

# Making Agent Roles Perceivable Through Proxy Bytecode Manipulation

Luca Ferrari  
Italy  
luca1978@gmail.com

Haibin Zhu  
Nipissing University, Canada  
haibinz@nipissingu.ca

## ABSTRACT

*Roles represent a great model to deal with interactions and sociality of autonomous entities like agents, and in fact in order to ease their adoption several role approaches have been developed in the Role Based Collaboration (RBC) field. Some of the main difficulties in designing and developing agent role based approaches are the needs of providing a good dynamism and a good perception of the played role. The former (dynamism) is the capability of an agent to assume, use and release a role at run-time; while the latter (perception) is the capability of an agent to perceive the role played by another agent without having to explicitly query such agent or the role environment about. While dynamism has been achieved with several techniques, the role perception is more difficult to reach and often requires deep changes in agent structures, like class refactoring. Other difficulties arise when the agent is masqueraded, for security reasons, by a proxy. This paper presents a role approach that enables Java agents to dynamically play and perceive roles. This approach exploits a dynamic class refactoring performed on the fly, in order to make visible and perceivable assumed roles.*

**KEYWORDS:** Java, Byte-code Manipulation, Roles, Agents

## 1. INTRODUCTION

Agents are autonomous and social problem solving entities that are often adopted to build today's complex systems. Thanks to their sociality, agents can cooperate (or compete) with other agents and entities in order to reach their aims. Several approaches to deal with such sociality have been proposed, including Tuple-Spaces [10], Group Computation [1], and Roles ([6], [11], [12]).

In human-being society, interactions are often modelled depending on human-being roles. This theory can be naturally applied even to the agent scenario, since agents can often play on behalf of their users, becoming a digital counterpart of human-beings. Although the great number of good quality role systems (see [9], [12]), there is still a lack in the *dynamism* of the role assumption/release process, that is often driven by compile time instructions. Moreover, since it is very difficult to achieve, almost all the existing role approaches do not present a good way to provide *role external visibility*, that is the played role *perception*. The latter can be defined as the capability of recognizing the role played by an agent simply "looking" at such agent from an external point of view.

To better explain what *dynamism* and *perception* means, consider a simple example taken from the real world: a person can play a role, say the *police person* role. Having such role allow him/her to interact with other people by means of the role facilities, for instance using the car radio. But the interesting part here is that the played role can be recognized by other people (the citizens) simply looking at the police person: in this case the uniform provides the role visible side, the role perceivable part. So other people can start interacting with the police person asking for police-services (e.g., need for help) simply recognizing the played role by its perceivable part, without the explicit need to ask him/her if is a police person. On the other hand, citizens will not start an interaction with other *police persons* who do not wear their uniforms since they cannot perceive the *police person* role. The dynamism here is emphasized when, at the end of the job, the police person releases his/her role and returns to be a normal citizen, becoming again a police person the day after. The same results can be applied to the computer science field, having dynamic role management and clear perception; the approach presented here aims to achieve these important features.

This paper presents the WhiteCat role approach, a framework written in pure Java that exploits dynamic byte-code manipulation in order to change agent class structures, en-

abling the role external visibility (perception). Thanks to the byte-code manipulation, it is possible to refactor the agent class structure on the fly in order to inject the role into the agent, changing the view that other agents have on it, allowing therefore the recognizability of the played role. WhiteCat does not aim at managing and ruling agent role-based interactions; the main aim of the WhiteCat approach is to provide a framework role developers can use to enhance agent capabilities, and that can be integrated in other higher level role systems like BRAIN [2] or E-CARGO [11]. This approach comes from the experience with the RoleX (also known as BlackCat) framework [2], keeping the same dynamism and flexibility, but simplifying the model and its adoption by designers and developers. Unlike its predecessor, the WhiteCat framework is not strictly tied to the agent field, and even if developed on top of a Multi-Agent System (MAS), it can be easily adopted in other scenarios. As an example, the authors are investigating the possibility to make services available as an OSGi bundle. In general, modularity, flexibility and ease to use of the approach makes it suitable for every Java role-based complex scenario, supporting both the role designers and developers as well as the role users (e.g., agent developers).

## 2. CONCEPTS AND DEFINITIONS

This section presents concepts, motivations and definitions behind the features provided by the WhiteCat framework.

