

Rapport IRIT/2004-5-R

# JAVACT 0.5.0 : principes, installation, utilisation et développement d'applications

J.-P. Arcangeli\*,  
V. Hennebert\*\*,  
S. Leriche\*,  
F. Migeon\*,  
M. Pantel\*\*

Institut de Recherche en Informatique de Toulouse

{arcangel, leriche, migeon}@irit.fr, {henneber, pantel}@enseeiht.fr

\* IRIT-UPS, 118 route de Narbonne 31062 Toulouse Cedex 4

\*\* IRIT-ENSEEIHT, 2 rue Camichel BP 7122 31071 Toulouse Cedex 7

# Sommaire

<b>1</b>	<b>Principes généraux</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	Les agents mobiles . . . . .	3
1.2.1	Principes et avantages . . . . .	3
1.2.2	Propriétés . . . . .	3
1.3	Les acteurs . . . . .	4
1.3.1	Le modèle de calcul . . . . .	4
1.3.2	La validation des applications . . . . .	4
1.3.3	La mobilité d'acteur . . . . .	5
<b>2</b>	<b>Installation</b>	<b>6</b>
2.1	Vérifions les pré-requis . . . . .	6
2.2	C'est parti! . . . . .	6
2.3	Organisation des fichiers . . . . .	7
<b>3</b>	<b>Utilisation</b>	<b>8</b>
3.1	Lancement des systèmes d'accueil . . . . .	8
3.2	Exécution d'un programme . . . . .	8
3.3	Compilation de la bibliothèque . . . . .	9
<b>4</b>	<b>Comment développer une application</b>	<b>10</b>
4.1	L'incontournable « Hello World » . . . . .	10
4.1.1	Profils des comportements et des acteurs . . . . .	10
4.1.2	Génération automatique . . . . .	11
4.1.3	Implantation des comportements . . . . .	12
4.1.4	Et pour finir... . . . . .	12
4.2	La cellule . . . . .	14
4.3	Le crible d'Eratosthène . . . . .	15
4.4	Un agent mobile de recherche d'information . . . . .	18
4.5	Remarques complémentaires . . . . .	23
<b>A</b>	<b>Un exemple de message généré : JAMprint.java</b>	<b>26</b>
<b>B</b>	<b>Un exemple de comportement généré : EmptyQuasiBehavior.java</b>	<b>27</b>
<b>C</b>	<b>Primitives pour la programmation de comportements</b>	<b>28</b>
C.1	Création d'un acteur . . . . .	28
C.2	Envoi de message . . . . .	28
C.3	Changement de comportement . . . . .	28
C.4	Déplacement de l'acteur . . . . .	29
C.5	Localisation et auto-référence de l'acteur . . . . .	29
C.6	Adaptation des micro-composants . . . . .	29

JAVACT est un intergiciel (ou *middleware*) Java pour la programmation d'applications concurrentes, réparties et mobiles [AMM01], développé par l'équipe IAM (Ingénierie des Applications Mobiles<sup>1</sup>) de l'IRIT (Institut de Recherche en Informatique de Toulouse, France) et distribué sous forme de logiciel libre sous licence LGPL.

JAVACT fournit des mécanismes pour la programmation d'agents mobiles adaptables qui s'appuient sur les concepts d'acteur et d'implémentation ouverte<sup>2</sup>. JAVACT étend le paquetage *java.rmi* (RMI : *Remote Method Invocation*<sup>3</sup>) dont il utilise les mécanismes de passage d'objets en argument et d'exception, et l'adapte aux communications asynchrones au moyen d'un mécanisme de type *send-receive*. Outre l'abstraction par rapport à *java.rmi*, JAVACT permet au programmeur de s'abstraire de la gestion de mécanismes de bas niveau tel les *threads*, leur ordonnancement et la synchronisation.

JAVACT est une bibliothèque Java standard : son utilisation ne demande pas de traitement particulier au niveau du code source ou du *bytecode*, ni de modification de la machine virtuelle. Dans sa version actuelle, JAVACT repose sur le *Software Development Kit 1.4*<sup>4</sup> (le *JRE 1.3* suffit pour l'exécution).

Ce document explique comment développer une application au moyen de JAVACT (version 0.5.0) et décrit :

- les principes généraux sur lesquels s'appuie la bibliothèque (section 1) ;
- la procédure d'installation (section 2) ;
- l'utilisation de JAVACT et l'exécution d'un programme (section 3) ;
- le processus de développement et les codes Java de différents exemples (section 4).

## 1 Principes généraux

### 1.1 Introduction

Les réseaux à grande échelle, tel l'Internet, et les grilles de calcul ou de stockage donnent accès à des quantités de données, de services, de ressources logicielles et matérielles répartis. Dans un tel contexte, qui touche également l'informatique omniprésente (*ubiquitous computing*) et l'informatique nomade, le développement et le déploiement d'applications demande des méthodes et des outils adaptés permettant de traiter les problèmes qui résultent de l'hétérogénéité (des données, du logiciel, du matériel), de la répartition des ressources et des utilisateurs, des volumes de données déplacés sur le réseau, de la concurrence, de la sécurité, etc. En particulier, les applications doivent faire face à de fortes variations des conditions d'exécution (capacités physiques des réseaux et des machines, disponibilité des composants) et il n'est pas possible de faire d'hypothèse sur la qualité des services requis ou rendus. Ainsi, les applications doivent être adaptables, si possible de manière dynamique. Dans ce cadre, les concepts d'agents, de mobilité, et d'implémentation ouverte semblent du plus grand intérêt.

Les intergiciels « classiques » permettent aux développeurs de s'abstraire de certains aspects techniques, parfois complexes, liés à la répartition et aux opérations distantes. Cependant, dans le contexte du calcul réparti à grande échelle et de l'informatique mobile, il est nécessaire, lors du développement, de prendre en compte la qualité des services et de s'y adapter. Afin de limiter la complexité du développement et de la maintenance et de faciliter la construction d'applications robustes capables de s'adapter dynamiquement aux conditions d'exécution, nous proposons JAVACT, un intergiciel de haut niveau à base d'agents mobiles adaptables.

---

<sup>1</sup><http://www.irit.fr/recherches/ISPR/IAM/>

<sup>2</sup>Les aspects concernant l'adaptation et l'ouverture de l'implémentation [LA04] ne sont pas décrits dans ce document

<sup>3</sup><http://java.sun.com/products/jdk/rmi>

<sup>4</sup><http://java.sun.com/j2se/>

## 1.2 Les agents mobiles

La programmation par agents mobiles est un paradigme de programmation des applications réparties, susceptible de compléter ou de se substituer à d'autres paradigmes plus classiques tel le passage de messages, l'appel de procédure à distance, l'invocation d'objet à distance, l'évaluation à distance [FPV98, BI02]. Elle est d'un grand intérêt pour la mise en œuvre d'applications dont les performances varient en fonction de la disponibilité et de la qualité des services et des ressources, ainsi que du volume des données déplacées. Le concept d'agent mobile facilite en effet la mise en œuvre d'applications dynamiquement adaptables, et il offre un cadre générique [CHK94] pour le développement des applications réparties sur des réseaux de grande taille qui recouvrent des domaines administratifs multiples.

### 1.2.1 Principes et avantages

Un agent logiciel est une entité autonome capable de communiquer, disposant d'une connaissance partielle de ce qui l'entoure et d'un comportement privés, ainsi que d'une capacité d'exécution propre. Un agent agit pour le compte d'un tiers (un autre agent, un utilisateur) qu'il représente sans être obligatoirement connecté à celui-ci, réagit et interagit avec d'autres agents.

Un agent mobile peut se déplacer d'un site à un autre en cours d'exécution pour accéder à des données ou à des ressources. Il se déplace avec son code et ses données propres, mais aussi avec son état d'exécution. L'agent décide lui-même de manière autonome de ses mouvements. Ainsi, la mobilité est contrôlée par l'application elle-même, et non par le système d'exécution comme dans le cas de la migration de processus dans les systèmes opératoires.

En pratique, la mobilité d'agent permet de rapprocher client et serveur et en conséquence de réduire le nombre et le volume des interactions distantes (en les remplaçant par des interactions locales), de spécialiser des serveurs distants ou de déporter la charge de calcul d'un site à un autre. Une application construite à base d'agents mobiles peut se redéployer dynamiquement suivant un plan pré-établi ou en réaction à une situation particulière, afin par exemple d'améliorer la performance ou de satisfaire la tolérance aux pannes, de réduire le trafic sur le réseau, ou de suivre un composant matériel mobile. La mobilité du code offre un premier niveau de flexibilité aux applications. La décentralisation de la connaissance et du contrôle à travers les agents, et la proximité physique entre les agents et les ressources du système renforce la réactivité et les capacités d'adaptation.

La mobilité ne se substitue pas aux capacités de communication des agents (la communication distante reste possible) mais les complète; afin de satisfaire aux contraintes des réseaux de grande taille ou sans fil (latence, non permanence des liens de communication), les agents communiquent par messages asynchrones.

### 1.2.2 Propriétés

Le déplacement d'une unité en cours d'exécution d'une machine à une autre se heurte aux problèmes d'hétérogénéité des matériels et des logiciels. Les intergiciels destinés au développement d'applications réparties à grande échelle doivent permettre de s'en abstraire. Par ailleurs, ils doivent fournir les mécanismes permettant le déplacement des agents et la communication entre agents. En particulier, la communication doit (a priori) être insensible aux déplacements des agents : les messages à destination ou provenant d'un agent doivent être acheminés indépendamment des mouvements de cet agent.

