الجمهورية الجزائرية الديمقراطية الشعبية

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université de 8 Mai 1945 – Guelma -

Faculté des Mathématiques, d'Informatique et des Sciences de la matière

Département d'Informatique

Mémoire de Fin d'études Master

Filière : Informatique

Option : Master Académique (AC)

Thème :

Un système d'aide à la compréhension du code source Java

Encadré Par :
Dr. Mohammed CHAOUI

Présenté par :
Zehor NOUICHI
Abbes BOUCHEBOUT

Juin 2016

A notre encadreur.
A nos Parents.
A notre Famille.
A nos Amis.

Remerciements

Nous remercions :

D'abord le tout puissant Allah, qui nous a accordé sa grâce, donné la santé et la force de réaliser ce travail.

Notre encadreur *Dr .Chaoui Mohamed*, pour ses conseils, sa supervision et son aide qui ont été plus que précieux.

Tout le personnel de l'administration de l'université 8 Mai 1945 de Guelma, plus particulièrement au corps professoral du département de l'informatique.

Nos amis, qui nous ont aidés de prés et de loin et qui n'ont cessé de nous encourager.

Résumé

Java est un langage de programmation et une plate-forme informatique. Savoir programmer en orienté objet est une question tout le temps posée spécialement pour les étudiants et surtout ceux qui sont débutants. En effet, la Programmation Orientée Objet « POO » a le potentiel d'être beaucoup moins complexe que la programmation procédurale, car elle est plus proche de l'expérience humaine qui manipule des objets dans sa vie quotidienne. D'ailleurs, cette approche de programmation nécessite un temps indéterminé dans l'aspect formation.

La grande partie des programmeurs souffrent non seulement dans l'algorithmique, mais aussi dans le nouveau mécanisme de POO et surtout les bases de cette programmation tels que : objets, classes, méthodes, encapsulation, polymorphisme, ... etc. C'est dans ce contexte que nous avons réalisé notre projet de fin d'études.

Ce projet de fin d'études traite la compréhension du code source Java, à l'aide des outils graphiques (programmation visuelle). Les principaux objectifs de notre système sont :

- La compréhension du code source Java (garantir les bases pour comprendre facilement le code des programmes crées).
- Fournir des aides à la décision lors la création du code source (les différentes propositions misent à la disposition aux programmeurs).
- La création d'un nouveau paradigme de programmation (par le développement d'un système qui assiste les programmeurs pendant la création du code source en assurant leur syntaxe).

Mots clés : Java, plate-forme, Programmation orienté objet, programmation procédurale, algorithmique, objets, classes, méthodes, encapsulation, polymorphisme, code source, programmation visuelle, paradigme, syntaxes.

Abstract

Java is a programming language and computing platform. Knowing how to program in object-oriented is an all the time asked question especially for students and mostly for those who are beginners. Indeed, the Object Oriented Programming « OOP » has the potential to be much less complex than procedural programming and that is for being closer to the human experience that manipulates objects in daily life. Although, this programming approach requires indeterminate time in formation aspect.

Most programmers suffer not only in algorithmic, but also in the new OOP mechanism: objects, classes, methods, encapsulation, polymorphism, etc... It is in this context that we have achieved our graduation project.

This project deals of the understanding of Java code sources, using graphical tools (visual programming). The main goals of our system are:

- Understanding the Java source code (securing the ground to understand in few time the code of the created programs).
- Provide decision support during the creation of the source code (various proposals will be available to programmers).
- The creation of a new programming paradigm (by the development of a system that assists the programmer in the creation of source codes and ensures their syntaxes).

Keywords: Java, platform, object-oriented programming, procedural programming, algorithms, objects, classes, methods, encapsulation, polymorphism, source code, visual programming, paradigm, syntaxes.

الملخص

جافا هي لغة برمجة ومنصة الحوسبة. معرفة البرمجة باللغات الموجّهة للكيانات هو سؤال يطرح دائما على الطلاب وخاصة المبتدئين منهم. وبالفعل، فإن البرمجة الموجّهة للكيانات لديها القدرة على أن تكون أقل تعقيدا بكثير من البرمجة الإجرائية لكونها أقرب إلى التجارب الإنسانية التي تعالج المسائل اليومية. مع ذلك، فان هذا النهج من البرمجة يتطلب وقتا غير محدد من ناحية التدريس.

الكثير من المبرمجين يعانون ليس فقط من الخوارزميات، ولكن أيضا من الآلية الجديدة للبرمجة الموجهة للكيانات: كيانات، الفئات، الإجراءات، التغليف، تعدد الأشكال، ... الخ. وفي هذا السياق حضرنا مشروع التخرج خاصتنا.

هذا المشروع يعالج فهم الكود المصدري للجافا باستخدام الأدوات البيانية (البرمجة المرئية). الأهداف الرئيسية لنظامنا هي:

- فهم الكود المصدري للجافا (تأمين الأساسيات للفهم السهل للبرامج المكتوبة).
- المساعدة في اتخاذ القرارات أثناء كتابة الكود المصدري (المقترحات المختلفة المعروضة على المبرمجين).
- إنشاء نمط برمجة جديد (من خلال إنشاء نظام يساعد المبرمج في كتابة كود المصدر مع ضمان التركيب الجيد للكود).

كلمات مفتاحية: جافا، منصة، البرمجة الموجّهة للكيانات، البرمجة الإجرائية، الخوارزميات، كيانات، الفئات، الإجراءات، التغليف، تعدد الأشكال، كود المصدر، البرمجة المرئية، تركيب جمل الكود.

TABLE DES MATIERES

TABLE DES ILLUSTRATIONS

LISTE DES FIGURES

LISTE DES TABLEAUX

LISTE DES LISTINGS

LISTE DES ABREVIATIONS

POO :	Programmation Orienté Objet
OO :	Orienté Objet
JDK :	Java Development Toolkit
JRE :	Java Runtime Environment
JRI :	Java Remote Invocation
CORBA:	Common Object Request Broker Architecture
EJB :	Entreprise Java Bean
API :	Application Programming Interface
MCD :	Modèle Conceptuel de Données
MLD :	Modèle Logique de Données
SGBD :	Systeme de Gestion de Base de Données.
SQL :	Strutured Query Language
MPD :	Modèle Physique de Données
EDI :	Environnement de Développement Intégré
BDD :	Base De Données
MopT :	Modèle opérationnel de Traitement
A :	Automatique
M :	Manuelle
S-A :	Semi-Automatique

Introduction générale

Contexte et cadre de la recherche

Java est l'un des langages de programmation les plus auréolés de succès de ces quatre dernières décennies. Ce qui mène à une nécessité à l'apprendre. Mais ce langage est un langage orienté objet et cette approche de programmation nécessite un temps important pour pouvoir former des programmeurs en Java.

Objectif et approche

Élaborer un système d'aide à la compréhension du code source Java par l'illustration des bases de programmation (En offrant au débutant programmeur une nouvelle approche de programmation visuelle qui lui guide et explique les bases de programmation). Puis, le système garantie la bonne syntaxe en réduisant le plus maximum l'écriture classique dans les environnements de développement.

Plan du mémoire

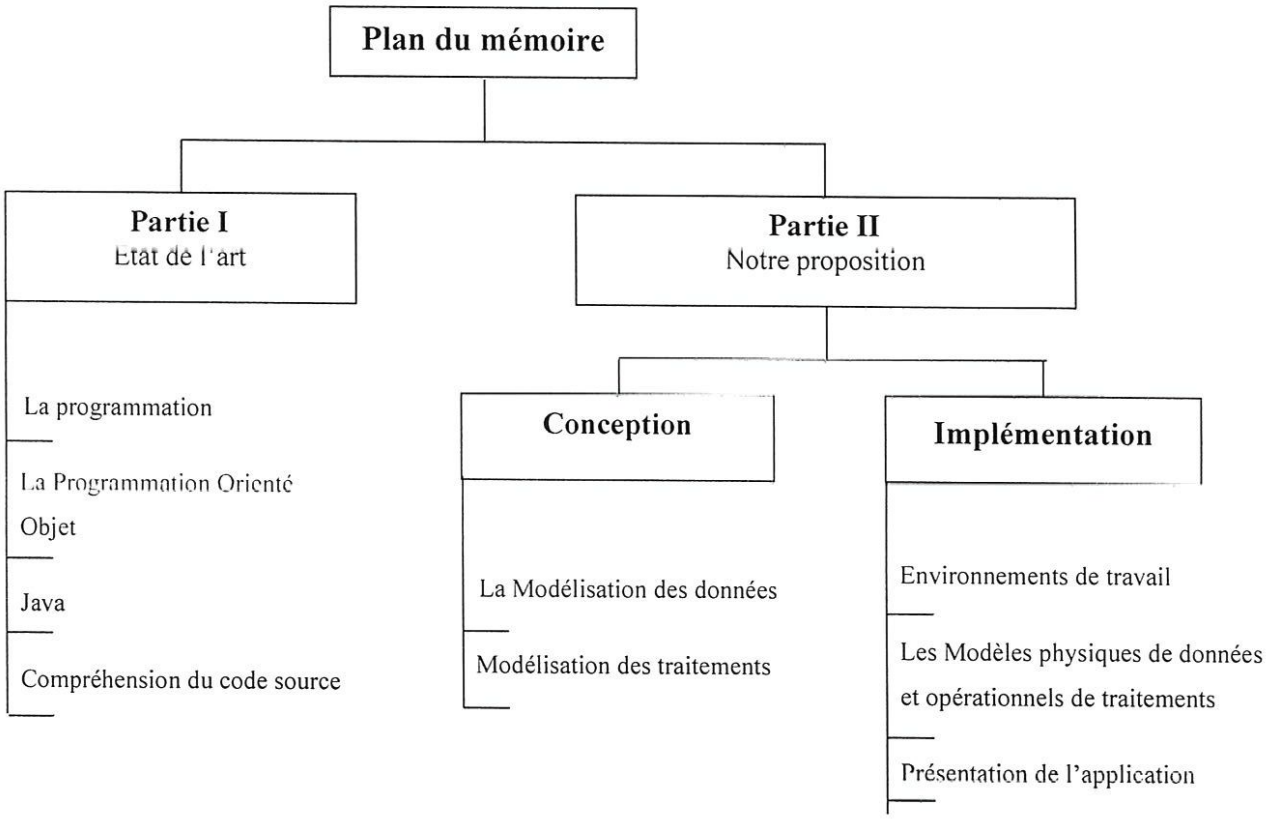


Figure 1. 1 : Plan du mémoire

Dans la partie I, nous faisons un rappel sur la programmation orienté objet et les bases de langage Java.

La partie II :

- conception : cette partie est dédiée à la conception du système.
- Implémentation : cette partie est consacrée à l'exposition du système réalisé.

PARTIE – I : ETAT DE L'ART

1. Introduction

D'après Douglas Rushkoff dans son dernier livre, la programmation est l'avenir car L'ordinateur est partout. Ce qui fait de gens qui ne savent pas programmer des analphabètes. [1]

Dans ce chapitre, nous allons introduire la programmation orienté objet, et quelques concept du langage JAVA, et les principes de la compréhension du code source.

2. La programmation

La programmation, c'est écrire des lignes de code, parfaitement lisibles humainement, mais dans un code très spécial (un langage de programmation). C'est une étape importante du développement de logiciels. Pour écrire le résultat de cette activité, on utilise un langage de programmation. L'un des langages les plus populaires auprès des développeurs est le **Java**.

3. La Programmation Orienté Objet

3.1. Vue d'ensemble de la Programmation Orientée Objet

3.1.1. L'objet

Pour bien comprendre la Programmation Orientée Objet il faut parler **d'*objet.***

Un objet est avant tout une **structure de données**. Autrement, il s'agit d'une entité chargée de gérer des données, de les classer, et de les stocker sous une certaine forme. En cela, rien ne distingue un ***objet*** d'une quelconque autre structure de données. La principale différence vient du fait que l'*objet* **regroupe les données et les moyens de traitement de ces données**.

Un *objet* rassemble de fait deux éléments de la programmation procédurale.

- Les **champs :**

  Les ***champs*** sont à l'objet ce que les variables sont à un programme : ce sont eux qui ont en charge les données à gérer. Tout comme n'importe quelle autre variable, un *champ* peut posséder un type quelconque défini au préalable : nombre, caractère... ou même un type objet.

- Les **méthodes :**

Les ***méthodes*** sont les éléments d'un objet qui servent d'interface entre les données et le programme. Sous ce nom obscur se cachent simplement des procédures ou fonctions destinées à traiter les données. [2]

3.1.2. Objet et Classe

Avec la notion d'**objet**, il convient d'amener la notion de **classe**. Une **classe d'objet**. Il s'agit donc du type à proprement parler. L'**objet** en lui-même est une **instance de classe**, plus simplement un exemplaire d'une classe, sa représentation en mémoire. Par conséquent, on déclare comme type une *classe*, et on déclare des variables de ce type appelées des *objets*.

3.1.3. Les trois fondamentaux de la POO

La POO est dirigée par trois fondamentaux qu'il convient de toujours garder à l'esprit **encapsulation**, **héritage** et **polymorphisme**.

3.1.3.1. Encapsulation

L'encapsulation est de réunir sous la même entité les données et les moyens de les gérer, à savoir les champs et les méthodes.

L'encapsulation introduit une nouvelle manière de gérer des données. Il ne s'agit plus de déclarer des données générales puis un ensemble de procédures et fonctions destinées à les gérer de manière séparée, mais bien de réunir le tout sous le couvert d'une seule et même entité.

L'encapsulation permet de garder une cohérence dans la gestion de l'objet, tout en assurant l'intégrité des données qui ne pourront être accédées qu'au travers des méthodes visibles.

3.1.3.2. Héritage

Si l'encapsulation pouvait se faire manuellement (grâce à la définition d'une unité par exemple), il en va tout autrement de l'**héritage**. Cette notion est celle qui s'explique le mieux au travers d'un exemple. Considérons un objet ***Bâtiment***. Cet objet est pour le moins générique, et sa définition reste assez vague. On peut toutefois lui associer divers champs, dont par exemple :

- les murs ;
- le toit ;
- une porte ;
- l'adresse ;
- la superficie.

On peut supposer que cet objet ***Bâtiment*** dispose d'un ensemble de méthodes destinées à sa gestion. On pourrait ainsi définir entre autres des méthodes pour :

- ouvrir le Bâtiment ;
- fermer le Bâtiment ;
- agrandir le Bâtiment.

Grâce au concept d'**héritage**, cet objet ***Bâtiment*** va pouvoir donner naissance à un ou des *descendants*. Ces descendants vont tous bénéficier des caractéristiques propres de leur *ancêtre*, à savoir ses champs et méthodes. Cependant, les descendants conservent la possibilité de posséder leurs propres champs et méthodes. Tout comme un enfant hérite des caractéristiques de ses parents et développe les siennes, un objet peut hériter des caractéristiques de son ancêtre, mais aussi en **développer de nouvelles**, ou bien encore se **spécialiser**.

Ainsi, si l'on poursuit notre exemple, nous allons pouvoir créer un objet ***Maison***. Ce nouvel objet est toujours considéré comme un ***Bâtiment***, il possède donc toujours des murs, un toit, une porte, les champs *Adresse* ou *Superficie* et les méthodes destinées par exemple à *Ouvrir le Bâtiment*.
Toutefois, si notre nouvel objet est toujours un ***Bâtiment***, il n'en reste pas moins qu'il s'agit d'une ***Maison***. On peut donc lui adjoindre d'autres champs et méthodes, et par exemple :

- nombre de fenêtres ;
- nombre d'étages ;
- nombre de pièces ;
- possède ou non un jardin ;
- possède une cave.

Ce processus d'héritage peut bien sûr être répété. Autrement dit, il est tout à fait possible de déclarer à présent un descendant de *Maison*, développant sa spécialisation : un *Chalet* ou

encore une *Villa*. Mais de la même manière, il n'y a pas de restrictions théoriques concernant le nombre de descendants pour un objet. Ainsi, pourquoi ne pas déclarer des objets *Immeuble* ou encore *Usine* dont l'ancêtre commun serait toujours ***Bâtiment***. Ce concept d'**héritage** ouvre donc la porte à un nouveau genre de programmation. [2]

3.1.3.3. Polymorphisme

Le **polymorphisme** (*poly* comme plusieurs et *morphisme* comme *forme*) traite de la capacité de l'objet à posséder plusieurs formes.

Cette capacité dérive directement du principe d'héritage vu précédemment. En effet, comme on le sait déjà, un objet va hériter des champs et méthodes de ses ancêtres. Mais un objet garde toujours la capacité de pouvoir **redéfinir une méthode** afin de la réécrire, ou de la **compléter**.

On voit donc apparaître ici ce concept de **polymorphisme** : choisir en fonction des besoins quelle méthode ancêtre appeler, et ce au cours même de l'exécution. Le comportement de l'objet devient donc modifiable à volonté.

Le **polymorphisme**, en d'autres termes, est donc la capacité du système à choisir dynamiquement la méthode qui correspond au type réel de l'objet en cours. Ainsi, si l'on considère un objet *Véhicule* et ses descendants *Bateau*, *Avion*, *Voiture* possédant tous une méthode *Avancer*, le système appellera la fonction *Avancer* spécifique suivant que le véhicule est un *Bateau*, un *Avion* ou bien une *Voiture*.

3.2. Différents types de méthodes

Parmi les différentes méthodes d'un objet se distinguent deux types de méthodes bien particulières et remplissant un rôle précis dans sa gestion : les **constructeurs** et les **destructeurs**.

3.2.1. Constructeurs et destructeurs

3.2.1.1. Constructeurs

Comme leur nom l'indique, les **constructeurs** servent à **construire l'objet en mémoire**. Un **constructeur** va donc se charger de mettre en place les données, d'associer les méthodes avec les champs et de créer le *diagramme d'héritage* de l'objet, autrement dit de mettre en place toutes les liaisons entre les ancêtres et les descendants.

Il faut savoir que s'il peut exister en mémoire plusieurs instances d'un même type objet, autrement dit plusieurs variables du même type, seule **une copie des méthodes** est conservée en mémoire, de sorte que chaque instance se réfère à la même zone mémoire en ce qui concerne les méthodes. Bien entendu, les champs sont distincts d'un objet à un autre. De fait, seules les données diffèrent d'une instance à une autre, la "machinerie" reste la même, ce qui permet de ne pas occuper inutilement la mémoire.

3.2.1.2. Destructeurs

Le **destructeur** est le contraire du constructeur : il se charge de **détruire l'instance de l'objet**. La mémoire allouée pour le *diagramme d'héritage* est libérée. Certains compilateurs peuvent également se servir des destructeurs pour éliminer de la mémoire le code correspondant aux méthodes d'un type d'objet si plus aucune instance de cet objet ne réside en mémoire.

3.2.2. Pointeur interne

Très souvent, les objets sont utilisés de manière **dynamique**, et ne sont donc créés que lors de l'exécution. Si les méthodes sont toujours communes aux instances d'un même type objet, il n'en est pas de même pour les données.

Il peut donc se révéler indispensable pour un objet de pouvoir se référencer lui-même. Pour cela, toute instance dispose d'un **pointeur interne** vers elle-même.

Ce pointeur peut prendre différentes appellations. En **Pascal**, il s'agira du pointeur **Self**. D'autres langages pourront le nommer ***this***, comme le Java.

3.2.2.1. Méthodes abstraites

Une méthode **abstraite** est une méthode qu'il est **nécessaire de surcharger**. Elle ne possède donc **pas d'implémentation**. Ainsi, si on tente d'appeler une méthode **abstraite**, alors une erreur est déclenchée en demandant de l'implémenter.

3.3. Visibilité

De par le principe de l'encapsulation, afin de pouvoir garantir la protection des données, il convient de pouvoir masquer certaines données et méthodes internes les gérant, et de pouvoir laisser visibles certaines autres devant servir à la gestion publique de l'objet. C'est le principe de la visibilité.

3.3.1. Champs et méthodes publics

Comme leur nom l'indique, les champs et méthodes dits **publics** sont accessibles depuis tous les descendants et dans tous les modules : programme, unité...

On peut considérer que les éléments **publics** n'ont pas de restriction particulière.

Les méthodes publiques sont communément appelées **accesseurs** : elles permettent d'accéder aux champs d'ordre *privé*. Il existe des accesseurs en *lecture*, destinés à récupérer la valeur d'un champ, et des accesseurs en *écriture* destinés pour leur part à la modification d'un champ.

3.3.2. Champs et méthodes privés

La visibilité **privée** restreint la portée d'un champ ou d'une méthode au **module où il ou elle est déclaré(e)**. Ainsi, si un objet est déclaré dans une unité avec un champ privé, alors ce champ ne pourra être accédé qu'à l'intérieur même de l'unité.

Généralement, les **accesseurs**, autrement dit les méthodes destinées à modifier les champs, sont déclarés comme **privés**.

3.3.3. Champs et méthodes protégés

La visibilité **protégée** correspond à la visibilité **privée** excepté que tout champ ou méthode protégé (e) est accessible dans tous les descendants, quel que soit le module où ils se situent. [1]

4. Java

Java est un langage Orienté Objet fortement typé avec des classes. C'est un environnement d'exécution (JRE) : Une machine virtuelle et un ensemble de bibliothèques.

C'est un environnement de développement (JDK) : Un compilateur et un ensemble d'outils. [3]

Java est devenu aujourd'hui l'un des langages de programmation les plus utilisés car il est incontournable dans plusieurs domaines :

- Systèmes dynamiques : Chargement dynamique de classes.
- Internet : Les Applets java.
- Systèmes communicants : RMI, Corba, EJB, etc.

4.1. Les types élémentaires

Type élémentaire	Intervalle de Variation	Nombre de Bits
Boolean	False, true	1bit
byte	[-128, +127]	8 bits
char	caractères unicode (valeurs de 0 à 65536)	16 bits
double	Virgule flottante double précision 5.10^{308}	64bits
float	Virgule flottante simple précision $\sim 9.10^{18}$	32 bits
int	entier signé : $[-2^{31}, +2^{31} - 1]$	32 bits
long	entier signé long : $[-2^{63}, +2^{63} - 1]$	64 bits
short	entier signé court : $[-2^{15}, +2^{15}-1]$	16 bits

***Tableau 1. 1** : Tableau synthétique des types élémentaires de Java*

4.2. Les tableaux, vecteurs et dictionnaires

Les listes comme les tableaux, vecteurs ou autres objets plus sophistiqués sont très couramment utilisées et doivent être parfaitement maîtrisées pour tirer partie du langage.

