(12) DEMANDE INTERNATIONALE PUBLIÉE EN VERTU DU TRAITÉ DE COOPÉRATION EN MATIÈRE DE BREVETS (PCT)

(19) Organisation Mondiale de la Propriété Intellectuelle
Bureau international

(43) Date de la publication internationale
20 décembre 2001 (20.12.2001)

PCT

(10) Numéro de publication internationale
WO 01/97013 A2

(51) Classification internationale des brevets[7] : G06F 9/00

(21) Numéro de la demande internationale :
PCT/FR01/01848

(22) Date de dépôt international : 14 juin 2001 (14.06.2001)

(25) Langue de dépôt : français

(26) Langue de publication : français

(30) Données relatives à la priorité :
00/07629 15 juin 2000 (15.06.2000) FR

(71) Déposant *(pour tous les États désignés sauf US)* : **CANAL + TECHNOLOGIES** [FR/FR]; 34 Place Raoul Dautry, F-75015 PARIS (FR).

(72) Inventeur; et

(75) Inventeur/Déposant *(pour US seulement)* : **GUISSOUMA, Habib** [FR/FR]; 116 Boulevard Jean Jaurés, F-92100 BOULOGNE (FR).

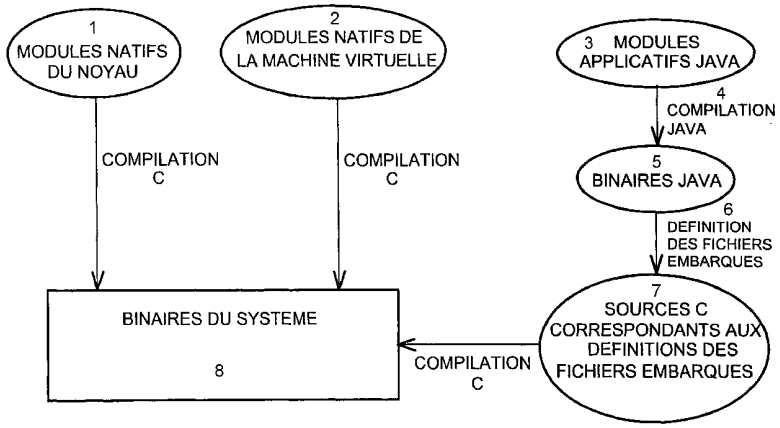
(74) Mandataire : **DU BOISBAUDRY, Dominique**; c/o BREVALEX, 3, rue du Docteur Lancereaux, F-75008 PARIS (FR).

(81) États désignés *(national)* : AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW.

[Suite sur la page suivante]

(54) **Title:** METHOD FOR MANAGING CLASSES FOR AN OBJECT-ORIENTED PROGRAMMING LANGUAGE WITH DYNAMIC RESOLUTION AND TOOL FOR GENERATING NATIVE STRUCTURES FROM JAVA BINARIES IMPLEMENTING SAID METHOD

(54) **Titre :** PROCEDE DE GESTION DE CLASSES POUR UN LANGAGE DE PROGRAMMATION ORIENTE OBJET A RESOLUTION DYNAMIQUE ET OUTIL DE GENERATION DES STRUCTURES NATIVES DEPUIS LES BINAIRES JAVA METTANT EN OEUVRE CE PROCEDE



1...CORE NATIVE MODULES
2...VIRTUAL MACHINE NATIVE MODULES
3...JAVA APPLICATIVE MODULES
4...JAVA COMPILATION
5...JAVA BINARIES
6...ONBOARD FILES DEFINITION
7...SOURCES C CORRESPONDING TO ONBOARD FILES DEFINITIONS
8...SYSTEM BINARIES

(57) **Abstract:** The invention concerns a method for managing classes for an object-oriented language with dynamic class resolution, which consists in transforming the dynamic class resolution for a group of self-contained classes into a static resolution system. The invention also concerns a tool for generating native structures from object binaries implementing said method

[Suite sur la page suivante]

WO 01/97013 A2

(84) États désignés *(régional)* **:** brevet ARIPO (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), brevet eurasien (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), brevet européen (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE, TR), brevet OAPI (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Publiée :

— *sans rapport de recherche internationale, sera republiée dès réception de ce rapport*

En ce qui concerne les codes à deux lettres et autres abréviations, se référer aux "Notes explicatives relatives aux codes et abréviations" figurant au début de chaque numéro ordinaire de la Gazette du PCT.

(57) Abrégé : L'invention concerne un procédé de gestion de classes pour un langage orienté objet avec résolution de classe dynamique, dans lequel on transforme la résolution de classe dynamique pour un groupe de classes autonomes en un système en résolution statique. L'invention concerne également un outil de génération des structures natives depuis les binaires objet mettant en oeuvre ledit procédé.

PROCEDE DE GESTION DE CLASSES POUR UN LANGAGE DE PROGRAMMATION ORIENTE OBJET A RESOLUTION DYNAMIQUE ET OUTIL DE GENERATION DES STRUCTURES NATIVES DEPUIS LES BINAIRES JAVA METTANT EN ŒUVRE CE PROCEDE

DESCRIPTION

Domaine technique

L'invention se rapporte à un procédé de gestion de classes pour un langage de programmation orienté objet à résolution dynamique et à un outil de génération des structures natives depuis les binaires Java mettant en œuvre ce procédé.

Etat de la technique

Pour un langage de programmation "être orienté objet" signifie qu'en dehors des types de base prédéfinis tels que les nombres ou les caractères, tout est objet. Les programmes sont constitués par assemblages des composants logiciels que sont les objets. L'objet est la réunion dans une même entité d'une structure de donnée et des algorithmes permettant de manipuler cette structure de donnée. Cette idée s'oppose à celle de la programmation conventionnelle où données et traitements sont séparés.

Le domaine de l'invention est celui des langages orientés objet à résolution dynamique, plus particulièrement celui du langage Java

Le langage Java

Le langage Java est un langage de programmation de haut niveau, orienté objet, simple, interprété, sécurisé, portable, et dynamique. Il élimine les défauts et reprend le meilleur aspect des langages orientés objets tels que le C++, tout en simplifiant son utilisation (la mémoire, par exemple, n'est plus gérée par le développeur). La portabilité du langage est assurée par son environnement d'exécution, la machine virtuelle Java, qui est adaptée au système sur lequel vont s'exécuter les taches et applications développées dans ce langage. Son développement à été favorisé par la montée en puissance du réseau Internet, l'abstraction de la plate-forme d'exécution se trouvant adaptée à ce réseau hétérogène.

L'approche objet (d'un système) en informatique se différencie de l'approche procédurale (découpe d'une tache en un ensemble de sous-taches). Les objets s'échangent des messages et interagissent de manière à réaliser les fonctionnalités requises.

Un objet peut être défini de la façon suivante :

- Un objet est une entité identifiée et nommée.
- L'état de cette entité est défini par un ensemble d'attributs :
- Un ensemble d'opérations (méthodes) déterminent le comportement de l'objet.
- Un objet est une instance de classe
- Une classe est un type de données

abstrait définissant des attributs et méthodes. Ces propriétés sont communes aux instances de la classe.

• Les méthodes et attributs d'une classe d'objets forment les propriétés de la classe d'objets.

Les principaux concepts objet sont décrit ci après :

• Encapsulation : ce concept masque les détails de mise en œuvre d'un objet en définissant une interface (vue externe de l'objet).

• Héritage : ce concept définit un transfert d'attributs et méthodes d'une classe vers une autre classe (sous-classe). Une classe peut se spécialiser en une sous classe. Une sous-classe peut hériter de plusieurs classes (multi-héritage).

• Agrégation : ce concept permet de définir des objets composés d'autres objets.

