

Programmation Impérative III

Initiation au génie logiciel : modularité, abstraction de type, analyse

Florent Hivert

Mél : `Florent.Hivert@lri.fr`

Adresse universelle : `http://www.lri.fr/~hivert`

1 Qu'est-ce que le génie logiciel

2 Programmation modulaire

- Notion d'interface
- Principe d'encapsulation
- Exemples

1 Qu'est-ce que le génie logiciel

2 Programmation modulaire

Petit historique de l'évolution de l'informatique

- Jusqu'en 1985, les ordinateurs appartenaient 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...

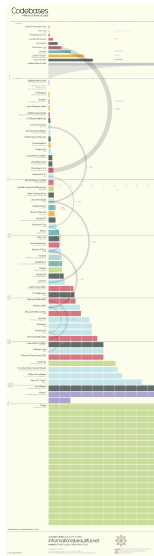
Retenir

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.

Lecture recommandée :

https://fr.wikipedia.org/wiki/Génie_logiciel

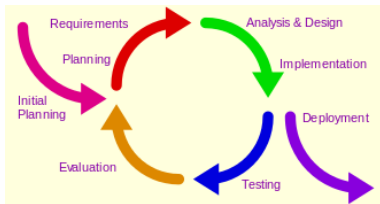
Retenir (arrêté ministériel du 30 décembre 1983)

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

1 Qu'est-ce que le génie logiciel

2 Programmation modulaire

- Notion d'interface
- Principe d'encapsulation
- Exemples

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.

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 sa réalisation.

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**.

Interface pour le Sac

bag/sac-interface.hpp

```

1  /** Créer un sac vide
2   * @param[out] e: le sac vide
3   */
4  void sacVide(Sac &e);
5
6  /** Ajoute un objet à un sac
7   * @param[in/out] e: le sac à modifier
8   * @param[in] n: l'objet à ajouter
9   */
10 void ajoute(Sac &e, Objet n);
11
12 /** Retire un objet à un sac
13 * @param[in/out] e: le sac à modifier
14 * @param[in] n: l'objet à retirer
15 * si n n'est pas dans e, ne fait rien
16 * @return si on a bien pris l'objet dans le sac
17 */
18 bool retire(Sac &e, Objet n);
19
20 /** Teste si le sac e est vide
21 * @param[in] e: le sac à tester
22 * @return: un booléen selon le résultat du test
23 */
24 bool estVide(const Sac &e);
25
26 /** Retourne un objet du sac e
27 * @param[in] e: le sac considéré
28 * @return: un objet de e si e n'est pas vide
29 *          le comportement est indéfini si e est vide
30 */
31 Objet objetSac(const Sac &e);

```

Remarque

L'interface

bag/sac-interface.hpp

```
void sacVide(Sac &e);
void ajoute(Sac &e, Objet n);
bool retire(Sac &e, Objet n);
bool estVide(const Sac &e);
Objet objetSac(const Sac & e);
```

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 précédente, on n'a pas décrit les types, il 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.*

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 la partie réalisation, 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.

Un type abstrait peut être défini **par spécialisation** d'un type abstrait existant.

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) et qu'on souhaite rendre plus **efficaces** en les optimisant pour le type concret choisi.

Dans notre exemple

■ Constructeur

```
void sacVide(Sac &e);
```

bag/sac-interface.hpp

■ Opérations de test

```
bool estVide(const Sac &e);
```

bag/sac-interface.hpp

■ Opérations d'accès

```
void ajoute(Sac &e, Objet n);  
bool retire(Sac &e, Objet n);  
Objet objetSac(const Sac & e);
```

bag/sac-interface.hpp

■ Autres opérations

```
int nbSac(const Sac &e);  
int nbObject(const Sac &e, Objet n);
```

bag/sac-interface.hpp

Boîte noire / boîte blanche

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**.

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**.

