Extensió de l'Entorn SMARD per Permetre la Depuració d'Aplicacions Basades en Agents Mòbils

Memòria del projecte de final de carrera corresponent als estudis d'Enginyeria Superior en Informàtica presentat per Oriol Navarro Ferrer i dirigit per Carles Garrigues Olivella.

Bellaterra, juny de 2008

El firmant, Carles Garrigues Olivella, professor del Departament d'Enginyeria de la Informació i de les Comunicacions de la Universitat Autònoma de Barcelona

CERTIFICA:

Que la present memòria ha sigut realitzada sota la seva direcció per Oriol Navarro Ferrer

Bellaterra, juny de 2008

Firmat: Carles Garrigues Olivella

A la meva família i amics.

Agraïments

Agraeixo al meu director de projecte, Carles Garrigues, tota l'ajuda que m'ha donat i sense la qual no hauria pogut realitzar aquest treball. Gràcies pels consells, la disposició i la paciència.

Vull donar les gràcies als meus pares. Per animar-me a estudiar, per haver-me recolzat sempre i perquè sense ells, res hauria estat possible.

També vull agrair als meus amics tots els moments que hem viscut i viurem. Als amics que he fet durant la carrera, dir-vos que no oblidaré mai aquests anys.

Índex

1 Introducció **1**

1.1 L'entorn SMARD i el desenvolupament d'agents mòbils segurs 2

1.2 Motivacions per a desenvolupar un simulador d'execució per agents mòbils 3

1.3 Objectius 4

1.4 Planificació 5

1.5 Continguts de la memòria 5

2 Fonaments **7**

2.1 Agents mòbils 7

2.1.1 Itinerari d'un agent mòbil 7

2.2 JADE 8

2.2.1 Estructura dels agents mòbils a JADE 8

2.3 Agents mòbils amb itinerari explícit 8

2.3.1 Agents mòbils auto-protegits 9

2.4 La reflexió computacional 9

2.4.1 Funcionament de la reflexió a Java 10

3 Accessos a recursos per a la simulació d'execució **13**

3.1 Anàlisi de requisits i consideracions prèvies 14

3.1.1 Requisits inicials 14

3.1.2 Problemàtica associada als valors de retorn dels mètodes d'accés a recursos 14

3.1.3 Mètodes de depuració implementats pel programador 16

3.1.4 Tractament d'excepcions dels mètodes d'accés a recursos 16

3.2 Disseny 16

3.2.1 Implementacions dels mètodes 16

3.2.2 El valor de retorn dels mètodes 19

3.2.3 Implementació personalitzada dels mètodes en mode simulació 20
3.2.4 Tractament d'excepcions amb la reflexió a Java 22
3.2.5 Estructura de les classes 22
3.3 Implementació 23
3.3.1 Limitacions de la reflexió a Java 24
3.3.2 Implementació del cercador de mètodes 25
3.3.3 Alternatives a la cerca reflexiva de mètodes de Java 25

4 Extensió de l'IDE de disseny d'itineraris per a l'ús del simulador d'execució 29
4.1 Disseny de l'IDE original 30
4.1.1 Model de dades i representació 31
4.1.2 Controlador 32
4.2 Anàlisi 33
4.2.1 Requisits inicials 33
4.2.2 Restriccions de disseny 33
4.2.3 Requisits de la interfície 33
4.2.4 Requisits funcionals 34
4.3 Disseny 34
4.3.1 El panell del depurador 34
4.3.2 Informació de la simulació 35
4.3.3 La comunicació entre l'agent i l'IDE 36
4.3.4 Modificacions a la plataforma JADE del depurador 37
4.4 Implementació 39
4.4.1 Actualització dels tipus de node de la interfície 39
4.4.2 El panell del depurador 39
4.4.3 Informació de la simulació 41
4.4.4 El *thread* de comunicacions del depurador 41
4.4.5 Modificacions a la plataforma JADE 42

5 Integració d'un constructor d'agents mòbils a la interfície de disseny 45
5.1 Descripció del constructor d'agents mòbils amb itinerari explícit 45
5.2 Anàlisi 47
5.2.1 Requisits inicials 47
5.2.2 Requisits de la interfície 47
5.2.3 Requisits funcionals 47

5.3 Disseny 47
5.3.1 Sistema d'accés a recursos per agents amb itinerari explícit 47
5.3.2 Lectura de l'itinerari 48
5.3.3 Integració del constructor dins de l'IDE 48
5.4 Implementació 49
5.4.1 La classe Builder 49

6 Proves **51**
6.1 Proves de funcionament de la interfície gràfica del depurador 51
6.1.1 Preparació de les proves 51
6.1.2 Construir l'agent 51
6.1.3 Llançar l'agent al depurador 52
6.1.4 Visualitzar la informació de simulació a partir del sistema d'accés a recursos 53
6.2 Proves del sistema d'accés a recursos 54
6.2.1 Proves de rendiment del sistema d'accés a recursos 55

7 Conclusions i línies d'ampliació **57**
7.1 Valoració dels objectius 57
7.2 Línies de treball futures 58

Bibliografia **61**

Índex de figures

1.1 Esquema de l'entorn SMARD 3
1.2 Planificació del projecte 5

2.1 Esquema del sistema d'auto-protecció d'un agent mòbil 9
2.2 Esquema del funcionament de la reflexió en Java 11

3.1 Disseny simplificat del sistema 17
3.2 Disseny simplificat del sistema amb els mètodes d'accés que aniria implementant el programador 17
3.3 Disseny simplificat del sistema amb els mètodes d'accés fixats 18
3.4 Classes implementades pel programador dins del package *debugmethods* 22
3.5 Disseny simplificat amb les classes de depuració. Versió definitiva . . 23
3.6 *Wrap* i *unwrap* d'un objecte en Java 24
3.7 Wrap i unwrap d'un tipus primitiu en Java 24

4.1 Interfície de disseny d'itineraris. Versió de 2004 30
4.2 Patró Model-Representació-Controlador 30
4.3 Estructura de la classe *Graph* 31
4.4 Esquema de la classe *Ide* 32
4.5 Diagrama del sistema per desar informació 35
4.6 Esquema de les classes que intervenen a les comunicacions 37
4.7 Diagrama d'interacció agent-depurador 38
4.8 Diagrama de classes del depurador 40
4.9 Comparació de les dues versions de l'IDE 40
4.10 Gràfica de la simulació d'un agent 41

5.1 Diagrama de classes del constructor d'itineraris 46
5.2 Diagrama de classes de la integració del constructor d'agents 49

6.1 Itinerari de proves 52
6.2 Procés de depuració de l'agent 53
6.3 Dades de la finestra emergent 54

6.4 Dades de la finestra emergent (2) 54
6.5 Comparativa del rendiment del sistema d'accés a recursos amb l'execució sense reflexió 55

Capítol 1

Introducció

Des de la popularització de les xarxes de comunicacions trobem que les dades deixen de ser un recurs local. L'aparició d'Internet ha canviat progressivament els hàbits d'individus i organitzacions, i la manera de treballar amb la informació ha anat evolucionant cap a un escenari on aquesta es troba distribuïda en diverses localitzacions. Aquest fet és irreversible, perquè amb la millora de les comunicacions i la facilitat per treballar amb dades remotes, deixa de ser viable que aquestes depenguin d'una estructura centralitzada. Al tenir cada cop volums més grans de dades, és més difícil poder coordinar la distribució de tota la informació des d'un sol punt. També pot ser físicament impossible donar cabuda a tots els recursos que voldríem en infraestructures d'emmagatzemament i control de dades.

Per treballar amb aquests nous escenaris, cal donar respostes que vagin més enllà de les solucions habituals de tractament de dades distribuïdes. En aquest sentit, paradigmes de software com ara el codi mòbil [1] o plataformes de càlcul distribuït (tipus grid [2]) cobreixen algunes de les necessitats que deriven d'aquesta situació. Tot i això, ens trobem amb carències quan, per exemple, no volem ni enviar una peça de software a una situació (codi mòbil) ni executar una gran tasca repartida en fragments a màquines remotes (grid computing). Aquestes solucions es centren en una distribució del codi executable del programa amb la possibilitat d'incloure-hi dades per a la seva execució local.

Una alternativa a aquests paradigmes són els agents mòbils. Un agent mòbil és un tipus de programa que basa el seu funcionament en una execució distribuïda, amb la característica diferencial que la distribució en aquest cas és de les dades, del codi del programa i del seu estat. L'agent anirà buscant aquestes dades per diverses plataformes on podrà anar migrant seguint un itinerari. L'agent mòbil migra d'una manera autònoma, amb l'ajuda de la plataforma. Pren les seves pròpies decisions, emportant-se les seves tasques, formades per codi i dades, i el seu estat. Un cop migra, té l'habili-

tat de poder seguir executant-se a la següent plataforma que visita. A més, els agents mòbils poden incloure funcionalitats que els permetin fer migracions de dades segures entre plataformes, i el fet de no ser vulnerables a diversos tipus d'atacs que s'han anat desenvolupant. Tot i ser una opció relativament nova, compta amb una acceptació cada cop més gran i amb l'IEEE-FIPA que s'encarrega d'estandarditzar-ne alguns aspectes. S'ha demostrat que aquest tipus de programes són molt útils i efectius en certs entorns, com el comerç electrònic o les aplicacions mar-de-dades [3]. A [4] i [5] es descriuen aplicacions per al tractament i obtenció d'informació mèdica mitjançant agents mòbils i la seva idoneïtat en aquestes tasques.

Un dels principals obstacles per a l'adopció d'aquest paradigma de software per part d'organitzacions o entitats per a qui aquest tipus de programes seria de gran ajuda, és que el desenvolupament d'agents mòbils segurs és una tasca difícil. S'ha de tenir en compte multitud de paràmetres entre quals hi tenim arquitectures de seguretat, opcions de migració o protecció d'itineraris. Això fa que sigui necessari tenir coneixements avançats en la matèria per tal de poder generar agents segurs amb les característiques que volem donar-los. Des del grup de recerca SeNDA s'està desenvolupant l'entorn SMARD (Secure Mobile Agent Rapid Development) per intentar donar un conjunt d'eines per fer, entre d'altres, aquest desenvolupament molt més lleuger i ràpid, permetent fer-lo més accessible.

1.1 L'entorn SMARD i el desenvolupament d'agents mòbils segurs

SMARD (Secure Mobile Agent Rapid Development) és un entorn per permetre un desenvolupament simplificat d'agents mòbils segurs.

Actualment trobem diverses iniciatives per facilitar aquest tipus de desenvolupament. Les podem dividir en dos grups. El primer es basa en l'ús de llenguatges declaratius per a facilitar la programació dels agents [6], el segon grup es basa en entorns de desenvolupament, que proveeixen ajuda al programador de les aplicacions i on també hi englobem la construcció d'agents basada en patrons de disseny [7].

L'SMARD simplifica la protecció de l'agent mitjançant arquitectures criptogràfiques. En canvi, els enfocs presentats fins ara tendeixen a simplificar el disseny i la programació de la tasca de l'agent.

L'element clau d'SMARD és el constructor d'agents. El seu funcionament es basa en crear un agent a partir d'un itinerari descrit en XML i una descripció dels mecanismes de protecció que tindrà aquest agent mitjançant un llenguatge declaratiu anomenat MACPL (Mobile Agent Cryptographic Protection Language). A partir d'aquests dos

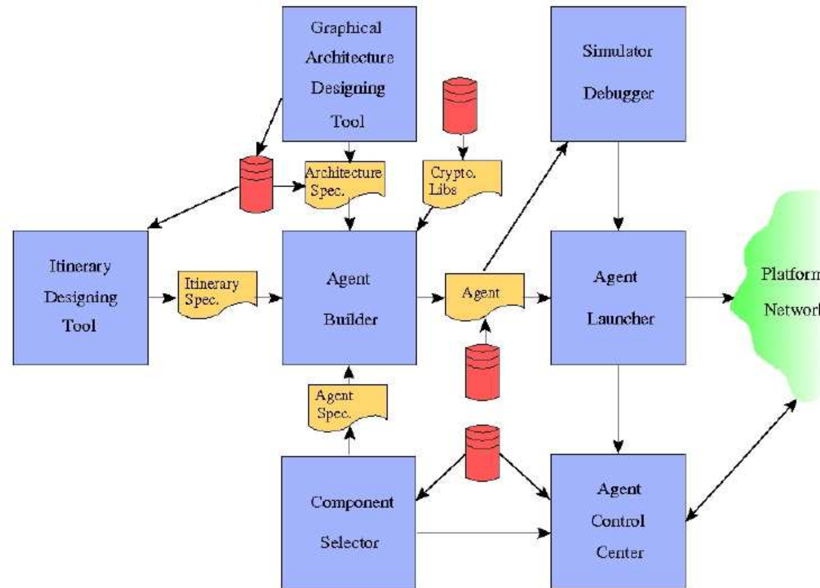

Figura 1.1: Esquema de l'entorn SMARD

elements es construeix un agent mòbil segur en llenguatge Java [8]. Cal esmentar que aquest constructor d'agents no està plenament desenvolupat, i només es disposa d'una prova de concepte. Podem afirmar que SMARD conté característiques del primer i segon grups de propostes. Tenim, per exemple, una eina gràfica per a definir els itineraris i fer que sigui més senzill traçar el recorregut que l'agent farà, un llançador d'agents, i components per al control de l'execució de l'agent i la tolerància a fallades. A la figura 1.1 podem veure l'esquema de l'entorn amb tots els seus components.

Considerant el conjunt d'elements dels que disposa SMARD, més els que està previst desenvolupar, podem dir que ara mateix és un entorn de desenvolupament d'agents mòbils segurs molt avançat. El fet de disposar d'agents protegits i amb un desenvolupament més fàcil i ràpid hauria de fer que més organitzacions tinguin en compte aquesta tecnologia, que fins ara no ha arribat al grau d'utilització que podria esperar-se'n.

1.2 Motivacions per a desenvolupar un simulador d'execució per agents mòbils

El principal motiu que fa necessària una aplicació, integrada a SMARD, que ens permeti fer una simulació d'execució o depuració (en anglès el terme és *debugging*) d'un agent mòbil abans de llençar-lo en un escenari real, és fer-ne proves per poder observar el seu comportament.

L'execució a plataformes remotes pot causar certes inconveniències com ara no

poder observar cap dels estats que ha tingut l'agent si no torna al seu origen i sobretot el fet de deixar el programa bloquejat a un sistema d'una altra organització. Aquest últim efecte és especialment greu, perquè provoca que l'administrador de la plataforma en qüestió hagi de monitoritzar i prendre mesures per solucionar-ho. Fent unes proves prèvies del comportament dels agents abans de fer-los servir en el seu entorn objectiu podríem trobar errors de programació i limitar aquests efectes. Així també facilitem la feina del programador que podrà observar fàcilment l'execució de l'agent, i podrà fer canvis al codi de l'agent d'una manera molt més efectiva.

Alguns paràmetres que ens interessaria observar són: Si itinerari és realment viable i si les seves tasques associades funcionen correctament, el temps d'execució que tindrà l'agent en cada plataforma i el total, o bé el seguit d'alternatives que ha anat prenent durant el seu itinerari. Totes aquestes dades faran que a més d'un depurador, també comptarem amb un sistema de *profiling*, d'obtenció i anàlisi d'informació de la simulació d'agents mòbils, que serà molt interessant de cara a estudiar el seu comportament dins l'entorn de proves.

Hi ha protocols de seguretat per agents mòbils que requereixen d'una simulació prèvia per obtenir una traça de les accions que fa l'agent. Quan l'agent s'executa realment, s'obté una altra traça, que es compara amb la inicial per saber si l'execució ha estat l'esperada [9]. La solució que oferim ens permetria fer ús d'aquest tipus de sistemes.

1.3 Objectius

L'objectiu d'aquest Projecte de Final de Carrera és afegir a l'entorn SMARD un depurador per a agents mòbils que ens permeti fer una execució simulada i observar el comportament dels agents. La intenció seria integrar-lo a la interfície de descripció gràfica d'itineraris, integrada a SMARD, que va desenvolupar en Jordi Nebot al seu Projecte Final de Carrera l'any 2004 [10]. Amb aquesta integració podrem conservar la visualització de l'itinerari d'un agent. El programador podrà aprofitar la mateixa interfície per desenvolupar el seu itinerari i provar-lo seguidament.

Com a extensió, integrarem un constructor d'agents. Amb aquest element l'usuari pot crear, des de la interfície, un agent a partir de l'itinerari que ha dibuixat.

Des de 2004 hi ha hagut canvis a SMARD i en com es defineixen els agents mòbils dins d'aquest entorn. El canvi que més afectarà aquest projecte és que els tipus de nodes que pot contenir un itinerari han canviat. Per tant, un objectiu també important serà actualitzar la interfície tant com sigui possible i deixar-la preparada per poder treballar-hi amb la resta de components d'SMARD.

1.4 Planificació

Per organitzar la feina a fer durant el cicle de vida del projecte, és indispensable marcar certes etapes i fites. La planificació que hem seguit durant el desenvolupament es pot veure al Diagrama de Gantt, figura 1.2.

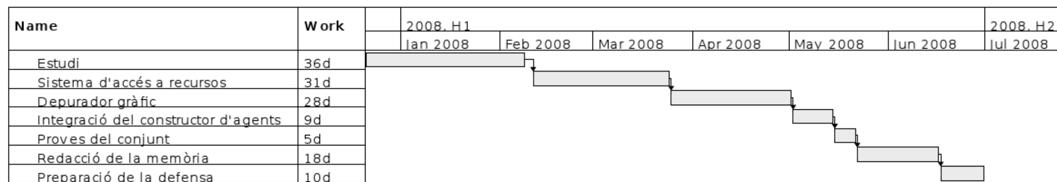

Figura 1.2: Planificació del projecte

Cadascuna d'aquestes parts del projecte seguirà diverses etapes: Anàlisi, disseny i implementació. Al final hi ha una fase de proves del conjunt.

Les parts de desenvolupar i integrar el depurador gràfic, i el sistema d'accés a recursos són les que tenen més temps assignat. Vam pensar que serien les més complexes, que requeririen un disseny més extens i una fase d'implementació més complicada. Per la redacció de la memòria hi vam posar un marge d'unes dues setmanes, que haurien de ser suficients.

1.5 Continguts de la memòria

Aquest projecte difereix de l'esquema habitual d'altres on es duen a terme solucions *software*, en que s'acostuma a seguir un esquema de desenvolupament reflexat en capítols per cada fase: Anàlisi, disseny, implementació i proves. En aquest cas hem de desenvolupar dos components molt diferenciats, que necessitaran les seves etapes d'anàlisi, disseny i implementació pròpies. La tercera part del projecte serà una extensió per a donar més completesa al conjunt i també es descriu en un capítol propi. Seguidament expliquem amb brevetat quin serà el contingut de cada un dels capítols:

- **Capítol 2: Fonaments.** En aquest capítol intentarem donar una base de coneixements per al lector no avesat a la matèria per tal que pugui seguir els continguts dels capítols següents de la memòria. Explicarem què són els agents mòbils, JADE i la reflexió computacional a Java.

