Roles in Building Web Applications using Java

Guido Boella
Dipartimento di Informatica
Università di Torino
Italy
+390116706711
guido@di.unito.it

Andrea Cerisara
Dipartimento di Informatica
Università di Torino
Italy
+390116706711
andreacerisara@gmail.com

Roberto Grenna
Dipartimento di Informatica
Università di Torino
Italy
+390116706711
grenna@di.unito.it

ABSTRACT
In this paper we apply the powerJava model of roles and relationships to a web application programming environment. First we show how the notion of role, as defined in powerJava, combines and automates several aspects which are important in web application programming, and which are now unrelated and dealt with separately and mostly by hand, and thus prone to errors. Second we show how from the powerJava code a web application can be automatically constructed using Struts and Spring.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *abstract data types, polymorphism, control structures*.

General Terms
Design, Languages.

Keywords
powerJava, roles, organizations, web, sessions.

1. INTRODUCTION
Programming a web application implies, among other things, to cope with several aspects which are loosely connected in current web programming frameworks and mostly coded by hand rather than by means of suitable programming abstractions. We list them with reference to the model of web applications built with the framework of Java servlets and JSP extended with Struts:

1. Different types of users (role types in the following), have different or partially overlapping sets of methods that they can invoke from the client interface. E.g., the operation for sending an order to a market done by the market administrator or a market customer. Methods are invoked by the user by actions corresponding to clicking links, pressing buttons, etc., on the client web interface. E.g., administrator, client, etc.
2. The same methods of an application can be invoked, in some cases, by different users playing different role types, possibly with different meanings depending on the role type of the caller. E.g., the sendOrder can have a different behavior for the administrator or a customer.
3. A user logged in a role type should not be authorized to invoke methods which are not associated to its role (invocation either due to bad interface programming or malevolence of the user). Consider for example, a RBAC style system of authorization (Role based access control) [7].
4. In the client web interface often the possible actions at disposal for the user must be reported, depending on the role type in which the user logged in the system. However, possible actions are often represented as methods of unrelated classes of the application, and the correspondence user-method must be made by hand.
5. Communication between client and server is sessionless in the HTTP protocol, thus, the notion of session is introduced indirectly using cookies or URL rewriting. However, the abstractions for sessions are modelled as hashtables of `Objects`, whose values must be casted to the right type by the programmer after being retrieved by using the name of the variable. This prevents type checking on the use of the session. Alternatively, it is necessary to add to the session a complex object with the relative get and set methods for each variable.
6. Interdependencies like variables shared between the methods associated to actions performed by the same user must be expressed using the session hashtable, since, often, like in Struts, a new instance of the class of the invoked method is made each time.
7. Interdependencies like shared variables between the methods associated to actions performed by different users are difficult to capture, since the session is associated to one user only. If, instead, the methods are all invoked on the same object shared between different user sessions, then the requirement (item 4) cannot be met, since they would have a behavior not dependent on the caller.
8. Part of the session must be often made persistent on a database when the user logs out. However, which parts are relevant must be selected and type casted by hand before transferring the values to the database.
9. Coordination between different role instances belonging to a relationship must be coded by hand like also in applications not based on the web. E.g., in an e-

```
class Printer {
  private int printedTotal;
  private void print(){...}

  definerole User {

    private int printed;

    public void print(){ ...
      printed = printed + pages;
      Printer.print(that.getName());
      }}}
```

```
role User playedby UserReq
   { void print();
     int getPrinted(); }

interface UserReq
   { String getName();
     String getLogin();}

jack = new AuthPerson();
laser1 = new Printer();
laser1.new User(jack);
laser1.new SuperUser(jack);
((laser1.User)jack).print();
```

Figure 1. A role *User* inside a *Printer*.

learning website associating to each user in a role of student of a course another user in the role of teacher of that course. The relationship has a specific state and behavior.