Le langage Java reprend les concepts objets et une syntaxe très proche du langage C/C++. Certaines spécificités du langage Java sont considérées ci-dessous :

• Machine virtuelle et environnement d'exécution : pour assurer la portabilité des sources (et binaires) Java, les binaires s'exécutent dans la machine virtuelle Java, qui est une couche d'abstraction entre le binaire et le l'environnement matériel (microprocesseur, ports de communication,…)

• Interfaces et mono-héritage : pour des raisons de performances, le multi-héritage n'est pas supporté. Cependant une notion d'interface (classe abstraite dont

on ne peut hériter) est introduite, une interface servant à décrire un ensemble de méthodes. Une classe d'objets peut mettre en œuvre une interface dès lors qu'elle définit l'ensemble des méthodes décrites par l'interface.

- Paquetages : les classes sont, éventuellement, regroupées par paquetages ("packages"), un paquetage consistant en un regroupement de classes par affinités.
- Paquetages standard Java : avec la machine virtuelle Java sont définis un certain nombre de paquetages de base sur lesquels peuvent se reposer les développeurs Java pour concevoir des applications portables. Les principaux sont les suivants :
  - Java.lang : paquetage contenant les classes de base du langage (chaînes de caractères,...)
  - Java.util : paquetage définissant des classes utilitaires telles que des tables de hachage
  - Java.io : paquetage fournissant des fonctionnalités, de type entrées/sorties (fichiers,...)
  - Java.net : paquetage concernant les fonctionnalités réseau de Java.

Chacune des propriétés d'une classe d'objet possède un ensemble d'attributs. L'un de ces attributs est la visibilité de la propriété :

- Accès privé ("private") : seule la classe peut accéder à sa propriété
- Accès paquetage ("package protected") : toutes les classes d'objets du paquetage peuvent accéder à la propriété.
- Accès protégé ("protected") : la propriété n'est

accessible que depuis la classe d'objet et ses sous-classes.

- Accès public : l'accès à la propriété est ouvert à toutes les classes d'objet.

Les machines virtuelles Java ne chargent que les classes d'objets utilisées par un applicatif. A la première utilisation d'une classe, celle-ci est définie de manière interne et mise à la disposition de l'applicatif.

Chaque classe possède une table des classes qu'elle va être amenée à utiliser. Lors de l'utilisation d'une de ces classes, elle demande une référence sur la structure interne de la classe pour pouvoir l'exploiter. Ce processus, pour des raisons de performances, n'est effectué qu'une seule fois par classe utilisée. Ceci est effectué en substituant une opération nécessitant une résolution de classe par une opération équivalente partant du principe que la classe est résolue (ie : connue).

Les systèmes embarqués

Les systèmes embarqués sont des systèmes logiciels assurant un service déterminé à l'avance. Ces systèmes s'exécutent dans un environnement matériel dédié à la fonctionnalité réalisée.

Les décodeurs de télévision numérique se basent sur un système embarqué temps réel ainsi qu'une boîte électronique, le tout étant conçu pour réaliser des services de télévision payante.

Le système "Media-Highway" ainsi que les

outils de développement associés fournissent les moyens applicatifs de développer des services interactifs tournant autour du métier de la télévision numérique, tels que des services de sélection de programmes.

Dans sa déclinaison Java ("MediaHighway Virtual Machine"), le système "MediaHighway" inclut une machine virtuelle Java, des outils de développement associés ainsi qu'un ensemble de services interactifs.

Les systèmes de télévision numérique développés par Canal+ sont destinés à des clients divers et se déroulent dans des machines de conceptions relativement différentes les unes des autres le choix des composants matériels étant laissé au constructeur.

Le système "MediaHighway+" (MHPLUS) est un système embarqué de télévision numérique dont les applications sont conçues en langage Java.

Le choix du langage Java est motivé d'une part du fait du potentiel de ce langage et d'autre part de l'abstraction qui est faite au niveau de l'environnement matériel.

Les sources Java sont compilées de manière à produire les binaires Java associés. Les binaires produits se présentent comme des fichiers. A chaque paquetage correspond un répertoire. A chaque classe correspond un fichier décrivant la classe. Ce fichier se trouve dans le répertoire associé au paquetage de la classe. Le noyau logiciel, sur lequel s'appuie la machine virtuelle Java embarquée, possède une interface logicielle pour accéder à un système de fichiers. Ce système de fichiers est capable de gérer des fichiers

dont les définitions peuvent se trouver à divers endroits. Une de ces localisations est le binaire lui-même. Dans les données du binaire, tout un ensemble de structures liées entre elle forme le système de fichiers dit "en ROM" (puisqu'il se retrouve au final dans la mémoire ROM non inscriptible de la machine cible). Les binaires Java se trouvent finalement dans le système de fichiers "en ROM" du gestionnaire de fichiers. L'arborescence de répertoire et de fichiers correspondant aux paquetages et classes des binaires Java est directement reproduite dans le système de fichiers "en ROM". Pour ce faire, un outil permet de générer du code source en langage C. Le code source une fois compilé définit l'arborescence relative aux binaires Java. Le résultat de cette compilation, une fois intégré au binaire final, fournit à la machine virtuelle Java les définitions de classes nécessaire à son fonctionnement.

La machine virtuelle embarquée possède un gestionnaire de classes, qui est un module applicatif chargé de la base de données des classes Java utilisées.

Lorsqu'une classe d'objet est nécessaire au fonctionnement d'un ensemble applicatif, et que cette classe n'est pas encore définie, la machine virtuelle Java procède de la manière suivante pour mettre à disposition la classe :

1. Chargement de la description de la classe (fichier binaire résultant de la compilation d'une source Java),

2. Définition, au besoin, de la classe parente de celle-ci (ce qui revient à appliquer ces mêmes étapes pour la super-classe),
3. Initialisation de la classe (exécution de code Java défini dans le source de la classe).

Si toutes ces étapes se passent sans erreurs, le gestionnaire de classes fournit à la machine virtuelle la définition interne de la classe.

Dans le cas général, la description de la classe est chargée depuis le système de fichiers.

Les contraintes liées à ce mode de fonctionnement sont très fortes dans les environnements embarqués, souvent assez limités en ressources telles que la mémoire :

- L'ensemble des classes utilisées doit être chargé, notamment les classes de base du langage Java, ce qui implique une perte de temps non négligeable.
- La place occupée en mémoire est considérable puisqu'aux fichiers en mémoire ROM, s'ajoutent des structures de données, formant la définition interne de la classe, ces structures de données se trouvant en mémoire vive RAM.

L'invention a pour objet une solution alternative à la solution décrite ci-dessus pour la mise à disposition de classes dans des environnements pauvres en mémoire et en puissance de traitement, en délocalisant le maximum du traitement effectué au chargement des classes utilisées au démarrage du système, et en réduisant au maximum les effets liés aux contraintes citées.

Exposé de l'invention

L'invention concerne un procédé de gestion de classes pour un langage orienté objet basé sur une résolution dynamique des classes d'objets, caractérisé en ce que l'on transforme la résolution d'au moins une classe dynamique, pour un groupe de classes autonomes, en résolution statique.

Avantageusement dans ledit procédé, utilisé dans une machine virtuelle Java pour un ensemble de classes autonomes, on inclut la structure native exploitée directement par le noyau, de ces classes en tant que données du binaire de la machine virtuelle (on se libère ainsi de la contrainte des fichiers). On fait une partie du travail de résolution, et on optimise les ressources mémoire du système.

Avantageusement dans ledit procédé, utilisé dans un système embarqué, on inclut la structure native, exploitée directement par le noyau Java, de ces classes en tant que données du binaire de la machine virtuelle sous une forme impliquant une optimisation (réduction du coût d'exploitation de l'ensemble des classes autonomes) des ressources mémoire du système ainsi qu'une optimisation en vitesse de l'exécution du bytecode des méthodes de classe.

On peut créer des fichiers source en langage natif décrivant les structures natives correspondant à l'ensemble autonome des classes.

