Cours Approche Objet

Lionel Clément

2017-2018

Table des matières

Informations administratives

- Code UE : 4TIN706U
- Cours : Amphi 1 Bât. A9 – Mardi 14 :00-16 :00 (5/09, 12/09, 19/09, 26/09, 3/10, 10/10, 17/10, 24/10, 7/11, 14/11, 21/11, 28/11)
- 4 groupes de TDs :
  - Lundi 16 :15-18 :15
  - Mardi 16 :15-18 :15
  - Mercredi 08 :00-10 :00
  - Vendredi 08 :00-10 :00
- 3 chargés de TD :
  - Patxi Laborde-Zubieta
  - Frédérique Carrère
  - Moi
- Contrôle des connaissances
  - 6 ECTS (environ 150 à 180 heures de travail)
  - Examen écrit 3h00 coef. 2/3
  - Contrôle continu 1/3 (deux notes : un TP noté, un DS)

Prérequis

- Connaissance de la programmation impérative
- Première utilisation d'un langage de programmation à objet (Java, C++, C#, Python, Php, Javascript, etc.)

Objectifs

- Comprendre le concept de programmation objet

— Concepts avancés : exceptions, clonage, classes génériques, collections, itérations, classes internes...
— Savoir développer une interface homme-machine
— Appliquer des modèles de conception
— Savoir tester les programmes

Références bibliographiques

— Joshua Bloch, "Effective Java, Programming Language Guide", Addison-Wesely 2001
— Maurice Naftalin et Philip Wadler, "Java Generics and Collections", O'Reilly 2006
— E. Gamma, R. Helm, R. Johnson et J. Vlissides, "Design Patterns. Catalogue de modèles de conception réutilisables", Vuibert, 1999

1 Introduction

Un programme informatique développé avec un langage de programmation impératif récent est modulaire et structuré. Cela suffit-il pour envisager de l'utiliser sur de très gros projets ?

Quels sont les problèmes qu'on rencontre avec un logiciel développé sans l'approche objet ?

Prenons le cas du développement d'un jeu vidéo : PacMan (petit rappel du jeu pour ceux qui ne connaissent pas)

— Est-on certain d'avoir réutilisé tout le code existant ?
  Exemple : Une procédure permettant à un personnage du jeu de se déplacer, peut-elle s'appliquer à tous les personnages ? La procédure pourra éventuellement être différente pour un personnage qui fuit, un autre qui attaque ou le joueur. Pourtant tous doivent éviter les obstacles de la même manière. Comment être certain que cette stratégie d'évitement convient à tous les types de personnages ?

— Peut-on tester tout le code ?
  Exemple : Si on teste l'évitement d'obstacle pour un personnage qui attaque, un autre qui fuit et enfin le joueur, doit-on écrire trois procédures de test ou une seule ?

— Peut-on organiser le développement de l'ensemble du jeu par une équipe en se partageant les tâches ?
  Exemple : Celui ou celle qui développe la procédure d'évitement des obstacles, peut-il le faire indépendamment de celui qui écrit la stratégie d'attaque ou de fuite dans le labyrinthe ?

— Peut-on reprendre une partie du code dans le cas d'une refonte importante ?
  Exemple : La demande est de porter le jeu pour une autre plateforme en utilisant une bibliothèque graphique différente. La partie graphique du jeu est-elle suffisamment indépendante pour que l'ensemble du code ne soit pas repris ?

— Peut-on distribuer le code facilement ?
  Exemple : Le code est documenté et confié en logiciel libre à la communauté. Est-il simple de faire des modifications, des ajouts pour améliorer le jeu en s'assurant qu'il est toujours exempt de bugs ?

Quiconque a développé un projet un peu ambitieux avec un langage de programmation non objet, sait qu'il est difficile, voire impossible de répondre favorablement à ces différentes questions.

D'autre part, regrouper en entités uniques les propriétés d'un même concept permet de concevoir de façon très différente un projet.

1.1 Bribes de l'histoire de la programmation objet

Dans les années 60, la programmation connait une période où l'on structure les programmes

— Algol 58, 60, 68 (1958-1975)

— Pascal (1971)

A la fin des années 60, on voit émerger le premier langage orienté objet :

— Simula I (1962) est destiné aux problèmes de simulation. C'est un sur-ensemble d'Algol qui intègre un mécanisme à événements discrets. C'est le premier langage qui regroupe sous une même entité données et procédures.

— Simula (1967) est le premier véritable langage de programmation orienté objet.
  Un programme est un ensemble d'objets qui sont autonomes et disposent de leurs propres données et procédures. Ceci permet de simuler des comportements parallèles.

Simula intègre la notion d'héritage. Les notions d'encapsulation et l'abstraction des données réhabilite la fusion des données et des programmes. Contredisant un principe fort de la programmation structurée des années 70-80. Ada et CLU sont issues de cette école.
Simula a servi comme modèle pour un ensemble de langages de programmation à typage statique dite de l'école scandinave (C++, Clascal, Object Pascal, Eiffel, Beta).

— Smalltalk-72, Smalltalk-76, Smalltalk-80
Smalltalk généralise la notion d'objet qui devient le seul élément de programmation (y compris au niveau de la structure des programmes). L'envoi de message est le seul moyen qui permet aux objets de communiquer entre eux. Smalltalk-76 introduit une relation d'héritage entre classes. Smalltalk-80 est de typage dynamique, il introduit la notion de métaclasse (classe dont les instances sont des classes).

— Flavors, Ceyx, Clos
Ces langages forment un mariage entre le langage de programmation Lisp et le paradigme objet.

2 Objet

2.1 Trio <entité, attribut, valeur>

L'information peut être représentée par un ensemble de trios comme <voiture, couleur, rouge>, <voiture, marque, Peugeot>, etc. Les entités se caractérisent par un ensemble de propriétés.

L'attribut prend une valeur en fonction de sa nature.

Les objets seront définis et stockés en mémoire sous la forme d'ensemble attribut/valeur.

Le stockage des objets se fait en typant les attributs selon des types primitifs, ou en référence à d'autres objets.

Rappelons que le référent est une variable informatique associée à un nom symbolique, codée sur k bits et contenant l'adresse physique d'un objet informatique. Plusieurs référents distincts peuvent désigner le même objet.

2.2 Envoi de message

Le seul mode de communication entre deux objets revient à appeler une méthode déclarée dans un objet depuis un autre objet. Ceci s'appelle l'envoi de message.

Synchrone par défaut, mais on peut concevoir une programmation par envoir de messages asynchrones

2.3 Objet composite

Les objets peuvent entrer dans une relation de type composition, où les uns se trouvent contenus dans les autres et ne sont accessibles qu'à partir de ces autres.

Premier exemple : un objet manufacturé contient l'ensemble des pièces qui le composent.

La composition d'objet induit nécessairement que l'existence du composite dépende de l'existence du composant. En d'autres termes, la destruction du composite extraînera la destruction du composant.