4.2.1. Les Tableaux

Un tableau est un objet ayant un nombre d'éléments fixe et pouvant contenir des objets ou des types primitifs. A la différence de certains langages, un tableau est lui-même un objet.
Un tableau peut également contenir des objets de mêmes types : *{new Integer(1), new Vehicule(), new Voiture()}* est un tableau d'***Object*** et { new Integer(1),new Integer(2),new Integer(3)} est un tableau d'***Integer***.

4.2.1.1. Syntaxe de déclaration

Tableau de 5 entiers primitifs :

```
int[] monTableau=new int[5];
Ou
int monTableau[]=new int[5];
```

La première notation est de loin la meilleure car de cette façon, nous visualisons bien que ***monTableau*** est de type tableau d'entiers primitifs. Dans la seconde notation, les crochets sont mis à la fin de la référence pour assurer une continuité avec les normes du C mais il faut absolument éviter de l'utiliser.

4.2.1.2. Un tableau est un objet

Un tableau Java peut contenir des primitifs (tel un *int*) ou des objets (tel l'objet *Voiture*).
Attention! Il faut bien comprendre qu'un tableau en Java est un objet à part entière. Ainsi, un tableau possède la propriété *length*:

*monTableau.**length*** vaut **5**.

Autre conséquence:
Object[] monTableau; ne réserve pas d'espace mémoire mais spécifie simplement que la variable *monTableau* est un tableau. Faire *monTableau[0]=1;* lèvera une Exception

4.2.1.3. Dimensions d'un tableau :

La notation d'accès au contenu d'un tableau est celle du C:
monTableau[2]=3;
Les tableaux peuvent être de n'importe quelle dimension, par exemple, il est possible de créer le tableau suivant:

```
int[][][] monTableau=new int[3][4][5];
```

Il est possible d'initialiser un tableau déjà rempli par la syntaxe:

```
int[] iMonTableau={1,2,3};
int[][] iMonTableau2={{1,2},{2,3},{4,5}};
Object[] oMonTableau={new Object(),new
Object()};
```

Dans cette syntaxe, nous ne précisons pas le nombre d'éléments et nous n'instancions pas le tableau car le compilateur va le faire à notre place. C'est simplement un confort du langage qui évite d'avoir à faire:

```
int iMonTableau[]=new int[5];
iMonTableau[0]=1;
iMonTableau[1]=2;
iMonTableau[2]=3;
```

Mais il y a bien instanciation du tableau : un tableau est un objet.

4.2.1.4. Copie d'un tableau à un autre

Une façon rapide de copier le contenu d'un tableau à un autre est d'utiliser la méthode statique ***arraycopy*** de la classe ***System***:

System.arraycopy*([Tableau d'origine], [index de la première donnée à copier], [tableau de destination], [index de la première donnée copiée], [nombre d'éléments à copier]);*

4.2.2. Les vecteurs

Lorsque l'on crée un tableau, il faut spécifier sa taille qui est fixe. Java fourni un objet très utilisé: le **vecteur** (classe *java.util.Vector*). L'utilisation d'un vecteur plutôt qu'un tableau est souvent avantageuse lorsque la taille finale du tableau n'est pas connue à l'avance. Un vecteur est un tableau dynamique. La première cellule d'un vecteur est à l'index zéro.

4.2.2.1. Principales méthodes de la classe Vector

Lorsque l'on crée un tableau, il faut spécifier sa taille qui est fixe. Java fourni un objet très utilisé : le vecteur (classe *java.util.Vector*). L'utilisation d'un vecteur plutôt qu'un tableau est souvent avantageuse lorsque la taille finale du tableau n'est pas connue à l'avance. Un vecteur est un tableau dynamique. La première cellule d'un vecteur est à l'index zéro.

4.2.2.2. Principales méthodes de la classe Vector

Action	**Syntaxe**
Création d'un vecteur *vMonVecteur*	*Vector vMonVecteur = new Vector();*
Insertion d'un objet *o*	*addElement(Object o);*
Suppression d'un élément à la position *i*	*removeElementAt(int i);*
Obtention du i ème élément	*Object elementAt(int i);*
Obtention du no de la ligne contenant l'objet *o*	*int indexOf(Object o);*
Nombre d'éléments dans le vecteur	*int size();*
Parcours d'un vecteur: Nous utilisons un objet Enumeration. Cf. 3.4- Les Enumérations	*Enumeration eEnum=vMonVecteur.elements();while(eEnum.hasMoreElements(){* *System.out.println(eEnum.nextElement());* *}*
Remplacement de l'objet à l'index *i* par l'objet *o*	*setElementAt(Object o, int i)*

Tableau 1. 2 : *Tableau des principales méthodes de la class Vector*

Remarque importante :

La méthode getElementAt(int) renvoie un objet qu'il faut caster.

Exemple:

```
Integer i1=(Integer)vMonVecteur.elementAt(0);
```

4.2.3. Les énumérations

Les énumérations sont des interfaces du package *java.util* qui permettent de parcourir rapidement les listes comme les vecteurs ou les dictionnaires. Elles sont plus rapides à l'exécution et plus pratique que les méthodes ***elementAt****(int)* de *Vector* par exemple.

Pour obtenir l'énumération sur un vecteur ou un dictionnaire (entre autres), il suffit d'appeler la méthode ***elements()*** de cette liste.

La classe *Enumeration* ne possède que deux méthodes:

- *Object* ***nextElement()*** qui renvoie l'objet suivant dans l'énumération. (Attention aux problèmes de cast !).
- *boolean* ***hasMoreElements()*** qui renvoie ***false*** si la fin de l'énumération est atteinte.

Après le premier appel à la méthode *nextElement*, l'énumération pointe sur le premier élément de la liste.

Pour repositionner l'énumération au premier élément, il faut refaire un appel à la méthode *elements()* de la liste qui renvoie une nouvelle énumération initialisée.

Attention aux cas imprévus ou indéterminés: si vous ajoutez ou effacez des éléments dans les listes, il faut récupérer une nouvelle énumération de la liste mise à jour.

4.3. Les chaînes de caractères

Nous aborderons dans cette partie les objets String, StringBuffer et StringTokenizer.

4.3.1. Les objets String

4.3.1.1. Instanciation d'un objet String

Les chaînes de caractères sont des instances de la classe *java.lang.String*. Elles peuvent être construites :

1- A partir d'un tableau de caractères:

```
char[] cListeDeChar={'h','e','l','l','o'};
String sMaChaine=new String(cListeDeChar);
```

2- Directement Par une chaîne:

```
String sMaChaine="hello";
```

La ligne de la méthode 2 est équivalente aux deux lignes de la méthode 1.

Il faut saisir correctement la signification de la ligne *String sMaChaine="hello";*. Le compilateur crée lui-même un objet *String* valant "*hello*" et en renvoie une référence. C'est la raison pour laquelle nous n'avons pas eu à instancier explicitement *sMaChaine*. Ce mode d'instanciation est particulier aux chaînes de caractères et a été mis en place pour d'évidentes raisons de simplicité.

4.3.1.2. Les String sont constants

Il n'existe pas de méthode permettant de modifier directement un String. Par exemple, un hypothétique "sMaChaine[3]='o' " n'existe pas. En revanche, il est possible de stocker le résultat d'une modification dans un autre String puis de positionner la référence du premier String sur le nouvel objet :

```
sMaChaine=sMaChaine.trim(); //supprime les blancs en début et fin de String
```

```
sMaChaine.trim() //renvoie un objet de type String et nous déplaçons la référence sur ce nouvel objet.
```

4.3.1.3. Longueur d'une chaîne

Pour obtenir la longueur d'une chaîne de caractère, il est possible d'utiliser la méthode ***length()*** qui renvoie un entier représentant le nombre de caractères de la chaîne. Attention! à ne pas confondre la méthode *String.length()* avec la propriété *length* d'un tableau.

4.3.1.4. Principales méthodes de la classe String

Méthode	Description
char charAt(int index)	Renvoie le caractère à la position *index*
boolean equals(String sOtherString)	Renvoie vrai si le String sur lequel est appliquée la méthode *equals* est égal à *sOtherString*
int indexOf(String sMaChaine)	Renvoie l'index de la première occurrence de la chaîne*sMaChaine*.
String substring(int iDepart,int iFin)	Renvoie un String contenant la chaîne de départ de l'index *iDebut* à l'index *iFin-1*. Plus simplement, *iFin-iDebut* vaut la longueur de la sous-chaine.
char[] toCharArray()	Renvoie un tableau de caractères
String toUpperCase()	Renvoie la version tout en majuscules de cette chaîne
String toLowerCase()	Renvoie la version tout en minuscules de cette chaîne
String trim()	Renvoie un String valant le String de départ sans les blancs et caractères spéciaux en début et fin de chaîne.
static String valueOf(int / boolean / double / float / long)	Renvoie un String correspondant à la valeur du primitif donné en argument. Par exemple, *String.valueOf(2)* renverra "*2*" Remarque : l'opération inverse peut être réalisée par: *int i=Integer.parseInt("2");*

Tableau 1. 3 : *Tableau des principales méthodes de la class String.*

4.3.1.5. Égalité de deux chaînes :

Pour comparer deux chaînes, c'est faux de faire:

```
String sMaChaine1="hello";
String sMaChaine2="hello";
if (sMaChaine1 == sMaChaine2){
```

Car ce test renverra toujours *false*. En effet, dans ce cas, ce sont les deux références qui sont comparées, c'est-à-dire les valeurs des zones mémoire de ces deux objets. Plutôt faire:

```
String sMaChaine1="hello";
String sMaChaine2="hello";
if (sMaChaine1.equals(sMaChaine2)){
```

Car nous utilisons la méthode ***equals*** de String qui va faire une comparaison caractères par caractères des deux chaînes.

4.3.1.6. Concatenation

L'opérateur '+' permet de concaténer deux chaînes ou une chaîne et un primitif:

```
String s1="Hello";
String s2="World";
String s=s1+" "+s2; // s vaut "Hello World"
```

```
String s1="x=";
int i=2;
String s=s1+i; // s vaut "x=2"
```

Ainsi, l'opérateur '+' connaît un polymorphisme paramétrique et ne se comporte pas de la même façon en fonction du type de ses deux arguments. Ainsi, Le '+' des exemples précédents est le '+' de la concaténation alors que le '+' prenant en argument deux primitifs (dans 1+2 par exemple) est le '+' arithmétique.

4.3.1.7. Méthode toString()

Toutes les classes depuis *Object* possèdent une méthode *toString()* qui renvoie un String.

Cette méthode sert à renvoyer une chaîne de caractère donnant des informations sur l'objet courant sous la forme d'une chaîne de caractères. Par exemple, la méthode *toString()* de la classe *Object* renvoie simplement le hashcode de l'objet. Il est ensuite possible de surcharger cette méthode pour donner davantage d'informations.

La méthode *println()* de *OutputStream* qui est très utilisée exécute toujours la méthode *toString()* et affiche la chaîne renvoyée. Donc faire:

```
Integer i=new Integer(2);
System.out.println(i);
```

Est équivalent à :

```
System.out.println(i.toString());
```

4.3.2. Les objets StringBuffer

Le StringBuffer est une chaîne de caractère présentant certains avantages de manipulations et de performances. Un StringBuffer est une chaîne de caractère modifiable. La classe StringBuffer possède en effet des méthodes permettant d'effectuer des modifications sur l'objet lui-même. Par exemple, ce qui était impossible pour le String (modifier un caractère de la chaîne) est possible pour le StringBuffer grâce à sa méthode ***setCharAt***.

4.3.2.1. Principales méthodes de la classe StringBuffer

Méthode	**Description**
char charAt(int index)	Renvoie le caractère à cette position
void delete(int iDebut,int iFin)	Supprime les caractères de la chaîne de *iDebut* à *iFin-1*
StringBuffer deleteCharAt(int iIndex)	Supprime le caractère à cette position
void setCharAt(int iIndex,char cChar)	Fixe le caractère de la chaine à l'index *iIndex* au char donné en argument.
boolean equals(String sOtherString)	Renvoie vrai si le String sur lequel est appliqué la méthode *equals* est égal à *sOtherString*
StringBuffer insert(int iIndex, String str)	Insére la chaîne *str* à l'index *iIndex*.
StringBuffer replace(int iDebut, int	Remplace la chaîne par la chaine *str* de l'index

iFin, String str)	*iDebut* à l'index *iFin-1.*
String substring(int iDepart,int iFin)	Ne conserve de la chaîne que les caractères de l'index *iDebut* à l'index *iFin-1.* Plus simplement, *iFin-iDebut* vaut la longueur de la sous-chaine.
String toString()	Renvoie le String correspondant à cette chaîne.

Tableau 1. 4 : *Tableau des principales méthodes de la class StringBuffer*

4.3.3. La classe StringTokenizer

Cette classe du package *java.util* sert à couper une chaîne en sous-chaînes de façon rapide. Le constructeur le plus courant est.

```
StringTokenizer(String sInput,String sSeparator){}
```

N'importe quelle chaîne peut servir de séparateur (un ou des espaces, des slashs, tirets...). La méthode *nextToken()* renvoie un *String* contenant la sous-chaîne suivante. La méthode *hasMoreTokens()* renvoie ***true*** s'il reste des chaînes. Exemple:

```
String sMonText="hello new world";
StringTokenizer st1=new StringTokenizer(sMonText," ");
String s1=st1.nextToken(); //s1 vaut "hello"
String s2=st1.nextToken(); //s2 vaut "new"
String s3=st1.nextToken(); //s3 vaut "world"
```

4.4. Les exceptions

L'un des éléments donnant sa puissance et sa robustesse au langage Java est sa gestion avancée des exceptions. Une application gérant correctement ses exceptions doit alors pouvoir réagir à toutes les situations y compris les cas imprévus par le développeur. Dans ce sens, une erreur d'exécution peut toujours être rattrapée et le programme pourra continuer sans sortie brusque ou « core dump ».

Dans la pratique, on utilise la gestion des exceptions dans deux cas :

- lorsque l'on s'attend à une erreur précise (par exemple, l'absence de fichier lors d'une lecture d'un disque).
- Pour protéger l'intégrité du programme: une erreur imprévue peut se produire et une partie du programme peut échouer. Dans ce cas, on peut décider de l'action à effectuer pour la suite du programme : sortie ou nouvel essai...

4.4.1. Lever une exception

Définitions :

- Une exception est un événement particulier apparaissant au cours de l'exécution (la plupart du temps une erreur).
- Une exception est un objet dérivant de la classe ***java.lang.Exception***.

Il existe toutes sortes d'exceptions dont certaines font partie de l'API Java standard comme les ***ArithmeticException***. D'autres comme une hypothétique « *VoitureSansPlaqueExceptio*n » sont developpees spécifiquement pour une application donnée

Exemple d'exception.

```
class VoitureSansPlaqueException extends Exception{
public String toString(){
return "Cette voiture ne possède pas de plaque d'immatriculation";  }}
```

Nous verrons plus loin à quoi sert la méthode *toString()*. Il est important de noter que dans la majorité des cas, vous n'aurez pas à implémenter de nouvelles exceptions mais simplement à utiliser celles déjà existantes. Les exceptions personnelles servent à créer des types d'erreur spécifiques qui permettront une granulosité plus fine à la gestion des événements. Dans le cas ou on ne cherche pas à déterminer la cause précise de l'erreur mais simplement à détecter un problème quelconque pour afficher un message générique, nous pouvons directement utiliser les objets de type ***Exception***.

Les exceptions sont dites levées ou envoyées par le mot-clé *throw.*

I. Le mot clé ***throw*** sert à initier le trajet de l'exception. Au rugby, ce serait le premier lancé du ballon.

4.4.2. Exemple

La méthode ***getPlaque*** prend en argument un objet *Voiture* et renvoie un objet ***PlaqueImmatriculation***. Si la voiture ne possède pas de plaque, le programme considère qu'une erreur applicative est commise et lève alors une ***VoitureSansPlaqueException***:

```
public PlaqueImmatriculation getPlaque(Voiture v) throws VoitureSansPlaqueException {
        if (v.plaque==null){      throw new VoitureSansPlaqueException();
                          }  else{      return v.plaque;
                                        }}
```

Le mot-clé *new* instancie l'exception et le mot-clé *throw* lance l'exception qui va alors être traitée ou propagée.

Remarque : Attention à ne pas confondre le mot clé **throw** qui sert à lever une exception avec le mot clé **throws** qui sert à propager l'exception.

4.4.3. Capturer ou propager

Lorsqu'une exception est levée, il y a deux possibilités pour le programme:

- Il propage l'exception (mot-clé ***throws***) ; au rugby, le joueur passe le ballon à un équipier.
- Il capture l'exception et la traite (mot-clés ***try/catch***) ; au rugby, le joueur marque l'essai.

Détaillons ces deux processus:

Propagation de l'exception :
Considérons l'exemple de la méthode *getPlaque* :

```
public PlaqueImmatriculation getPlaque(Voiture v) throws VoitureSansPlaqueException {
        if (v.plaque==null){      throw new VoitureSansPlaqueException();   }
                          else{      return v.plaque;   }}
```

Nous voyons que dans certains cas, cette méthode peut lever une exception. Cependant, elle ne la traite pas. Elle n'affiche pas de message d'erreur et n'effectue aucun traitement particulier. Elle se contente de propager l'exception à la méthode appelante par le mot-clé *throws*.

Dans la déclaration de la méthode, " *throws VoitureSansPlaqueException*" signifie que la méthode est susceptible de propager l'exception *VoitureSansPlaqueException*.

Dans ce cas, ce sera à la méthode appelante de traiter l'exception ou alors de la propager à son tour. La propagation d'une exception de méthodes en méthodes peut être affichées par la méthode *printStackTrace()* de l'exception.

Traitement de l'exception :
L'exception sera propagée de méthodes en méthodes tant qu'elle ne sera pas traitée. Pour traiter une exception, il suffit d'utiliser les mots-clés ***try*** et ***catch***:

```
try{   getPlaque(v1);   afficherPlaque();}
catch(VoitureSansPlaqueException vspe){   vspe.printStackTrace();
afficherMessageErreur();}
```

Le mot-clé ***try*** précise que les instructions contenues dans ces accolades peuvent lever des exceptions dont le type sera précisé en argument du ***catch*** associé.
Si la voiture *v1* ne possède pas de fournisseur, la méthode *getPlaque()* - comme on l'a vu- lèvera une exception de type *VoitureSansPlaqueException* et la propagera. A la ligne *getPlaque(v1)*, l'exception est alors capturée. La ligne *afficherPlaque()* n'est pas exécutée et le programme passe alors aux instructions du *catch* correspondant.

4.4.4. Compléments sur les exceptions

Maintenant, nous allons détailler certaines méthodes de programmation.

Les méthode *toString()* et *printStackTrace()* de la classe *Exception*

Cette méthode renvoie la chaîne de caractère décrivant l'erreur. Il est souvent judicieux de surcharger cette méthode à l'implémentation d'une classe fille d'Exception. La classe Exception quant à elle ne contient pas de message par défaut, il peut être utile de construire un objet Exception avec un String décrivant le problème.

Exemple :

```
public void test() throws Exception{
if ( - pbm de division par zéro -){      throw new Exception("Division par Zéro");
}
if (- pbm de nombre trop petit -){
throw new Exception("Nombre trop petit");
}}
```

L'instruction

```
try{   test();
}
catch(Exception e){
System.out.println(e);
}
```

Affichera "Division par zéro" ou "Nombre trop petit" selon le cas.

Mieux encore: L'instruction *e.printStackTrace()* ou '*e*' est l'exception affichera toujours les messages d'erreurs mais donnera le parcours complet de l'exception du moment où elle a été levée jusqu'à celui où elle a été capturée. Cette méthode de la classe Exception est très utile au debugage.

Capture sélective

Il est possible de réaliser des filtres d'exceptions dans des cas complexes. Par exemple, admettons que la méthode *getPlaque* vue précédemment puisse lever une *VoitureSansPlaqueException* mais également tout autre type d'exception. Dans ce cas, le code sera le suivant:

```
try{   getPlaque(v1);
}
catch( VoitureSansPlaqueException vspe){
vspe.printStackTrace(); }
catch( Exception e){
e.printStackTrace();
}
```

Nous effectuons ainsi une capture sélective avec granulosité décroissante. Bien entendu, aucune exception ne pourra passer outre un *catch(Exception)* puisque toutes les exceptions dérivent de la classe *Exception*.

Si nous voulons réaliser une action qu'il y ait une exception ou non, il faut utiliser le mot-clé ***finally***

```
try{    getPlaque(v1); }
catch( VoitureSansPlaqueException vspe){
vspe.printStackTrace(); }
catch( Exception e){    e.printStackTrace(); }
finally{
System.out.println("Traitement terminé"); } //ce code est exécuté qu'il y ait exception ou non
```

Remarque sur les déclarations de variables

Comme pour les boucles itératives, les variables locales définies dans un ***try*** ou un ***catch*** ne sont connues que dans ces zones.

4.5. L'héritage

Ici seront présentés les notions d'héritage, de classe abstraite, d'interface et l'ensemble des phénomènes touchant à l'héritage qu'il est nécessaire de maîtriser.

4.5.1. L'héritage simple

Définition
L'héritage permet de spécialiser une classe qui possédera non seulement les propriétés et méthodes de son unique mère mais d'autres méthodes spécifiques ou redéfinies. Dans l'objet fille, on trouve:

- Des méthodes ou propriétés qui *surchargent*, c'est-à-dire redéfinissent celles de la classe mère.
- Les propriétés et méthodes de la classe mère qui n'ont pas été surchargées.
- De nouvelles méthodes ou propriétés.

Pour spécifier qu'une classe dérive d'une autre classe, on utilise l'opérateur ***extends*** lors de la déclaration de la classe:

```
class Voiture extends Vehicule{
}
```

La classe mère commune à toutes les autres classes est la classe *Object*. Tous les objets dérivent implicitement d'*Object* et les déclarations suivantes sont identiques :

```
class Voiture{ }
class Voiture extends Object{
}
```

4.5.2. Notion de classe abstraite

Dans certains cas, nous pouvons désirer implémenter deux objets proches possédant une partie de méthodes et de propriétés en commun comme *Voiture* et *Moto*. Certaines de ces méthodes comme *rouler()*sont totalement identiques d'un objet à l'autre alors que certaines autres diffèrent comme la méthode *seGarer()* qui est différente pour une *Voiture* et pour une *Moto*. Pour ce faire, l'OO introduit la notion de classe abstraite.

Définition :

Une classe abstraite est une classe non-instanciable dont une partie des méthodes sont abstraites et restent à implémenter.

Remarque :

- Une classe abstraite ne peut être instanciée car elle n'est pas "complète".
- Elle peut avoir de zéro (comme une classe normale) jusqu'à l'intégralité (comme une interface) de méthodes abstraites.

