République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur de la Recherche Scientifique

Université M'hamed Bougara – Boumerdes

Faculté des Sciences
Département d'Informatique

Domaine : Mathématiques Informatique
Filière : Informatique
Spécialité : Ingénierie du Logiciel et Traitement de l'Information

Master II
Mémoire de fin d'études

Thème

Développement d'un évaluateur d'états mémoire pour programmes en bytecode Java Card

Réalisé par :

Amzal Hanane Mokhtari Tarek

Soutenu le 02/07/2015 devant le jury composé de :

Mme. LOUNAS Razika	*Présidente*
M. MOKRANI Hocine	*Examinateur*
Mlle. HAMADOUCHE Samiya	*Promotrice*

Table des matières

Liste des figures

Liste des tableaux

Liste des abréviations

AID	Application IDentifier
APDU	Application Protocol Data Unit
API	Application Programming Interface
CAP	Converted APplet
Cap Map	Cap File Manipulator
FILO	First In Last Out
GC	Garbage collector
IDE	Integrated Development Environment
JBCD	Java ByteCode Debugger
JCAPI	Java Card API
JCRE	Java Card Runtime Environment
JCVM	Java Card Virtual Machine
JIT	Just In Time
JOE	Java Overall Editor
JVM	Java Virtual Machine
PC	Program Counter
SP	Stack Pointer

Introduction générale

La Technologie Java Card combine un sous-ensemble du langage de programmation Java avec un environnement d'exécution optimisé pour les cartes à puce et d'autres appareils à ressources réduites. L'objectif de la technologie Java Card est d'apporter de nombreux avantages du langage de programmation Java au monde de ressources limitées des cartes à puce.

Un programme source écrit en Java et représentant une applet peut être converti en un fichier binaire (fichier CAP) pour être utilisé sur une carte à puce de type Java Card (ou bien un simulateur de la carte). Le flux de traitement commence par la compilation du programme source (.class), ensuite sa transformation par le convertisseur en un fichier CAP afin de satisfaire les contraintes de ressources, puis son chargement sur la carte.

Une fois l'applet installée sur la plate-forme et sélectionnée, son bytecode est exécuté par la machine virtuelle embarquée JCVM (Java Card Virtual Machine).

Notre travail consiste à développer un outil qui simule, partiellement, le fonctionnement de cette machine. Il prend comme entrée un fichier en bytecode (représente une méthode extraite d'un fichier CAP) et fournit l'état mémoire de la pile d'opérandes et le tableau des variables locales, obtenu suite à l'interprétation de chaque instruction de ce dernier.

Le présent rapport se décline en quatre chapitres :

- **Chapitre 1** : va présenter la machine virtuelle Java, son rôle, sa structure et son fonctionnement.
- **Chapitre 2** : va présenter la machine virtuelle Java Card, son rôle, sa structure, son fonctionnement et la différence entre les deux machines virtuelles Java et Java Card.
- **Chapitre 3** : va exposer notre problématique et fixe l'approche que nous allons suivre pour la résoudre.
- **Chapitre 4** : va présenter la conception, les outils utilisés et les étapes suivies pour implémenter la solution avec ses différents versions.

Pour finir, une conclusion générale résume le travail fait avec quelques perspectives.

Chapitre 1

Java et la machine virtuelle Java

Sommaire

Ce chapitre présente le langage et la Machine Virtuelle Java. Il introduit le rôle de la machine virtuelle, décrit sa structure générale, retrace le cycle de vie des classes et explique son fonctionnement.

1.1 Le langage Java

Java est un langage de programmation orienté objet et une plate-forme informatique qui ont été créé par Sun Microsystems en 1995.

Ce langage est rapide, fiable, sécurise le code lors de la compilation et de l'exécution, il est nécessaire pour le fonctionnement de plusieurs applications, car il est basé sur un code **source** écrit en Java qui sera compilé avec **java compiler** en un fichier class contenant le **Bytecode** exécuté par la machine virtuelle java (**JVM**) en utilisant la machine (**Hardware**), comme le montre la figure ci-dessous (Figure 1.1) [26].

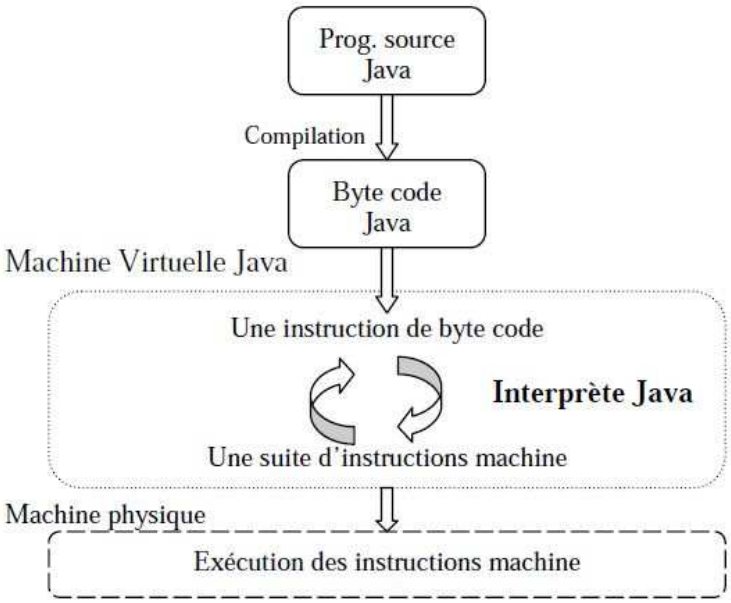


FIGURE 1.1 – Architecture globale de la plate-forme Java [3].

1.2 La machine virtuelle Java JVM

La Machine Virtuelle Java (Java Virtual Machine JVM) est l'environnement d'exécution pour les applications Java qui fournit une programmation indépendante de la plateforme car il garantit l'abstraction du matériel et du système d'exploitation, en utilisant un code compilé de taille réduite écrit en langage intermédiaire indépendant de tout système d'exploitation. Et qui permet aussi à un programme d'être exécuté de la même façon sur n'importe quelle plateforme (Figure 1.2) [10].

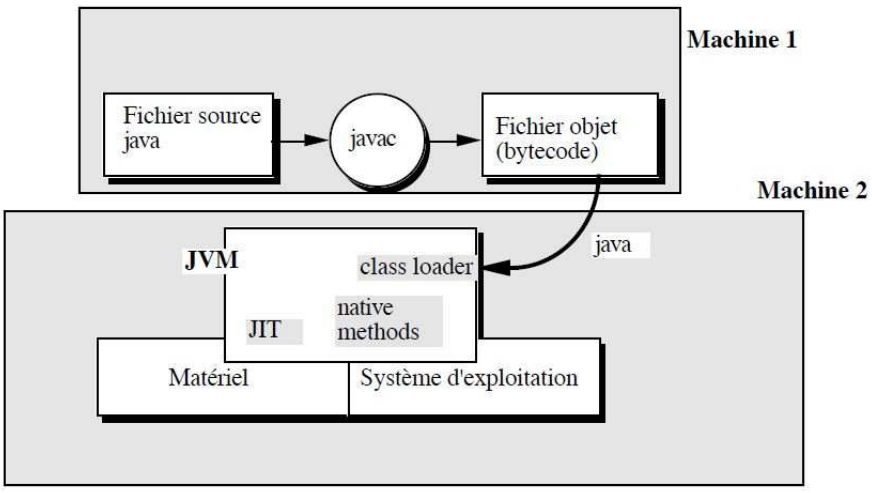


FIGURE 1.2 – L'indépendance de la plateforme de programmation et la plateforme d'exécution [13].

JVM est une machine de calcul abstraite, qui utilise un **ensemble d'instructions** écrites en bytecode et manipule plusieurs **espaces mémoires** pendant l'exécution de ces instructions.

Les fonctionnalités de la JVM sont décrites dans les spécifications de cette dernière mais se sont des fonctionnalités abstraites qui ne fournissent aucunes mise en œuvre particulière, comme par exemple : le chargement du fichier *.class*, le bytecode, la gestion du thread ,etc .Ces implémentations sont prises en charge par le fournisseur de la JVM. Les fournisseur plus connus : Sun Microsystems, IBM, BBA, etc [15].

1.2.1 La gestion de la mémoire

La gestion de la mémoire dans la JVM est représentée par Java Memory Model et qui propose une gestion simple basée sur deux règles [15]

— Pas d'allocation de mémoire explicite, mais seulement l'allocation de la mémoire requise à l'aide de l'opérateur new ;

— La libération de la mémoire des objets inutiles par le ramasse miette GC [1].

Les données dans JVM sont stockées dans des zones mémoires réparties en 2 catégories [15]

— Zone mémoire dont la durée de vie est égale à celle de la JVM, elle sera crée au lancement de la JVM et détruite à son arrêt ;

— Zone mémoire liée à un thread dont la durée de vie est égale à celle du thread concerné.

Plusieurs zones mémoires sont utilisées par JVM durant l'exécution (Figure 1.3) [10]

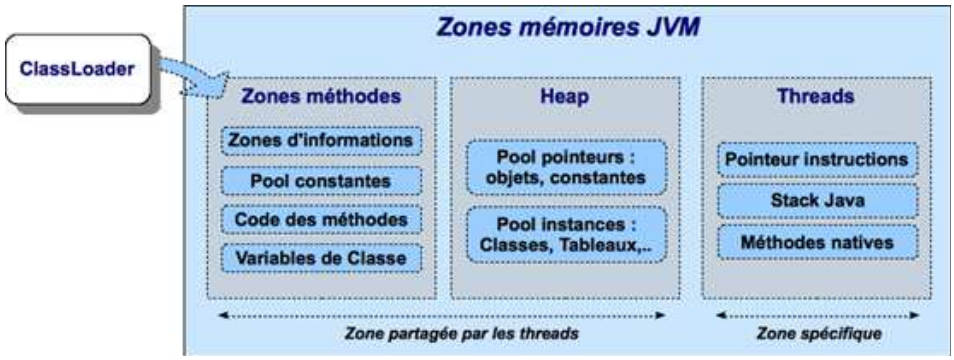


FIGURE 1.3 – Architecture de la mémoire de JVM [19].

1. **PC registre (Register)** : plusieurs threads peuvent être exécutée en même temps dans une JVM et chaqu'un dispose de son propre PC registre (*Program Counter*) qui contient l'adresse de l'instruction de la méthode qui est en cours d'exécution si la méthode exécutée par un thread n'est pas native [2] sinon le contenu de ce registre est indéfini.
2. **Une ou plusieurs pile (Stack)** : au cours de la création d'un thread, une pile d'opérandes privée sera créée et associée à celui-ci avec un pointeur SP (*Stack Pointer*) qui pointe sur le dernier élément inséré dans la pile, cette pile détient un cadre (Expliquer dans la suite) pour chaque méthode d'exécuter par ce thread et utilise la politique FILO pour First In Last Out. Une pile d'opérandes peut contenir : des variables locales, des résultats intermédiaires,les paramètres, les valeurs de retour de chaque méthode invoquée par le thread, etc.

1. Garbage Collector GC est un processus ou système chargé de libérer les espaces mémoire qui sont devenus inutilisable, en libérant les objets inutilisé qui ne sont référencé par aucun autre objet.
2. Une méthode native est une méthode quelle n'est pas implémenté en Java mais dans un autre langage de programmation (C, C++, assembleur...

3. **Un tas (Heap)** : un tas est un espace mémoire partagé par plusieurs threads qui stocke les objets (instances de classe) et les tableaux, et sa gestion est garantit par un mécanisme automatique implémenté dans JVM connu comme ramasse miette qui n'a pas un type particulier mais une seule technique sera choisit selon les besoin des fabricants. Heaps sont des espaces dynamiques et qui ne sont pas nécessairement adjacents.
4. **Une zone des méthodes (Method Area)** : une zone des méthodes est une zone mémoire partagée par les threads et qui contient une structure de chaque classe comme : constant pool de l'exécution [3], des données sur la méthode et son code, les constructeurs et les méthodes spéciales. Method Area est une zone dynamique contenant des cases mémoires qui ne sont pas nécessairement adjacentes.
5. **Pile de méthode native (Native Method Stack)** :
Certaines implémentations de la JVM utilisant des piles dynamiques nommée "C stacks" qui supportent des méthodes écrites dans un autre langage de programmation différent de Java. Native Method Stacks peuvent être utilisé dans l'implémentation d'un interpréteur JVM spécialisé pour des instructions écrite dans un autre langage (C par exemple).
6. **Cadres (Frames)** :
une frame est utilisée pour stocker des données, des résultats partiels, les valeurs de retour d'une méthode, les liens dynamiques et envoyer des exceptions. Une frame est créée dans la pile du thread à l'appel de la méthode et chaqu'une est divisée en 3 parties (Figure 1.4) :
   (a) ***Pile des opérandes*** qui contient les opérandes des instructions bytecode ;
   (b) ***Tableau des variables locales*** contient les variables locales d'une méthode ;
   (c) ***Référence au constant pool d'exécution*** qui contient une référence vers le constant pool d'exécution de la classe qui contient la méthode actuellement exécuter .

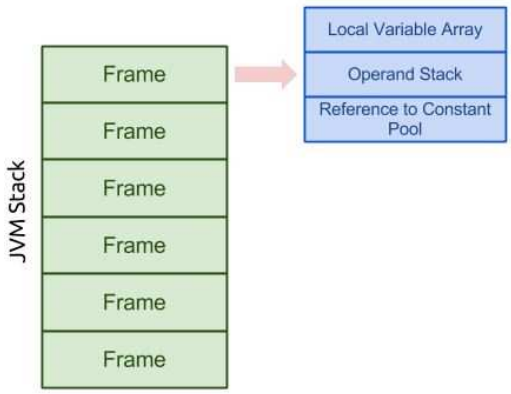


FIGURE 1.4 – Les composants d'une frame [27].