La capacité de capture et de restauration de l'état d'exécution d'un agent, appelée mobilité forte, est un point critique. La mobilité faible représente la capacité pour un système de déplacer le code des agents accompagné seulement de données d'initialisation (et non de l'état complet). En fait, c'est essentiellement un problème d'abstraction et d'expressivité, lié à la manipulation des processus qui exécutent les agents, qui est posé : dans les systèmes à mobilité faible, c'est au programmeur de gérer lui-même le mécanisme de reprise. Cette propriété est liée (et parfois confondue) avec la capacité pour un agent de se déplacer à n'importe quel point de son exécution (mobilité *anytime*).

Un frein à l'utilisation réelle de cette technologie réside cependant dans les problèmes de sécurité qu'elle introduit. La mise en place d'une politique de sécurité peut demander, d'une part la protection des ressources et des données des machines hôtes (en limitant les droits d'accès et la consommation des ressources), et d'autre part, la préservation de l'intégrité et de la confidentialité des agents eux-mêmes et de leurs communications.

La large diffusion de l'environnement Java (et l'utilisation courante des *applets*) a contribué à l'intérêt porté aux agents mobiles, de par les nombreux avantages offerts (machine virtuelle, sérialisation, invocation de méthode à distance, gestionnaires de sécurité, ...). Java ne permet cependant pas la capture de l'état d'exécution des *threads*, et par conséquent, l'état dans lequel le calcul sera repris à distance doit être explicitement programmé, ce qui réduit l'expressivité : cette propriété est appelée mobilité faible (il existe des solutions Java non-standard basées sur la modification de la machine virtuelle, ou un pré-traitement code source, ou une modification du *bytecode*).

### 1.3 Les acteurs

#### 1.3.1 Le modèle de calcul

Les acteurs [Agh86] sont des entités anthropomorphes qui communiquent par messages. Au sein d'une communauté, un acteur est uniquement connu par sa référence. La communication est point à point, asynchrone, unilatérale et supposée sûre. Les messages reçus sont stockés dans une boîte aux lettres et traités en série (en exclusion mutuelle). Le comportement décrit la réaction de l'acteur à un message. Le comportement est privé : il contient les données et les fonctions de l'acteur, et constitue son état. L'acteur encapsule données et fonctions mais également une activité (fil d'exécution) unique qui les manipule. Un comportement d'acteur n'est donc, à un instant donné, traversé que par un fil d'exécution au plus (la synchronisation est implicite).

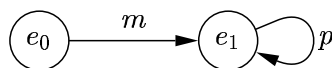
Lors du traitement d'un message, un acteur peut créer (dynamiquement) de nouveaux acteurs, envoyer des messages aux acteurs qu'il connaît, et changer de comportement c'est-à-dire définir son comportement pour le traitement du prochain message. Le changement de comportement est un mécanisme puissant d'évolution et d'adaptation individuelles. Il permet à un acteur de modifier dynamiquement son interface ; mais, en conséquence, il n'est pas possible de garantir dans tous les cas qu'un message envoyé pourra effectivement être traité par son destinataire. Cependant, pour le programmeur, le changement de comportement offre une alternative à la gestion de variables d'état et à l'utilisation de gardes, qui accroît l'expressivité. Programmer un acteur revient donc à programmer ses comportements, l'enchaînement des comportements étant décrit dans les comportements eux-mêmes.

#### 1.3.2 La validation des applications

La capacité de changement de comportement introduit une difficulté particulière dans le contrôle des envois de messages. En effet, il est délicat de déterminer statiquement si un message envoyé à un acteur pourra effectivement être traité. C'est le problème des *messages orphelins* [DP+00] :

- les orphelins de *sûreté* sont des messages que l'acteur ne peut pas prendre en compte, ou bien qu'il ne peut plus prendre en compte au moment où il doit les traiter (à cause par exemple d'un changement de comportement) ;
- les orphelins de *vivacité* sont des messages appartenant au potentiel de traitement de l'acteur, mais celui-ci est bloqué en attente d'un message préalable qu'il n'a pas reçu.

**Exemple** Soit l'acteur dont l'automate des comportements est le suivant :



Si on lui envoie deux fois le message  $m$ , l'un des deux messages sera orphelin de sûreté. Si on lui envoie seulement le message  $p$ , alors celui-ci sera orphelin de vivacité; il ne pourra être traité tant qu'aucun message  $m$  n'aura été envoyé à l'acteur.

### 1.3.3 La mobilité d'acteur

L'idée de l'utilisation des acteurs pour la programmation d'applications concurrentes et réparties n'est pas neuve. Néanmoins, elle semble particulièrement pertinente dans le cadre du calcul réparti à grande échelle de par les propriétés qui distinguent les acteurs du modèle d'objet classique (encapsulation de l'activité, changement d'interface, communications asynchrones). D'une part, la communication par messages asynchrones est bien adaptée aux réseaux de grande taille, voire sans fil, où les communications synchrones sont trop difficiles et coûteuses à établir. D'autre part, l'autonomie d'exécution et l'autonomie comportementale favorisent l'intégration de mécanismes de mobilité et garantissent un certain niveau d'intégrité.

La mobilité d'acteur peut être naturellement définie en se basant sur le traitement des messages en série et le changement de comportement, ainsi que sur la création dynamique et l'envoi de messages. Lors du changement de comportement (donc, entre deux messages), l'acteur se réduit d'une part au contenu de sa boîte aux lettres, et d'autre part au comportement défini pour le traitement du prochain message. Par nature, le comportement est une entité de première classe qui matérialise l'état et qui est manipulable par programme. Il est alors possible de créer à distance un autre acteur défini à partir du comportement. On obtient ainsi simplement une évolution de l'acteur d'origine à qui on peut faire suivre tous les messages qui étaient en attente. L'état est donc entièrement transporté (via le comportement) et restauré sans que le programmeur n'ait à développer de code spécifique pour cela. Ainsi, après migration, l'acteur peut poursuivre son activité à distance en bénéficiant des acquis résultant des traitements de messages précédents. La mobilité est ainsi différée au moment où l'acteur change de comportement.

La communication entre acteurs étant supposée sûre, la mobilité pose le problème de l'acheminement des messages. Pour cela, on définit le protocole suivant (d'autres sont possibles) : l'acteur d'origine se transforme lors du déplacement en un *proxy* du nouvel acteur (distant); il ne traite plus les messages qu'il reçoit mais les transmet. Ainsi, l'acteur reste connu à son adresse d'origine qui demeure valide, et l'envoi de message reste possible à tout instant y compris pendant que l'acteur se déplace (les messages sont stockés au niveau de l'acteur d'origine, puis transmis quand le clone distant est opérationnel). D'autre part, les communications sortantes (émissions de messages) restent valides (pour cela, les références d'acteur doivent être visibles à travers le réseau). Le mécanisme de mobilité est donc transparent pour le système de communication (il reste cependant les problèmes de liaison avec les ressources et les fichiers ouverts). La multiplication des déplacements d'un acteur a alors pour effet de construire une chaîne de liens de poursuite à travers laquelle l'acteur mobile peut être localisé et les messages acheminés (en pratique, on peut optimiser ce mécanisme).

Tout acteur est donc potentiellement mobile, et, ainsi définie, la mobilité n'altère pas la sémantique des programmes. On peut noter que les problèmes de cohérence de copies et de synchronisation que l'on rencontre quand on introduit la mobilité dans le modèle objet ne se posent pas ici du fait de l'unicité du fil d'exécution et de l'exclusion mutuelle sur le traitement des messages.

## 2 Installation

### 2.1 Vérifions les pré-requis

JAVACT est un logiciel développé en Java, il est donc nécessaire d'avoir un environnement Java correctement installé<sup>5</sup>. La bibliothèque JAVACT peut être installée et les machines virtuelles JAVACT exécutées avec le JRE 1.3 minimum. Pour compiler des application JAVACT le JSDK 1.4 minimum est nécessaire.

Il faut également avoir récupéré le programme d'installation et l'archive de la bibliothèque : `installJavAct.class` et `JavActv050.jar`. Ces fichiers sont disponibles sur le site de JAVACT <sup>6</sup>.

### 2.2 C'est parti !

Le fichier `installJavAct.class` est un programme permettant une installation portable de la bibliothèque sur n'importe quel système où l'on a pu installer Java. Son usage est le suivant :

```
java installJavAct <archive javact.jar> <répertoire de destination>
```

où `<archive javact.jar>` est le chemin complet de l'archive de la bibliothèque (`JavActv050.jar`), et `<répertoire de destination>` le répertoire dans lequel vous souhaitez installer la bibliothèque. Un répertoire `JavActv050` sera construit dans le répertoire désigné.

**Exemple** - Dans le répertoire `/home/leriche` :

```
java installJavAct JavActv050.jar .
```

```
Installing javact from 'JavActv5.jar' to '/home/leriche/JavActv050'
```

```
[...]
```

```
Generating scripts... (platform=Linux)
```

```
javact is now correctly installed and configured.
```

```
To run, build, compile applications for javact, you can use the scripts in  
'/home/leriche/JavActv050/bin'
```

```
[...]
```

Lors de cette étape, la bibliothèque est installée dans le répertoire `JavActv050`. Des scripts de lancement (cf. section suivante) sont générés en fonction de la plate-forme dans le sous-répertoire `bin`.

#### Remarques :

- Il est possible de préciser pour l'archive JAVACT une localisation plus complexe. Exemple de récupération depuis un site internet : `java installJavAct http://noman.zerezo.com/JavActv050.jar` .
- Si aucun argument n'est fourni, le programme d'installation cherche le fichier `JavActv050.jar` dans le répertoire courant. S'il le trouve, l'installation se fait dans le répertoire courant à partir de ce fichier.