Listing 1 – Composition voiture.cpp

```
#include <iostream>

class Carburateur
{
public:
  Carburateur(){
    std::cerr << "Carburateur" << this << std::endl;
  }
  ~Carburateur(){
    std::cerr << "~Carburateur" << this << std::endl;
  }
  void message(){
    std::cerr << "Message carburateur" << this << std::endl;
  }
};

class Voiture
{
private:
  Carburateur carbu;
public:
  Voiture(){
    std::cerr << "Voiture" << this << std::endl;
  }
  ~Voiture(){
    std::cerr << "~Voiture" << this << std::endl;
  }
  void message(){
    std::cerr << "Message voiture" << this << std::endl;
    carbu.message();
  }
};

int
main(int argn,
     char **argv)
{
  class Voiture *voiture = new Voiture();
  voiture->message();
  delete voiture;
}
```

Listing 2 – Composition Carburateur.java

```
public class Carburateur {
    Carburateur(){
        System.out.println("Carburateur");
    }
}
```

Listing 3 – Composition Voiture.java

```
public class Voiture {
    private Carburateur carbu;
    public Voiture(){
        System.out.println("Voiture");
        carbu =  new Carburateur();
    }
}
```

Listing 4 – Composition Main.java

```
public class Main {
    public static void main(String[] args) {
        Voiture v1 = new Voiture();
        Voiture v2 = new Voiture();
    }
}
```

2.4 Objet agrégés ou associés

C'est une généralisation de la composition qui n'entraîne pas l'appartenance.

Exemple : un produit manufacturé contient des types de pièces qui peuvent être répertoriées dans un magasin de pièces de rechange. Ces types de pièces ne disparaissent pas avec la disparition de l'objet manufacturé.

Listing 5 – Agrégation C++

```
#include <iostream>

class Conducteur
{
public:
  Conducteur(){
    std::cerr << "Conducteur" << this << std::endl;
  }
  ~Conducteur(){
    std::cerr << "~Conducteur" << this << std::endl;
  }
  void message(){
    std::cerr << "Message conducteur" << this << std::endl;
  }
};

class Voiture
{
private:
  Conducteur *conducteur;
public:
  Voiture(Conducteur *conducteur){
    this->conducteur = conducteur;
    std::cerr << "Voiture" << this << std::endl;
  }
  ~Voiture(){
    std::cerr << "~Voiture" << this << std::endl;
  }
  void message(){
    std::cerr << "Message voiture" << this << std::endl;
    conducteur->message();
  }
};

int
main(int argn,
     char **argv)
{
  class Conducteur *conducteur = new Conducteur();
  class Voiture *voiture1 = new Voiture(conducteur);
  class Voiture *voiture2 = new Voiture(conducteur);
  voiture1->message();
  voiture2->message();
  delete voiture1;
```

```
  delete voiture2;
}
```

Listing 6 – Agrégation Conducteur.java

```
class Conducteur {
    Conducteur(){
        System.out.printf("Conducteur\n");
    }
}
```

Listing 7 – Agrégation Voiture.java

```
class Voiture {
    private Conducteur conducteur;
    public Voiture(Conducteur conducteur){
        System.out.printf("Voiture\n");
        this.conducteur = conducteur;
    }
}
```

Listing 8 – Agrégation Main.java

```
public class Main {
    public static void main(String[] args) {
        Conducteur c = new Conducteur();
        Voiture v1 = new Voiture(c);
        Voiture v2 = new Voiture(c);
    }
}
```

2.5 Objets dépendants

Listing 9 – Dépendance C++

```
#include <iostream>

class Conducteur
{
public:
  Conducteur(){
    std::cerr << "Conducteur" << this << std::endl;
  }
  ~Conducteur(){
    std::cerr << "~Conducteur" << this << std::endl;
  }
  void message(){
    std::cerr << "message conducteur" << this << std::endl;
  }
};

class Voiture
{
public:
  Voiture(){
    std::cerr << "Voiture" << this << std::endl;
  }
  ~Voiture(){
    std::cerr << "~Voiture" << this << std::endl;
  }
  void message(class Conducteur *conducteur){
    std::cerr << "message voiture" << this << std::endl;
    conducteur->message();
  }
};

int
main(int argn,
     char **argv)
{
  class Conducteur *conducteur = new Conducteur();
  class Voiture *voiture = new Voiture();
  voiture->message(conducteur);
  delete voiture;
}
```

Listing 10 – Dépendance Conducteur.java

```
class Conducteur {
    Conducteur(){
        System.out.printf("Conducteur\n");
    }

    void message(){
        System.out.println("Message " + this);
    }
}
```

Listing 11 – Dépendance Voiture.java

```
class Voiture {
    public Voiture(){
        System.out.println("Voiture");
    }

    public void message(Conducteur conducteur){
        System.out.println("Message " + this);
        conducteur.message();
    }

}
```

Listing 12 – Dépendance Main.java

```
public class Main {
    public static void main(String[] args) {
        Conducteur c = new Conducteur();
        Voiture v = new Voiture();
        v.message(c);
    }
}
```

— Agrégation forte = Agrégation par valeur = Composition
— Agrégation = Agrégation faible = Association

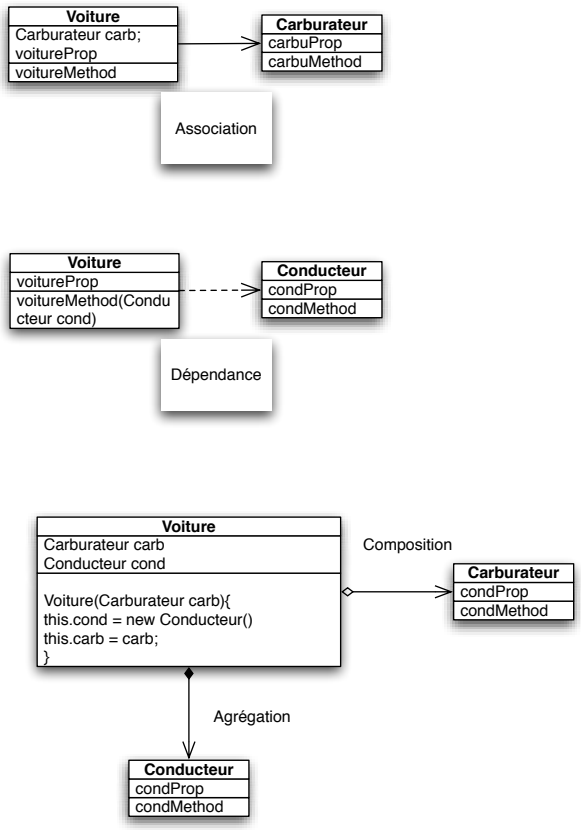

FIGURE 1 – **Association – Dépendance**

2.6 Encapsulation des attributs

Pourquoi les attributs d'un objet ne devraient-ils être accessible que par l'intermédiaure des méthodes locales à ces objets ?

— Intégrité des classes :
  Une classe et chargée de péserver l'intégrité des objets qu'elle définit.
— Gestion des exceptions
— Gestion des tests
— Cloisonnement des traitements, perennisation du code
— `public` : à tous ;
— `protected` : seulement aux objets de la classe elle-même et aux classes dérivées ;
— `private` : seulement par les objets de la classe elle-même ;
— `package friendly` par les objets de la classe elle-même et tout mon package

getters et *setters*

Règle générale. Si l'objet ne sert pas à partager des données :

— Attributs : `private` ou `protected`
— *getters* : `public` si l'on ne retourne pas de référence à des objets, sinon `private` ou `protected`
— *setters* : `private` ou `protected`

2.7 Encapsulation des méthodes

Les méthodes `private` sont responsables de l'implémentation de la classe, les méthodes `public` sont responsables de l'implémentation de l'*interface*.

— `public` : Membres, Dérivés, Clients ;
— `protected` : Membres, Dérivés ;
— `private` : Membres.
— `package friendly` Membres, Clients du package.