On peut découper l'ensemble autonome de classes en sous-ensembles correspondant aux paquetages Java, ces sous-ensembles n'étant pas autonomes

individuellement, et établir des règles de nommage des symboles (nom+description de structures) de manière à pouvoir référencer une classe d'un paquetage depuis une classe d'un autre paquetage.

Dans le cadre de la description des méthodes et plus particulièrement dans le cadre de la description du bytecode des méthodes, on peut remplacer le bytecode nécessitant la résolution d'une entité par le bytecode réalisant la même fonctionnalité mais qui part du principe que l'entité est résolue.

L'invention concerne également un outil de génération des structures natives depuis les binaires Java mettant en œuvre le procédé.

Le procédé de l'invention trouve notamment son utilité dans des systèmes pauvres en ressource mémoire, tels que des décodeurs de télévision numérique.

Brève description des figures

La figure 1 illustre un processus de génération de binaire de l'art connu.

La figure 2 illustre un processus de génération de binaire selon l'invention.

La figure 3 illustre des structures associées aux classes Java.

Description détaillée de modes de réalisation de l'invention

La solution adoptée par le procédé de l'invention consiste à délocaliser le maximum du traitement effectué au chargement des classes utilisées au démarrage du système, en réduisant au maximum les effets liés aux contraintes citées.

Plutôt que d'intégrer les descriptifs de classe générés lors de l'étape de compilation Java, on intègre les descriptions des structures internes utilisées par le gestionnaire de classes. Le gestionnaire de classe, au démarrage du système, pré-charge toutes les classes définies de cette manière et exécute le code Java d'initialisation associé. Le chargement de ces classes se trouve simplifié pour le gestionnaire de classes, puisque les structures internes des classes sont déjà définies.

L'ensemble de la structure se retrouve localisée dans la mémoire ROM. Seules les informations susceptibles d'être modifiées à l'exécution d'applicatifs se retrouvent délocalisées dans la mémoire RAM.

Le procédé de l'invention, pour accomplir la création de la structure interne d'une classe, est tel que :

- Les structures internes des classes à utiliser par celle-ci sont connues lors de l'intégration du binaire, des références directes à ces

classes étant utilisées.

• Les structures des classes utilisées étant connues, on substitue les opérations demandant une résolution de classe par les opérations correspondantes se passant de la résolution de classe.

Les graphes illustrés sur les figures 1 et 2 définissent les étapes menant à la création du binaire final respectivement avec et sans utilisation du procédé de l'invention.

Les classes ainsi définies sont initialisées au démarrage du système Java. Elles sont regroupées par paquetage. Chacun des paquetages fournit la liste de ses classes ainsi que l'ensemble des paquetages possédant des classes utilisées par une des entités du paquetage. Ce regroupement en paquetages fournit, au moment de l'exploitation de l'invention, l'opportunité de faire des simplifications relatives aux affinités de classes d'un même paquetage, notamment sur la visibilité des propriétés. Ce regroupement présente également l'avantage de réduire les traitements lors de la mise en œuvre du procédé de l'invention.

Le procédé de l'invention permet donc de préparer les classes Java en vue d'une utilisation quasi-immédiate au démarrage du système.

L'invention concerne également l'outil de génération des structures natives depuis les binaires Java, qui permet de préparer ainsi les classes Java.

Compilation des sources Java/création des fichiers classe.

La compilation est le processus de transformation de code source décrivant un logiciel informatique en un ensemble d'opérations élémentaires directement comprises, c'est-à-dire interprétées, par la machine sur laquelle va s'exécuter ce programme. Dans le cas du langage Java, l'unité de base du code source est la classe Java. Le résultat de la compilation est un ensemble de fichiers, chaque fichier décrivant une classe, ainsi que ses méthodes. La machine virtuelle Java est l'environnement dans lequel se déroulent les programmes ainsi compilés.

L'outil de l'invention exploite ces structures de classes de manière à les préparer pour une intégration immédiate à la base de données des classes du gestionnaire de classes.

Intégration des fichiers classe au binaire

Les fichiers classes sont inclus dans le binaire de la machine virtuelle en tant que structures de données directement comprises par le gestionnaire de classes. Ces structures de données sont décrites par le biais de code source en langage C.

Outil de préparation des classes

Le code source est construit par un outil à même de comprendre les fichiers classes et familier avec les structures de données du gestionnaire de classes.

Cet outil travaille sur des fichiers

classes, tels que décrits par les spécifications du langage Java, interprète leur contenu, et fournit des structures de données compatibles avec celles du gestionnaire de classes de la machine virtuelle.

Exemple de mise en œuvre

La mise en œuvre de l'invention est très liée à la définition interne des structures de classes et entités associées.

On va donc considérer successivement les structures utilisées (base de données des classes), une manière de résoudre les contraintes liées à l'invention, et enfin un exemple de résultat produit sur application de l'invention

Base de données des classes

Le gestionnaire de classes maintient une base de données des classes chargées. Les structures, entités pertinentes dans le cadre de l'invention, sont décrites ci après ; il s'agit des structures de classes, des propriétés, ainsi que des tables de liens associées à chacune des classes.

Structure décrivant une classe

Diverses informations sont associées à chacune des classes, entre autres :

- Une référence vers la super-classe.
- La liste des méthodes de la classe (une liste de références vers des structures décrivant les méthodes).

- La liste des références vers les structures décrivant les attributs définis par la classe.
- Une table des références non résolues vers des classes, méthodes ou attributs nécessaire au bon fonctionnement des méthodes de la classe.

Structure décrivant un attribut

Chaque attribut d'une classe est décrit par une structure fournissant, entre autres, les informations suivantes :

- Le nom et la signature de l'attribut.
- Une référence vers la valeur de l'attribut.
- La visibilité de l'attribut.

Structure décrivant une méthode

Les méthodes d'une classe sont décrites par une structure définissant, entre autres, les informations suivantes, pour chacune des méthodes :

- Le nom et la signature de la méthode.
- La visibilité de la méthode.
- Le code de la méthode.

Table de références table/des constantes

La table de référence est définie par le nombre d'éléments, ainsi que par chacun de ses éléments.

Plus généralement, il s'agit de la table des constantes de la classe.

Chacun des éléments de la table est défini par l'ensemble d'informations suivant :

- Un indicateur de résolution de l'élément, indiquant si son accès nécessite une résolution préalable.
- Le type de l'élément.
- La référence vers la (définition de la) constante.

Résoudre une entrée de la table des constantes revient à retrouver la valeur de la constante et à substituer la référence vers la définition de la constante par la constante elle-même.

Un schéma récapitulatif des structures associées aux classes Java est illustré sur la figure 3.

Résolution des contraintes associées au procédé de l'invention

Les contraintes précédemment citées sont passées en revue et les solutions techniques associées sont décrites ci-dessous.

Les entités utilisées doivent être connues

Les entités utilisées par une classe (autres classes, propriétés,...) doivent être résolues de manière à pouvoir conserver la table des constantes d'une classe dans la mémoire ROM. Ces entités se trouvent dans la table des constantes.

Les constantes de la table sont initialement non résolues. A chacune des constantes sont associées les informations nécessaires pour la résoudre (pour une constante de type classe, le nom de

la classe est par exemple précisé). La résolution d'une constante se fait une seule fois pendant la durée de vie de la classe.

L'application du procédé de l'invention pour une classe passe par la résolution des entrées de la table des constantes.

Concernant les constantes de type classe, résoudre la classe revient au final à appliquer le procédé de l'invention à la classe.

Le code Java doit se baser sur les entités résolues

Certains bytecodes Java sont des opérations utilisant un argument dont la valeur se trouve dans la table des constantes. Pour des raisons de performances, les bytecodes nécessitant une résolution de constante sont associés à un bytecode équivalent réalisant la même fonctionnalité, mais partant du principe que la constante est déjà résolue, nommé, ci-dessous, "bytecode quick".

