE(s.retire(Course::poire));
    CHECK_FALSE(s.retire(Course::salade));
    CHECK(s.retire(Course::pomme));
    // p1 o1
    CHECK(s.retire(Course::pomme));
    // p0 o1
    CHECK_FALSE(s.retire(Course::poire));
    CHECK_FALSE(s.retire(Course::pomme));
    CHECK_FALSE(s.retire(Course::salade));
    CHECK(s.retire(Course::orange));
    CHECK(s.estVide());
    CHECK_FALSE(s.retire(Course::pomme));
    CHECK_FALSE(s.retire(Course::salade));
    CHECK_FALSE(s.retire(Course::orange));
    CHECK_FALSE(s.retire(Course::poire));
}
```

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
- 3 Exemples**
  - Utilisations d'un type abstrait
  - Implémentation d'un type abstrait
  - Évolution d'un type abstrait
- 4 Résumé et compléments

## Implémentation 1 du type abstrait Sac

Pour réaliser un type abstrait, il faut choisir un type concret.

### Exemple

Première implantation : si le nombre de valeurs du type `Objet` n'est pas trop grand, on peut faire un **tableau qui à chaque objet associe le nombre de fois qu'il est dans le sac**.

sac-inttab.hpp

```
using Objet = Course;
const int MaxSac = NbCourse;

class Sac {
private:

    std::array<int, MaxSac> tab_;
};
```

## Implémentation 1 du type abstrait Sac (1)

sac-inttab.cpp

```
14 Sac::Sac() {
15     for (int i=0; i < MaxSac; i++) tab_[i] = 0;
16 }

19 bool Sac::estVide() const {
20     for (int i=0; i < MaxSac; i++)
21         if (tab_[i] != 0) return false;
22     return true;
23 }
```

## Implémentation 1 du type abstrait Sac (2)

sac-inttab.cpp

```
26 void Sac::ajoute(Objet n) {
27     tab_[int(n)]++;
28 }

31 bool Sac::retire(Objet n) {
32     if (tab_[int(n)] == 0) return false;
33     tab_[int(n)]--;
34     return true;
35 }

38 Objet Sac::objet() const {
39     for (int i=0; i < MaxSac; i++)
40         if (tab_[i] != 0) return Objet(i);
41     throw std::runtime_error("Objet sur sac vide");
42 }
```

## Implémentation 2 du type abstrait Sac

### Exemple

Deuxième implantation : on stocke la **liste des objets du Sac en utilisant un vecteur**.

sac-vect.hpp

```
const int MaxSac = NbCourse;  
  
class Sac {  
  
    std::vector<Objet> tab_  
};
```

## Implémentation 2 du type abstrait Sac (1)

sac-vect.cpp

```
7  Sac::Sac() : tab_{} {}  
  
10 bool Sac::estVide() const {  
11     return tab_.size() == 0;  
12 }  
  
33 Objet Sac::objet() const {  
34     return tab_[0];  
35 }
```

## Implémentation 2 du type abstrait Sac (2)

sac-vect.cpp

```
15 void Sac::ajoute(Objet n) {
16     tab_.push_back(n);
17 }

20 bool Sac::retire(Objet n) {
21     // Cherche un n dans le sac
22     for (int i = 0; i < tab_.size(); i++)
23         if (tab_[i] == n) {
24             // échange avec le dernier et supprime ce dernier
25             std::swap(tab_[i], tab_.back());
26             tab_.pop_back();
27             return true;
28         }
29     return false;
30 }
```

## Autres implémentations possibles

- par un `vector<pair<Objet, int>>` où l'on stocke l'objet et sa multiplicité
- par un `vector<pair<Objet, int>>` trié. On peut alors retrouver un objet plus rapidement par recherche dichotomique
- par une table de hachage : `unordered_multiset<Objet>`

sac-uset.hpp

```
#include <unordered_set>  
};
```

- par un arbre binaire de recherche : `multiset<Objet>`

sac-set.hpp