- **Capítol 3: Accessos a recursos per a la simulació d'execució.** L'accés dels agents a recursos de la plataforma quan s'estan executant a l'entorn de simulació s'ha de tractar com un cas especial. Durant el capítol farem l'anàlisi, el disseny

i la implementació de la solució necessària per tal que l'execució simulada no falli.

- **Capítol 4: Extensió de l'IDE de disseny d'itineraris per a l'ús del simulador d'execució.** Durant l'anterior capítol s'han desenvolupat els elements necessaris per fer que un agent pugui executar-se en una plataforma local de proves sense que això comporti cap problema. En aquest capítol intentarem completar la feina afegint un entorn gràfic per la simulació. Per aquesta part del projecte explicarem amb brevetat el funcionament de la versió prèvia de la interfície i també inclourem les etapes d'anàlisi, disseny i implementació de l'ampliació.
- **Capítol 5: Integració d'un constructor d'agents mòbils a la interfície de disseny.** Aquesta tercera part del desenvolupament del projecte és una ampliació que permetrà utilitzar el depurador des de la mateixa interfície gràfica sense haver de fer servir cap altre programa separadament. Es podran construir agents mòbils utilitzant la mateixa IDE. Descriurem el constructor que farem servir i la seva integració dins de la interfície gràfica.
- **Capítol 6: Proves.** Farem proves de l'aplicació, amb tots els elements integrats.
- **Capítol 7: Conclusions i línies d'ampliació.** S'hi farà una valoració dels objectius acomplerts, a més de proposar línies de treball futures basades en aquest projecte.

Capítol 2

Fonaments

En aquest capítol descriurem el conjunt de conceptes i eines que són necessaris per entendre el treball fet en el nostre Projecte de Final de Carrera.

2.1 Agents mòbils

Un agent *software* és una entitat que actua amb un cert grau d'autonomia per acomplir uns objectius [11]. L'agent és autònom, reactiu i proactiu. Són capaços de percebre i reaccionar, relacionar-se amb el seu entorn i prendre les seves pròpies decisions sobre la seva execució, sense cap intervenció humana. La seva activitat va relacionada amb el seu entorn, la plataforma.

Si un agent és capaç de desplaçar-se a d'altres entorns per continuar la seva execució, diem que es tracta d'un agent mòbil. Quan l'agent es mou s'emporta les seves tasques, formades per codi i dades, i el seu estat. Aquesta acció de desplaçar-se d'una plataforma a una altra s'anomena *migració*.

2.1.1 Itinerari d'un agent mòbil

L'itinerari d'un agent mòbil és el conjunt de plataformes (o nodes) que ha de visitar durant la seva execució per acomplir els seus objectius [12]. Aquesta ruta és flexible, ja que l'agent mòbil és capaç d'actuar segons les condicions que tingui implementades. Els itineraris flexibles poden incloure diferents tipus de nodes, com per exemple:

- Seqüències: En aquests nodes, l'agent executarà la tasca assignada i migrarà a la plataforma següent.

- Alternatives: Nodes als que l'agent pot tenir diverses destinacions possibles. Aquest node té associat un mètode de decisió que permet a l'agent triar el següent node que visitarà.

- Bucles: Aquest tipus de node farà que hi hagi iteracions sobre el conjunt de plataformes que agrupa.

- Descobriment: Permet descobrir noves plataformes que no són a l'itinerari i afegir-les-hi.

2.2 JADE

FIPA (Foundation for Intelligent Physical Agents) [13] és una organització dins de l'IEEE (Institute of Electrical and Electronics Engineers) [14]. Promou les aplicacions basades en agents i la seva interoperabilitat amb altres tecnologies.

Una de les especificacions de FIPA són estàndards per plataformes d'agents. De les plataformes que compleixen aquest estàndard, el grup SeNDA(Security of Networks and Distributed Applications) [15], dins del qual està emmarcat aquest projecte, treballa amb JADE (Java Agent DEvelopment framework) [16]. Aquest entorn d'execució d'agents és de codi obert i basat en Java.

2.2.1 Estructura dels agents mòbils a JADE

Un agent mòbil de JADE es desenvolupa fent una classe principal que hereta de la classe *Agent*, proporcionada per l'entorn. Aquesta classe principal contindrà uns *behaviours*, que representen els diferents comportaments de l'agent. Els comportaments seran les tasques que realitzarà l'agent, com per exemple, la consulta d'un fitxer de dades.

2.3 Agents mòbils amb itinerari explícit

A la secció anterior hem vist què són els agents mòbils, quin entorn farem servir per desenvolupar-los i quina és la seva estructura. En aquest apartat descriurem un cas particular d'agents mòbils, els que utilitzen un itinerari explícit.

Hi ha dues possibilitats per definir l'itinerari d'un agent mòbil [12]:

- Itinerari implícit: Les instruccions de migració s'intercalen amb les tasques de l'agent en un mateix codi.

- Itinerari explícit: Hi ha una estructura externa on es defineix la ruta i les tasques de l'agent. Aquest itinerari explícit estarà format per nodes que que tenen associats una plataforma, un codi i unes dades.

2.3.1 Agents mòbils auto-protegits

A [17] es descriu un sistema d'agents mòbils auto-protegits, basat en que els agents puguin gestionar els seus propis mecanismes de protecció. Aquest sistema permet a les plataformes comprovar la integritat dels agents, i desxifrar les seves dades utilitzant la clau privada de la plataforma.

Aquest tipus d'agents fan servir un codi de control, que gestiona l'execució de les tasques de l'itinerari explicit. En aquest itinerari, les dades i el codi de la tasca local estan habitualment xifrats amb la clau pública de la plataforma que es vol que hi pugui accedir. Només aquesta, amb la seva clau privada podrà extreure la tasca local i executar-la. A la figura 2.1 podem veure un esquema del funcionament d'aquest sistema.

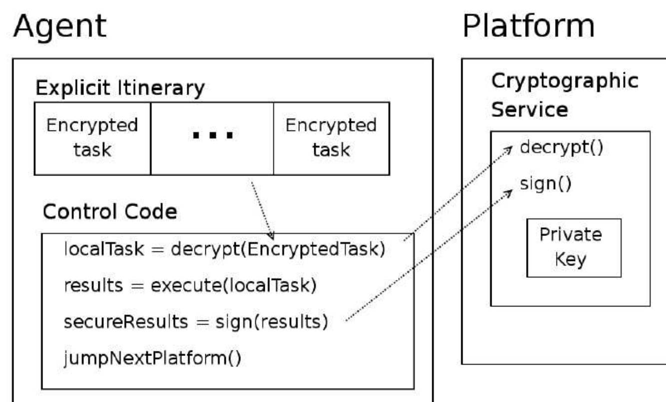

Figura 2.1: Esquema del sistema d'auto-protecció d'un agent mòbil

Tal i com es veu a la figura, les plataformes proporcionen als agents un servei públic de desxifrat de dades, per tal que els agents puguin extreure informació protegida de l'itinerari explícit sense tenir accés directe a la clau privada de la plataforma.

2.4 La reflexió computacional

La reflexió suposa que el programa pugui interpretar les seves dades, de manera que es modifiqui a si mateix en temps d'execució. Habitualment, sense reflexió, el programador escriu el seu codi font, que serà executat tal i com és escrit perquè el codi màquina (o bytecode intermedi en cas de Java) es genera en temps de compilació. En canvi, utilitzant aquesta característica del llenguatge, podem fer que el comportament del programa sigui modificat dinàmicament en temps d'execució.

L'ús en aquest projecte es restringirà a la reflexió per a la invocació de mètodes i constructors, encara que aquesta propietat té moltes altres possibilitats i aplicacions.

2.4.1 Funcionament de la reflexió a Java

En general, el funcionament de la reflexió a Java es basa en la capacitat d'un programa per obtenir informació sobre mètodes, classes i constructors de si mateix. Les classes *Method*, *Class* i *Constructor* implementen aquests conceptes. Podem aconseguir la instància d'una classe *Class* des de qualsevol objecte o a partir del seu nom. Per exemple:

```
String cadena = new String("abc");
Class classe = cadena.getClass();
```

Com es veu a l'exemple, a partir d'aquesta instància de *Class* podem trobar característiques de la classe *String*. A l'API de Java hi veiem mètodes que ens serviran per trobar els seus camps (*getFields()*) o una llista amb els seus constructors (*getConstructors()*), entre molts d'altres.

Per buscar un mètode de la classe, tenint el nom d'aquest i una llista amb els seus tipus de paràmetres, faríem (on *integer.TYPE* representa un tipus primitiu a Java):

```
Method metode = classe.getMethod("charAt",
                                 new Class{integer.TYPE});
```

Amb aquesta línia de codi obtenim un objecte *Method*, que representa el mètode *charAt(int)* de la classe *String*. La classe mètode dóna informació i accés a un únic mètode d'una classe o interfície. Aquest mètode el podem cridar de la següent manera:

```
metode.invoke(cadena,1);
```

Aquesta crida invoca el mètode que representa "metode" sobre l'objecte de tipus *String* "cadena" i donaria com a resultat 'b'. Per fer-ho més entenedor, es pot representar gràficament el mateix procés, com a la figura 2.2.

Un altre exemple d'utilització de la reflexió, en aquest cas orientada a constructors, seria obtenir una nova instància d'un objecte tenint només el nom de la classe. Tal i com mostra el codi següent, es crea una nova instància de la classe *String*, se n'obté un objecte de tipus *Constructor* i a partir d'aquest últim, es fa una nova instància.

```
Class classe = Class.forName("java.lang.String");
Constructor const =
            classe.getConstructor(new Class{String.class});
String cadena = const.newInstance("abc");
```

Podem veure que la diferència principal de la classe *Constructor* respecte a *Method* és el seu mètode *newInstance(Object[] args)* enlloc de *invoke(Object obj, Object[] args)*. La finalitat de fer la crida sobre si mateix és la mateixa, però per *Constructor* obtenim un objecte nou.

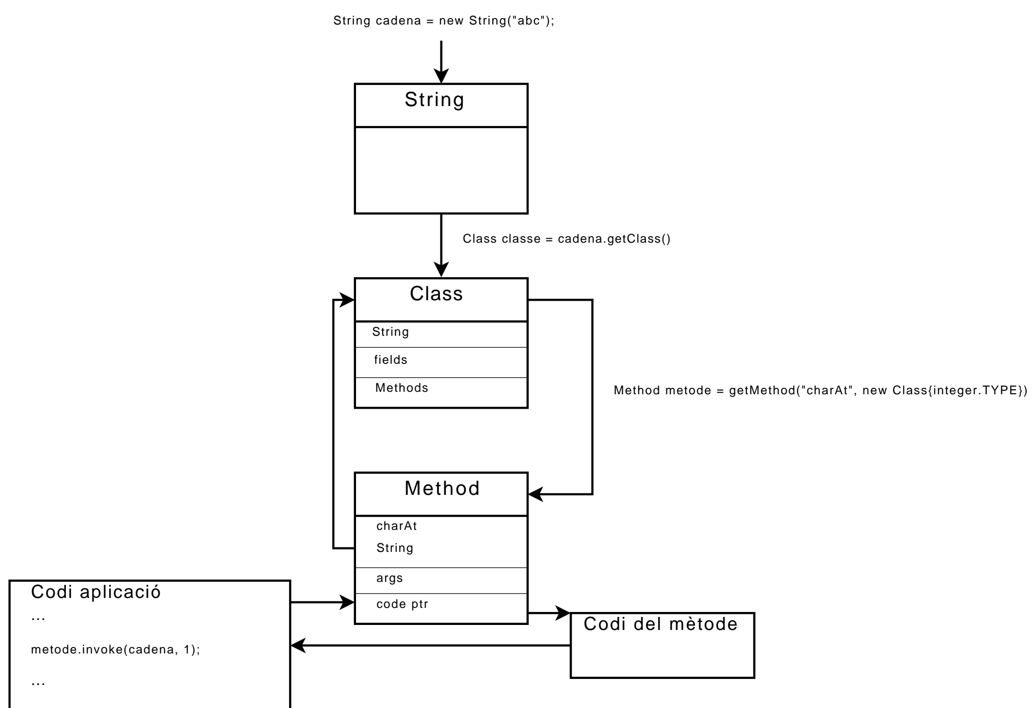

Figura 2.2: Esquema del funcionament de la reflexió en Java

Capítol 3

Accessos a recursos per a la simulació d'execució

Durant una execució normal, un agent mòbil acostuma a intentar accedir a recursos o dades de la plataforma que es troba.

Ens podem trobar que l'agent del que en simulem l'execució busqui l'accés a un recurs que en principi nosaltres no podem proporcionar dins de la plataforma local que utilitzarem com a entorn per les proves. Es podria considerar poder oferir recursos de manera simulada, donar uns arxius preparats pel programador des de l'entorn de proves, però només funcionaria en casos on el recurs esperat fos ben simple, com ara fitxers de text. Això és degut al fet que no podem demanar que per fer una prova de simulació d'execució s'hagi de preparar estructures complexes, com un servidor web o una base de dades amb contingut útil. Com que aquesta opció és impossible, l'alternativa és proporcionar una solució per al programador d'agents mòbils segurs on es pugui diferenciar entre els dos modes d'execució possibles, l'execució real i la simulació, i que resolgui la problemàtica derivada dels accessos a recursos.

Fins ara el sistema utilitzat per a aquests accessos de l'agent a la plataforma on es troba era cridar els mètodes que implementen aquests accessos directament al codi de l'agent. Pel mode d'execució real, en el qual el previsible és que trobem el recurs i, en cas contrari, sabem que es tracta d'un error, és suficient. En canvi, considerant el mode de depuració, cal tenir algun altre sistema per controlar els mètodes del programa que fan un accés cap a recursos de la plataforma. En aquest capítol presentarem un sistema de simulació d'accés a recursos, que facilitarà desenvolupar el comportament de l'agent en els dos modes d'execució: el mode real i el mode de simulació.

3.1 Anàlisi de requisits i consideracions prèvies

De les consideracions de la introducció d'aquest capítol en podem treure algunes de les idees que aquest sistema ha d'acabar implementant i que completarem en aquesta secció.

3.1.1 Requisits inicials

El requisit principal és que l'agent ha de poder executar els mètodes d'accés a recursos quan estigui en mode simulació, sense que això provoqui errors.

El programador ha de poder triar i diferenciar entre els dos modes d'execució, fent el menor número possible de canvis al programa. Es tracta de donar una nova eina al programador, no de perjudicar-lo afegint dificultats. No pot ser que quan vulguem que l'agent s'executi d'una manera haguem de fer servir uns mètodes i que al voler utilitzar l'altre mode d'execució sigui necessari reescriure'ls o canviar-los. Això impedeix que es puguin fer proves del codi còmodament o pitjor encara, que s'introdueixin errors al canviar d'un mode a l'altre. El requisit és que al principi del codi de l'agent sigui possible establir aquesta opció i que a la resta de codi no s'hi hagi de fer cap canvi.

No totes les crides a mètodes han de passar per aquest sistema, sinó només les que el programador cregui necessàries. El programador ha de tenir la llibertat de decidir quan utilitzar aquest sistema.

També cal considerar el cas de les migracions. Hem dit que executaríem l'agent en un entorn de proves situat en una plataforma local. Si volem mantenir aquesta condició caldrà que quan l'agent executi els mètodes que el fan migrar, nosaltres ho detectem i evitem aquesta situació.

És previsible, tot i que caldrà minimitzar-ho tant com sigui possible, que aquest sistema tingui un impacte negatiu al rendiment de l'agent. Fent que tots els mètodes del codi de l'agent l'utilitzin, l'impacte seria màxim. En canvi, deixant decidir al programador quins mètodes passaran pel sistema, minimitzem aquest efecte.

La plataforma d'agents que s'utilitzarà és JADE, i els agents es programaran amb llenguatge Java. Aquest requisit ve donat pel treball previ en que es basa aquest projecte, que hem explicat al capítol 2.

3.1.2 Problemàtica associada als valors de retorn dels mètodes d'accés a recursos

La primera qüestió a tractar és el comportament dels mètodes d'accés a recursos quan l'execució és simulada. El que sabem segur és que no podem comptar amb trobar

els recursos que busquem a la plataforma de proves i per tant caldrà que el programa entengui aquesta situació i continuï executant-se.

En mode d'execució real, els mètodes han de poder executar-se de forma normal, però per al mode de simulació farà falta que aquests mètodes, per defecte, retornin un valor neutre que el programador pugui interpretar. Aquest valor neutre indicarà que l'acció s'ha simulat correctament. El programador podrà interpretar aquest valor neutre de la manera que li convingui. D'aquesta manera la simulació pot continuar normalment.

Aquesta solució, no obstant, pot tenir problemes. El valor de retorn dels mètodes que accedeixen a recursos, en molts casos, servirà per decidir quina serà l'acció següent a fer. Normalment el valor de retorn podria permetre al programador distingir el mode d'execució real del de simulació, i així fer les accions que corresponguin en cada cas. No obstant, per qualsevol valor neutre de retorn que establim per a la depuració, és possible que aquest coincideixi amb el que retornen alguns mètodes de Java quan l'acció que havien de dur a terme ha fallat. En aquest cas ens trobaríem que no hi ha manera de distingir els dos modes d'execució, ens trobaríem amb un conflicte. En un dels modes el valor vol dir una cosa i en l'altre potser vol dir justament el contrari.

Potser, en mode simulació, si es rep el valor neutre es seguiria normalment. En canvi, quan executéssim l'agent en plataformes remotes, aquest prendria la decisió d'acabar el seu itinerari. Aquesta situació provocaria que l'agent fallés, perquè al fer l'accés que donés lloc a aquest problema, el codi de l'agent sempre el faria continuar i en canvi originalment la situació era d'error.

D'aquí establim que a més de retornar un valor caldrà tenir algun mecanisme per saber en quin mode ens trobem des de qualsevol punt del programa i que això permeti decidir finalment què fer.

Un exemple de la situació que hem descrit pot ser la del següent codi. L'agent està funcionant en mode d'execució real en una plataforma remota. Aleshores executa un mètode d'accés a recursos i aquest falla. El retorn és *null*. Suposadament, hauria de finalitzar, però en canvi segueix.

```
neutre = null;
retorn = executarmètodeaccés();

si (retorn == neutre)      //mode de simulació
      continuar();
sinó si (retorn == null)    //mode d'execució real
      finalitzar();
```

En aquest cas veiem que sempre que es rebi el valor *null*, l'agent pensarà que està en mode de simulació i continuarà.

3.1.3 Mètodes de depuració implementats pel programador

També cal incorporar alguna altra solució per tal que el programador pugui establir un valor de retorn personalitzat, i fer el retorn de valors dels mètodes de simulació més flexible. Aquesta característica pot ser necessària per algunes de les tasques que s'hagi de programar. Per exemple, quan s'accedeix a una base de dades en mode simulació, el programador pot decidir retornar unes dades per defecte.