Après résolution avec succès d'une constante, au bytecode ayant demandé la résolution de la constante est substitué son bytecode quick.

L'outil de l'invention peut donc substituer les bytecodes utilisant des constantes à résoudre par des bytecode quick, puisque les entités utilisées doivent être résolues.

Symboles associés au code source généré

- Découpage en paquetage

L'invention est appliquée sur des

paquetages. Au paquetage sont associées les informations suivantes : un tableau de classes, une classe étant connue par son index dans le tableau.

Des constantes de compilation sont définies, ces constantes établissent le lien « index d'une classe dans le tableau <-> structure de classe ». Le nom de symbole associé au tableau est construit à partir du nom du paquetage.

- Référence sur une classe

La référence à une classe d'un autre paquetage se fait alors sous la forme : & Tableaupaquetage[INDEX_DE_LA_CLASSE], qui est l'adresse de l'élément INDEX_DE_LA_CLASSE dans le tableau.

- Référence sur les méthodes d'une classe

De même que pour les classes d'un paquetage, les méthodes sont regroupées dans un tableau. Le tableau des méthodes est accessible à partir d'un symbole, dont le nom est associé au nom de la classe.

- Référence sur les attributs d'une classe

Les attributs d'une classe sont aussi regroupés en une liste. Cette liste se présente sous la forme d'un tableau. Le nom de symbole associé au tableau est fonction du nom de la classe.

Exemple de résultat produit sur application du procédé de l'invention

Description des classes Java

L'architecture de classes Java utilisée pour l'exemple est la suivante :

Les classes appartiennent au paquetage test.brevet. Pour donner un exemple concret mais simple, deux classes sont utilisées : Classe1 et Classe2, ainsi qu'une interface Interface1, telles que :

- Classe1 met en œuvre l'interface Interface1 et hérite de la classe Java.lang.Object, classe dont héritent toutes les autres classes du système.
- Classe2 hérite de Classe1.
- Interface1 décrit une méthode methode1 qui est définie dans les classes (Classe1 et Classe2) mettant en œuvre cette interface.
- Classe1 définit deux méthodes : la méthode String toString() surcharge celle de la classe Java.lang.Object ; la méthode methode1() affiche un petit message sur la sortie texte de la machine virtuelle.
- Classe2 définit deux méthodes : la méthode methode1() surcharge celle de Classe1 et affiche un autre petit message ; la méthode methode2 affiche encore un autre message sur la sortie texte.

Les définitions des classes et interfaces sont les suivantes :

fichier source Interface1.java

```
package test. Brevet ;
public interface Interface1 {
public void methode();
}

fichier source Classe1.java

package test. brevet ;
public class Classe1 extends java.lang.Object
implements Interface1 { public String toString() {
return « Classe1 »
}
public void methode1() {
System.out.println(«un petit message»)
}
}

fichier source Classe2.java

package test. brevet
public class Classe2 extends Classe1 {
      public void methode1() {
        System.out.println(«un autre petit message»);
      }
}
```

Structures produites

Le résultat de l'application du procédé de l'invention est défini ci-dessous : ici, il s'agit de la transformation des sources Java précédemment

définies, après leur compilation, en code source dans le langage C décrivant les structures correspondantes de la base de données des classes de la machine virtuelle.

Définition externes

```
#include <MhwTypes.h>
#include <MhwMemory.h>
#include <MhwObject.h>
#include <MhwClassF.h>
#include <MhwInterpreterF.h>
#include <MhwClassRom.h>
#include <java/lang/stubs/java_lang_String.h>
#include <test/brevet/stubs/ClassRom.h>
#include <java/lang/stubs/ClassRom.h>
#include <test/brevet/stubs/ClassRom.h>
```

Chaînes de caractère

```
static const char cString_0[] = "<init>";
static const char cString_1[] = "()V";
static const char cString_2[] = "methode1";
static const char cString_3[] = "toString";
static const char cString_4[] = "()Ljava/lang/String;";
static const char cString_7[] = "test/brevet/Classe1";
static const char cString_8[] = "Classe1.java";
static const char cString_10[] = "test/brevet/Classe2";
static const char cString_11[] = "Classe2.java";
static const char cString_12[] =
"test/brevet/Interface1";
static const char cString_13[] = "Interface1.java";
#ifndef CLASSROM_OPTIMIZE
static const char cString_5[] = "this";
static const char cString_6[] =
"Ltest/brevet/Classe1;";
static const char cString_9[] =
"Ltest/brevet/Classe2;";
#endif /* CLASSROM_OPTIMIZE */
```

Bytecode de la classe Classe1

```
static CLASSROM_CONST Card8 byteCode_0[] =
```

```
{
  /*    0 */ 0x2A, 0xDC, 0x00, 0x08, 0xB1
};

#ifndef CLASSROM_OPTIMIZE
static const MhwClmLineno Lineno_0[] =
{
  { /* 0 */
    0,
    32
  }
};
#endif /* CLASSROM_OPTIMIZE */


#ifndef CLASSROM_OPTIMIZE
static const MhwClmLocalVar LocalVar_0[] =
{
  { /* 0 */
    0,
    5,
    5,
    6,
    0
  }
};
#endif /* CLASSROM_OPTIMIZE */


static CLASSROM_CONST Card8 byteCode_1[] =
{
  /*    0 */ 0xD2, 0x00, 0x02, 0xCB, 0x03, 0xD6, 0x1D,
0x02, 0xB1
};

#ifndef CLASSROM_OPTIMIZE
static const MhwClmLineno Lineno_1[] =
{
  { /* 0 */
    0,
    37
  },
  { /* 1 */
    8,
    36
  }
};
```

```
#endif /* CLASSROM_OPTIMIZE */


#ifndef CLASSROM_OPTIMIZE
static const MhwClmLocalVar LocalVar_1[] =
{
  { /* 0 */
    0,
    9,
    5,
    6,
    0
  }
};
#endif /* CLASSROM_OPTIMIZE */


static CLASSROM_CONST Card8 byteCode_2[] =
{
  /*   0 */ 0xCB, 0x01, 0xB0
};

#ifndef CLASSROM_OPTIMIZE
static const MhwClmLineno Lineno_2[] =
{
  { /* 0 */
    0,
    34
  }
};
#endif /* CLASSROM_OPTIMIZE */


#ifndef CLASSROM_OPTIMIZE
static const MhwClmLocalVar LocalVar_2[] =
{
  { /* 0 */
    0,
    3,
    5,
    6,
    0
  }
};
#endif /* CLASSROM_OPTIMIZE */
```

Données de la table des constantes/références de la classe Classe1

```
static const Card16 character_0[] =
{
  /*   0 */ 0x0043, 0x006C, 0x0061, 0x0073, 0x0073,
0x0065, 0x0031
};

static const HArrayOfChar arrayOfChar_0 =
{
  (const ArrayOfChar)character_0,
  (void*)(0x7055)
};

static const Classjava_lang_String stringStruct_0 =
{
  (struct HArrayOfChar *)&arrayOfChar_0,
  0,
  7
};

static const Hjava_lang_String string_0 =
{
  (Classjava_lang_String*)&stringStruct_0,
  (void*)(&java_lang_String_MethodTable)
};

static const Card16 character_1[] =
{
  /*   0 */ 0x0075, 0x006E, 0x0020, 0x0070, 0x0065,
0x0074, 0x0069, 0x0074, 0x0020, 0x006D,
  /*  10 */ 0x0065, 0x0073, 0x0073, 0x0061, 0x0067,
0x0065
};

static const HArrayOfChar arrayOfChar_1 =
{
  (const ArrayOfChar)character_1,
  (void*)(0x10055)
};

static const Classjava_lang_String stringStruct_1 =
{
  (struct HArrayOfChar *)&arrayOfChar_1,
  0,
  16
```

```
};

static const Hjava_lang_String string_1 =
{
  (Classjava_lang_String*)&stringStruct_1,
  (void*)(&java_lang_String_MethodTable)
};

static const Hjava_lang_Class
test_brevet_Classe1_Handle =
{

(Classjava_lang_Class*)&(test_brevet_Class_Entry[TEST_B
REVET_CLASSE1]),
  (void*)&java_lang_Class_MethodTable
};
```

