Role Agent Pattern: a Developer Guideline

Giacomo Cabri
Dipartimento di Ingegneria
dell’Informazione
Università di Modena e Reggio Emilia
Via Vignolese, 905 41100 Modena Italy
cabri.giacomo@unimo.it

Luca Ferrari
Dipartimento di Ingegneria
dell’Informazione
Università di Modena e Reggio Emilia
Via Vignolese, 905 41100 Modena Italy
ferrari.luca@unimo.it

Letizia Leonardi
Dipartimento di Ingegneria
dell’Informazione
Università di Modena e Reggio Emilia
Via Vignolese, 905 41100 Modena Italy
leonardi.letizia@unimo.it

Abstract - Agents are an emerging technology that grants programmers a new way to exploit distributed resources. One of the hardest difficult in the development of this kind of application is the managing of agent interactions, since agents must interact in a collaborative and/or competitive way to achieve their task. Roles are a powerful paradigm that can be used to model agent interactions, both between two (or more agents and between an agent and the platform/node on which it is running. The concept of role has already been used as an extension of the object oriented programming, but this is not perfectly suitable to the agent’s paradigm. In this paper we propose a guideline to role developing and use, with regard to the agent technology. In particular we focus on the main problems that a role system and a role developer must take into account.

Keywords: roles, agent interactions, mobility, Java.

1 Introduction

An agent is an autonomous entity, able to perform its task(s) without requiring continues user interaction(s). Agents belong to two main different categories, not perfectly orthogonal, based on intelligence or mobility. Agents which belong to the above categories are called, respectively, intelligent agents (IA) and mobile agents (MA). This paper focuses on the latter type of agents (MA) and on their interactions on visited sites.

A mobile agent is able to move, autonomously and spontaneously, from an host to another, stopping and restarting its execution. This grants it the capability to act as an active network component, since it can inspect the network by itself. Mobility grants an easy and fast way to develop distributed applications. For example is possible to build an agent able to move from a loaded host to another with more available resources (i.e. unloaded) to keep a good load-balance on all network nodes. Of course mobility leads to a set of problems, such as the process/thread migration, security both for the agent (which could run on untrusted hosts) and for the hosts (which will support agents execution), interaction issues, and adaptability issues.

Thread migration and security issues are treated by the agent platform, which is a secure box that provides support for agent execution and mobility. The other issues instead (adaptability and interactions) are leaved to the agent programmer, and they are very important in developing efficient and scalable agent applications. In fact, since agents move themselves among different hosts or execution environments (i.e. platforms), they must quickly adapt to the environment, in which they are going to interact with other agents or with the execution environment itself.

Roles, a designing approach for applications ([2], [4], [5], [7], [18], [23] [24]), try to solve the above problems leading to more scalable, adaptable, and easy to handle applications. In the following sections we are going to explain how roles can be successful used by agents and developers.

This paper focuses on Java agents for two main reasons: (i) Java is the most exploited language to implement (mobile) agent platforms, thank to its portability, security, and network-orientedness; and (ii) the fact that Java relies on an intermediate bytecode allows modifying it to add new functionalities at runtime.

The paper is organized as follow: section 2 describes what a role is and how to develop robust roles and role systems; section 3 describes how the assumption/release of roles can be done by an agent; section 4 describes how to exchange data among different roles assumed by the same agent; section 5 introduces synchronization problems on role operations; section 6 describes how the agent should use the role operations, also showing some code examples. Finally section 7 reports conclusions.

2 Roles and Role-Systems

There are several role definitions, but the one that probably better targets the concept of role is: a set of capabilities, expected behaviour and knowledge that an agent can exploit by needing ([4], [5], [7], [17], [19]). When an agent decides to exploit role features it must assume the role, which means it must obtain the role implementation in a proper way that grants it the capability to use the role. After the use of the role the agent can release it, that means that the agent discard all role’s features.

Roles embed all information and capabilities needed in a particular execution environments. Thanks to this the agent (and its programmer) does not need to know particulars about the current execution environment but only on which role to assume and use to interact with (or to exploit) the environment itself. For example imagine an agent in charge of querying databases sited on different hosts. Since databases can be very different the agent should embed in itself all the needed knowledge to perform queries...
(for example the database driver name, the queries statement, etc.). A better solution is to use roles: if every host provides an appropriate role, simply called database_querier, which embeds every needed detail, the agent must only assume and use that role on the fly (see Figure 1). Using such role, the agent can simply discard every detail about the type of database (object database, relational database, etc.), about the needed drivers, the database location (local database, remote database), the socket type to use and so on.