2.8 Création et suppression d'objets

2.8.1 Ramasse-miettes (*garbage collector*)

Listing 13 – Garbage Collector Carburateur.java

```
class Carburateur {
    public Carburateur(){
        System.out.println(this);
    }

    protected void finalize(){
        System.out.println("~" + this);
    }

}
```

Listing 14 – Garbage Collector Voiture.java

```
class Voiture {
    private Carburateur carbu;

    public Voiture(){
        System.out.println(this);
        carbu = new Carburateur();
    }

    protected void finalize(){
        System.out.println("~" + this);
    }
}
```

Listing 15 – Garbage Collector Main.java

```
public class Main {
    public static void main(String[] args) {
        Voiture v = new Voiture();
        v = new Voiture();
        v = null;
        System.gc();
    }
}
```

Listing 16 – Garbage Collector voiture.py

```
import gc
class Carburateur:
    def __init__(self):
        print ("Carburateur")
    def __del__(self):
        print ("~Carburateur")

class Voiture:
    def __init__(self):
        print ("Voiture")
        self.__carbu = Carburateur()
    def __del__(self):
        print ("~Voiture")

v1 = Voiture()
v1 = Voiture()
print ("coucou1")
#v1 = None
#gc.collect()
print ("coucou2")
```

2.9 Cohésion et couplage

- Cohésion
  - Traitements et Données forment un tout cohérent
  - Préférer petits objets
- Couplage
  - Minimiser communication entre objets

3 Classe

Une classe est un ensemble d'objets qui partagent les même types de propriétés et qui appliquent les mêmes traitements à leurs données.

Attention : l'identité est toujours portée par l'objet.

- Nom
- Attributs
- Méthodes

3.1 Instanciation

Un objet est construit à partir d'une classe.

Où se trouve-t-il en mémoire ? Comment est-il accessible ?

Constructeur

Destructeur

this, self

3.2 Héritage

— Est une sous-classe
— Est une classe fille
— « Est un »
— sous-ensemble
Relation : `A` hérite de `B` si les objets instance de `A` sont instances de `B`.

Du bon usage de l'héritage

— Extension de classe
— Adaptation du code
— Factorisation du code
— Extension de classe
On ajoute une méthode à une classe qui en étend une autre.
Exemple : Modification de la taille d'un ballon et de son élasticité (ballon gonflable).
— Adaptation du code
On `redéfinit` une méthode.
Exemple : Modification de la méthode conférant une force à une balle : Une balle de Jokari aura une force inverse proportionnelle à l'élastique en plus de son poids.
Méthode : On appelle `super` et on complète.
— Factorisation du code
On étend une classe abstraite qui contient du code destiné à être réutilisé à différents endroits du projet.
Exemple : Une balle contient l'implémentation de l'affichage d'un rond à l'écran. Selon le type de balles, on aura d'autres affichages, mais celui-ci sera partagé pour toutes les balles.

Classe, Classe abstraite et Interface

Java
— Une classe implémente une ou plusieurs interfaces
— Une classe étend une seule classe
C++, PHP, Python
— C++ : Une classe implémente un ou plusieurs headers
— Une classe étend une ou plusieurs classes

Listing 17 – Héritage A.java

```
public class A {
    public void message(){
        System.out.println("message d'un objet instance de A");
    }
}
```

Listing 18 – Héritage B.java

```
public class B extends A{
    public void message(){
        System.out.println("message d'un objet instance de B");
    }
}
```

Listing 19 – Héritage Main.java

```
public class Main {
    public static void main(String[] args) {
        A a = new A();
        B b = new B();
        a.message();
        a = b;
        a.message();
        // b = a; ERROR
        b = (B) a;
        b.message();
    }
}
```

4 Types abstraits de données (ADT) et classes

— Un ensemble d'instances T
— Un ensemble d'opérations sur les éléments de T
— Une encapsulation des propriétés et des opérations de T
— Une responsabilité des opérations sur T

Exemple en C (de Hanson via Narbel) :

Listing 20 – stack.h

```
#ifndef STACK_H
#define STACK_H

typedef struct stack *stack;

int empty(stack);
void push_front(stack, void *);
void push_back(stack, void *);
void *pop_front(stack);
void *pop_back(stack);

#endif // STACK_H
```

Listing 21 – stack.cpp

```
#include "stack.h"

struct stack {
  int count;
  struct element {
    void *x;
    struct element *next;
  } *head;
};

stack new_stack(void){
  stack stk = (stack) malloc(sizeof(struct stack));
  stk->count = 0;
  stk->head = NULL;
  return stk;
}

int empty(stack stk){
  return stk->count == 0;
}

void push_front(stack stk, void *x){
  struct element e = (stack) malloc(sizeof(struct element));
  e->x = x;
  e->next = stk->head;
  stk->head = e;
  stk->count++;
}
```

4.1 Modularité

Prenons le cas d'une pile implémentée sous la forme d'une liste chaînée et d'une pile implémentée sous la forme d'un tableau.

La méthode `push_front` n'est pas implémentée de la même manière. Pour autant, du point de vue d'une `ADT` « stack », c'est la même chose.

5 Polymorphisme

Polymorphisme

Implémentations différentes de la même interface pour un opérateur, une procédure ou une fonction donnée.

Polymorphisme ad-hoc / overloading / surcharge

Définition différente des fonctions, procédures et opérateurs homonymes selon les types des opérandes (exclu le type de self passé en paramètre en Python)

Exemples : surcharge d'opérateurs en C++, surcharge de constructeurs en Java, etc.

Exemple en C++

Listing 22 – generics-1.cc

```
#include <iostream>
using namespace std;

int calculerSomme(int operande1, int operande2){
  int resultat = operande1 + operande2;
  return resultat;
}

string calculerSomme(string operande1, string operande2){
  string resultat = operande1 + operande2;
  return resultat;
}

int myMax (int o1, int o2){
  return o1 > o2 ? o1 : o2;
}

string myMax (string o1, string o2){
  return o1 > o2 ? o1 : o2;
}

int main(){
  int n1 = 4, p1 = 12;
  cout << n1 << " + " << p1 << " = " << calculerSomme(n1, p1) << endl;

  string n2 = "4", p2 = "12";
  cout << n2 << " + " << p2 << " = " << calculerSomme(n2, p2) << endl;

  cout << "Le max de " << n1 << " et " << p1 << " est " << myMax(n1, p1) << endl;

  cout << "Le max de " << n2 << " et " << p2 << " est " << max(n2, p2) << endl;
  return 0;
}
```

Polymorphisme paramétrique / type variable / template / type générique

Même définition des fonctions, procédures et opérateurs homonymes avec un type variable
Exemples : Types variables en Ocaml, templates en C++ et Java

Polymorphisme d'héritage / overriding / redéfinition

Définition des méthodes par spécialisation. Quand les méthodes ont déjà été définies ou non (interface en Java, méthode virtuelle pure en C++), je parle encore d'overriding, bien que cette

terminologie ne soit pas toujours employée.

Exemple : Implémentation d'une méthode d'une interface en Java, Redéfinition d'une méthode héritée en Java, C++ ou Python.