Aquest fet segurament estalviarà que el codi de simulació es barregi amb el codi d'execució real, simplificant la programació de l'agent.

3.1.4 Tractament d'excepcions dels mètodes d'accés a recursos

Hi ha la possibilitat que alguna de les accions que es vol fer llanci una excepció. És important tenir en compte aquest aspecte. Al cas de simulació no té cap sentit que falli l'accés a un element que hem assumit que no existeix. Per tant la solució escollida no ha de llançar excepcions en mode depuració.

Tot i això, com és evident, quan estigui fent l'execució real sí que les ha de poder llançar. Per tant, el sistema d'accés a recursos que realitzem també haurà de tenir presents les diferències entre els dos casos per a aquesta característica.

3.2 Disseny

El disseny d'aquesta part del programa ha de tenir en compte els dos modes d'execució. Per això hi haurà una classe *ResourceAccess* que tindrà dues especialitzacions: *ExecuteResourceAccess* i *DebugResourceAccess*. Aquesta manera de representar les classes és la més còmoda pel programador perquè quan vulgui fer servir un tipus d'execució només haurà d'instanciar la classe ResourceAccess com a una de les dues possibilitats a l'inici del codi on l'hagi de fer servir. La representació la trobem a la figura 3.1.

3.2.1 Implementacions dels mètodes

Una primera possibilitat per establir les implementacions dels mètodes per als dos modes d'execució és donar al programador l'estructura de classes buida en la qual hagi de posar-hi tots els mètodes d'accés a recursos que el seu agent necessita.

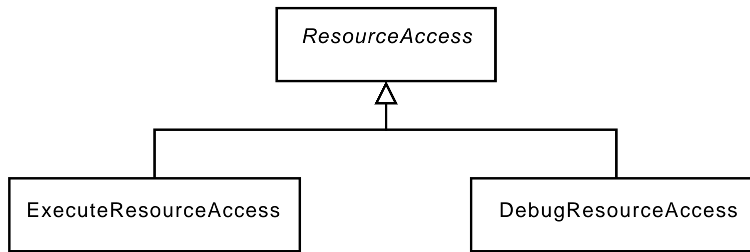

Figura 3.1: Disseny simplificat del sistema

Els mètodes de la part d'execució que el programador implementarà contindran el codi font necessari per realitzar cada una de les accions que l'agent necessitarà fer a les plataformes remotes. Per exemple: mètode per l'accés a una base de dades MySQL, mètode per l'accés a un fitxer de text simple, etc. Els mètodes de depuració seran una versió simplificada que no executaran cap funció real. A la figura 3.2 n'hi ha un exemple.

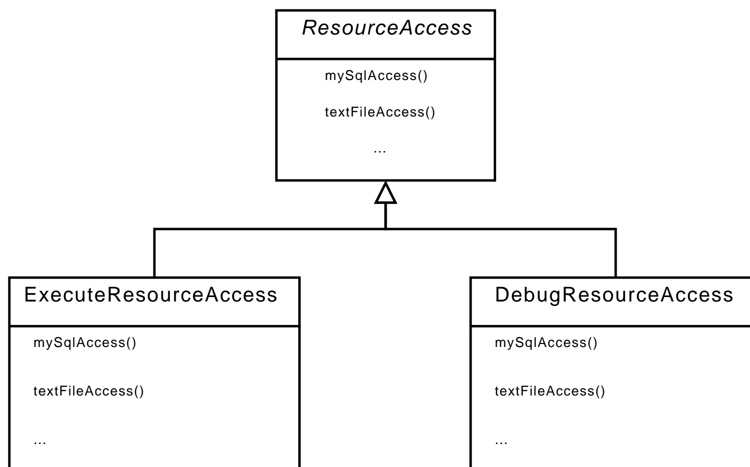

Figura 3.2: Disseny simplificat del sistema amb els mètodes d'accés que aniria implementant el programador

Un altre opció és donar uns mètodes genèrics d'accés segons el tipus de recurs. Tindríem uns quants mètodes que farien accessos concrets. Per exemple, podríem permetre només accessos a fitxers de text i bases de dades MySQL. De tal manera que proporcionaríem mètodes fixats d'accés a aquests recursos: *textFileAccess()* i *mySqlAccess()*.

A les dues possibilitats anteriors hi ha problemes similars. Deixem massa feina en mans del programador. No només ha d'implementar el codi d'accés com ja feia abans, també ha de preocupar-se de posar aquest codi en una estructura de classes manualment. A més, també ha d'implementar els mètodes de depuració, que si bé serien buits i només haurien de retornar un valor neutre, cal tenir-los en compte i posar-los

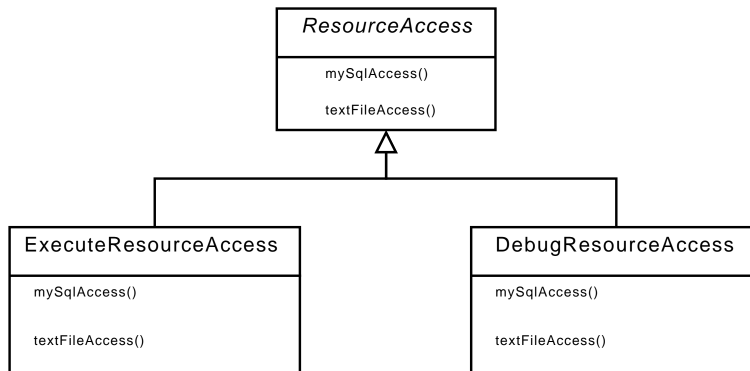

Figura 3.3: Disseny simplificat del sistema amb els mètodes d'accés fixats

a la classe *DebugResourceAccess*. Fent servir la segona possibilitat, amb els mètodes genèrics, sembla que reduïm aquesta feina, però és impossible arribar a considerar tots els possibles tipus d'accessos a recursos diferents.

Qualsevol opció que consisteixi en traduir el codi habitual en altres peces genèriques en forma d'interfície per l'accés patirà els mateixos problemes. El que busquem és una solució completament general, i la solució anterior es queda a mig camí.

El llançador generalitzat de mètodes

La solució completament general s'ha acomplert fent un *llançador generalitzat de mètodes*. Es tracta d'un mètode de *ResourceAccess* que pot executar qualsevol mètode del llenguatge Java que li demanem. Aquest mètode rebrà l'objecte sobre el que volem executar-lo, el nom de la classe on és el mètode, el seu nom i una llista amb els paràmetres. Pels constructors caldrà fer un altre mètode, tot i que molt similar, perquè no es criden sobre cap objecte. Al cas d'execució farà realment la crida, i per la simulació simplement retornarà *null*. El fet de passar-li el nom de la classe on hi ha el mètode pot no semblar estrictament necessari, però si no fós així perdríem la possibilitat de fer servir mètodes que tenen el mateix nom i implementacions a diverses classes. Un exemple són els mètodes de les superclasses de la classe a la que pertany qualsevol objecte, i que també són compatibles amb ell.

Fent servir aquesta solució, descarreguem de feina el programador, que no haurà d'intervenir al sistema d'accés a recursos. Només caldrà que tradueixi la crida del mètode que faria normalment a aquest de nou que li proporcionem. De tota manera, sempre es podran afegir altres mètodes personalitzats i la decisió final sobre què farà servir recau al programador. Aquesta seria una solució mixta amb les primeres, on el programador implementa els seus propis mètodes. Podria decidir, per exemple, fer algun tipus d'implementació al cas de depuració si per al seu programa fes falta algun

valor en concret, sense tenir això cap efecte a la part d'execució.

El tipus de retorn del mètode haurà de ser *Object*, perquè és l'única manera que sigui completament general. Quan el programador faci la crida sempre haurà de fer un cast cap al tipus que espera. El tipus de la llista de paràmetres que li passem serà un array d'*Objects*, pel mateix motiu.

El mètode s'anomena *executeMethod*, i la seva signatura és:

```
public Object executeMethod(Object baseobject, String classname,
String methodname, Object[] args)
```

En principi, el procés que segueix *executeMethod* és:

1. Obtenir una instància de la classe de la que en passem el nom.
2. Buscar-hi el mètode, segons el nom i els tipus dels arguments que li passem.
3. Fer la crida del mètode que hem trobat amb l'objecte sobre el que l'hem de cridar i els arguments que hem passat per paràmetre.
4. Retornar el resultat de la crida.

El cas del constructor és semblant al del mètode. La diferència principal és que els constructors no s'executen sobre cap objecte, sinó que en retornen un. Això fa que necessitem un altre mètode que s'anomenarà *executeConstructor* i que rep com a paràmetres el nom de la classe on s'ha de buscar el constructor i els arguments. La seva signatura serà:

```
public Object executeConstructor(String classname,
                                        Object[] args)
```

3.2.2 El valor de retorn dels mètodes

Tot i fer servir aquest llançador de mètodes, encara cal establir quin podria ser el valor de retorn que haurà de permetre distingir el mode d'execució del programa. Aquest valor de retorn també ha de fer saber al programador si l'execució ha estat correcte o ha donat errors.

Com hem discutit a la fase d'anàlisi i consideracions, només sabent el valor de retorn no n'hi ha prou. Podria ser que aquest fós el retorn per un mètode de Java quan falla, i per tant tindríem una ambigüitat. Farà falta algun mecanisme o sistema que permeti saber en quin mode d'execució ens trobem i tractar els retorns dels mètodes d'accés a recursos d'una manera diferent.

Un exemple molt simple del tractament de valors seria:

```
valor = executeMethod(paràmetres);
si (mode == debug)
    segueix();
sinó si (mode == execució)
    si (valor == v1)
        acció1();
    sinó si (valor == v2)
        acció2();
    sinó
        finalitzar_execució();
    fi si
fi si
```

Com es veu a l'exemple de codi, els mètodes s'executen sempre, però tenint en compte el valor de retorn per no interpretar erròniament els resultats. Fer això és ben diferent de la solució trivial de no executar els mètodes d'accés a recursos quan estem simulant l'execució. La raó que ens porta a executar sempre tots els mètodes, fent servir el llançador de mètodes, és que així podrem observar el comportament del programa, cosa que seria impossible si en cas de depuració evitéssim executar-ne alguns.

La solució que hem adoptat per poder saber en quin mode es troba el programa, ha estat afegir un atribut indicador a la classe *ResourceAccess* que s'inicialitza dins dels constructors de *ExecuteResourceAccess* i *DebugResourceAccess*. Aquest atribut és un booleà que es s'anomena *mode* i establirem el valor *true* per a l'execució real i false per la depuració.

El valor de retorn d'*executeMethod* pel mode real serà l'objecte que retorni el mètode. Per a la simulació, per defecte retornarem el valor *null*.

3.2.3 Implementació personalitzada dels mètodes en mode simulació

Hem comentat a la fase d'anàlisi que també és necessari que el programador pugui decidir obtenir un valor de retorn diferent a *null* per a una crida a un mètode en mode simulació.

Per aconseguir aquest fet cal que el programador pugui desenvolupar les seves pròpies implementacions dels mètodes que s'executen en mode simulació.

Una opció molt senzilla i primitiva seria que, dins mateix del codi del llançador de mètodes, hi hagués la implementació dels mètodes que es criden, i que per cada un

s'executés el que el programador hi posés. El programador hauria de modificar el codi proporcionat pel nostre sistema d'accés a recursos, de la següent manera:

```
public Object executeMethod(Object baseobject,
    String classname, String methodname, Object[] args){

    if ((methodname.compareTo("intValue") == 0) &&
        (classname.compareTo("Integer") == 0)) {
            return 3;
    }
}
```

Quan l'agent executés el mètode *intValue()* per un objecte *Integer*, rebria sempre un 3 enlloc de *null*

Aquesta solució no és idònia. Estem buscant una solució general, i en canvi, tal i com proposem, l'usuari ha de modificar el codi del nostre entorn per obtenir la funcionalitat que necessita.

Classes amb els mètodes personalitzats

Fent ús del llançador de mètodes generalitzat hem trobat una solució millor: Situar implementacions dels mètodes fetes pel programador en un altra classe i fer que *executeMethod*, en cas de simulació, vagi a buscar-les.

Hem creat un *package* que imita l'estructura de paquets de l'API de Java, és allà on el programador situa les classes que hagi implementat.

El *package debugmethods* amb algunes implementacions fetes pel programador tindrà la forma de la figura 3.4.

Per accedir-hi, hem modificat *executeMethod* pel cas de simulació. Canvia el nom de la classe que se li passa per paràmetre i hi afegeix un prefixe. D'aquesta manera *java.lang.String* es converteix en *debugmethods.java.lang.String*. Aquesta és la classe on es busquen les implementacions del programador. Si troba la classe aleshores fem una cerca del mètode. En cas de no trobar la classe, o de trobar la classe però no el mètode, retorna *null*.

Així no cal fer cap altre modificació a la resta d'elements del sistema d'accés a recursos, només fa falta que *executeMethod* capturi l'excepció *ClassNotFoundException* si la implementació del programador no existeix.

Pel cas dels constructors es fa el mateix, però fent només la cerca per constructors. Si troba el constructor que es busca, implementat pel programador, fa una nova instància, sinó retorna *null*.

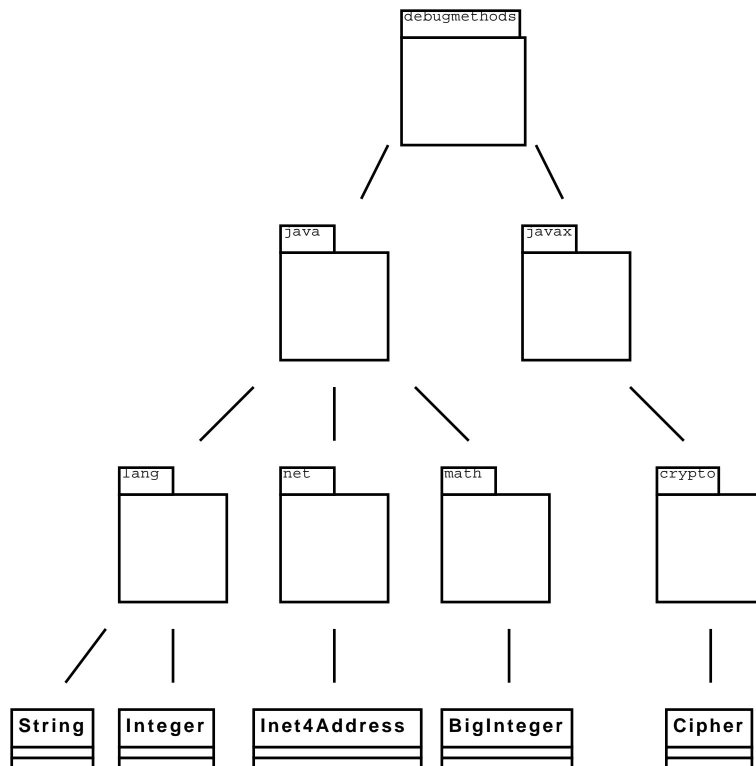

Figura 3.4: Classes implementades pel programador dins del package *debugmethods*

3.2.4 Tractament d'excepcions amb la reflexió a Java

Si el mètode d'accés a recursos que cridem llança una excepció, hem de ser capaços de controlar-la i enviar-la al programador que crida a *executeMethod* o *executeConstructor*.

El recurs que fem servir és la gestió d'excepcions que ens permet fer *InvocationTargetException*. Quan cridem un mètode o constructor amb *executeMethod* o *executeConstructor*, capturem aquesta excepció, que inclou quina ha estat l'excepció que ha llançat el mètode original. Des del codi de l'agent la tornem a capturar i llançar. D'aquesta manera fa arribar l'excepció al programador. El programador es capaç de trobar l'excepció retornada originalment utilitzant el mètode *getTargetException()* de la classe *InvocationTargetException*, i fer-la servir com si es tractés de l'excepció original. També pot trobar la causa d'aquesta excepció amb *getCause()* de la mateixa classe.

3.2.5 Estructura de les classes

La classe *ResourceAccess* haurà de tenir les abstraccions dels mètodes que implementin l'accés a recursos, *executeMethod* i *executeConstructor*, a més de l'atribut mode.

ResourceAccess s'ha implementat com una classe abstracta i no com una interfície. Al llenguatge Java no hi ha una gran diferència, però les interfícies tenen una propietat

que ens limita en aquest cas. L'atribut *mode* ha de ser propi de *ResourceAccess*, perquè és comú a les subclasses. Si la fem interfície aleshores aquest atribut es *final*, el que el converteix en constant i ja no ens deixa utilitzar-lo com ens cal.

Per tant, *ResourceAccess* serà una classe abstracta de la qual n'hereden *ExecuteResourceAccess* i *DebugResourceAccess*.

Al *package debugmethods*, que és on el programador situarà les implementacions de les seves classes de depuració, només s'hi accedeix des de *DebugResourceAccess*.

Així, el diagrama de classes d'aquest sistema d'accés a recursos és el de la figura 3.5.

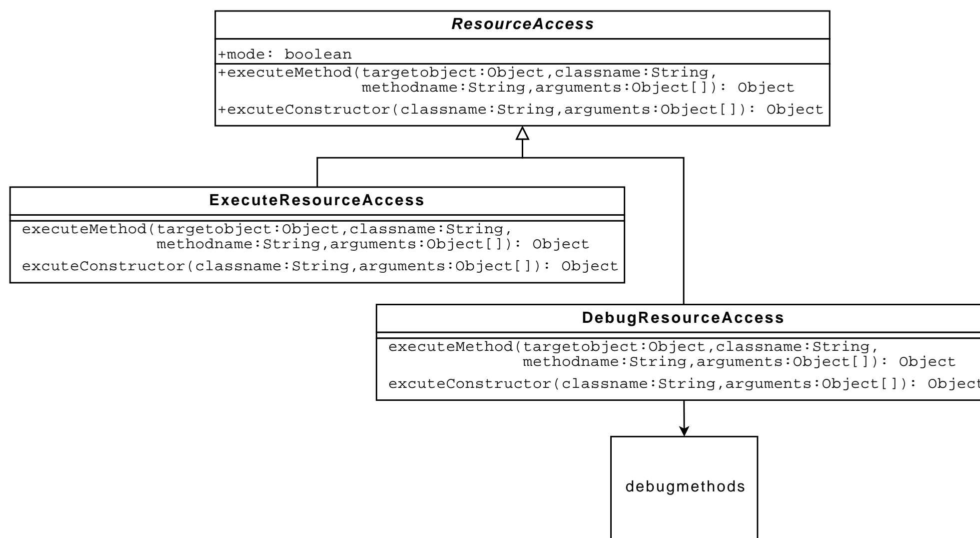

Figura 3.5: Disseny simplificat amb les classes de depuració. Versió definitiva

3.3 Implementació

Al capítol de disseny, hem argumentat que per poder fer servir tots els mètodes de Java en els dos modes d'execució, real i simulada, farem servir el mètode *executeMethod*. Aquest mètode fa la cerca i execució dels mètodes que se li indica, en temps d'execució. Per aquest motiu, per poder implementar *executeMethod* ens farà falta utilitzar la reflexió de Java.