Table des méthodes de la classe Classe1

```
static const struct
{
  const MhwClmClass *classdescriptor;
  const MhwClmMethod *methods[13];
} test_brevet_Classe1_MethodTable =
{
  (const
MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLAS
SE1]),
  {
    NULL,
    (MhwClmMethod*)&(java_lang_Object_Method[2]) /* 1
*/,
    (MhwClmMethod*)&(test_brevet_Classe1_Method[2]) /*
2 */,
    (MhwClmMethod*)&(java_lang_Object_Method[11]) /* 3
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[9]) /* 4
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[3]) /* 5
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[10]) /* 6
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[4]) /* 7
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[5]) /* 8
*/,
```

```
    (MhwClmMethod*)&(java_lang_Object_Method[6]) /* 9
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[7]) /* 10
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[1]) /* 11
*/,
    (MhwClmMethod*)&(test_brevet_Classe1_Method[1]) /*
12 */
  }
};

static const Card16 interfacesImplemented_0[] =
{
  /*    0 */ 0x0004
};
```

Bytecode de la classe Classe2

```
static CLASSROM_CONST Card8 byteCode_3[] =
{
  /*    0 */ 0x2A, 0xD7, 0x00, 0x03, 0xB1
};

#ifndef CLASSROM_OPTIMIZE
static const MhwClmLineno Lineno_3[] =
{
  { /* 0 */
    0,
    32
  }
};
#endif /* CLASSROM_OPTIMIZE */


#ifndef CLASSROM_OPTIMIZE
static const MhwClmLocalVar LocalVar_3[] =
{
  { /* 0 */
    0,
    5,
    4,
    5,
    0
  }
};
#endif /* CLASSROM_OPTIMIZE */
```

```
static CLASSROM_CONST Card8 byteCode_4[] =
{
  /*   0 */ 0xD2, 0x00, 0x01, 0xCB, 0x02, 0xD6, 0x1D,
0x02, 0xB1
};

#ifndef CLASSROM_OPTIMIZE
static const MhwClmLineno Lineno_4[] =
{
  { /* 0 */
    0,
    34
  },
  { /* 1 */
    8,
    33
  }
};
#endif /* CLASSROM_OPTIMIZE */


#ifndef CLASSROM_OPTIMIZE
static const MhwClmLocalVar LocalVar_4[] =
{
  { /* 0 */
    0,
    9,
    4,
    5,
    0
  }
};
#endif /* CLASSROM_OPTIMIZE */
```

Données de la table des constantes/références de la classe Classe2

```
static const Card16 character_2[] =
{
  /*   0 */ 0x0075, 0x006E, 0x0020, 0x0061, 0x0075,
0x0074, 0x0072, 0x0065, 0x0020, 0x0070,
  /*  10 */ 0x0065, 0x0074, 0x0069, 0x0074, 0x0020,
0x006D, 0x0065, 0x0073, 0x0073, 0x0061,
  /*  20 */ 0x0067, 0x0065
```

```
};

static const HArrayOfChar arrayOfChar_2 =
{
  (const ArrayOfChar)character_2,
  (void*)(0x16055)
};

static const Classjava_lang_String stringStruct_2 =
{
  (struct HArrayOfChar *)&arrayOfChar_2,
  0,
  22
};

static const Hjava_lang_String string_2 =
{
  (Classjava_lang_String*)&stringStruct_2,
  (void*)(&java_lang_String_MethodTable)
};

static const Hjava_lang_Class
test_brevet_Classe2_Handle =
{
(Classjava_lang_Class*)&(test_brevet_Class_Entry[TEST_B
REVET_CLASSE2]),
  (void*)&java_lang_Class_MethodTable
};
```

Table des méthodes de la classe Classe2

```
static const struct
{
  const MhwClmClass *classdescriptor;
  const MhwClmMethod *methods[13];
} test_brevet_Classe2_MethodTable =
{
  (const
MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLAS
SE2]),
  {
    NULL,
    (MhwClmMethod*)&(java_lang_Object_Method[2]) /* 1
*/,
```

```
    (MhwClmMethod*)&(test_brevet_Classe1_Method[2]) /*
2 */,
    (MhwClmMethod*)&(java_lang_Object_Method[11]) /* 3
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[9]) /* 4
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[3]) /* 5
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[10]) /* 6
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[4]) /* 7
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[5]) /* 8
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[6]) /* 9
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[7]) /* 10
*/,
    (MhwClmMethod*)&(java_lang_Object_Method[1]) /* 11
*/,
    (MhwClmMethod*)&(test_brevet_Classe2_Method[1]) /*
12 */
  }
};

static const Hjava_lang_Class
test_brevet_Interface1_Handle =
{

(Classjava_lang_Class*)&(test_brevet_Class_Entry[TEST_B
REVET_INTERFACE1]),
  (void*)&java_lang_Class_MethodTable
};
```

Table des méthodes de l'interface Interface1

```
static const struct
{
  const MhwClmClass *classdescriptor;
  const MhwClmMethod *methods[13];
} test_brevet_Interface1_MethodTable =
{
  (const
MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_INTE
RFACE1]),
  {
```

```
        NULL,
        (MhwClmMethod*)&(java_lang_Object_Method[2]) /* 1
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[8]) /* 2
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[11]) /* 3
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[9]) /* 4
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[3]) /* 5
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[10]) /* 6
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[4]) /* 7
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[5]) /* 8
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[6]) /* 9
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[7]) /* 10
    */,
        (MhwClmMethod*)&(java_lang_Object_Method[1]) /* 11
    */,
        (MhwClmMethod*)&(test_brevet_Interface1_Method[0])
    /* 12 */
      }
    };
```

Définition des méthodes de la classe Classe1

```
    const MHW_CLM_METHOD_STRUCT_TYPE
    test_brevet_Classe1_Method[] =
    {
      { /* 0 */
        {

    (MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLA
    SSE1]), /* class */
          (char*)cString_0, /* name : <init> */
          (char*)cString_1, /* type : ()V */
          0|ACC_PUBLIC,
          3409, /* hashcode */
          0 /* offset */
        },
        5, /* code length */
        (Card8*)byteCode_0,
```

```
        MhwIntInvokeJavaMethod,
        NULL, /* catch table */
        0, /* catch count */
        1, /* arg size */
        1, /* max stack */
        1 /* nlocals */
        ,CLASSROM_LINENO_TABLE(Lineno_0),
        CLASSROM_LINENO_COUNT(1),
        CLASSROM_LOCALVAR_COUNT(1),
        CLASSROM_LOCALVAR_TABLE(LocalVar_0),
        NULL /* thrown exceptions */
      },
      { /* 1 */
        {
(MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLA
SSE1]), /* class */
          (char*)cString_2, /* name : methode1 */
          (char*)cString_1, /* type : ()V */
          0|ACC_PUBLIC,
          4247, /* hashcode */
          12 /* offset */
        },
        9, /* code length */
        (Card8*)byteCode_1,
        MhwIntInvokeJavaMethod,
        NULL, /* catch table */
        0, /* catch count */
        1, /* arg size */
        2, /* max stack */
        1 /* nlocals */
        ,CLASSROM_LINENO_TABLE(Lineno_1),
        CLASSROM_LINENO_COUNT(2),
        CLASSROM_LOCALVAR_COUNT(1),
        CLASSROM_LOCALVAR_TABLE(LocalVar_1),
        NULL /* thrown exceptions */
      },
      { /* 2 */
        {
(MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLA
SSE1]), /* class */
          (char*)cString_3, /* name : toString */
          (char*)cString_4, /* type : ()Ljava/lang/String;
*/
          0|ACC_PUBLIC,
          2189, /* hashcode */
```

```
        2 /* offset */
      },
      3, /* code length */
      (Card8*)byteCode_2,
      MhwIntInvokeJavaMethod,
      NULL, /* catch table */
      0, /* catch count */
      1, /* arg size */
      1, /* max stack */
      1 /* nlocals */
      ,CLASSROM_LINENO_TABLE(Lineno_2),
      CLASSROM_LINENO_COUNT(1),
      CLASSROM_LOCALVAR_COUNT(1),
      CLASSROM_LOCALVAR_TABLE(LocalVar_2),
      NULL /* thrown exceptions */
    }
  };
```