Figure 1. The agent uses a role to access the database

As shown in Figure 1 the agent simply assumes the database_querier role and performs its action query that is in charge of querying the local database returning results to the agent.

Starting from the above considerations, it should be clear how roles allow separation of concern, leading to a simplest development. In fact, every local detail can be embedded in the role, while global details (such as the movement plan, the application logic, etc.) can be embedded in the agent itself. This also leads to “lighter” agents, since they do not need to carry on context-specific code. Furthermore local policies can be simply applied to agents via roles, since roles are like an execution environment interface for agents [5].

Roles are available to agents thanks to a role system, a system that allows role assumption, use and release.

2.1 What a Role System Must Do

A role system must provide a suitable way to manage roles. In particular it is very important that it provides:
1. an efficient way to store and manage available roles;
2. agents to freely choose and assume roles at run-time;
3. an abstraction level of roles information;
4. local policies can be applied to roles;
5. history of assumption/release is kept;
6. a role-state keeping mechanism is provided;
7. heterogeneous system compatibility must be possible.

The first issue is necessary when the number of roles grows up. In particular, since roles can be used to embed local services (such as database access), the number of roles provided by an execution environment depends on the number of services provided to agents on that host. For example, imagine that one host allows different services to mobile agents, such as database access, mail accounting, document exchange, and so on. In this scenario the role system administrator can provide a role for database update, one for database query, one for mail account registration, one for information exchange, etc. In this situation both agents and the system administrator need an efficient way to find roles, with regard also to the second issue, run-time choosing and assumption process. In fact, to grant a good level of adaptability to agents, it does not suffice to force them using roles, but it is need to leave they free to choose the best role provided by the execution environment (for their task) and to assume it. Since the agent programmer does not know particulars about the roles installed on one host, she cannot tie the agent with “static code dependencies” on a particular role, but must use a dynamic selection, of course provided by the role system.

This leads to the third issue: if the agent must assume a role, without knowledge about its real implementation, it must use semantic information. For example, with regard to the previous example on database query, an agent can ask to the role system to provide a “role to perform database queries”, without specifying which particular role. Semantic data is very important to face the dynamism required from today’s applications ([11], [12], [14]), since it allows changing the role implementation (maybe to a newest version) without changing the role searching mechanism.

The fourth and the fifth issues are related to security. As already mentioned before, a role can be exploited to force local policies to agents. With regard to the example of the database queries, the database_querier role could write queries on a log, an action that an agent which perform direct queries (i.e., without a role) could not do. In particular, if a role is implemented as first entity class, also language-embedded security capabilities, such as the Java security manager, can be exploited. Furthermore access lists, which can be based also on the history, can be applied to roles. For example, the role system can deny a role assumption if the agent has already assumed that role earlier, and this is the case of registration services, where a local policy can impose that only one registration per agent is possible.

A role system should also provide a role-state storing process (sixth issue), needed in some applications where the same agent (or different agents tied to the same application) returns to the same host re-assuming the same role. This allows agent to expand the role base knowledge provided by role developer and/or host administrator. A deeper analysis of this issue would be found in a next section.

The final issue makes easier the development of roles, since the same role can be exploited in different systems such as Java, Perl, Agent-Tcl, and so on. In this scenario the capabilities of the XML language can be exploited to build role definitions based on such language ([6], [11], [12]). Of course since the role system will be developed in a specific programming language, such as Java, all roles and data in general must be translated from XML to that language.

After the above considerations we stress that a role system should be composed of the following modules:
- a role archive module, that is a physical storage in which role are stored. It can be either a simple filesystem, a relational database and so on;
• a **role database module**, in charge of managing installed roles. It should provide users and agents front-end to the roles, allowing queries to its content and allowing installation and deinstallation of roles. We have separated the modules *role archive* and *role database* to stress how the latter is only an interface used to access the archive in a transparent way.

• an **assumption/release module**, in charge of adding or removing roles to agents, to update role assumption history and to interact with access control lists;

• a **security module**, that stores and applies access control lists to agents on the base of roles, and also interacts with language level security systems (such as Java security manager).

• an **XML parser module**, in charge of translating XML documents into roles or security entries (like ACLs) understandable for the role system.

Figure 2 shows how modules interact the one with the other.