3. Run-time constant pool est une représentation instantanée de la table constant_pool qui contient plusieurs types de constantes créées à partir du code de la méthode gardée dans Method Area, ces constantes sont rangées pour être utilisé comme référence dans l'exécution.

1.2.2 Les types supportés par JVM

Le type des données qui sont traités par une JVM peuvent être divisé en 2 catégories selon le type de la valeur qui sera stockée dans des variables, passée en argument, ou retournée par une méthode [10].

Ils sont classés comme suit :

1. **Type simple**
   (a) **Type numérique**
   — **Entier**(*integer*)

Type	Codage	Valeur par default
byte	8-bit signé	zéro
short	16-bit signé	zéro
int	32-bit signé	zéro
long	64-bit signé	zéro
char	16-bit non signé(Unicode)	null ('ŏ000')

TABLEAU 1.1 – Tableau des entiers.

   — **Réel** (*Float*)

Type	Codage	Valeur par default
float	32-bit simple précision	zéro positive
double	64-bit double précision	zéro positive

TABLEAU 1.2 – Tableau des réels.

   (b) **Type booléen** (*boolean*) : les valeurs prise par ce type sont True ou False et la valeur par default est False.
   (c) **Adresse de retour** (*returnAdress*) : les valeurs de ce types sont des pointeurs sur des instructions de la JVM.
2. **Type de référence** : représente les éléments qui seront créés dynamiquement et leur valeur par default est null.
   (a) **Type class** : représente une classe ;
   (b) **Type array** : représente un tableau ;
   (c) **Type interface** : représente une interface.

1.2.3 Le cycle de vie d'une classe

Chaque Machine Virtuelle spécialement dans la zone de méthode, il existe un système de chargement de classe qui contient un ensemble de chargeurs de classes (*classloader*) [8].

Au sein de la VM, une classe suit un cycle de vie précis définit comme suit [8] :

1. **Chargement de la classe** : les programmes java sont un ensemble de classes, chaqu'une génère un fichier compilé qui sera chargé dynamiquement. Pour la première création d'une instance d'une classe, le classloader recherche le fichier, charge les classes héritées, réalise l'édition des liens et vérifie la syntaxe et la sémantique des fichiers chargés.
2. **Liaison** : appelé aussi la résolution, elle consiste à remplacer toutes les références symboliques de la tables de constantes en références mémoires après le chargement du code des méthode de la classe dans la mémoire et établir les liens entre eux.

3. **Initialisation** : après l'établissement des liens, les variables de la classe seront initialisées selon les valeurs précises dans le code source de cette classe, la VM alloue de l'espace dans le tas pour la création et l'initialisation de l'instance associée à la classe.
4. **Instanciation** : après l'initialisation de la classe, un objet sera instancié de celle-ci à l'aide du constructeur [4] associé à cette classe, qui sera appelé automatiquement lors de l'utilisation de l'opérateur "new", ce constructeur va initialisé les champs de cet objet, et l'objet même sera contenu dans le tas pour commencer le traitement.
5. **Retrait** : c'est la fin du cycle de vie de la classe, où le ramasse miette GC sépare les objets qui ont encore une référence (objet vivant) avec celle qui ne possède pas de référence (objet inutilisé), quand l'application ne fait rien. Puis le GC va appelé la méthode *finalize()* définit dans la classe de l'objet qui permet de nettoyer les ressources utilisés par les objets inutilisés afin de libérer l'espace mémoire occupé par ces objets. Après la libération de tous les objets de la classe, cette dernière sera déchargée.

1.2.4 Les threads dans Java

Lors du démarrage de la machine virtuelle, un seul thread exécute le code main, et durant la vie de la machine plusieurs threads peuvent être exécutés séparément en meme temps, manipulant des valeurs et des objets Java qui se trouvent dans des zones mémoires partagées [3].

Chaque thread est associée à une pile java qui contient des frames, chaque frame est associé à la méthode appelée par le thread [3].

1.2.5 Le fichier .class et le bytecode

Le fichier .class est un fichier généré par le compilateur java à partir d'un fichier source écrit en java puis le chargeur de classe (Class Loader) charge le bytecodes dans les zones de données d'exécution et le moteur d'exécution l'exécute, comme le montre la figure 1.5 [12].

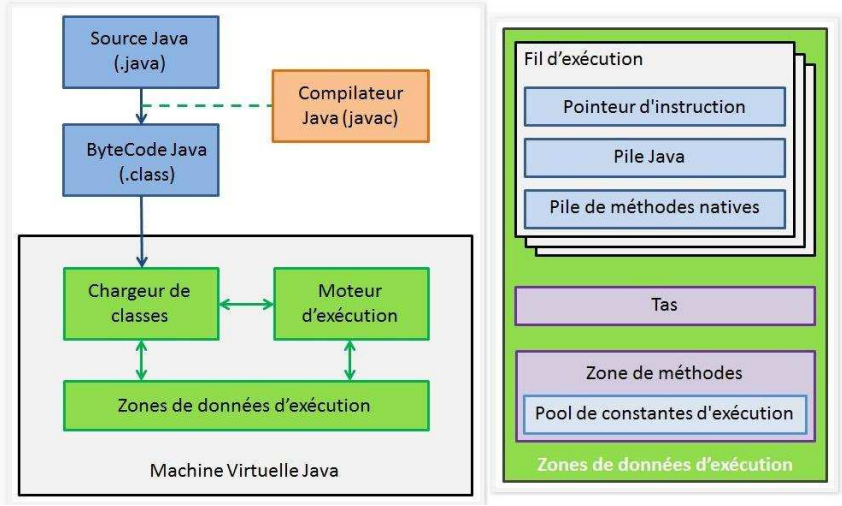


FIGURE 1.5 – Le processus d'exécution d'un code Java et les zones de données d'exécution [12].

4. Un constructeur est une méthode particulière d'une classe qui porte le même nom de la classe dans laquelle se trouve. Le constructeur est appelé automatiquement lors de la création d'une instance de classe afin d'initialiser les champs de l'objet créé. L'instanciation se fait dynamiquement grâce à l'opérateur new.

Chaque fichier .class contient une définition d'une seule classe ou interface, il est représenté comme flux de 8-bit Byte et les 16-bits, 32 bits et 64-bits sont construit en lisant 2, 4 et 8 bytes (8-bits) respectivement. Cette écriture représente le langage intermédiaire entre le code source et le code machine : le bytecode.

Le fichier .class sera compilé dans la JVM par un traducteur par exemple JIT pour Just-in-time [5] en un ensemble instructions spécifique au CPU, chaque instruction à compilé doit être sous la forme suivante [10] :

```
<index> <opcode> [ <operand1> [ <operand2>... ]] [<comment>]
```

FIGURE 1.6 – La forme d'une instruction du bytecode [10].

Ou *<index>* est l'offset de instruction, *<opcode>* est une mnémonique, *<operandN>* sont les opérandes de l'instruction (Une ou plusieurs) et *<comment>* représente un commentaire sur l'instruction, par exemple :

```
8   bipush 100     // Push int constant 100
```

FIGURE 1.7 – Exemple d'une instruction du bytecode [10].

1.2.5.1 Mnémonique

Une mnémonique est une forme textuelle simplifiée, lisible par les humains, représentant une opération (addition, charge, stocke, etc.). Chaque mnémonique correspond à un nombre entre 0 et 255 dans un fichier .class. Ce nombre est appelé le code d'opération (opcode) [10].

En bytecode, tout comme en assembleur, une instruction effectue des opérations atomiques. En d'autres termes, il est généralement nécessaire d'avoir plusieurs instructions pour effectuer une opération prenant une ligne en Java (ou tout autre langage de haut niveau) [10].

Les instructions en bytecode sont présentées uniquement dans des méthodes (Figure 1.8).

```
void sspin() {
    short i;
    for (i = 0; i < 100; i++) {
        ;    // Loop body is empty
    }
}
```

```
Method void sspin()
0   iconst_0
1   istore_1
2   goto 10
5   iload_1        // The short is treated as though an int
6   iconst_1
7   iadd
8   i2s            // Truncate int to short
9   istore_1
10  iload_1
11  bipush 100
13  if_icmplt 5
16  return
```

FIGURE 1.8 – Exemple d'une méthode convertit vers le bytecode [10].

5. JIT est un compilateur qui traduit le bytecode en code machine pour le processeur de la machine où la JVM est lancé, cela permet d'améliorer les performances des applications java exécutés. Il est appelé "Juste à temps" car il intervient juste avant le premier appel de chaque méthode.

1.2.5.2 Opcode

Il existe plusieurs opcodes réservés, il y on a environ 200 opcodes chaqu'un a sa fonctionnalité [10].

Le tableau 1.4 montre les opcodes les plus utilisés avec l'action associée à chaque opcode, il est représenté sous forme d'un descripteur (T) qui reflète le type des données et l'opcode [10].

Le tableau 1.3 montre les descripteurs possibles avec leurs type associé.

Descripteur	**Type**
b	Byte
s	Short
i	Int
l	Long
f	Float
d	Double
c	Char
a	Reference

TABLEAU 1.3 – Tableau des descripteurs possibles [10].

Opcode	**Signification**
Tipush <n>	Charger une constante de type T dans la pile des opérandes (Empiler)
Tconst_<n>	Charger une constante n de type T dans la pile des opérandes (Empiler)
Tload_<n>, Tload <n>	Charger une variable local de type T dans la pile des opérandes à l'index n(Empiler)
Tstore_<n>, Tstore <n>	Stocker une valeur de type T qui existe au sommet de la pile des opérandes dans la liste des variables locale à l'index n (Dépiler)
Tinc	Incrémentation d'une valeur de type T
Taload	Charger un tableau de type T dans la pile des opérandes (Empiler)
Tastore	Stocker un tableau de type T de la pile des opérandes dans la liste des variables locales
Tadd	Addition de type T avec chargement du résultat dans la pile des opérandes
Tsub	Soustraction de type T avec chargement du résultat dans la pile des opérandes
Tmul	Multiplication de type T avec chargement du résultat dans la pile des opérandes
Tdiv	Division de type T avec chargement du résultat dans la pile des opérandes
Trem	Reste de la division du type T avec chargement du résultat dans la pile des opérandes
Tneg	Inverser le signe d'une valeur de type T
Tshl	Décalage arithmétique de bit vers la gauche pour une valeur de type T
Tshr	Décalage arithmétique de bit vers la droite pour une valeur de type T
Tushr	Décalage logique de bit vers la droite pour une valeur de type T
Tand	ET logique entre deux valeur de type T
Tor	OU logique entre deux valeur de type T

Txor	OU Exclusif entre deux valeur de type T
i2T	Conversion de type depuis int au type T
l2T	Conversion de type depuis long au type T
f2T	Conversion de type depuis float au type T
d2T	Conversion de type depuis double au type T
Tcmp	Comparaison entre les entiers
Tcmpl	Comparaison entre les réels
Tcmpg	Comparaison entre les réels
if_TcmpOP	Branche conditionnelle, saut si la fonction OP entre deux valeurs de type T est satisfaite
Treturn	Retourne à l'appelant une valeur de type T et l'ajouter au sommet de la pile (Empiler)

TABLEAU 1.4 – Tableau des opcodes fréquents [10].

Chaque opcode sera convertit en binaire (Figure 1.9), ils existent des opcodes pour représenter les constantes, les opérations sur les zones mémoires (load, store, stack), sur les types (fonctions mathématiques, conversions, comparaisons), de références, de contrôles, des opérations améliorés et d'autres réservés [10].

```
// Bytecode stream: 03 3b 84 00 01 1a 05 68 3b a7 ff f9
// Disassembly:
iconst_0      // 03
istore_0      // 3b
iinc 0, 1     // 84 00 01
iload_0       // 1a
iconst_2      // 05
imul          // 68
istore_0      // 3b
goto -7       // a7 ff f9
```

FIGURE 1.9 – Exemple d'une suite binaire et sa correspondance en opcode [10].

1.2.5.3 Exemple pratique

Pour mieux illustrer les composants de la mémoire de la JVM et leurs fonctionnements, on peut voir les changements portés sur les zones mémoires essentiellement sur le contenu d'un cadre en cours d'exécution, par exemple de la méthode paire() qui retourne 1 si la valeur donnée en entrée est paire sinon elle retourne 0, l'exemple suivant (Figure 1.10) prend en entrée la valeur 2 (n=2) sauvegardée dans la liste des variables locales à l'indice 0 :

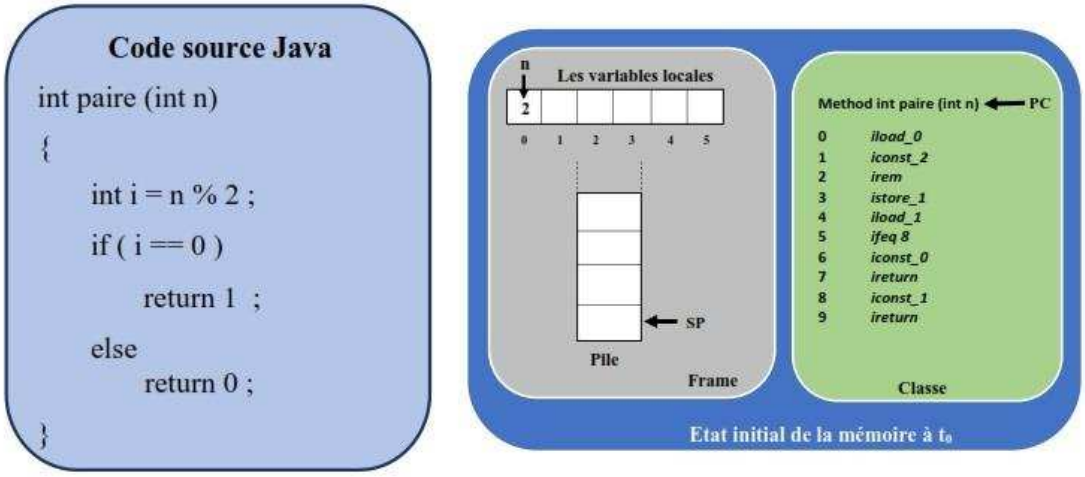


FIGURE 1.10 – Le code source de la méthode paire() et l'état initial de la mémoire.