10. Type checking in web application programming is very loose due to points 4, 6, 8, above.

Many of these issues should be addressed by means of suitable abstractions which allow programming web applications in a more structured way and to keep under control several related aspects which are now coded separately. In this paper we propose to use in a web application development framework the abstraction of role as it has been defined in the object-oriented programming language powerJava to cope with the above issues in a unified way. In brief, roles are modelled as a sort of inner classes of Java expressing different ways of interactions with the outer class. In the underlying metaphor, the inner classes describe role types and the outer class the institution the roles belong to, in this scenario, the web application. The possibility of playing a role is conditioned to the conformance to some requirements, expressed as an interface.

A user from a web client, at the moment he logs on the web application, will be associated to an instance of some role type, associated with the instance of the outer class representing the web application. Each time the user calls a methods, it will be invoked on the same instance. Since roles, as instances of classes, have a state and a behavior, they constitute the link between the notion of session and the notion of behavior specific for a type of user. Moreover, since they specify the behavior as a set of methods inside the class, the list of method names can be used as a specification of the list of actions at disposal of the user to be shown in the client interface.

The role model of powerJava is typed since it is translated in pure Java. Roles allow simplifying the above problems also by extending to them the typing too, in different degrees. The methodology we use is to exploit the potentiality of combining Struts and Spring for developing web applications, using the powerJava code not only to generate the Java code but also the information to configure suitably Struts and Spring. It is outside the scope of this paper to extend the model to JSP code producing the web pages in the Model-View-Controller framework of Struts, to deal with arguments of methods in Struts, and to describe the extended powerJava precompiler in detail. The paper is structured as following. First we briefly summarize the powerJava language (Section 2), then we describe a running example for web applications (Section 3); in Section 4 we show how the proposed framework can be used in web applications providing services, like in B2B applications, while Section 5 concludes the paper.

2. ROLES IN POWERJAVA

Baldoni *et al.* [1] introduce roles as affordances in powerJava, an extension of the object oriented programming language Java. Java is extended with:

1. A construct defining the role with its name, the requirements and the operations (called powers).
2. The implementation of a role, inside an object and according to its definition.
3. How an object can play a role and invoke the operations of the role.

Figure 1 shows the use of roles in powerJava. First of all, a role is specified as a sort of interface (`role` - right column) by indicating with an interface or class who can play the role (`playedby`) and which are the operations acquired by playing the role. Second (left column), a role is implemented inside an object as a sort of inner class which realizes the role specification (`definerole`).

```
interface AdminReq extends Certified
{
    String getCredentials();
    String getBankAccountNumber();
}

interface CustomerReq extends Certified
{
    String getCredentials();
    String getAccountNumber();
}

role Administrator playedby AdminReq
{
    String sendOrder();
    void addProduct(Product pr);
}

role Customer playedby CustomerReq
{
    String sendOrder();
}
```

Figure 2. This figure shows how the requirements and roles are declared.

The inner class implements all the methods required by the role specification as it were an interface. Each `definerole` will be written by the precompiler as a class extending `RoleInstance` from which the methods for managing roles are inherited. So, the `definerole User` become a `class User extends RoleInstance`.

In the bottom part of the right column of Figure 1 the use of powerJava is depicted. First, the candidate player `jack` of the role is created. It implements the requirements of the roles (`AuthPerson` implements `UserReq` and `SuperUserReq`).

```
public class Market extends AbstractInstitution
{
    private int numberOfOrders;
    private int numberOfProducts;
    private Product[] products = new Product[1000];

    definerole Administrator
    {
        private int privateOrders;
        private ProductInCart[] cart = new ProductInCart[100];
        private int prodsInCart;
        public Administrator(Player that)
        {  super(that);
        }
        private String getCredentials()
        {  return ((Certified) that).getCredentials();
        }
        @RoleMethod(jsp="market", description="send an order to the market")
        public String sendOrder()
        {  if (check(this.getCredentials()))
            {   private double totalCost;
                totalCost = 0.0;
                for (int i=0; i<prodsInCart; i++)
                    {   totalCost += cart[i].costPrice * cart[i].quantity;
                    }
                numberOfOrders++;
                privateOrders++;
                prodsInCart = 0;
                ...
                return ActionSupport.SUCCESS;
            }
            else return ActionSupport.FAIL;
        }
        @RoleMethod(jsp="product", description="add a product to the market")
        public void addProduct(Product pr)
        {   numberOfProducts++;
            products[numberOfProducts] = pr;
        }
    }
    definerole Customer
    {
        private int privateOrders;
        private ProductInCart[] cart = new ProductInCart[100];
        private int prodsInCart;
        ...
        @RoleMethod(jsp="market", description="send an order to the market")
        public String sendOrder()
        {
            if (check(this.getCredentials()))
            {
                private double totalCost;
                totalCost = 0.0;
                for (int i=0; i<prodsInCart; i++)
                    {
                        totalCost += cart[i].sellPrice * cart[i].quantity;
                }
                numberOfOrders++;
                privateOrders++;
                prodsInCart = 0;
                ...
                return ActionSupport.SUCCESS;
            }
            else return ActionSupport.FAIL;
        }
    }
}}
```