Notation d'une classe abstraite

```
abstract class MaClasse{ }
```

Et pour une méthode

```
abstract void maMethode(); // jamais d'implémentation
```

Une méthode abstraite est une simple déclaration sans implémentation. Elle est simplement la marque que cette méthode devra être implémentée par la classe dérivée. Lorsqu'une classe dérive d'une classe abstraite, elle doit alors implémenter *toutes* les méthodes abstraites de celle-ci ou être elle-même abstraite. Cela permet lors d'un projet d'éviter d'"oublier" d'implémenter des méthodes communes. [4]

5. Comprehension du code source

Une façon d'améliorer les compétences de programmation est de Lire des Grands Programmes. Des techniques telles que l'Auto Documentation du Code ont été développés vers l'objectif d'écriture des programmes qui peut être lus facilement.

Cependant, la plupart des débutants lisent beaucoup de code qui ne répondent pas aux normes. Quelles sont les meilleures façons de faire sens à ces énormes, non structurés, maintenu par des dizaines-de-personnes, non-documentés codes sources que nous devons comprendre et absorber ?

5.1. Construire et exécuter le programme : (Build et Run)

- Être capable d'exécuter le programme et observer son comportement extérieur sera utile lors de l'étude de ses internes. La documentation du programme peut être utile aussi, mais la documentation ne peut pas décrire avec précision le comportement du programme.
- Être capable de construire le programme nous - même nous aidera à trouver les bibliothèques externes qui sont utilisées, et quel compilateur et options du Linker sont en vigueur, etc. Et si on peut construire et déboguer une version du programme, on peut parcourir le code avec le débogueur.

5.2. Trouver le haut niveau Logique

- En commençant par le point d'entrée du programme (par exemple, **main ()** en C / C ++ / Java), savoir comment le programme s'initialise, fait le traitement, puis se termine.
- La plupart des programmes ont une boucle principale qui faut la trouver. (Mais si le programme utilise un la bibliothèque d'une Frame-Work, la boucle principale peut être dans le Frame-Work et non dans le code de l'application).

- Trouver les conditions qui terminent le programme. Cela inclut des sorties anormales, en plus des conditions de sortie normales.

5.3. Examiner les appels des bibliothèques

Si le programme utilise des bibliothèques externes, examiner les appels de bibliothèque et lire la documentation de ces appels. (Cela peut être la seule "vraie documentation" on a, alors il faut la profiter)

Quelques techniques sont disponibles, avec un bon outil de recherche de texte, ou mieux encore un outil qui permet d'analyser le code source et trouver des références, permettent de répondre aux questions suivantes, pour le code orienté objet :

- Qui appelle cette méthode ?
- Qui implémente cette interface ou étend de cette classe ?
- Quelles sont les superclasses de cette classe ?
- Où sont les instances de cette classe créées, ou passés comme arguments ou retournés ?
- …etc.

5.4. Qu'est-ce qui influence la compréhension d'un code source ?

Les mécanismes cachés derrière la compréhension de texte ou de code source se basent beaucoup sur le fonctionnement de la mémoire. On peut en déduire rapidement que pour rendre un programme facile à lire, il faudra prendre en compte la mémoire du lecteur.

5.4.1. Segmentation du code : « chunking »

Pour éviter de surcharger la mémoire à court terme, il existe une solution : segmenter le code en petites unités. En faisant ainsi, nous allons répartir la charge cognitive dans les segments ; chaque segment sera traité en une seule fois, et aura donc une charge cognitive assez faible.

5.4.2. Tracing

Une métrique importante consiste à évaluer le nombre de liens entre différents morceaux de code.

Selon Cant et al (1995), la complexité d'un code source dépend non seulement de son *chunking*, mais aussi de son *tracing*. Le *tracing* correspond tout simplement à la linéarité du code : plus la lecture est linéaire, moins il y a de *tracing*.

Certains morceaux de code ont fatalement une lecture non-linéaire, qui impose de faire des sauts dans le code. L'exemple le plus typique est l'appel d'une fonction ou d'une méthode. Un autre exemple consiste en une variable déclarée assez loin dans le code. On peut aussi donner l'exemple d'un programmeur qui va lire la documentation pour comprendre quelle est le rôle des arguments d'une fonction.

Dans ces situations, le relecteur du code doit arrêter sa lecture du code, et chercher dans la page ou dans d'autres modules une occurrence de cette variable/déclaration de fonction, ou aller lire la documentation de la fonction. Ce genre de situation demande donc d'aller voir ailleurs si la variable/fonction/documentation y est. C'est ce qu'on appelle une situation de *tracing*.

On peut se demander pourquoi ce tracing est une catastrophe. Et il y a plusieurs raisons à cela, qui impliquent toute la mémoire à court terme.

- **Split Attention Effect : « effet de l'attention divisée »**

  Le *tracing* est signe qu'un morceau de code utile pour la compréhension locale d'un autre morceau de code est délocalisé dans une autre unité de code. Comprendre le code dans sa totalité demande donc de rassembler les informations de plusieurs morceaux de code séparés en un seul *chunk* dans la mémoire à court terme. Si les deux morceaux de code se suivent, pas de problèmes : l'assemblage se fait rapidement, sans trop d'efforts. Mais si les deux sont relativement séparés, la mémoire à court terme va devoir maintenir les informations du premier morceau de code lu, pour comprendre le suivant, ce qui impose une forte charge cognitive.

- **Task Switching : « Le changement de tâche »**

  Lorsqu'un programmer "trace" quelque chose, son superviseur attentionnel va devoir changer de tâche, et faire ce qu'il faut pour retrouver le code source à lire : l'attention est déportée sur la recherche de la fonction/variable/documentation de destination. Seule une très faible partie de l'attention servira à maintenir le contenu du code précédent en mémoire à court terme, qui sera oublié. Dans ces conditions, un gros *tracing* sera à éviter, alors qu'un petit *tracing* (quelques lignes en haut ou en bas) jouera peu.

- **Charge cognitive en trop :**

Ensuite, le programmeur devra fatalement revenir en arrière une fois la fonction lue. Cela demande de se souvenir de l'endroit où reprendre la lecture. À chaque fois que l'on trace quelque chose, la charge en mémoire à court terme augmente, diminuant les performances. Plus un code contient de composants imbriqués, pire est la situation. En général, un humain atteint une limite de performance au-delà de 3 à 4 niveaux d'imbrication, que ce soit pour le niveau d'imbrication d'un code source ou le nombre de niveaux d'héritage dans un code.

5.5. Solution

L'utilisation de certaines pratiques de conception OO. L'utilisation de ces principes force les développeurs à créer un grand nombre de petites classes, chacune très simple, mais chacune ayant une fonctionnalité très précise, avec une encapsulation très importante. Méthodes ayant chacune peu de lignes, évidemment.

L'usage de l'héritage est aussi fortement corrélé à une utilisation du *tracing*. Plus l'arbre d'héritage est profond, plus les effets se font sentir. Un bon arbre d'héritage est donc un arbre large, plus que profond. [5]

6. Conclusion

Dans ce chapitre, nous avons introduit quelques notions qui sont nécessaire a apprendre pour le mieux comprendre notre travail, telles que la POO, le Java, et la compréhension du code source.

Dans le prochain chapitre, nous allons décrire les différant modèles de la conception du notre projet.

PARTIE – II : NOTRE PROPOSITION

1. Conception

1.1. Introduction

Dans ce chapitre, nous allons aborder la conception de notre projet, en adoptant à méthodologie MERISE pour la conception des systèmes informatiques.

La conception sera articulée autour de deux aspects complémentaires : Conception des données et conception des traitements.

1.2. Modélisation des données

1.2.1. Modèle conceptuel de données

En suivant la démarche décrite par la méthodologie MERISE nous avons abouti au MCD suivant :

1.2.2. Dictionnaire de données

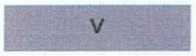
N°	Donnée	Codage	Type	Taille	Observation
1	Le nom de la class main	classWmain	A	45	Obligatoire
2	Code source de la boucle while	code	AN	1000	Obligatoire
3	Code source de la méthode	codeArea	AN	1000	Obligatoire
4	Code source de la boucle for	codeS	AN	1500	Obligatoire
5	Le code source de la class	codeSource	AN	2500	Obligatoire
6	Condition de répétition	condition	AN	100	Obligatoire
7	Début de la boucle for	debut	AN	45	Obligatoire
8	Code source du bloc if	do	AN	2000	Obligatoire
9	Code source du bloc else	else	AN	2000	Obligatoire
10	Référence sur le bloc else	elseLocation	N	11	
11	Les accolades fermantes	fermeture	A	10	Obligatoire
12	Fin de la boucle for	fin	AN	45	Obligatoire
13	Modifier final	final	A	45	Obligatoire
14	Référence sur la boucle for	forLocation	N	11	Obligatoire
15	référence de la boucle for	idBF2	N	11	Obligatoire
16	Identifiant des la boucle for	idBoucleF	N	10	Obligatoire
17	Identifiant de la boucle while	idBoucleW	N	11	Obligatoire
18	Référence de la boucle while	idBW2	N	11	Obligatoire
19	Reference de la class	idC	N	11	Obligatoire
20	Identifiant de la class	idclass	N	11	Obligatoire
21	Référence de la class	idClass	N	11	Obligatoire
22	Identifiant de la class main	idCMain	N	11	Obligatoire
23	Référence de class	idCMn	N	11	Obligatoire
24	Référence du bloc else	idElse	N	11	Obligatoire
25	Référence de la boucle for	idfor	N	11	Obligatoire
26	Reference du bloc if/ else	idIE	N	11	Obligatoire
27	Référence du bloc if	idIf	N	11	Obligatoire
28	Identifiant du bloc if/ else	idIfElse	N	11	Obligatoire
29	Référence sur le bloc if/ else	idIfElse2	N	11	Obligatoire
30	Identifiant de l'instance	idinstance	N	11	Obligatoire
31	Identifiant de la class instancelocale	idinstanceLocale	N	11	Obligatoire

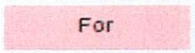
32	Référence sur la class	idLocation	N	11	Obligatoire
33	Référence de la méthode	idMD	N	11	Obligatoire
34	Identifiant d'une méthode	idmethode	N	11	Obligatoire
35	Identifiant du paramètre	idparametres	N	11	Obligatoire
36	Identifiant d'utilisateur	Idusers	N	10	Obligatoire
37	Identifiant de la variable globale	idvariableG	N	11	Obligatoire
38	Identifiant de la variable locale	idvariableLcale	N	11	Obligatoire
39	Référence de la boucle while	idWhil	N	11	
40	Référence de la boucle while	idwhile	N	11	
41	Référence sur le bloc if	ifLocation	N	11	
42	La date de la dernière modification	lastMod	DATETIME	/	Obligatoire
43	Référence sur la class main	mainLocation	N	11	
44	Le code source de la partie main	mainSource	AN	2500	Obligatoire
45	Référence sur la méthode	methodeLocation	N	11	
46	Modifier de la variable	modfier	A	45	
47	Nom de l'instance	name	AN	45	
48	Nombre de paramètres de la méthode	nbrPar	N	11	
49	Nom d'un utilisateur	nom	A	45	Obligatoire
50	Nom de la class	nomclass	A	45	Obligatoire
51	Nom de la méthode	nommeth	A	45	
52	Nom du paramètre	nomPr	A	45	
53	Nom du projet	nomprj	A	45	Obligatoire
54	Nom de la variable	nomvar	A	45	Obligatoire
55	Mot de passe du compte	password	AN	45	Obligatoire
56	Le retour de la méthode	returN	AN	45	
57	Modifier static	static	A	45	
58	Type du paramètre	typePr	A	45	
59	Type de retour de la méthode	typeretour	A	45	
60	Type de la variable	typevar	A	45	Obligatoire
61	Référence sur la boucle while	whileLocation	N	11	

Tableau 2. 1 : *Dictionnaire de données épuré*

1.2.3. Modèle logique de données

Nous avons fixé les éléments et les sous éléments du système dans le tableau ci-dessous :

Symbole Des éléments	Description	Données liées	Sous éléments possibles
	variable	• Nom • Type • Valeur • Modifier • Description • Visibilité	La valeur peut être : - saisie - une autre variable - un appel de méthode - une comparaison (booléen)
	méthode	• Nom de méthode • Paramètres • corps • Type de retour • Description	- variable (locale) - boucle (for , while) - if/else - affichage - un appel de méthode - Affectation - Instance de classes

	Boucle for	• Taille
		• corps
		• Description
			- variable (locale)
			- boucle (for , while)
			- if/else
			- affichage
			- Un appel de méthode
			- Affectation
			- Instance de classes
			- la taille peut être :
			  • saisie
			  • variable
			  • Un appel de méthode
			  • équation

	Boucle while	• Comparaison • corps • Description	- variable (locale) - boucle (for , while) - if/else - affichage - Un appel de méthode - Affectation - Instance de classes

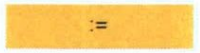
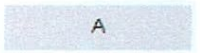
			- la comparaison peut être entre 2 valeurs : • saisies • variable • Un appel de méthode • Equation
	Condition if	• Comparaison • Corps • Description	- variable - boucle (for , while) - if/else - affichage - Un appel de méthode - Affectation - Instance de classes - la comparaison peut être entre 2 valeurs : • variable • saisie • Un appel de méthode • Equation
	instance d'une class	• Nom de la classe • Nom d'instance • Description	
	Affectation	• Opérande 1 • Opérande 2 • Description	L'opérande 2 peut être : - un appel d'une méthode - une autre variable - une équation
	Affichage sur console	• Valeur	La valeur à afficher peut être : - saisie - un appel d'une méthode - une autre variable - une équation.
	Appel d'une méthode	• Nom de la méthode • nom de l'instance • Paramètres • Description	

Tableau 2. 2 : *Les éléments et les sous éléments du système*

Après application des différentes règles de passage au MCD, nous avons obtenu le MLD suivant qui représente le schéma de la base de données.

1. bouclef (idBoucleF, debut, fin, codeS, #idM, #idBF2, #idBoucleW, #idIfElse, #idCMain).
2. bouclew (idBoucleW, condition, code, #idMM, #idBF, #idBW2, #idIE, #idC).
3. class (idclass, nomclass, codeSource, mainSource, fermeture, # idprj).
4. ifelse (idIfElse, condition, do, else, #ldm, #idfor, #idwhile, #idIfElse2, #idCMn).
5. instance (idinstance, name, #idClass, #idLocation).
6. instancelocale (idinstanceLocale, name, #idclass, #mainLocation, #forLocation, #whileLocation, #ifLocation, #elseLocation, #methodeLocation).
7. methode (idmethode, nommeth, nbrPar, typeretour, codeArea, returN, #idclass).
8. parameters (idparametres, nomPr, typePr, #idMD).
9. projet (idprj, nomprj, classWmain, lastMod, #idU).
10. users (idusers, nom, password).
11. variableglobale (idvariableG, nomvar, typevar, modifier, final, static, #idclass).
12. variablelcale (idvariableLcale, nomvar, typevar, final, #idmethode, #idCM, #idf, #idWhil, #idIf, #idElse).

1.2.4. Modèle physique de données

Ce modèle sera abordé dans la phase implémentation (prochain chapitre).
Le MLD conçu sera traduit directement en MPD en utilisant le SGBD MySQL.

1.3. Modélisation des traitements

1.3.1. Modèle conceptuel du traitement

Notre projet possède 4 grands processus à modéliser en MCT.

- **Le processus de la création d'un projet :**

Les programmes java s'organisent en projets. Un projet va contenir un certain nombre de fichiers java. C'est à nous de choisir la granularité de notre projet. Par exemple, on pourrait décider de faire un projet par cours, un projet par séance de TD ou un seul projet pour tout.

Ce processus représente l'enchainement des opérations de la création d'un nouveau projet.

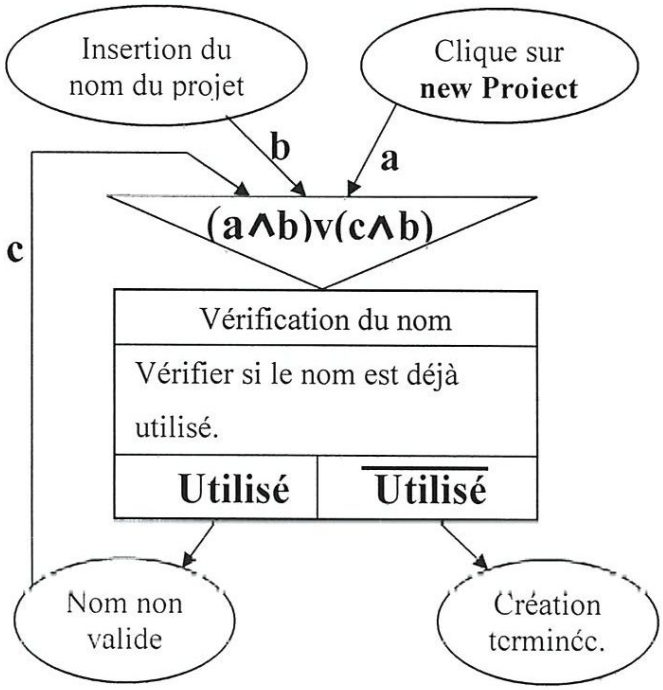


Figure 2. 3 : *MCT du processus de la création d'un projet.*

- **Le processus de la création d'une classe :**

Une classe peut être comparée à un moule qui, lorsque nous le remplissons, nous donne un objet ayant la forme du moule ainsi que toutes ses caractéristiques.

Ce processus désigne la représentation de l'enchainement des opérations de la création d'une nouvelle class.

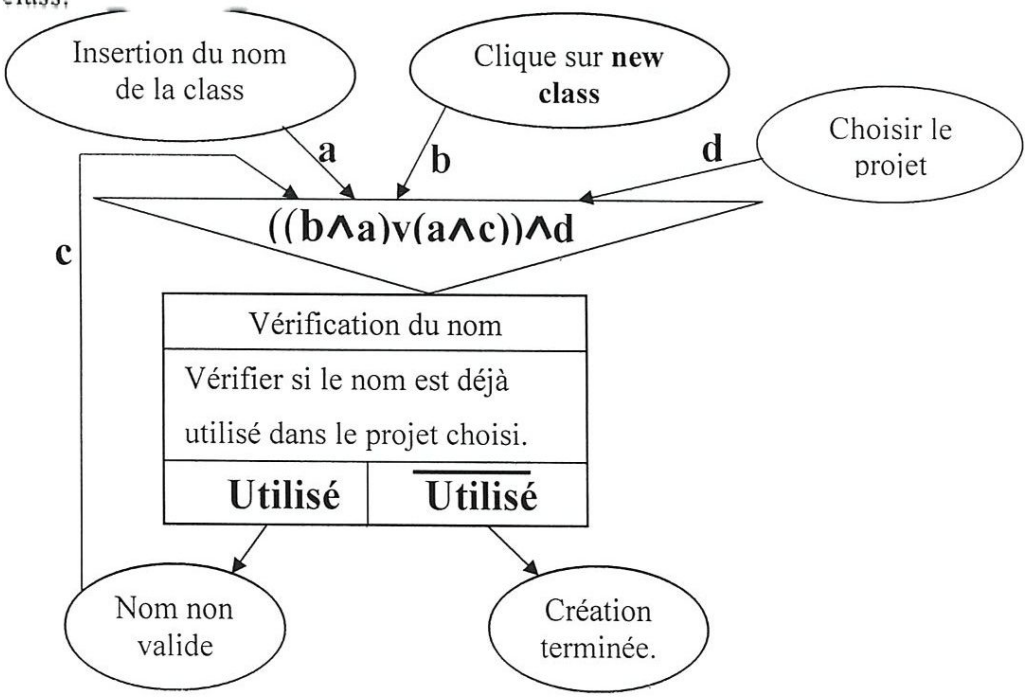


Figure 2. 4 : *MCT du processus de la création d'une class.*

- **Le processus de la création d'une variable :**

Ce modèle représente l'enchainement des opérations de la création d'une nouvelle variable (regroupe les trois procédures : procédure de la création d'une nouvelle variable ; procédure d'initialisation de la variable ; procédure de l'affectation).

Pour bien comprendre le processus nous allons illustrer la contrainte du porté des variables.

Lorsqu'une variable est déclarée directement dans la classe, c'est-à-dire à l'extérieur de toute méthode, elle sera accessible dans toute la classe. On parle alors de champ de la classe (Fields en anglais).

Lorsqu'on déclare une variable à l'intérieur d'un « bloc-instructions » (entre des accolades), sa portée se restreint à l'intérieur de ce bloc (dans les lignes qui suit sa déclaration), on parle alors de variable locale.

Il est accepté d'avoir deux variables de même nom si elles n'ont pas une portée commune.

Exemple de déclarations licites et de visibilité dans trois blocs d'instructions imbriqués :

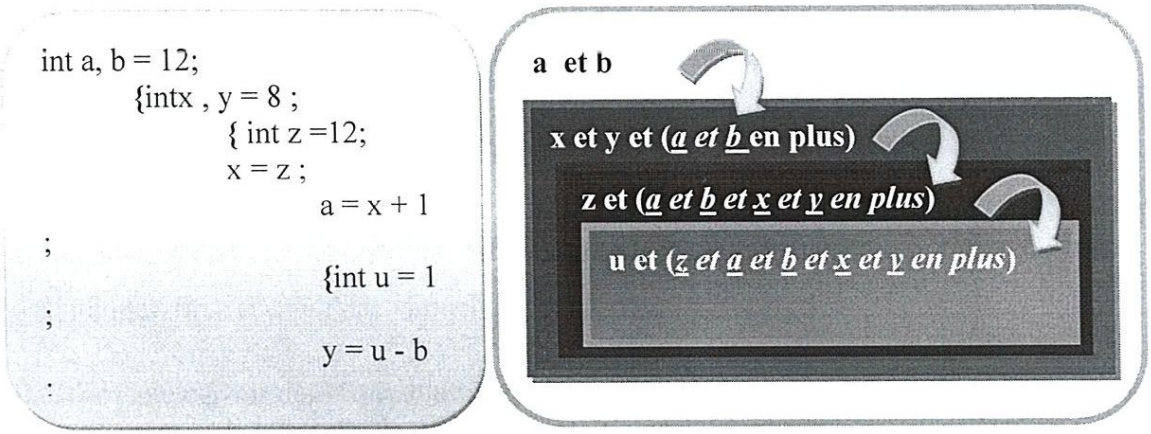


Figure 2. 5: *Exemple de la visibilité des variables.*

Nous voyons que x et a ont une portée commune, donc nommer x « **a** » va déclencher une erreur.