Exemple en Java

Listing 23 – polymorphism-1/src/A.java

```
public class A {
        public A(){};
        public final void identifyMyself1(){
                System.out.println("Je suis un A et on ne revient pas dessus");
        }
        public void identifyMyself2(){
                System.out.println("Je suis un A sauf contre-ordre");
        }
}
```

Listing 24 – polymorphism-1/src/B.java

```
public class B extends A {

        public B(){};

        public void identifyMyself2(){
                System.out.println("Je suis un B");
        }



}
```

Listing 25 – polymorphism-1/src/Main.java

```
public class Main {

        public static void main(String[] args) {

                B b = new B();

                b.identifyMyself1();
                b.identifyMyself2();

                A a = new A();
                a = (A)b;

                a.identifyMyself1();
                a.identifyMyself2();

                }
}
```

Exemple en C++

Listing 26 – polymorphism-1.cc

```
#include <iostream>
using namespace std;

class A{
public:
  void identifyMyself(void){
    cout << "Je suis un A " << endl;
  }
};

class B : A {
 public:
  void identifyMyself(void ){
    cout << "Je suis un B " << endl;
  }
};

class C : A{
 public:
  void identifyMyself(void){
    cout << "Je suis un C " << endl;
  }
};

int main(){
  class A a;
  class B b;
  class C c;
  a.identifyMyself();
  b.identifyMyself();
  c.identifyMyself();
  return 0;
}
```

Exemple en C++

Listing 27 – polymorphism-2.cc

```
#include <iostream>
using namespace std;

class A{
public:
  void identifyMyself1(void){
    cout << "Je suis un A " << endl;
  }
  virtual void identifyMyself2(void){
    cout << "Je suis un A sauf contreordre" << endl;
  }
};

class B : public A {
 public:
  void identifyMyself1(void ){
    cout << "Je suis un B " << endl;
  }
  void identifyMyself2(void ){
    cout << "Je suis un B " << endl;
  }
};

int main(){
  cout << "class A a:" << endl;
  class A a;
  a.identifyMyself1();
  a.identifyMyself2();

  cout << endl << "class B b:" << endl;
  class B b;
  b.identifyMyself1();
  b.identifyMyself2();

  cout << endl << "class A &ra = b" << endl;
  class A &ra = b;
  ra.identifyMyself1();
  ra.identifyMyself2();


  return 0;
}
```

Classes génériques

Exemple en C++

Listing 28 – generics-1-bis.cc

```
#include <iostream>
using namespace std;

int *calculerSommeInt(int *operande1, int *operande2){
  int *resultat = new int(*operande1 + *operande2);
  return resultat;
}

string *calculerSommeString(string *operande1, string *operande2){
  string *resultat = new string(*operande1 + *operande2);
  return resultat;
}

int main(){
  void *(* calculerSomme)(void *, void *);
  int n1 = 4, p1 = 12;

  calculerSomme = (void* (*)(void*, void*))&calculerSommeInt;

  cout << n1 << " + " << p1 << " = " << *(((int *)(calculerSomme(&n1, &p1)))) << en

  string n2 = "4", p2 = "12";

  calculerSomme = (void* (*)(void*, void*))&calculerSommeString;

  cout << n2 << " + " << p2 << " = " << *(((string *)(calculerSomme(&n2, &p2)))) <<

  return 0;
}
```

Exemple en C++

Listing 29 – generics-2.cc

```
#include <iostream>
using namespace std;

template<class T>
T calculerSomme(T operande1, T operande2)
{
  T resultat = operande1 + operande2;
  return resultat;
}

template<typename T>
T myMax (T o1, T o2){
  return o1 > o2 ? o1 : o2;
}

int main(){
  int n1 = 4, p1 = 12;
  cout << n1 << " + " << p1 << " = " << calculerSomme(n1, p1) << endl;

  string n2 = "4", p2 = "12";
  cout << n2 << " + " << p2 << " = " << calculerSomme(n2, p2) << endl;

  cout << "Le max de " << n1 << " et " << p1 << " est " << myMax(n1, p1) << endl;

  cout << "Le max de " << n2 << " et " << p2 << " est " << max(n2, p2) << endl;
  return 0;
}
```

Exemple en C++

Listing 30 – generics-3.cc

```
#include <iostream>
using namespace std;

class A{
public:
  string toString(){return "Je suis un A ";}
};

class B{
public:
  string toString(){return "Je suis un B ";}
};

template<class T1, class T2>
string conc(T1 operande1, T2 operande2)
{
  string resultat = operande1.toString() + operande2.toString();
  return resultat;
}

int main(){
  class A a;
  class B b;

  cout << conc(a, b) << endl;
  return 0;
}
```

Exemple en C++

Listing 31 – generics-4.cc

```
#include <iostream>
using namespace std;

template <class T>
class Item {
public:
  T element;
  Item<T> *next;

  Item();
  ~Item();
};

template <class T>
Item<T>::Item(){
  cout << "Item" << endl;
}

template <class T>
Item<T>::~Item(){
  cout << "~Item" << endl;
}

template <class T>
class LinkedList{

private:
  Item<T> *head;

public:
  LinkedList();
  ~LinkedList();

  T pop();
  void push(T);
  bool empty();
  void flush();
};

template <class T>
LinkedList<T>::LinkedList(){
  cout << "LinkedList" << endl;
  head = NULL;
```

```
}

template <class T>
LinkedList<T>::~LinkedList(){
  cout << "~LinkedList" << endl;
}

template <class T>
void LinkedList<T>::push(T element)
{
  Item<T> *tmp = new Item<T>();
  tmp->element = element;
    tmp->next = head;
    head = tmp;
    return;
}

template <class T>
T LinkedList<T>::pop()
{
    T tmp;
    Item<T> *ptmp = head;
    if (head != NULL){
        tmp = head->element;
        head = head->next;
        delete ptmp;
    }
    return tmp;
}

template <class T>
bool LinkedList<T>::empty()
{
  return head == NULL;
}

template <class T>
void LinkedList<T>::flush()
{
  while(head != NULL)
    pop();
}


int main(){
```

```
  LinkedList<string> l;
  l.push("A");
  l.push("B");
  l.push("C");
  cout << l.pop() << endl;
  cout << l.pop() << endl;
  l.flush();
  return 0;
}
```

Exemple en C++

Listing 32 – generics-5.cc

```
#include <iostream>
using namespace std;

class A{
public:
  string toString();
};

class B extends A{
public:
  string toString(){return "Je suis un B ";}
};

class C extends A{
public:
  string toString(){return "Je suis un C ";}
};

template<class T1, class T2>
string conc(T1 operande1, T2 operande2)
{
  string resultat = operande1.toString() + operande2.toString();
  return resultat;
}

int main(){
  class A a;
  class B b;
  class C c;

  cout << conc(a, b) << endl;
  return 0;
}
```

5.0.1 Héritage multiple

Listing 33 – Héritage multiple

```
class A {
public:
  void fa() { /* ... */ }
protected:
  int _x;
};

class B {
public:
  void fb() { /* ... */ }
protected:
  int _x;
};

class C: public B, public A {
public:
  void fc();
};
void C::fc() {
  int i;
  fa();
  //i = _x; // ERROR
  i = A::_x + B::_x; // resolution
}

int main(){
}
```

6 Interface Homme-Machine

6.1 MVC

Présentation à l'écran d'un exemple complet d'implémentation Java d'HIM.

- Windows : JWindow, JFrame, JDialog
- Layout : BorderLayout, BoxLayout, CardLayout, FlowLayout, GridBagLayout, GridLayout, GroupLayout, SpringLayout
- Container
  JFrame ⇒ Content pane JFrame ⇒ Menu Bar `frame.getContentPane().add(...)`
- Widget
  JButton, JTextfield, etc.