Principe d'encapsulation

Retenir

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

- permet de au concepteurs de modifier les structures de données internes sans modifier l'interface de celle-ci et donc sans affecter les utilisateurs.
- Fréquent lorsque l'on veut améliorer l'efficacité
- permet d'assurer l'intégrité de la structure de données
- évite l'effet plat de spaghettis, où l'on ne sait plus par qui, par quoi et comment les données sont modifiées

Encapsulation et langage de programmation

Remarque

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

public protected private

Comme nous n'utilisons pas les objets dans ce cours, ce principe est **sous la responsabilité du concepteur** (qui peut l'appliquer correctement ou pas).

Utilisations du type abstrait Sac

bag/marche.cpp

```

1  Course c;
2  Sac s;
3  int action;
4
5  sacVide(s);
6  switch (action) {
7  case 1:
8      if (estVide(s)) cout << "Le sac est vide";
9      else cout << "Le sac n'est pas vide";
10     break;
11 case 2: cout << "Quoi ?";
12     c = lireCourse();
13     ajoute(s, c);
14     break;
15 case 3: cout << "Quoi ?";
16     c = lireCourse();
17     if (not retire(s, c))
18         cout << "Il n'y a pas de " << stringCourse(c) << " dans le sac";
19     break;
20 case 4:
21     if (estVide(s)) cout << "Le sac est vide";
22     else
23         cout << "Dans le sac, il y a : " << stringCourse(objetSac(s));
24     break;
25 }
```

bag/test-sac.cpp

```
1 void testSacVide() {
2     Sac s;
3     sacVide(s);
4     ASSERT(estVide(s));
5 }
6
7 void testAjoute() {
8     Sac s;
9     sacVide(s);
10    ASSERT(estVide(s));
11    ajoute(s, pomme);
12    ASSERT(not estVide(s));
13 }
```

bag/test-sac.cpp

```
1 void testRetire() {
2     Sac s;
3     sacVide(s);
4     ajoute(s, pomme);
5     ASSERT(not estVide(s));
6     ASSERT(not retire(s, viande));
7     ASSERT(retire(s, pomme));
8     ASSERT(estVide(s));
9     ASSERT(not retire(s, pomme));
10
11    ajoute(s, pomme);
12    ajoute(s, pomme);
13    ASSERT(retire(s, pomme));
14    ASSERT(retire(s, pomme));
15    ASSERT(estVide(s));
16    ASSERT(not retire(s, pomme));
17
18    // Ces tests sont très incomplets
```

Réalisation 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**.

`bag/sac-inttab.hpp`

```
const int MaxSac = NbCourse;
```

```
struct Sac {  
    int t[MaxSac];  
};
```


R alisation 1 du type abstrait Sac (1)

bag/sac-inttab.cpp

```

1 void sacVide(Sac &e) {
2     for (int i=0; i < MaxSac; i++) e.t[i] = 0;
3 }
4
5 bool estVide(const Sac &e) {
6     for (int i=0; i < MaxSac; i++)
7         if (e.t[i] != 0) return false;
8     return true;
9 }
    
```

Réalisation 1 du type abstrait Sac (2)

bag/sac-inttab.cpp

```

1 void verifieBorne(int n) {
2     if (not (0 <= n and n < MaxSac)) {
3         std::cerr << "Sac en dehors des bornes" << std::endl;
4         exit(1);
5     }
6 }
7 void ajoute(Sac &e, Objet n) {
8     verifieBorne(n);
9     e.t[n]++;
10 }
11 bool retire(Sac &e, Objet n) {
12     verifieBorne(n);
13     if (e.t[n] == 0) return false;
14     e.t[n]--;
15     return true;
16 }

```

Réalisation 1 du type abstrait Sac (3)

bag/sac-inttab.cpp

```
1  Objet objetSac(const Sac &e) {  
2      int i;  
3      for (i=0; i < MaxSac; i++)  
4          if (e.t[i] != 0) return Objet(i);  
5      return Objet(i);  
6  }
```