```
#include <set>  
};
```

## Autres implémentations possibles

### Compléments

Le C++, fournit deux implémentations du type abstrait multiset `unordered_multiset<Objet>`, et `multiset<Objet>`

### Compléments

En Java, il y a trois implémentations du type abstrait Bag `DefaultMapBag`, `HashBag`, `TreeBag`.

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
- 3 Exemples**
  - Utilisations d'un type abstrait
  - Implémentation d'un type abstrait
  - **Évolution d'un type abstrait**
- 4 Résumé et compléments

## Évolution du type abstrait (1)

Si l'on essaye de réaliser les fonctions

```
int Sac::nbObjet(const Sac& s);
```

```
int Sac::nbObjet(const Sac& s, Objet);
```

en respectant l'encapsulation, on obtient un code qui peut être très inefficace.

En effet, on va devoir faire une boucle avec `Sac::objet` et `Sac::retire`, alors que, par exemple la taille est connue pour l'implémentation 2.

## Évolution du type abstrait (1)

Si l'on essaye de réaliser les fonctions

```
int Sac::nbObjet(const Sac& s);
```

```
int Sac::nbObjet(const Sac& s, Objet);
```

en respectant l'encapsulation, on obtient un code qui peut être très inefficace.

En effet, on va devoir faire une boucle avec `Sac::objet` et `Sac::retire`, alors que, par exemple la taille est connue pour l'implémentation 2.

## Évolution du type abstrait (2)

### Compléments

Dans le cycle de vie d'un développement, il y a une phase dite de **refactorisation**, où l'on est amené à réorganiser le découpage en composants d'un code.

Cette phase permet, entre autres, de résoudre les problèmes de

- code confus car un composant n'est pas à sa place naturelle
- code inefficace car un composant n'a pas accès directement à la donnée
- code dupliqué car on n'a pas défini un composant suffisamment général pour toutes les utilisations

## Implémentation 1 du type abstrait Sac (3)

sac-inttab.cpp

```
45 int Sac::nbObjet() const {
46     int res = 0;
47     for (int i=0; i < MaxSac; i++)
48         res += tab_[i];
49     return res;
50 };
51
52 int Sac::nbObjet(Objet n) const {
53     return tab_[int(n)];
54 };
```

## Implémentation 2 du type abstrait Sac (3)

sac-vect.cpp

```
38 int Sac::nbObjet() const {
39     return tab_.size();
40 }
41
42 int Sac::nbObjet(Objet n) const {
43     int res = 0;
44     for (int i = 0; i < tab_.size(); i++)
45         if (tab_[i] == n) res++;
46     return res;
47 }
```

# Plan

- 1 Qu'est-ce que le génie logiciel
- 2 Programmation modulaire
- 3 Exemples
- 4 Résumé et compléments**

## Structure ou Classe ? (1)

Pour modéliser un objet du monde réel :

### Retenir

*On utilise une `struct` avec **tous les attributs publics** quand*

- *on veut juste **regrouper des variables** liées entre elles*
- *il n'y a **pas de contraintes de cohérence** et d'intégrité (pas d'invariants)*

*Exemple : les coordonnées d'un point, les informations sur une personne dans un carnet d'adresse.*

## Structure ou Classe ? (2)

Pour modéliser un objet du monde réel :

### Retenir

*Sinon, on utilise une class avec **tous les attributs privés** :*

- **getter** pour **rendre accessibles** certains attributs
- **setter** pour **rendre modifiables** certains attributs
- **vérification de la cohérence** à la création et après une modification (si besoin).
- **tous les objets sont dans un état cohérent** (sauf pendant les modifications à l'intérieur des méthodes de l'objet).

## Bonne pratiques (2)

### Retenir

*Il faut suivre des **conventions de noms cohérentes***

- *Type et Classes commencent par une majuscule (ex. EnsCoord).*
- *fonctions et méthodes en camelCase*
- *fonction qui **modifie** l'objet : **verbe conjugué qui décrit l'action** que l'on fait sur l'objet. ex. Ensemble::ajoute(x)*
- *fonction qui **retourne un résultat** : **nom qui décrit le résultat du calcul**. ex. Ensemble::union(a, b)*

# Résumé

## Retenir (Méthodologie du développement par type abstrait)

*La démarche est la suivante*

**1** *Choix du nom*

**2** *Écriture de l'interface :*

- *Spécification des constructeurs*
- *Spécification des méthodes (comportement, type de retour, paramètres, propriété const...)*
- *Spécification des fonctions et opérateurs*

**3** *Écriture des tests*

**4** *Choix d'un type concret (tous les attributs sont privés)*

**5** *Implém. des méthodes internes (accès aux attributs privés)*

**6** *Implém. des méthodes, fonctions et opérateurs externes (pas d'accès au attributs privés)*

## Problème des opérateurs externes

### Problème

Si on définit une fonction ou un opérateur (ex. l'affichage) qui n'est pas une méthode de la classe, on n'a pas accès aux attributs.

### Retenir

*Plusieurs solutions :*

- 1 *Les attributs sont accessibles par getter ex. `Rationnel.num()` ;*
- 2 *Recommandée : Ajouter une **méthode publique** dans la classe qui fait le travail ; l'opérateur est juste un raccourci (sucre) syntaxique et délègue le travail à la méthode.*
- 3 *Donner l'accès à la fonction par le mot clef `friend`.*

## Problème des opérateurs externes

### Problème

Si on définit une fonction ou un opérateur (ex. l'affichage) qui n'est pas une méthode de la classe, on n'a pas accès aux attributs.

### Retenir

*Plusieurs solutions :*

- 1 Les attributs sont accessibles par *getter* ex. `Rationnel.num()` ;
- 2 **Recommandée** : Ajouter une **méthode publique** dans la classe qui fait le travail ; l'opérateur est juste un raccourci (sucre) syntaxique et délègue le travail à la méthode.
- 3 Donner l'accès à la fonction par le mot clef `friend`.

## Affichage : méthode du sucre syntaxique

Déclaration :

sac-print.hpp