Figure 3. The code for a `Market` institution is shown. We can identify two role types: `Administrator`, and `Customer`, which have both the `sendOrder` method, but with a different implementation. The annotation `@RoleMethod(...)` is used by the precompiler for generating the XML code in Figure 4, in particular, to associate with the action sendOrder the JSP which presents the results of the action.

```
<action name="sendOrder" class="market"
   method="sendOrder">
    <result>/jsp/market.jsp</result>
</action>
```

Figure 4. Example of a Struts configuration. This means that when an `http://server:port/context/sendOrder.action` call is done, an object of a class labeled `market` (the label is mapped to the id property of a `bean` tag of Spring, see Figure 5) is created by Spring, then it's invoked on it the method `sendOrder` by Struts; the obtained instance is used to realize the mappings in `market.jsp`. Spring remains transparent to Struts in our roles management.

Before the player can play the role, however, an instance of the object hosting the role must be created first (a `Printer laser1`). Once the `Printer` is created, the player `jack` can become a `User` too. Note that the User is created inside the `Printer laser1 (laser1.new User(jack))` and that the player `jack` is an argument of the constructor of role `User` of type `UserReq`. Moreover `jack` plays the role of `SuperUser`. The player `jack` to act as a `User` must be first classified as a `User` by means of a so-called role casting `((laser1.User) jack)`. Note that `jack` is not classified as a generic `User` but as a `User` of `Printer laser1`. Once `jack` is casted to its `User` role, it can exercise its powers, in this example, printing (`print()`). Such method is called a power since, in contrast with usual methods, it can access the state of other objects: namespace shares the one of the object defining the role. In the example, the method `print()` can access the private state of the `Printer` and invoke `Printer.print()`.

3. ROLES FOR WEB APPLICATIONS

Consider as a running example an electronic market where different users can sell and buy goods. We model two kinds of types of users by means of two role types: `Administrator` and `Customer`, see Figure 2. The two roles can have different methods, e.g., only the `Administrator` can add new products (see Figure 3).

Moreover, both roles have the `sendOrder` method: however, the `sendOrder` can be implemented in different ways in the two role implementations. The two roles requires different methods to be played, represented in the two requirement interfaces. In Figure 3 we represent the `Market` institution, containing the implementation of the two roles with state and behavior (the implementation of the powers). An `Administrator` buys products at the cost price, while the Customer has to pay the sell price. Note that both roles have instance variables which persists during the session (e.g., `privateOrders`). Moreover, due to the visibility rules of Java, the roles of the methods can access the private variables of the outer class instance (`numberOfOrders`), thus allowing the coordination of the whole web application. Finally, in the role it is possible to invoke methods of the player via the powerJava that variable, referring to a subclass of `Player` satisfying the requirements of the role (see Section 4 and Figure 2). At runtime we want to make possible the following behavior:

1. We program the web application in powerJava, designing the behavior of each user type as a role class of the web application.

2. We program the JSP code, completing the view of the Model- View-Controller pattern [6] of Struts, which is related to each method of the roles in the institution class.