L'*executeMethod* rep els paràmetres del mètode que ha d'executar com una llista d'elements *Object*. El *cast* que farem a aquests elements cap al tipus *Object* en produirà alguns problemes, que descriurem a la secció següent.

3.3.1 Limitacions de la reflexió a Java

El mètode de la classe *Class*, *getMethod(String name, Class parameterTypes)*, fa una cerca entre els mètodes d'una classe donada en temps d'execució, fins que troba una coincidència exacta. Per al nom del mètode no hi ha cap problema. Com es tracta d'un *String*, sempre trobarà una coincidència exacta amb un o més mètodes de la classe on es cerca. En canvi, pels tipus dels paràmetres del mètode, que són el segon element per discriminar quan busca, no podem fer la mateixa afirmació.

Al passar els arguments a *executeMethod* com una llista d'*Objects* fem un *wrapping* (embolcallament) de les instàncies de diverses classes que hi passem, i els posem dins d'*Objects*. Aquest és l'únic mecanisme que ens permet fer-lo funcionar amb qualsevol tipus de dades. Però això passa tant pels objectes que són instàncies d'una classe, com per els tipus de dades primitius. El problema que fa que no podem utilitzar *getMethod* és que per la seva implementació, no és capaç de fer un *unwrapping* dels tipus primitius que estan dins d'objectes. El seu funcionament es basa en buscar la signatura exacta del mètode. La signatura és el seu conjunt de característiques (nom, tipus de retorn, i número i tipus de paràmetres). Això no passa amb el mètode *invoke(Object obj, Object... args)*, que és capaç de buscar una coincidència més àmplia amb tipus compatibles. Aquest mètode actua com el compilador de Java.

Per saber quina és la classe real d'un objecte *Object*, només podem utilitzar el mètode *getClass()* de la classe *Object*, com veiem a la figura 3.6.

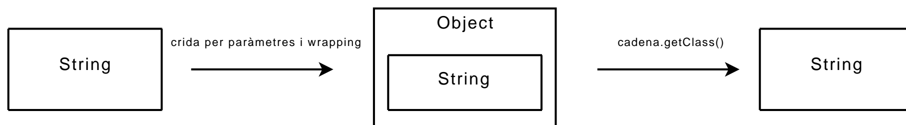

Figura 3.6: *Wrap* i *unwrap* d'un objecte en Java

En aquest cas, *cadena.getClass()* retorna el mateix que teníem abans de fer la crida a executeMethod, Un *String.class*.
En canvi per a tipus de dades primitius, al cridar *getClass* no obtenim el mateix. Per exemple, a la figura 3.7 al fer-ho amb un tipus primitiu *int*.

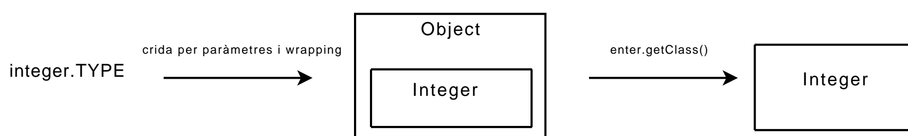

Figura 3.7: Wrap i unwrap d'un tipus primitiu en Java

En aquest cas teníem un *integer.TYPE* i hem obtingut un *Integer.class*.

Només comptem amb aquesta eina per saber el tipus de dades original d'un *Object*. A l'argument *parameterTypes* de *getMethod*, li haurem de posar sempre els elements que obtenim amb *getClass*.

Per tant, ens serà impossible trobar cap mètode que utilitzi algun paràmetre d'un dels vuit tipus primitius de Java: *integer*, *float*, *double*, *short*, *long*, *boolean*, *character* i *byte*. Evidentment això és una limitació no assumible i hem de buscar alternatives a aquest fet.

3.3.2 Implementació del cercador de mètodes

La primera opció, i a priori més senzilla, és passar per paràmetre a *executeMethod* la llista dels tipus dels arguments. Tot i la senzillesa de la solució cal dir que dóna una feina extra al programador que voldríem evitar. Cada vegada que crida un mètode d'accés a recursos ha de crear una llista d'elements *Class*, i passar-la per paràmetre a *executeMethod*.

Això té l'inconvenient d'haver d'introduir manualment els tipus primitius perquè *getClass* els detecta com a objectes. Per exemple, un tipus primitiu *int* el detectaria com un *Integer*. Deixar aquesta feina a mans del programador és donar només una part de la solució. Abans de cada crida a executeMethod hi hauria d'haver tot un seguit de codi, amb la construcció de tots els objectes *Class*, o una altra crida a un mètode auxiliar que se n'encarregués.

És possible implementar una eina que faci aquesta tasca d'una manera més ordenada. Descartem l'ús de *getMethod*, perquè les dificultats que dóna pel fet sempre que busqui un mètode amb la signatura exacta limita que pugui ser utilitzat per la finalitat que necessitem.

3.3.3 Alternatives a la cerca reflexiva de mètodes de Java

Per solucionar aquests fets, ha fet falta consultar experiències d'altres programadors en Java. La reflexió en aquest llenguatge és un tema que genera força controvèrsia i que té una dificultat associada, força elevada segons l'aplicació que es vol desenvolupar.A [18] s'explica justament les limitacions del paquet *java.lang.reflect* que s'ha comentat a la secció d'aquest projecte que les tracta. A més, però, proposa algunes solucions que fan que s'aconsegueixi una solució similar a la que buscàvem. Aquestes propostes prescindeixen directament de *getMethod* ja que és l'obstacle principal que es troba.

El que ens falta és conceptualment prou simple. Cal un cercador de mètodes, que a més de trobar els que tinguin els arguments exactament coincidents amb els tipus de la nostra llista d'objectes *wrapped*, miri si n'hi ha que tenen tipus primitius coincidents

amb ells. El mètode trobat es pot cridar posteriorment amb el mètode *invoke(Object obj, Object... args)*.

MethodFinder

A [18] es fa servir primer una classe que anomena *MethodFinder* i que compleix amb aquesta necessitat. Conté un mètode *findMethod(String methodName, Object[] params)* que implementa d'una manera senzilla una cerca àmplia, però simple segons els tipus de dades. Per cerca àmplia i simple entenem buscar per objectes idèntics o assignables, i que quan busquem tipus primitius als que s'ha fet un *wrapping* cap a objecte també els reconegui. Per a constructors utilitzarem la mateixa idea però anomenant el mètode *findConstructor*.

El primer pas per utilitzar *MethodFinder* és cridar al seu constructor, passant-li la classe amb la qual el volem en volem buscar un mètode. Aquest constructor de *MethodFinder* farà un índex dels mètodes de la classe i dels seus paràmetres. Aleshores farem la crida a *findMethod*. El seu funcionament és el següent:

```
findMethod(String methodName, Object[] params) retorna: Mètode
per (tots els mètodes amb nom == methodName) fer:
  m = mètode [i] dels que s'ha trobat
  si (número de paràmetres > 0)
    per (cada un dels paràmetres de m)
      p = paràmetre [i] de m
      si p és primitiu
        si p coincideix amb el tipus wrapped de params [i]
          correcte, seguim comprovant per la resta de paràmetres
        sinó
          avaluar un altre mètode de la llista
      sinó
        si el paràmetre és assignable
          correcte, seguim comprovant
        sinó
          avaluar un altre mètode
  si ha estat correcte o número de paràmetres == 0
    retorna el mètode
  sinó
    segueix comprovant
si no hi ha hagut cap coincidència
  llença excepció
```

Cal considerar certes limitacions que no es poden evitar amb aquest enfoc més simple:

1. Quan fem crides a mètodes no podem comptar amb compatibilitat entre tipus primitius. Per exemple, un mètode que accepti *double* també pot ser cridat amb *float*, però tal i com que el nostre *findMethod* farà la cerca, no serà capaç de trobar el mètode que li demanem. El més lògic, pensant en com actua el compilador de Java, és que n'hauríem de trobar diversos, el que té la signatura exacta i tots els que contenen tipus compatibles; però aquest no és el cas amb l'algorisme anterior. La solució és cridar sempre els mètodes amb els tipus de paràmetre per defecte.

2. Si un mètode rep un tipus primitiu com a paràmetre, i cridem aquest mètode passant-li un *null*, no podrà trobar el mètode. Quan es tracta d'un paràmetre on hi rep un objecte, en canvi, ho farà correctament. El problema és que el cercador de mètodes no és capaç d'identificar *null* com a tipus assignable a tipus primitius.

3. En alguns casos és impossible el fet d'utilitzar com a classe base un iterador de *List*, per exemple:

   ```
   List list = new ArrayList();
   MethodFinder f = new MethodFinder(
                         list.iterator().getClass());
   Method m = f.findMethod("hasNext", new Object[0]);
   ```

   En aquest cas l'iterador realment és una *inner anonymous class* la qual cosa fa que hi hagi un problema d'accés prohibit que genera una excepció. Aquest problema és fàcilment evitable fent un ús més pràctic de la classe *ArrayList*. Una alternativa possible per a l'exemple és construir un objecte *List*, i a partir d'aquest objecte operar-hi mitjançant *executeMethod*.

Com hem vist, per cadascun dels problemes hi ha una manera d'evitar-los. Així que podem afirmar que la classe *MethodFinder* ens dóna una solució general per al problema sempre i quan el programador tingui en compte aquestes limitacions superables. *MethodFinder* seria una opció viable per la nostra aplicació.

A [18] també es comenta quina seria la manera de no patir les restriccions 1 i 2, per tal d'aconseguir que el buscador de mètodes sigui capaç de trobar el que busquem, encara que li passem algun paràmetre compatible però no el que s'espera per defecte.

BetterMethodFinder

Al mateix article s'hi proposa una solució molt més completa que l'anterior.

El funcionament de *BetterMethodFinder* comença d'una manera semblant, però va molt més enllà. Un altre cop el primer pas és instanciar *BetterMethodFinder* amb la classe de la que volem buscar el mètode com a paràmetre. En aquest moment, com a *MethodFinder*, el constructor carrega tots el mètodes de la classe i uns índexs. A més, i aquesta és una diferència important, si la classe no és accessible, carrega tots els mètodes de la seva super classe, de les seves interfícies i les seves herències recursivament.

El mètode cercador de *BetterMethodFinder* és *findMethod(String methodName, Class[] parameterTypes)*. Enlloc d'una llista d'objectes se li passa, com a paràmetre, una dels tipus d'aquests. A partir de llavors, l'algorisme que fa servir tracta de buscar la signatura exacta. En cas de trobar-la la retorna directament. Si això no és possible, per cada un dels arguments intenta buscar-ne *Classes* compatibles. Si es tracta d'objectes, ho pot fer comprovant si són assignables, però si són primitius, ha de fer un *mapping* per saber quins tipus primitius són compatibles amb quins. Aquest element és clau, perquè fa aquest algorisme més general que el de *MethodFinder*. No només compara el tipus *wrapped* pel seu primitiu, sinó que d'entre els primitius escull els que permeten fer un *cast* automàtic.

Un cop troba una llista de mètodes compatibles, fa una cerca sobre ells per saber quin és el més específic i que millor s'ajusta a la crida que hem fet. Aquest càlcul es fa segons el número de *casts* necessaris. Si troba una ambigüitat llança una excepció.

Havent vist aquestes dues maneres d'implementar una solució, que parteixen d'una mateixa base però en que una de les dues és capaç d'estendre molt la funcionalitat, hem avaluat quina resulta més convenient per a la nostra aplicació.

Clarament, *BetterMethodFinder* ens dóna una solució més adient, perquè dóna més llibertat al programador. També hauríem de considerar quina té més impacte al rendiment de l'aplicació. A [19] i [20] hi trobem anàlisis d'aplicacions utilitzant la reflexió de Java. Tot i haver millorat des de versions anteriors de la màquina virtual, veiem que la pèrdua de rendiment respecte la invocació directe d'un mètode encara és important. La implementació de *BetterMethodFinder*, és més òptima que no *MethodFinder*, i ens ajudarà a minimitzar aquest impacte. Sembla que per ser més general hagi de ser més lenta, però *MethodFinder* fa la cerca sobre els mètodes de la classe passant per tots ells. *BetterMethodFinder*, en canvi, fa unes tries progressives sobre taules *hash*, que ajuden a estalviar temps. Havent-ne fet proves, *BetterMethodFinder* és de l'ordre d'un 20% més ràpid quan busquem i cridem 10000 mètodes. Aquesta diferència és molt important, i ens ajudarà a aconseguir un millor rendiment de l'aplicació.

Capítol 4

Extensió de l'IDE de disseny d'itineraris per a l'ús del simulador d'execució

Amb la possibilitat de depurar l'aplicació basada en un agent mòbil, fent servir el sistema d'accés a recursos, el programador és capaç de fer-ne proves en una plataforma local. El següent pas és donar al programador més facilitats de cara a dur a terme el seu desenvolupament i fase de proves.

El 2004, en Jordi Nebot va realitzar, durant el seu Projecte de Final de Carrera, una interfície gràfica de disseny d'itineraris d'agents mòbils. Aquesta es pot veure a la figura 4.1. Constava bàsicament d'un panell on hi podíem dibuixar l'itinerari i afegir les tasques associades a cada node. En aquest capítol integrarem un depurador a l'IDE de disseny, aprofitant les seves característiques per permetre que el programador pugui visualitzar l'itinerari que l'agent recorre quan en simulem l'execució.

Un altre objectiu serà actualitzar els tipus de node que pot tenir l'itinerari. Farà falta canviar les imatges associades als nodes i també la seva representació als arxius de projecte guardats per l'usuari.

Primer estudiarem la feina original d'en Jordi Nebot, per tal de poder comprendre-la i saber quina és la millor manera d'estendre-la, sobretot des d'un punt de vista del disseny de software. La principal font d'informació per dur a terme aquesta tasca serà la seva memòria del projecte [10]. Farem especial èmfasi en les parts que seran més útils de cara al desenvolupament del depurador.

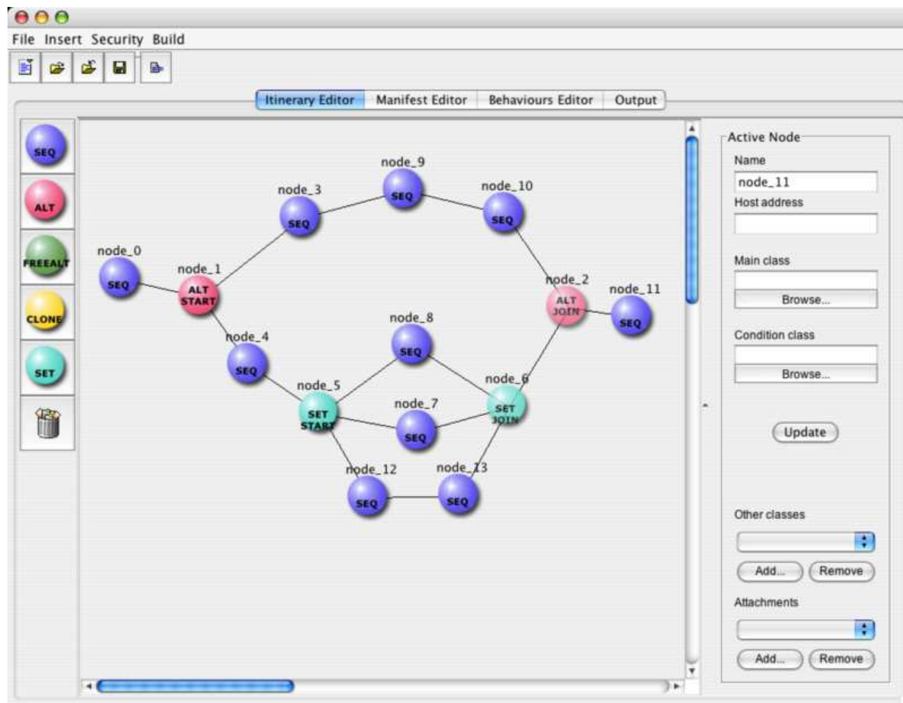

Figura 4.1: Interfície de disseny d'itineraris. Versió de 2004

4.1 Disseny de l'IDE original

La base del disseny de l'IDE és l'aplicació del patró de disseny model-representació-controlador. La seva finalitat és separar la organització de les dades que utilitzem (model) amb la seva representació gràfica (representació). La tercera part és l'element que fa d'interfície entre els dos, el controlador. A la figura 4.2 hi ha una representació d'aquest paradigma.

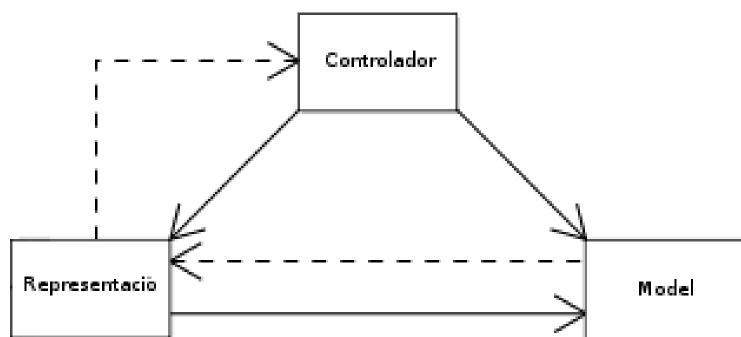

Figura 4.2: Patró Model-Representació-Controlador

Utilitzant-lo aconseguim que tant la lògica de representació com el model de dades que fem servir siguin més independents i, per tant, més reutilitzables. La part de representació té accés sobre el model, que és d'on treu les dades que mostra a l'usuari,

però és el controlador qui en gestiona els canvis i a més s'encarrega de la interacció amb l'usuari.

4.1.1 Model de dades i representació

A la figura 4.3 podem veure el diagrama de classes de la part del model de dades; que formen les classes *Node*, *Graph* i *GraphIO*. La classe *GraphPlotter* és la part de representació gràfica. Com es pot veure a la figura, la classe *GraphPlotter* accedeix a l'objecte *Graph* per tal de dibuixar l'itinerari a la interfície gràfica. La classe *Graph-Plotter* no forma part del model de dades, però l'hem inclòs al diagrama perquè té accés a l'objecte *Graph*.