Réalisation 2 du type abstrait Sac

Exemple

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

bag/sac-tab.hpp

```
const int MaxSac = NbCourse;  
const int Capacite = 100;  
struct Sac {  
    Objet t[Capacite];  
    int nb;  
};
```

Réalisation 2 du type abstrait Sac (1)

bag/sac-tab.cpp

```
1 void sacVide(Sac &e) {  
2     e.nb = 0;  
3 }  
4  
5 bool estVide(const Sac &e) {  
6     return e.nb == 0;  
7 }
```

Réalisation 2 du type abstrait Sac (2)

bag/sac-tab.cpp

```
1 void ajoute(Sac &e, Objet n) {  
2     verifieBorne(n);  
3     if (e.nb == Capacite) {  
4         std::cerr << "Sac plein" << std::endl;  
5         exit(1);  
6     }  
7     e.t[e.nb] = n;  
8     e.nb++;  
9 }
```

Réalisation 2 du type abstrait Sac (2)

bag/sac-tab.cpp

```
1  bool retire(Sac &e, Objet n) {
2      verifieBorne(n);
3      // Cherche un n dans le sac
4      for (int i = 0; i < e.nb; i++)
5          if (e.t[i] == n) {
6              e.nb--;
7              if (e.nb != 0) e.t[i] = e.t[e.nb];
8              return true;
9          }
10     return false;
11 }
```

Autres réalisations possibles

- par un `vector<Objet>`
- 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 du C++ qui permet de stocker une table qui à un objet associe un entier
`unordered_map<Objet, int>`
- la même chose avec un arbre binaire de recherche
`map<Objet, int>`

Compléments

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

Évolution du type abstrait (1)

Si l'on essaye de réaliser les fonctions

bag/sac-interface.hpp

```
/** Le nombre d'objets dans le sac
 * @param[in] : e un sac
 * @return le nombre d'objet dans le sac */
int nbSac(const Sac &e);

/** Le nombre d'occurrence d'un objet dans le sac
 * @param[in] : e un sac
 * @param[in] : n un objet
 * @return le nombre de fois où l'objet n est dans le sac */
int nbObject(const Sac &e, Objet n);
```

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

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

Résumé

Réalisation d'un type abstrait :

- choix du type concret : découle des contraintes *concrètes* liées à l'utilisation prévue des données du type abstrait ;
- réalisation des opérations du type abstrait :
 - 1 opérations de base : toute opération pour laquelle il est impératif de connaître le type concret
 - 2 autres opérations : utilisent les opérations de base pour accéder au type concret
- opérations externes : opérations utilisant le type abstrait sans lui être uniquement dédié (généralement, utilisent plusieurs types abstraits simultanément)
 - accès aux données uniquement au travers des opérations du type abstrait !

Exemple de type abstrait courants : la liste

Voir 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. lenght, size)

Applications : suite finie, type concret pour d'autre types abstrait.

Exemple de type abstrait courants : 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. lenght, size)

Applications :

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

Exemple de type abstrait courants : 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. lenght, size)

Applications :

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

Exemple de type abstrait courants : 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 tampons, communication asynchrone
- serveur d'impression (angl. *spool*)

Exemple de type abstrait courants : 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 tampons, communication asynchrone
- serveur d'impression (angl. *spool*)

Exemple de type abstrait courants : l'ensemble

Compléments

Ensemble (angl. Set) premier sorti)

- 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 courants : l'ensemble

Compléments

Ensemble (angl. Set) premier sorti)

- 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 courants

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

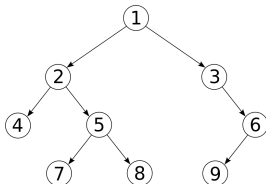
Exemple de type abstrait courants : l'arbre

Compléments

Arbres (angl. Trees)

Applications :

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



Exemple de type abstrait courants : le graphe

Compléments

Graphes (angl. Graph)

Applications :

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