---

<sup>5</sup>Pour vérifier votre version : `java -version` renvoie des informations sur l'environnement installé ; ensuite si la commande `javac` renvoie un message d'erreur vous disposez du JRE (exécution seulement) sinon vous avez le JSDK (compilation possible). Pour télécharger et installer Java : `http ://java.sun.com/j2se/`

<sup>6</sup>`http://www.irit.fr/recherches/ISPR/IAM/JavAct.html`

## 2.3 Organisation des fichiers

JavActv050

```
bin                                # contient les scripts d'exécution pour :
    build                          # recompiler la bibliothèque
    javact                         # lancer une application
    javactc                        # compiler une application
    javactgen                      # générer les messages de l'application
    javactvm                       # lancer un système d'accueil
    installJavAct.java             # installer la bibliothèque sur un autre site

doc                                # documentation 'javadoc' de la bibliothèque
    [...]

examples                           # exemples d'applications
    net                            # fonctionnement en mode réparti
    [...]
    local                          # fonctionnement en local
    [...]

javact                             # code source de la bibliothèque
    compiler                       # générateur de messages
    lang                           # code principal
    local                          # code spécifique au fonctionnement local
    net                            # code spécifique au fonctionnement réparti
    util                           # classes utiles, complémentaire de javact.lang

awfullPolicy                       # exemple de politique de sécurité
javact.jar                         # code compilé de la bibliothèque
LGPL.TXT                          # licence LGPL de la bibliothèque
places.txt                        # exemple de fichier de places
```



## 3 Utilisation

Maintenant que la bibliothèque est correctement installée, nous allons compiler et exécuter un exemple, le classique « Hello World » en mode réparti. Les commandes sont données au format Linux<sup>7</sup>, mais grâce à la portabilité de Java, la façon de procéder est normalement identique pour d'autres systèmes<sup>8</sup>.

### 3.1 Lancement des systèmes d'accueil

Pour exécuter une application JAVACT en mode réparti, il faut disposer d'un système d'accueil sur chacune des places<sup>9</sup> potentiellement accessibles aux agents mobiles. Une place est localisée par une URL (complète, adresse IP, nom de machine en réseau local...) et éventuellement un numéro de port. Chaque place peut mettre à disposition des agents mobiles un ensemble d'adresses d'autres places directement accessibles. Ces informations sont stockées dans un fichier (`places.txt`), qui contient pour chaque ligne une adresse de place.

Le lancement se fait grâce au script `javactvm` (dans le sous-répertoire `bin`), qui s'utilise de la manière suivante :

```
javactvm [chemin d'accès au fichier places.txt] [port]
```

où `[chemin d'accès]` est facultatif. S'il est précisé, il doit être impérativement sous forme absolue, sinon le fichier `places.txt` dans le répertoire courant sera utilisé, et où `[port]` ( $1099 \leq port \leq 65535$ ) est facultatif (il vaut 1099 par défaut).

**Exemple** - Dans `/home/leriche/JavActv050/bin` :

```
javactvm /home/leriche/JavActv050/places.txt 2010
```

```
Starting javact : 2010
```

```
javact v0.5.0 [chambord:2010] - 5:25:13 PM
```

**Remarque :** Si dans un réseau local une machine sur laquelle on aura besoin d'effectuer des accès depuis l'extérieur possède un nom réseau local (hostname) différent de son nom réseau internet, une exception liée au fonctionnement de RMI<sup>10</sup> sera levée (*java.net.UnknownHostException* après une instruction *Naming.lookup()*) et l'application ne pourra pas fonctionner correctement. Pour remédier à cela, il est possible de passer en paramètre du script `javactvm` l'option `-Djava.rmi.server.hostname=` suivie de l'adresse externe de la machine (URL ou IP). Par exemple sur une machine appelée `kinder` sur un réseau local accessible depuis l'extérieur par `zerezo.com`, il faut passer le paramètre `-Djava.rmi.server.hostname=zerezo.com`.

### 3.2 Exécution d'un programme

Une fois l'application compilée, le lancement est facilité par le script `javact` (dans le sous-répertoire `bin`) qui prend en paramètre le nom de la classe principale et ses éventuels paramètres. Si les classes nécessaires à l'exécution de l'application ne sont pas fournies sur chacun des sites d'exécution, il faut donner une adresse où chaque machine virtuelle pourra les récupérer<sup>11</sup>. Cela se fait en passant comme

---

<sup>7</sup>Sous Linux/Unix si vous n'avez pas le répertoire courant (`.`) dans le `path`, il faudra préfixer les scripts par `./` pour pouvoir les exécuter : `./javactvm`.

<sup>8</sup>Sur les systèmes Windows, le caractère de séparation des chemins sera simplement `'/'` au lieu de `'\'` :

```
javactvm d:\JavActv050\places.txt.
```

<sup>9</sup>Machine virtuelle java exécutant une instance de la classe `javact.net.rmi.Creator`

<sup>10</sup><http://java.sun.com/j2se/1.4.2/docs/guide/rmi/faq.html#domain>

<sup>11</sup><http://java.sun.com/j2se/1.4.2/docs/guide/rmi/codebase.html>

premier argument `-Djava.rmi.server.codebase=` suivi d'une ou plusieurs localisation des fichiers entre " ", qui peuvent être des répertoires, des archives jar, etc.

S'il existe un système de fichiers partagés (type nfs), on utilisera par exemple :

```
-Djava.rmi.server.codebase=
    "file:/home/leriche/JavActv050/examples/net/rmi/HelloWorld/"
```

(attention au '/' final!), on peut aussi utiliser un serveur web externe, par exemple :

```
-Djava.rmi.server.codebase="http://noman.zerezo.com/JavAct/HelloWorld/"
```

**Exemple** - Après avoir lancé un système d'accueil sur `caraibe`, `chambord` et `chambord:1100`, on se place dans le répertoire `/home/leriche/JavActv050/examples/net/rmi/HelloWorld` où se trouve le fichier `HelloWorld.class`. On peut lancer l'exécution du programme `HelloWorld` en passant les références des 3 places en paramètre (ceci provoque l'affichage de « Hello » sur la première place, de « Wonderful » sur la deuxième et de « World » sur la troisième) :

```
/home/leriche/JavActv050/bin/javact
-Djava.rmi.server.codebase=
    "file:/home/leriche/JavActv050/examples/net/rmi/HelloWorld/"
HelloWorld
caraibe chambord chambord:1100
```

**Remarque :** Si l'application est construite avec des paquetages, il faut se placer dans le répertoire supérieur au paquetage pour lancer l'application (comme pour les programmes java standard). Par exemple, si le programme précédent était dans un paquetage `examples.net.rmi.helloWorld`, il faudrait se placer dans le répertoire supérieur à `examples` puis appeler le script `javact` avec :

```
~/examples>/home/leriche/JavActv050/bin/javact
-Djava.rmi.server.codebase="file:/home/leriche/JavActv050/"
examples.net.rmi.helloWorld.HelloWorld
caraibe chambord chambord:1100
```

### 3.3 Compilation de la bibliothèque

Pour les utilisateurs qui souhaitent modifier la bibliothèque pour leur propre usage, nous avons inclus un script de compilation `build` (dans le sous-répertoire `bin`) qui archive toutes les classes compilées dans le fichier `javact.jar`, placé dans le répertoire `JavActv050`.

**Exemple** - Dans `/home/leriche/JavActv050/bin` : `build`

```
Compiling javact ...
Compiling Creator for RMI ...
Compiling ReceiveCt for RMI ...
Compiling ClientSideAnswerBackImpl for RMI ...
Building javact.jar...
[...]
Done !
```

## 4 Comment développer une application

Il convient de respecter un certain ordre dans les étapes de développement d'une application JAVACT :

1. Spécification des profils des comportements et des acteurs (définition d'interfaces Java)
2. Génération automatique des classes pour les messages et des classes abstraites pour les comportements
3. Implantation des comportements (codage des classes qui étendent les classes abstraites générées)

Nous allons détailler le processus de développement dans un 1<sup>er</sup> exemple ; puis, les exemples suivants serviront chacun à expliquer la mise en œuvre de certaines caractéristiques de JAVACT.

### 4.1 L'incontournable « Hello World »

Nous utilisons trois acteurs qui s'exécutent sur trois places. Le premier se chargera d'imprimer « Hello », puis activera le second qui affichera alors « Wonderful » et activera le troisième, qui lui affichera « World ».

#### 4.1.1 Profils des comportements et des acteurs

La fig. 1 présente l'articulation entre les diverses interfaces à définir. Chaque comportement doit être spécifié par une interface appropriée qui doit hériter de **BehaviorProfile**, et déclarer :

- les méthodes correspondant aux messages que ce comportement peut traiter ;
- les changements de comportement possibles (les comportements possibles pour le message suivant).

Ensuite, pour chaque acteur de l'application il faut rassembler l'ensemble des comportements qu'il est susceptible d'adopter, en définissant une interface héritant de **ActorProfile**. Il s'agit simplement d'une interface de marquage qui doit hériter des interfaces de tous les comportements adoptables par l'acteur. Cependant, par mesure de simplicité, si l'acteur ne prend qu'un seul comportement il est possible d'utiliser une seule interface qui héritera en même temps de **BehaviorProfile** et de **ActorProfile**.