### 2.1 Role Perception

The WhiteCat approach aims to support dynamic roles providing *external perception*, that is the capability of an agent to perceive and recognize a role played by another agent simply “looking” at the latter. This is inspired by real life, where human roles are often perceivable by an external point of view. In fact, a role is in general recognizable depending on the behavior (and capabilities) it provides to the playing person, or by some visible features. With regard to the example of the previous section, a *police person* role applied to a person, the role is immediately perceivable based on a visible feature (the uniform), as well as the behavior of the person playing the role (e.g., talking to the police radio, driving a police car, and so on). While it is still possible to recognize a person as a police person, depending only on his/her behavior, it is surely simple to recognize the role he/she is playing starting from a perceivable feature (like the uniform). Please note that several role approaches allow to get information about the role played by an agent at a specific time, but such information must be provided by

the agent itself or by a component of the role system, resulting in an “on-demand” process. The idea behind WhiteCat is, instead, to skip such step, exploiting the programming language facilities to dynamically and instantly obtain information about a played role. This means, for instance, exploiting the Java language `instanceof` operator, or performing run-time introspection, against an agent class in order to understand what role, if any, the agent is playing. Therefore a key concept in the WhiteCat approach is that once a role is assumed, the role is perceivable in the agent class structure. To achieve this, our approach injects the role into the agent class, changing its “language-side” structure (i.e., its class structure) in order to reflect the assumed role, and thus enabling other agents to introspect the modified class structure and perceive the assumed role. The fact that the agent class structure is modified on the fly depending on the role assumption process means that the class concept in the WhiteCat approach is a little different from that of Object Oriented Programming. While in the latter a class statically represents the unique type for all the running instances, in the WhiteCat framework a class represents only a *starting point* common to all the running agents (i.e., class instances), that can change dynamically. This guarantees that the Liskov substitution principle [15] will work, but since each running agent could have different run-time class structure from each other, it is important to stress that even the agent class is dynamic.

It is important to take into account that the role perception should not always be applied directly to agents. There are situations where agents run behind proxies ([4], [16]), objects that, for security or efficiency reasons, do masquerade agents making them not directly reachable. As a result, agents will not be able to “see” each other, but will just “see” each other proxies (see Figure 1 a). In such scenarios, modifying the agent class structure will not change the proxy one, and so will not be perceivable at all from an external point of view. To deal with an agent environment that exploits proxies, the WhiteCat approach can be configured to perform the role injection either directly into agents or their proxies (see Figure 1 b), therefore:

- agent behind a proxy: then the agent proxy class structure is changed to reflect the role external visibility;
- agent without a proxy: then the agent class structure is changed according to the role external visibility.

It must be clear that manipulating agent proxies instead of agents themselves has several advantages, in particular the agents result simpler and do not require to be stopped, reloaded and restarted due to their class manipulation [2]. Moreover manipulating the proxy represents a natural choice, since the proxy is the agent external visible part.

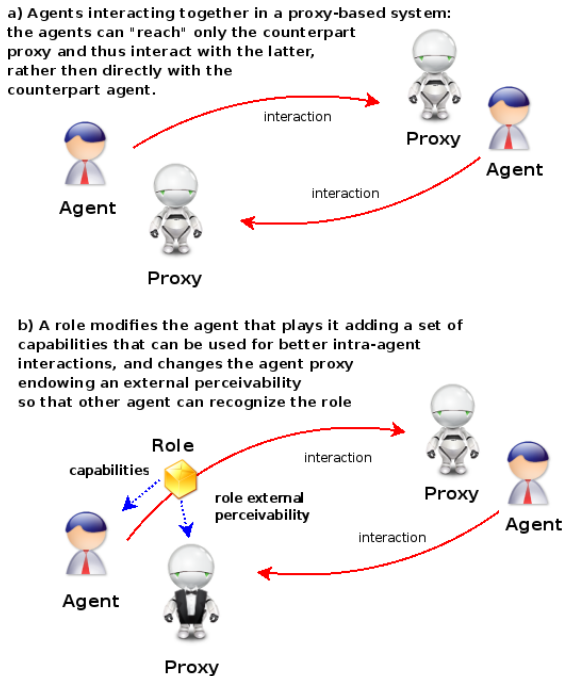


Figure 1. Agents, Proxies and Roles.

Please note that in either proxy or agent manipulation, there is a problem with “old” references: what happens to all the references to objects (either agents or proxies) that have been manipulated? They will still refer to an old instance (i.e., an instance with the old class structure), and thus should be invalidated. Invalidating all the references across the agent platforms is a task that cannot be performed in a portable way, so the WhiteCat does not force any invalidation of the references. Instead, it provides facilities to know if and when a reference is still valid or not, and thus to update the reference itself to the new agent/proxy.

## 2.2 Level of Perception and Role Definition

Before proceeding with the definition of role adopted by this approach it is important to express the so called *level of perception* that can be applied to a role. The system recognizes three possible levels of visibility/perception that can be applied to a role:

- *invisible*: the role is not perceivable at all from an external point of view. This means that an agent is using a role that other agent cannot discover easily (i.e., only “observing” the agent structure);
- *visible*: the role is perceivable from any external entity, that means the agent structure reveals the role it is playing;

and

- *public*: a *visible* role that exposes also a set of capabilities that even agents not playing the role can use.