![Diagram](image)

Figure 2. Modules of a role system

Of course, the role system must be integrated with the mobile agent platform, so that all its services are available to the agents running in the platform. This suggests building the role system with the same technology/language of the platform itself.

### 2.2 How to Develop a Role

The role system must allow some services to make the role utilization by agents easier, but the core of all interactions and of the application designing is the role itself. This paragraph details which things a role developer must take into account in her job.

The first phase of developing a role is the identification of which knowledge the role itself must provide, that means which capabilities can be exploited in different situations. Those capabilities must be encapsulated in the role, and removed from the agent. This step is quite similar to the Object Oriented developing phase in which a developer draws the class diagram, deciding which classes will be part of her system. The difference with the Object Oriented design is that the role is an entity on which access control list application is more relevant, so the granularity of design (i.e., which role must be developed) is more important. Please note that the access control list details, or more in general local policies, should not be taken into account by role developers, since it must be a local administrator choice granting one or more rights to roles. The developer must only take into account the possible use of local policies, but not which permissions will be granted to the role.

The second phase is the determination of the role context, which means with other roles, systems or applications the role itself will interact with. This means that all roles that will interact the one with the other will be developed in the same context, so that the data exchange becomes easier. The problem of the communication of course is related only to application contexts that exploits more than one role at the same time. For those applications in which roles are used only as an interface to local environment, excluding other agents, such as the querier role described above, it is not so important to tie roles to a specific context. Of course also these roles must be developed with regard to the execution environment, but since only one role is used per time, the development is simpler. Every role must be developed with regard to the roles it will interact with and also exchanged data must be formatted accordingly (for example by events).

The third step is the choice of the services available via the role. This means that is needed to choose which capabilities a role must grant to the agent, but also which capabilities used by the role must be hidden to the agent. For example, with regard to the database querier role, the agent can access the query only service without accessing some other available services, such as load_drivers that loads database drivers before starting querying. This phase is quite similar to the Object Oriented one, where the developer chooses the services available for each object. This choice is tied to the choice of semantic information about the role.

The fourth phase is the choice of the communication type, and this depends also on the role system on which roles will be installed (and used). For example, some role systems allow the use of events, other uses messages and so on ([8], [21]).

After these phases the main designing step of the role is complete, and the role developer can start the implementation process.

We remind that all agent platforms enable the *SecurityManager*, since they work with possible untrusted code. If a role needs a particular permission, the role developer should indicate it in the library documentation, so that platform administrators can set correctly their policy files. It is a very good solution to ship with the library a file which contains policy entries, so that is simpler to install (and get working) the role library. Here there is an example:

```java
grant codeBase
"file:${platform.home}/lib/roles/MyLibrary/**" {
permission java.util.PropertyPermission
"role.version", "read";
};
```
The above example supposes that roles need to read a generic role.version runtime property. The role developer should also provide an automatic way to install policies, but it is an administrator choice to use this mechanism, since it could be very dangerous installing permissions in a completely automatic way (think at what could happen with malicious roles). A good idea is the use of some compilation/installation, such as Ant [1], providing a specific target, for example install-policy, which installs a default policy that the administrator can use to check the library, modifying it later. Here there is an example for Ant:

```xml
<target name="install-policy" >
  <copy file="${java.policy}" tofile="${user.home}/.java.policy"/>
</target>
```

The above example installs a specific policy file in the user’s home directory.

3 The Role System Implementation

The assumption/release module (see Figure 2) is the core of a role system since it defines how role can be owned by an agent, and implicitly how it can use them, implementing appropriate mechanisms. Different role systems have been proposed in the area of mobile agents, and everyone has advantages and disadvantages. Since we are not interested to specific implementation issues, but only to how design better role and role systems, this paragraph gives a brief presentation of some existent role systems and of their assumption/release module.

The main difference among role systems is the level of dynamism provided for the role assumption/release. The dynamism influences also the agent programming style, since a more dynamic and flexible role system does not require a lot of specific source code to assume/use/release a role.

The more static system is the one who uses the normal Java reference mechanism [10]. In this approach a role is encapsulated in a stand-alone entity, but rather affects the behavior of components, it is evident the similarity with a role. The use of AOP, even if very dynamic, has intrinsic problems with roles. As first note there must be an external entity which order to “fuse” the agent with the role, so the assumption is not done by the agent itself but from something external. As second issue the aspect must know which class (i.e., agent) it will be applied, and this lead to less code reusability, since each role must be developed with regard to each different agent that can assume it.