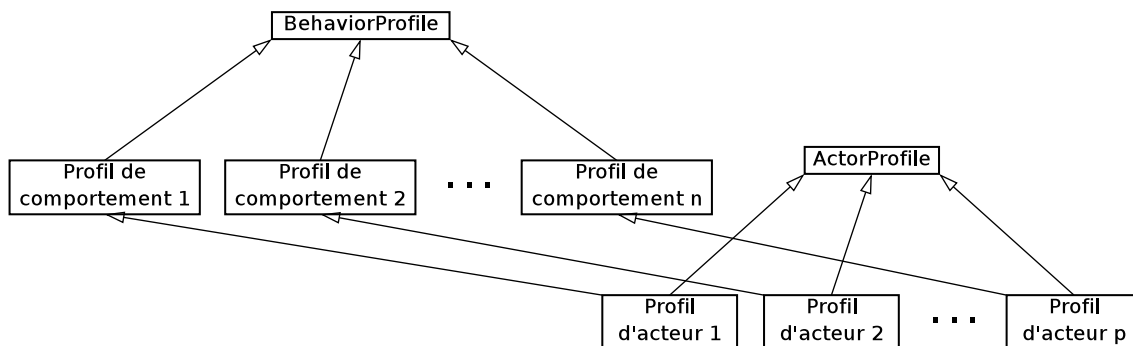


FIG. 1 – Structure d'une application JAVACT

**Exemple** Nous avons trois profils de comportement, chacun acceptant un message **print** avec éventuellement en paramètre le (les) acteur(s) à activer ensuite. Chaque acteur n'adopte qu'un seul comportement (respectivement **Hello**, **Wonderful**, **World**). Pour simplifier, on peut donc fusionner la définition des profils de comportement et d'acteur :

```

import javact.util.BehaviorProfile;
import javact.util.ActorProfile;
import javact.lang.Actor;
interface Hello extends BehaviorProfile, ActorProfile {
    /* Paramètres : les acteurs chargés d'afficher "Wonderful" et "World" */
    public void print(Actor wonderful, Actor world);
}

import javact.util.BehaviorProfile;
import javact.util.ActorProfile;
import javact.lang.Actor;
interface Wonderful extends BehaviorProfile, ActorProfile {
    /* Paramètre : l'acteur chargé d'afficher "World" */
    public void print(Actor world);
}

import javact.util.BehaviorProfile;
import javact.util.ActorProfile;
interface World extends BehaviorProfile, ActorProfile {
    public void print();
}

```

En fait, on peut spécifier deux sortes de méthodes auxquelles correspondent deux sortes de messages :

- les messages *asynchrones* classiques correspondent aux méthodes renvoyant `void` ;
- les messages *avec retour* correspondent aux méthodes qui renvoient une valeur.

Les messages *avec retour* sont des messages spéciaux, qui restent asynchrones et non bloquants pour l'émetteur mais qui lui permettent d'obtenir une réponse. Quand il en a besoin, l'émetteur doit la demander explicitement en faisant un appel à la méthode `getReply()` de l'objet message ; alors, il se bloque jusqu'à son obtention. Auparavant il aura pu effectuer d'autres actions.

#### 4.1.2 Génération automatique

Deux types de fichiers sont générés :

- les implantations de `QuasiBehavior` implantant le code des méthodes `become` ;
- les classes correspondant aux messages.

Pour les messages *asynchrones* classiques, le préfixe **JAM** est ajouté devant le nom du message pour obtenir le nom du fichier java correspondant. Pour les messages *avec retour*, on ajoute le préfixe **JSM** ainsi que le nom du type de retour, pour distinguer les messages de même nom mais renvoyant des valeurs de types différents<sup>12</sup>, et la méthode `getReply()` est engendrée avec le « bon » type de retour.

**Exemple** Pour Hello World nous aurons, pour les messages, une seule classe **JAMprint** contenant les trois signatures possibles ; son code source est donné en annexe A (p. 26).

Pour effectuer la génération automatique, il suffit d'exécuter le script `javactgen` (dans le sous-répertoire `bin`) qui se chargera d'extraire toutes les informations nécessaires à la génération. Cela donnera un résultat semblable au suivant :

```
$ javactgen *.java
```

---

<sup>12</sup> Cela a pour conséquence que les types tableaux ne sont pas acceptés comme types de retour : leurs noms contiennent des crochets (« [ ») qui ne sont pas des caractères autorisés dans les noms de variables. Ce problème sera résolu dans une version ultérieure de JAVACT.

```

Found the following actor profile(s): Hello, Wonderful, World.
Compiling files defining actor profile interfaces...
Extracting messages...
Handling the actor profile Hello...
    This is also a behavior profile.
Handling the actor profile Wonderful...
    This is also a behavior profile.
Handling the actor profile World...
    This is also a behavior profile.
Generating QuasiBehaviors...
Generating messages...
Compiling generated files...

```

Pour Hello World les fichiers générés seront :

- HelloQuasiBehavior.java, WonderfulQuasiBehavior.java et WorldQuasiBehavior.java
- JAMprint.java.

**Remarque :** Si l'application utilise des paquetages, il faut le préciser au générateur en utilisant l'option suivante : `-P <nom du paquetage>`. Si l'application déclare le paquetage `examples.net.rmi.helloWorld`, il faut pour générer les messages se placer dans le répertoire supérieur à `examples` (contenant `examples`), puis appeler le script de la façon suivante :

```
javactgen -P examples.net.rmi.helloWorld examples/net/rmi/helloWorld/*.java.
```

#### 4.1.3 Implantation des comportements

Il faut maintenant écrire le contenu des méthodes déclarées dans les profils de comportement. Pour cela il faut écrire des classes qui héritent des classes générées `XxxQuasiBehavior`.

**Exemple** Le code de la classe de `HelloBeh` pourrait être le suivant :

```

import javact.lang.Actor;
class HelloBeh extends HelloQuasiBehavior {
    public void print(Actor wonderful, Actor world) {
        System.out.println("<" + myPlace() + "> -- Hello ...");
        send(new JAMprint(world), wonderful);
        suicide();
    }
}

```

Lorsqu'il traite le message `print`, l'acteur affiche « Hello » puis transmet le message aux acteurs suivants. Enfin il se suicide puisqu'il a terminé sa tâche.

Nous obtenons finalement la structure présentée en fig. 2.

#### 4.1.4 Et pour finir...

Il n'y a plus désormais qu'à écrire la fin de l'application. Il est nécessaire d'amorcer le processus en créant « à la main » les premiers acteurs et en envoyant le premier message. C'est à cela que servent `CreateCt.STD.create()` et `SendCt.STD.send()`<sup>13</sup>.

<sup>13</sup>Il faut noter que les méthodes `create()` et `send()` ne doivent théoriquement être appelées que par des objets acteurs, précisément par leurs comportements. Les objets statiques `CreateCt.STD` et `SendCt.STD` permettent de créer des acteurs

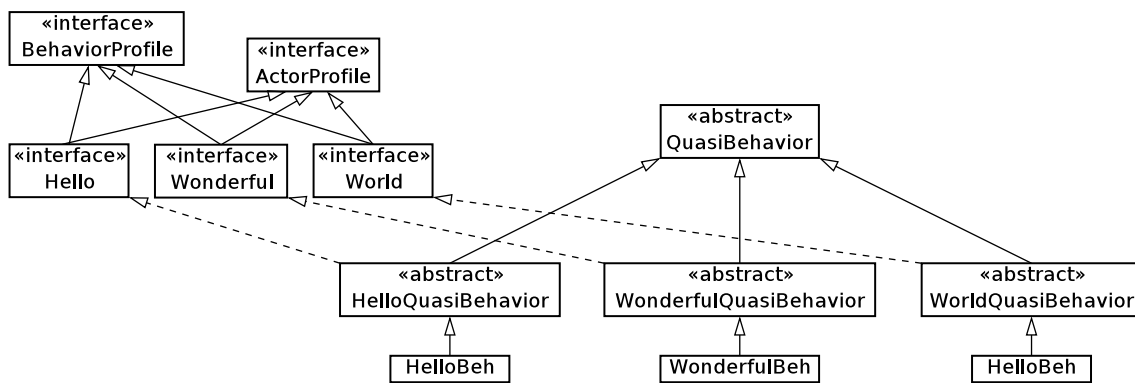


FIG. 2 – Structure de l'application Hello World

**Exemple** Notre méthode `main` pourrait contenir les lignes suivantes :

```

import javact.lang.Actor;
import javact.net.rmi.CreateCt;
import javact.net.rmi.SendCt;
public class HelloWorld {
    public static void main(String[] args) {
        if (args.length == 3) {
            Actor hello = CreateCt.STD.create(args[0], new HelloBeh());
            Actor wonderful = CreateCt.STD.create(args[1], new WonderfulBeh());
            Actor world = CreateCt.STD.create(args[2], new WorldBeh());
            SendCt.STD.send(new JAMprint(wonderful, world), hello);
        } else {
            System.out.println("Running on randomly determinated places");
            Actor hello = CreateCt.STD.create(new HelloBeh());
            Actor wonderful = CreateCt.STD.create(new WonderfulBeh());
            Actor world = CreateCt.STD.create(new WorldBeh());
            SendCt.STD.send(new JAMprint(wonderful, world), hello);
        }
    }
}

```

Les trois acteurs sont créés sur trois places dont les noms (`String`) sont passés en ligne de commande (ou, par défaut, choisies aléatoirement au sein du domaine défini dans le fichier `places.txt`), puis le premier est activé en lui envoyant un message « `print` ». C'est ce message qui permet d'amorcer la communication.

La compilation des classes restantes se fait avec le script `javactc` (sous-répertoire `bin`). Il s'utilise exactement comme le compilateur `javac` et doit recevoir en paramètre le fichier contenant la déclaration « `public static void main...` ».

Pour l'exécution de cet exemple ou pourra se reporter à la section précédente.

---

ou de leur envoyer des messages en dehors des comportements d'acteur, par exemple dans le `main`, et doivent être réservés à cela.