The first level of perception (*invisible roles*) is the simplest one to deal with: briefly an agent uses the role in a “private” way, that means without making its role assumption publicly available. This means that other agents cannot understand what role an agent is playing simply “observing” its structure. However, such agent is enhanced by the role capabilities and can exploit them to interact with other agents. With regard to the police person example, it is like he/she is playing a *plain clothes* role. Please note that a lot of role approaches implicitly use this kind of role perception; this means that other existing role(s) (and approaches) can be imported into WhiteCat using this level of perception. The *visible role* perception level makes publicly available the information of the role played, so that external entities can independently recognize the played role without needing to ask to the role system or the playing agent itself. This is the case of the above mentioned example of the police person. Finally, the *public role* perception level is the most powerful one: not only the role is perceivable, but it is also exploitable from external entities. This means that other agents or entities can exploit a service offered by the role played by the target agent. In this case there could be a direct interaction with the played role instead of with the agent that is playing it. With regard to the example of the police person, this could be the case where a lieutenant exploits the police person role to force the latter to do some particular task.

Having defined the perception levels that can be applied to a role, it is now possible to define a role in the WhiteCat framework. The WhiteCat approach considers a role as a stereotype of behavior(s) common to different agents (or entities), and in fact a role defines a set of services and requests a playing agent can offer or must obey to [12]. Once assumed, a role provides additional capabilities to the agents that are playing it, granting at the same time a certain level of external perception of the role itself. In short a role can be considered as a capability enhancer, and its definition is very general in order to make it easily adaptable to not strictly agent-tied scenarios. In particular, each piece of code that can enhance the capabilities of an agent can be used as a role<sup>1</sup> in the WhiteCat approach.

Figure 2 shows how role perception levels can be nested: a private role is always applied directly to the agent (never to its proxy), since the agent is the only one that can exploit the role capabilities. A private role can be extended

<sup>1</sup>WhiteCat roles must match a few requirements in order to be usable by the framework; however, as it will be explained in Section , there are only a few non-invasive changes required.

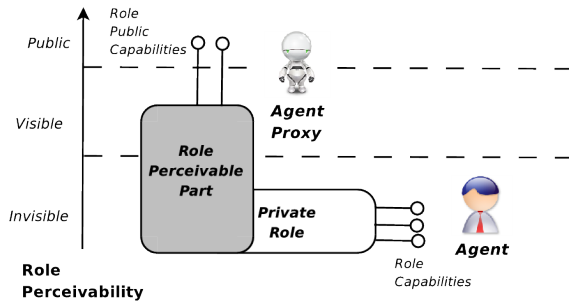


Figure 2. Role Visibility Levels.

to export a part as perceivable, becoming so a visible role, where the invisible part is still applied to the agent and the visible part is applied to the proxy. Finally, if a visible role exports a set of public capabilities or services, it becomes a public role. Again, the public part (the perceivable property and the public services) is applied to the proxy, while the private part to the agent. The choice among the above kind of perception is left to the role designer, and cannot be changed during the role assumption.

### 2.3 Role Repositories and Descriptors

WhiteCat proposes as a complete framework for supporting role engines and environments, and therefore it provides not only role design facilities and concepts, but also a *role repository*. This is a central storage for roles, that support dynamic installation and removal of roles making them available (or unavailable) at run-time. Agents can query the role repository in order to get the list of available roles, and can search through its content to find a role that is suitable to their aims. The role repository allows the query by means of *role descriptors* [8], semantic abstractions over a role. A role descriptor describes what a role does without the need to know its artifacts (classes, methods and so on). This means that a descriptor can be used with semantic meaning (e.g., the aim of a role) disregarding implementation details (syntactical details). Descriptors provides also a good degree of flexibility in the management of the whole system, including the installation of roles into the repository. For more details about descriptors please see [8], [2].

## 3. IMPLEMENTATION DETAILS

This section details the WhiteCat implementation, that has been realized in pure Java, exploiting the Javassist ([3], [18]) byte-code manipulation library and the AspectJ [17] support, and is publicly available under the GNU GPLv3 licence. Even if the WhiteCat approach can work either

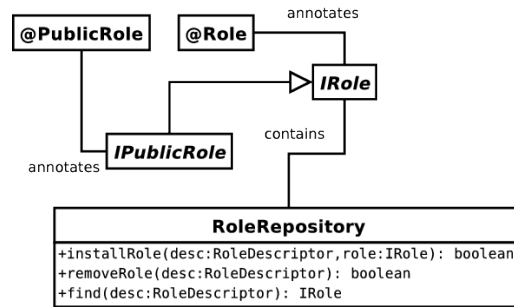


Figure 3. Main Role-Related Classes.

with or without agent proxies, in the following sections it is supposed that roles will be adopted in a proxy-enabled agent system.