7 Design Patterns

7.1 Introduction

Rappelons les objectifs :
— Augmenter la modularité
— Augmenter la cohésion Exemple : MVC
— Abaisser la couplage
Deux principes fondamentaux :
— Préférer programmer des interfaces
Interfaces des objets :
Chaque objet déclare un ensemble d'opérations :
— Nom
— Objets qu'elle prend en paramètre
— Objet retourné
C'est précisément ce qui définit le type d'un objet.
Si l'interface d'un objet `A` contient l'interface d'un objet `B`, alors le type de B est un sous-type de l'interface de B. Il y a héritage.
Il ne faut pas confondre héritage de classe et héritage d'interface :
— Héritage de classe : héritage des implémentations
— Héritage d'interfaces : sous-typage

Exemple Classe abstraite en C++ : Héritage d'interface

Listing 34 – signature.hh

```
#ifndef SIGNATURE_H
#define SIGNATURE_H
#include <string>
using namespace std;

class Signature {

private:
  string value;
  void evalSignature();

public:
  Signature();
  void setSignatureValue(string);
  string getSignatureValue();
  string signature();
  void resetSignature();
  virtual string toString() const=0;

};

#endif // SIGNATURE_H
```

Listing 35 – arg.hh

```
#ifndef ARG_H
#define ARG_H

#include "signature.hh"

class Arg: public Signature {

private:
  int data;

public:
  Arg();
  string toString() const;
};

#endif // ARG_H
```

Listing 36 – arg.cc

```
#include "arg.hh"
#include <sstream>

Arg::Arg() {
  data=0;
}

string Arg::toString() const{
  std::ostringstream oss;
  oss << data;
  return oss.str();
}

Signature::Signature(){ value=""; }
string Signature::getSignatureValue(){ return value; }
void Signature::setSignatureValue(string value){ this->value=value; }
string Signature::signature(){ evalSignature(); return value; }
void Signature::evalSignature(){
  if (value == "")
    value = toString();
}
void Signature::resetSignature(){
  value="";
  evalSignature();
}

int main(int argn, char** argv){
  return 0;
}
```

Exemple héritage privé en C++ : Héritage de code

Listing 37 – array.hh

```
#ifndef ARRAY_H
#define ARRAY_H

class Array {

private:
  int* array;

public:

  Array();
  ~Array();
  int pop_front();
  void push_front(int);

};

#endif // ARRAY_H
```

Listing 38 – stackArray.hh

```
#ifndef STACKARRAY_H
#define STACKARRAY_H

#include "array.hh"

class StackArray: private Array {

public:
  StackArray();
  ~StackArray();

  int pop();
  void push(int);

};

#endif // STACKARRAY_H
```

Listing 39 – stackArray.cc

```
#include "array.hh"
#include "stackArray.hh"
#include <iostream>
```

```
using namespace std;

StackArray::StackArray(): Array(){
  cout << "StackArray" << endl;
}

StackArray::~StackArray(){
  cout << "~StackArray" << endl;
}

int StackArray::pop(){
  pop_front(); cout << "pop" << endl;
}

void StackArray::push(int k){
  push_front(k); cout << "push " << k << endl;
}

Array::Array(){
  cout << "Array" << endl;
}

Array::~Array(){
  cout << "~Array" << endl;
}

int Array::pop_front(){
  cout << "pop_front" << endl;
}

void Array::push_front(int k){
  cout << "push_front " << k << endl;
}

int main(int argn, char** argv){
  StackArray stack;
  stack.push(1);
  stack.pop();
  // ERROR stack.push_front(1);
  return 0;
}
```

Pourquoi manipuler les objets par classes abtraites ?
Ceci procure au développeur des classes héritées un double avantage :
— Il peut ignorer les détails d'implémentation des classes, et ne savoir que les interfaces
— Il peut ignorer quel objet envoie le message
Exemple : Classe `Signature`

Exemple de composition en C++

Listing 40 – array.hh

```
#ifndef ARRAY_H
#define ARRAY_H

class Array {

private:
  int* array;

public:

  Array();
  ~Array();
  int pop_front();
  void push_front(int);

};

#endif // ARRAY_H
```

Listing 41 – stackArray.hh

```
#ifndef STACKARRAY_H
#define STACKARRAY_H

#include "array.hh"

class StackArray {

private:
  Array data;

public:
  StackArray();
  ~StackArray();

  int pop();
  void push(int);

};

#endif // STACKARRAY_H
```

Listing 42 – stackArray.cc

```
#include "array.hh"
#include "stackArray-2.hh"
#include <iostream>
using namespace std;

StackArray::StackArray(){
  cout << "StackArray" << endl;
}

StackArray::~StackArray(){
  cout << "~StackArray" << endl;
}

int StackArray::pop(){
  data.pop_front(); cout << "pop" << endl;
}

void StackArray::push(int k){
  data.push_front(k); cout << "push " << k << endl;
}

Array::Array(){
  cout << "Array" << endl;
}

Array::~Array(){
  cout << "~Array" << endl;
}

int Array::pop_front(){
  cout << "pop_front" << endl;
}

void Array::push_front(int k){
  cout << "push_front " << k << endl;
}

int main(int argn, char** argv){
  StackArray stack;
  stack.push(1);
  stack.pop();
  //stack.data.push_front(2); ERROR: private!
  return 0;
}
```

— Préférer la composition à l'héritage
Deux techniques pour faire réutiliser des fonctionnalités :

1. Héritage de classe
2. Composition d'objets

— Spécificités

1. Héritage de classe Boîte blanche : le contenu des classes parentes est visible depuis les classes héritées.

   Définition statique à la compilation.

   Possibilité de surcharger des opérations.
2. Composition d'objets Boîte noire : le contenu de l'objet est accessible depuis son interface.

   Définie dynamiquement lors de création de références à d'autres objets.

— Avantages

1. Héritage de classe

   Facilité de compléter un composant pour en créer un nouveau.
2. Composition d'objets

   Conserve l'encapsulation de chaque classe.

— Inconvénients

1. Héritage de classe

   Pas possible de modifier le code hérité.

   Rompt l'encapsulation : modification de classe parente impose modification des classes héritées.
2. Composition d'objets

   Coût en nombre d'objets... perte d'efficacité à l'exécution.

7.2 Singleton

Listing 43 – Singleton

```
public class Main {

        public static void main(String[] args) {
                Uniq instance = Uniq.instance(5);
                instance.message();
                instance = Uniq.instance(6);
                instance.message();
                }

}

public class Uniq {

        private static Uniq instance;
        private int data;

        private Uniq(int data){
                this.data=data;

        }

        public static Uniq instance(int data){
                if (instance==null)
                        instance = new Uniq(data);
                return instance;
        }

        public void message(){
                System.out.println(data);
        }
}
```

7.3 Abstract Factory

Listing 44 – Abstract Factory

```
public abstract class AbstractFactory {
        abstract Shape createSquare();
        abstract Shape createRectangle();
}

public class ColoredRectangle implements Shape {

        @Override
        public void message() {
                System.out.println("I am a Colored Rectangle");
        }

}
public class ColoredShapeFactory extends ShapeFactory {

        @Override
        Shape createRectangle() {
                return new ColoredRectangle();
        }

}

public class Main {

        public static void main(String[] args) {
                ShapeFactory shapeFactory = new ShapeFactory();
                ShapeFactory coloredShapeFactory = new ColoredShapeFactory();
                Shape s1 = shapeFactory.createSquare();
                Shape s2 = shapeFactory.createRectangle();
                Shape s3 = coloredShapeFactory.createRectangle();
                s1.message();
                s2.message();
                s3.message();
        }

}

public class Rectangle implements Shape {

        @Override
        public void message() {
```

```
                System.out.println("I am a Rectangle");

        }

}

public class ShapeFactory extends AbstractFactory {

        @Override
        Shape createSquare() {
                return new Square();
        }

        @Override
        Shape createRectangle() {
                return new Rectangle();
        }

}

public interface Shape {

        public void message();
}

public class Square implements Shape {

        @Override
        public void message() {
                System.out.println("I am a Square");
        }

}
```