## 4.2 La cellule

Nous allons ici illustrer le changement de comportement avec une cellule alternativement vide et pleine<sup>14</sup>. Les profils de comportement sont les suivants :

```
interface Full extends BehaviorProfile {
    public void get(Actor a);
    public void become(Empty bp);
}

interface Empty extends BehaviorProfile {
    public void put(int v);
    public void become(Full bp);
}
```

L'acteur correspondant est décrit par le profil suivant :

```
interface Cell extends Empty, Full, ActorProfile {
}
```

Pour illustrer le fonctionnement de la cellule nous définissons un client qui récupèrera son contenu :

```
interface Client extends BehaviorProfile, ActorProfile {
    public void val(int v) ;
}
```

L'annexe B p. 27 donne un exemple de comportement automatiquement généré.

L'implantation des comportements sera la suivante :

```
class EmptyCellBeh extends EmptyQuasiBehavior {
    public void put(int i) {
        System.out.println("Empty cell " + ego() + " receives put(" + i + ") request");
        become(new FullCellBeh(i, this));
    }
}

class FullCellBeh extends FullQuasiBehavior {
    int i;
    Empty empty;

    FullCellBeh(int i, Empty empty) {
        this.i = i;
        this.empty = empty;
    }

    public void get(Actor a) {
        System.out.println("Full cell " + ego() +
            " receives get request; will return " + i);
        send(new JAMvalue(i), a);
        become(empty);
    }
}

class ClientBeh extends ClientBehavior {
    public void value(int i) {
        System.out.println("Client " + ego() + " receives " + i);
    }
}
```

---

<sup>14</sup>A partir d'ici, nous omettons de donner les lignes d'import

Et le programme principal pourra être le suivant :

```
public static void main(String[] args) {
    Actor cell = CreateCt.STD.create("localhost", new EmptyCellBeh());
    SendCt.STD.send(new JAMput(2004), cell);

    Actor customer = CreateCt.STD.create("localhost:1100", new ClientBeh());
    SendCt.STD.send(new JAMget(customer), cell);
}
```

Ici, la cellule et le client sont respectivement créés sur les places `localhost` et `localhost :1100`.

### 4.3 Le crible d’Eratosthène

L’exemple présenté dans cette partie concerne le calcul des N premiers nombres premiers par la méthode du Crible d’Eratosthène. Cet exemple illustre en particulier la faculté de création dynamique des acteurs. L’algorithme est proche d’une version parallèle initialement proposée par D. Gries<sup>15</sup>.

Le calcul repose sur la construction dynamique d’un pipeline d’acteurs, chacun représentant un nombre premier, les acteurs étant chaînés dans l’ordre croissant. Chacun des acteurs joue le rôle d’un crible plus sélectif que le crible précédent. Tout acteur est conçu pour recevoir et traiter (méthode `sift`) un message contenant un nombre à cribler (message `JAMsift`). En terme de communication, tous les acteurs du crible offrent donc la même interface.

Tous les messages sont envoyés au premier acteur du pipeline, un par un et dans l’ordre croissant. Lorsque le nombre reçu est un multiple du nombre (premier) que l’acteur représente, il ne traverse pas le crible et il est abandonné. En revanche, lorsque ce nombre n’est pas multiple, deux cas se présentent, selon que l’acteur qui le traite est à l’extrémité du pipeline ou pas. Chacun de ces cas est implémenté par un comportement différent (classe `LastBeh` pour le comportement de l’acteur extrémité, classe `IntermediateBeh` dans l’autre cas) :

- si l’acteur est à l’extrémité du pipeline, il crée un nouvel acteur qui représente le nombre reçu (la nouvelle extrémité), puis prend un comportement d’acteur intermédiaire;
- si l’acteur n’est pas à l’extrémité, il fait suivre le nombre reçu à son suivant dans le pipeline (dont il connaît la référence puisqu’il l’a créé).

Quand un acteur du pipeline a pris un comportement d’acteur intermédiaire, il le conserve définitivement. Ceci conduit à spécifier deux interfaces de comportement qui diffèrent par l’existence de la méthode `become`<sup>16</sup>. Alors, on peut spécifier les profils ainsi :

```
public interface IntermediateSieve extends BehaviorProfile {
    public void sift(int i);
}

public interface LastSieve extends IntermediateSieve {
    public void become(IntermediateSieve b);
}

public interface Sieve extends IntermediateSieve, LastSieve, ActorProfile {
}
```

Les comportements peuvent alors s’implémenter comme suit :

```
class IntermediateBeh extends IntermediateSieveQuasiBehavior {
    protected int prime;
```

---

<sup>15</sup>dans : C.A.R. HOARE, Communicating Sequential Processes, *Communications of the ACM* 21(8), 1978, pp. 666-677

<sup>16</sup>Ne pas coder de changement de comportement équivaut à exprimer plus simplement un `become` sur soi-même



```

protected Actor next;

public IntermediateBeh(int i, Actor a) {
    super();
    prime = i;
    next = a;
}

public void sift(int i) {
    if ((i % prime) != 0)
        send(new JAMSift(i), next);
}
}

class LastBeh extends LastSieveQuasiBehavior {
    protected int prime;

    public LastBeh(int i) {
        super();
        prime = i;
        System.out.println("*** " + prime + " is a prime number ***");
    }

    public void sift(int i) {
        if ((i % prime) != 0) {
            Actor next = create(new LastBeh(i));
            become(new IntermediateBeh(prime, next));
        }
    }
}

```

L'application commence par la création de l'acteur pour l'entier 2 auquel sont envoyés les nombres dans l'ordre croissant<sup>17</sup>.

```

public class PrimeNumbers {
    public static void main(String[] args) {
        if (args.length != 0) {
            int valint = Integer.parseInt(args[0]);
            if (valint > 2) {
                Actor two = CreateCt.STD.create(new LastBeh(2));
                for (int i = 3; valint >= i; i++)
                    SendCt.STD.send(new JAMSift(i), two);
            } else System.out.println("Give an int > 2");
        }
    }
}

```

Les deux captures d'écran (figures 3 et 4) montrent la trace de l'exécution (ici, les acteurs sont créés aléatoirement sur une des places du domaine, ce qui en l'occurrence engendre une inefficacité certaine). Dans la figure 4, on voit en bas à gauche la fenêtre montrant **caraibe** station Sun sous SunOS, en haut à droite la ligne de commande du crible (notez l'utilisation d'un serveur de classe sur **chambord:8080**), et en bas à droite le listing des classes demandées au serveur de classe

---

<sup>17</sup>Cet algorithme nécessite que l'ordre d'envoi des messages soit préservé à la réception, ce que nous appelons propriété de régularité de la communication. Ceci assure que le premier nombre non multiple traité par l'acteur est bien un nombre premier. Cette propriété de régularité ne fait pas partie du modèle d'acteur. Cependant, l'implantation de JavAct à partir de RMI, lui-même basé sur TCP/IP garantit, de fait, cette propriété. Ceci permet d'avoir un code des comportements plus simples, dans lesquels on ne se préoccupe pas de l'ordre des messages reçus.