### 3.1 Role Classes

WhiteCat provides different classes, interfaces and annotations in order to implement each role visibility described in Section (see Figure 3).

The idea behind role perception is quite simple: each perceivable role must be identified by a custom annotation, that is an annotation developed by the role designer. Such annotation will be applied to the agent proxy (or the agent itself if there is no proxy support) to indicate that the agent is playing the role identified by the annotation. In this sense, the custom annotation is equivalent to the uniform of a police person role. Roles are required also to implements a few interfaces, however developers are left free of defining their role types.

Everything that is related to a role must be annotated with the `@Role` annotation. Moreover, each role must implement the `IRole` interface, that is a tagging interface used to identify a role implementation; the role repository (see Section ) deals only with this kind of interface. Being a tagging interface, it is easy to refactor legacy code to make it available as role within the WhiteCat framework. The implementation of the `IRole` interface is the minimum requirement for a role, and therefore corresponds to the only requirement for the definition of the private role.

A visible role must also annotate its `IRole` implementation with a custom annotation, that must be itself annotated with the `@Role` one. The custom annotation is used as a tag to indicate the type of (visible) role the agent is playing, and therefore will be applied to the agent proxy once it has assumed the role.

A public role is the most complex one to define: it must provide an implementation of the `IRole` interface, a cus-

tom annotation used for external visibility and also a custom interface of type `IPublicRole`. The latter interface represent the role perceivability and will be applied to the agent proxy in order to make the role perceivable. The `IRole` implementation must be annotated with the `@PublicRole` annotation, that contains information about the custom interface to apply to the agent proxy.

All the three role types above must have an implementation of the `IRole` interface, and can be upgraded/downgraded through the visibility levels simply changing the annotation mechanism. In this way the role developers are free to define roles and to change role perceptions depending on the application contexts. Such freedom is emphasized by the fact that the visible role annotation and the public role interface can be customized by role developers, enabling also the reuse of legacy code. What defines a role visibility level is the mixing of interfaces and annotations at the design/deploy phase; Table 1 summarizes the steps required to provide each role visibility level. It is important to stress how perception levels are nested, simplifying the upgrade/downgrade of role visibility just providing required annotations and/or interfaces.

**Table 1. Role Visibility Required Steps.**

Role level	Implementation steps
<i>invisible</i>	implement the <code>IRole</code> interface.
<i>visible</i>	implement the <code>IRole</code> interface; define a custom role annotation, annotated with <code>@Role</code> annotate the <code>IRole</code> implementation with the custom annotation.
<i>public</i>	implement the <code>IRole</code> interface; define a custom role annotation, annotated with <code>@Role</code> annotate the <code>IRole</code> implementation with the custom annotation. create a <code>IPublicRole</code> subinterface implemented by the role and applied to the proxy annotate the <code>IRole</code> implementation with <code>@PublicRole</code> store information about the role interface in the annotation. height

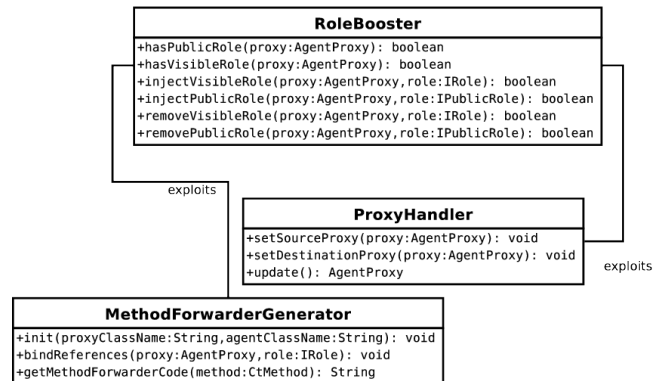
### 3.2 The Role Booster

In order to apply changes to the agent/proxy class structure, the WhiteCat approach performs run-time byte-code manipulation: the idea is to change a class definition dynamically as required. Changes are applied during class loading: a role assumption/removal forces a proxy class reload. For instance reloading and manipulating a proxy class will allow external entities (e.g., other agents) to obtain dynamic role information from the (changed) proxy class structure.

This means that introspection must be performed against an agent proxy to perceive the role(s) its agent is playing.

The `RoleBooster` is a special Java class loader used to manipulate classes to add (or remove) role properties and perception. The `RoleBooster` provides also static services to easily obtain information about a proxy and the role of its agent without requiring developers to deal with Java introspection.