La figure 1.11 présente les différents états que les zones mémoires peuvent prendre pendant l'exécution de la méthode paire().

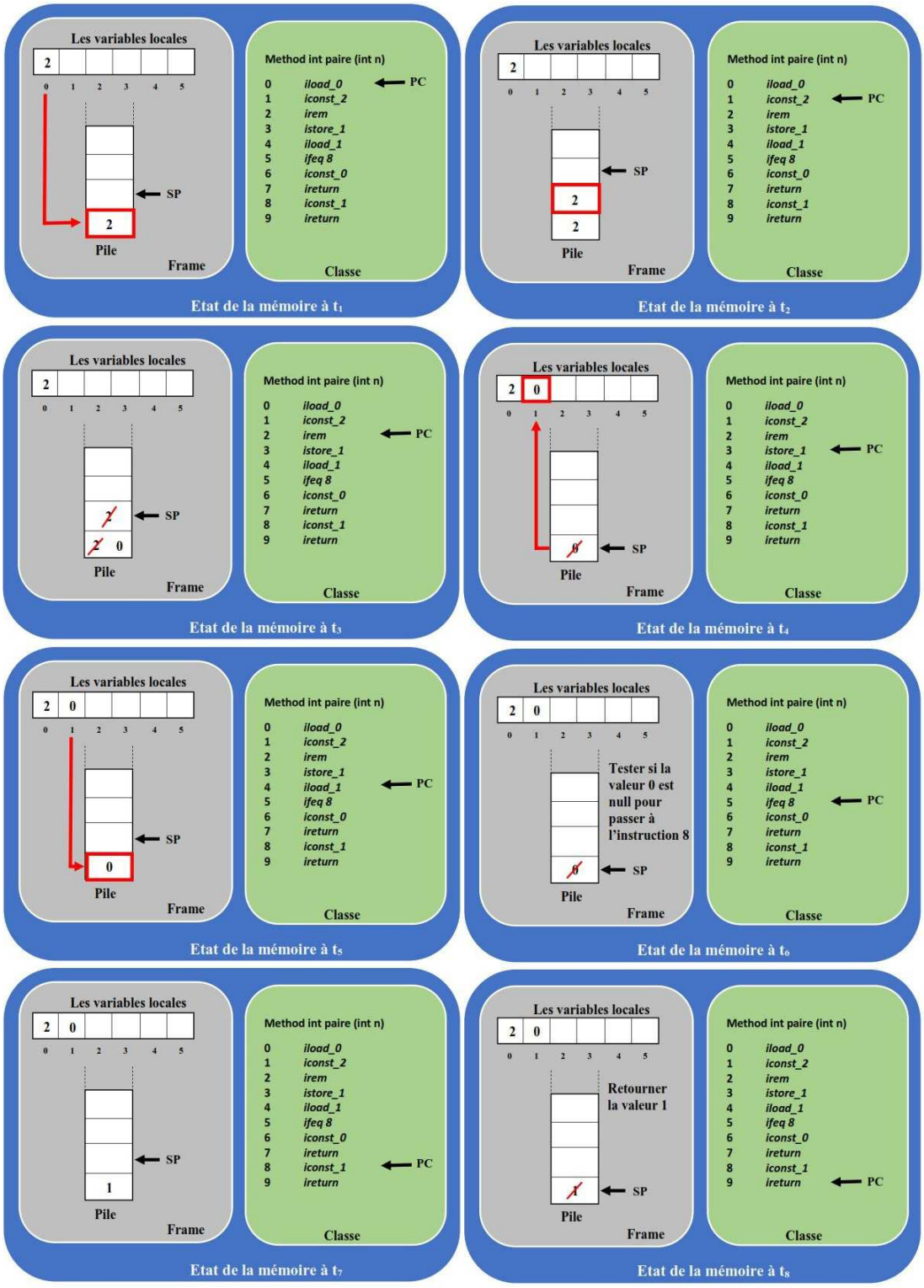


FIGURE 1.11 – Les différents états mémoire pendant l'exécution de la méthode paire().

1.3 Conclusion

Nous pouvons conclure que le langage Java est un langage de programmation informatique orienté objet ayant pour particularité et objectif central la portabilité des logiciels écrits dans ce langage. Cette portabilité est assurée par la machine virtuelle Java. Cette dernière interprète le bytecode résultant de la compilation d'un programme Java.

Chapitre 2

Java Card et la machine virtuelle Java Card

Sommaire

Ce chapitre présente le langage et la Machine Virtuelle Java Card. Il introduit le rôle de la machine virtuelle, décrit sa structure générale, retrace le cycle de vie des applets Java Card et explique son fonctionnement.

2.1 Carte à puce

Une carte à puce est un support électronique, portable et sécurisé, capable de conserver des données personnelles [9]. C'est une carte plastique de taille limitée, incorporant un circuit électronique de taille réduite qui contient un microprocesseur ou des circuits de mémorisation non volatile (Figure 2.1) [9].

FIGURE 2.1 – Une carte à puce [9].

2.2 Java Card

La technologie Java card est une adaptation de la plateforme java pour faire fonctionner java sur des équipements fortement contraints tels que les cartes à puce et les dispositifs de ressources limitées en mémoire et en processeur. Cette technologie préserve plusieurs bénéfices du langage Java.[9].

Java card est un environnement d'exécution pour des applications java spécifiques pour des cartes à puce qui a une structure qui ne supporte que les **applets** comme modèle d'application, qui seront exécutée par l'environnement d'exécution Java Card **(JCRE** [1] **)** contenant les **API** [2] et **Java Card Virtual Machine JCVM**, en utilisant le **système d'exploitation** et les **composantes électroniques** (Figure 2.2) [9].

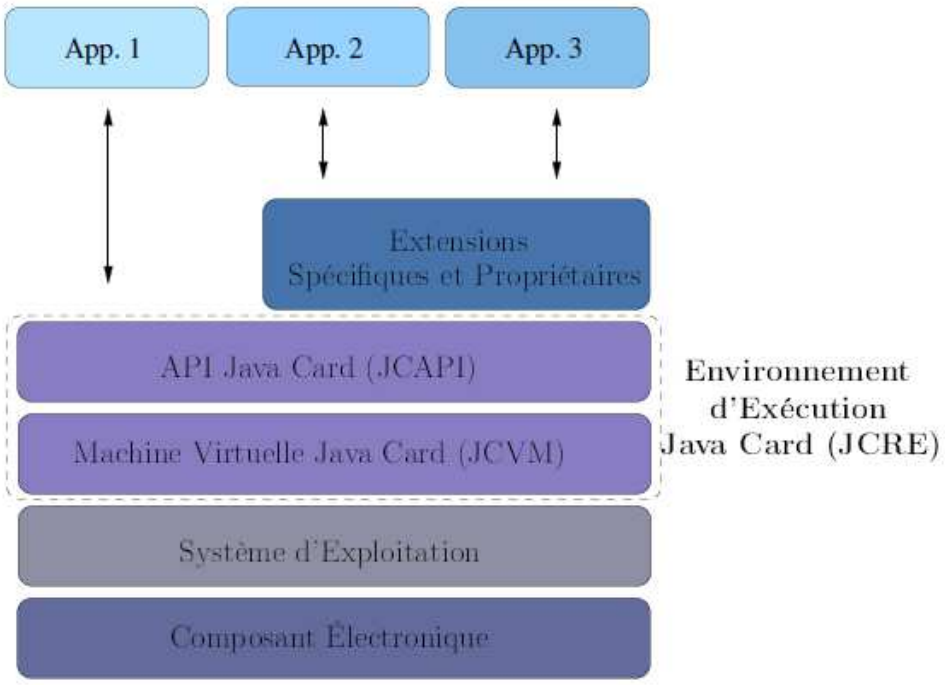


FIGURE 2.2 – Architecture globale de Java Card [9].

1. JCRE (*Java Card Runtime Environment*) : précise le comportement de l'exécution de la Java Card.
2. API (*Application Programming Interface*) : est un ensemble de classes et paquets nécessaire pour la programmation des cartes à puce et aussi quelques extensions optionnelles.

2.3 La machine virtuelle Java Card JCVM

La machine virtuelle Java Card (Java Card Virtual Machine JCVM) est l'environnement d'exécution des applications Java Card dans des cartes à puce, elle est similaire à la JVM mais voyant les ressources limitées dans la carte, cette machine virtuelle est divisée en 2 parties (Figure 2.3) [6]

- **Une partie hors carte (Off-Card)** : c'est la partie logée sur une station de travail ou ordinateur, qui contient un vérificateur du code et un convertisseur pour convertir le fichier Class en un fichier CAP (Converted APplet).
- **Une partie sur carte (On-Card)** : c'est la partie embarquée dans la carte, elle contient un interpréteur pour exécuter le bytecode existant dans le fichier .cap.

Ces 2 parties ensemble fournissent toutes les fonctionnalités d'une machine virtuelle [6].

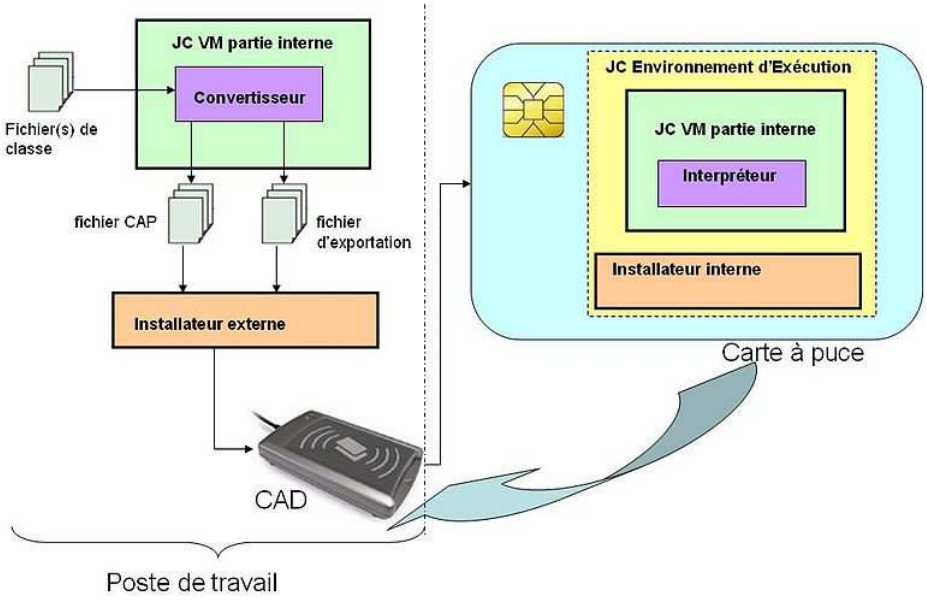


FIGURE 2.3 – La machine virtuelle Java Card.

2.3.1 La gestion de la mémoire

Les zones mémoires dans JCVM ont généralement la même architecture que celle de JVM, avec quelque spécialisation :

- Un seul thread peut être exécuté en même temps.
- Pas de ramasse miette.
- Pas de pile pour les méthodes natives.

2.3.2 Les types supportés par JCVM

Comme le Java Card est un sous ensemble de Java, alors la JCVM supporte seulement un sous ensembles des types supportés par la JVM,mais globalement supporte les mêmes types : le type simple et le type de référence [10].

1. **Type simple**
   (a) **Type numérique**

Type	Codage	Valeur par default
byte	8-bit signé	zéro
short	16-bit signé	zéro
int	32-bit signé	zéro

TABLEAU 2.1 – Tableau des types numériques.

   (b) **Type booléen** *(boolean)* : les valeurs prise par ce type est 1 pour true et 0 pour false et la valeur par défault est 0.
   (c) **Type de retour** *(returnAdress)* : les valeurs de ce types sont des pointeurs sur des instructions de la JCVM.
2. **Type de référence** : représente les éléments qui seront créer dynamiquement et leur valeurs par default est null.
   (a) **Type class** : représente une classe ;
   (b) **Type array** : représente un tableau à une seule dimension ;
   (c) **Type interface** : représente une interface.

— La machine virtuelle Java Card définit une unité élémentaire pour représenter tous ses types, mais il n'y a pas une définition claire sur le nombre de bit qui contient cette unité qui est appelé **Word**, l'essentiel que un word peut présenté simultanément une valeur prise par les types byte, short, reference et addresse de retour, et deux word peut présenté un int [10].

2.3.3 Le cycle de vie d'une applet Java Card

Une applet est l'application qui sera exécutée dans une carte carte à puce en interagissant avec le JCRE. Plusieurs peuvent être gardées dans une carte mais une seule peut être active à la fois. Chaque applet est identifiée par un identifiant unique AID (*Application IDentifier*) [6].

Pour qu'une applet puisse être exécutée, il est nécessaire de définir les méthodes suivantes :
— *install/unstall* : afin d'installer ou désinstaller une applet de la carte.
— *register* : afin d'enregistrer l'applet dans le JCRE.
— *select/deselect* : afin d'activer ou désactiver une applet.
— *process* : afin de traiter ou envoyer une commande APDU.

Quand une applet est chargée dans la carte, elle n'est pas accessible tant qu'elle n'est pas installée, donc le cycle de vie d'une applet (Figure 2.4) commence par la création d'une instance avec la méthode *install()* et par l'enregistrement de celle-ci au sein du JCRE par la méthode *register()*, mais cela rend l'applet inactive jusqu'à ce qu'elle soit séléctionnée via une commande SELECT APDU envoyée au JCRE [6].

Ce dernier consulte sa table interne pour trouver l'applet de l'AID correspond à la commande, puis la méthode *select()* sélectionne l'applet demandée et le JCRE fait suivre toutes les

commandes APDU à la méthode *process()* où chaque commande APDU est interprétée pour exécuter la tâche qu'elle spécifie [6].