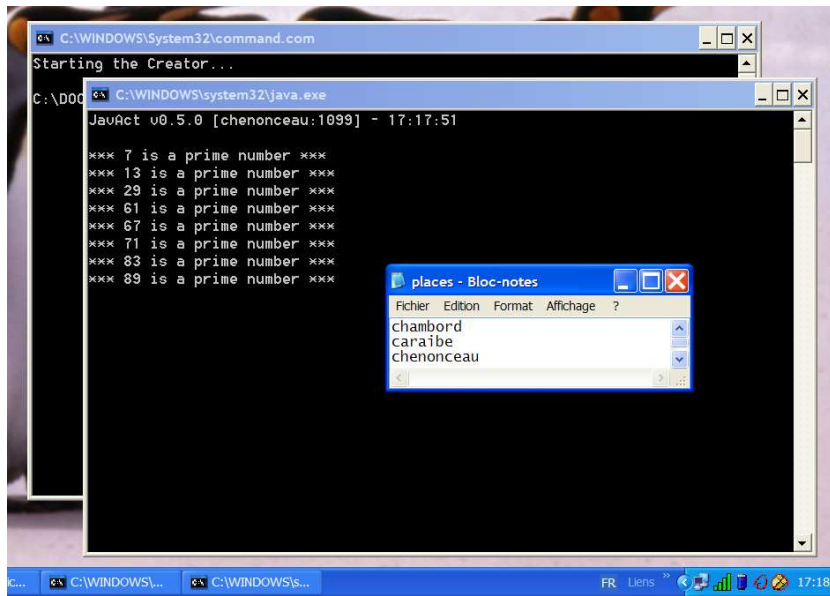


FIG. 3 – Capture sous Windows XP, PC chenonceau

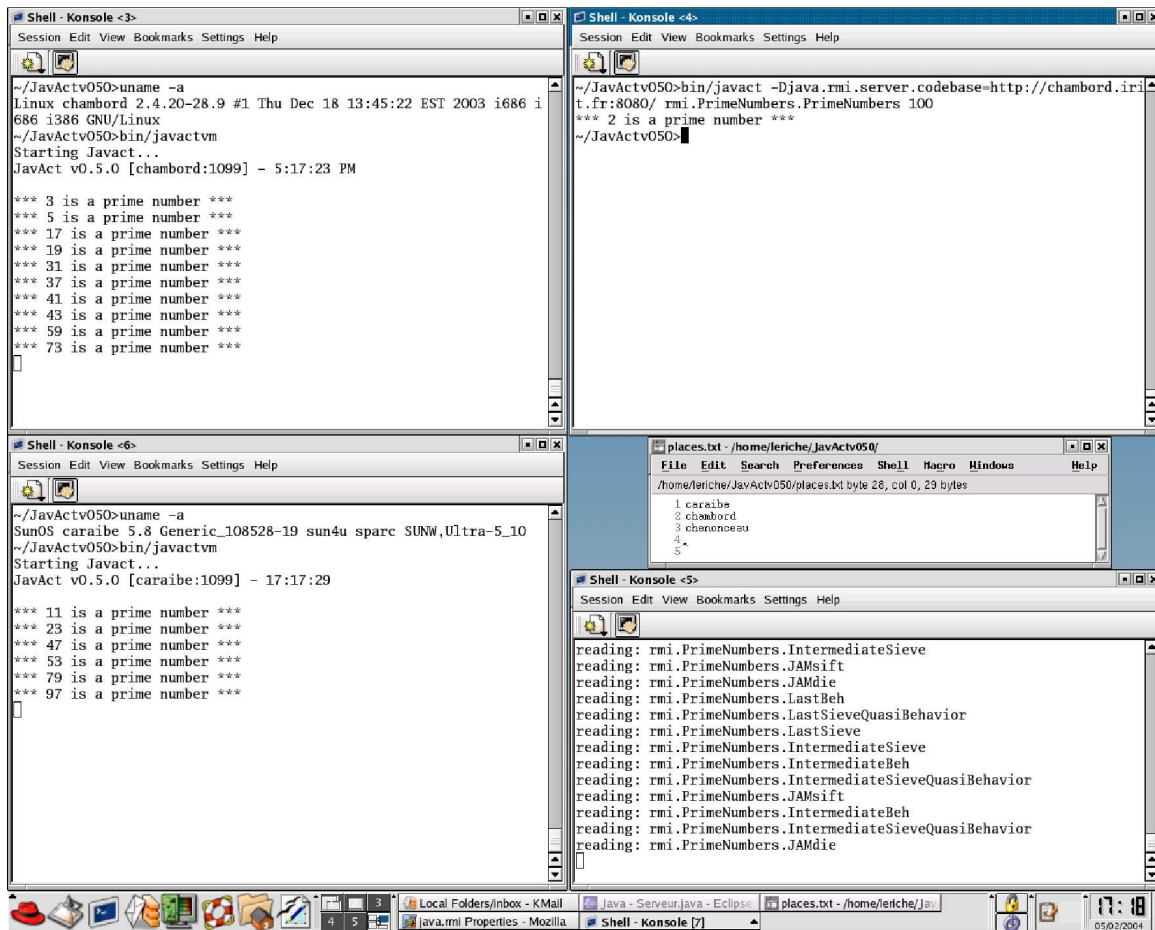


FIG. 4 – Capture sous Linux, PC chambord

## 4.4 Un agent mobile de recherche d'information

Nous considérons maintenant une application dans laquelle un agent mobile recherche des informations sur des places qu'il visite<sup>18</sup>. Par cet exemple, nous illustrons d'une part la mobilité d'agent et ses différents aspects, d'autre part le mécanisme d'envoi de messages avec retour de résultats.

Ici, pour l'exemple, l'information recherchée est le volume de mémoire disponible. L'agent mobile de recherche est susceptible de recevoir et de traiter trois types de requêtes :

- des demandes de recherche simple, sous forme de messages asynchrones standard contenant un identificateur de place ;
- des demandes de recherche complexe, sous forme de messages asynchrones standard contenant un itinéraire et un identificateur de place ;
- des demandes d'information, sous forme de messages asynchrones avec retour de résultat.

Selon le type de demande, l'agent opère comme suit :

- pour une demande de recherche simple, l'agent récupère et mémorise le volume de mémoire disponible sur la place où il se trouve, puis se déplace sur la place donnée dans le message ;
- pour une demande de recherche complexe, l'agent récupère et mémorise le volume de mémoire disponible sur la place où il se trouve ainsi que sur toutes les places de l'itinéraire qu'il parcourt, puis à la fin du parcours, il se déplace sur la place donnée dans le message ;
- pour une demande d'information, l'agent retourne au demandeur les informations acquises depuis la dernière demande de résultat.

Pour l'agent mobile, de type **browser**, les méthodes **browseAndJump()** et **getInfos()** définissent l'ensemble des messages qu'il peut traiter. Le traitement d'une demande de recherche complexe impose à l'agent de prendre un comportement particulier dans lequel il va parcourir l'itinéraire et rechercher l'information, sans traiter les messages en attente.

Toutes les requêtes étant envoyées à l'agent mobile par le programme principal, nous ne spécifions pas d'agent client. En revanche, l'application implique un agent superviseur (dont la référence est connue de l'agent mobile) et qui a pour seul rôle d'être informé par message lorsqu'un déplacement de l'agent échoue (méthode **trace()**).

La spécification des profils des comportements et des acteurs conduit donc aux définitions ci-dessous :

```
public interface OneStepBrowser extends BehaviorProfile {
    public void browseAndJump(String place);
    public void browseAndJump(Vector itinerary, String place);
    public Vector getInfos();
    void become(MultiStepBrowser b);
}
public interface MultiStepBrowser extends BehaviorProfile {
    void become(OneStepBrowser b);
}

public interface Browser extends OneStepBrowser, MultiStepBrowser, ActorProfile {
}

public interface Supervisor extends BehaviorProfile, ActorProfile {
    public void trace(String s);
}
```

Le prétraitement de ces interfaces (via le script **javactgen**) permet d'engendrer les classes :

---

<sup>18</sup>Pour une étude plus approfondie concernant les agents mobiles de recherche d'information, on pourra se reporter à [ALP03]

- `OneStepBrowserQuasiBehavior`, `MultiStepBrowserQuasiBehavior` et `SupervisorQuasiBehavior`,
- `JAMbrowseAndJump`, `JAMtrace` et `JSMgetInfosVector`.

La classe `JSMgetInfosVector` offre une méthode `getReply()` qui permet la récupération du résultat de la requête :

```
public class JSMgetInfosVector extends MessageWithReply {
    private java.util.Vector result ;
    ...
    public java.util.Vector getReply() {
        ...
    }
}
```

Pour implanter les comportements, on dispose de la méthode `go(String p)` qui permet d'indiquer que le prochain message sera traité (avec le prochain comportement) sur la place identifiée par `p`. Comme c'est le cas pour le changement de comportement, le déplacement n'est pas réellement effectué au moment de l'appel à `go()` mais avant de traiter le message suivant<sup>19</sup>. Précisons que :

- d'une part, on peut invoquer `go()` comme `become()` n'importe où dans le comportement,
- d'autre part, en cas d'appel successifs à `go()` comme à `become()`, seul le dernier sera pris en considération,
- enfin, un échec du déplacement ne provoque pas le jet d'une exception dans `go()` ; aussi, du point de vue du déplacement, la méthode `go()` n'échoue pas, et toutes les instructions du comportement qui suivent l'invocation `go()` sont exécutées.

En revanche, via un mécanisme d'exception interne à la bibliothèque, il est possible pour le programmeur, de contrôler un échec éventuel lors du déplacement : la méthode `go(String p, HookInterface h)` permet d'associer au déplacement un objet dédié à la reprise du calcul en cas d'échec au déplacement, et dont la méthode `void resume()` sera alors automatiquement invoquée. On peut programmer la reprise du calcul au sein d'une classe qui implante l'interface `HookInterface` en implantant la méthode `resume()` ; si cette classe est interne à la classe comportement, on peut accéder à tous les éléments du comportement courant, et en particulier redéfinir le déplacement et le comportement suivant de l'agent.

Pour que l'agent puisse opérer de manière autonome sur son itinéraire, le comportement de recherche complexe doit implanter l'interface `javact.util.StandAlone` : cette interface spécifie une méthode `run()` qui est exécutée automatiquement à chaque fois qu'un agent est installé (par création ou déplacement) sur une place. Via cette interface, on peut d'une part créer des agents auto-actifs sans qu'il soit nécessaire de les activer par message, d'autre part (c'est ici le cas) des agents capables de se déplacer de manière autonome sur le réseau<sup>20</sup>.

On code alors les comportements comme suit (la structure de donnée est déclarée `Serializable` afin de pouvoir être déplacée et transmise sans provoquer une exception) :

```
class Info implements Serializable {
    private long freeMemory;
    private String place;
```

---

<sup>19</sup>Ainsi liée au changement de comportement, l'introduction de la mobilité ne réduit pas l'expressivité et le niveau d'abstraction du support de programmation : la mobilité n'impose pas le développement de code supplémentaire, et par conséquent, on contourne les inconvénients de la mobilité faible de Java

<sup>20</sup>L'interface `javact.util.StandAlone` contribue grandement à l'expressivité de la mobilité car elle permet d'éviter au programmeur une gestion artificielle des messages en attente. Ajouté à l'expression naturelle de la mobilité liée au changement de comportement, ceci permet de contourner les difficultés dues à la mobilité faible de Java, c'est-à-dire à l'impossibilité de capturer un l'état d'exécution d'un agent

```

    public Info(String place, long freeMemory) {
        this.place = place;
        this.freeMemory = freeMemory;
    }

    long getFreeMemory() {
        return freeMemory;
    }
    String getPlace() {
        return place;
    }
    public String toString() {
        return place + " : " + freeMemory;
    }
}

class SimpleHook implements HookInterface {
    Actor supervisor;
    SimpleHook(Actor supervisor) {
        this.supervisor = supervisor;
    }
    public void resume(GoException e) {
        // Nothing to do : the agent stays in its current place. A message is sent.
        SendCt.STD.send(new JAMtrace("Unreachable place : " + e.getPlace()), supervisor);
    }
}

/**
 * The behavior for (when activated) local browsing, then moving.
 */
class OneStepBrowserBeh extends OneStepBrowserQuasiBehavior {
    Actor supervisor;
    Vector collectedInfos;

    class Hook implements HookInterface {
        Vector itinerary;
        String place;
        Hook(Vector itinerary, String place){
            this.itinerary = itinerary;
            this.place = place;
        }
        public void resume(GoException e) {
            String message = ego() + " : " + e.getPlace() + " is unreachable from "
                + myPlace() + " - Moves to " + itinerary.firstElement();
            send(new JAMtrace(message), supervisor);
            go((String) itinerary.firstElement(), new Hook(itinerary, place));
            itinerary.removeElementAt(0);
            become(new MultiStepBrowserBeh(supervisor, collectedInfos, itinerary, place));
        }
    }

    // Constructors
    OneStepBrowserBeh(Actor supervisor) {
        this.supervisor = supervisor;
        this.collectedInfos = new Vector();
    }
    OneStepBrowserBeh(Actor supervisor, Vector collectedInfos) {