As shown in Figure 4, the `RoleBooster` provides methods to inject and remove changes tied to role perception, but please note that there are no methods for invisible roles. In fact, in the case of invisible roles there is no need to contact the `RoleBooster`, since the role will be exploited without any changes to the external visibility. Instead, in the case of a visible or public role, the `RoleBooster` must load a new manipulated proxy class. In this case the `RoleBooster` defines a subclass of the current proxy class (even if already manipulated); such subclass will guarantee the compatibility between the old (i.e., not manipulated) class and the new (i.e., manipulated) one [15] and will avoid problems with the Java class definitions. In fact, in Java it is not possible to define different classes with the same name more than once using the same class loader, therefore the `RoleBooster` assigns each proxy subclass a dynamically computed unique name. It then creates a new empty class, subclass of the current (i.e., not manipulated) proxy class, and assigns to it the computed name. After that the class is endowed with the role capabilities: the role annotation (if the role is at least visible) and the role public interface (if the role is public). The subclass definition process is the same as that in the case of role assumption or release. Multiple manipulations (i.e., role assumptions or releases) will result in the definition of multiple subclasses of agent proxies. Summarizing, the original inheritance chain of an agent proxy will extended as much as the agent assumes visible or public roles, and will shrink down to the



**Figure 4. Role Booster Related Classes.**

original chain as much as the agent releases such roles.

The `RoleBooster` has a modular structure, and exploits a few pluggable components that can be customized to take control over the byte-code manipulation process. The two most important pluggable components are the *Proxy Handler* (implemented through the `ProxyHandler` class) and the *Method Forwarder Code Generator* (implemented through the `MethodForwarderGenerator` class). The *Proxy Handler* is responsible of keeping the internal state of a proxy coherent among byte-code manipulation and class reloading. The *Method Forwarder Code Generator* is responsible of exporting public role services through the agent proxy. In fact, when a public role is assumed, its services (e.g., methods) must be available to other agents through the agent proxy. To achieve this, the *Method Forwarder Code Generator* injects an implementation of each role method in the agent proxy. The default method implementation simply forwards method calls from the proxy to the role.

To better understand how the `RoleBooster` and the other components work, consider Figure 5 that shows a visible role assumption. In the beginning the agent searches for a visible role by means of role descriptors (step 1), then it requests the `RoleBooster` to inject the chosen role (step 2). The `RoleBooster` creates a subclass of the current agent proxy class (step 3), assigns a unique name to it and applies the visible role annotation. After that, the `ProxyHandler` creates a new instance of such manipulated class, copies the state of the old proxy state to such new instance, and makes the instance available (step 4). Since this moment the role is perceivable on the proxy, and an external entity can obtain an updated reference to the new proxy and can analyze the role the agent is playing. If the agent, after the above role assumption, wants to assume a public role, the `RoleBooster` performs similar steps but, as shown in Figure 6, also the `MethodForwarderGenerator` is exploited this time. The latter creates implementations for each method specified in the public role interface (step 3), so that the new proxy class is a consistent implementation of the role interface. Again, the new proxy object is created and initialized using the `ProxyHandler` object (step 4).

It is important to note that while the entire process manipulates a proxy class structure, external entities such as other agents can keep their references to the original (i.e., not manipulated) agent proxies. Such references will still work even after a role assumption/release, and this thanks to the fact that the byte-code manipulation process *extends* the proxy inheritance chain and performs manipulation at a subclass level, without changing or invalidating the superclass, that is not the agent proxy. It is clear that in the case of a role assumption, each agent or entity that refer-

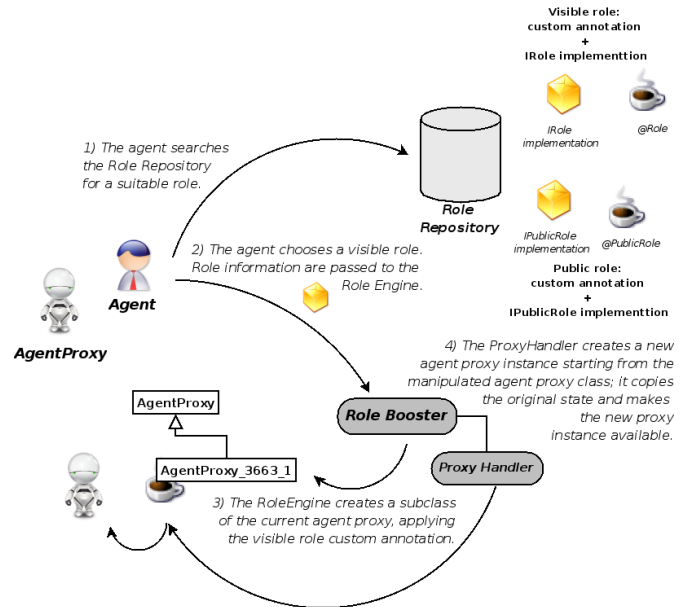


Figure 5. A Visible Role Assumption Process.

ences the unmanipulated proxy class instead of its manipulated subclass will have not up-to-date information about the played role. Each entity can require, at every time, a *proxy update*, obtaining a new proxy instance with the possibly updated class structure, and therefore can obtain the updated information about agent roles. Please note that this asynchronous update does not represent a limitation, since entities could not be interested in an agent role assumption case-by-case, but in the agent role status only in specific conditions or at specific times.