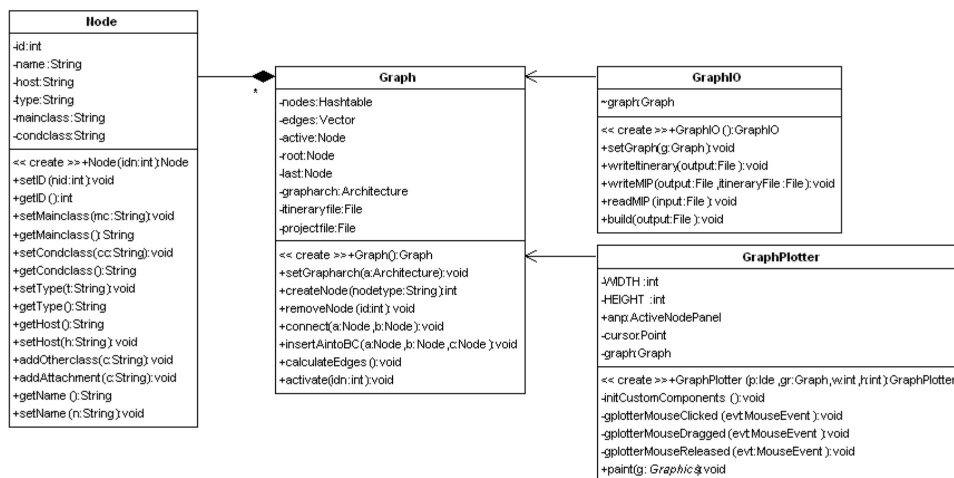

Figura 4.3: Estructura de la classe *Graph*

La classe *Node* és l'abstracció dels elements bàsics del model de dades. Contenen tota la informació d'un dels elements de l'itinerari i els mètodes necessaris per tal d'accedir-hi. *Graph* és la classe que representarà l'itinerari en forma de *Hashtable* del conjunt de nodes. Aquestes operacions les implementa a la seva llista de mètodes, que donen les eines necessàries al controlador per interaccionar-hi.

La lògica de representació està situada a *GraphPlotter*. Aquesta classe conté els mètodes de dibuix a partir d'un objecte *Graph*. Inclou el control dels events que fem sobre el dibuix i que tenen a veure amb la visualització dels elements, com ara el fet de moure un node amb el mouse. *Graphplotter* hereta de *JPanel*, una classe de Swing que representa un contenidor de dades lleuger. Normalment utilitzem aquest tipus de contenidor per situar-hi altres components: botons, àrees de text o gràfics. En aquest cas, però, es fa un panell personalitzat sobreescrivint-ne el mètode *paint(Graphics g)*. Amb aquest mètode implementarem el dibuix de l'itinerari representat a l'objecte *Graph*.

4.1.2 Controlador

Aquest rol el pren la classe *Ide*. És la classe principal de l'aplicació i s'encarrega de la gestió de tots els components gràfics de la interfície (excepte del panell *GraphPlotter*) i de capturar tots els events que genera l'usuari sobre ells. A partir d'aquesta classe es fan les crides als mètodes que implementen totes les accions que podem executar al programa: Des de guardar o tancar projectes, fins seleccionar l'arquitectura de protecció que utilitzarà l'agent del que n'estem dissenyant l'itinerari. Seguint la nomenclatura de Java Swing, *Ide* és un *JFrame* (contenidor tipus finestra principal) on s'integren altres components, i els components d'aquests, jeràrquicament.

Tant *Graph* com *GraphPlotter* són membres d'aquesta classe, que degut al seu paper de controlador actuarà entre elles dues i s'encarregarà de la seva interacció. A la figura 4.4 en veiem la realització en diagrama de classes.

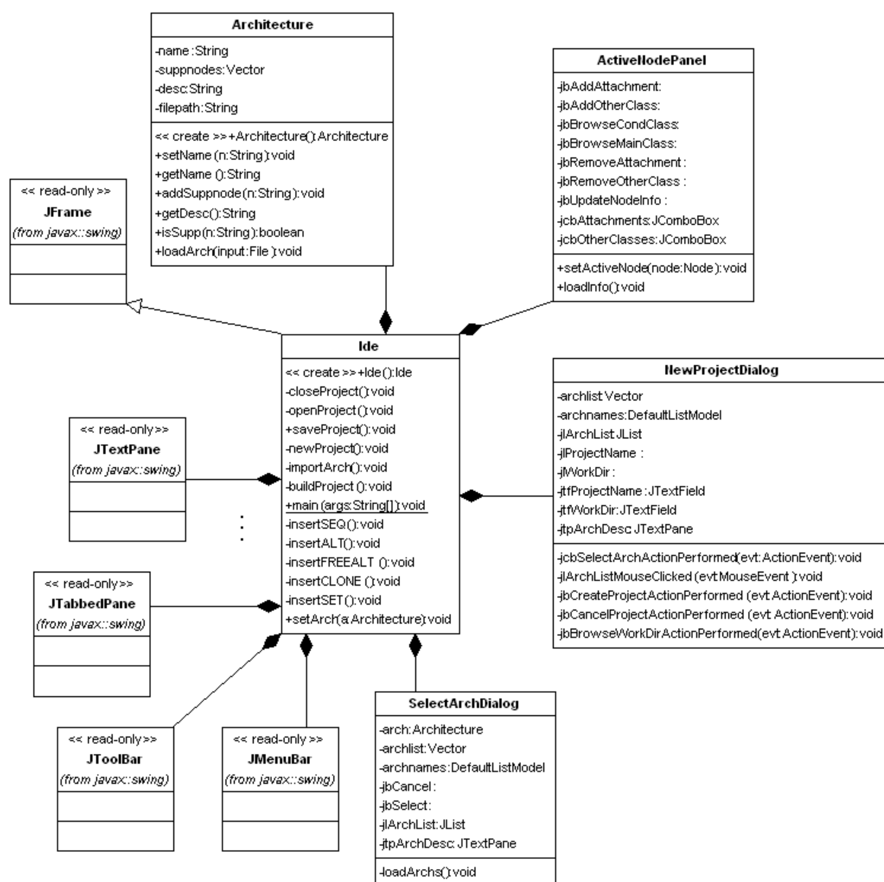

Figura 4.4: Esquema de la classe *Ide*

4.2 Anàlisi

Per estendre l'IDE original caldrà tenir en compte un nombre important de consideracions. A més de desenvolupar una solució nova, farà falta integrar-la i adaptar-se a la feina feta prèviament. Això afegeix un cert grau de dificultat al procés. Caldrà estudiar bé la solució i establir un anàlisi correcte per poder encarar la fase de disseny.

4.2.1 Requisits inicials

El depurador ha de permetre una execució completa d'una aplicació basada en agents mòbils que utilitzi el sistema d'accessos a recursos que hem desenvolupat. Només així podrem garantir que el programa que estem simulant no fallarà degut a fer una acció d'aquest tipus.

Hem de donar al programador informació suficient per comprovar si el seu agent funciona d'una manera correcta o incorrecta. Aquesta informació d'ajuda la donarem mitjançant una representació gràfica del recorregut de l'agent. Afegirem una sortida de dades on hi donarem resultats de la simulació. Per exemple: el temps que ha trigat a la seva execució, quins mètodes ha executat, etc. Les dades recollides sobre la simulació permetran fer *Profiling*, l'anàlisi del comportament i del rendiment de l'agent.

Un dels objectius d'aquest entorn de desenvolupament és que sigui fàcil de distribuir. Això farà que no puguem dependre de cap llibreria externa ni d'arxius de cap tipus que no peguem incloure en la versió final. Si no ho fem així posarem entrebancs al programador que vulgui utilitzar-lo.

4.2.2 Restriccions de disseny

La versió anterior estava feta en llenguatge Java i la seva llibreria gràfica Swing. Això s'haurà de tenir en compte a la fase de disseny i implementació, perquè la integració dels diversos components que desenvolupem dependran molt d'aquest fet.

Un altre punt important és que hem d'intentar reutilitzar la major part de disseny i codi de l'IDE en la seva versió de 2004. No és possible redissenyar tot el projecte anterior perquè no és dins l'abast d'aquest Projecte de Final de Carrera. Per això és molt important trobar una solució òptima de disseny que s'adapti al màxim a la feina ja feta.

4.2.3 Requisits de la interfície

El depurador gràfic s'ha de mostrar com una opció més de l'IDE. El seu funcionament ha de seguir la lògica dels components que ja hi estan inclosos. També cal que el seu

disseny sigui coherent amb la resta d'elements gràfics.

La informació extra que donem no cabrà en un dels panells de l'IDE. Haurem d'incloure algun sistema per poder visualitzar el total de les dades que donem al programador.

Hi haurà d'haver uns botons de control a la interfície per fer que l'agent comenci, es pausi, segueixi o es mori. D'aquesta manera, el programador podrà controlar-ne la simulació.

4.2.4 Requisits funcionals

El fet d'executar un agent mòbil basat en JADE fa que haguem de tenir una plataforma en funcionament paral·lelament amb l'IDE. Hauríem d'estudiar com reduir al mínim la complexitat a l'hora d'haver d'arrencar l'IDE i la plataforma, sense deixar un excés de feina al programador cada vegada que vulgui simular l'execució d'un agent.

Hem d'aconseguir que en el cas que un agent falli, la interfície es mantingui inalterada i mostri aquesta situació normalment, per tant, el depurador ha d'estar aïllat del funcionament de l'agent que estem provant.

L'execució simulada d'un agent s'ha de fer pas a pas, sinó podria anar massa ràpid. Si no en controlem la progressió, el programador podria perdre la possibilitat de veure el progrés de l'agent node a node per l'itinerari. L'usuari pitjarà el botó corresponent per indicar que l'agent continuï.

Seguint amb el punt anterior, durant la simulació farà falta que l'agent i el depurador es comuniquin. L'agent haurà d'informar dels seus resultats de simulació, i el depurador haurà de transmetre a l'agent les comandes que rebi a través dels botons de control.

4.3 Disseny

4.3.1 El panell del depurador

Per integrar el depurador a l'IDE i fer que en sigui un component més, utilitzarem tres contenidors lleugers (light containers) de Swing. Representaran els components que podrà visualitzar l'usuari: Una zona on es dibuixarà l'itinerari, una altra on hi posarem els botons de control de l'agent, i una altra on hi situarem un resum de les dades sobre la simulació de l'agent. Aquests tres elements els afegirem al contenidor amb pestanyes de l'IDE, on ara hi ha posades les diferents eines que componen la interfície gràfica i ens permeten canviar ràpidament entre elles.

Les parts dels botons i la informació seran contenidors amb elements simples, com ara botons i etiquetes de text. En canvi, la zona on es dibuixarà l'itinerari serà més complexa. Serà semblant a la classe *GraphPlotter*. Per tant, també utilitzarem un panell que hereti de *JPanel* i que implementi un mètode de dibuix personalitzat. De manera similar, també utilitzarà l'itinerari definit al panell de disseny per fer el dibuix. Aquesta classe s'anomenarà *Debugger*. Als següents subapartats es veurà que també integrarà les funcionalitats de mantenir la informació de la simulació.

4.3.2 Informació de la simulació

Utilitzarem una estructura organitzada per guardar la informació de la simulació, com es veu a la figura 4.5. L'element principal és *Profiler*, que contindrà un conjunt de *MigrationElements*. A cada un d'aquests hi ha la informació que s'ha generat per la simulació en un node. La informació inclosa en un *MigrationElement* és: El temps d'execució i quins han estat els mètodes d'accessos a recursos que s'han cridat, representats per objectes *ResourceAccessElement*.

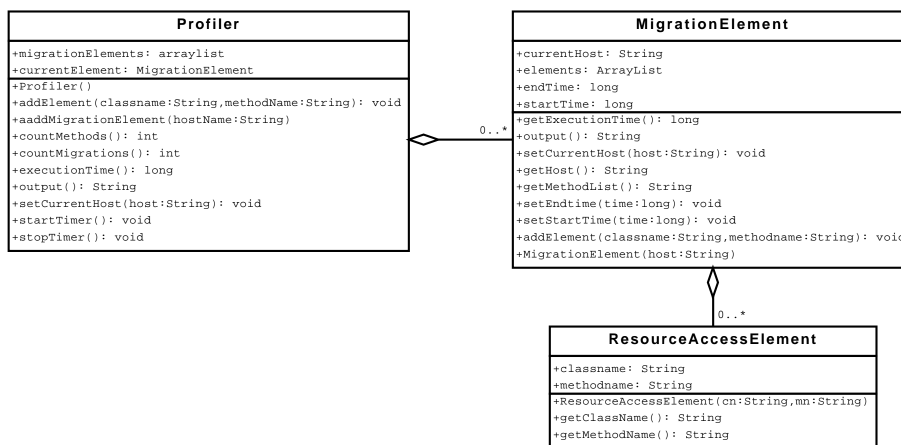

Figura 4.5: Diagrama del sistema per desar informació

L'agent mòbil conté un objecte *Profiler* que serà enviat després de cada migració a *Debugger* a cada migració simulada. Així podrem anar-ne seguint el progrés per cada pas. Per cada migració mostrarem a l'IDE un resum dels resultats de la simulació, i al final donarem la opció de veure tota la informació que s'ha recollit en una finestra apart.

Aquest esquema és extensible, seria fàcil afegir-hi més característiques per recollir altres dades.

4.3.3 La comunicació entre l'agent i l'IDE

Un cop hem descrit els elements necessaris per la visualització i interacció del programador, el següent pas és establir com es rebran els resultats de la simulació de l'agent mòbil i com l'IDE li transmetrà les ordres necessàries per controlar la seva execució.

Inicialment executarem, per una banda, un agent que es troba dins d'una plataforma JADE i, per l'altra, la interfície gràfica amb el depurador. La comunicació es farà amb pas de missatges. En aquest cas farem servir *sockets* de tipus UDP perquè ens trobarem sempre en una màquina local. Com que aquesta comunicació serà bidireccional, hem de lligar els *sockets* a adreces per poder rebre els missatges.

A la interfície gràfica no hi podem posar un *socket* en espera bloquejant, perquè si es queda aturada no podem fer-hi cap acció. Per evitar-ho implementarem una classe, que anomenarem *DebuggerComm*. Funcionarà paral·lelament a *Debugger* i li entregarà les dades de l'agent quan les rebi pel seu *socket*. El funcionament concurrent el tindrà estenent *Thread* de Java. Per passar les dades que rebi a *Debugger*, *DebuggerComm* farà servir els mètodes *setHost(String hostName, byte[] profileData)* i *agentEnd(byte[] ProfileData)* de *Debugger*. Aquests serviran, respectivament, per posar-hi les dades en cas de migració i de fi de la simulació de l'agent.

Com hem vist, la comunicació entre l'agent i la interfície ha de ser bidireccional. Això vol dir que l'agent també hauria de tenir un *socket* en espera per rebre les comandes que li doni l'usuari a través de la interfície. L'agent, per tant, també es quedaria bloquejat. Aquest fet ja va bé, perquè així l'agent es queda parat fins que l'usuari indiqui el contrari, permetent un control sobre la simulació.

L'inconvenient de tenir enviaments entre les dues parts és que farà difícil controlar la sincronització. Si el funcionament fos fixat i seqüencial no hi hauria d'haver cap problema, però amb la intervenció de l'usuari fàcilment ens podem trobar amb les dues parts bloquejades esperant un missatge, en un cas clar d'espera mútua (*deadlock*).

Utilitzar un *socket* per enviar dades de l'agent a la interfície és la millor solució, perquè l'agent haurà d'enviar les dades de la simulació i informació sobre les migracions. Aquestes dades les serialitzarà l'agent i les enviarà a *Debugger* que les deserialitzarà per tractar-les. En canvi, per implementar els controls de l'agent podem considerar una altra manera de fer-ho. A l'API de JADE hi trobem la classe *jade.core.Runtime*, que ens permet executar una plataforma JADE completa dins de qualsevol programa Java, i el *package jade.wrapper*, amb el que podem arrencar i controlar agents mòbils i contenidors d'agents dins d'aquesta plataforma. Hem fet servir aquestes classes a *Debugger* per controlar l'agent.

Els missatges de comunicació entre l'agent i DebuggerComm

Quan l'agent es comuniqui amb la interfície, utilitzarà dos tipus de missatges. Aquests es correspondran a les accions que pot fer: La migració a diferents plataformes i la seva fi. L'inici no cal que el notifiquem a la interfície, perquè és justament aquesta qui arrenca l'agent. En canvi, si que ha d'informar de les migracions i de quan finalitza la seva execució.

- Quan l'agent migri tindrem un missatge del tipus: `currenthost:X/hostname[dades serialitzades]`; on X és la longitud del nom del host. Indiquem aquest factor perquè just darrera hi haurà les dades serialitzades, i necessitem saber el byte exacte on començaran.

- Quan l'agent finalitzi rebrem: `agentfinished[dades serialitzades]`.

La interacció entre l'agent i la interfície la podem veure representada a l'esquema de la figura 4.6, on hi veiem les classes que hi intervindran, i al diagrama d'interacció de la figura 4.7.

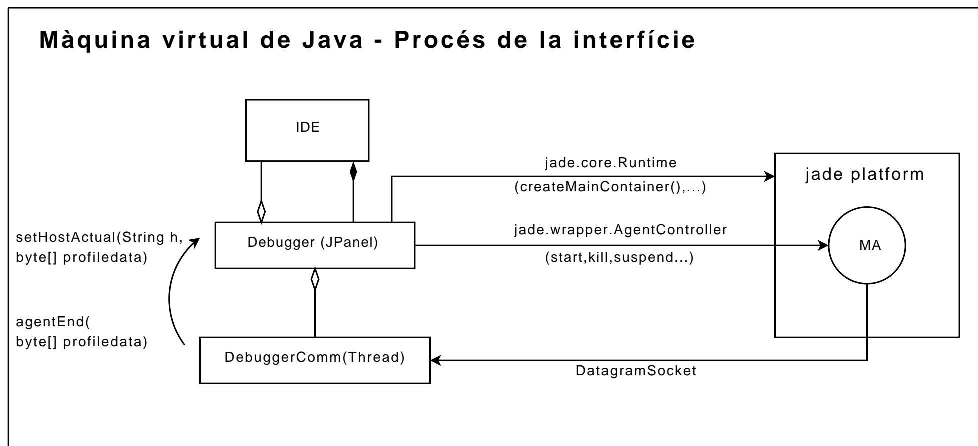

Figura 4.6: Esquema de les classes que intervenen a les comunicacions

4.3.4 Modificacions a la plataforma JADE del depurador

La comunicació entre l'agent i l'IDE, i el fet d'incloure la informació de la simulació, requereixen fer canvis a la plataforma JADE on llançarem l'agent. Aquest fet serà important, perquè caldrà que distribuïm la plataforma modificada com a part de l'entorn.