7.4 Adapter

Listing 45 – Adapter

```
public class ColoredRectangle {

        void drawWithColor(){
                // draw a colored rectangle
        }

}

public class Rectangle implements Shape {

        private ColoredRectangle cr;

        public Rectangle(){
                cr = new ColoredRectangle();
        }

        @Override
        public void draw() {
                cr.drawWithColor();
        }

}

public interface Shape {

        public void draw();

}
```

7.5 Template method

Listing 46 – Template Method

```
public abstract class A {

        void method(){
                primitive_method();
        }

        abstract void primitive_method();

}


public class B extends A {

        @Override
        void primitive_method() {

        }

}
```

7.6 State

Listing 47 – State

```
#include <iostream>

using namespace std;

class IState
{
public:
  virtual void handle(class Context* m)=0;
};

class Context
{
private:
  class IState* state;

public:
  Context();
  void setState(IState* s) {
    state = s;
  }
  void request(){
    state->handle(this);
  }
};

class ON: public IState
{
public:
  void handle(Context*);
};

class OFF: public IState
{
public:
  void handle(Context*);
};

void ON::handle(Context* c)
{
  cout << "on" << endl;
  c->setState(new OFF());
  delete this;
```

```
}

void OFF::handle(Context* c)
{
  cout << "off" << endl;
  c->setState(new ON());
  delete this;
}

Context::Context()
{
  state = new OFF();
}

int main()
{
  Context context;
  context.request();
  context.request();
  context.request();
  context.request();
}
```

7.7 Observer

Listing 48 – Observer

```
#include <string>
#include <list>
#include <iostream>

using namespace std;

class IObserver {
public:
  virtual void notify(int) = 0;
};

class IObservable {
public:
  virtual void notifyObservers() = 0;
  virtual void addObserver(class Observer*) = 0;
  virtual void removeObserver(class Observer*) = 0;
};

class Observer: public IObserver {
private:
  int state;

public:
  void notify(int state) {
    cout << "receipt from " << this << ": " << state << endl;
  }
};

class Observable: public IObservable {
private:
  list<Observer*> observers;
  int state;

public:
  Observable() {
    state = 0;
  }

  void setState(int state) {
    this->state = state;
  }
```

```
  void notifyObservers() {
    for (list<class Observer*>::const_iterator it = observers.begin(); it != observ
      (*it)->notify(state);
  }

  void addObserver(Observer* observer) {
        observers.push_front(observer);
    }

  void removeObserver(Observer* observer) {
        observers.remove(observer);
  }
};

class Observable1: public Observable {

public:

  void run() {
    while(true) {
      int input;
      cin >> input;
      setState(input);
      notifyObservers();
    }
  }
};


int main(){

  Observer observer1, observer2, observer3, observer4, observer5, observer6;
  Observable1 observable1;
  observable1.addObserver(&observer1);
  observable1.addObserver(&observer2);
  observable1.addObserver(&observer3);
  observable1.addObserver(&observer4);
  observable1.addObserver(&observer5);
  observable1.addObserver(&observer6);
  observable1.run();
  return 0;
}
```

Listing 49 – Observer

```
package fr.ubordeaux.oa;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

public class Controller implements ActionListener {

        Model model;
        View view;

        public Controller(Model model, View view){
                this.model=model;
                this.view=view;
                view.getButton().addActionListener(this);
                for (JLabelObserver i: view.getTargetLabels())
                        model.addObserver(i);
        }

        @Override
        public void actionPerformed(ActionEvent arg0) {
                model.setSource(Integer.parseInt(view.getTextfield().getText()));
                model.calculate();
        }


}
package fr.ubordeaux.oa;

import java.util.Observable;
import java.util.Observer;

import javax.swing.JLabel;

public class JLabelObserver extends JLabel implements Observer {

        private int index;

        public JLabelObserver(int index) {
                super();
                this.index=index;
        }

        @Override
        public void update(Observable observable, Object object) {
```

```
                Model model = (Model)observable;
                this.setText(String.valueOf(model.getTargets()[index]));
        }

}
package fr.ubordeaux.oa;

public class Main {

        public static void main(String[] args) {
                Model model = new Model();
                View view = new View();
                new Controller(model, view);
        }

}
package fr.ubordeaux.oa;

import java.util.Observable;

public class Model extends Observable {

        private int source;
        private int targets[];

        public Model(){
                targets = new int[10];
        }

        public void setSource(int source) {
                this.source = source;
        }

        public void calculate(){
                for (int i=0; i<=9; i++)
                        targets[i] = source * (i+1);
                this.setChanged();
                this.notifyObservers();
        }

        public int[] getTargets() {
                return targets;
        }
```

```
}
package fr.ubordeaux.oa;

import java.awt.GridLayout;
import java.util.Observable;
import java.util.Observer;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JTextField;

public class View {

	private JFrame frame;
	private JPanel panel;
	private JTextField textfield;
	private JLabel sourceLabels[];
	private JLabelObserver targetLabels[];
	private JButton button;

	public View(){
		frame = new JFrame("Convertisseur");
		//frame.setSize(5000, 100);
		panel = new JPanel();
		frame.add(panel);
		GridLayout layout = new GridLayout();
		layout.setColumns(12);
		layout.setRows(2);
		panel.setLayout(layout);

		textfield = new JTextField(6);
		sourceLabels = new JLabel[10];
		targetLabels = new JLabelObserver[10];
		for (int i=0; i<=9; i++){
			sourceLabels[i] = new JLabel(String.valueOf(i+1));
			targetLabels[i] = new JLabelObserver(i);
		}
		button = new JButton("=");
		panel.add(textfield);
		panel.add(button);
		for (JLabel label: sourceLabels)
			panel.add(label);
		panel.add(new JLabel());
```

```
                panel.add(new JLabel());
                for (JLabel label: targetLabels)
                        panel.add(label);

                frame.pack();
                frame.setVisible(true);
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }

        public JTextField getTextfield() {
                return textfield;
        }

        public JButton getButton() {
                return button;
        }

        public JLabelObserver[] getTargetLabels() {
                return targetLabels;
        }

}
```

8 Exceptions

Une exception est une condition exceptionnelle pendant l'exécution d'un programme. Par exemple, l'accès à une zone de mémoire inaccessible, l'accès à un indice inexistant d'un tableau, l'écriture dans un fichier protégé, l'envoi d'un signal par le système d'exploitation ou une division par zéro.

La gestion des exceptions dans un langage de programmation impératif consiste à séparer l'exécution normale du programme de l'exécution exceptionnelle. Le but est d'éviter un plantage du processus et sa continuation vers l'affichage d'un message d'erreur ou encore la réparation avant poursuite.

8.1 Exceptions en Java

Java propose une implémentation de la gestion des exceptions sans reprise ni réparation. Il s'agit simplement de détourner le code par l'intermédiaire de l'instruction `try/catch/finally`.

Une exception est un objet de type `java.lang.Throwable`. Leurs sous-classes sont `Error` et `Exception`.