## 4. AN APPLICATION EXAMPLE

In order to better explain how the WhiteCat framework works and enable developers to deal with dynamic and visible roles, this section presents a simple agent-based application example.

Imagine a set of agents that must perform queries against a database, for instance to retrieve good prices or to store results of computations. To better abstract the agents from database related details, and to get a classification of agents by their aims and behaviors, `DatabaseUser` and `DatabaseAdministrator` roles are built (see Figure 7). The former allows an agent to retrieve the price of a product, the latter allows an agent to perform administrative tasks on a database, like maintenance or permission grant.

As readers can see from Figure 7, both roles are an-

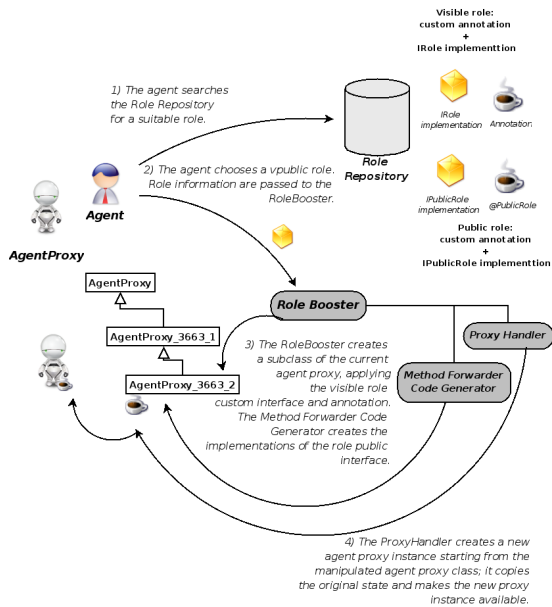


Figure 6. A Public Role Assumption Process.

notated with the `@Role` annotation, meaning that they can be used in the WhiteCat role system. Moreover, both roles implement the `IRole` interface, that makes the roles at least invisible. Let's concentrate on the `DatabaseAdministrator` role first: such role should be visible, since other agents should be able to find an agent acting as administrator (e.g., to ask it to perform administrative tasks or grant permissions), moreover such agent is going to provide services through its role (e.g., granting permissions), so the role must be promoted to be *public*. This means that the `IRole` implementation (i.e., the `DatabaseAdministrator` class) must be annotated with the `@PublicRole` annotation, that provides the role interface name to be applied to the agent proxy. Please note that such interface, the `IDatabaseAdministrator` of Figure 9, is implemented by the role itself, keeping the role and its external perspective coherent. Setting the `DatabaseAdministrator` role public means that an agent can cast a proxy reference to the `IDatabaseAdministrator` interface, assuming that the proxy is hiding an agent implementing such role, and can then invoke methods of the public role interface having access to the played role (see for example Figure 10). It is important to note that invoking a role method through the public role interface does not mean that an agent is overtaking control over the role played by another agent. In the example of Figure 10, an agent is exploiting the `IDatabaseAdministrator` to ask for a permission granting (method `IDatabaseAdministrator.grant(..)`);

```

@Role()
@DBRoleAnnotation()
public class DatabaseUser implements IRole {
    public float getPrice(String productCode){..}
}

@Role() @DBRoleAnnotation()
@PublicRole(
    roleInterface =
        "whitecat.example.IDatabaseAdministrator",
    roleAnnotation =
        "whitecat.example.ExampleRoleAnnotation"
)
public class DatabaseAdministrator
implements IRole, IDatabaseAdministrator {
    StringBuffer backupDatabase(String dbName) {..}
    boolean createDatabase(String dbName) {..}
    void reindexDatabase(String dbName){..}
    boolean grantPermission(String dbName,
        String user,String permission){..}
}

```

Figure 7. Application Roles.

```

@Role() @Retention(RetentionPolicy.RUNTIME)
public @interface DBRoleAnnotation { }

```

Figure 8. Custom Annotation To Make A Role Visible.

such method call will be passed to the `DatabaseAdministrator.grant(..)` method in the role implementation. What will happen then depends on the role implementation: in some cases the method call will produce the permission grant without any decision by the agent playing the `DatabaseAdministrator` role, in other cases the method call will just produce a request for the administrator agent that will decide how to proceed. In other words, exploiting public roles does not implicitly means that external entities will use other agent's roles without having to assume them, but that they will be able to easily perform requests for such role services.