```
10 class Sac {
11     public:
46     std::ostream& print(std::ostream& out) const;
53 };
56 std::ostream& operator<<(std::ostream& out, const Sac& s);
```

Implémentation :

sac-print.cpp

```
50 std::ostream& Sac::print(std::ostream& out) const {
51     out << "{";
52     for (int i = 0; i < int(tab_.size()); i++)
53         out << tab_[i] << ",";
54     return out << "}";
55 }
56 std::ostream& operator<<(std::ostream& out, const Sac& s) {
57     return s.print(out);
58 }
```

# Le mot clé `friend`

## Compléments

On peut donner accès aux membres privés (attributs, méthodes, ...) d'une classe en déclarant `friend`, le code (classe, fonction, opérateur, ...) auquel on veut donner accès. Il a alors les mêmes droits que les membres de la classe.

## Attention

On contourne l'encapsulation ! Beaucoup de spécialistes considèrent que c'est de la mauvaise programmation, mais c'est très courant dans les codes, en particulier pour les affichages.

## Le mot clé `friend`

### Compléments

On peut donner accès aux membres privés (attributs, méthodes, ...) d'une classe en déclarant `friend`, le code (classe, fonction, opérateur, ...) auquel on veut donner accès. Il a alors les mêmes droits que les membres de la classe.

### Attention

On contourne l'encapsulation ! Beaucoup de spécialistes considèrent que c'est de la mauvaise programmation, mais c'est très courant dans les codes, en particulier pour les affichages.

## Affichage : méthode de la déclaration friend

Déclaration :

sac-friend.hpp

```
10 class Sac {
11     public:
46     friend std::ostream &operator<<(std::ostream &out, const Sac& s);
53 };
56 std::ostream &operator<<(std::ostream &out, const Sac& s);
```

Implémentation :

sac-friend.cpp