Et voici le MCT de ce processus :

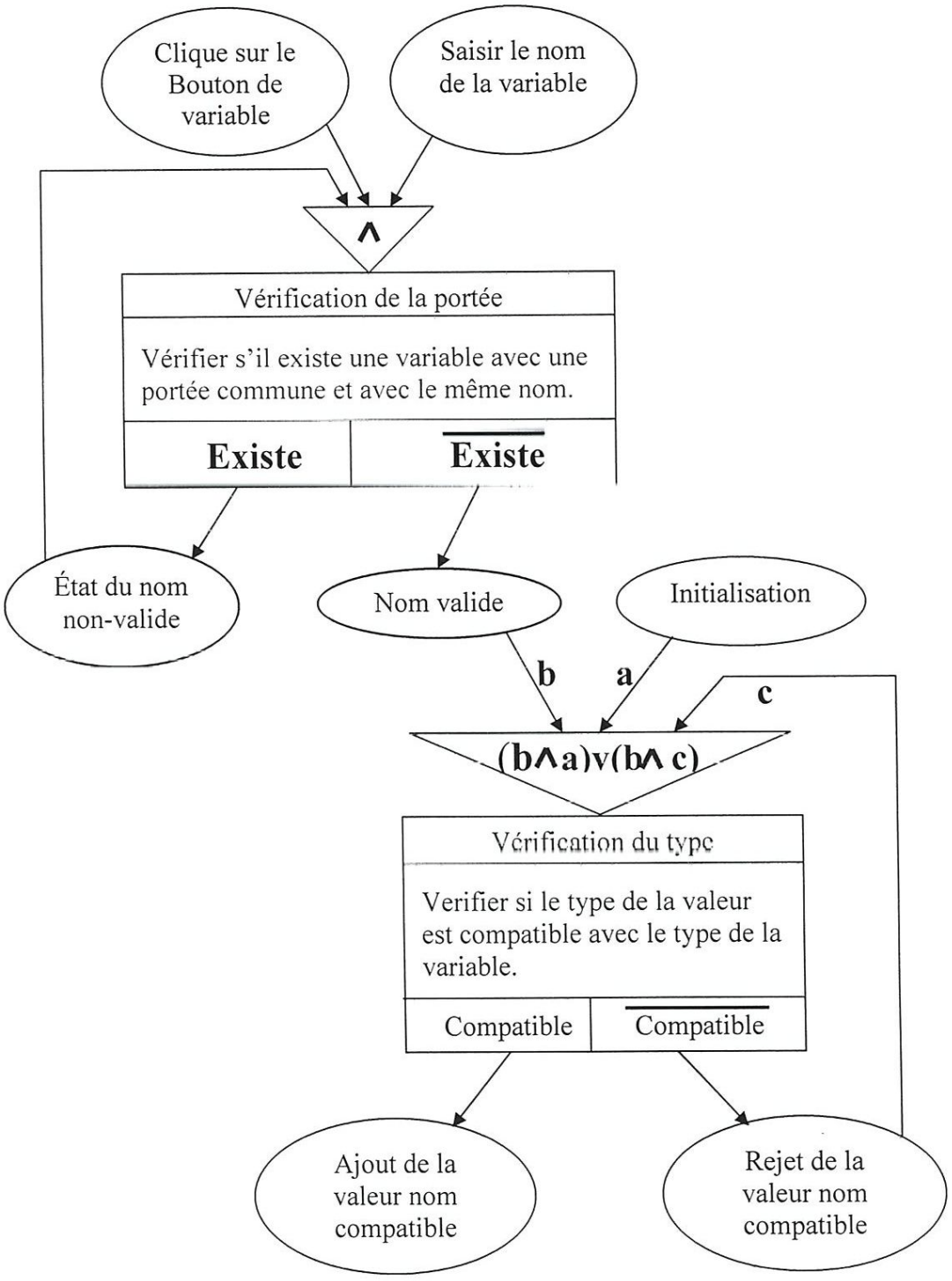


Figure 2. 6 : *MCT du processus de la création d'une variable.*

- **Processus de la création d'une affectation :**

L'affectation est une structure qui permet d'attribuer une valeur à une variable.

Ce processus regroupe trois processus (affectation, appelle de méthodes et la création des formules arithmétiques et logiques).

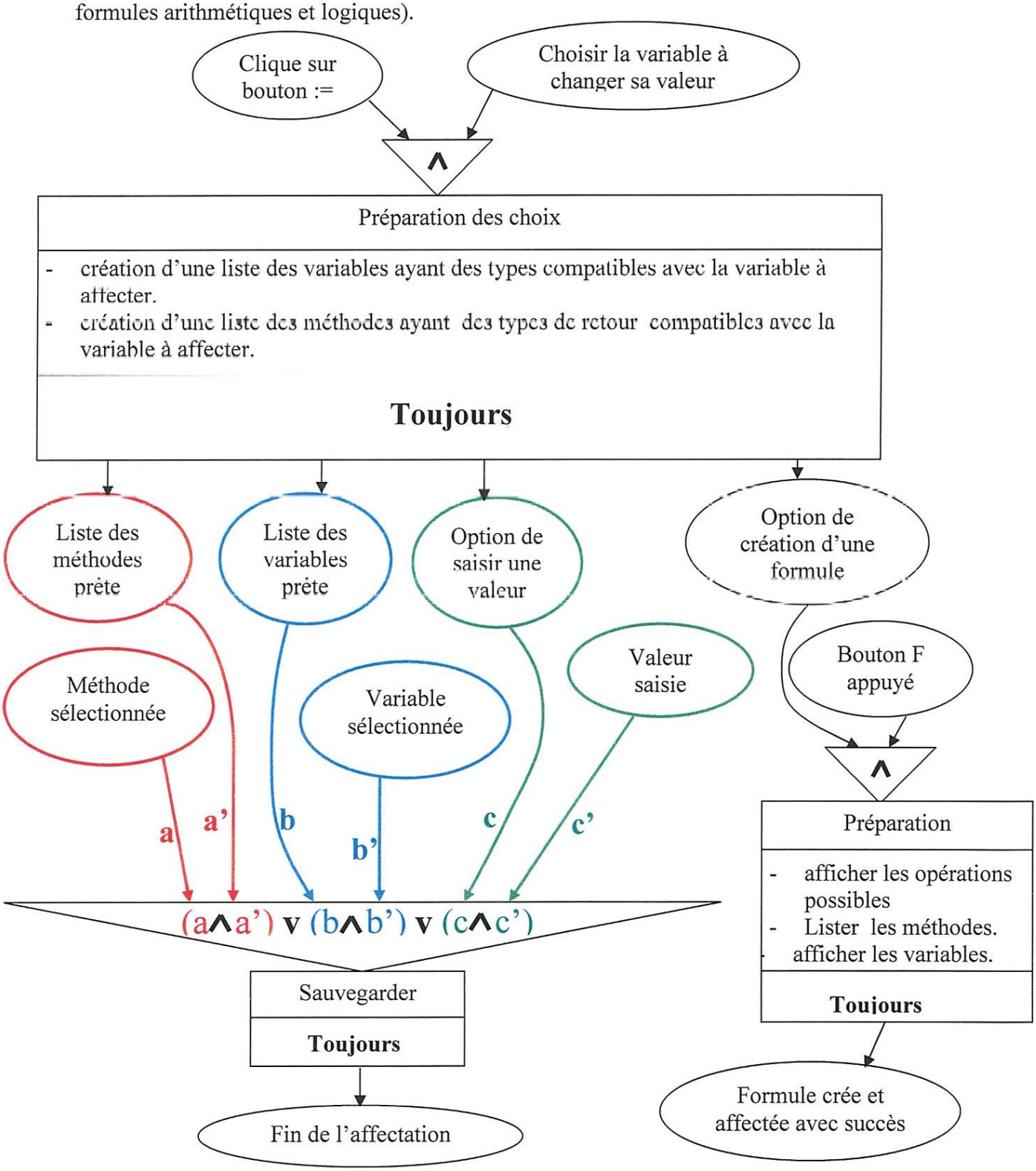


Figure 2. 7: *MCT du processus de l'affectation.*

- **Processus d'exécution du code source crée :** représente l'enchainement des opérations pour exécuter le code.

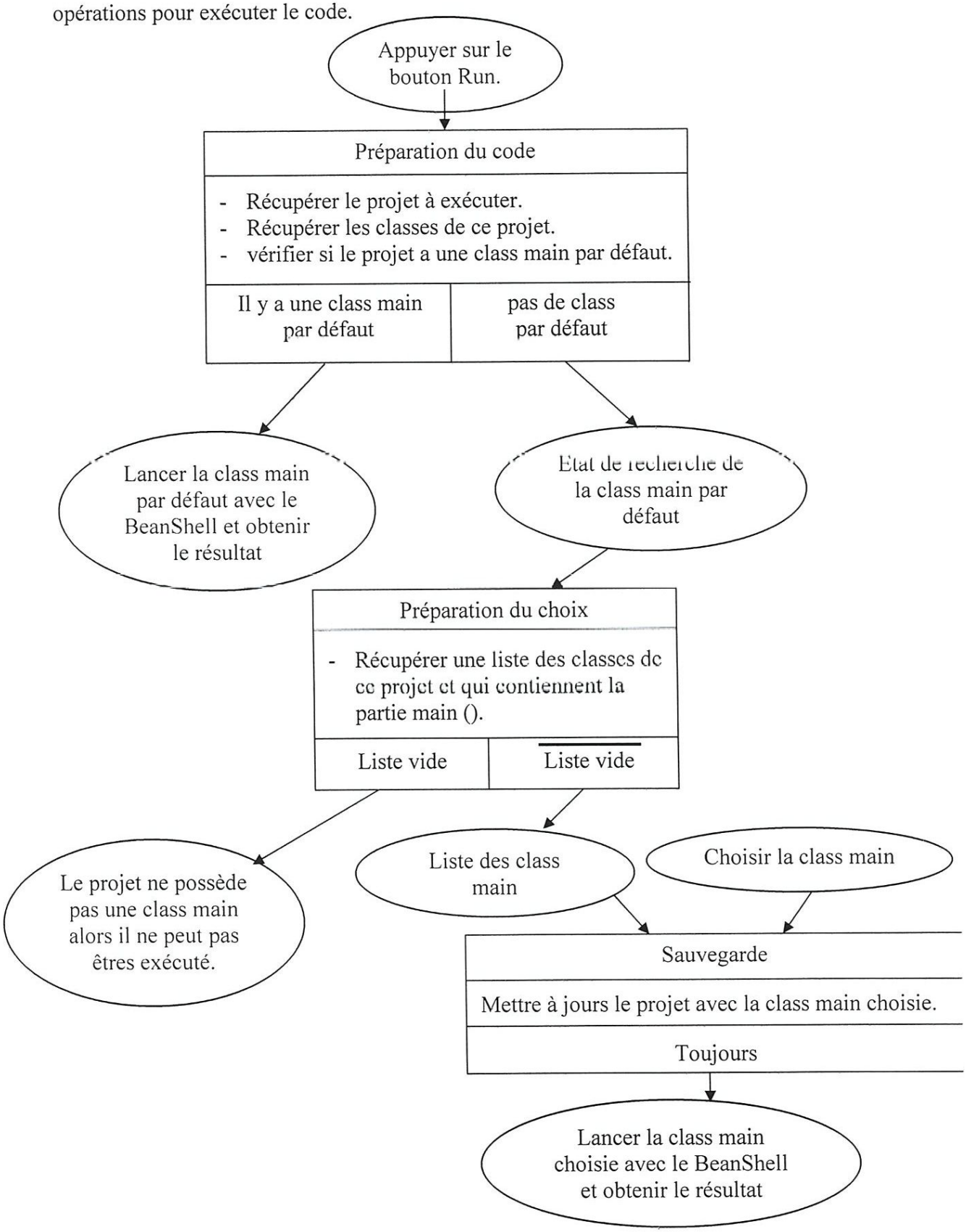


***Figure 2. 8** : MCT d'exécution du code.*

1.4. Conclusion

Dans ce chapitre, nous avons élaboré la conception de notre projet on se basant sur la démarche Merise.

Nous avons présenté les modèles de données et les modèles de traitements.

Dans le chapitre suivant, nous allons aborder la phase implémentation.

2. Implémentation

2.1. Introduction

Dans cette phase, nous allons concrétiser en termes de programmation notre conception effectuée dans la phase précédant pour obtenir un produit finale (logiciel) qui satisfait les contraintes de notre domaine d'étude.

Pour cela, nous avons utilisé quelques outils.

2.2. Environnements de travail

Ce sont les logiciels que nous avons utilisé pour réaliser notre logiciel.

2.2.1. MySQL server

Notre modèle physique de données (MPD) a été générer avec cet outil qui est un système de gestion de base de données (SGBD) gratuit et disponible sur les plateformes Unix, Linux et Windows. Il permet le stockage des données d'une base.

2.2.2. MySQL Workbench: (anciennement *MySQL administrator*).

C'est un logiciel de gestion et d'administration de bases de données MySQL créé en 2004. Via une interface graphique intuitive, il permet, entre autres, de créer, modifier ou supprimer des tables, des comptes utilisateurs, et d'effectuer toutes les opérations inhérentes à la gestion d'une base de données. Pour ce faire, il doit être connecté à un serveur MySQL. [6]

2.2.3. NetBeans

C'est un environnement de développement intégré (EDI), En plus de Java, il permet également de supporter différents autres langages, comme C, C++, JavaScript, XML, Groove, PHP et HTML de façon native par l'ajout de greffons. Il comprend toutes les caractéristiques d'un IDE moderne (éditeur en couleur, projets multi-langage, refactoring, éditeur graphique d'interfaces et de pages Web). [7]

2.2.4. PowerAMC Evaluation

PowerAMC est un logiciel de modélisation. Il permet de modéliser les traitements informatiques et leurs bases de données associées. [8]

L'environnement de modélisation de PowerAMC prend en charge plusieurs types de modèles, cet outil concentre sur la conception et la construction des modèles avec une source de données connectée.

Pour nous, on a utilisé le PowerAMC pour construire notre MCD.

2.2.5. BeanShell

BeanShell est un **interpréteur Java** disponible en standard (au moins dans l'installation de Java6 sous Debian). L'idée de BeanShell est de fournir un interpréteur Java pouvant être **embarqué** dans une application pour la rendre *scriptable*.

Mais BeanShell est aussi livré avec un **interpréteur de commande** (*shell*). Tout comme un shell Unix nous permet d'exécuter des tâches en invoquant des commandes Unix, le *shell* BeanShell nous permet d'exécuter des instructions écrites en Java. Cet outil se révèle donc bien pratique pour expérimenter du code Java. Ou à fortiori pour se familiariser avec ce langage. [9]

2.3. Les Modèles physiques de données et opérationnels de traitements

Ces Modèles répondent à la question « comment ? » évoquée dans la phase conception.

2.3.1. Modèle physique de données

Le MPD spécifie l'organisation physique des données c'est-à-dire leur représentation sur un SGBD (ici MySQL).

Ce modèle est obtenu en créant les tables relationnelles du MLD listées ci-dessus sur le SGBD en utilisant des clauses SQL.

Exemple de clauses SQL :

```
CREATE TABLE `class` (
  `idclass` int(11) NOT NULL AUTO_INCREMENT,
  `nomclass` varchar(45) NOT NULL,
  `idprj` int(11) NOT NULL,
  `codeSource` varchar(2500) NOT NULL,
  `mainSource` varchar(2500) NOT NULL,
  `fermeture` varchar(10) NOT NULL,
  PRIMARY KEY (`idclass`),
  KEY `class_ibfk_1` (`idprj`),
  CONSTRAINT `class_ibfk_1` FOREIGN KEY (`idprj`) REFERENCES `projet` (`idprj`)
  ON DELETE CASCADE ON UPDATE NO ACTION
)
```

Listing 2. 1 : *Code de création de la table « class »*

Pour consulter le code de la création de la Base de données, reportez-vous au fichier **Dump20160411.SQL** de l'application.

Notre Base de données est composée de douze (12) tables dont la liste est ci dessous (table 6).

Nom de la table sur SQGB	Description de la table
bouclef	La boucle *for*
bouclew	La boucle *while*
class	Une class Java
ifelse	Condition if/ else
instance	Instanciation globale d'une class
instancelocale	Instanciation locale d'une class
methode	Une méthode en java
parameters	Les paramètres d'une méthode
projet	Un projet en Java
users	Un utilisateur
variableglobale	Une variable globale « Field »
variablelcale	Une variable locale

Tableau 2. 3 : *La liste des tables de la BDD.*

2.3.2. Modèle organisationnel de traitement (MoT)

Le modèle organisationnel du traitement représente le choix d'organisation du système, la construction de modèle organisationnel répond aux questions suivantes :

- **Qui ?** : Qui fait la tâche.
- **Quand ? :** Enchainement des tâches.
- **Comment ?** : La nature de traitement (manuelle ou automatique)
- **Où** : le poste qui fait la tâche.