Table des constantes/références de la classe Classe1

```
static const void* test_brevet_Classe1_CP[] =
{
  (const void*)NULL,
  /*  1 */ (const void*)&string_0,
  /*  2 */ (const void*)&(java_lang_System_Field[2]),
  /*  3 */ (const void*)&string_1,
  /*  4 */ (const
void*)&(test_brevet_Class_Entry[TEST_BREVET_INTERFACE1]
)
#ifndef CLASSROM_OPTIMIZE
,
  /*  5 */ (const void*)cString_5 /* C String */,
  /*  6 */ (const void*)cString_6 /* C String */
#endif /* CLASSROM_OPTIMIZE */
};
```

Définition des méthodes de la classe Classe2

```
const MHW_CLM_METHOD_STRUCT_TYPE
test_brevet_Classe2_Method[] =
{
  { /* 0 */
    {
```

```
    (MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLA
    SSE2]), /* class */
          (char*)cString_0, /* name : <init> */
          (char*)cString_1, /* type : ()V */
          0|ACC_PUBLIC,
          3409, /* hashcode */
          0 /* offset */
        },
        5, /* code length */
        (Card8*)byteCode_3,
        MhwIntInvokeJavaMethod,
        NULL, /* catch table */
        0, /* catch count */
        1, /* arg size */
        1, /* max stack */
        1 /* nlocals */
        ,CLASSROM_LINENO_TABLE(Lineno_3),
        CLASSROM_LINENO_COUNT(1),
        CLASSROM_LOCALVAR_COUNT(1),
        CLASSROM_LOCALVAR_TABLE(LocalVar_3),
        NULL /* thrown exceptions */
      },
      { /* 1 */
        {

    (MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLA
    SSE2]), /* class */
          (char*)cString_2, /* name : methode1 */
          (char*)cString_1, /* type : ()V */
          0|ACC_PUBLIC,
          4247, /* hashcode */
          12 /* offset */
        },
        9, /* code length */
        (Card8*)byteCode_4,
        MhwIntInvokeJavaMethod,
        NULL, /* catch table */
        0, /* catch count */
        1, /* arg size */
        2, /* max stack */
        1 /* nlocals */
        ,CLASSROM_LINENO_TABLE(Lineno_4),
        CLASSROM_LINENO_COUNT(2),
        CLASSROM_LOCALVAR_COUNT(1),
        CLASSROM_LOCALVAR_TABLE(LocalVar_4),
        NULL /* thrown exceptions */
```

```
  }
};
```

Table des constantes/références de la classe Classe2

```
static const void* test_brevet_Classe2_CP[] =
{
  (const void*)NULL,
  /*  1 */ (const void*)&(java_lang_System_Field[2]),
  /*  2 */ (const void*)&string_2,
  /*  3 */ (const
void*)&(test_brevet_Classe1_Method[0])
#ifndef CLASSROM_OPTIMIZE
,
  /*  4 */ (const void*)cString_5 /* C String */,
  /*  5 */ (const void*)cString_9 /* C String */
#endif /* CLASSROM_OPTIMIZE */
};
```

Définition des méthodes de l'interface Interface1

```
const MHW_CLM_METHOD_STRUCT_TYPE
test_brevet_Interface1_Method[] =
{
  { /* 0 */
    {

(MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_INT
ERFACE1]), /* class */
      (char*)cString_2, /* name : methode1 */
      (char*)cString_1, /* type : ()V */
      0|ACC_PUBLIC|ACC_ABSTRACT,
      4247, /* hashcode */
      12 /* offset */
    },
    0, /* code length */
    NULL, /* byte code */
    MhwIntInvokeAbstractMethod,
    NULL, /* catch table */
    0, /* catch count */
    1, /* arg size */
    0, /* max stack */
    0 /* nlocals */
    ,NULL, /* lineno table */
    0, /* lineno count */
```

```
        0, /* localvar count */
        NULL, /* localvar table */
        NULL /* thrown exceptions */
      }
};
```

Définition des classes du paquetage

```
const MhwClmClassRom test_brevet_Class_Entry[] =
{
  { /* 0 */
    (char*)cString_7, /* name : test/brevet/Classe1 */

(MhwClmClass*)&(java_lang_Class_Entry[JAVA_LANG_OBJECT]
), /* superclass */
    92, /* hashcode */
    CLASSROM_ACCESS(CLASS_ROM|CLASS_PUBLIC),
    0, /* object size */
    NULL, /* outerclass */
    NULL, /* adrStaticFields */
    0, /* sizeStaticFields */
    NULL, /* fields */
    CLASSROM_FIELD_COUNT(0,0,0),
    3, /* method count */
    (MhwClmMethod*)test_brevet_Classe1_Method,
    (MhwClmConstantPool*)test_brevet_Classe1_CP,
    NULL, /* class loader */
    NULL, /* clinit */
    NULL, /* finalizer */
    (Hjava_lang_Class*)&test_brevet_Classe1_Handle, /*
class handle */

(MhwClmMethodTable*)&test_brevet_Classe1_MethodTable,
    13, /* methodtable size */
    1, /* interface count */
    (Card16*)interfacesImplemented_0,
    (char*)cString_8, /* source name : Classe1.java */
    NULL, /* protection domain */
    NULL, /* super class name */
    NULL, /* strings */
    (Card16) 3, /* major_version */
    (Card16) 3, /* minor_version */
    (Card16) 7, /* cp_count */
    (Card8) 29 /* status */
  },
  { /* 1 */
```

```
        (char*)cString_10, /* name : test/brevet/Classe2 */

    (MhwClmClass*)&(test_brevet_Class_Entry[TEST_BREVET_CLA
    SSE1]), /*superclass */
        93, /* hashcode */
        CLASSROM_ACCESS(CLASS_ROM|CLASS_PUBLIC),
        0, /* object size */
        NULL, /* outerclass */
        NULL, /* adrStaticFields */
        0, /* sizeStaticFields */
        NULL, /* fields */
        CLASSROM_FIELD_COUNT(0,0,0),
        2, /* method count */
        (MhwClmMethod*)test_brevet_Classe2_Method,
        (MhwClmConstantPool*)test_brevet_Classe2_CP,
        NULL, /* class loader */
        NULL, /* clinit */
        NULL, /* finalizer */
        (Hjava_lang_Class*)&test_brevet_Classe2_Handle, /*
    class handle */

    (MhwClmMethodTable*)&test_brevet_Classe2_MethodTable,
        13, /* methodtable size */
        0, /* interface count */
        NULL, /* interfaces implemented */
        (char*)cString_11, /* source name : Classe2.java */
        NULL, /* protection domain */
        NULL, /* super class name */
        NULL, /* strings */
        (Card16) 3, /* major_version */
        (Card16) 3, /* minor_version */
        (Card16) 6, /* cp_count */
        (Card8) 29 /* status */
      },
      { /* 2 */
        (char*)cString_12, /* name : test/brevet/Interface1
    */

    (MhwClmClass*)&(java_lang_Class_Entry[JAVA_LANG_OBJECT]
    ), /* superclass */
        394, /* hashcode */

    CLASSROM_ACCESS(CLASS_ROM|CLASS_PUBLIC|CLASS_INTERFACE|
    CLASS_ABSTRACT),
        0, /* object size */
        NULL, /* outerclass */
        NULL, /* adrStaticFields */
```

```
        0, /* sizeStaticFields */
        NULL, /* fields */
        CLASSROM_FIELD_COUNT(0,0,0),
        1, /* method count */
        (MhwClmMethod*)test_brevet_Interface1_Method,
        NULL, /* constant pool */
        NULL, /* class loader */
        NULL, /* clinit */
        NULL, /* finalizer */
        (Hjava_lang_Class*)&test_brevet_Interface1_Handle,
    /* class handle */

    (MhwClmMethodTable*)&test_brevet_Interface1_MethodTable
    ,
        13, /* methodtable size */
        0, /* interface count */
        NULL, /* interfaces implemented */
        (char*)cString_13, /* source name : Interface1.java
    */
        NULL, /* protection domain */
        NULL, /* super class name */
        NULL, /* strings */
        (Card16) 3, /* major_version */
        (Card16) 3, /* minor_version */
        (Card16) 1, /* cp_count */
        (Card8) 29 /* status */
      }
    };
```