Other approaches use the concept of roles and groups [3] to achieve a more scalable and dynamic systems, or use a meta-model level to define model of organizations [10].

A more flexible approach is the use of Aspect Oriented Programming (AOP) ([15], [20]), that is a new programming paradigm not developed specifically for roles. AOP starts from the consideration that there are behaviors and functionalities that are orthogonal to the algorithmic parts of the objects. So, it proposes the separate definition of components and aspects, to be joined together by an appropriate compiler (the Aspect Weaver), which produces the final program. The separation of concerns introduced by AOP permits to distinguish the algorithmic issues from the behavioral issues. Since an aspect is a property that cannot be encapsulated in a stand-alone entity, but rather affects the behavior of components, it is evident the similarity with a role. The use of AOP, even if very dynamic, has intrinsic problems with roles. As first note there must be an external entity which order to “fuse” the agent with the role, so the assumption is not done by the agent itself but from something external. As second issue the aspect must know which class (i.e., agent) it will be applied, and this lead to less code reusability, since each role must be developed with regard to each different agent that can assume it.

Other approaches use the behavioural reflection to achieve a more dynamic and reusable system ([12], [13], [14]). Also if these systems target their scope, they are very difficult to use for an “usual” programmer since they require to think in a very dynamic way.

Since the following paragraphs do not depend on the particular implementation of the assumption/release module, we will use the first mechanism (Java reference) where showing code samples, because it is simpler to understand and does not distract the reader with too specific details.

4 Implementation Issues

This section describes some relevant issues with regard to the implementation phase of a role. In particular issues such as how to maintain the role state for multiple sequential assumptions, the importance of role locality and how to exchange data through different roles are treated in next subsections.

4.1 Keeping the Role State

There are situations where the same agent must assume more than once the same role, or where different agents must assume the same role continuing the execution of the previous one. In these situations it is important to understand if the role state (i.e., role acquired data) must be kept. For example, imagine that the agent assumes a role that builds a list of preferred hosts (maybe via the user
interaction) to visit. With regard to Figure 3, the agent assumes a `HostListBuilder` role to get the list of preferred hosts from the user. The agent can then take back that list from the role deciding which hosts it will visit. This is an example of using roles to build a more flexible mobility, which is not completely embedded in the agent but depends on a more configurable component (the role). Nevertheless the important thing is that the user’s list can be modified by the agent, for example because it found that one host is no longer running, or something else.

![Figure 3. The use of a role to build a list of preferred hosts](image)

In this scenario the role state (i.e., the host list) should be kept also after the agent execution, so that another agent instance, later, can use the same role to get the update host list. This leads to the need that the state of the role should remain for later executions.

It is possible to use a state-writing on a storage system (for example a file or a tuple space) instead of the Java serialization mechanism. The role is in charge of saving and restoring its state from a file (or something else), so the role system is freed to keep a serialized roles repository and to understand if an agent needs a new role or a serialized one. The serialization mechanism is embedded in the role itself: for example the role can save its state in an XML document that can be processed by other roles, by humans and so on. Using this mechanism the refactoring of roles is no longer a problem, since new roles can be compatible with older state-saving because the state itself is not directly dependent on the role class definition.

4.2 Role Locality

A role is a context-dependent interface that allows the agent to communicate with the execution environment. With regard to the previous examples, the agent can use either a role to access a database or to communicate with the platform or other agents. The most important thing to note is that the role execution is strongly coupled to the local execution environment (for example the database type or the communication system of the platform). Since the execution environments could be different from one host to another, the above consideration leads to the promotion of locality ([7], [8]). This means that a role should be thought as an execution environment component, and should not be transferred from it. In other words, the role cannot move with the agent that has assumed it, so the agent must release the role before it moves to another host. Imagine, for example, that the agent in Figure 1 assumes the database querier role and, after accessing the database, it moves without releasing the role. As Figure 4 shows, the agent can successful access the database on host 1, since the role is designed to access a relational database; nevertheless if the agent tries to use the same role on a different host (i.e., moves with the assumed role) many problems can happen. In particular in Figure 4 b) the agent cannot access the database since now it is an Object Oriented database, while the assumed role is designed to access a relational database.

![Figure 4. Using the same role on different platforms](image)

Furthermore, even if the database on host 2 should be relational, the query statement used by the role could be invalid, or the database could not accept the connection from that role, and so on. The above considerations show why the role should be “a local role”, tied to the execution environment (i.e. the platform and the role system).