- **MoT de la création d'un nouveau projet :**

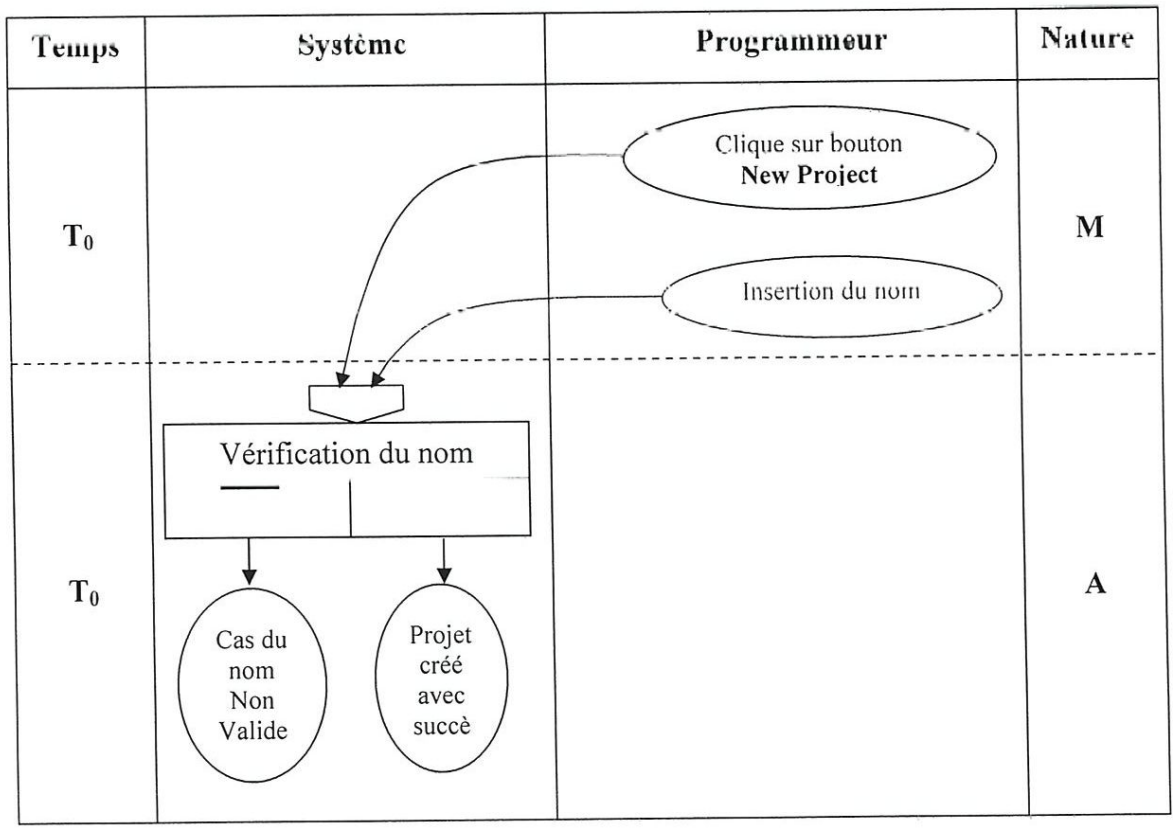


Figure 2. 9 : *MoT de la création d'un nouveau projet*

- **MoT de la création d'une nouvelle classe :**

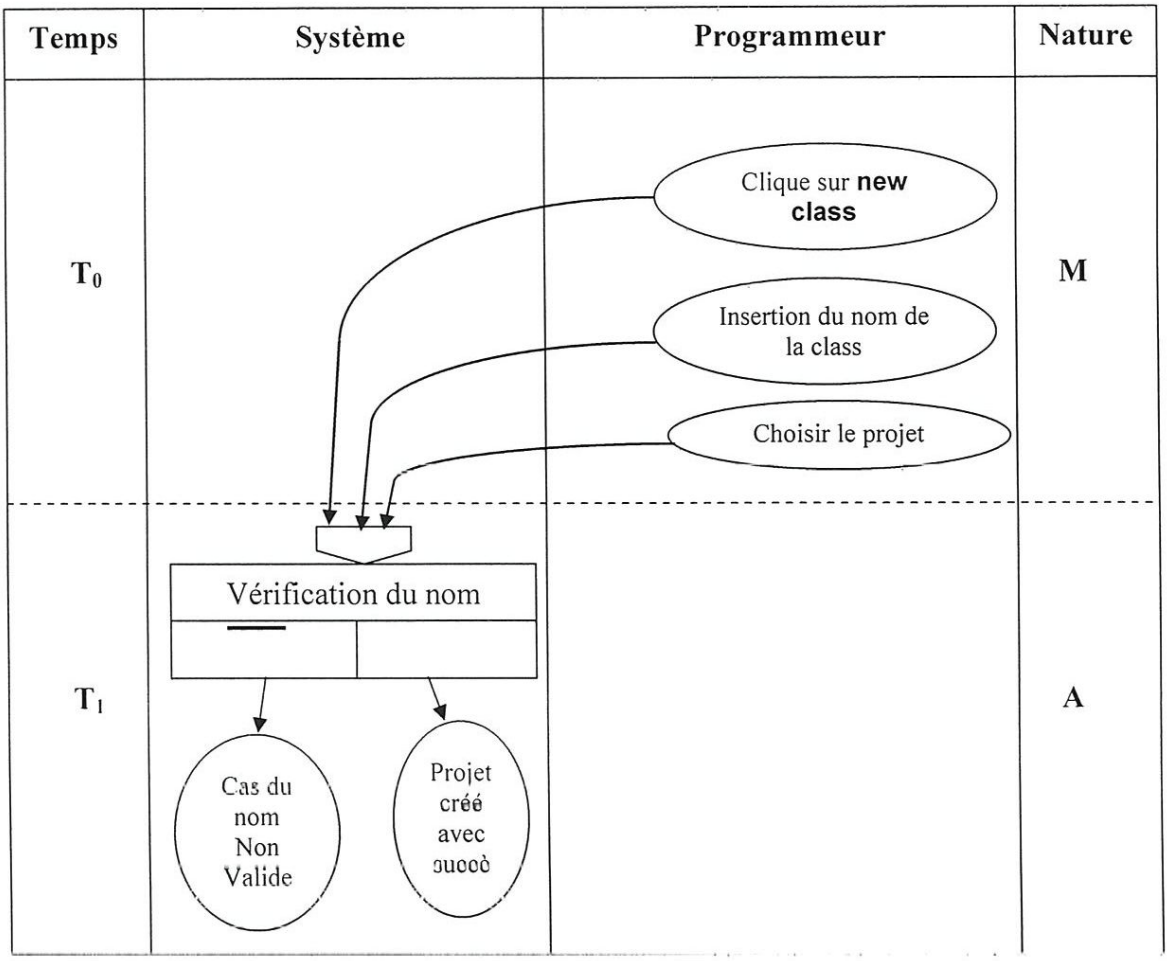


Figure 2. 10 : *MoT la création d'une nouvelle classe*

- **MoT de la création d'une nouvelle variable :**

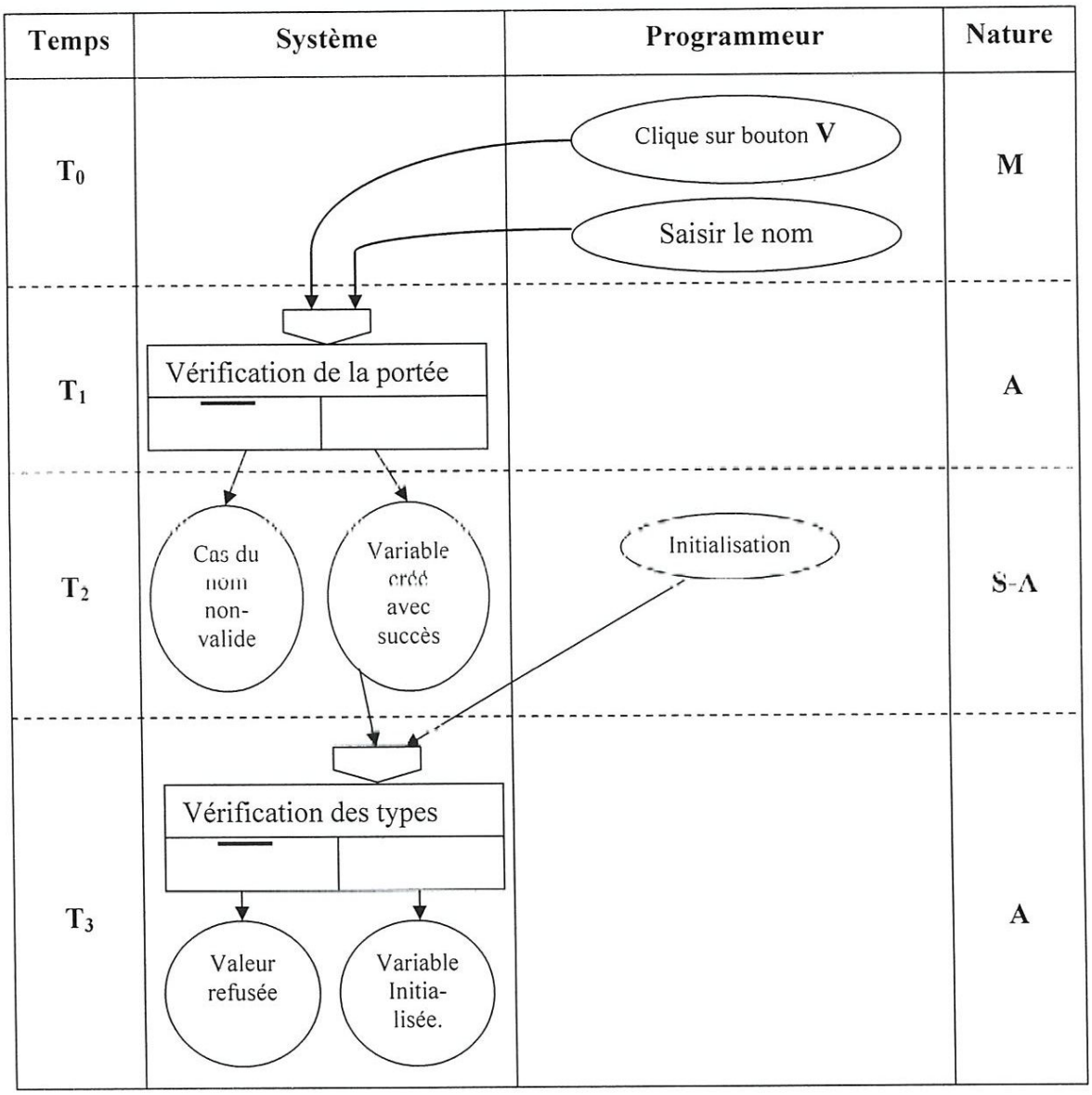


Figure 2. 11 : *MoT de la création d'une nouvelle variable*

- **MoT de la création d'une affectation :**

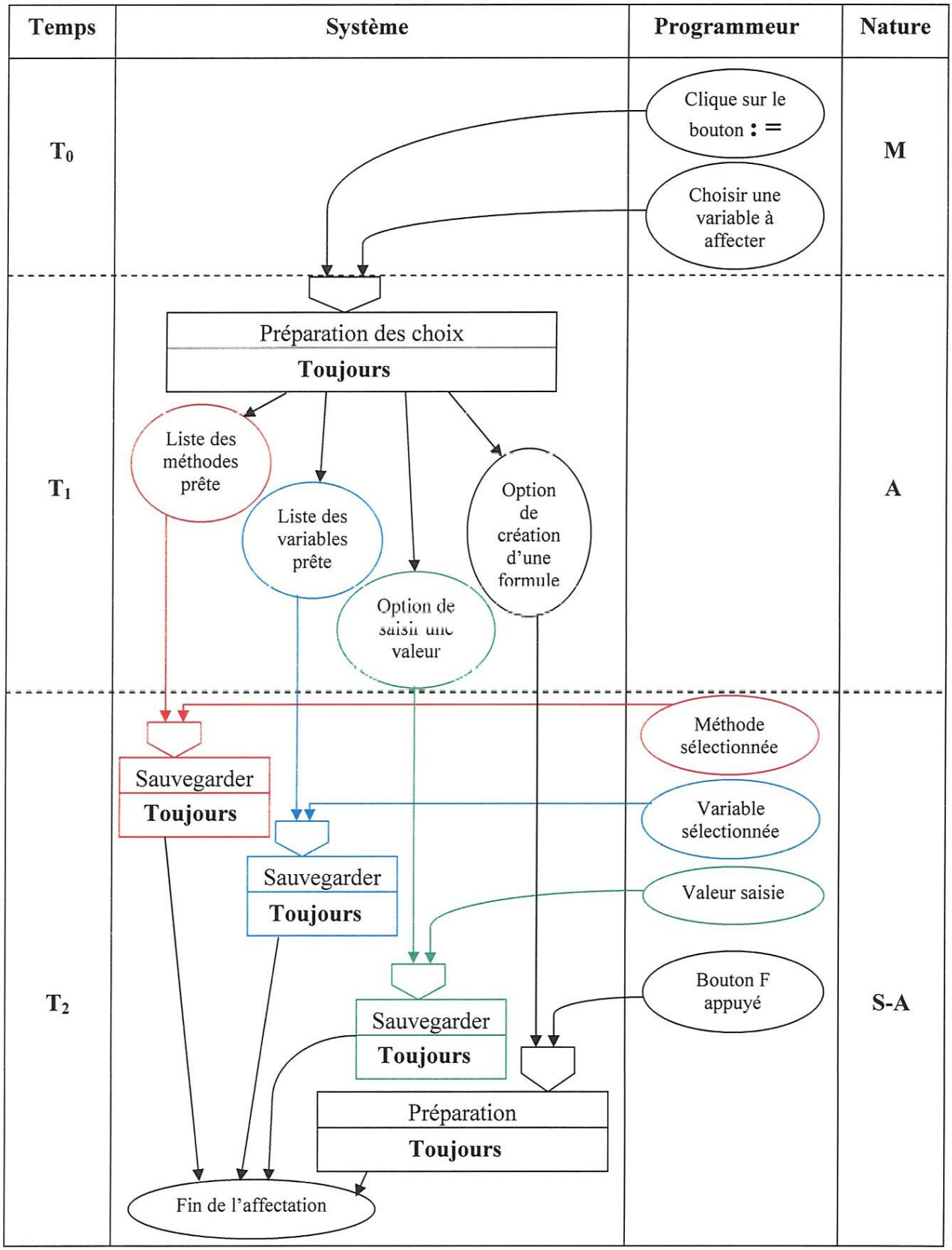


Figure 2. 12 :*MoT de la création d'une affectation*

- **MoT de l'exécution de code source :**

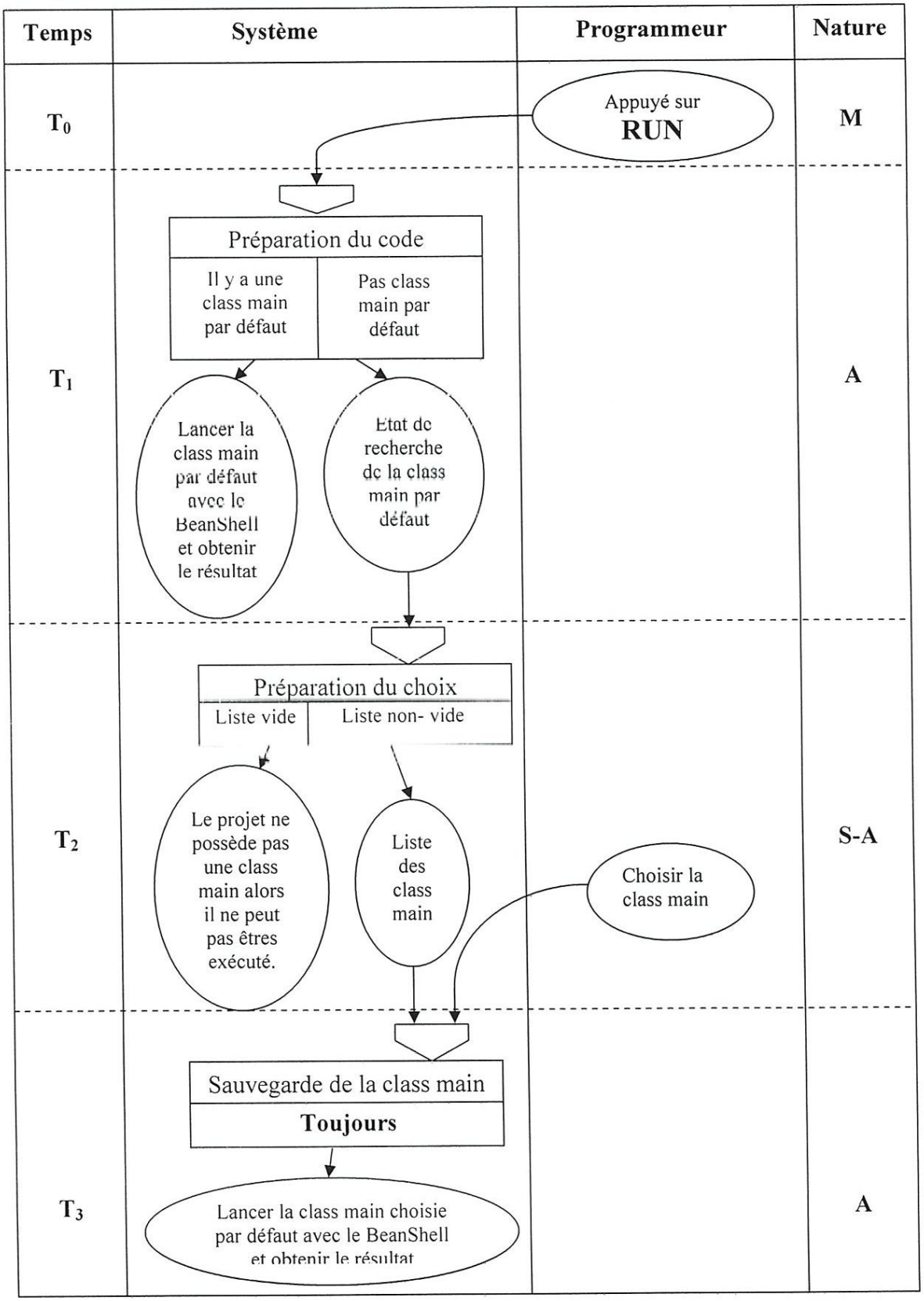


Figure 2. 13 : *MoT de l'execution de code source*

2.4. Présentation de l'application

2.4.1. Les Fonctions du système

1) Création des projets.

2) Création des classes.

3) Création du code.

   - Création des variables.
   - Création des boucles (simples ou imbriquées).
   - Création des Méthodes.
   - Création des instances (des classes ou de méthodes).
   - Création des formules (logiques ou arithmétiques).

4) Modification du code.

5) Exécution du code source créé.

6) Suppression des classes.

7) Suppressions des projets.

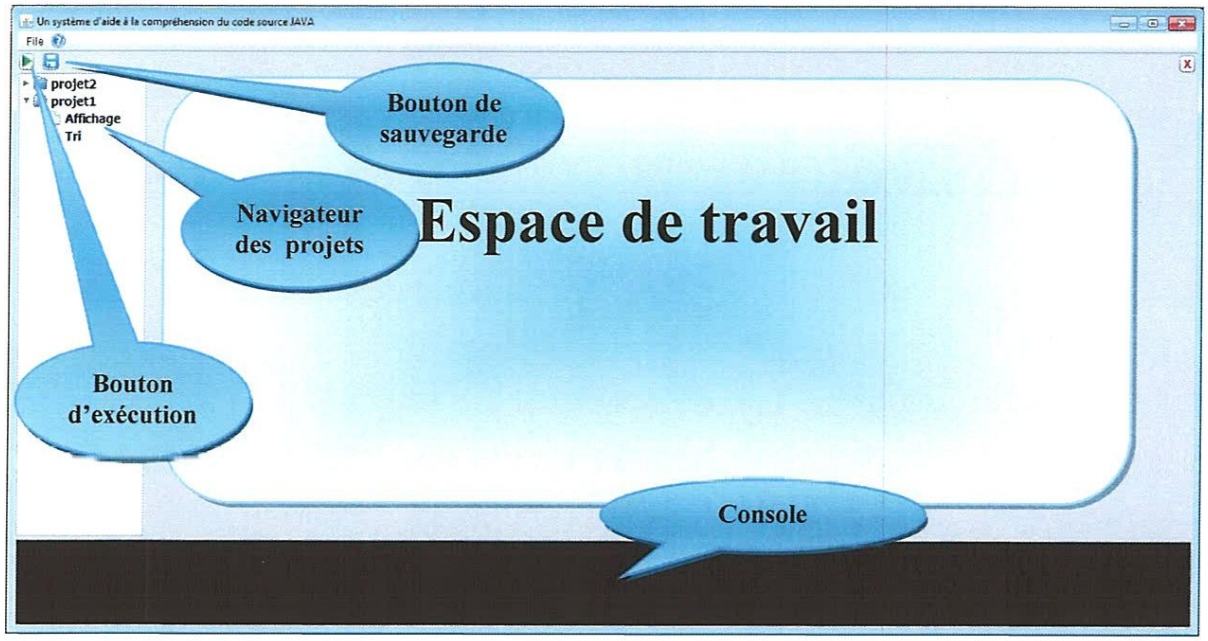
2.4.2. Les Contrôles

Tous les contrôles relatifs au typage, plages de valeur et présence des données, Les portées communes et la bonne syntaxe des différentes composantes sont intégrés dans notre système.

Exemple :

Tous les attributs nom (nomvar, nomprj, nomclass...) sont obligatoires, ne commencent pas par des chiffres, ne peuvent pas être des mots clés et deux noms ne peuvent pas être identiques.

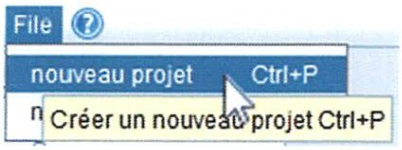
2.4.3. Interface d'accueil



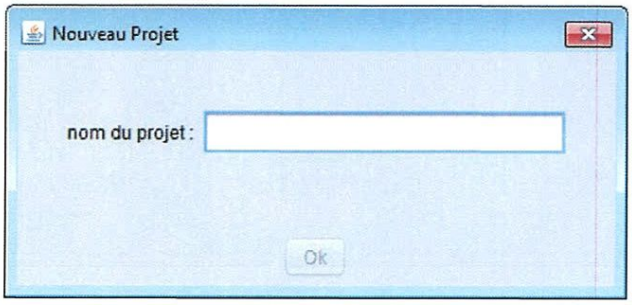
A. Création d'un projet

Tout d'abord, il faut ouvrir un projet s'il existe en le sélectionnant dans la zone de navigation des projets.

Nous pouvons aussi créer un nouveau projet en appuyant sur « File » comme suit :

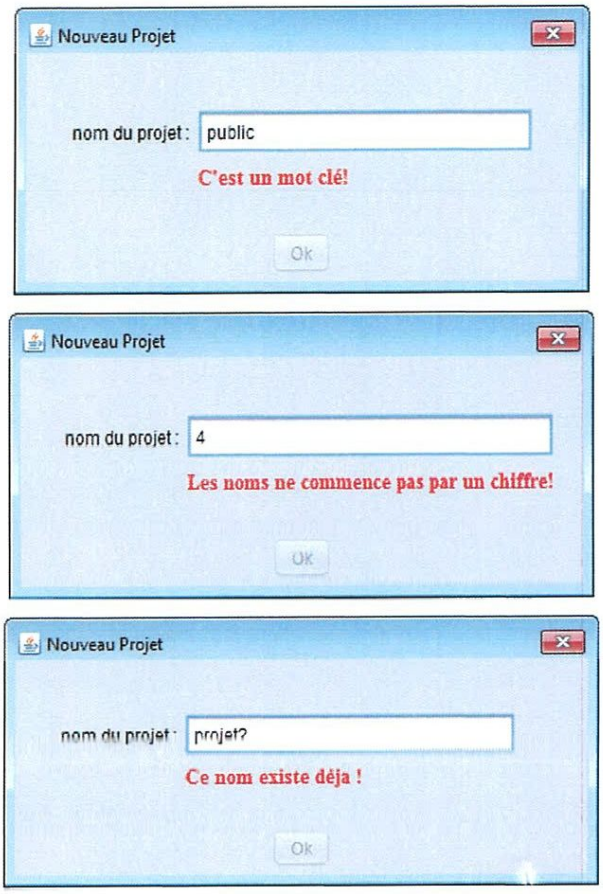


En choisissant « nouveau projet », la fenêtre d'ajout d'un projet apparait sur l'écran :



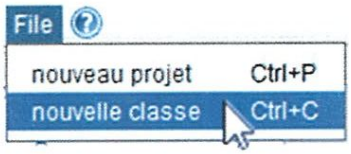
Comme nous l'avons indiqué ci-dessus (dans le titre contrôle) ; le nom est obligatoire, il doit être unique, commence par une lettre et ne peut pas être un mot clé.