Informations diverses

```
const Card32 test_brevet_ClassNumber = 3;

const Card32 test_brevet_CheckSum = 773706915;

const Card32 test_brevet_PackageUsedNumber = 2;

const PackageUsedStruct test_brevet_PackageUsedArray[]
=
{
  {
    JAVA_LANG_CHECKSUM_VALUE,
    &java_lang_CheckSum,
    "java/lang"
  },
  {
```

```
        TEST_BREVET_CHECKSUM_VALUE,
        &test_brevet_CheckSum,
        "test/brevet"
      }
  };
```

REVENDICATIONS

1. Procédé de gestion de classes pour un langage orienté objet basé sur une résolution dynamique des classes objets, caractérisé en ce que l'on transforme la résolution d'au moins une classe dynamique, pour un groupe de classes autonomes, en résolution statique.

2. Procédé selon la revendication 1, utilisé dans une machine virtuelle Java pour un ensemble de classes autonomes, dans lequel on inclut la structure native exploitée directement par le noyau Java de ces classes en tant que données du binaire de la machine virtuelle.

3. Procédé selon la revendication 2 utilisé dans un système embarqué, dans lequel on inclut la structure native de ces classes en tant que données du binaire de la machine virtuelle sous une forme impliquant une optimisation des ressources mémoire du système ainsi qu'une optimisation en vitesse de l'exécution du bytecode des méthodes de classe.

4. Procédé selon la revendication 3, dans lequel on crée des fichiers source en langage natif décrivant les structures natives correspondant à l'ensemble autonome des classes.

5. Procédé selon la revendication 4, dans lequel on découpe l'ensemble autonome de classes en

sous-ensembles correspondant aux paquetages Java, ces sous-ensembles n'étant pas autonomes individuellement et on établit des règles de nommage des symboles de manière à pouvoir référencer une classe d'un paquetage depuis une classe d'un autre paquetage.

6. Procédé selon la revendication 5, dans lequel dans le cadre de la description des méthodes et plus particulièrement dans le cadre de la description du bytecode des méthodes, on remplace le bytecode nécessitant la résolution d'une entité par le bytecode réalisant la même fonctionnalité mais qui part du principe que l'entité est résolue.

7. Outil de génération des structures natives depuis les binaires objets, mettant en œuvre le procédé selon l'une quelconque des revendications précédentes.

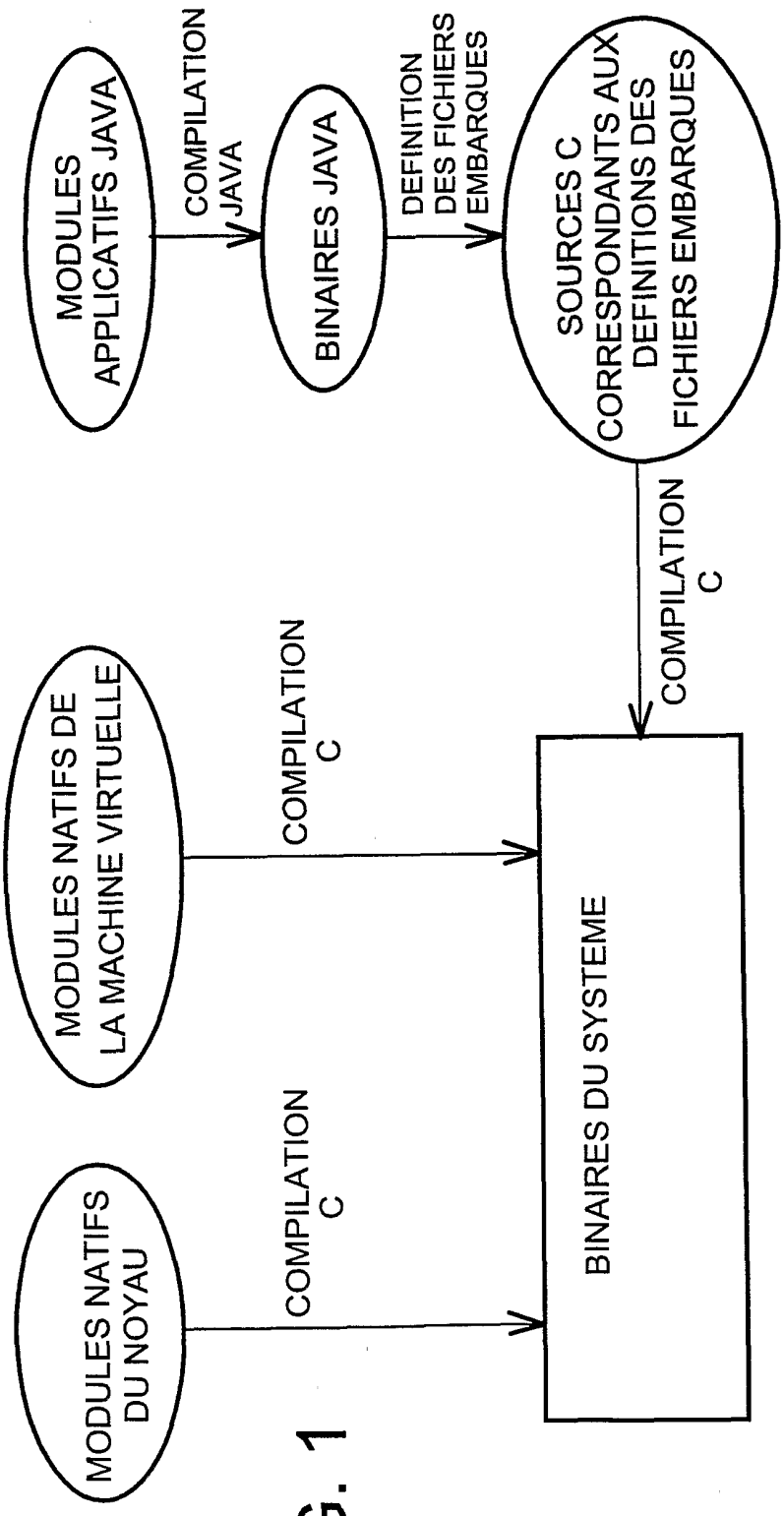

MODULES APPLICATIFS JAVA
COMPILATION JAVA
BINAIRES JAVA
DEFINITION DES FICHIERS EMBARQUES
SOURCES C CORRESPONDANTS AUX DEFINITIONS DES FICHIERS EMBARQUES
COMPILATION C
MODULES NATIFS DE LA MACHINE VIRTUELLE
COMPILATION C
MODULES NATIFS DU NOYAU
COMPILATION C
BINAIRES DU SYSTEME