On déclare une exception dans la signature d'une méthode : `throws`

```
public static String readFile(String filename) throws IOException {
...
}
```

L'instruction `try/catch/finally`

L'instruction `try` contient le code à exécuter controlé par un ensemble de blocs d'exception `catch`. Enfin, le bloc facultatif `finally` est exécuté quoi qu'il arrive.

L'instruction `throw` prend en argument un objet dont le type est `Throwable`. Elle saisie l'exception

```
public class Main {
        public static void main(String[] args) throws Throwable {
                try{
                        System.out.println("Begin");
                        throw new Throwable();
                        //System.out.println("Continue");
                } catch (Throwable e) {
                        System.out.println("Exception");
                        System.out.println(e.toString());
                        System.out.println("End exception");
                } finally {
                        System.out.println("End");
                }
        }
}
```

Listing 50 – Exception 1

```
package e1;

public class Main {

	public static void main(String[] args) {
		int i, j, k;
		i=1;
		j=0;
		try{
			k=i/j;
		} catch (ArithmeticException e){
			e.printStackTrace();
			System.out.println(e.toString());
		} finally {
			System.out.println("ok");
		}


	}

}
```

Listing 51 – Exception 2

```
package e2;

        public class MyException extends Exception {

                public MyException(String string) {
                        super(string);
                }

}
```

Listing 52 – Exception 2

```
package e2;

public class Main {

        public static void main(String[] args) throws MyException {
                try{
                        throw new MyException("ceci est le message");
                }
                catch (MyException e){
                        System.err.println(e.getMessage());
                }


        }

}
```

Listing 53 – Exception 3

```
public class MyException extends Exception {


        public MyException(String string, Exception e) {
                System.err.println("Message: " + string);
                System.err.println("Cause: " + e.getMessage());
        }



}
```

Listing 54 – Exception 3 (suite)

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

public class Main {

        public static void main(String[] args) throws MyException {
                String data = readFile();
                System.out.println(data);
        }

        private static String readFile() throws MyException {
                //File file = new File("doesnotexist");
                File file = new File("unreadable");
                FileReader reader = null;
                StringBuffer data = new StringBuffer();
                char[] cbuf = new char[1024];
                try{
                                reader = new FileReader(file);
                                int lenght = 0;
                                while ((lenght=reader.read(cbuf)) > 0)
                                        data.append(cbuf, 0, lenght);

                } catch (IOException e) {
                        throw new MyException("Erreur lecture", e);
                } finally {
                        try {
                                if (reader!=null)
                                        reader.close();
                        } catch (IOException e) {
                                e.printStackTrace();
```

```
                }
            }
            return data.toString();
        }

}
```

9 Clonage

Listing 55 – Clone

```
public class A implements Cloneable {

        public int attr;
        public B link;

        public A(int attr, B link) {
                this.attr=attr;
                this.link=link;
        }

        public boolean equals(Object o){
                if (this==o)
                        return true;
                else {
                        if (o instanceof A){
                                if ((attr == ((A)o).attr) &&
                                        (link.equals(((A)o).link)))
                                                return true;
                                else
                                        return false;

                        }
                }
                return false;
        }

        public Object clone() throws CloneNotSupportedException {
                A newA = (A)super.clone();
                newA.link = (B)link.clone();
                return newA;
        }



}
```

Listing 56 – Clone

```
public class B implements Cloneable {

        public int attr;

        public B(int attr) {
```

```
                this.attr = attr;
        }

        public boolean equals(Object o){
                if (this==o)
                        return true;
                else {
                        if (o instanceof B){
                                if (attr == ((B)o).attr)
                                        return true;
                                else
                                        return false;

                        }
                }
                return false;
        }

        public Object clone() throws CloneNotSupportedException {
                return super.clone();
        }

}
```

Listing 57 – Clone

```
public class Main {

        public static void main(String[] args) {
                B b = new B(5);
                B otherb = new B(5);
                B auxb = new B(10);

                auxb = b;

                A a = new A(10, b);
                A othera = new A(10, otherb);
                A auxa = null;
                try {
                        auxa = (A) othera.clone();
                } catch (CloneNotSupportedException e) {
                        e.printStackTrace();
                }

                if (b == otherb)
```

```
                    System.out.println("b = otherb");
                if (b.equals(otherb))
                    System.out.println("b equals otherb");

                if (b == auxb)
                    System.out.println("b = auxb");
                if (b.equals(auxb))
                    System.out.println("b equals auxb");

                if (a == othera)
                    System.out.println("a = othera");
                if (a.equals(othera))
                    System.out.println("a equals othera");

                if (a == auxa)
                    System.out.println("a = auxa");
                if (a.equals(auxa))
                    System.out.println("a equals auxa");


        }

}
```

```
b equals otherb
b = auxb
b equals auxb
a equals othera
a equals auxa
```

10 JUnit3 et JUnit4

Listing 58 – Exception 2

```
package test;
import static org.junit.Assert.assertEquals;

import org.junit.*;

import calculette.Calculette1;
import calculette.Calculette2;

public class Calculette1Test {

	@BeforeClass
	public static void beforeClass(){
		System.out.println("Before Class");
	}

	@AfterClass
	public static void afterClass(){
		System.out.println("After Class");
	}

	@Before
	public void before(){
		System.out.println("Before");
	}

	@After
	public void after(){
		System.out.println("After");
	}

	@Test
	public void test1() {
		Calculette1 cal = new Calculette1();

		assertEquals("would be 5", 5, cal.evaluate("5"));
		assertEquals("would be 6", 6, cal.evaluate("5+1"));
		assertEquals("would be 7", 7, cal.evaluate("5+1+1"));
		assertEquals("would be 20", 20, cal.evaluate("5*4"));
		assertEquals("would be 22", 22, cal.evaluate("5*4+2"));



	}
```

```
        @Test
        public void test2(){
                Calculette2 cal = new Calculette2();
                assertEquals(38, cal.evalE("(35)+3"));


        }

        @Test(expected = java.lang.NumberFormatException.class)
        public void test3(){
                Calculette2 cal = new Calculette2();
                cal.evaluate("(3+5)+3");
        }


}
```

11 SOLID

Rappels
- Objectifs
  - Développement, évolution plus simple
  - Travail distribué
  - Qualité augmentée
- Réalisations
  - Travail modulaire
  - Peu de modifications lors des évolutions du projet
  - Réutilisation du code

Étapes d'un projet :
- Cahier des charges
- Conception
- Implémentation
- Tests unitaires
- Intégration
- Tests fonctionnels
- Mise en production, évolution, correction

Modifications :
- Implémenter de nouvelles fonctionnalités, ne pas toucher aux anciennes
- Ne pas réécrire, mais redéfinir.
- Adapter l'architecture

SOLID Robert Cecil Martin,
- — Single Responsibility Principle
- — Open/Closed Principle
- — Liskov Substitution Principle
- — Interface Segregation Principle
- — Dependency Inversion Principle

Single Responsibility Principle

A class should have one, and only one, reason to change.

Open/Closed Principle

You should be able to extend a classes behavior, without modifying it.

Liskov Substitution Principle

Derived classes must be substitutable for their base classes.

Interface Segregation Principle

Make fine grained interfaces that are client specific.

Dependency Inversion Principle

Depend on abstractions, not on concretions.

12 Classes membres

- — Classe membre statique

Listing 59 – Membre Static

```
public class LinkedStack {

        public static interface Linkable{
                public Linkable getNext();
                public void setNext(Linkable next);
        }

        Linkable head;
        public void push(Linkable node){}
        public Object pop(){return null;}


}
```

Listing 60 – Membre Static

```
public class LinkableInteger implements LinkedStack.Linkable {

        int data;
        LinkedStack.Linkable next;