Pour chaque commande APDU, l'applet répond en envoyant une réponse APDU informant du résultat du traitement de la commande. Ce dialogue commande-réponse continue jusqu'à où l'applet sera inactive en utilisant la méthode *deselect()* ou la carte sera retirée du lecteur. Lorsque le JCRE reçoit une demande d'effacement d'instance de l'applet et/ou du paquetage auquel elle appartient avec la méthode *delete()* le cycle de vie de l'applet se termine et s'il n'y a pas une telle demande, la fin du cycle de vie de l'applet est celle de la carte [6].

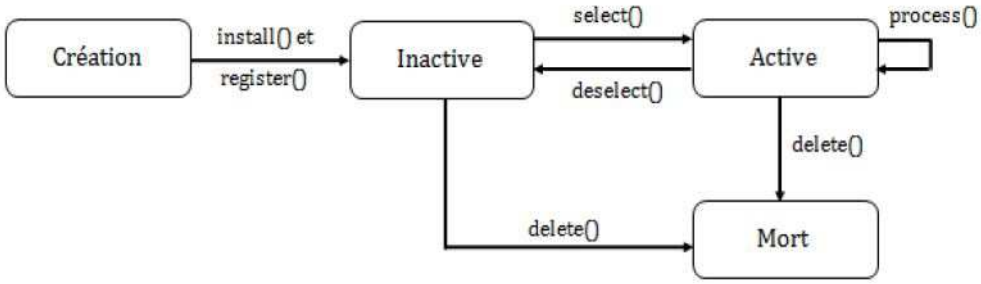


FIGURE 2.4 – Cycle de vie d'une applet Java Card [6].

2.3.4 Les threads dans Java Card

Contrairement aux threads Java, dans Java Card seulement un seul thread sera exécuté dans la JCVM durant la vie de la machine et le programme Java Card ne peut pas utiliser la classe thread ou n'importe quelle mot clé lié à la manipulation des threads [10].

2.3.5 Le fichier CAP

Le Fichier CAP (**Converted APplet**) est le résultat de la conversion du fichier Class et qui sera chargé dans la carte car il représente un format plus simple à exécuter pour une plateforme disposant de peu de ressources. Le format du fichier CAP repose sur la notion de composants. Il est spécifié dans [11] comme étant constitué de 12 composants standards. Chacun de ses composants contient des informations spécifiques obtenues à partir du package définissant le fichier CAP correspondant. De plus, les composants possèdent des liens d'interdépendances entre eux [7].

Un fichier CAP consiste en un flux de bytes de 8 bits. Toutes les autres quantités de 16 bits et 32 bit sont respectivement construites par la lecture de deux et quatre bytes consécutifs. Les objets multi-bytes sont stockés selon une orientation big-endian [3] [11].

2.3.5.1 Format général

```
component {
  u1 tag    ⟶ indique le type du composant.
  u2 size   ⟶ indique le nombre d'octets dans info[].
  u4 info[] ⟶ Varie selon le type du composant.
}
```

FIGURE 2.5 – Format général d'un composant du fichier CAP [6].

3. L'octet de poids le plus fort est enregistré à l'adresse mémoire la plus petite.

2.3.5.2 Les composants du fichier CAP

Le tableau 2.2 ci-dessous résume la description des 12 composants contenus dans un fichier CAP tel que présenté dans [11].

Tag	Composant	Contenu
1	Header	- Regroupe les informations générales sur le fichier CAP et le package qui est défini.
2	Directory	- Liste la taille de chaque composant défini dans le fichier CAP. - Contient aussi des entrées vers les nouveaux composants ajoutés (*custum components*).
3	Applet (optionnel)	- Contient une entrée pour chaque applet contenue dans le fichier CAP. - Si aucune applet n'est définie dans le package, ce composant ne sera pas présent dans le fichier CAP.
4	Import	- Liste l'ensemble des packages importés par les classes du package défini (ce dernier ne figure pas dans la liste).
5	ConstantPool	- Contient une entrée pour chaque : classe, méthode et champ référencé par les éléments du composant *Method* dans le fichier CAP.
6	Class	- Décrit chacune des classes et interfaces définies dans le package. - Les classes représentées dans le composant *class* référencent d'autres entrées dans le même composant sous forme de : Superclass, Superinterface, références des interfaces implémentées. - Les classes représentées dans ce composant contiennent aussi des références des méthodes virtuelles définies dans le composant *Method* du fichier CAP.
7	Method	- Décrit : chaque méthode déclarée dans le package, les méthodes abstraites définies par les classes, les handlers d'exception associés à chaque méthode.
8	StaticField	- Contient toutes les informations requises pour créer et initialiser une image de tous les champs statiques définis dans le package.
9	ReferenceLocation	- Représente la liste des décalages dans le composant *Method* vers des éléments qui ont des points d'entrée dans le composant *Constant Pool*.
10	Export (optionnel)	- Liste tous les éléments statiques qui peuvent être importés par des classes dans d'autres packages.
11	Descriptor	- Donne les informations nécessaires pour analyser et vérifier tous les éléments du fichier CAP.
12	Debug (optionnel)	- Contient toutes les méta-données nécessaires pour débuger un package sur une JCVM [4] appropriée (ce comosant n'est pas chargé sur la carte).

TABLEAU 2.2 – Descriptions des composants du fichier CAP [6].

2.3.5.3 Liens d'interdépendance

En analysant la structure de chaque composant du fichier CAP telle que définie dans la spécification Java Card [11], il a été constaté par l'auteur de [6] que les composants se référencent entre eux (i.e. il y a de l'information partagée). La figure 2.6 résume l'ensemble des liens d'interdépendance identifiés.

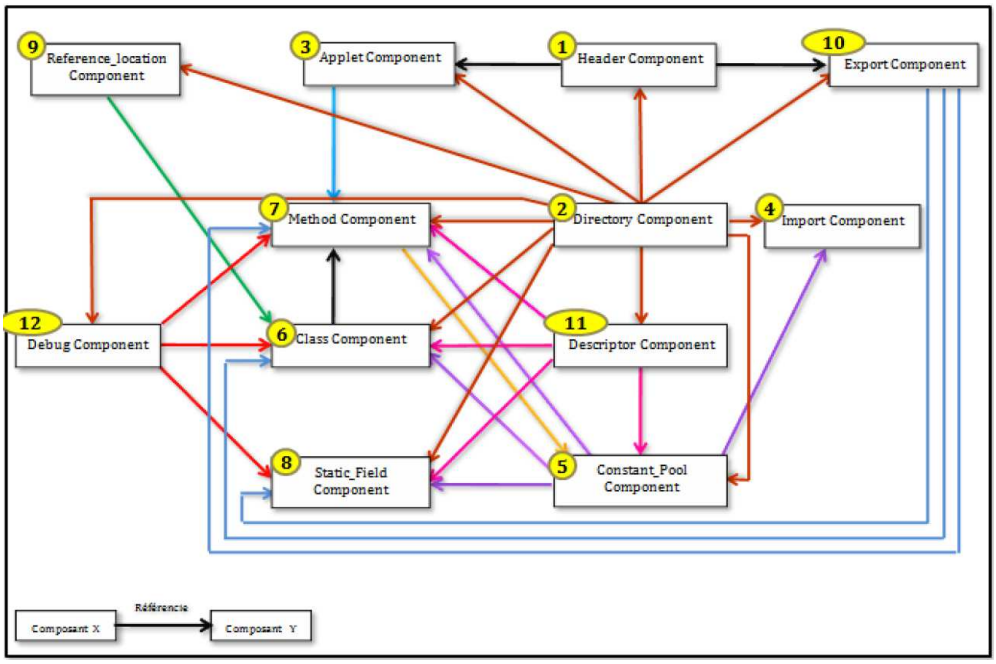


FIGURE 2.6 – Liens d'interdépendance entre les composants du fichier CAP [6].

2.3.6 Bytecode Java Card

Après le chargement des bibliothèques nécessaire et le code source (.java) le compilateur Java produit le fichiers class(.class), après ce fichier sera vérifié et convertit par Java Card Converter en un fichier .cap (Converted APplet) de taille réduite qui sera chargé sur la carte pour une interprétation ultérieur (Figure 2.7). D'où la base du principe du bytecode "Write once, run anywhere" [3'

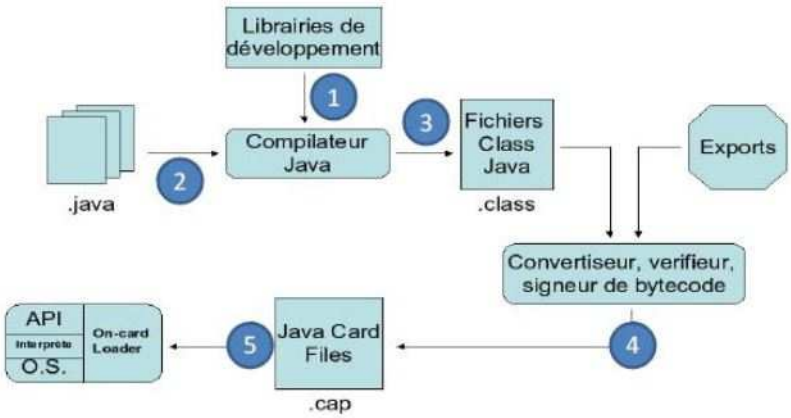


FIGURE 2.7 – Le processus d'exécution d'un code Java Card [2].

A cause des ressources limitées dans les cartes à puce et pour faire fasse à cette contrainte, le langage Java Card est né avec un jeu réduit d'instructions bytecode. Le bytecode Java Card est un sous ensemble des bytecodes Java avec quelques instructions en plus.

Les instructions Java Card ont un format similaire à celle des instructions Java qui sont aussi présentes seulement dans des méthodes, et on a exactement 186 opcodes réservés donc 186 instructions possibles. Ces opcodes peuvent être classés selon les changements apportés au zones mémoires [10] (Figures 2.8 et 2.9).

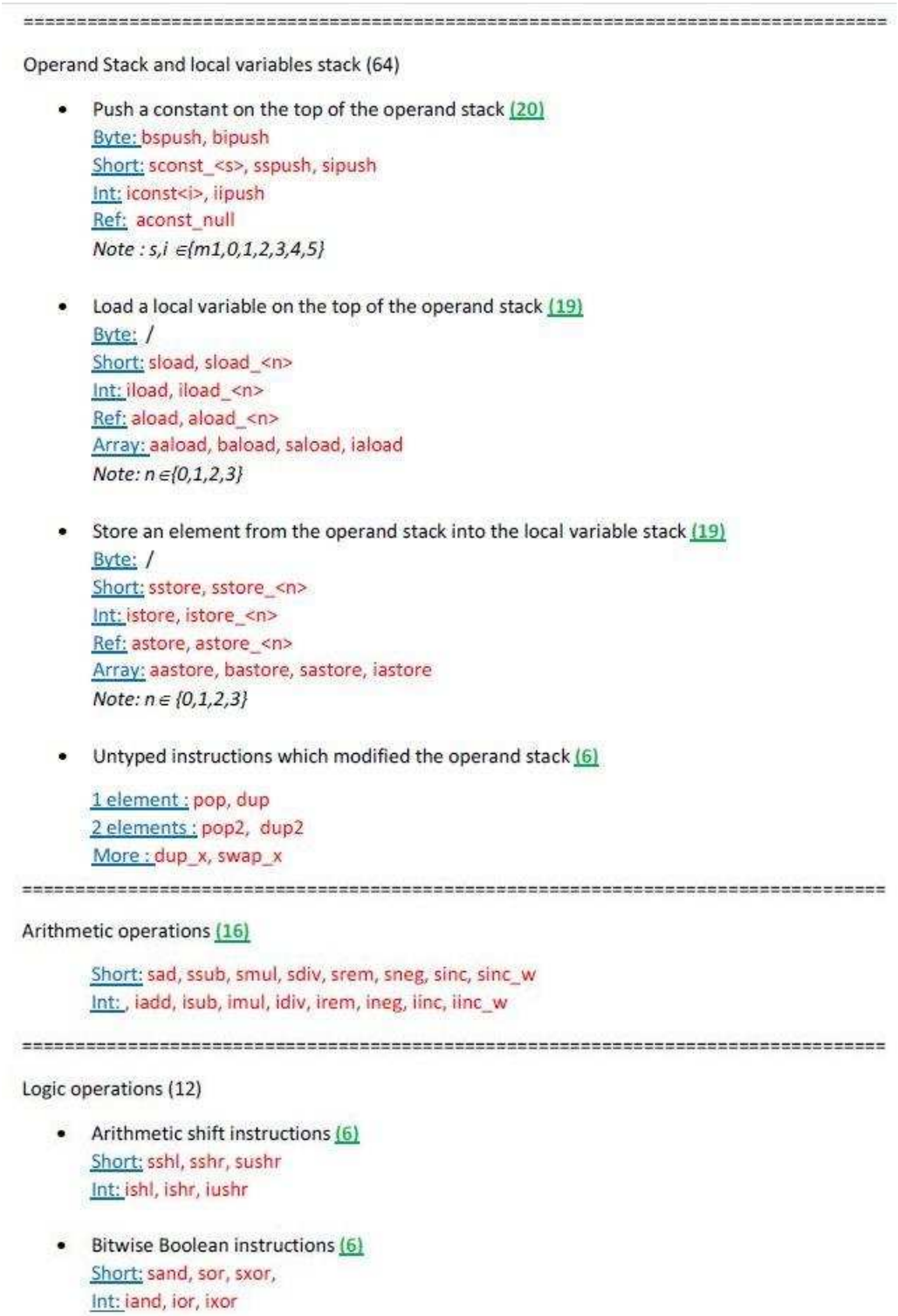


FIGURE 2.8 – Catégorisation du bytecode Java Card (1/2).

FIGURE 2.9 – Catégorisation du bytecode Java Card (2/2).