Voici quelques cas non acceptés :

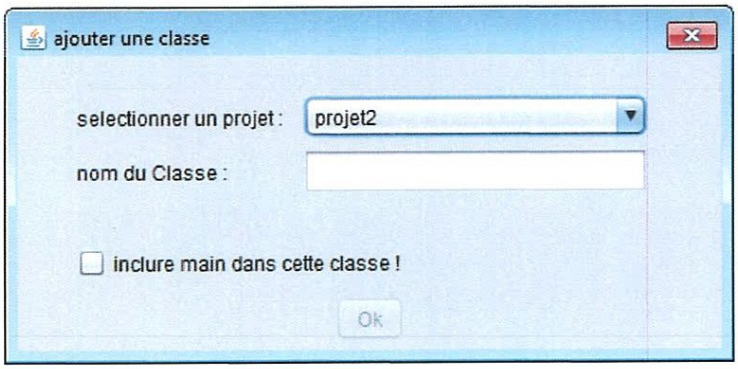


B. Création d'une classe

Pour pouvoir écrire le code source il faut préciser la classe en ouvrant cette dernière depuis le navigateur des projets ou par la création d'une nouvelle classe en appuyant sur « File » :

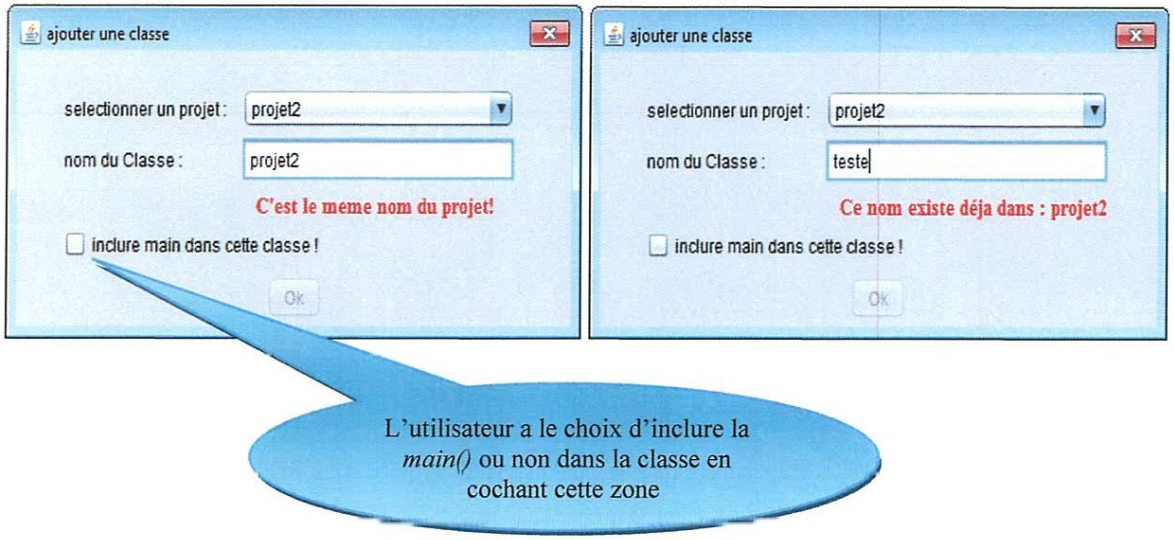


En choisissant « nouvelle classe», le formulaire d'ajout d'une classe apparait :

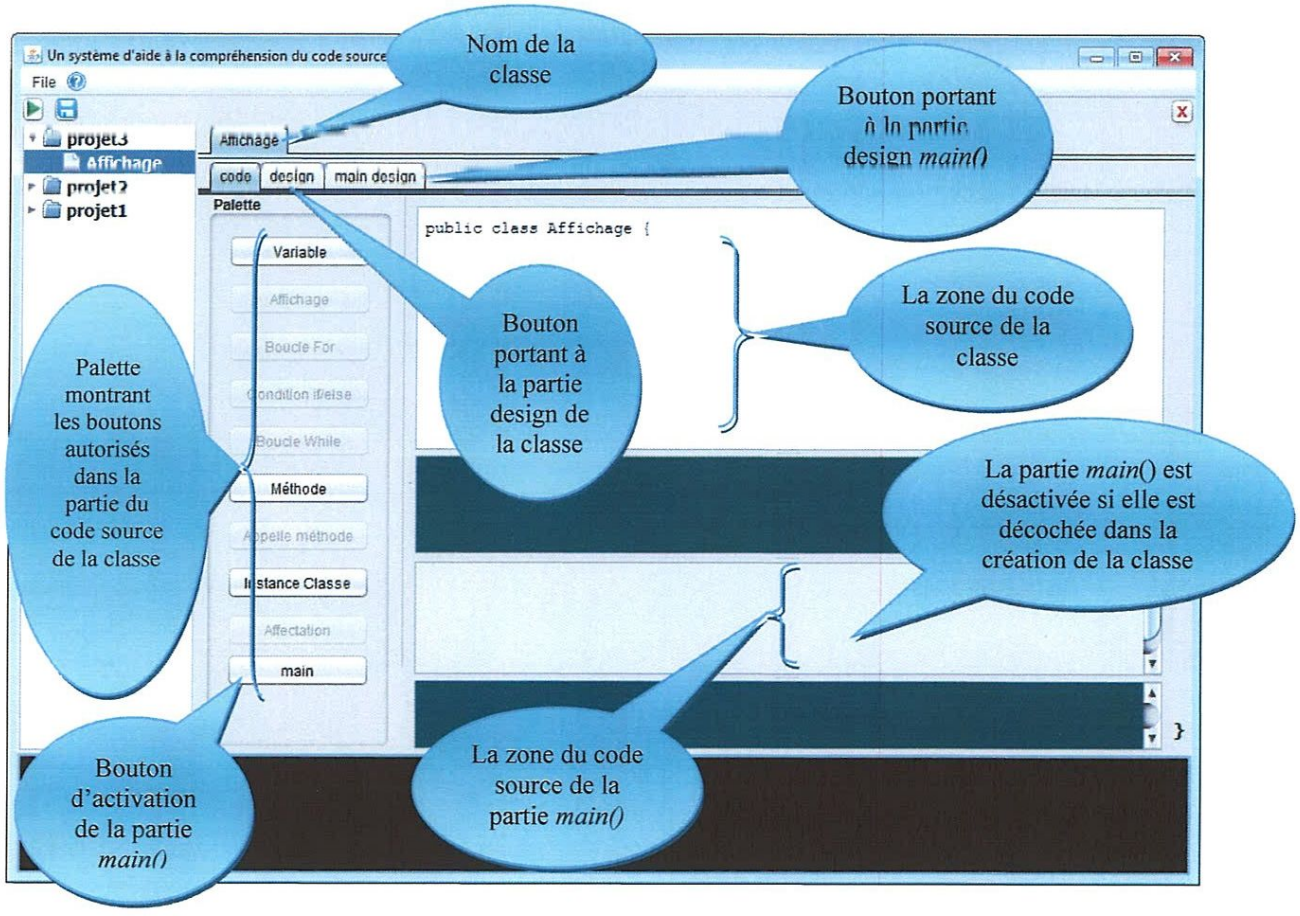


Pour valider l'ajout d'une classe il faut sélectionner un projet déjà créé.

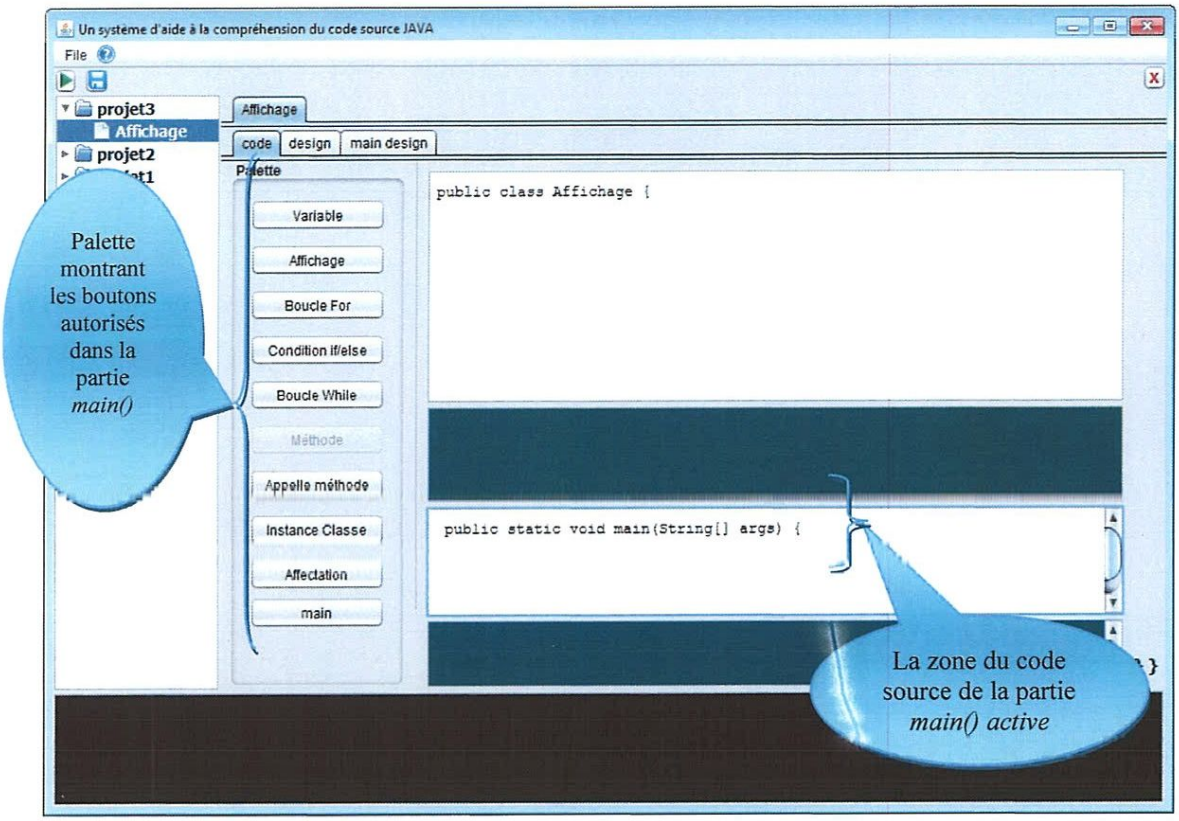
Voici quelques exemples de noms de la classe qui sont non acceptés :



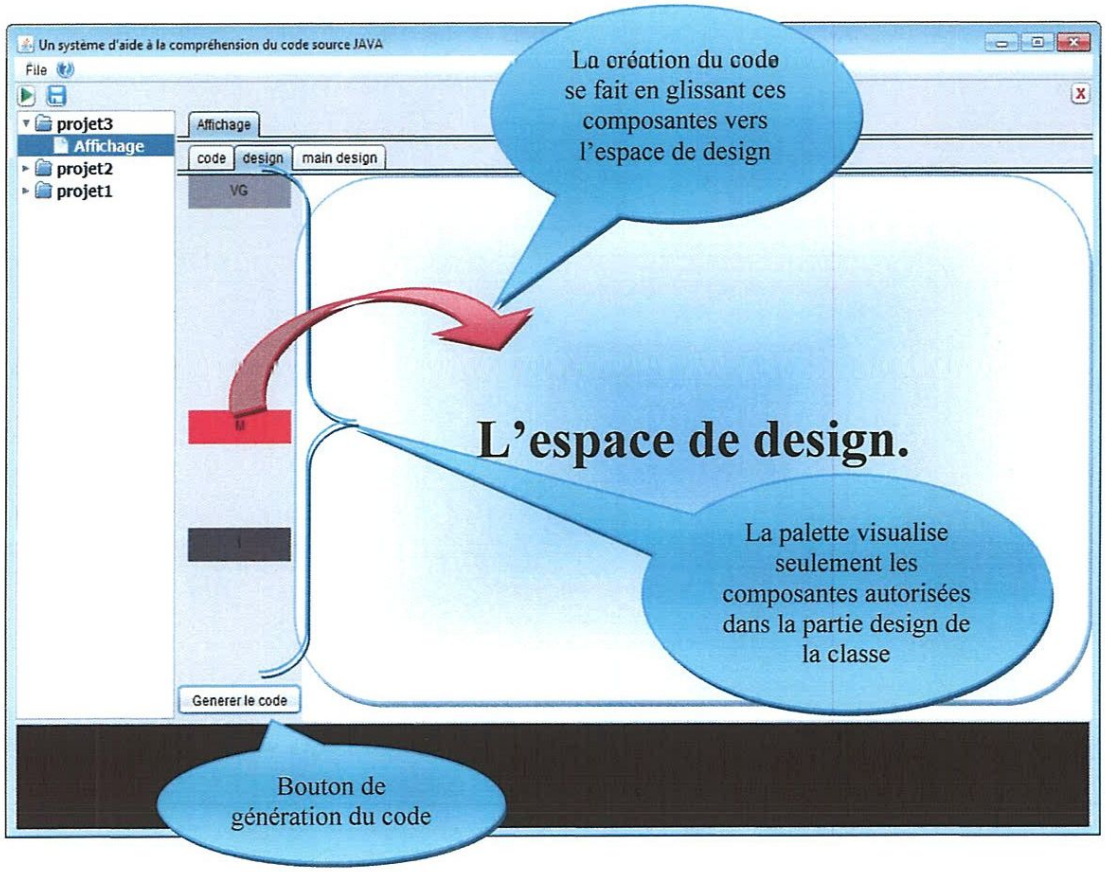
Voici l'espace de travail après l'ouverture de la classe :



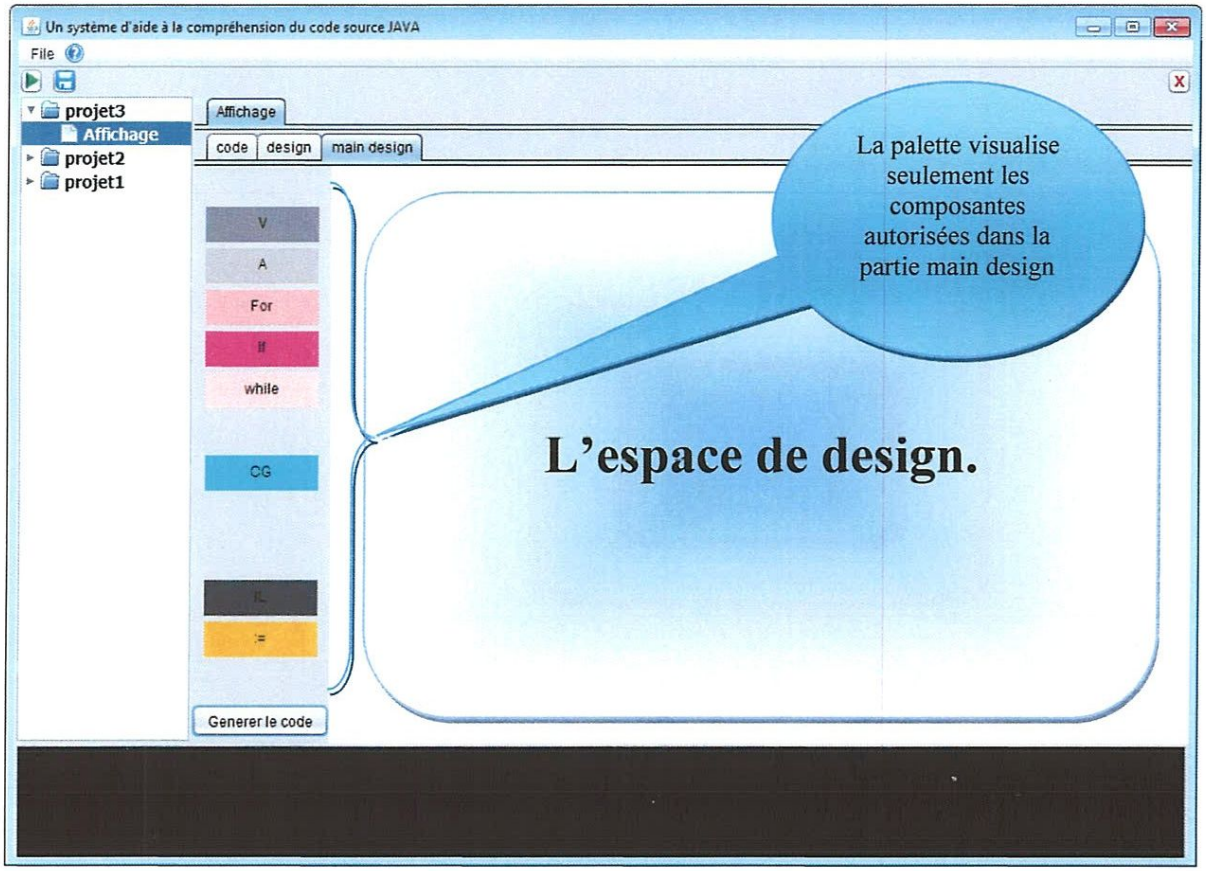
Si nous appuyons sur le bouton « **main** », la partie *main()* sera activée :



L'onglet « **design** » menant à la partie visuelle de la classe via un espace graphique ci-dessous



Et voici la partie « **main design** » de cette classe :



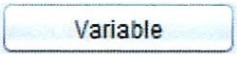
Index :

Symbole	Signification
V	Variable locale
VG	Variable globale
A	Affichage
For	Boucle For
while	Boucle While
If	Condition If/Else
M	Méthode
CG	Appel d'une méthode
I	Instance Globale
IL	Instance locale
:=	Affectation

C. Création du code source

Pour créer n'importe quelle instruction disponible dans la palette il faut appuyer tout simplement sur le bouton approprié dans l'onglet « code » ou par le glissement des composantes dans les deux onglets « design » et « main design ».

Exemple :

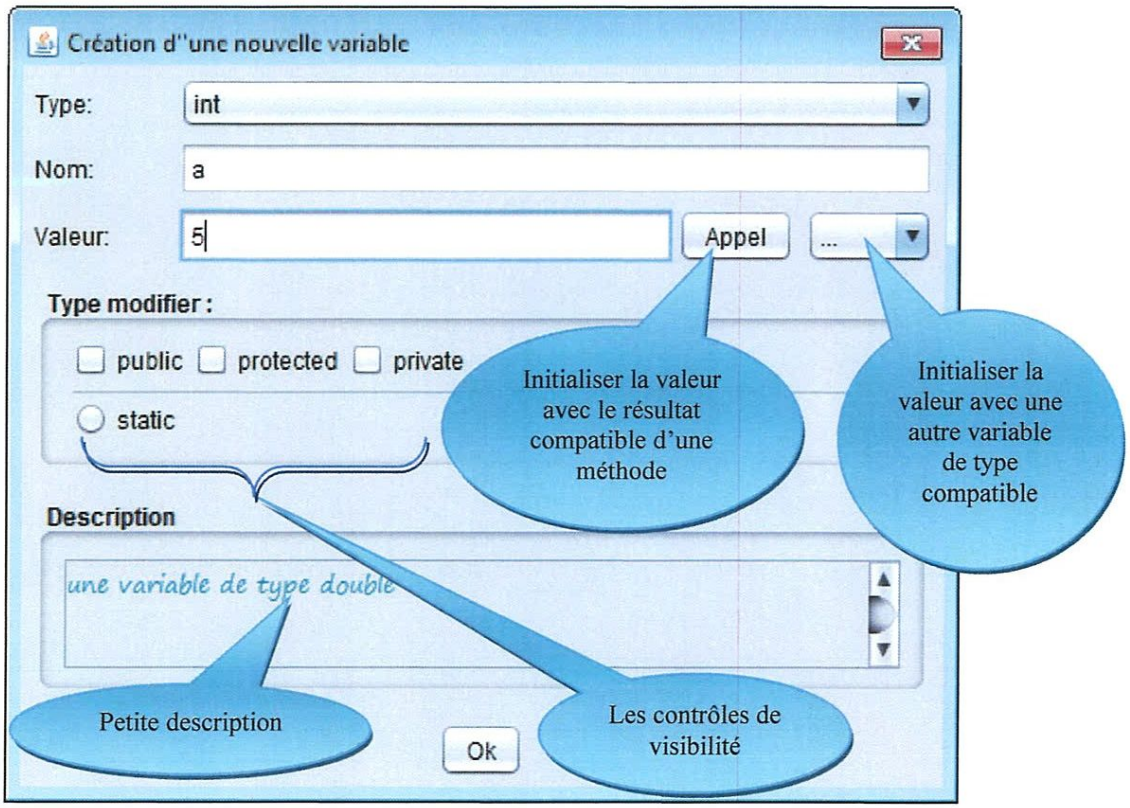
Le click sur le bouton de variable 

ou bien le glissement de la composante 

vers l'espace de travail.

- **Création d'une variable :**

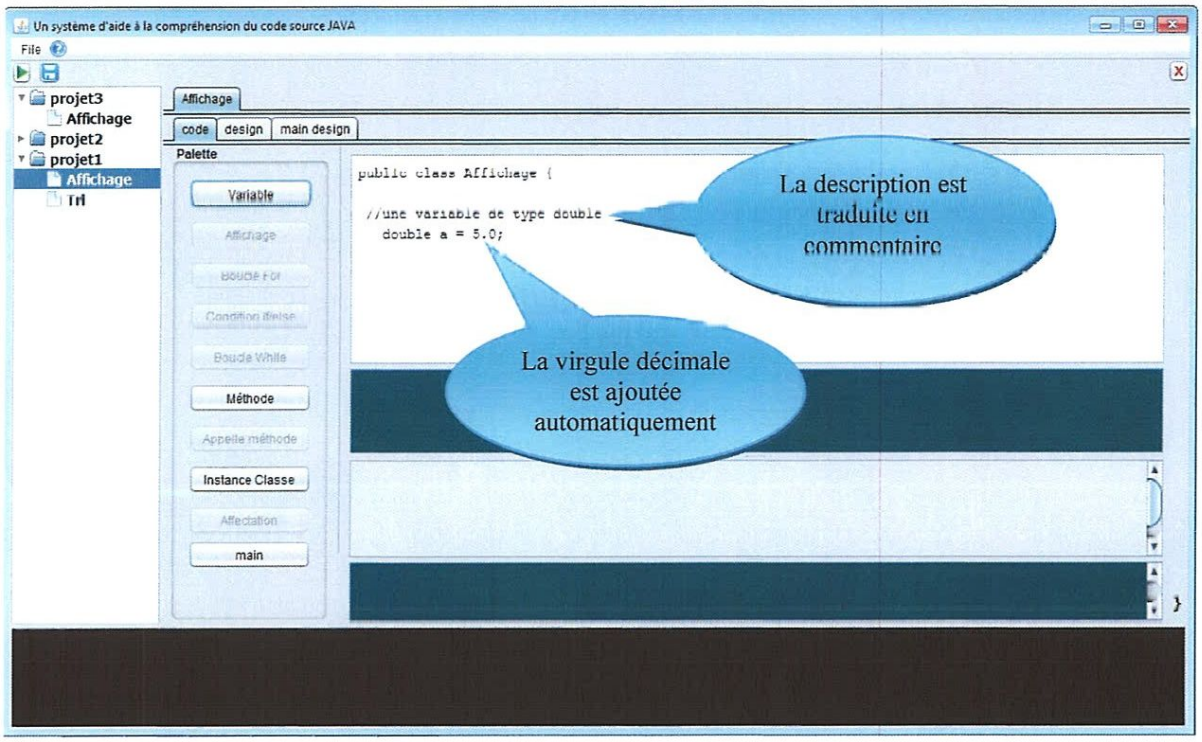
  Selon la partie (code, design ou main design, boucles, méthodes, if/else) où l'action de la création est faite, le système décide si c'est une variable globale ou locale (l'importance de cette décision est le respect des contraintes de la portée et de la visualisation). Dans le cas d'une variable globale, le formulaire de création est le suivant :



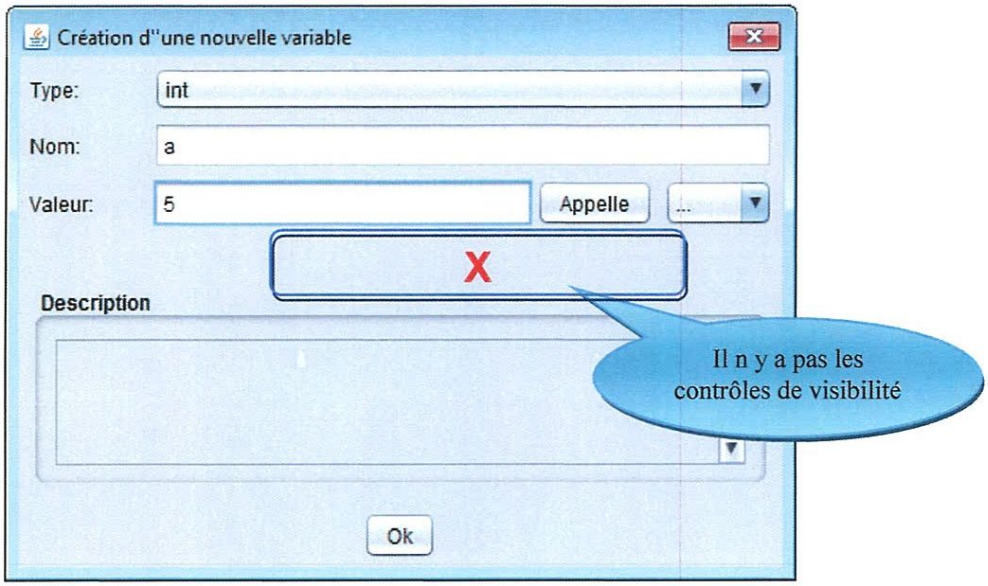
La valeur de la variable doit être de même type que le type choisi ;

- en cas d'un « int », dans la zone valeur seulement les chiffres sont autorisés.
- en cas d'un « boolean », la valeur doit être soit « true » soit « false ».
- en cas d'un « double », les chiffres et la virgule décimale sont autorisés (la virgule est obligatoire : en l'écrivant manuellement ou bien automatiquement par le système).