The `DatabaseUser` role of Figure 7 does not mind to be public, since it does not offer any service but rather exploits services. However, it could be interesting to make

```

@Role()
public interface IDatabaseAdministrator {
    StringBuffer backupDatabase(String dbName);
    boolean createDatabase(String dbName);
    void reindexDatabase(String dbName);
    boolean grantPermission(String dbName,
        String user,String permission);
}

```

Figure 9. The `IDatabaseAdministrator` Interface.

```

// get a proxy reference of the database
// administrator agent
AgentProxy adminProxy = ...
// is the agent playing the
// IDatabaseAdministrator role?
if(adminProxy instanceof IDatabaseAdministrator){
    // ask for a permission grant
    ((IDatabaseAdministrator)adminProxy)
        .grantPermission("goods", "fluca", "SELECT");
}

```

**Figure 10. Exploiting A Public Role Services.**

```

public class SnoopAgent extends Aglet{
    public void run(){
        // get proxies of other agents
        AgentProxies proxies[] = getContext().
            getProxies();
        for(int i=0; i<proxies.length; i++){
            // update this proxy status
            proxies[i] = proxies[i].update();
            // is this an administrator agent?
            if(proxies[i].class.isAnnotationPresent(
                DBRoleAnnotation.class)){
                // the agent is playing a database role
                if( proxies[i] instanceof
                    IDatabaseAdministrator)
                    // require admin services
                else
                    // the agent is playing a DatabaseUser role
            } } } }

```

**Figure 11. Perceiving Other Agents Roles.**

such role visible, so that other agents playing the same role can recognize each other as database users and can start communicating and cooperating together (e.g., exchanging data retrieved from different databases). For such reason a custom annotation has been created: the `@DBRoleAnnotation` (see Figure 8) is used to mark the role `DatabaseUser` as a visible role, and it will be applied to the agent proxy once the former has assumed the role. Please note that such annotation has been used to mark also the `DatabaseAdministrator` role. In other words, the annotation `@DBRoleAnnotation` is a custom annotation used to make a database related role visible. Please note that the two roles could have been annotated with different annotations, it does not matter since the `RoleBooster` applies each annotation present in the role classes to the manipulated proxies.

Having defined two different roles, an administrator one (public) and a user one (visible), an agent can then analyze the proxy structure of another agent to understand which role it is playing. Figure 11 reports an example of Aglets code [16] that shows an agent introspecting other agent proxies to understand which roles they are playing.

It is important to note how, thanks to the role perception, agents are able to trim their interactions depending on the role played by their counterparts, making the whole system playing in a more *autonomic* way [19].

## 5. RELATED WORK

There are three main role approaches that are worth be compared with WhiteCat; a more complete role approach comparison is reported in [9].

Kendall's approach [14] exploits AOP ([17], [13]) in order to encapsulate agent cross-cutting behaviors into roles. Briefly this approach binds agents to roles, chosen among those available in a role catalogue, using AOP. The approach results in a good dynamism, but it has two main drawbacks: first of all the agent cannot autonomously decide about the role assumptions, since they are decided through the AOP weaver; second it does not impact the role external visibility or perception.

The R4R approach [5] is another AOP based approach that does not deal with role assumption/dismissal, but rather with role deployment. The idea is to exploit AOP in order to better integrate a role with the context where it will be adopted, and therefore this approach can be considered as complementary to Kendall's or other role approaches. Again, this approach does not consider the role external visibility.

RoleX [2] is of course the main similar approach, being the predecessor of the WhiteCat one. However, RoleX drawbacks motivated the development of WhiteCat. First of all RoleX manipulates directly the running agent in order to inject the role into it. This requires the running agent to be stopped for the time required by the byte-code manipulation; the agent must then be restarted in order to continue with its life cycle. RoleX does not consider at all the possibility of having agent proxies, and more important, it does not consider that agents could not be always directly accessible to other agents. This means that, even injecting the role into the agent, other agents could not be able to see the role external visibility simply because they cannot "see" the target agent directly. On the contrary, WhiteCat recognizes the importance of proxies as *agent external shields* and therefore concentrates on them as the points to which external visibility must be applied. Another RoleX drawback is its complexity: in RoleX an agent dynamically changes its structure, meaning that its class will have members it did not have before and that will no more be available after a role dismissal. While this is really dynamic and challenging, it is very complex to deal with for a developer. In the WhiteCat approach, the agent is never changed, never stopped, no members are added (or removed) from

its classes; the only thing that could change is its proxy. Moreover, changing the agent class structure requires to immediately invalidate any reference to the agent, while in WhiteCat this is not required, since the agent/proxy inheritance chain is only extended, and thus all the references in the system are always valid (even if not updated from a role point of view). Finally, WhiteCat has better performance than RoleX [7], since it manipulates less bytecode for every role assumption/release.