```

```

        this.supervisor = supervisor;
        this.collectedInfos = collectedInfos;
    }

    // Methods
    public void browseAndJump(String place) {
        // local browsing first
        collectedInfos.addElement(new Info(myPlace(), Runtime.getRuntime().freeMemory()));
        // then, moving
        go(place, new SimpleHook(supervisor));
    }

    public void browseAndJump(Vector itinerary, String place) {
        // local browsing first
        collectedInfos.addElement(new Info(myPlace(), Runtime.getRuntime().freeMemory()));
        if (!itinerary.isEmpty()) {
            // then travelling
            go((String) itinerary.firstElement(), new Hook(itinerary, place));
            itinerary.removeElementAt(0);
            become(new MultiStepBrowserBeh(supervisor, collectedInfos, itinerary, place));
        } else
            // else moving
            go(place, new SimpleHook(supervisor));
    }

    public Vector getInfos() {
        Vector saveCollectedInfos = collectedInfos;
        collectedInfos = new Vector();
        return saveCollectedInfos;
    }
}

/**
 * The behavior for automatic browsing and moving on a list of places
 */
class MultiStepBrowserBeh extends MultiStepBrowserQuasiBehavior implements StandAlone {
    Actor supervisor;
    Vector collectedInfos;
    Vector itinerary;
    String place;

    class Hook implements HookInterface {
        Vector itinerary;
        String place;
        Hook(Vector itinerary, String place){
            this.itinerary = itinerary;
            this.place = place;
        }

        public void resume(GoException e) {
            String message = ego() + " : " + e.getPlace() + " is unreachable from "
                + myPlace() + " - Moves to ";
            if (itinerary.size() > 0) {
                message += itinerary.firstElement();
                send(new JAMtrace(message), supervisor);
                go((String) itinerary.firstElement(), new Hook(itinerary, place));
                itinerary.removeElementAt(0);
            } else {
                message += place;
                send(new JAMtrace(message), supervisor);
            }
        }
    }
}

```

```

        go(place);
    }
}

// Constructor
MultiStepBrowserBeh(Actor supervisor, Vector collectedInfos,
                    Vector itinerary, String place) {
    this.collectedInfos = collectedInfos;
    this.itinerary = itinerary;
    this.place = place;
}

// Method
public void run() {
    // local browsing first
    collectedInfos.addElement(new Info(myPlace(), Runtime.getRuntime().freeMemory()));
    if (!itinerary.isEmpty()) {
        // then travelling
        go((String) itinerary.firstElement(), new Hook(itinerary, place));
        itinerary.removeElementAt(0);
    } else {
        // else moving
        become(new OneStepBrowserBeh(supervisor, collectedInfos));
        go(place, new SimpleHook(supervisor));
    }
}
}

class SupervisorBeh extends SupervisorQuasiBehavior {
    public void trace(String s) {
        System.out.println(s);
    }
}

```

Enfin, on écrit le programme principal. Rappelons que l'envoi de messages avec retour de résultat n'est pas bloquant pour l'émetteur de la requête : ici, l'émetteur n'est pas bloqué pendant que l'agent parcourt son itinéraire, mais seulement au moment où il demande explicitement le résultat en attendant son arrivée.

```

public class MobileBrowser {
    public static void main(String[] args) {
        if (args.length >= 2) {
            Actor supervisor = CreateCt.STD.create(new SupervisorBeh());
            Actor browser = CreateCt.STD.create(args[0], new OneStepBrowserBeh(supervisor));

            for (int i = 1; i < args.length; i++)
                SendCt.STD.send(new JAMbrowseAndJump(args[i]), browser);
            SendCt.STD.send(new JAMbrowseAndJump(args[0]), browser);
            JSMgetInfosVector m1 = new JSMgetInfosVector();
            SendCt.STD.send(m1, browser);

            Vector itinerary = new Vector();
            for (int i = 1; i < args.length; i++) itinerary.addElement(args[i]);
            SendCt.STD.send(new JAMbrowseAndJump(itinerary, args[0]), browser);
            JSMgetInfosVector m2 = new JSMgetInfosVector();
            SendCt.STD.send(m2, browser);
        }
    }
}

```

```

        System.out.println(m1.getReply());
        System.out.println(m2.getReply());
    }
}
}

```

La capture d'écran (figure 5) montre l'exécution de cet exemple, avec deux machines **caraibe** et **chambord** actives. La place **XXX** n'existe pas, et la trace d'exécution montrée par le superviseur (fenêtre de gauche) indique les actions de reprise sur erreur. La fenêtre de lancement (en haut à droite) contient également les lignes de commande pour la génération des classes intermédiaires ainsi que la compilation du paquetage **MobileBrowser**.

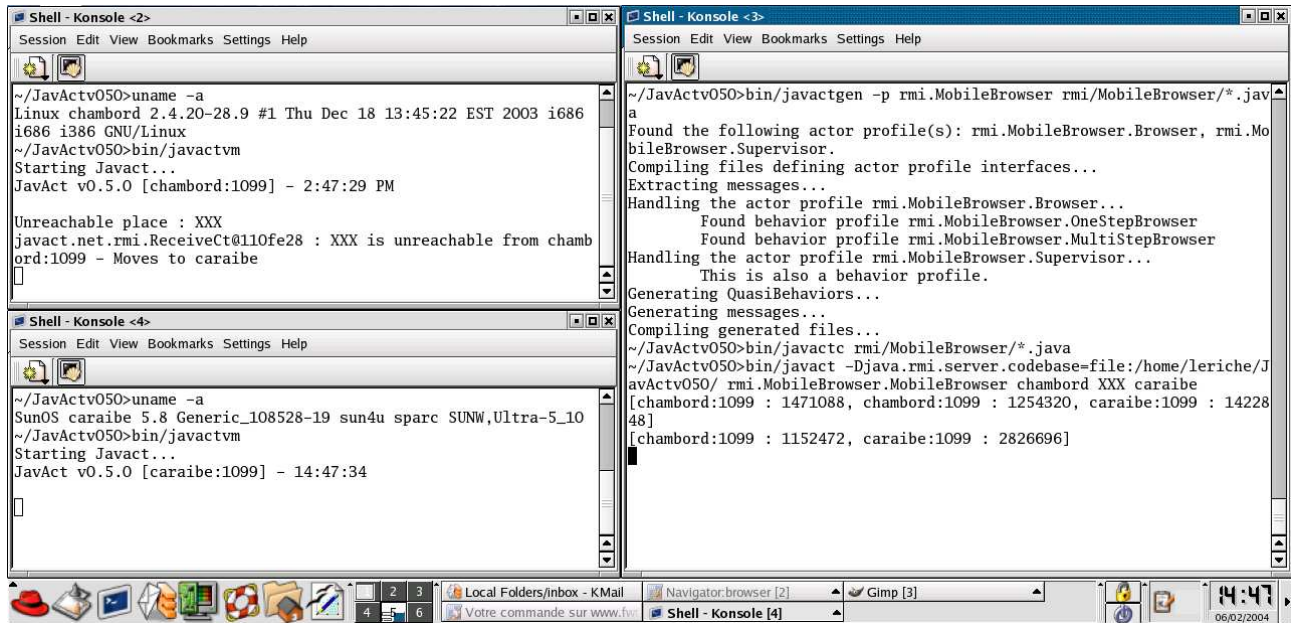


FIG. 5 – Execution du MobileBrowser

## 4.5 Remarques complémentaires

1. Tous les exemples ont été développés pour une exécution en environnement réparti, mais il est tout à fait possible de les exécuter en mode local, c'est-à-dire au sein d'une machine virtuelle unique. Il suffit pour cela de changer au début des fichiers les lignes d'importation `import javact.net.rmi.*`; en `import javact.local.*`. En mode local, les appels à `create(String p, QuasiBehavior b)` et à `go(String p)` restent corrects : la création se fait au sein de l'unique machine virtuelle, et le déplacement se réduit à un clonage local avec lien de poursuite. Le mode local est conseillé pour la mise au point des programmes.
2. La version actuelle offre un mode rudimentaire d'identification des places par une chaîne de caractères. A terme, ceci sera modifié et les places auront un véritable statut. Une évolution de l'architecture répartie basée sur la notion de domaine hiérarchiques (correspondant à une organisation physique et administrative des réseaux et des systèmes, ainsi qu'à une organisation interne des applications) est en cours d'étude [HHP04].
3. La génération automatique des classes pour les messages et leur utilisation systématique permet de s'assurer de l'envoi de messages convenablement constitués grâce au typage standard de



Java. Cependant, cela ne résoud que très marginalement le problème de la détection des messages orphelins. La structure que doit respecter une application JAVACT a été étudiée afin de permettre un certain nombre de contrôles. Elle permettra à terme d'extraire pour chaque acteur son automate de comportement, et de comparer son potentiel de traitement avec ce qui lui sera effectivement envoyé.