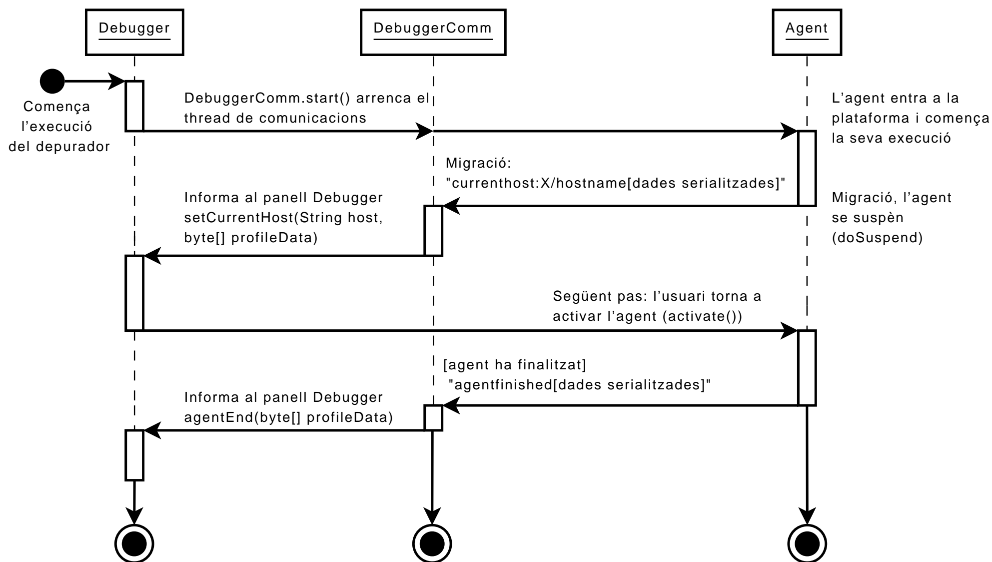

Figura 4.7: Diagrama d'interacció agent-depurador

Canvis deguts a la informació de la simulació

Quan programem un agent mòbil basat en JADE heretem de la classe *Agent* de *jade.core*, que conté les operacions i atributs bàsics que ha de tenir. En aquesta classe hi afegirem com a atribut la informació de *profiling*: Un objecte de la classe *Profiler*. Així, el programador no s'haurà de preocupar mai de la seva gestió, perquè això és feina de l'entorn. Hi implementarem també un mètode *setProfileData(String className, String methodName)* que servirà justament per posar les dades de simulació a l'agent. Aquest mètode és el que es cridarà des d'*executeMethod* de *DebugResourceAccess* quan estem fent la simulació. Fent això, *executeMethod* afegirà la informació de la simulació a l'objecte *Profiler*.

Hi ha un problema important si l'agent es llança en una altra plataforma que no tingui la classe *Agent* modificada. Aquesta no implementarà el mètode *setProfileData*. Si el programador hagués triat *DebugResourceAccess*, tot i ser conscient que s'està utilitzant una altra plataforma, al cridar *executeMethod* ens trobaríem amb una excepció, perquè *setProfileData* no estarà implementat a la seva classe *Agent*. Per solucionar-ho, crearem una interfície, *DebuggableAgent*, que només definirà un mètode: *setProfileData*. La classe *Agent* de la plataforma JADE modificada la implementarà. A la classe *executeMethod* de *DebugResourceAccess* farem una comprovació de si l'agent que en aquell moment s'està simulant és un *DebuggableAgent*. Si ho és, es podrà cridar al mètode *setProfileData* d'una manera segura. Si no, *executeMethod* no farà la crida a aquest mètode.

Canvis deguts a les comunicacions

Els canvis per poder implementar les comunicacions es centren als mètodes *doMove(Location destination)* i *doDelete()*, de la classe *Agent*.

El mètode de migració *doMove* el modificarem completament. Ara, enlloc de fer la seva funció de migrar l'agent, enviarà el missatge de migració amb les dades de la simulació a la interfície, i crearà un *MigrationElement* al seu *Profiler*. Amb aquesta acció iniciarà un nou interval de recollida de dades i de mesura de temps. També farem que aquest mètode suspengui l'agent cada cop que el cridem. Tal i com hem explicat prèviament, és la manera de fer que l'usuari pugui executar l'agent pas a pas.

Per *doDelete* en mantindrem la funcionalitat, canviar l'estat de l'agent per indicar a la plataforma que ja ha acabat el seu itinerari, i li afegirem el fet que enviï el missatge d'agent finalitzat amb les dades de simulació finals.

El diagrama de classes d'aquesta part de l'aplicació, amb tots els nous components integrats, és el de la figura 4.8.

4.4 Implementació

4.4.1 Actualització dels tipus de node de la interfície

Un dels objectius plantejats era actualitzar els tipus de nodes que es poden utilitzar amb la interfície. El procés per canviar-ho ha estat modificar la manera en la que es genera el fitxer XML, que representarà l'itinerari desat. Els canvis han estat fets a la classe *GraphIO*, on hi ha els mètodes dedicats a escriure i llegir aquest fitxer.

També s'ha canviat les icones que representen els nodes a la interfície gràfica. A la figura 4.9 veiem l'IDE amb les noves icones.

4.4.2 El panell del depurador

La classe *Debugger* s'encarregarà de les funcions de dibuix de la representació gràfica de la simulació, d'iniciar el *thread* de comunicacions i d'executar la plataforma JADE i llançar-hi l'agent.

Per implementar la funció de dibuix, hem sobreescrit el mètode *paint(Graphics g)*. Aquest mètode és heretat de *JPanel* i de les seves superclasses de Swing. La classe

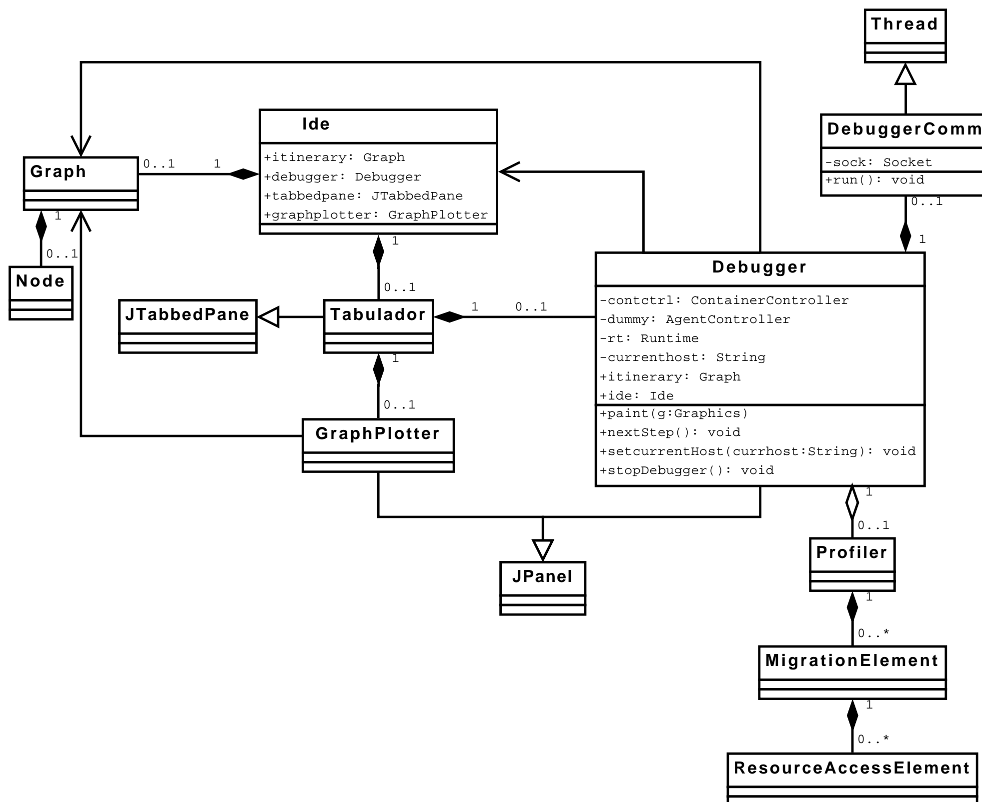

Figura 4.8: Diagrama de classes del depurador

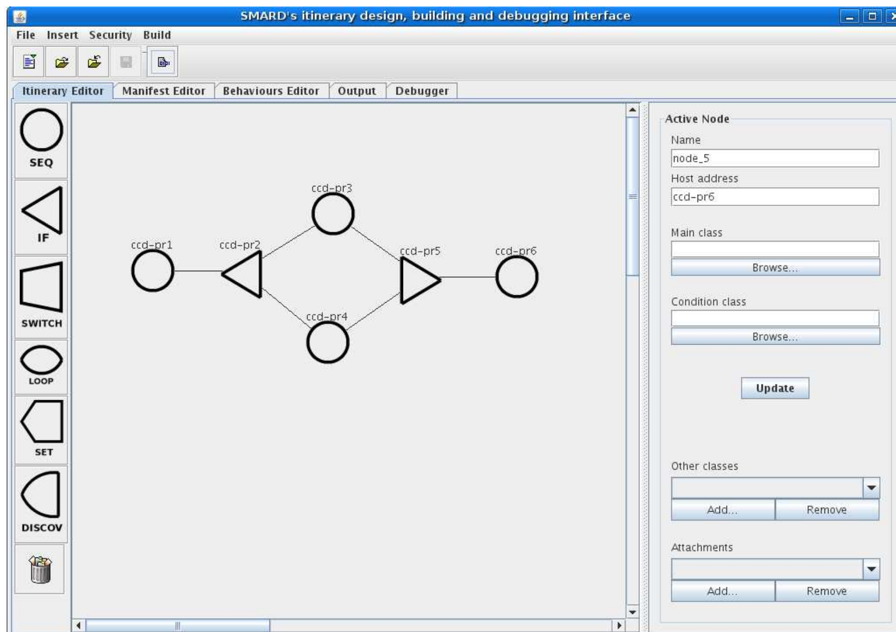

Figura 4.9: Comparació de les dues versions de l'IDE

Graphplotter del disseny anterior de l'IDE ja tenia aquesta característica. El que fem és utilitzar el mateix mètode, però incorporant-li una acció més: Comparar si el nom del node on es troba l'agent en la simulació coincideix amb algun dels noms dels nodes

que s'estan dibuixant. En cas que coincideixi mostrarà una icona diferent. A la figura 4.10 en veiem un exemple, en el qual el node actual és el corresponent a la plataforma ccd-pr3.

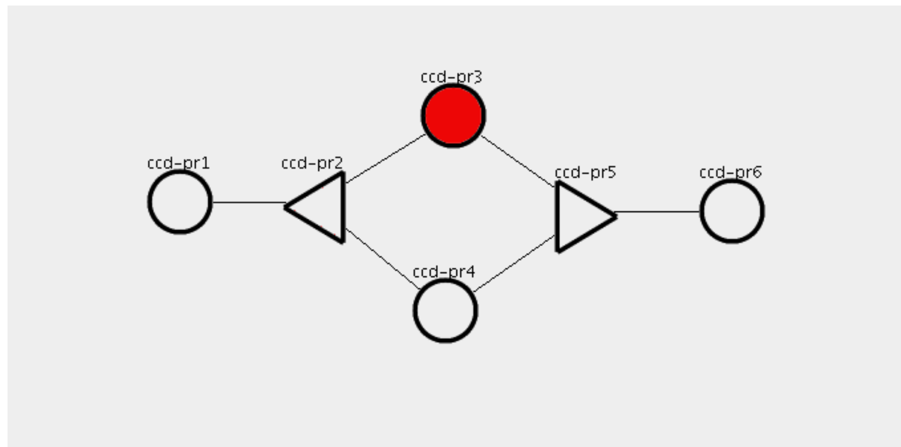

Figura 4.10: Gràfica de la simulació d'un agent

Les accions d'executar la plataforma JADE, llançar-hi l'agent i iniciar el *thread* que rebrà les dades de l'agent, les hem implementat al constructor de *Debugger*. Quan l'usuari decideix tancar el depurador fa servir el mètode *stopDebugger()*, que atura el *thread* i tanca la plataforma JADE. Tancant la plataforma també s'elimina l'agent mòbil.

4.4.3 Informació de la simulació

A l'apartat de disseny hem explicat l'estructura de classes que tindrà aquesta part de l'aplicació. El *Profiler* conté un *ArrayList* de *MigrationElements*, que són les unitats d'informació que inclouen les dades de la simulació produïdes en un node de l'itinerari. Els *MigrationElements* contenen un *ArrayList* de *ResourceAccessElements* i una mesura del temps que s'està calculant a cada pas de l'itinerari. Per mesurar aquest temps utilitzarem el mètode *System.nanotime()*. Aquest és el mètode de Java que ens dóna més precisió, així evitem donar errors d'arrodoniment.

Debugger i *Agent* tenen una única instància cadascú de *Profiler*. *Agent* l'envia a *Debugger* a cada migració i també quan finalitza.

4.4.4 El *thread* de comunicacions del depurador

La classe *DebuggerComm* s'encarrega de la recepció de les dades que l'agent mòbil enviarà cap a la interfície, perquè la instància de *Debugger* no pot estar bloquejada pel *socket* que s'ha de fer servir. La seva tasca és identificar el missatge que rep, i segons

el tipus d'aquest, informar de l'estat de l'agent i donar la informació de la simulació a *Debugger*.

La recepció de les dades i el tractament dels missatges que porten és senzill d'implementar, però aturar el *thread* és més complicat. És necessari parar-lo perquè el *socket* té una adreça lligada que és on l'agent envia les dades, i si no el tanquem aturant el *thread* no en podrem obrir un altre amb la mateixa adreça. A més, hem de pensar en la intervenció de l'usuari, que en qualsevol moment pot decidir que l'execució del depurador s'aturi.

Aturar el *thread* de comunicacions

La classe *Thread* de Java tenia uns mètodes adients per poder parar la instància en qualsevol moment. Aquests mètodes eren *stop()*, *stop(Throwable obj)* o *destroy()*. A partir de la versió de Java SE 5, però, aquests mètodes estan obsolets. La causa és la consistència d'algunes referències que feia la instància de *Thread* que s'ha destruït, que causa un comportament imprevisible de l'aplicació a partir d'aquell moment. A [21] podem trobar una explicació més detallada d'aquest fet.

A [22], trobem que hi ha una solució possible a aquest problema. Es basa en interrompre el *thread* que volem aturar (mitjançant *interrupt()* de la classe *Thread*) i capturar aquesta interrupció al seu mètode principal *run()*. En cas que el *thread* hagi estat interromput, tancarem el *socket* i llançarem una *InterruptedException*. Aquesta excepció ens servirà per parar el *thread*.

Hi ha un altre problema específic d'aquesta aplicació. El *thread* pot estar bloquejat esperant rebre dades del *socket* després que l'usuari hagi aturat el depurador, amb l'agent inclòs, si ha pitjat el botó corresponent. Si ens trobem en aquest cas, és impossible capturar la interrupció perquè no es rebrà cap altre missatge de l'agent; aleshores l'execució del *thread* es quedarà bloquejada. Per solucionar-ho enviem un tipus nou de missatge a *DebuggerComm*: `stopdebugger`. Quan es rebi, desbloquejarà el *thread*, se saltarà les passes que fa pels altres missatges i anirà directament a la part on es comprova si s'ha rebut una interrupció. Aquest serà el procediment que seguirem sempre per aturar el depurador. Així podem assegurar que el *thread* també s'ha tancat.

4.4.5 Modificacions a la plataforma JADE

Hem modificat la plataforma JADE que s'executa a l'IDE per tal que implementi les funcionalitats que necessitem. Els canvis els hem realitzat als mètodes *doMove(Location destination)* i *doDelete()* de la classe *Agent*.

Modificacions a *doMove(Location destination)*

En el mètode *doMove* hem eliminat el seu codi original, que inicia el procés de migració de l'agent. Ara s'encarrega d'aturar el temporitzador del seu objecte *Profiler* (perquè s'ha acabat un dels intervals entre migracions), de serialitzar aquest objecte, enviar-lo a l'IDE, i afegir un nou *MigrationElement* a la informació de la simulació.

Un cop ha fet aquestes accions l'agent es suspendrà a si mateix amb el mètode *doSuspend()*.

Modificacions a *doDelete()*

El mètode *doDelete()* conservarà la seva funció inicial de fer finalitzar l'agent canviant-lo d'estat. No obstant, li afegirem el fet d'aturar el temporitzador de l'objecte *Profiler* i serialitzar aquesta instància, per fer com a *doMove* i enviar-la a l'IDE.

Capítol 5

Integració d'un constructor d'agents mòbils a la interfície de disseny

En aquest capítol descriurem, en primer lloc, el constructor d'agents mòbils que farà que donem un entorn de desenvolupament més complet. Un cop integrat, el programador podrà dissenyar un itinerari, construir l'agent que el faci servir i depurar-lo.

A continuació, exposarem els canvis realitzats en aquest constructor per adaptar-lo a l'IDE de disseny i depuració d'agents mòbils.

5.1 Descripció del constructor d'agents mòbils amb itinerari explícit

El constructor serveix per generar agents mòbils auto-protegits a partir d'un fitxer XML, que representa l'itinerari, i diversos fitxers *jar*, que implementen les tasques locals. La versió actual d'aquest constructor d'agents, però, encara no genera agents protegits.

El constructor funciona en tres etapes. El primer consisteix en llegir la informació del fitxer XML i situar-la en objectes *InitialItinNode*, que seran continguts en un *InitialItinerary*. Aquesta acció l'implementa el mètode *readInitialItin(File itineraryFile)* de la classe *ItineraryBuilder*.

Les dues següents etapes serien xifrar les tasques d'aquest itinerari inicial que s'ha llegit, i escriure'l a un fitxer. Però com hem dit abans, la part de protecció no està encara implementada.

Els elements de l'itinerari inicial es guarden en objectes *SimpProtClearNode*, que a més de la informació que ja es troba als *InitialItinNode*, contenen els fitxers *jar* de les tasques locals serialitzats. Un cop es té un *HashMap* amb el conjunt d'objectes

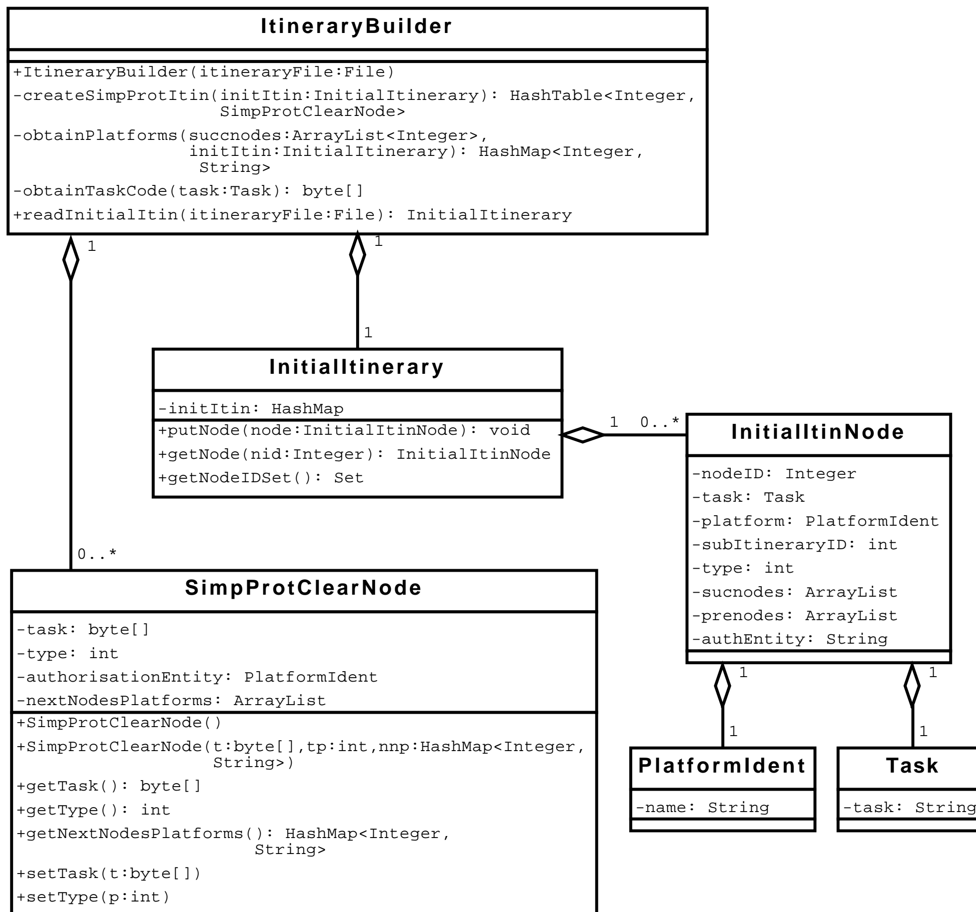

Figura 5.1: Diagrama de classes del constructor d'itineraris

SimpProtClearNode, el constructor serialitzarà tota aquesta instància en un fitxer.