## 6. CONCLUSIONS AND FUTURE WORK

This paper presents the WhiteCat framework to support dynamic role assumptions by agents focusing on the role external visibility or perception by means of byte-code manipulation on the agents' proxies. However, the approach works well even if the platform does not support proxies. The WhiteCat approach defines a role in a very general and flexible way, allowing developers to even reuse legacy code for defining roles. Moreover, thanks to a clear and simple separation of role perception levels, an existing role can be easily modified to another level with a minimal effort. Moreover the framework keeps the use of roles very simple and easy to be integrated in the programming language. Future work includes the design of a *proxy central storage* that will ease the proxy retrieval and update, as well as a *notification system* to notify cooperating agents when a proxy they refer to has changed due to a role assumption/release. In this way agents can require a proxy update to get the latest proxy state. Finally, authors are investigating the release of the framework as an OSGi bundle.

## REFERENCES

- [1] C. Ghidini B. Hirsh, M. Fisher. Programming group computations. *Proceedings of the First European Workshop on Multi-Agent System (EUMAS)*, December 2003.
- [2] Leonardi L. Cabri G., Ferrari L. Injecting roles in java agents through run-time bytecode manipulation. *IBM Systems Journal*, 44/1:185–208, 2005.
- [3] S. Chiba. Javassist, a reflection-based programming wizard for java. *In Proceedings of the ACM OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, October 1998.
- [4] R. Johnson J. Vlissides E. Gamma, R. Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*, ISBN 0-201-63361-2. Addison Wesley.
- [5] L. Ferrari. Binding agent roles to environments: the r4r approach. *In Proceedings of the 2008 International Symposium on Collaborative Technologies and Systems (CTS 2008)*, May 2008.
- [6] F. Zambonelli G. Cabri, L. Ferrari. Role-based approaches for engineering interactions in large-scale multi-agent systems. *Software Engineering for Multi-Agent Systems II, Lecture Notes in Computer Science*, 2940:243–263, "April" 2004.
- [7] L. Leonardi G. Cabri, L. Ferrari. Manipulation of java agent bytecode to add roles. *In Proceedings of the 2nd International Conference on the Principles and Practice of Programming in Java*, June 2003.
- [8] L. Leonardi G. Cabri, L. Ferrari. Role agent pattern: a developer guideline. *Proceedings of The 2003 IEEE Systems, Man and Cybernetics Conference*, "October" 2003.
- [9] L. Leonardi F. Zambonelli G. Cabri, L. Ferrari. A survey about role-based interaction proposals for agents. *Technical report DII-AG-2005-1*, 2004.
- [10] L. Leonardi M. Mamei F. Zambonelli G. Cabri, L. Ferrari. *Uncoupling Coordination: Tuple-based Models for Mobility (chapter of the book Mobile Middleware)*. Taylor and Francis CRC Press, 2006.
- [11] M. Zhou H. Zhu. Role-based collaboration and its kernel mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 36:578–589, July 2006.
- [12] M. Zhou H. Zhu. Roles in information systems: A survey. *IEEE Trans. on Systems, Man and Cybernetics, Part C*, 38:377–396, May 2008.
- [13] A. Mendhekar C. Maeda C. Lopes J. M. Loingtier J. Irwin K. Gregor, J. Lamping. Aspect-oriented programming. *Proceedings of the European Conference on Object-Oriented Programming*, 1241:220–242.
- [14] E. A. Kendall. Role modelling for agent systems analysis, design and implementation. *IEEE Concurrency*, 8:34–41, April 2000.
- [15] B. Liskov. Data abstraction and hierarchy. *Proceedings on Object-oriented programming systems, languages and applications*, ISBN:0-89791-266-7, 1987.
- [16] The Aglets Mobile Agent Platform. Web site: <http://aglets.sourceforge.netp>.
- [17] The AspectJ Project. Web site: <http://www.eclipse.org/aspectj/index.php>.
- [18] M. Nishizawa S. Chiba S. An easy-to-use toolkit for efficient java bytecode translators. *Proc. of 2nd Int'l Conf. on Generative Programming and Component Engineering (GPCE '03)*, LNCS 2830, pages 364–376, 2003.
- [19] H. Zhu. Role-based systems are autonomic. *7th IEEE International Conference on Cognitive Informatics*, pages 144–152, August 2008.