        @Override
        public LinkedStack.Linkable getNext() {
                return next;
        }

        @Override
        public void setNext(LinkedStack.Linkable next) {
                this.next = next;

        }

}
```

— Classe membre

Listing 61 – Membre

```
public class LinkedStack {

        public static interface Linkable{
                public Linkable getNext();
                public void setNext(Linkable next);
        }

        private Linkable head;
        public void push(Linkable node){}
        public Object pop(){return null;}

        protected class Enumerator {
                Linkable current;
                public Enumerator(){current = head;}
                public Object nextElement(){
                        Object value = current;
                        current = current.getNext();
                        return value;
                }
        }


}
```

Listing 62 – Membre

```
public class LinkableInteger implements LinkedStack.Linkable {

        int data;
        LinkedStack.Linkable next;

        @Override
        public LinkedStack.Linkable getNext() {
                return next;
        }

        @Override
        public void setNext(LinkedStack.Linkable next) {
                this.next = next;

        }

}
```

— Classe locale

Listing 63 – Locale

```
public class LinkedStack {

        public static interface Linkable{
                public Linkable getNext();
                public void setNext(Linkable next);
        }

        private Linkable head;
        public void push(Linkable node){}
        public Object pop(){return null;}

        public java.util.Enumeration enumerate(){

                class Enumerator implements java.util.Enumeration{
                        Linkable current;
                        public Enumerator(){current = head;}
                        public Object nextElement(){
                                Object value = current;
                                current = current.getNext();
                                return value;
                        }
                        @Override
                        public boolean hasMoreElements() {
```

```
                                        // TODO Auto-generated method stub
                                        return false;
                                }
                }

                return new Enumerator();
                }



}
```

Listing 64 – Locale

```
public class LinkableInteger implements LinkedStack.Linkable {

        int data;
        LinkedStack.Linkable next;

        @Override
        public LinkedStack.Linkable getNext() {
                return next;
        }

        @Override
        public void setNext(LinkedStack.Linkable next) {
                this.next = next;

        }

}
```

— Classe anonyme

Listing 65 – Anonyme

```
public class LinkedStack {

        public static interface Linkable{
                public Linkable getNext();
                public void setNext(Linkable next);
        }

        private Linkable head;
        public void push(Linkable node){}
        public Object pop(){return null;}
```

```
        public java.util.Enumeration enumerate(){

                return new java.util.Enumeration(){
                        Linkable current;
                        {current = head;}
                        public Object nextElement(){
                                Object value = current;
                                current = current.getNext();
                                return value;
                        }
                        @Override
                        public boolean hasMoreElements() {
                                // TODO Auto-generated method stub
                                return false;
                        }
        };

}



}
```

13 Thread

Listing 66 – thread

```
public class Terrain {

}
```

Listing 67 – thread

```
public class Ballon extends Thread implements Runnable {

        private Terrain terrain;

        public Ballon(Terrain terrain) {
                super();
                this.terrain = terrain;
                //this.setPriority(MAX_PRIORITY);
        }

        void message(){
//              synchronized (terrain) {

                        for (int i = 0 ; i<= 30; i++){
                                System.out.println("Ballon bouge");
                                //this.yield();
                        }
        //      }
        }

        public void run(){
                message();
        }

}
```

Listing 68 – thread

```
public class Raquette extends Thread implements Runnable {

        private Terrain terrain;

        public Raquette(Terrain terrain) {
                super();
```

```
                this.terrain = terrain;
                //this.setPriority(MIN_PRIORITY);
        }

        void message() throws InterruptedException{

                //synchronized (terrain) {
                        for (int i = 0 ; i<= 15; i++){
                                System.out.println("Raquette bouge");
                        }
                //}

                Thread.sleep(100);
                //this.yield();

                //synchronized (terrain) {
                        for (int i = 0 ; i<= 15; i++){
                                System.out.println("Raquette bouge");
                        }
                //}
        }

        public void run(){
                try {
                        message();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }
        }



}
```

Listing 69 – thread

```
public class Main {

        public static void main(String[] args) {
                Terrain terrain = new Terrain();
                Raquette raquette = new Raquette(terrain);
                Ballon ballon = new Ballon(terrain);
                //raquette.setPriority(Thread.NORM_PRIORITY+1);
                raquette.start();
```

```
                ballon.start();
                }

}
```

Listing 70 – thread 2

```
public class Table {

        int table[];

        Table(int k){
                table = new int[k];
                for (int i = 0; i<k ; i++)
                        table[i] = (int)(Math.random()*k);
        }

        public int lenght() {
                return table.length;
        }

        public int get(int i) {
                return table[i];
        }

        public void set(int i, int j) {
                table[i]=j;
        }

}
```

Listing 71 – thread 2

```
public class Sorter extends Thread{

        Table tab;

        public Sorter(Table tab){this.tab = tab;}

        public synchronized void triBulles() {
                int lenght = tab.lenght();
                int aux = 0;
                boolean flag;

                synchronized (tab) {
                        do {
                        flag = false;
                        for (int i = 0; i < lenght - 1; i++) {
                                if (tab.get(i) > tab.get(i + 1)) {
                                                aux = tab.get(i);
```

```
                                                tab.set(i, tab.get(i + 1));
                                                tab.set(i + 1, aux);
                                                flag = true;
                                        }
                                }
                        } while (flag);
                        }
        }

        public void run() {triBulles();}

        public Table getTab() {
                return tab;
        }
}
```

Listing 72 – thread 2

```
public class Writer extends Thread{

                Sorter s;

                public Writer(Sorter s){this.s = s;}

                public void affichage() throws InterruptedException {
                        synchronized(s.getTab()){
                                for (int i = 0; i<s.getTab().lenght(); i++)
                                        System.out.print(s.getTab().get(i)+" ");
                                System.out.println();
                        }
                }

                public void run(){try {
                        affichage();
                } catch (InterruptedException e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }}
        }
```

Listing 73 – thread 2

```
public class Main {

        static Table tab;
```

```
        public static void main(String[] args) throws InterruptedException {
                tab = new Table(80);
                Sorter s = new Sorter(tab);
                Writer w = new Writer(s);
                s.start();
                w.start();
        }

}
```

14 Flux d'entrée-sortie

- Stream
  - Entrée : InputStream / Reader
  - Sortie : OutputStream / Writer
- `java.io.*`

InputStream
- ByteArrayInputStream : tableau d'octets
- FileInputStream : fichier
- ObjectInputStream : objet
- PipeInputStream : tube
- SequenceInputStream : concaténation de plusieurs flux
- AudioInputStream : audio

InputStream
- int available()
- void close()
- void mark(int)
- boolean markSupported()
- int read()
- void reset()
- long skip(long)

Listing 74 – Input Stream

```
import java.io.*;

public class Main {

        public static void main(String[] args) throws IOException {
                byte[] table = {3, 5, 7};
                InputStream is = new ByteArrayInputStream(table);
                while(is.available()>0)
                        System.out.println(is.read());

                InputStream fis = new FileInputStream("monfichier.txt");
                while(fis.available()>0)
                        System.out.println(fis.read());
                fis.close();

                Reader r = new StringReader ("abc\ndef");
                int c;
                while ((c = r.read())!= -1 )
                        System.out. println ((char)c);
                r = new StringReader ("abc\ndef");
                BufferedReader br = new BufferedReader(r);
```

```
                String line;
                while((line = br.readLine()) != null)
                        System.out.println(line);

        }

}
```