2.4 Java vs Java Card

Comme Java Card est un sous langage de Java qui contient en plus quelques instructions pour le développement sur la carte, on peut citer la différences entre Java et Java Card, en présentant les éléments supportés par chaque langage dans le tableau suivant (Tableau 2.2) :

Éléments	**Java**	**Java Card**
Types de données	Tous les types sont supportés	Seulement les types : byte, short, int, boolean, adresse de retour et reference sont supportés
Tableaux	Multi-dimensionnels	Mono-dimensionnels
Thread	Multi-threading	Mono-threading
Classloader	plusieurs classloaders	Pas de classloader, une seule classe sera exécutée
Composantes mémoire de la machine virtuelle	Toutes les composantes sont présentes	Toutes sauf le ramasse miette et la pile des méthodes natives
Bytecode	205 instructions	186 instructions

TABLEAU 2.2 – Tableau des différences entre Java et Java Card.

2.5 Conclusion

Nous pouvons conclure que Java Card est l'adaptation de Java pour des dispositifs à petite mémoire, notamment les cartes à puces, et que le fichier CAP est le résultat de la conversion du fichier Class et qui sera chargé dans la carte car il représente un format plus simple à exécuter pour une plateforme disposant de peu de ressources.

Chapitre 3

Évaluateur d'état mémoire

Sommaire

Ce chapitre présente la problématique traitée dans ce projet, avec quelques outils existant qui traitent des problématiques similaire .

3.1 Problématique

L'objectif principal de notre travail consiste à développer un évaluateur qui prendra en entrée le bytecode d'une méthode et rendra comme sortie l'évolution de l'état des zones mémoires de la machine virtuelle Java Card après l'interprétation de chaque instruction. Toutefois, on se limite à la pile d'opérandes et le tableau des variables locales comme zones mémoire à prendre en considération dans le présent projet.

3.2 Démarche à suivre

Nous avons opté pour la division de la problématique en trois sous problèmes. Notre démarche suivra alors les trois étapes suivantes :

1. Nous allons commencer par l'implémentation du noyau de notre interpréteur, qui prendra comme entrée le bytecode de la méthode à interpréter. Et fournira en sortie l'état de la mémoire après l'exécution de chaque instruction. Donc, nous nous intéresserons dans cette étape à l'analyse lexicale et syntaxique du bytecode, et à l'implémentation de la sémantique des instructions.

2. En deuxième lieu nous allons ajouter la faculté d'extraire le bytecode des méthodes directement depuis un fichier CAP.
3. La troisième étape est d'implémenter une interface graphique qui permettra de visualiser clairement les résultats fournis par l'interpréteur.

Pour ce faire, Nous utiliserons le langage de programmation Java pour le développement de l'outil.

3.3 Travaux connexes

D'après nos recherches bibliographiques, nous avons constaté qu'il n'y a pas de solution qui traite cette problématique. Cependant, nous avons trouvé quelques solutions qui ont traité des problématiques similaires, mais pour des programmes destinés à la machine virtuelle Java, et qui peuvent nous inspirer.

3.3.1 Java ByteCode Debugger JBCD

Java ByteCode Debugger est un déboggeur interactif pour le bytecode Java, développé afin de permettre la visualisation des instructions bytecode, le défilement, la génération et la modification. Le JBCD peut être utilisé avec n'importe quel compilateur Java [22].

3.3.2 Dirty JOE Java Overall Editor

Dirty JOE est un éditeur complexe et un visionneur pour les fichiers Java compilés (fichier .class). Développé pour améliorer les fonctionnalités des éditeurs simple et visualiser le code binaire généré au cours de l'exécution. C'est un éditeur gratuit, écrit en C++ et utilise la bibliothèque python comme moteur de script. Il contient un visionneur pour les constants pool, les méthodes, les champs, les attributs, et un éditeur pour les constants pool, le bytecode, la tête de fichier et les attributs [23].

3.3.3 Java Snoop

Java Snoop est un outil développé au vouloir du piratage des applications Java au cours d'exécution. Il permet d'intercepter les méthodes, modifier les données et voir se qui se passe dans le système [24].

3.3.4 Jswat

Jswat est un débogueur Java présenté par une interface graphique écrite pour utiliser l'architecture de la plateforme de déboggeur Java *Java Platform Debugger Architecture*. Il est caractérisé par des points d'arrêt sophistiqués, d'affichage mobiles montrant les threads, une interface de commande avec des fonctionnalités plus avancées, et un visionneur de la pile d'appel. Il est développé à la nécessité de connaitre le contenu de la pile d'appel, les variables et les classes chargées [25].

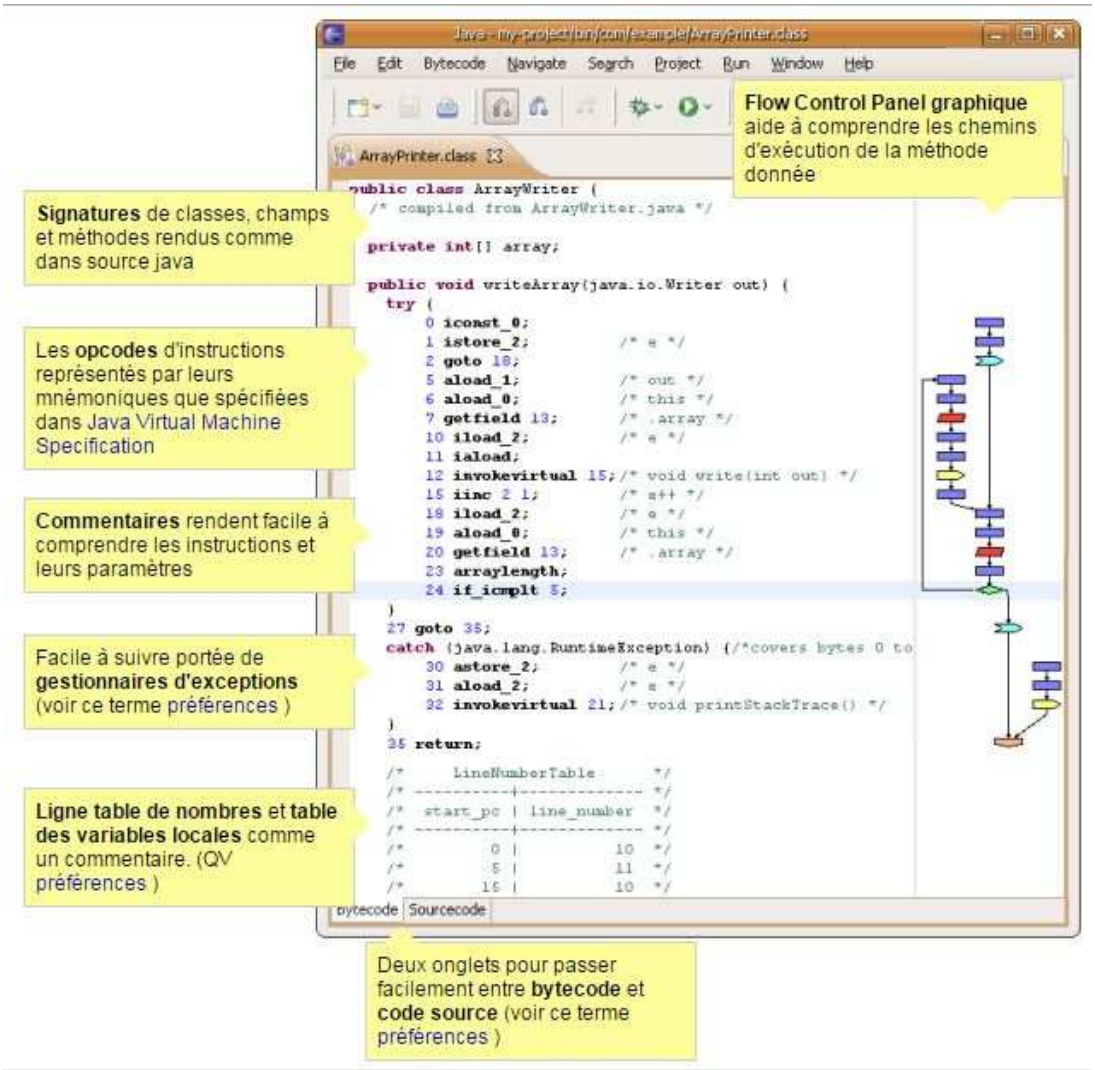
3.3.5 Dr. Garbage

Dr. Garbage Tools est un ensemble de plugins[1] pour *Eclipse*[2] disponible sous la licence "Apache Open Source licence".

Son objectif principal été de voir les instructions de la machine virtuelle Java à partir d'un code source Java, et les différents liens d'indépendances entre ces instructions, et les changements apportés sur la pile d'opérandes durant l'exécution de chaque instruction.

Dr. Garbage Tools contient 3 outils :

1. **Bytecode Visualizer** : c'est un outil qui débogue le bytecode, affiche le résultat de ces instructions malgré si le code source n'est pas complet (portion du code) et visualise les liens entres ces instructions (Figure 3.1), et contient plusieurs sous outils, parmi lesquels on trouve : l'**Oprand Stack Viewer** qui permet de voir la pile des opérandes d'une méthode sélectionnée (Figure 3.2).



1. Un paquet qui complète un logiciel hôte pour lui apporter de nouvelles fonctionnalités.
2. Un IDE (Environnement de Développement Intégré).

FIGURE 3.1 – le bytecode affiché par Dr. Garbage [21].

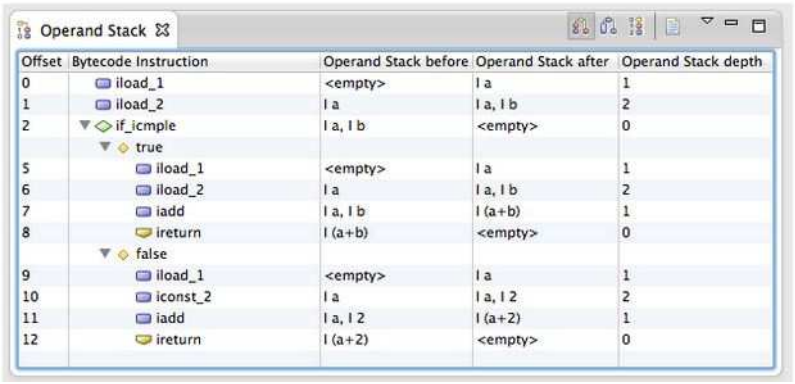


FIGURE 3.2 – La pile d'opérandes affichée par Dr. Garbage [21].

2. **SourceCode Visualizer** : c'est un outil pour examiner le code source. Il est composé de (Figure 3.3) :
   — **Editeur du code source** : c'est un éditeur qui a les mêmes caractéristiques que l'éditeur Eclipse Java.
   — **Panneau graphique du contrôle de flux** : c'est une fenêtre qui représente les instructions sous forme d'un graphe de flux de contrôle où chaque nœud représente une instruction de bytecode. Ce panneau peut être placé dans l'éditeur ou dans une fenêtre séparée.

   Pour chaque modification sur le code source, le graphique sera mise à jour automatiquement .

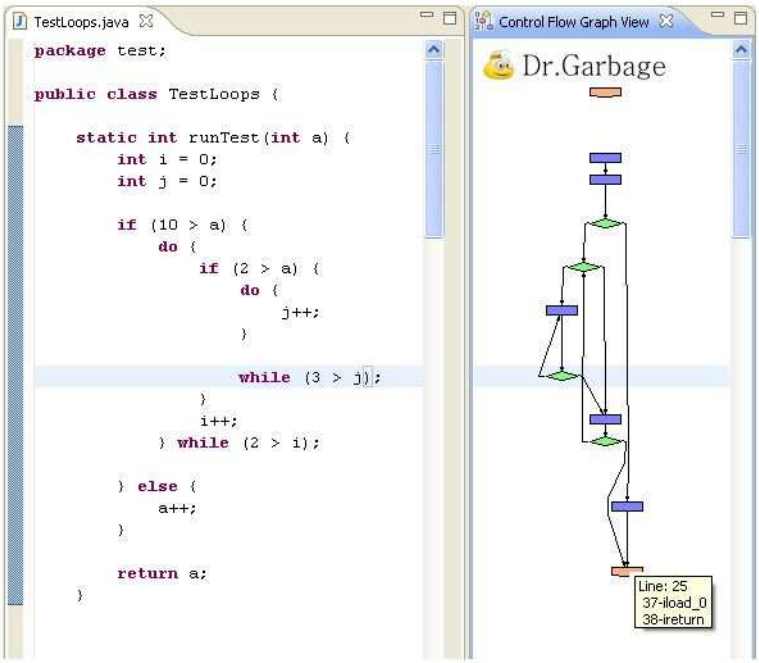


FIGURE 3.3 – Le code source et le flux de contrôle associé affichés par Dr. Garbage [21].

3. **Control Flow Graph Factory** : est un outil qui génère plusieurs types de graphe de contrôle de flux (Figure 3.4) :
   — Graphe de bytecode.
   — Graphe de code source.
   — Graphe de blocs de base.