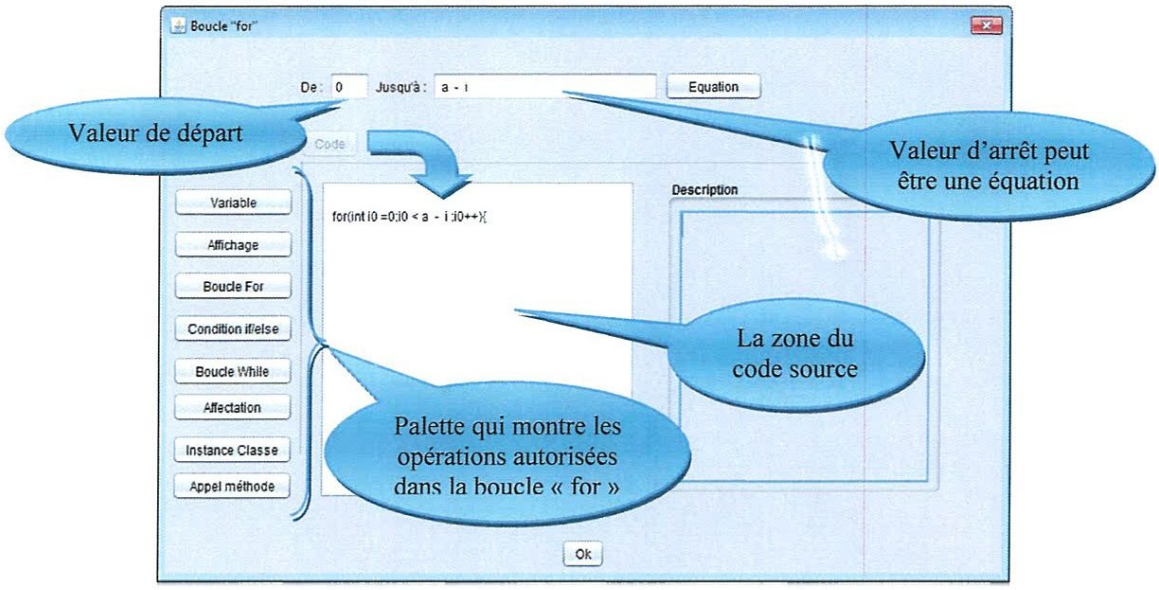
Et voici le code source généré :



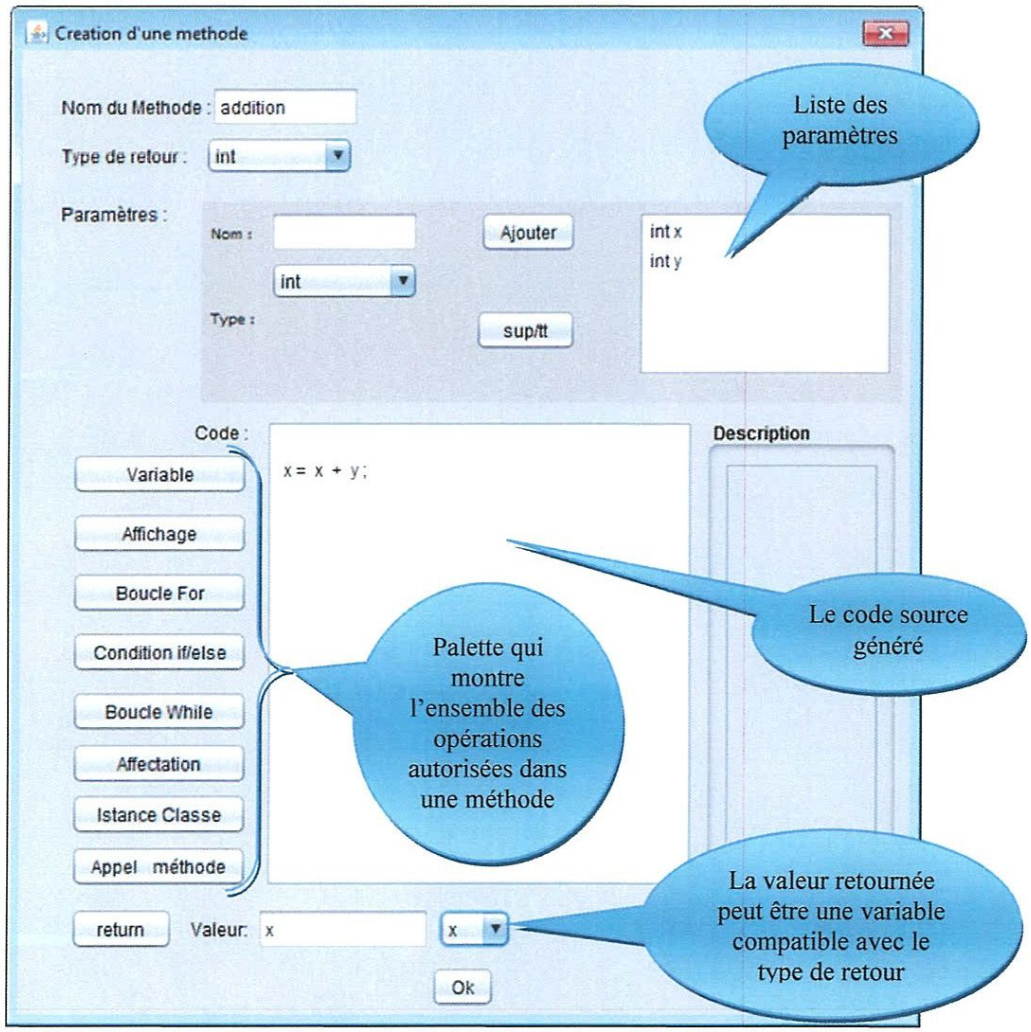
Dans le cas d'une variable locale, nous s'apercevrons qu'il y a une seule différence dans la partie visualisation car les variables locales sont restreintes à la visibilité locale.

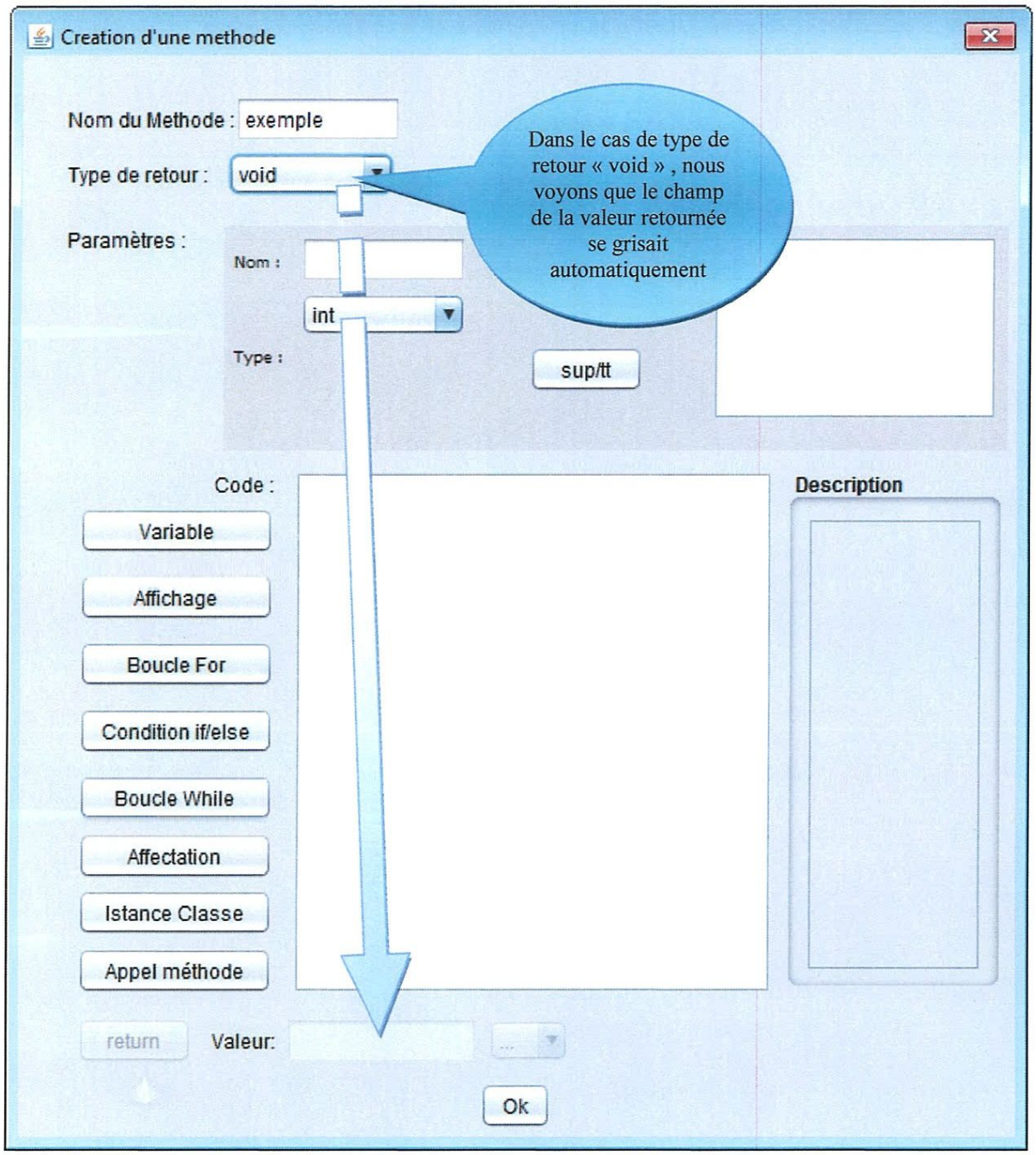


- Création des Boucles (nous prenons l'exemple de la boucle « for ») :

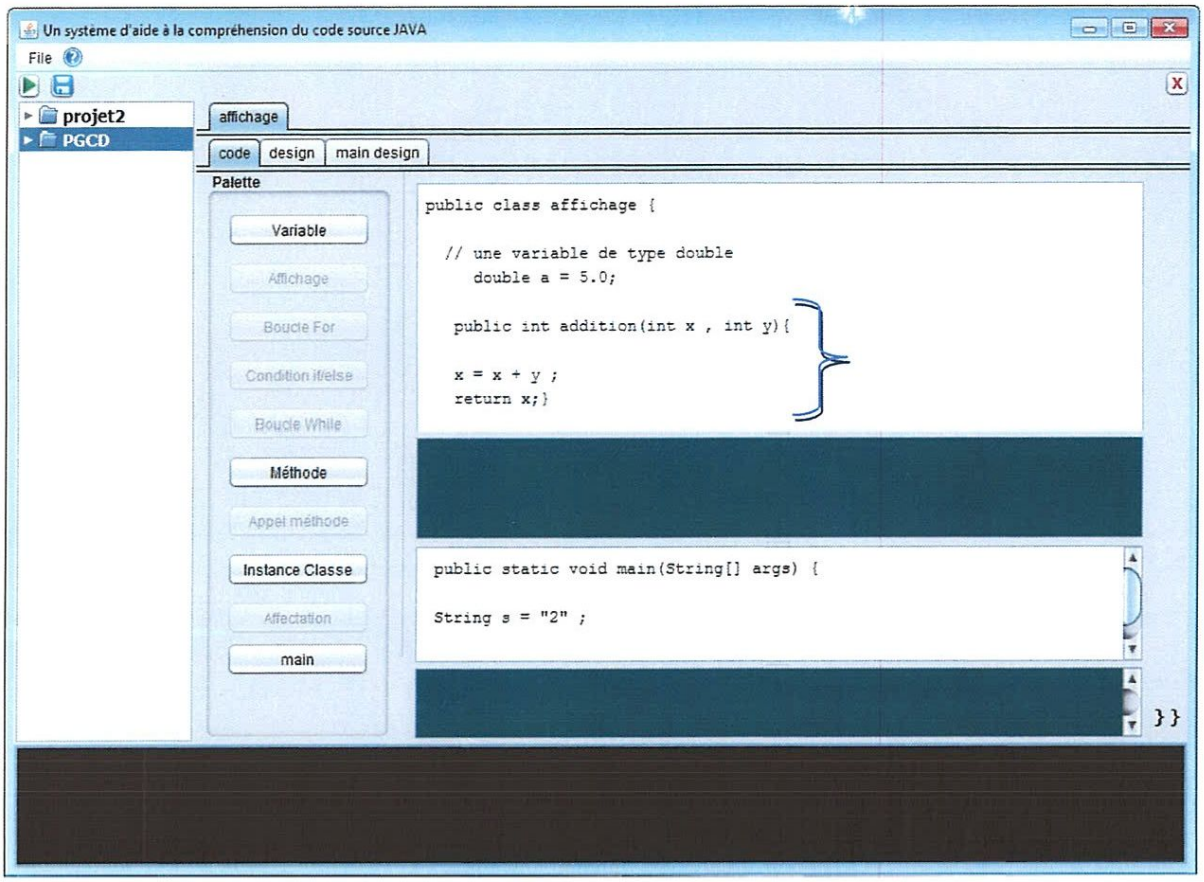


- Création d'une méthode :

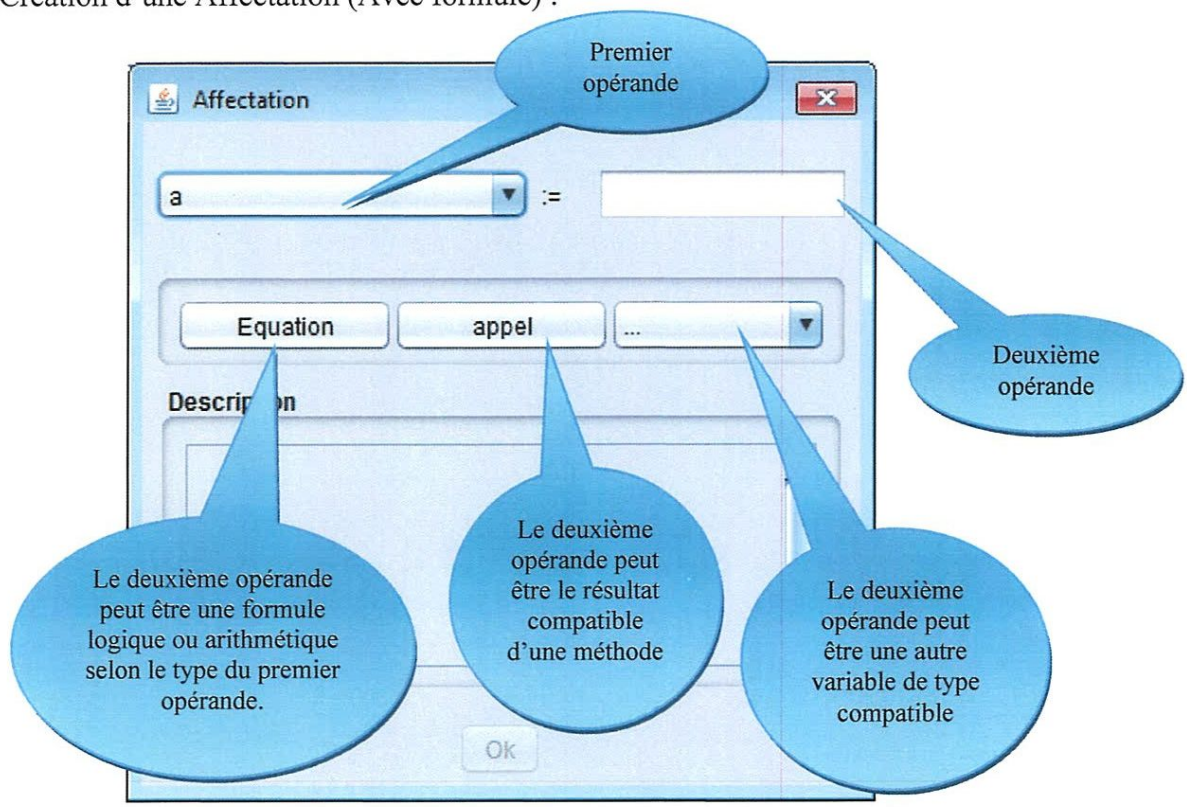



Creation d'une methode
Nom du Methode : exemple
Type de retour : void
Dans le cas de type de retour « void » , nous voyons que le champ de la valeur retournée se grisait automatiquement
Paramètres :
Nom :
int
Type :
sup/tt
Code :
Description
Variable
Affichage
Boucle For
Condition if/else
Boucle While
Affectation
Istance Classe
Appel méthode
return
Valeur:
Ok

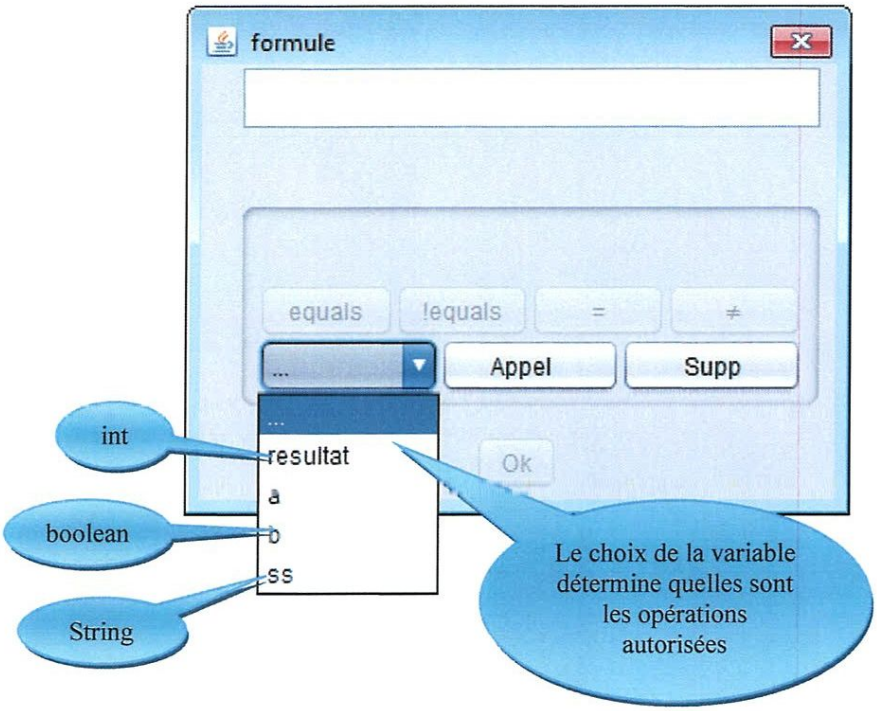
Et voici le code généré :



- Création d'une Affectation (Avec formule) :



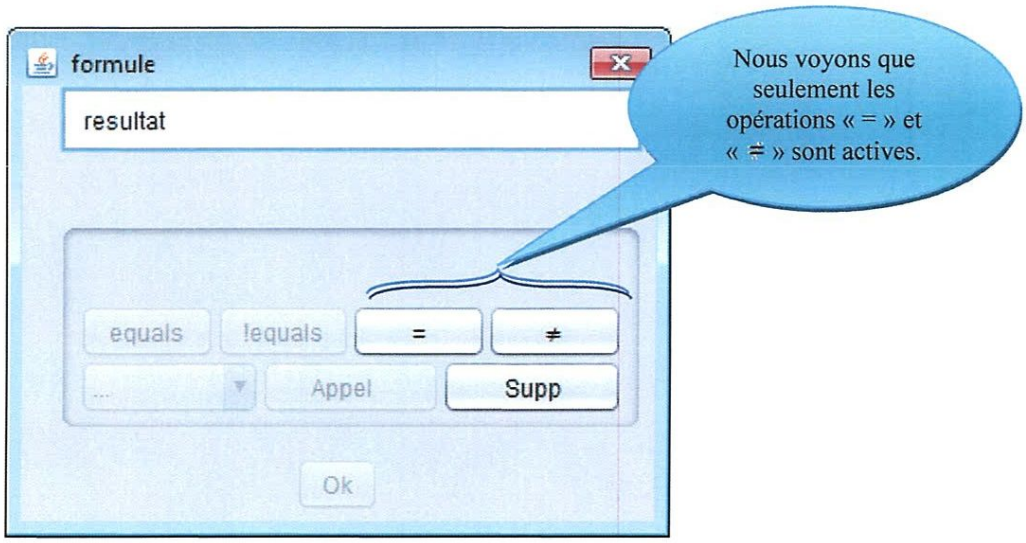
- Si le premier opérande est de type « *boolean* », le bouton « **Equation** » mène à ce formulaire de création des formules logiques :



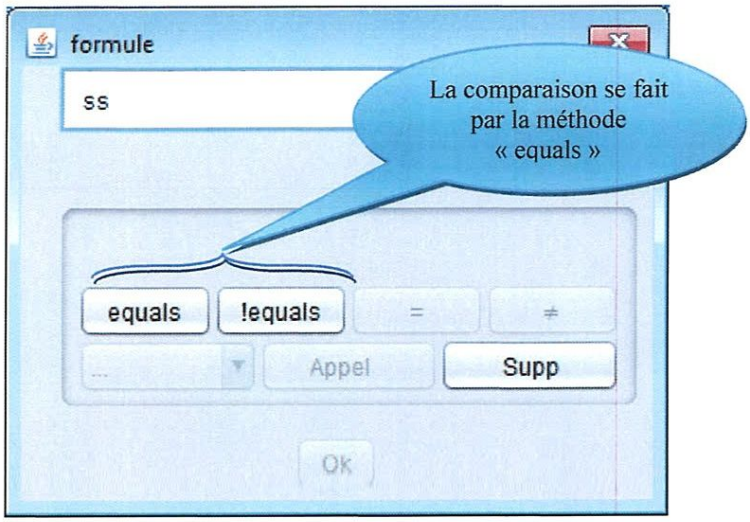
Si nous choisissons un type élémentaire (*Int* , *double*...) ; les opérations logiques seront les opérations de comparaison simple ;

Exemple :