A la figura 5.1 podem veure un diagrama amb les classes que s'utilitzen en aquest procés.

Codi de control

Una vegada s'ha creat l'itinerari explícit de l'agent, el codi de control és qui s'encarregarà de gestionar-lo. El codi de control que utilitzarem serà una classe, *MobileAgent*, que té dos *behaviours* propis: El *ControlBehaviour*, que dirigeix l'extracció de les tasques locals; i el *MigrationBehaviour*, que implementa el control de la migració de l'agent. Aquests dos *behaviours* són *inner classes* de *MobileAgent*.

Al principi de l'execució, *ControlBehaviour* llegeix el fitxer itinerari i carrega un *HashMap* d'objectes *SimpProtClearNode* a l'agent. Per cada plataforma, s'extreu la tasca local corresponent i s'afegeix al codi de l'agent com un *behaviour*. El mètode utilitzat per extreure cada tasca és *getTask* de *SimpProtClearNode*. Al següent pas, *ControlBehaviour* instància un *MigrationBehaviour*, que farà la migració quan s'hagi

acabat l'execució per aquella plataforma. Quan no quedin més nodes per visitar, el codi de control finalitzarà l'agent.

5.2 Anàlisi

5.2.1 Requisits inicials

S'ha d'integrar el constructor d'agents a l'IDE, de manera que utilitzi les estructures de dades de la interfície gràfica existent per construir l'itinerari.

El depurador ha de poder utilitzar l'agent i l'itinerari resultants per fer les proves. Serà necessari que la classe principal de l'agent, on hi ha el codi de control, tingui el format necessari per executar-lo en una plataforma JADE.

5.2.2 Requisits de la interfície

L'usuari ha de poder cridar el constructor d'una manera senzilla. A partir de la informació que haurà introduït a l'itinerari, i amb les tasques locals programades, s'ha de construir l'agent executable. L'objectiu és que només pitjant un botó de la interfície ja es generi l'agent final.

5.2.3 Requisits funcionals

El codi de les tasques locals que hi hagi a l'itinerari ha de poder utilitzar el sistema d'accés a recursos del capítol 3. Si el programador decideix que el seu agent ha de poder funcionar en mode simulació, aquelles tasques que es vulgui depurar hauran d'estar implementades tenint això en compte. Si hi ha tasques que no han de fer servir el sistema, el constructor també les ha de poder afegir a l'itinerari sense problemes.

5.3 Disseny

5.3.1 Sistema d'accés a recursos per agents amb itinerari explícit

L'agent que construïm ha de poder utilitzar el sistema d'accés a recursos que hem implementat al capítol 3. Quan es programa un agent mòbil fent servir aquest sistema, es crea un objecte *ResourceAccess* que s'utilitza durant l'execució real o la simulació per fer els accessos a recursos.

Quan s'inicialitza *ResourceAccess*, es tria el mode d'execució que tindrà l'agent. Per tant, el codi de control de l'agent amb l'itinerari explícit haurà de passar aquest objecte a les seves tasques locals, que són les que executaran els accessos a recursos, per tal que totes tinguin una referència a l'objecte *ResourceAccess*. Així compartiran el mode d'execució.

Crearem la interfície *ResourceAccessBehaviour*. Aquesta interfície tindrà els mètodes *setMyAgent(Agent a)* i *setRA(ResourceAccess RA)*, que s'implementaran a les tasques locals que utilitzin el sistema d'accés a recursos. Aquests dos mètodes els cridarà el codi de control cada cop que obtingui una tasca de l'itinerari, per passar-li una referència al propi agent i una a la instància de l'objecte *ResourceAccess*.

Quan una tasca local no implementi *ResourceAccessBehaviour*, el codi de control no cridarà aquests mètodes. De ser així, com que no estaran implementats a la tasca, al cridar-los es llançaria una excepció.

5.3.2 Lectura de l'itinerari

El constructor que volem adaptar llegeix l'itinerari d'un fitxer XML diferent del que s'utilitza internament a l'IDE. Per a la nostra aplicació hauríem d'adaptar el parser XML del constructor per tal que llegís els fitxers que genera la interfície de disseny d'itineraris, el que suposaria un desenvolupament addicional.

En canvi, el que hem fet ha estat integrar-lo de manera que utilitzi l'estructura de dades *Graph* de la interfície gràfica de disseny. Aquesta té totes les dades necessàries per construir l'estructura *InitialItinerary* del constructor.

El mètode d'*ItineraryBuilder* (la classe principal del constructor d'itineraris) que s'encarrega de fer la lectura de l'itinerari és *readInitialItin*. Modificarem aquest mètode per fer que llegeixi un objecte *Graph* que rebrà des de l'IDE.

5.3.3 Integració del constructor dins de l'IDE

Crearem la classe *Builder*, que servirà per gestionar el procés de construcció de l'agent, tant l'itinerari com el codi de control.

Builder serà membre de la classe principal de la interfície, *Ide*. Quan es construeixi l'objecte *Builder*, se li passarà l'objecte *Graph* i el directori on s'ha executat l'aplicació. Aquest segon paràmetre és necessari per quan haguem de guardar l'itinerari a disc.

Quan es pitgi el botó corresponent a la interfície, *Ide* capturarà l'event i cridarà el mètode *doBuild()* de *Builder*. Aquest mètode farà les dues accions del nostre constructor: Primer instanciarà *ItineraryBuilder*, creant el fitxer itinerari. En segon lloc,

agruparà les classes del codi de control en un fitxer *jar*.

El diagrama de classes d'aquesta part de l'aplicació és el de la figura 5.2. Al diagrama només hi hem posat els detalls de les classes que s'han modificat.

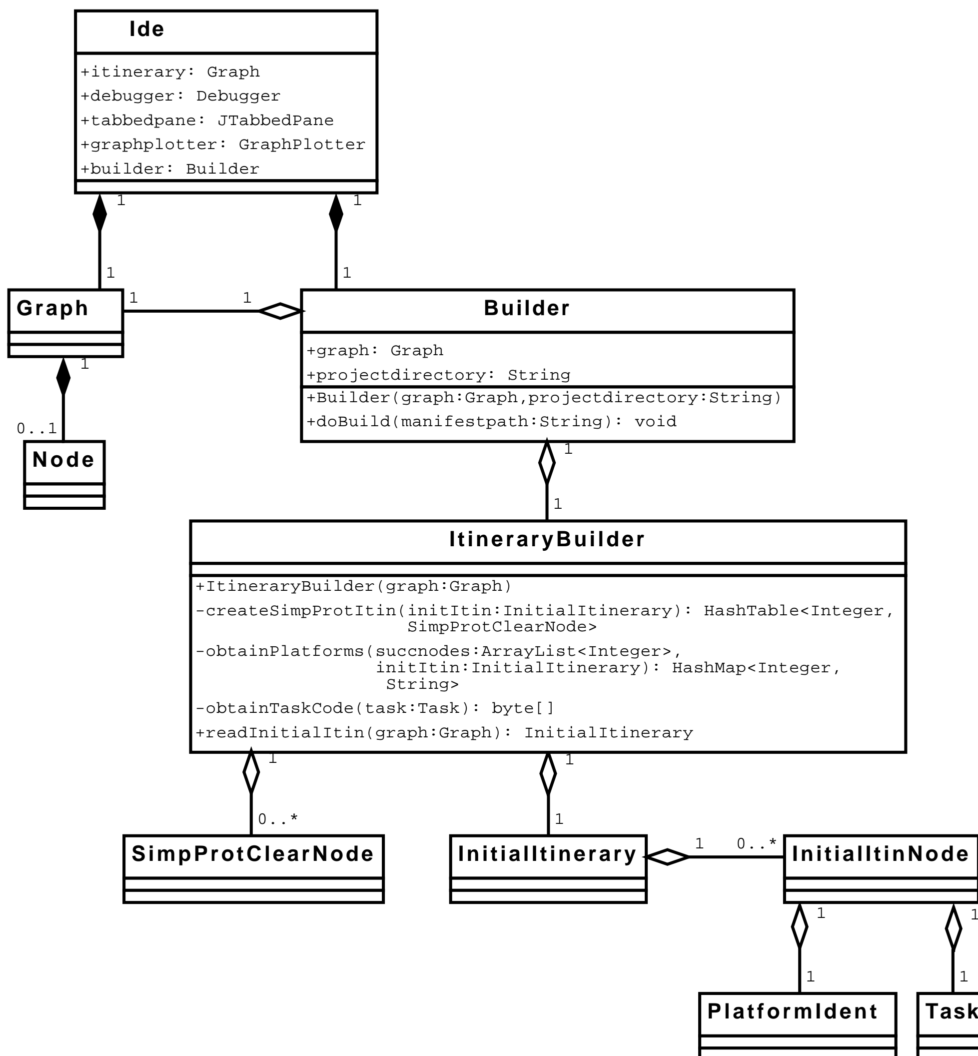

Figura 5.2: Diagrama de classes de la integració del constructor d'agents

5.4 Implementació

5.4.1 La classe Builder

La classe *Builder* utilitzarà *ItineraryBuilder* per construir l'itinerari explícit en un fitxer apart, amb el procés que hem descrit prèviament.

El fitxer *jar* amb el codi de control i les classes que aquest fa servir, es crearà cridant el mètode *exec* de la classe *JavaShell*. Amb aquest mètode executarem la comanda

jar del sistema. La classe *JavaShell* va ser desenvolupada per en Jordi Nebot al seu projecte, i ja es feia servir en una altra part de l'IDE.

Capítol 6

Proves

Hem decidit dividir les proves en dos apartats. Per una banda, proves sobre la interfície gràfica i els elements amb els que pot interaccionar el programador. Per l'altra, provarem el sistema d'accessos a recursos.

6.1 Proves de funcionament de la interfície gràfica del depurador

6.1.1 Preparació de les proves

Necessitem tenir alguns elements preparats per poder començar a fer proves. Hem programat algunes tasques locals que es podran executar a les plataformes simulades. Agafarem els fitxers *.class* on estan implementades i els empaquetarem dins d'arxius *jar*, amb el seu *manifest* corresponent. Algunes d'aquestes tasques faran servir el sistema d'accés a recursos del capítol 3.

També hem dissenyat l'itinerari d'un agent mòbil, utilitzant la part de l'aplicació que va implementar en Jordi Nebot. No provarem extensivament aquesta part, perquè aquesta feina està feta a [10]. Amb el panell d'edició dels nodes, els inclourem les tasques associades. L'itinerari que provarem és el de la figura 6.1.

Per poder provar el depurador sencer, intentarem fer el mateix procés que hauria de seguir un programador que tingués el seu agent dissenyat.

6.1.2 Construir l'agent

La construcció de l'agent és el primer pas que farà el programador, un cop tingui el projecte obert. Per construir un agent, només cal prémer el botó corresponent del menú. Això farà que apareguin dos fitxers al directori *projects*. Un porta el codi de control de

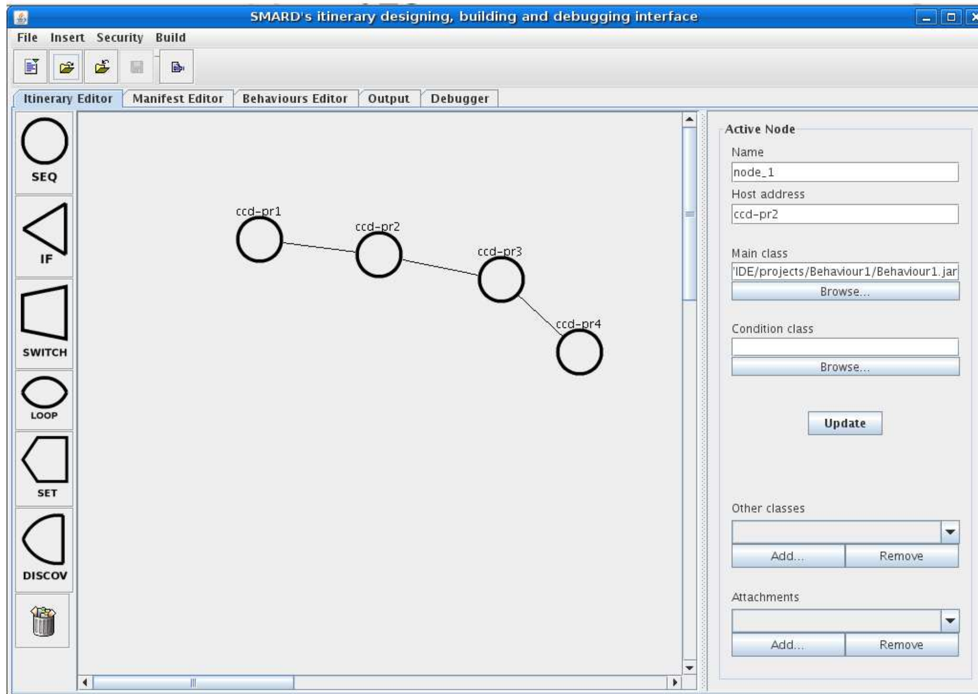

Figura 6.1: Itinerari de proves

l'agent, amb les seves classes. L'altre és l'itinerari amb les tasques serialitzades.

Hem fet diverses proves sobre aquest constructor. El pas més crític és la lectura de l'itinerari de la interfície. Hem intentat construir itineraris amb un, diversos i cap node, i només ha fallat el cas en que l'itinerari era buit. Hem corregit aquest fet per tal que només comenci el procés si hi ha nodes. Hem fet el mateix per quan un node no té una tasca local associada. El constructor fallaria a l'intentar serialitzar-la, i per tant, el procés s'ha de parar i no deixar construir els fitxers.

La resta del funcionament d'aquesta part ha estat correcta, no hi ha més casos en els quals pugui fallar.

6.1.3 Llançar l'agent al depurador

Per iniciar la depuració de l'agent, només cal anar a la pestanya corresponent de l'IDE i pitjar el botó *Start*. En cas de trobar-nos amb un itinerari buit no respondrà. Si hi ha un projecte obert, l'agent es començarà a executar, i veurem a la gràfica el canvi de color dels nodes que visita. A la figura 6.2 n'hi ha un exemple.

Les proves que hem fet sobre aquest element són, sobretot, intentar replicar totes les accions que podria fer l'usuari. Hem iniciat l'agent, l'hem aturat i tornat a iniciar repetidament. El fet que deshabilitem els botons que no s'haurien d'utilitzar en certs moments, per exemple intentar aturar el depurador quan l'agent no ha iniciat, ajuda a

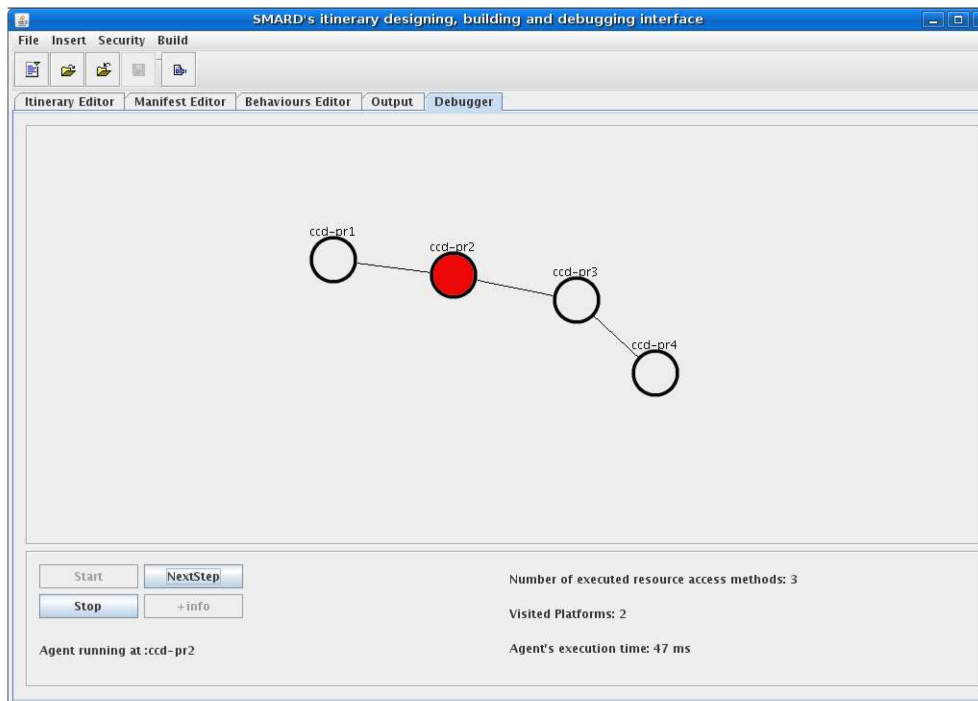

Figura 6.2: Procés de depuració de l'agent

evitar la majoria d'errors.

6.1.4 Visualitzar la informació de simulació a partir del sistema d'accés a recursos

A cada migració simulada de l'agent, podem veure un breu resum de la informació de la simulació. L'usuari no pot fer cap acció que produeixi algun error perquè aquesta zona del panell no és interactiva. Al final de la simulació s'habilita un botó (+*info*) que ens permet veure una finestra emergent que conté les dades completes de la simulació. A la figura 6.3 hi ha la primera part de les dades que ens ofereix, un resum complet de la simulació.

Després del resum general sobre la simulació, hi trobem la informació rebuda de cada plataforma. A la figura 6.4 hi ha un exemple d'aquestes dades, els mètodes que s'han simulat a la plataforma ccd-pr2.

El sistema de recollida de dades ha funcionat bé. Es reben correctament les dades dels nodes que han fet servir el sistema d'accés a recursos. Tots els mètodes que es mostrem a la finestra emergent s'han executat correctament a través d'*executeMethod*.