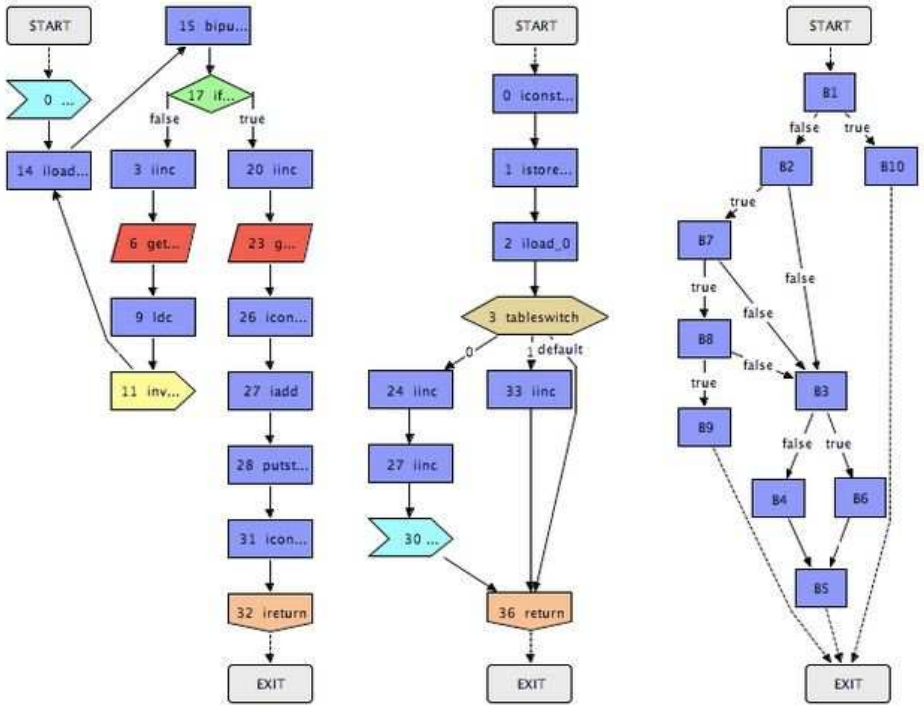


FIGURE 3.4 – Les types de graphe de contrôle affichés par Dr. Garbage [21].

3.4 Conclusion

Dans ce chapitre, nous avons introduit notre problématique et la démarche que nous allons suivre pour la résoudre. Nous avons précisé aussi qu'il n'existe actuellement aucune solution à cette dernière. Cependant, nous avons citer quelques outils qui peuvent nous inspirer.

Chapitre 4

Conception et implémentation

Sommaire

4.1 Problématique

L'objectif de notre travail est d'implémenter un outil qui permet de visualiser le contenu de la pile d'opérandes et le tableau des variables locales en représentant un état mémoire fournit à l'interprétation de chaque instruction bytecode d'une méthode donnée en entrée.

4.2 Principe

Pour l'implémentation de notre solution nous nous sommes basés sur un principe général qui regroupe les fonctionnalités viser par cet outil, présenté sous forme d'étapes :

1. Chargement d'un fichier CAP à partir d'un répertoire.
2. Séparations des méthodes contenues dans ce fichier.
3. Sélection d'une méthode à interpréter.
4. Transformation du bytecode de la méthode (binaire) en un fichier textuel (Figure 4.1).

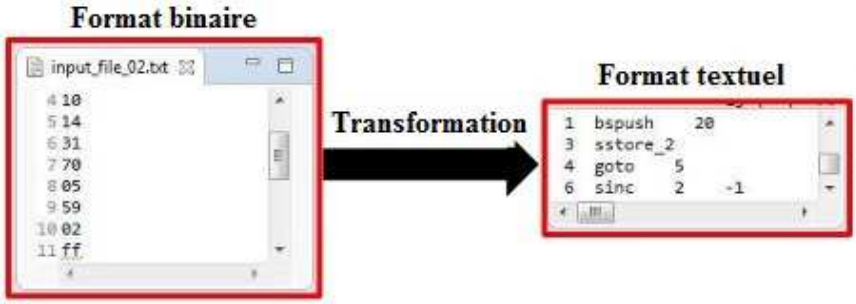


FIGURE 4.1 – Transformation d'un fichier binaire vers un fichier textuel.

5. Interprétation de la méthode choisie en affichant l'état mémoire courant (après l'exécution de chaque instruction byte code).

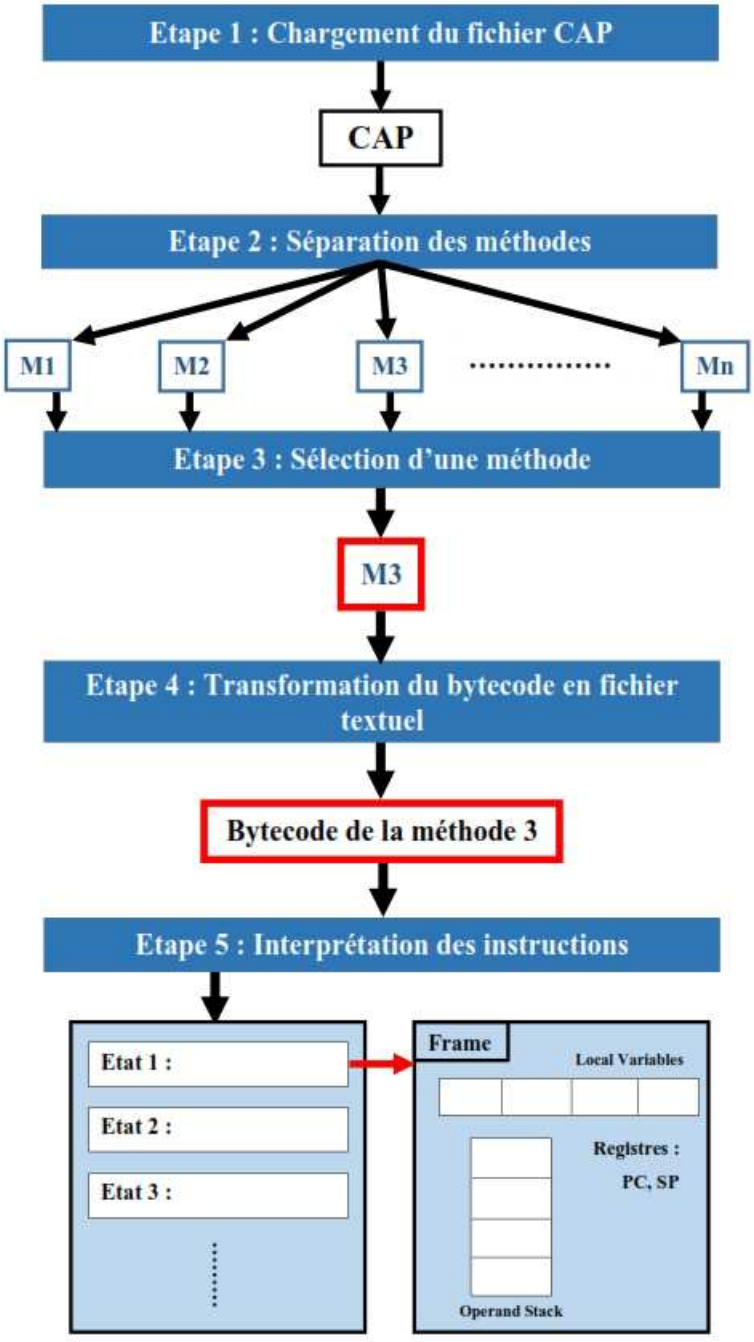


FIGURE 4.2 – Schéma du principe général de l'implémentation.

4.3 Conception

La résolution du problème posé est basée sur la conception de l'outil : la visualisation de l'état des zones mémoires de la machine virtuelle Java Card pendant l'exécution d'une méthode.

Une zone mémoire est l'espace où la machine virtuelle Java Card exécute ses instructions et stocke des données. Elle contient les registres, un tas, une ou plusieurs piles qui contiennent des frames. Chaque frame est associée à un thread, et elle contient une pile d'opérandes et un tableau des variables locales et une référence vers le code de la méthode à exécuter.

Donc pour développer cet outil, il faut avoir une frame. Sa conception est basée sur une classe *frame* qui contient une pile d'opérandes représentée par la classe *operand_ stack*, un tableau des variables locales représenté par la classe *local_ variables*, et le code de la méthode à exécuter représenté par la classe *method* qui utilise la classe *base* pour convertir le bytecode en instructions de la JCVM, à la fin une interface graphique est représentée par la classe *interface* pour visualiser le contenu des différents états mémoires (Voir la Figure 4.3).

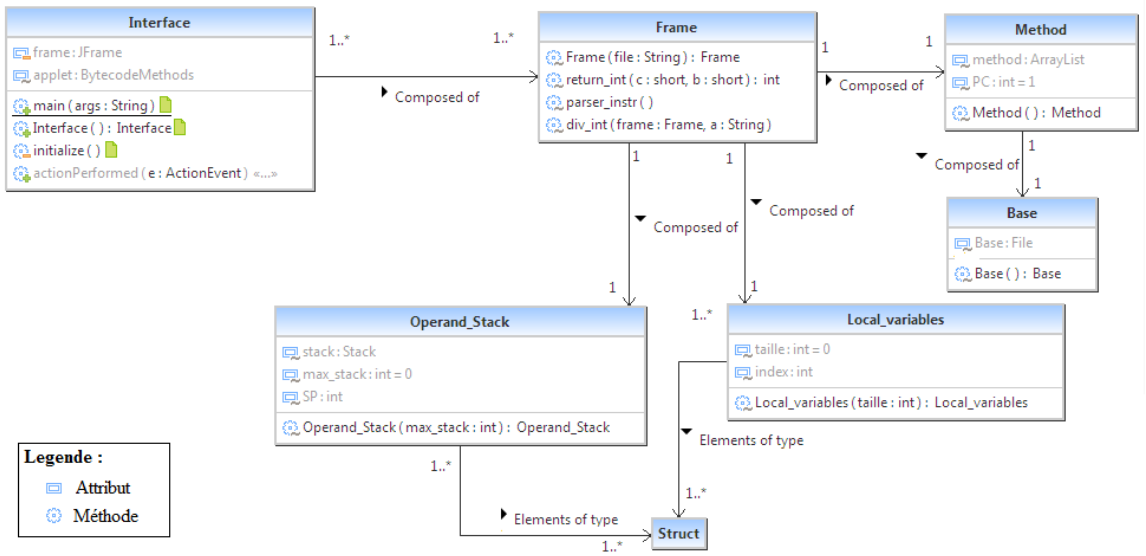


FIGURE 4.3 – Diagramme de classes de la conception.

4.4 Choix d'implémentation

Le développement de l'évaluateur est fait à l'aide d'Eclipse de Java , et avec les plugins et les bibliothèques nécessaires pour le fonctionnement de Java Card à fin de générer les fichier Cap des applets.

Eclipse est un IDE Environnement de Développement Intégré basé sur un langage orienté objet, simple et facile à programmer : Java [16].

The Cap File Manipulator (Cap Map) est choisit pour récupérer le fichier Cap et toutes les informations contenues dans ce fichier. C'est une bibliothèque Java qui permet de lire et modifier des fichiers CAP. Il est compatible avec Java Card Classic Edition 3.x [20].

L'interface est développée à l'aide de Java WindowBuilder (GUI Designer plugin), c'est un concepteur Java bi-directionnel qui propose deux Designer SWT et Swing, il permet de passer facilement entre le code et l'interface. Il facilite la création des application Java GUI, sans dépenser le temps à écrire du code [17].

4.5 Implémentation

Pour l'implémentation de cet outil, on a passé par les versions suivantes :

4.5.1 Version 1

Dans la première version de l'outil, on a commencé par implémenter le noyau de l'application, il prend en entrée le bytecode de la méthode à exécuter (la suite des instructions a été codée en dur dans le code de l'application), et le jeu d'instructions bytecode prise en considération été réduit (seulement celles qui manipule les bytes, les shorts et les ints), avec des contraintes fixes sur le max_stack (taille maximale de la pile d'opérandes) et max_locals taille maximale du tableau des variables locales).

4.5.1.1 Jeu d'instructions

Les instructions suivantes sont celles qui ont été implémentées dans la première version de l'outil (**Total** : 62 instructions).

- Opérations sur la pile des opérande et le tableau des variables locales :
  - Empiler une constante dans la pile des opérande :

    Byte : bspush.

    Short : sconst_<n>, sspush.

    Int : iconst_<n>.

    Note : n $\in \{m1, 0, 1, 2, 3, 4, 5\}$.
  - Retourner l'élément au sommet de la pile d'opérandes :

    Short : sload, sload_<n>.

    Int : iload, iload_<n>.

    Note : n $\in \{0, 1, 2, 3\}$.
  - Stocker un éléments depuis la pile d'opérandes vers le tableau des variables locales :

    Short : sstore,sstore_<n>.

    Int : istore, istore_<n>.

    Note : n $\in \{0, 1, 2, 3\}$.

- Opérations arithmétiques :

  Short : sadd, ssub, smul, sdiv, srem, sneg, sinc.

  Int : iadd, isub, imul, idiv, irem, ineg, iinc.

- Opérations logiques :

  Short : sand, sor, sxor, sshl, sshr, sushr.

  Int : iand, ior, ixor, ishl, ishr, iushr.

De plus, la sortie (contenus des états mémoires) peut être visualisée dans la console (Figure 4.4).

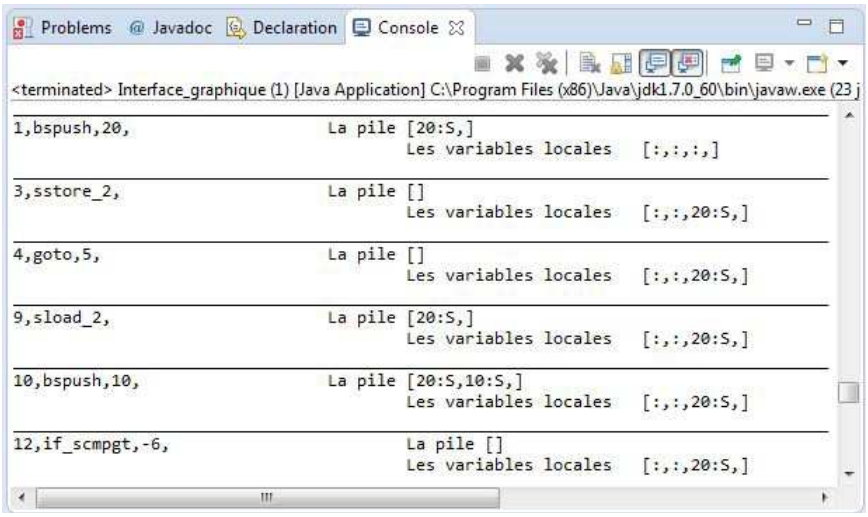


FIGURE 4.4 – Le résultat fournit par la version 1 de l'outil.