```
50 std::ostream &operator<<(std::ostream &out, const Sac& s) {
51     out << "{";
52     for (int i = 0; i < int(s.tab_.size()); i++)
53         out << s.tab_[i] << ",";
54     return out << "}";
55 }
```

## Exemple de type abstrait courant : la liste

Voir [https://fr.wikipedia.org/wiki/Type\\_abstrait](https://fr.wikipedia.org/wiki/Type_abstrait)

### Compléments

#### Liste (angl. List)

- Insérer : ajoute un élément dans la liste (angl. insert, add)
- Retirer : retire un élément de la liste (angl. remove, delete)
- La liste est-elle vide ? : (angl. isNil, isEmpty)
- Nombre d'éléments dans la liste : (angl. length, size)

Applications : suite finie, type concret pour d'autres types abstraits.

## Exemple de type abstrait courant : la Pile

### Compléments

Pile (angl. Stack), Last-In-First-Out (dernier entré premier sorti).  
Push, Pop Liste (angl. List)

- Empiler : ajoute un élément sur la pile (angl. push)
- Dépiler : enlève un élément de la pile et le renvoie (angl. pop)
- La pile est-elle vide ? : (angl. isEmpty)
- Nombre d'éléments dans la pile : (angl. length, size)

Applications :

- algorithme récursif
- évaluation des expressions mathématiques
- fonction «annuler» d'un éditeur

## Exemple de type abstrait courant : la Pile

### Compléments

Pile (angl. Stack), Last-In-First-Out (dernier entré premier sorti).  
Push, Pop Liste (angl. List)

- Empiler : ajoute un élément sur la pile (angl. push)
- Dépiler : enlève un élément de la pile et le renvoie (angl. pop)
- La pile est-elle vide ? : (angl. isEmpty)
- Nombre d'éléments dans la pile : (angl. length, size)

Applications :

- algorithme récursif
- évaluation des expressions mathématiques
- fonction «annuler» d'un éditeur

## Exemple de type abstrait courant : la File d'attente

### Compléments

File d'attente (angl. Queue), First-In-First-Out (premier entré premier sorti)

- Enfiler : ajoute un élément sur la file (angl. enqueue)
- Défiler : enlève un élément de la file et le renvoie (angl. dequeue)
- La file est-elle vide ? : (angl. isEmpty)
- Nombre d'éléments dans la file : (angl. lenght, size)

Applications :

- mémoire tampon, communication asynchrone
- serveur d'impression (angl. *spool*)

## Exemple de type abstrait courant : la File d'attente

### Compléments

File d'attente (angl. Queue), First-In-First-Out (premier entré premier sorti)

- Enfiler : ajoute un élément sur la file (angl. enqueue)
- Défiler : enlève un élément de la file et le renvoie (angl. dequeue)
- La file est-elle vide ? : (angl. isEmpty)
- Nombre d'éléments dans la file : (angl. length, size)

Applications :

- mémoire tampon, communication asynchrone
- serveur d'impression (angl. *spool*)

## Exemple de type abstrait courant : l'ensemble

### Compléments

#### Ensemble (angl. Set)

- ajoute : ajoute un élément de l'ensemble (angl. add)
- supprime : enlève un élément de l'ensemble (angl. remove, delete)
- L'ensemble est-il vide ? : (angl. isEmpty)
- Nombre d'éléments dans l'ensemble : (angl. size)

#### Applications :

- concept mathématique d'ensemble fini

## Exemple de type abstrait courant : l'ensemble

### Compléments

#### Ensemble (angl. Set)

- ajoute : ajoute un élément de l'ensemble (angl. add)
- supprime : enlève un élément de l'ensemble (angl. remove, delete)
- L'ensemble est-il vide ? : (angl. isEmpty)
- Nombre d'éléments dans l'ensemble : (angl. size)

#### Applications :

- concept mathématique d'ensemble fini

## Exemples de types abstraits courants

- Sac / Multi-ensemble (angl. Bag, MultiSet)
- Table d'associations (angl. Map)
- Table d'associations multiples (angl. MultiMap)

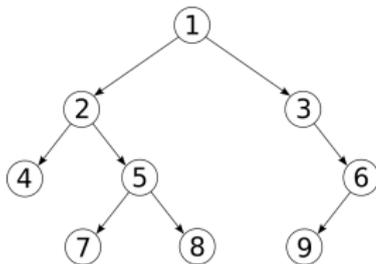
## Exemple de type abstrait courant : l'arbre

### Compléments

### Arbres (angl. Trees)

#### Applications :

- Compression de données (Huffman)
- Recherche (arbres binaires de recherche)
- Répertoire sur un disque
- Arbres de décisions



## Exemple de type abstrait courant : le graphe

### Compléments

### Graphes (angl. Graph)

Applications :

- Algorithme de routage
- Recherche de chemin
- Réseaux, Flots