FIG. 1

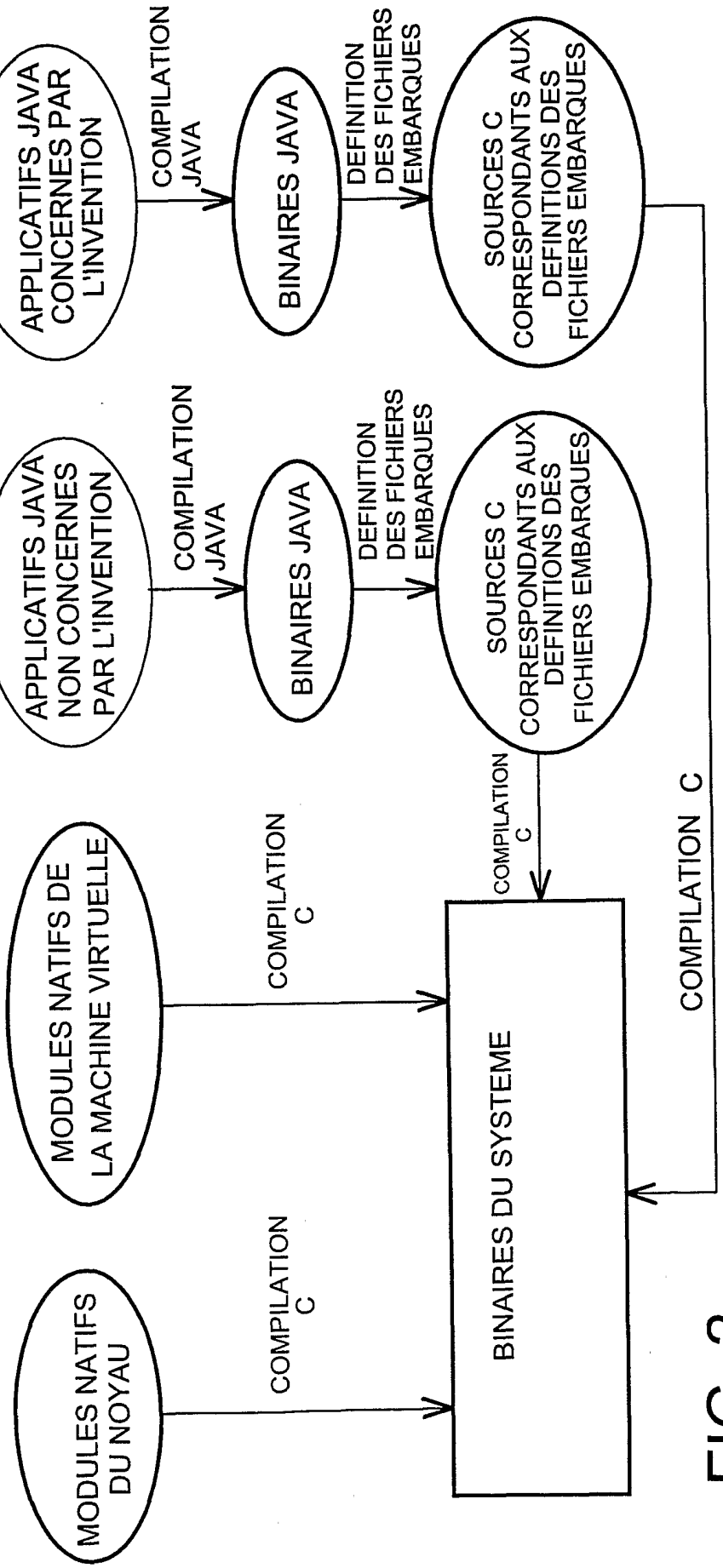

APPLICATIFS JAVA CONCERNES PAR L'INVENTION
COMPILATION JAVA
BINAIRES JAVA
DEFINITION DES FICHIERS EMBARQUES
SOURCES C CORRESPONDANTS AUX DEFINITIONS DES FICHIERS EMBARQUES
APPLICATIFS JAVA NON CONCERNES PAR L'INVENTION
COMPILATION JAVA
BINAIRES JAVA
DEFINITION DES FICHIERS EMBARQUES
SOURCES C CORRESPONDANTS AUX DEFINITIONS DES FICHIERS EMBARQUES
COMPILATION C
MODULES NATIFS DE LA MACHINE VIRTUELLE
COMPILATION C
MODULES NATIFS DU NOYAU
COMPILATION C
BINAIRES DU SYSTEME
COMPILATION C

FIG. 2

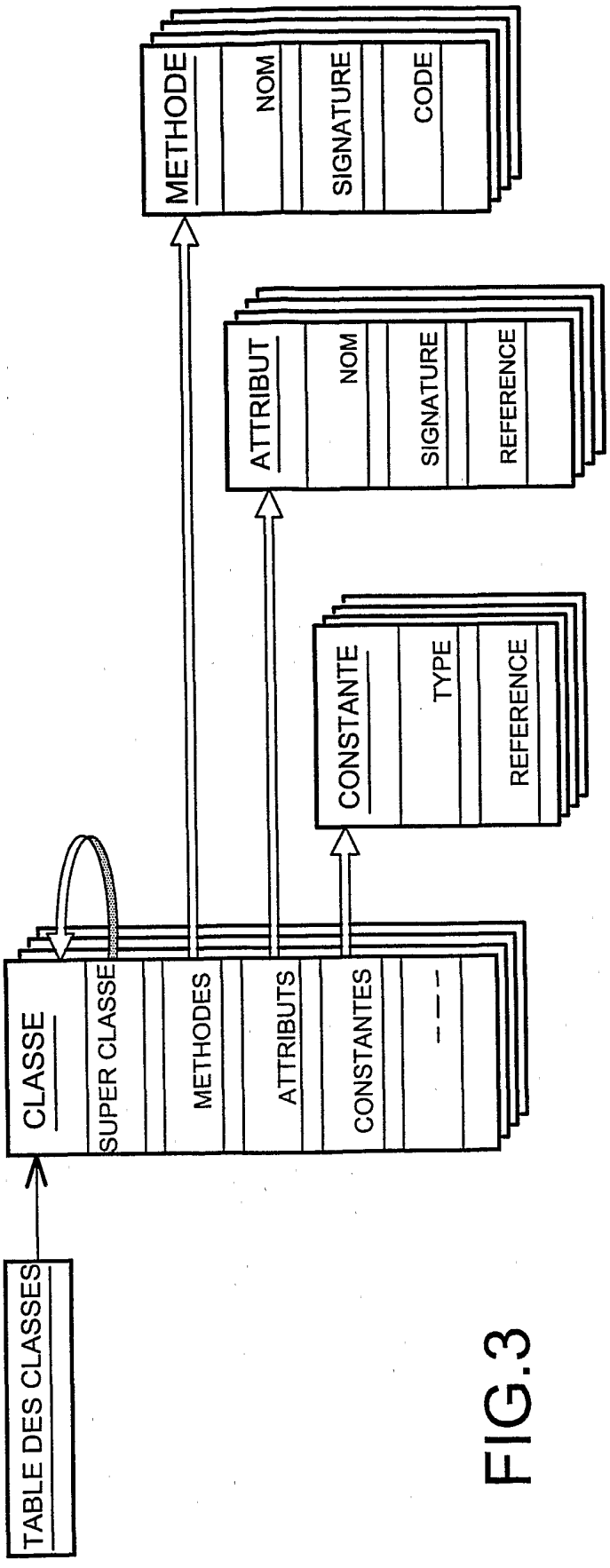


TABLE DES CLASSES
CLASSE
SUPER CLASSE
METHODES
ATTRIBUTS
CONSTANTES
- - - -
METHODE
NOM
SIGNATURE
CODE
ATTRIBUT
NOM
SIGNATURE
REFERENCE
CONSTANTE
TYPE
REFERENCE

FIG.3