4.5.2 Version 2

Dans la deuxième version, on a ajouté la faculté d'extraire le code de la méthode à partir du fichier CAP avec l'extraction des contraintes sur le max_stack et max_locals, on utilisant la bibliothèque CapManipulator [20]. De plus, le jeu d'instructions a été enrichi.

4.5.2.1 Jeu d'instructions

La liste des instructions précédemment implémentées a été complétée par les instructions suivantes (**Total** = (62 + **56**) instructions).

- Opérations sur la pile des opérandes et le tableau des variables locales :
  - Empiler une constante dans la pile des opérandes :

    Byte : bipush.

    Short : sipush.

    Int : iipush.

    Ref : aconst_null.

  - Retourner l'élément au sommet de la pile d'opérandes :

    Ref : aload, aload_<n>.

    Note : n $\in \{0, 1, 2, 3\}$.

— Stocker un éléments depuis la pile d'opérandes vers le tableau des variables locales :

Ref : astore, astore_<n>.

Note : n $\in \{0, 1, 2, 3\}$.

— Instructions sans type qui modifient la pile d'opérandes :

1 élément : pop,dup.

2 éléments : pop2, dup2.

- Opérations arithmétiques :

  Short : sinc_w.

  Int : iinc_w.

- Conversion de type :

  Short : s2b, s2i.

  Int : i2b, i2s.

- Opérations modifiant le contrôle du flux :
  — Instructions de branchements :
    * Branchement conditionnel :
      if<cond>, if<cond>_w, if_scmp<cond>, if_scmp<cond>_w.
      Note : cond $\in \{eq, ne, lt, le, gt, ge\}$.
    * Branchement inconditionnel :
      goto, goto_w
  — Opération de comparaison :
  icmp.
  — Opérations sur les méthodes :
    * Opérations de retour des méthodes :
      areturn, sreturn, ireturn, return.

- Instruction qui ne change rien :

  nop.

4.5.2.2 Classes implémentées

Le développement est basé sur 8 classes, le diagramme de classe les montre (Figure 4.5) : la classe *Struct*, *Operand_ stack*, *Local_ variable*, *BytecodeManipulator*, *Method*, *BytecodeMethods*, *Frame*, *Interface*.

1. La classe ***Struct***
   Cette classe représente la structure utilisée pour coder les éléments du tableau des variables locales et la pile d'opérandes.
   Elle contient 2 variables de type **String** :
   — **val** représente la valeur de l'élément.
   — **type** représente son type.
   Et un constructeur :
   — **Struct (String val, String type)** : qui crée une instance de type *Struct* avec les valeur "val" et "type".
2. La classe ***Operand_ stack***
   Cette classe représente la pile d'opérandes, elle contient 3 variables : *stack*,*max_ stack*, *SP*.
   — **stack** : une variable de type Java.util.Stack qui manipule des éléments de type *Struct*.

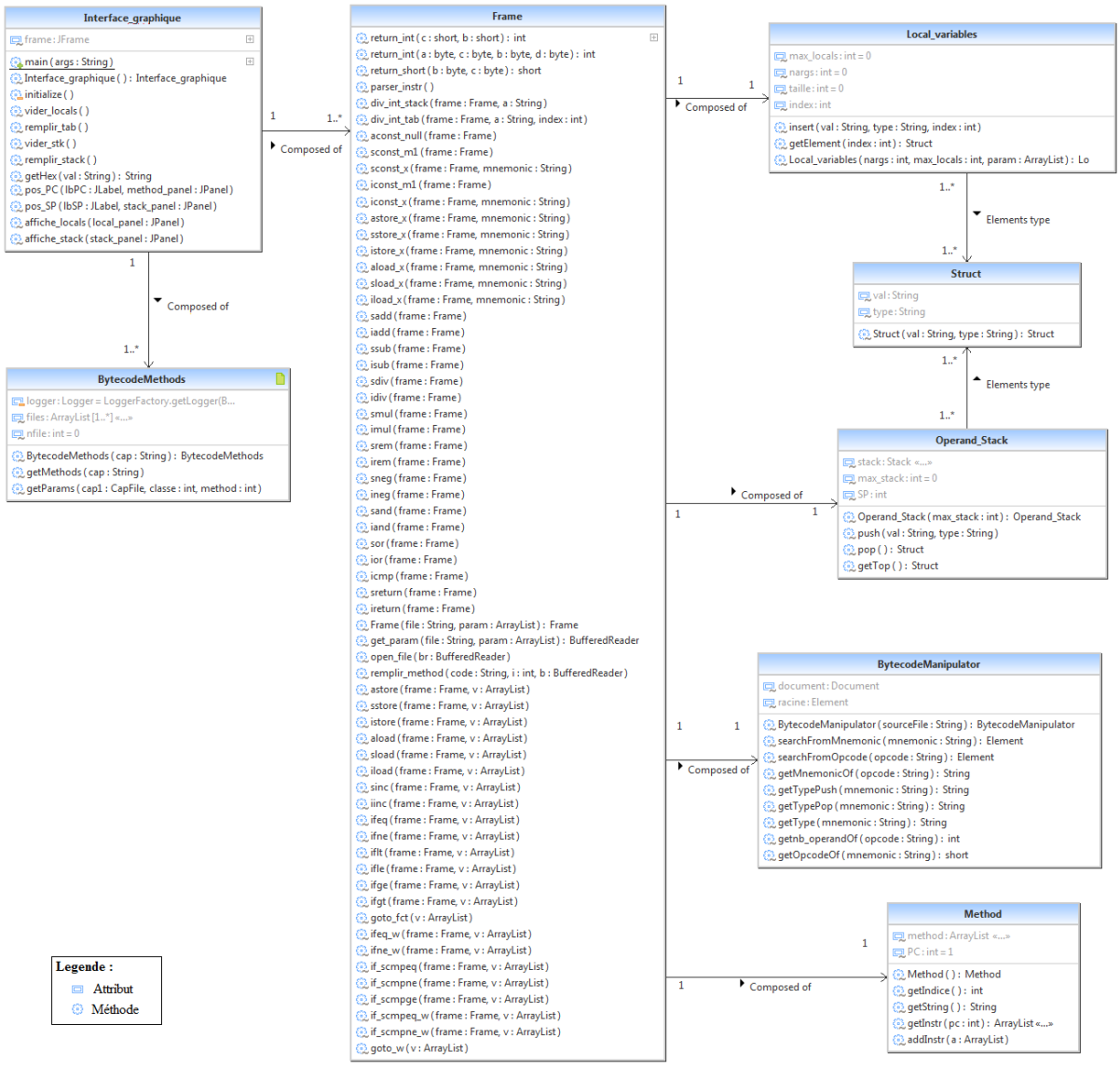


FIGURE 4.5 – Le diagramme de classes de l'implémentation de l'application.

— **max_stack** : un entier qui représente le nombre maximum les éléments contenus dans la stack.
— **SP** : un entier qui représente l'indice dans la stack où la prochaine valeur sera stockée.
La structure Java.util.Stack est une structure implémenté par java qui représente une pile avec toutes les fonctions nécessaires pour sa manipulation.

Cette classe regroupe 4 fonctions :
— **operand_stack (int max_stack)** : est un constructeur pour déclarer une pile avec une taille maximale *max_stack*.
— **push(String val, String type)** : permet d'empiler une valeur "val" de type "type" dans la pile.
— **pop()** : permet de dépiler un element de type *Struct* à partir du sommet de la pile.
— **get_top()** : permet de récupérer l'élément qui est au sommet de la pile.

3. La classe ***Local_variables***

Cette classe représente le tableau des variables locales, elle définit 5 variables :
— **local_var** : est un tableau dynamique de type *Struct* qui contient les variables locales.
— **nargs** : est le nombre de paramètres de la méthode en cours.
— **max_locals** : est le nombre des variables locales de la méthode.
— **taille** : est le nombre maximum des éléments que peu contenir le tableau.
— **index** : est l'indice dans le tableau où le prochain élément sera inséré.
Elle regroupe 3 fonctions :
— **Local_variable (int nargs, int max_local, ArrayList<String> param)** : constructeur pour créer un tableau des variables locales de taille max = (nargs + max_local) et l'initialisé par "param".
— **insert (String val, String type, int index)** : permet d'insérer un élément de valeur "val" et de type "type" à l'indice "index".
— **getElement (int index)** : permet de récupérer l'élément à l'indice "index".

4. La classe ***BytecodeManipulator***

Cette classe permet de récupérer des informations à partir de la base des instructions (Dictionnaires d'instructions) sauvegardées dans un fichier .xml.

(a) Fichier XML :

La base des instructions et leurs informations ont été stockées dans un fichier .xml car il a une structure simple à modifier selon les besoins, il minimise l'accès au fichier pour le remplir, facilite la modification et la récupération des données.

Une instruction est stockée avec (Figure 4.6) :

- Son opcode.
- Son mnemonic.
- Nombre d'opérandes de l'instruction "nb_operande".
- Nombre de Production sur la pile "nb_push".
- Type des éléments produits "type_push".
- Nombre de Consommation depuis la pile "nb_pop".
- Type des éléments consommés "type_pop".

FIGURE 4.6 – Un fragment du fichier .xml (Dictionnaire d'instructions).

(b) BytecodeManipulator : Cette classe utilise 2 variables :
- **document** de type Document est le fichier .xml utilisé comme dictionnaire des instructions.
- **racine** de type Element qui représente une instruction avec toutes ses information.

Elle regroupe 9 fonctions :
- **BytecodeManipulator (String sourceFile)** : constructeur qui créé une nouvelle instance de la classe BytecodeManipulator qui utilise le fichier .xml "sourceFile".
- **getMnemonicOf (String opcode)** : permet de retourner le mnemonic de l'instruction qui a l'opcode correspondant.
- **getOpcodeOf (String mnemonic)** : permet de retourner l'opcode de l'instruction qui a le mnemonic correspondant.
- **getnb_operandOf (String opcode)** : permet de retourner le nombre d'opérandes de l'instruction qui a l'opcode correspondant.
- **getType (String mnemonic)** : permet de retourner type de l'instruction à partir du mnemonic.
- **getTypePush (String opcode)** : permet de retourner le type des éléments qui seront insérés dans la pile pendant l'exécution de cette instruction.
- **getTypePop (String opcode)** : permet de retourner le type des éléments qui seront supprimés depuis la pile pendant l'exécution de cette instruction.
- **searchFromOpcode (String opcode)** : permet de retourner l'élément qui a l'opcode correspondant.
- **searchFromMnemonic (String mnemonic)** : permet de retourner l'élément qui a le mnemonic correspondant.

5. La classe ***Method***

Cette classe représente le bytecode de la méthode qui sera exécutée. Elle définit 2 variables :
- **method** est une liste dynamique de type Java.util.ArrayList à 2 dimensions qui contient des éléments de type String permettant de stocker les instructions bytecode (l'offset de l'instruction, le mnemonic et les opérandes avec le numéros de l'instruction).
- **PC** est un entier qui représente l'offset de l'instruction en cours d'exécution.

Elle regroupe 5 fonctions :
- **Method()** : constructeur qui crée une instance de la classe Method.
- **addInstr (ArrayList a)** : permet d'ajouter une instruction dans la liste des instructions.
- **getInstr (int pc)** : permet de retourner l'instruction qui a l'offset "pc".

— **getString ()** : permet de retourner toutes les instructions sous forme de chaines de caractères.
— **getIndice ()** : permet de retourner le numéro de l'instruction en cours d'exécution.

6. La classe ***Frame***

Cette classe représente une frame de la machine virtuelle, elle contient :
— **stack** : une pile de type *operand_ stack*.
— **local_var** : un tableau pour les variables locales de type *Local_ variables*.
— **base** : une base d'instructions de type *BytecodeManipulator*.
— **method** : une méthode à exécuter de type *Method*.

Parmi les fonctions contenues dans cette classe, on cite :
— **Frame (String file, ArrayList<String> param)** : constructeur qui crée une instance de type frame qui utilise le fichier "file" (contenant le bytecode de la méthode) et initialise le tableau des variables locales avec "param".
— **open_file (BufferedReader br)** : permet d'ouvrir le fichier contenant le bytecode de la méthode et de le parcourir ligne par ligne à l'aide du Buffer "br".
— **get_param (String file)** : permet de retourner les paramètres des constructeurs de la stack et du tableau des variables locales depuis le fichier "file".
— **remplir_method (String code, int i,BufferedReder b)** : permet d'insérer l'instruction qui a le numéro "i" et l'opcode "code" avec son offset et toutes ses opérande récupérer à l'aide du Buffer "b" qui parcours ligne par ligne, dans la liste des instructions (method).
— **parser_instr ()** : permet d'exécuter une instruction.
— **return_int (Short c, Short b)** : permet de retourner un entier à partir de 2 valeurs de type short.
— **return_int (Byte a, Byte b, Byte c, Byte d)** : permet de retourner un entier à partir de 4 valeurs de type Byte.
— **return_short (Byte b, Byte c)** : permet de retourner un short à partir de 2 valeurs de type Byte.
— **div_int_stack (Frame frame, String a)** : permet de diviser un entier en 2 short et de les insérer dans la stack.
— **div_int_tab (Frame frame, String a, int index)** : permet de diviser un entier en 2 short et de les insérer dans le tableau des variables locales contenant dans "frame" à l'indice "index".

Le reste des fonctions sont les fonctions qui traitent les instructions bytecode de façon individuelle (une fonction/instruction).

7. La classe ***BytecodeMethods***

Cette classe représente la classe qui manipule le fichier CAP à l'aide du Cap Map. Elle définit 2 variables :
— **Logger** de type "Logger".
— **files** : tableau dynamique de type ArrayList<String> pour stocker le nom des fichiers contenant les méthodes du fichier CAP à exécuter et les arguments de chaque méthode.