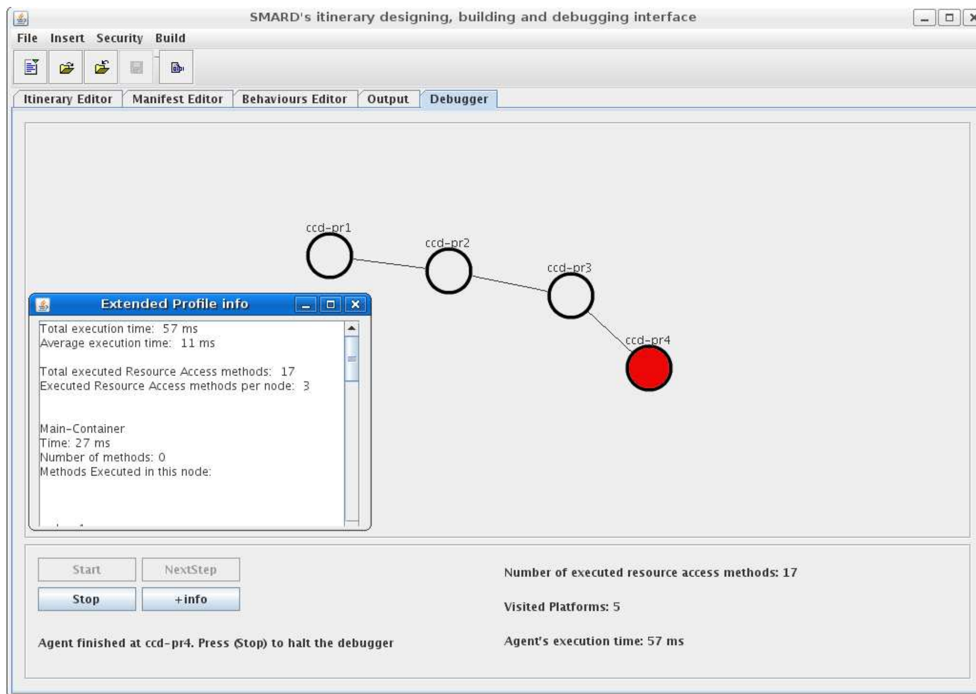

Figura 6.3: Dades de la finestra emergent

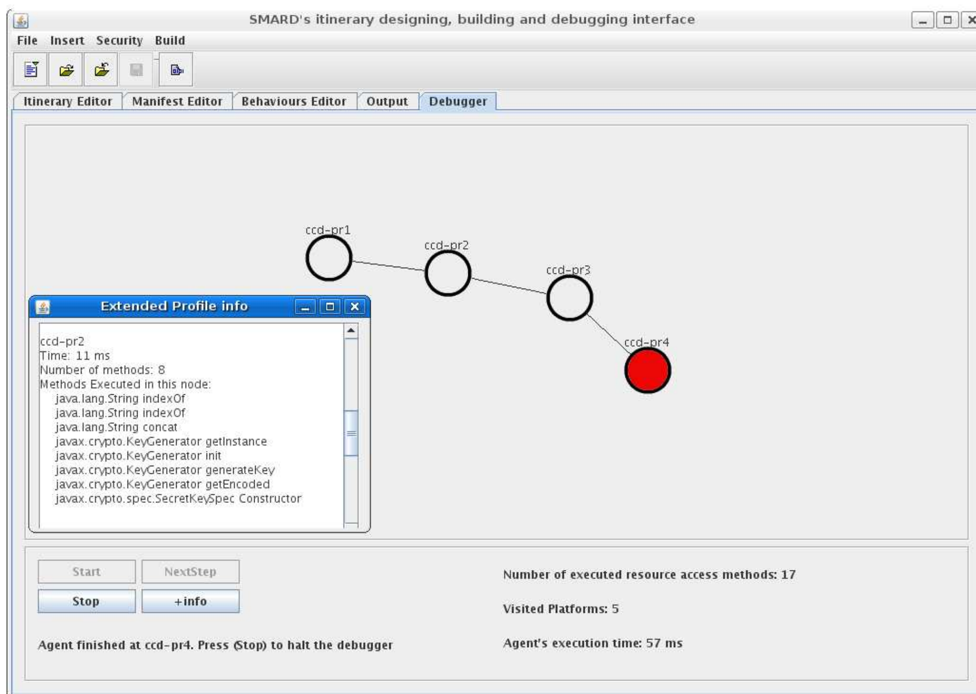

Figura 6.4: Dades de la finestra emergent (2)

6.2 Proves del sistema d'accés a recursos

A la secció anterior, s'ha explicat que els mètodes dels quals en donem informació són els cridats des del sistema d'accés a recursos, en concret, fent servir *executeMethod* i

executeConstructor.

Les proves que hem realitzat sobre aquest sistema han estat intentar cridar la major varietat de mètodes possibles. Com es descriu a [18], podem executar qualsevol tipus de mètodes menys els que intenten accedir a *inner anonymous classes*. Un tipus particular de crides són les que hem fet a mètodes estàtics, que podem cridar si operem com a l'exemple següent:

```
Cipher cipher = (Cipher)RA.executeMethod(Cipher.class,
     "javax.crypto.Cipher", "getInstance", new Object["AES"]);
```

A l'exemple, utilitzant *Cipher.class* aconseguim la referència a l'objecte estàtic, i la crida es fa correctament.

6.2.1 Proves de rendiment del sistema d'accés a recursos

Per simular l'impacte que tindrà el sistema d'accessos a recursos, hem executat un bucle amb els mètodes que hem provat a l'agent. Provant diferents números d'iteracions, hem valorat quin efecte tenien segons el volum de càlcul que fa l'agent. Hem provat de comentar alguns d'aquests mètodes, per fer una sèrie de tests variada. Els resultats que donem són la mitjana dels que hem obtingut en diverses execucions.

A la gràfica de la figura 6.5 hi trobem una comparativa dels resultats que hem rebut (dades en mil·lisegons).

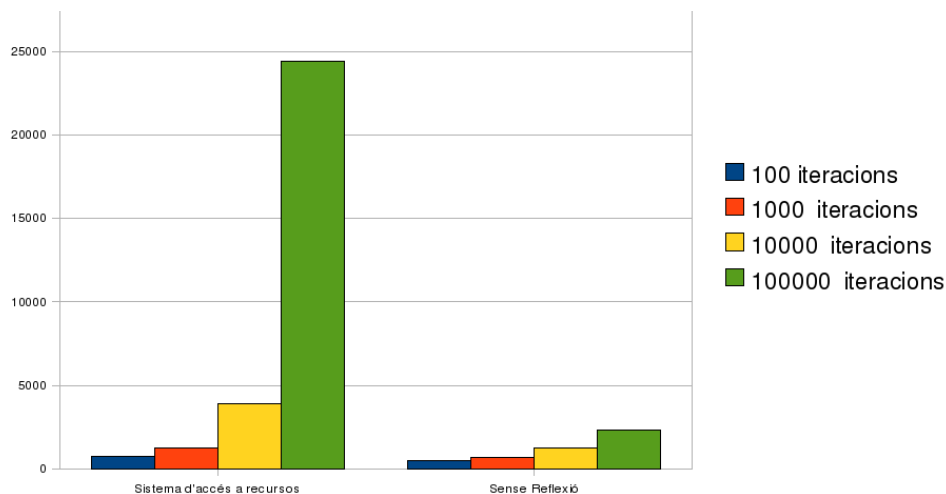

Figura 6.5: Comparativa del rendiment del sistema d'accés a recursos amb l'execució sense reflexió

Com podem veure, l'augment del temps que triga Java a executar els mètodes sense l'ús de la reflexió és aproximadament lineal. En canvi, fent servir el sistema d'accés a recursos que hem desenvolupat, l'augment és exponencial. Fins a les 10000 iteracions

el sistema respon bé, triga el triple que sense utilitzar-lo. Havent consultat [19] i [20], veiem que és una xifra més que acceptable. Però quan passem d'aquest número, la latència que provoca al nostre sistema és molt més gran.

Les conclusions són que, si bé aquest sistema d'accés a recursos perjudica el rendiment de l'agent, aquesta pèrdua d'eficiència és acceptable, tenint en compte les característiques que ens ofereix. S'hauria de considerar, però, que pot no ser idoni per executar agents que facin càlculs molt intensius, o pels que el rendiment sigui un factor molt sensible.

Capítol 7

Conclusions i línies d'ampliació

Un cop fet tot el desenvolupament, integració i proves de les parts que composen aquest entorn, és el moment de valorar l'acompliment dels objectius que ens havíem plantejat. També aprofitarem aquest capítol per tractar les qüestions que han aparegut durant el procés, que poden suposar noves línies de treball basades en les solucions que hem donat.

7.1 Valoració dels objectius

Al principi de l'estudi d'aquest projecte vam plantejar-nos un seguit d'objectius. Aquests objectius estaven directament relacionats amb desenvolupar les tres parts que formen aquesta aplicació. El procés que hem seguit es correspon amb el que vam establir a la planificació del projecte.

Primer de tot ens havíem proposat implementar una eina gràfica per poder depurar agents mòbils. La primera qüestió que s'ha tractat és com podíem executar un agent mòbil en una plataforma local, si l'agent intenta fer accessos a recursos no existents a la plataforma de simulació. Estudiant possibles solucions, s'ha vist que feia falta un sistema que ens permetés tractar aquest fet. D'aquí hem començat un procés d'anàlisi, disseny i implementació, que ha resultat en el sistema d'accés a recursos plenament funcional. Fent-lo servir, podem executar un agent que el faci servir en una plataforma local de proves, sense que doni cap error per no trobar un recurs que li faci falta.

Amb el sistema d'accés a recursos acabat, ja s'ha pogut desenvolupar l'eina gràfica per poder observar el comportament de l'agent a la plataforma de proves. Aquesta eina gràfica l'hem afegit a la interfície de disseny d'itineraris de l'entorn SMARD. El procés seguit per dur-la a terme ha estat similar al seguit amb el sistema d'accés a recursos, amb les seves fases d'anàlisi, disseny i implementació. A l'acabar el cicle de desenvolupament d'aquesta part, tenim un entorn de proves d'agents mòbils plenament

funcional, que ofereix la possibilitat d'analitzar la simulació d'aquests agents.

Un altre objectiu era integrar un constructor d'agents mòbils per tal de poder construir l'agent que s'havia dissenyat amb l'IDE. Enlloc d'implementar-lo partint de zero, hem agafat un constructor funcional i hem seguit un procés de desenvolupament que consistia en adaptar-lo a les nostres necessitats. Tenint-lo a l'IDE de desenvolupament i depuració d'agents, el programador pot fer el procés de disseny de l'itinerari, construcció i proves de l'agent des del nostre entorn gràfic.

Per últim, hem fet un seguit de proves sobre la integració d'aquestes parts, que han resultat en petites correccions. Un cop fetes aquestes proves podem afirmar que el funcionament ha estat l'esperat, i que per tant, hem acomplert amb èxit els objectius que ens havíem proposat al principi del projecte.

7.2 Línies de treball futures

En aquest apartat exposarem algunes possibles ampliacions que han sorgit durant el treball realitzat, i que es podrien desenvolupar a partir de les solucions que hem aportat.

Sistema de permisos i autoritzacions

La solució adoptada pel sistema d'accés a recursos ens proporciona uns mètodes (*executeMethod*, *executeConstructor*) que serveixen per executar el mètode de Java que li indiquem. En principi, estan concebuts per executar mètodes d'accessos a recursos, però seria possible implementar-hi un sistema de control de permisos d'accés i autoritzacions. *executeMethod* i *executeConstructor* podrien implementar algun mecanisme que donés permisos d'execució sobre els mètodes que l'administrador decidís a cada agent que s'executés a la plataforma on estigués implementat.

Informació dels errors de l'agent

La informació de simulació per fer *profiling* que desa l'IDE és el temps d'execució i els mètodes cridats dels agents a cada node que visiten. Aquesta informació és útil per comprovar quin ha estat el comportament de l'agent, però no ens dóna informació explicita sobre les excepcions que llança l'agent en cas de fallar. Es podria pensar en afegir aquest tipus d'informació de simulació. Una altra possibilitat de veure aquest tipus de dades seria donar l'*output* de la plataforma JADE de l'IDE, que normalment es pot veure per consola, en un panell de la interfície gràfica.

Constructor d'agents mòbils segurs

El constructor que hem utilitzat genera agents mòbils que no són auto-protegits. Aquest constructor no es troba en una versió final, actualment no xifra les tasques que hi ha a l'itinerari. L'ampliació lògica seria completar aquest constructor, per tal que el programador pogués provar les mesures de protecció de l'agent que volgués implementar.

Per poder fer agents mòbils auto-protegits s'hauria d'afegir a l'agent construït la capacitat de gestionar els seus mecanismes de seguretat. La plataforma JADE de proves també hauria de ser modificada, se l'hi hauria d'assignar una parella de claus, pública i privada, i implementar el servei per desxifrar les tasques de l'agent.

Bibliografia

[1] Fuggetta, A.; Picco, G. P.; Vigna, G., *"Understanding code mobility"*, IEEE Transactions on Software Engineering, Volume: 24, Issue: 5, Mai. 1998. Pages: 342-361.

[2] English Wikipedia; *"Grid Computing"*; <http://en.wikipedia.org/wiki/Grid_computing>. Últim accés Jun. 2008.

[3] Navarro, G.; Robles, S.; Borrell, J., *"Role-Based Access Control for E-commerce Sea-of-Data Applications"*, Springer: Berlin/Heidelberg. Lecture Notes in Computer Science, Volume 2433/2002. 2002. Pages: 102-116.

[4] Vieira-Marques, P. M.; Robles, S.; Cucurull, J.; et. al., *"Secure Integration of Distributed Medical Data Using Mobile Agents"*, IEEE, Intelligent Systems, Volume 21, Issue 6, Nov.-Dec. 2006. Pages: 47 - 54.

[5] Martín, A., *"Medigs, un sistema segur basat en agents per al descobriment i obtenció de dades mèdiques distribuïdes. Estudi i desenvolupament"*, UAB, Projecte de Final de Carrera d'Enginyeria Informàtica, Jun. 2007.

[6] Suna, A.; Fallah-Seghrouchni, A. E., *"A mobile agents platform: architecture, mobility and security elements"*, Springer-Verlag, Lecture Notes in Computer Science, Programming Multi-Agent Systems, ProMAS 2004, vol. 3346. 2005. Pages: 126-146.

[7] Lima, E. F. A.; Machado, P. D. L.; Sampaio, F. R.; et. al., *"An Approach to Modelling and Applying Mobile Agent Design Patterns"*, ACM Press, SIGSOFT Software Engineering Notes, vol. 29. 2004. Pages: 1-8.

[8] Garrigues, C.; Robles, S.; Moratalla, A.; et. al., *"Building Secure Mobile Agents using Cryptographic Architectures"*, 2nd European Workshop on Multi-Agent Systems. 2004. Pages: 243-254.

[9] Vigna, G., *"Cryptographic Traces for Mobile Agents"*, Springer: Berlin/Heidelberg. Lecture Notes in Computer Science, 1998. Pages: 137-153.

[10] Nebot, J., *"Entorn de desenvolupament d'agents mòbils per a la plataforma JADE/Marism-a"*, UAB, Projecte de Final de Carrera d'Enginyeria Informàtica, Jun 2004.

[11] English Wikipedia, *"Software Agent"*;
<http://en.wikipedia.org/wiki/software_agents>. Últim accés Jun. 2008.

[12] Robles, S., *"Mobile Agent Systems and Trust, a Combined View toward Secure Sea-of-Data Applications"*, UAB, Tesi Doctoral, Jul. 2002. Pàgines 11 - 12.

[13] FIPA <www.fipa.org>

[14] IEEE <www.ieee.org>

[15] SeNDA <https://senda.uab.cat>

[16] JADE <jade.tilab.com>

[17] Ametller, J.; Robles, S.; Ortega-Ruiz, J. A., *"Self-Protected Mobile Agents"*, AAMAS, Third International Joint Conference on Autonomous Agents and Multiagent Systems - Volume 1 (AAMAS'04), 2004. Pages: 362-367.

[18] Holser, P., *"Limitations of reflective method lookup"*;
<http://www.adtmag.com/java/article.aspx?id=4276>. Últim accés Jun. 2008.

[19] Newsom, J., *"What are the performance costs involved in Java reflection? E.g., looking up a method by name and then invoking it"*;
<http://www.jguru.com/faq/view.jsp?EID=246569>. Últim accés Jun. 2008.

[20] Sosnoski, D., *"Java programming dynamics, Part 2: Introducing reflection"*;
<http://www.ibm.com/developerworks/library/j-dyn0603/>. Últim accés Jun. 2008.

[21] Sun Microsystems, *"Why Are Thread.stop, Thread.suspend, Thread.resume and Runtime.runFinalizersOnExit Deprecated?"*; <http://java.sun.com/j2se/1.5.0/docs/guide/misc/threadPrimitiveDeprecation.html>. Últim accés Jun. 2008.

[22] Ford, M., *"How to Stop a Thread or a Task"*; <http://forward.com.au/javaProgramming/HowToStopAThread.html>. Últim accés Jun. 2008.

Firmat: Oriol Navarro Ferrer

Bellaterra, juny de 2008

Resum

Un dels principals obstacles per l'adopció dels agents mòbils, fins i tot per tasques per les que s'ha demostrat la seva idoneïtat, és la dificultat de desenvolupar-los. Aquest projecte aporta solucions per ajudar en aquest procés. Més concretament, per provar agents i observar-ne el seu comportament en un entorn de simulació, abans d'executar-los en un escenari real. Hem afegit un depurador i un constructor d'agents mòbils a la interfície gràfica de disseny d'itineraris de l'entorn SMARD. Fent servir afegit aquests nous components, el programador pot dissenyar, construir i fer proves d'un agent mòbil des d'aquesta aplicació.

Resumen

Uno de los principales obstáculos para la adopción de los agentes móviles, incluso para tareas en las que se ha demostrado su idoneidad, es la dificultad de su desarrollo. Este proyecto aporta soluciones para ayudar a realizar este proceso. Más específicamente, para probar agentes y observar su comportamiento en un entorno de simulación, antes de ejecutarlos en un escenario real. Hemos añadido un depurador y un constructor de agentes móviles a la interfaz gráfica de diseño de itinerarios del entorno SMARD. Utilizando estos nuevos componentes, el programador puede diseñar, construir y probar un agente móvil desde esta aplicación.

Abstract

One of the most important handicaps for the adoption of mobile agents, even for tasks in which they have proved to be suitable, is the difficulty involving their development. This project provides solutions to help in this process. Specifically, to test them and watch their behaviour in a simulation environment before running them in a real-world scenario. We have added a debugger and an agent builder to the SMARD's itinerary designing interface. By using these new elements, the programmer is able to design, build and test a mobile agent from this application.