To avoid cases where the agent moves itself with a role assumed, the assumption/release module (see Figure 2) should keep a list of which agents have a role, so that it is possible to deny the agent mobility until it has released every role. This solution implies that the role system and the mobile agent platform communicate in a strict way.

4.3 How the Agent and its Role Can Exchange Data

One of the most important phases of the implementation sequence is the way by which the agent and the role can exchange data. In fact, the agent will probably pass arguments to the role actions, and the role will return results to the agent. The problem can be simply solved implementing each role service as a single method, that means that the agent invokes a method on the role and that method returns the result. But the data exchanging between the role and the agent is not only related to single services. Since a role is tied to the local execution environment, there is no reason why the agent, changing environment, must carry with itself the role. Typically it happens that the agent releases the role before moving and, eventually, assumes another role on the destination host. This is not a problem if the role is stateless, which means that all role variables are used only to store temporary values, not important for the agent. Instead, if the role stores its own variables values that the agent must carry within during its trip, the agent must ask the role for these values and store them into its variables. If every service in the role is implemented via a method, the agent can simply take the result of the method invocation (i.e., the return value) and store it into an internal variable.
Nevertheless, the agent cannot store the returned data in an ad-hoc variable, since this means tying the agent to the role implementation (i.e., coupling the agent and the role), skipping the semantic level of the role itself. Remember that the agent should search for and assume a role by its semantic information, which means by knowing what a role does but not how. This leads to a situation where the agent requires a particular service without knowing what type of results (and not what results) will be returned. Of course it is possible to exploit polymorphism, in particular in Java, where every object can be stored in an Object reference. Even if right, this situation presents a limitation: if an agent assumes more than one role, and all of them need to store data in an agent private variable, how can a single Object reference make it possible? A first answer can rely on the fact that the Object reference can store also an array, so everything can be stored in an Object reference. But this can lead to conflicts: if two or more roles try to store their data in the same location (i.e., at the same index of the array), what will happen? There will be a data-clash among roles. Furthermore, searching for a general purpose solution to this problem, we must take in account also different languages such as Perl, AgentTcl, etc. In these languages it is not possible to use polymorphism on an array, so the use of a solution as the above one is only related to Java simple cases.

The best solution to this problem is the use of one hash, an associative array that stores data by a key instead of an index [11]. In Java one hash can store every type of object, so it can be used to store data unknown by agents. Choosing a good key every conflict will be skipped, because every role will store data only in its location. Of course it is possible to perform introspection on the hash, obtaining every key and then every object stored for each key. The solution to this problem could be using encrypted data, and this is strictly recommended whenever the agent is carrying user’s personal data, such as a credit card number for on-line shopping. We repeat once more that the agent (and its developer) does not worry about the store data format, since data interpretation is a role task.

With regard to the hash used as a storage media, we are going to focuses on the key selection. As first issue note that the agent is not in charge of choosing a key, since only roles tied to the same application logic should know keys. But how can roles on different hosts deal with key use? A first solution could be the use of hard coded keys: the role itself stores keys as, for example, private variables. This solution, even if very simple, cannot deal with role refactoring that could, for example, imply a role key change. A better solution could be the use of a static class containing the key for a specific role. For example, in the following piece of code the class DatabaseRoleKeys is a singleton class and roles can use it and its getKey() method to get the hash key at runtime.

```java
public class DatabaseRoleKeys{
    private static String key="Role_key";
    public static String getKey() {
        return key;
    }
    protected DatabaseRoleKeys() {...}
}
```

Even if this solution is more Object-Oriented prone, it cannot deal with role refactoring, in particular with versioning. In distributed system the versioning problem is very relevant, since different versions of the same code could be incompatible. With regard to roles, consider this example: an agent is using the database_querier role on two different hosts. The role library is update on one host (host 2) but not on the other host (host 1). Imagine that the update implies the role key update, so keys used by the two role sets are different, so and an agent that assumes an agent on host 1 will find a different key on host 2. This means that the role on host 2 cannot use data acquired on host 1 (and vice versa) since the hash does not contains the right key (see Figure 5).

![Figure 5. Hash keys mismatch](image)

To avoid this problem we use once more semantic data: since each role has a semantic description the host key must be built on the base of this information. In fact semantic data is more stable than the role itself since, if it changes, the whole role behaviour changes. Please note that the key must be built only on the intersection of all semantic descriptors regarding tied roles. In fact each descriptor describes a different role, so key built on role B is different from key built on role A.