4. Dans la version actuelle, avec l'implantation standard de la boîte aux lettres, les messages sont traités dans l'ordre de réception. Un message (orphelin) non traitable par le comportement courant d'un acteur est abandonné après affichage d'un message d'erreur. D'autres politiques peuvent être implantées par redéfinition du micro-composant d'acteur « boîte aux lettres ».
5. Java impose que les méthodes déclarées dans les interfaces des comportements soient publiques. Bien évidemment, il n'est pas conforme au modèle de programmation, et donc pas permis, d'invoquer directement ces méthodes sur l'objet comportement extérieurement à lui-même (sa référence n'est connue que de l'objet qui le crée).
6. On a vu qu'il est possible de définir des acteurs purement fonctionnels qui ne changent pas de comportement. En conséquence du style impératif de Java, il est possible qu'un acteur change d'état sans changer de comportement via un appel à `become()` (cf. l'exemple de l'agent mobile de recherche). Sémantiquement, ne pas exprimer de changement de comportement équivaut à `become(this)` ou à `become(o)` où `o` est un clone du comportement courant.
7. La méthode `becomeAny()` est accessible depuis n'importe quel comportement. Elle permet des changements de comportement non typés. Elle peut permettre à un acteur d'adopter n'importe quel comportement dont il ignore le type (par exemple un comportement reçu dans un message). Hors cas particulier, l'utilisation de `becomeAny()` est néanmoins très fortement déconseillée.
8. Les objets encapsulés dans des comportements déplaçables -par création à distance, mobilité, ou envoi dans un message- (par exemple, les informations dans le cas du `MobileBrowser`) doivent être déclarés *sérialisables* (`implements java.io.Serializable`). Dans le cas contraire, une exception de type `java.io.NotSerializableException` peut être levée au moment du déplacement, empêchant le bon déroulement de l'application en mode réparti.

## Références

- [Agh86] G. AGHA, *Actors : a model of concurrent computation in distributed systems*, M.I.T. Press, Cambridge, Ma., 1986.
- [ALP03] J.-P. ARCANGELI, S. LERICHE, M. PANTEL, Vers des Agents Mobiles Adaptables pour le Partage et la Recherche d'Information, Rapport IRIT/2003-25-R, IRIT, Déc. 2003.
- [AMM01] J.-P. ARCANGELI, C. MAUREL, F. MIGEON, An API for high-level software engineering of distributed and mobile applications, *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems*, Bologna (It.), IEEE-CS Press, 2001, pp. 155-161.
- [BI02] G. BERNARD, L. ISMAIL, Apport des agents mobiles à l'exécution répartie, *Revue des sciences et technologies de l'information, série Technique et science informatiques (RSTI-TSI) 21(6)*, Hermès Science Publications, Lavoisier, 2002, pp. 771-796.
- [CHK94] D. CHESSE, C. HARRISON, A. KERSHENBAUM, Mobile agents : Are They a Good Idea? Technical Report, IBM Research Division, New York, 1994.
- [DP+00] F. DAGNAT, M. PANTEL, M. COLIN, P. SALLÉ, Typing Concurrent Objects and Actors, *L'Objet 6(1)*, Hermès, 2000, pp. 83-106.
- [FPV98] A. FUGGETTA, G.P. PICCO, G. VIGNA, Understanding Code Mobility, *IEEE Transactions on Software Engineering* 24(5), 1998, pp. 342-361.
- [HHP04] A. HURAUULT, V. HENNEBERT, M. PANTEL, Répartition et mobilité en JAVACT : une approche dérivée d'un modèle formel, Actes de la conférence Langages et Modèles à Objets LMO'2004, à paraître, Mars 2004.
- [LA04] S. LERICHE, J.-P. ARCANGELI, Une architecture pour les agents mobiles adaptables, Journées Composants JC'2004, à paraître, Mars 2004.

## A Un exemple de message généré : JAMprint.java

```
public class JAMprint implements javact.lang.Message
{
    private int signatureNumber ;

    private javact.lang.Actor sig0attr0 ;
    private javact.lang.Actor sig0attr1 ;

    public JAMprint(javact.lang.Actor _p0, javact.lang.Actor _p1)
    {
        signatureNumber = 0 ;
        sig0attr0 = _p0 ;
        sig0attr1 = _p1 ;
    }

    private javact.lang.Actor sig1attr0 ;

    public JAMprint(javact.lang.Actor _p0)
    {
        signatureNumber = 1 ;
        sig1attr0 = _p0 ;
    }

    public JAMprint()
    {
        signatureNumber = 2 ;
    }

    public final void handle(javact.lang.QuasiBehavior _behavior)
    {
        switch (signatureNumber)
        {
            case 0 :
                if (_behavior instanceof Hello)
                    ((Hello) _behavior).print(sig0attr0, sig0attr1) ;
                else
                    throw new javact.lang.MessageHandleException() ;
                break ;
            case 1 :
                if (_behavior instanceof Wonderful)
                    ((Wonderful) _behavior).print(sig1attr0) ;
                else
                    throw new javact.lang.MessageHandleException() ;
                break ;
            case 2 :
                if (_behavior instanceof World)
                    ((World) _behavior).print() ;
                else
                    throw new javact.lang.MessageHandleException() ;
                break ;
            default :
                throw new javact.lang.MessageHandleException() ;
        }
    }
}
```

## B Un exemple de comportement généré : EmptyQuasiBehavior.java

```
import javact.lang.QuasiBehavior;

public abstract class EmptyQuasiBehavior extends QuasiBehavior implements Empty
{
    public void become(Full b)
    {
        try
        { becomeAny((QuasiBehavior) b); }
        catch (RuntimeException e)
        { throw new javact.lang.BecomeException(e);}
    }
}
```

## C Primitives pour la programmation de comportements

Cette section présente brièvement les primitives offertes par la bibliothèque JAVACT pour la programmation de comportements d'acteurs. Toutes ces méthodes sont accessibles par héritage (de la classe `QuasiBehavior`) et sont déclarées `final`, il n'est donc pas possible de les redéfinir. Pour plus de détails (notamment sur les types) vous pouvez vous référer à la documentation javadoc du code de la bibliothèque.

### C.1 Création d'un acteur

Ces primitives permettent de créer un acteur dont on récupère la référence, ou `null` si la création a échoué. Dans ce cas, une `Runtime` exception `CreateException` est levée.

```
Actor create(QuasiBehavior b)
```

Création d'acteur sur la place locale à partir du comportement `b`.

```
Actor create(String p, QuasiBehavior b)
```

Même chose, mais création sur une place `p`.

```
Actor create( String p, QuasiBehavior b, MailBoxCtI box, BecomeCtI bec, CreateCtI crt, LifeCycleCtI lif, MoveCtI mve, SendCtI snd)
```

Même chose, mais en donnant l'ensemble des micro-composants qui régissent le fonctionnement de l'acteur.

### C.2 Envoi de message

Ces primitives permettent d'envoyer un message à un acteur. Si l'émission échoue, une `Runtime` exception `SendException` est levée.

```
void send(Message m, Actor a)
```

Envoie le message `m` à l'acteur de référence `a` de manière asynchrone.

```
void send(MessageWithReply m, Actor a)
```

Envoie le message `m` à l'acteur de référence `a`. On *peut* ensuite attendre le résultat avec `m.getReply()`. En cas d'erreur (ré-émission du même message par exemple), une `Runtime` exception `JSMSEndException` est levée. En cas de problème côté récepteur, une `Runtime` exception `JavActException` est levée (côté récepteur inaccessible à l'émetteur).

### C.3 Changement de comportement

Ces primitives permettent de changer le comportement d'un acteur après la fin du traitement du message courant.

```
void becomeAny(QuasiBehavior b)
```

Depuis la version 0.5.0 de JAVACT, les primitives typées de changement de comportement `become(XXXQuasiBehavior b)` sont générées automatiquement d'après la spécification des comportements d'acteurs (cf. §4.1.2). Néanmoins, il reste possible mais déconseillé de faire des changements de comportements non typés (et donc potentiellement moins sûrs) avec cette primitive.

```
void suicide()
```

Permet de terminer l'exécution de l'acteur, qui ne répondra plus à aucun message et ne traitera pas les éventuels messages en attente.

## C.4 Déplacement de l'acteur

Ces primitives permettent de déplacer un acteur vers une place accessible du réseau. Pour des raisons de cohérence, la mobilité n'est effectuée que lors du changement de comportement (soit à la fin du traitement du message courant). L'appel de ces primitives retourne sans erreur liée à la mobilité et le code qui suit leur exécution est toujours exécuté. Par contre, si une erreur se produit durant le déplacement (place inaccessible, problème de réseau...) une `RuntimeException GoException` est levée.

```
void go(String p)
```

Déplacement de l'acteur sur une place `p`, sans gérer les erreurs.

```
void go(String p, HookInterface h)
```

Déplacement de l'acteur sur une place `p`. Si une `GoException` est levée, la bibliothèque redonne la main à la méthode `resume(GoException e)` de l'objet `h`. Cet objet peut être l'instance d'une classe interne du comportement, afin de pouvoir accéder à toutes les informations nécessaires au recouvrement de l'erreur.

## C.5 Localisation et auto-référence de l'acteur

```
String myPlace()
```

Retourne l'adresse de la place sur laquelle l'acteur s'exécute, sous la forme d'une chaîne de caractères (`nom:port`).

```
Actor ego()
```

Retourne l'auto-référence de l'acteur.

## C.6 Adaptation des micro-composants

JAVACT permet l'adaptation dynamique par remplacement de ses micro-composants. Cette adaptation n'est effective lors du changement de comportement.

```
void with(MailBoxCtI box)
```

Remplacement du composant boîte aux lettres.

```
void with(BecomeCtI bec)
```

Remplacement du composant de changement de comportement.

```
void with(CreateCtI crt)
```

Remplacement du composant de création d'acteur.

```
void with(MoveCtI mve)
```

Remplacement du composant de mobilité.

```
void with(SendCtI snd)
```

Remplacement du composant d'envoi de messages.

```
void with(LifeCycleCtI lif)
```

Remplacement du composant de gestion du cycle de vie.