Elle regroupe 2 fonctions :
— **BytecodeMethods (String cap)** : un constructeur qui permet de créer une instance de la classe *BytecodeMethodes*.
— **getMethods (File cap)** qui permet de séparer les méthodes d'un fichier CAP donné en paramètre, et de mettre chacune dans un fichier texte et garder leur noms dans le tableau "files".

8. La classe ***Interface_ graphique***

   Cette classe représente l'interface de l'application, elle utilise 3 variables :
   - **frame** : instance de la classe javax.swing.JFrame : classe java permettant de créer et de générer des objets de type fenêtre de haut niveau divisée en sous composant et qui peut contenir d'autres composants d'interface tel que les boutons et les zones de saisie [?].
   - **frame_code** : instance de la classe *Frame*.
   - **applet** : instance de la classe*BytecodeMethodes*.

   Elle regroupe 12 fonctions :
   - **main()** : est la fonction principale qui permet d'exécuter cette classe.
   - **Interface_graphique()** : constructeur qui permet de créer une instance de la classe *Interface_ graphique*.
   - **inisialize()** : permet d'initialiser les champs de l'interface.
   - **vider_locals()** : permet de vider le contenu du champ dédié au tableau des variables locales dans l'interface.
   - **remplir_tab()** : permet de récupérer le contenu de l'instance de la classe *local_ variables* et de l'afficher dans le tableau des variables locales de l'interface.
   - **vider_stk()** : permet de vider le contenu du champ de la pile dans l'interface.
   - **remplir_stack()** : permet de récupérer le contenu de l'instance de la classe *operand_ stack* et de l'afficher dans la pile de l'interface.
   - **getHex()** : permet de retourner la valeur hexadécimale d'une valeur donnée.
   - **affiche_locals()** : permet d'afficher le cadre du tableau des variables locales et le numéro d'indice.
   - **affiche_stack()** : permet d'afficher le cadre de la pile et le numéro d'indice.
   - **pos_PC()** : permet de positionner le registre PC dans le code de la méthode.
   - **pos_SP()** : permet de positionner le registre SP dans la pile.

4.5.3 Version 3

Dans la dernière version, on a développé l'interface graphique qui permet de visualiser l'état mémoire et les changements portés sur la pile et le tableau des variables locales ainsi que les registres PC et SP. Ceci après l'exécution de chaque instruction bytecode de la méthode.

4.5.3.1 Interface graphique

Voici l'interface graphique de l'application (Figure 4.7). Le cadre "Local variables" représente le tableau des variables locales, "Operand stack" représente la pile et "Method" représente le code de la méthode à exécuter (écrite sous forme textuelle).

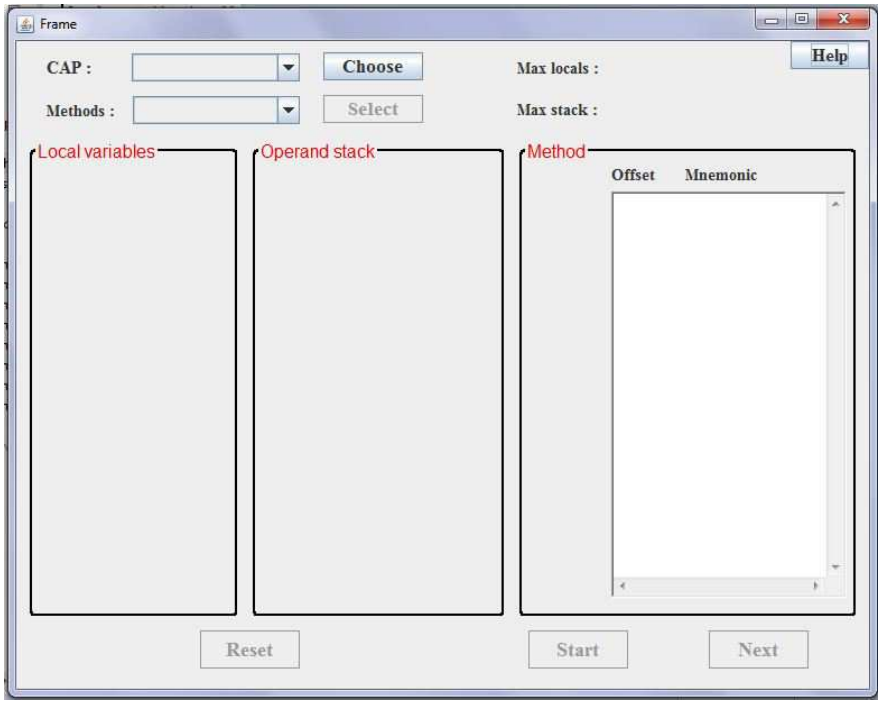


FIGURE 4.7 – L'interface graphique de l'application.

- Le bouton "Help" permet d'afficher un aide sur le fonctionnement de l'outil.
- Le bouton "Choose" permet de choisir et de charger un fichier CAP.
- Le bouton "Select" permet de choisir une méthode à exécuter, d'afficher le nombre maximum des éléments (max_stack) qui peuvent être contenus dans la stack et dans le tableau des variables locales (max_locals), afficher l'état initiale de la stack et du tableau des variables locales et de remplir la méthode avec son code.
- Le bouton "Start" permet de commencer l'exécution de la première instruction.
- Le bouton "Next" permet d'exécuter la prochaine instruction.
- Le bouton "Reset" permet de mettre la stack, le tableau des locales, la taille max à zéro et de vider la méthode pour une nouvelle exécution d'une autre méthode.

4.5.3.2 Exemple Pratique

Pour illustrer les différents champs et options de l'application, la Figure 4.8 montre une capture de l'application prise durant l'exécution d'une méthode extraite à partir d'un fichier cap (exécution d'une instruction de la méthode 2 du fichier Cap "myapplet_2.cap").

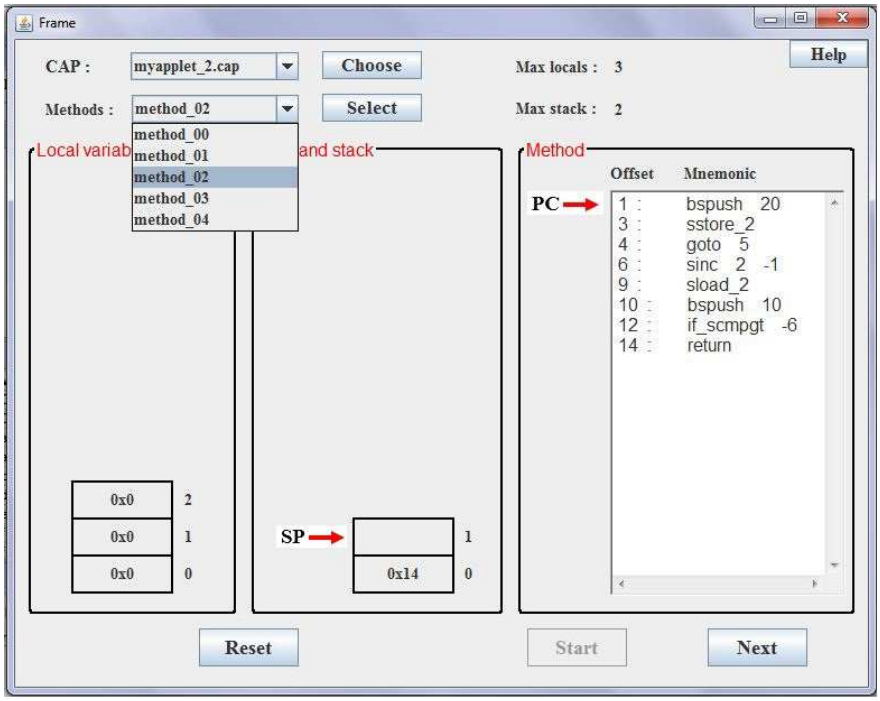


FIGURE 4.8 – Exemple pratique pour l'utilisation de l'application.

4.6 Conclusion

Ce chapitre a permis de présenter d'abord la conception de notre outil qui nous a permis de mieux choisir les outils d'implémentation. Par la suite, on a procédé à l'implémentation de la solution, en passant par 3 versions (enrichissement lors du passage d'une version à une autre).

Conclusion et perspectives

Objectif principal du projet est de développer un outil permettant d'évaluer l'état des zones mémoires de la machine virtuelle Java Card durant l'exécution d'une méthode donnée (un état par instruction exécutée).

Nous avons commencé le travail par une recherche bibliographique afin d'étudier les spécificités des deux langages Java et Java Card et leurs machines virtuelles respectives, faire le point sur les différences entre les deux, et comprendre le mécanisme de manipulation des composantes mémoire.

Par la suite, il fallait étudier la spécification du jeu d'instructions bytecode Java Card, et comprendre le contenu du fichier CAP pour pouvoir identifier les composants qui rentrent en jeu lors de l'interprétation, et étudier les différents étapes de cette dernière.

Pour atteindre notre objectif, nous avons commencé par développer un noyau avec un jeu d'instruction réduit qui prend en entrée directement le bytecode de la méthode, puis l'améliorer en terme d'instructions prises en considération, les contraintes et l'entrée de l'outil (un fichier CAP). Par la suite, on est passé au développement d'une interface graphique qui améliore la visualisation des différents états mémoires à chaque instant de l'interprétation du bytecode.

Les perspectives d'amélioration de ce travail portent essentiellement sur les instructions bytecodes non traitées, comme les instructions d'appel de méthodes internes au package ou externes, les instructions qui manipulent les références, les objets, celles qui ont un nombre variant d'opérandes, et celles traitant les exceptions. De plus, l'ergonomie de l'interface pourrait être améliorée.

Références

Bibliographie

[1] André Etienne. Mise en œuvre et évaluation d'analyses de bytecode Java spécifiées avec DATALOG. Rapport de PFE 5ème Informatique. Institut National des Sciences Appliqués de Rennes, 2006.

[2] Bada Mousaad. Les problèmes de sécurité dans les systèmes embarqués. Mémoire de Magister. Université El-Hadj Lakhdar - BATNA, 2012.

[3] Bouchenak-Khelladi Sahra. Mécanismes pour la migration de processus : Extension de la machine virtuelle Java. Mémoire de DEA. Laboratoire INRIA Rhônes-Alpes – Projet SIRAC, 12 juin 1998.

[4] Boumassata Meriem. Vérification de code pour plates-formes embarquées. Mémoire de Magister. Université El-Hadj Lakhder-Batna, 2012.

[5] Charreteur Florence et Gottlieb Arnaud. Raisonnement à contraintes pour le test de bytecode Java. Universitéde Rennes 1, INRIA, 2008.

[6] Hamadouche Samiya. Étude de la sécurité d'un vérifieur de byte code et génération de tests de vulnérabilité. Mémoire de Magister, Université M'Hamed Bougara De Boumerdes, 2012.

[7] Hamadouche Samiya et Lanet Jean-Louis et Mezghiche Mohamed. Méthode d'Analyse de Vulnérabilité Appliquée à un Composant de Sécurité d'une Carte à Puce. Rencontres sur la Recherche en Informatique R2I, Tizi Ouzou, Algerie, 12-14 juin 2011.

[8] Mesbah abdelhak et Ferkout Hamza. Réalisation d'un système de mise à jour pour les programmes Java. Mémoire de Master. Uuniversité de Boumerdes, 2013.

[9] Mokhtari Tarek et Amzal Hanane. Etude des attaques par injection de fautes contre les cartes à puce. Mémoire de projet tuteuré Master 1. Université M'Hamed Bougara De Boumerdes, 2014.

[10] Oracle : Lindholm Tim et Yellin Frank et Bracha Gilad et Buckly Alex. The Java Virtual Machine Specification. Java SE 8 Edition, 2015.

[11] Oracle. Java Card 3 Platform : Virtual Machine Specification, Classic Edition, Version 3.0.4, September 2011.

Webographie

[12] Beschi Yohan, JVM HARDCORE - Part 1 - Introduction à la JVM, SOAT (`blog.soat.fr/2013/09/02-jvm-hardcore-part1-introduction-a-la-jvm`), consulté le 27 fevrier 2015.

[13] Conseil en Systèmes d'Information, Urbanisation, Architectures et Expertise JEE, IT^{TM}, `http://it.fr/index.php/tutoriels/java-ee.html/`, consulté le 03 mars 2015.

[14] `docs.oracle.com/javase/8/docs/api/javax/swing/JFrame.html`, consulté le 24 juin 2015.

[15] Doudoux Jean-Michel, `http://www.jmdoudoux.fr/java/dej/chap-jvm.htm/`, consulté le 02 mars 2015.

[16] `eclipse.developpez.com/faq/?page=plateform#definitionEclipse`, consulté le 18 juin 2015.

[17] `eclipse.org/windowbuilder/`, consulté le 18 juin 2015.

[18] Lexique informatique, (`www.lexiqueinformatique.fr/lex_info/java-card/`), consulté le 03 mars 2015.

[19] Parageaud Christophe, Ippon Technologies, disponible au `http://blog.ippon.fr/2011/11/03/java-acces-directs-a-la-memoire-off-heap/`, consulté le 01 mars 2015.

[20] `secinfo.msi.unilim.fr/software/`, consulté le 18 juin 2015.

[21] Site web officiel de Dr. Garbage (`www.drgarbage.com`), consulté le 16 avril 2015.

[22] Site web officiel de JBCD (`jbcd.sourceforge.net`), consulté le 18 avril 2015.

[23] Site web officiel de Dirty JOE (`dirty-joe.com`), consulté le 18 avril 2015.

[24] Site web de Java Snnop (`code.google.com/P/javasnoop`), consulté le 18 avril 2015.

[25] Site web de Jswat (`github.com/nlfiedler/jswat/`), consulté le 18 avril 2015.

[26] Site officiel de Java (`https://www.java.com/fr/download/faq/whatis_java.xml`), consulté le 02 mars 2015.

[27] `www.programceek.com/2013/04/jvm-run-time-data-areas/`, consulté le 03 mars 2015.