It is a good idea to use a hash function to calculate the key to avoid key collisions. Of course it is still possible to use a singleton class to obtain keys at runtime, such as shown below:

```java
public class DatabaseRoleKeys{
    public static String getKey() {
        // calculate the hash
        return descriptor.hashCode();
    }
    protected DatabaseRoleKeys() {...}
}
```

We stress how hashes, and their keys, are useful in managing data exchange between roles assumed by the same agent.
5 Using Role Operations

After an agent has assumed a role, it will probably use it to perform a particular task. Since a role is composed of actions, realized through methods, the agent should invoke those methods to perform one or more role action. Nevertheless, the usual method invocation strongly couples the agent and the role itself, leading to a more difficult code to manage in case of role refactoring. For example, supposing that the agent is using a role via a normal reference, consider the following piece of code:

```java
public class OperationDescription{
    public String keywords[]= . . . ;
    public int version= ... ;
    private String methodSignature= . . . ;
    // the method signature
    // a method to perform the invocation
    public Object performOperation() {
        // search for the method
        Method m;
        m=role.getClass().getMethod(signature);
        return m.invoke();
    }
}
```

Thanks to the above descriptor the agent can “understand” what operation it needs, and then invokes the associated method without using the standard notation. To do this, the agent needs a way to perform introspection on the role to find the method(s) to invoke. This is done with the method `performOperation` of the `OperationDescription` class. That method performs introspection on the role object which the operation belongs, and then invokes it using the given signature and returning to the agent the result(s).

In an approach such as the above, the agent is in charge of finding the right operation (i.e., the operation with the right description), invoking it via the description itself. The code will result as the following:

```java
public class MyAgent extends Aglets{
    protected Role myRole=new Role(..);
    // the agent role
    public void run(){
        // search for a good descriptor
        for(int i=0;op!=null && i<op.length;i++){
            if(op[i].keywords[0].equals(…) && … ){
                operationIndex=i; break;
            }
        }
        // tests keys, aim, etc.
        if(op[operationIndex].equals(…) && … ){
            result=op[operationIndex].performOperation();
        }
    }
}
```

Using the above approach the refactoring is no longer a problem, since it suffices to change the method signature in the `OperationDescription` to obtain a new behaviour. Please note that, since the method signature is contained in the operation description, the agent does not need to be recompiled. Every change (i.e. every role update) is completely transparent to the agent.

The fact that the agent has no warranties to which method will be executed seems to be a dangerous situation. As first issue consider that, since the `OperationDescription` is a normal class, it can be signed, so that the agent can decide to trust the description or not. As second issue consider that also the normal direct invocation can produce unexpected behaviour if the class loading is under spoofing. In fact, imagine that the agent loads a role reference to `it.unimo.polaris.roles.MyRole`; if the system is running a malicious class loader that has already loaded a previous class with the same name (and appearance) but different behaviour a method invocation will produce unexpected results. Simpler, discarding the class loader spoofing, if the role system administrator installs, instead of the previous role, a similar role with the same name and methods, but with a malicious behaviour, the agent will not be recognize the role substitution, trusting the role and invoking methods on it. The only solution to these problems is the use of a class signature, which is also the same solution to the operation descriptions, so operation description does not create more risks, but simply introduces more flexible run time behaviour.

6 Conclusions

This paper has proposed an analysis of the use of roles in developing mobile agent applications. We are toward the use of roles since they grant an user-friendly way to develop scalable applications, allowing separation of concerns, code
reusability and the capability to better adapt the agent itself to its current execution environment. Even if roles have been exploited in other branches of computer science, their application to mobile agents requires a deeper analysis to focus what is the more suitable way to implement a role, accordingly to the role system that the agent will find during its life/mobility.

The role system itself should provide a good interface to both developers and administrator(s), improving the use of roles. At the same time the role system should be developed with regard to needs of today’s applications, in particular interoperability and dynamism.

Roles should be developed in a stricter way than the normal Object Oriented design phase, since new problems are present in agent applications (serialization, synchronization, assumption/release, data exchange, role locality). We have stressed the more important phases that the role developer should take into account during the role designing. Even if this paper is related to Java agents (and roles) the concepts explained here can be applied also to other systems/languages.

Acknowledgment Work supported by the NOKIA Research Center of Boston, by the Italian Research Council (CNR) within the project “Strategical Project IS-MANET, Infrastructures for Mobile ad-hoc Networks”, and also by Italian MIUR.

References