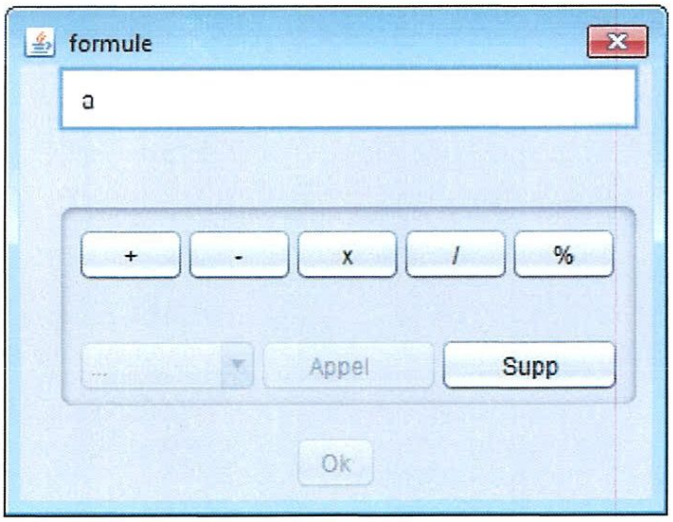
- si nous choisissons une variable de type « *int* » :



- si nous choisissons une variable de type objet (comme le « *String* ») :

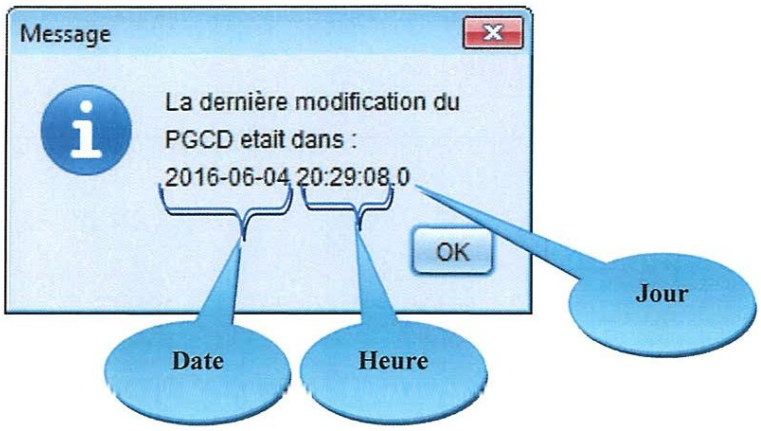


- Si le premier opérande est d'un type « *int* » ou « *double* », le bouton « **Equation** » mène à la fenêtre de création des formules arithmétiques (pas d'opérations de comparaison) :



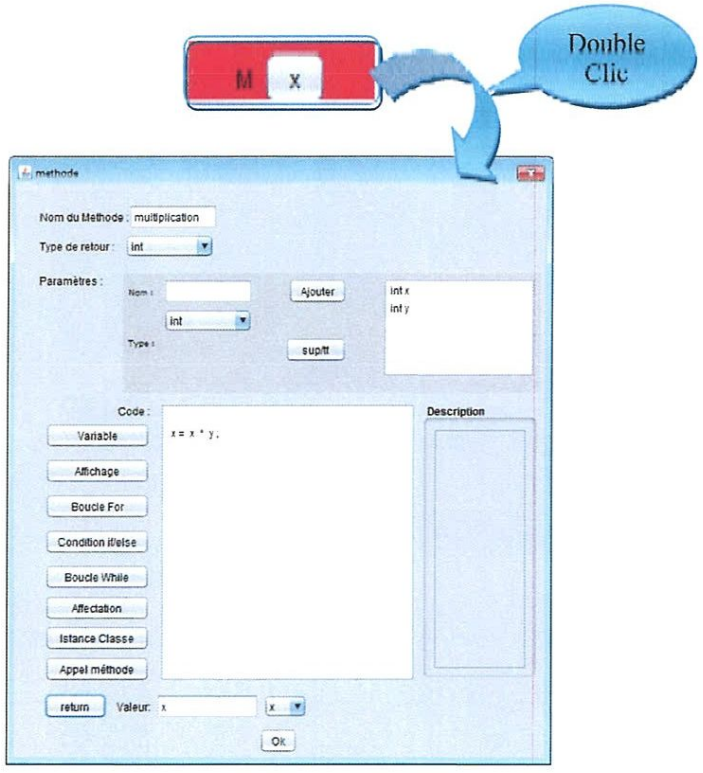
- Si le premier opérande est de type « *String* » ; le bouton « **Equation** » mène à la fenêtre qui manipule les « *String* » (concaténation, ou autres méthodes).

- Pour consulter la dernière modification appliquée sur un projet ; il suffit de faire un double clic sur le nom du projet :



D. Modification du code

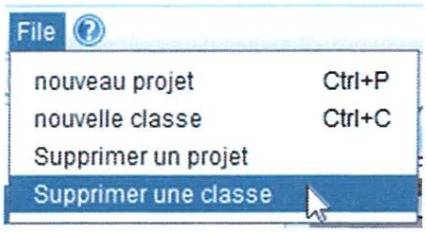
Pour modifier une composante, il suffit d'un double clic sur cette composante est le formulaire approprié va s'afficher (rempli avec les ancienne valeurs).



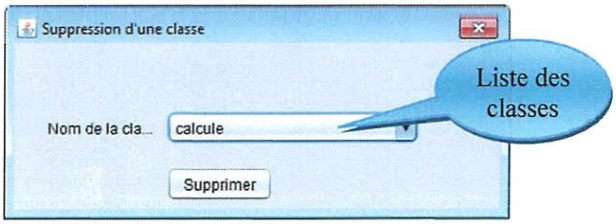
Pour supprimer une composante, il suffit d'appuyer sur le bouton « **X** » (à condition qu'elle soit la dernière composante insérée).

E. Suppression d'une classe

Pour supprimer une classe, appuyez sur le bouton « **File** », puis Supprimer une classe, comme suit :

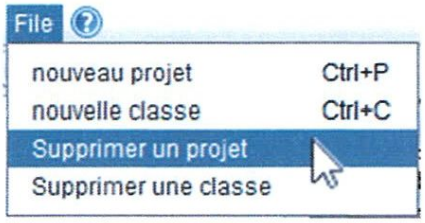


Il faut sélectionner une classe, puis « **Supprimer** » pour accomplir la suppression comme suit :

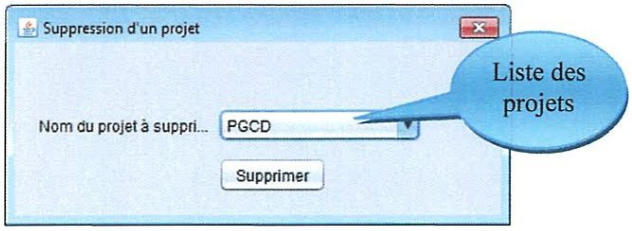


F. Suppression d'un projet

Pour supprimer une classe, appuyez sur le bouton « **File** », puis Supprimer un projet, comme suit :



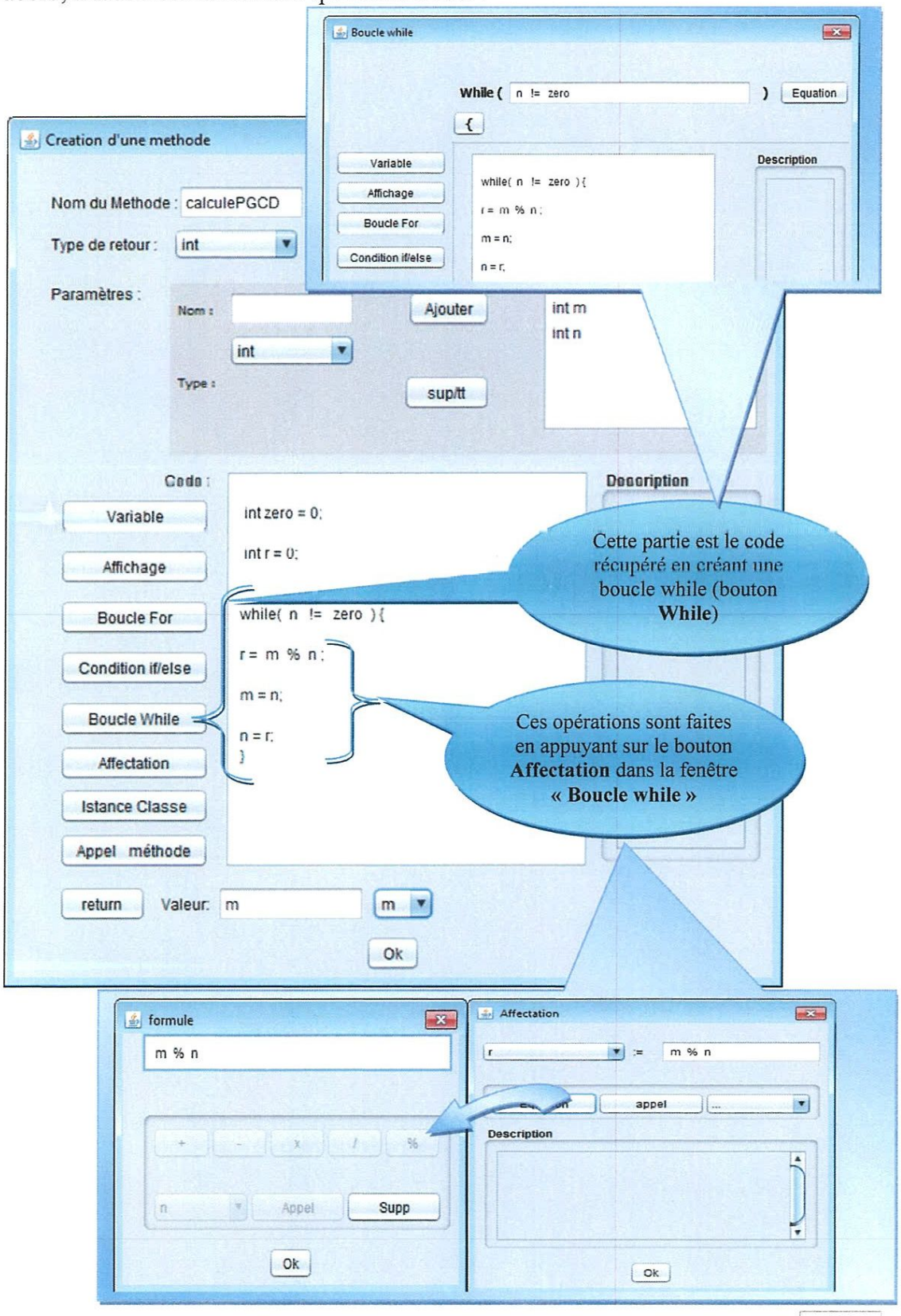
Il faut sélectionner un projet, puis « **Supprimer** » pour accomplir la suppression comme suit :



2.5. Exemple réel

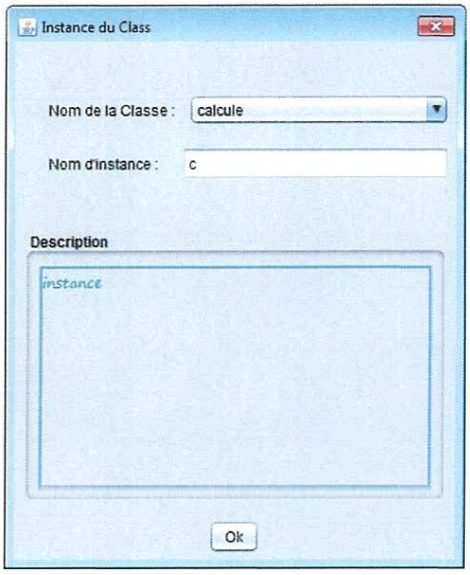
Nous allons créer un code qui calcule le PGCD de deux nombres entiers. Le traitement va être dans la classe « *calcule* » et le système d'appel va être dans la classe « *affichage* ».

- tout d'abord, il faut créer la méthode qui fait le calcul :

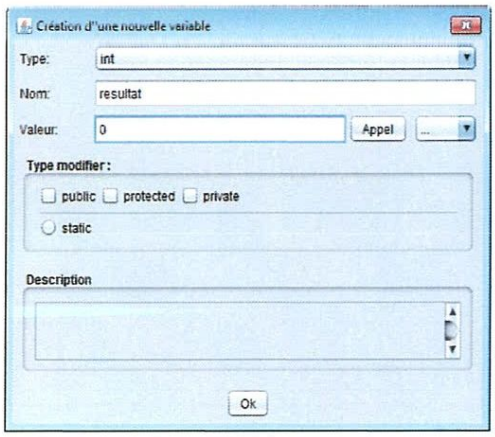


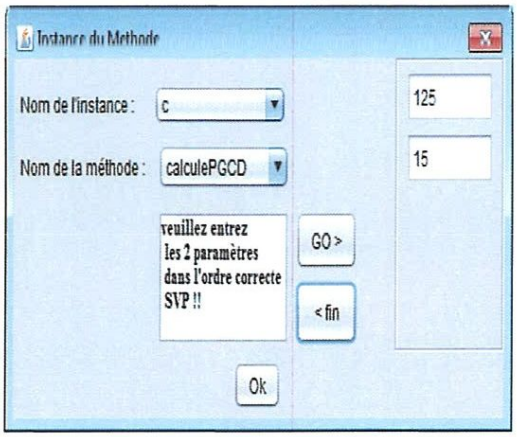
- Ensuite pour afficher le résultat il faut :

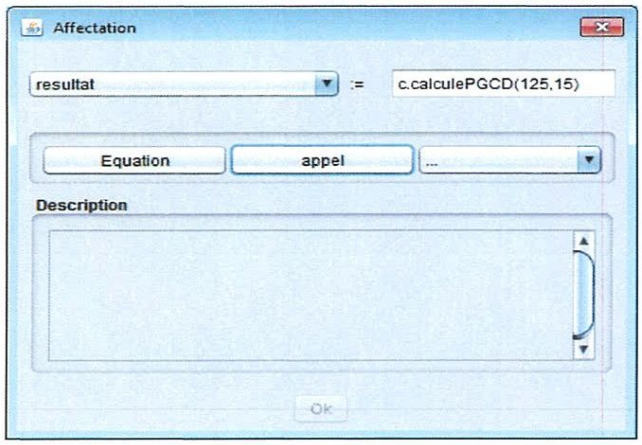
- créer une « *instance* » de la classe « *calcule* ».



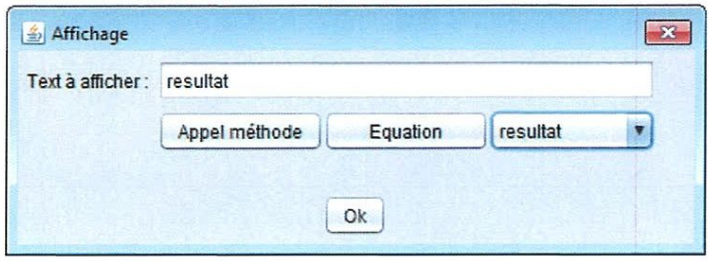
- Faire « *appel* » à la méthode « *calculePGCD* » pour récupérer le résultat dans une variable par l'Affectation.



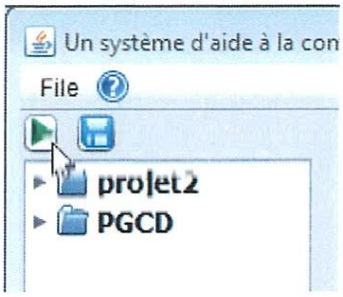




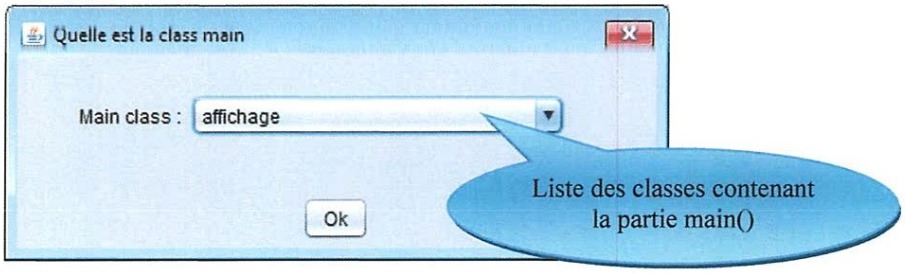
- Afficher la variable portant le résultat par « *Affichage* ».



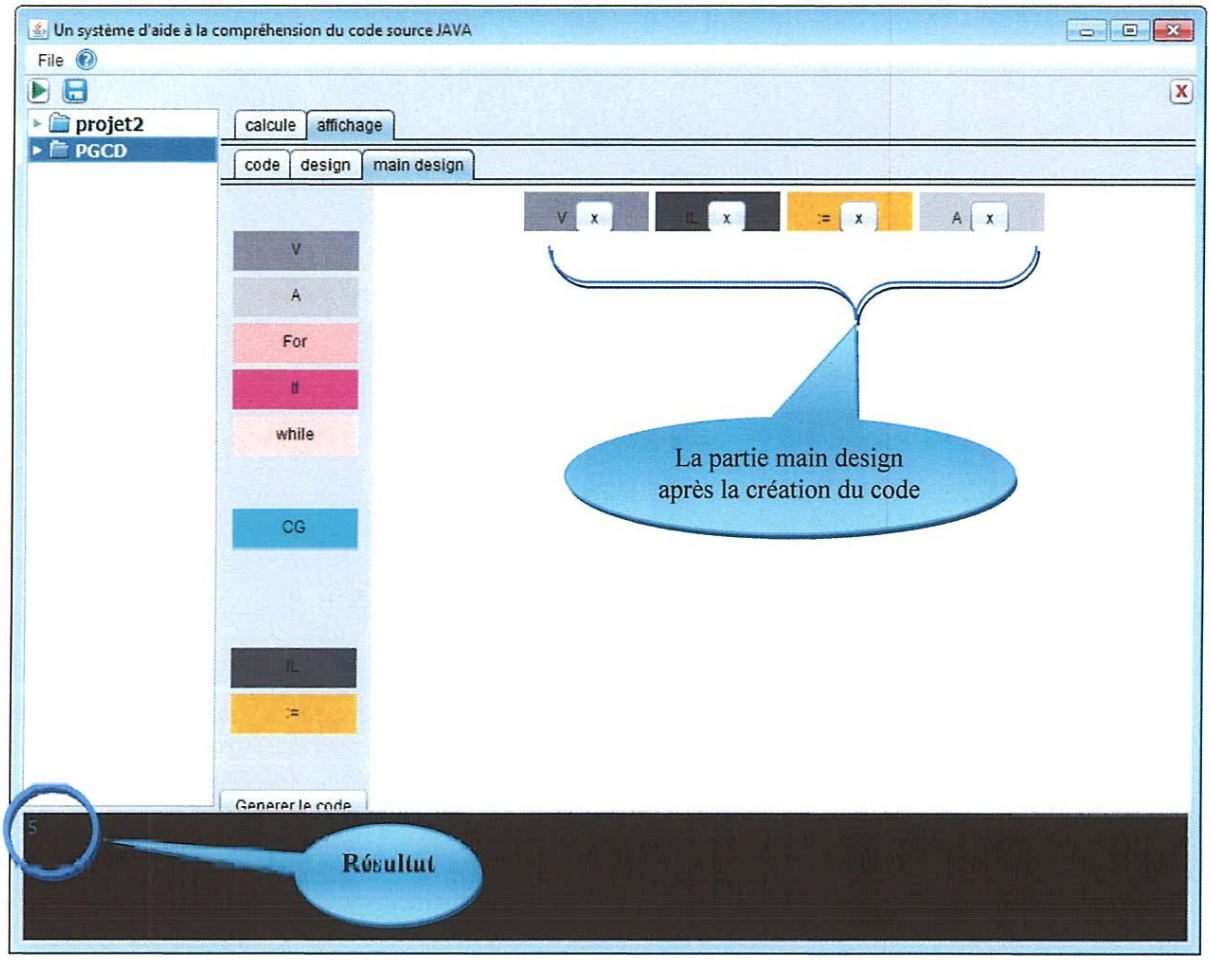
- Enfin, pour lancer l'exécution, il faut appuyer sur le bouton comme suit :



Si le projet n'a pas une classe *main*() par défaut , le système va demander quelle est la classe main() :



Le résultat s'affiche sur la console en-dessous :



2.6. Conclusion

Nous avons réalisé un premier prototype de l'application, et ses fonctionnalités couvrent les bases et la majorité des activités de l'environnement.

Nous avons effectué une première évaluation avec les enseignants, dont le but la validation du code source et les différentes fonctionnalités offertes par notre système. Les résultats étaient encourageants avec la satisfaction des enseignants.

Nous comptons à raffiner notre produit afin de couvrir d'autres besoins.

CONCLUSION GENERALE

La nécessité d'apprendre les langages orientés objets et surtout Java impose le développement d'une nouvelle approche de programmation qui réduit le plus maximum l'écriture classique du code source pour ce focaliser à la former des compétences nécessaires pour la création des codes de façon autonome.

Donc, l'objectif de notre travail était de concevoir et de réaliser un système d'aide à l'apprentissage du code source Java, Pour atteindre nos objectifs, nous avons entamé une étude théorique sur les systèmes d'apprentissage humain (Qu'est-ce qu'influence l'apprentissage et la compréhension du code source), et les bases de la programmation Orienté objet et Java. Ensuite, nous avons passé à la description détaillée de nos propositions.

Ces dernières sont basées sur la proposition d'un nouveau paradigme de programmation qui assiste les débutants programmeurs lors la création du code.

Pour tester l'apport du système développé, une petite expérimentation a été effectuée dans le département d'informatique avec les enseignants afin de valider la génération du code source et les différentes composantes du système. Les résultats montrent une satisfaction encourageante des enseignants.

PERSPECTIVES

Comme perspectives, nous proposons :

- Améliorer l'ergonomie du système.
- D'intégrer d'autres notions importantes de java (les exceptions, graphisme, polymorphisme…etc.).
- L'intégration de notre système dans les systèmes éducatifs (utiliser le système comme un environnement de programmation pour la formation des débutants programmeurs).
- Exploitation réelle du système.

Bibliographie

[1] Livre Program or be programmed.

[2] Eric Sigoillot, Publié le 25 juillet 2004 - Mis à jour le 10 septembre 2011

[3] Les bases du langage Java by Julien Sopena.

[4] Bertrand Florat,Java - notions élémentaires, Copyright (c) 2001

[5] LISIBILITÉ D'UN CODE SOURCE "Mewtow" 29 octobre 2015

[6] https://fr.wikipedia.org/wiki/MySQL_Workbench Dernière modification de cette page le 19 février 2016, à 02:03.

[7] https://fr.wikipedia.org/wiki/NetBeans Dernière modification de cette page le 11 mars 2016, à 23:45.

[8] Mme Dalila Taouri, Mi M.C Belaid « Introduction aux SI». Edition page bleu, P. 26

[9] chicoree.fr:
Travailler avec le BeanShell Dernière modification de cette page le 10 juin 2009 à 16:42.