3. At runtime, when a user logins from a client to the web application it is assigned a role instance of the type depending from his identity (a new one the first time, or the previous one when he returns to the website).

4. Each get or post message sent from the client is mapped to the method of the role played by the user and executed in the state of the role instance.

5. The JSP code producing the web page gets the information directly from the role instance, together with information about the possible interaction possibilities contained in the role type class.

6. When the user logs out, the role instance is made persistent and associated with the credentials of the user.

```
<bean scope="prototype" id="market"
          factory-bean="marketRoleLocator" factory-method="instance" />

<bean scope="singleton" id="marketInstitution" class="Market" />

<bean scope="session" id="marketRoleLocator" class="RoleLocator">
    <constructor-arg><ref bean="marketInstitution" /></constructor-arg>
</bean>
```

Figure 5. `marketInstitution` is the institution defining roles as inner classes (it's defined as singleton); the market present in the Struts configuration file is mapped on an instance of RoleLocator (marketRoleLocator), which persists along the user session. More specifically the method instance() is invoked. The institution, a single instance across all the sessions, is passed in the constructor by Spring.

```
public class RoleLocator {
    private Player player;
    private Institution institution;

    public RoleLocator(Institution institution)
    {
        this.institution = institution;
    }

    public Object instance()
    {
        RoleContext context = player.activeRole().lookupContext(institution);

        return context.instance();
    }

    public void setPlayer(Player player)
    {
        this.player = player;
    }
}
```

Figure 6. The `RoleLocator` is the object which dispatches the Struts calls to the role instance contained in the role player.

To achieve this scenario we must first resolve some problems, in particular, the management of the association of a role instance with a user. As explained in the Introduction, Struts creates an instance of the class the invoked method belongs to at each invocation by the user. Instead, we want that the role instance is created only at the beginning of the interaction and that it becomes the medium of the interaction with the web application in a transparent way. The methodology we adopt is to rely on the Java servlet infrastructure extended with Struts and Spring. The programmer uses powerJava, programming the web application in terms of roles, all part of the same institution (the web application). In particular:

1. The powerJava source file modelling the web application is compiled with the powerJava precompiler into Java code and some support classes are added.

2. At the same time, the precompiler creates the XML files for configuring Struts and Spring, associating the methods with the JSP pages.

Struts and Spring are used in combination with a standard plugin to manage the invocation of a method from the client. Consider the code in Figure 4: it specifies that when Struts receives a message `sendOrder` it must invoke the method `sendOrder` of the class labeled as `market`. The problem is that Struts maps the action specified in the get or post message into a method of a class. In Struts the class is instantiated and the method invoked each time, independently from the identity of user. All the communication among methods can be done only via the session object.

We exploit Struts to intercept the get or post messages and to map it to a method of a class as usual. However, the role instance should associated to an outer class instance (the web application), and the method specified in the action field of get or post must be invoked on the role instance and not on the web application instance. Moreover, Struts does not know that both instances remain the same during the interaction of the user with the web application. Two issues are involved: first the persistence of objects along the interaction with the user, second the method invocation must be dispatched to the role instance and not to the institution instance. Thanks to the combination with Spring, via a plugin, the lifecycle of the specified class can be manipulated, and the invoked method is dispatched to the role. In the example above the class labeled as market is embedded in Spring, and Struts passes the control to Spring when it receives the message `http://server:port/context/sendOrder.action` to create an instance of the bean `market`.

Spring has been developed to help programmers to deal with the so called inversion of control principle. For this reason Spring manages object lifecycles, dependencies in the constructors and factories for creating objects or returning existing ones after some logic is performed. We exploit this capability of Spring in our setting. We use Spring in order to intercept the method calls of Struts, and elaborate them, instead of calling a method on a new instance of a class.

First, a factory is associated to the outer class, so that when

```
if (!persisted(player))
    {
       Class<?> rinst = Class.forName(roleContextName(player));
       Constructor<?> rconst = rinst.getConstructor(getClass(), Player.class);
       return new RoleContext(this, (RoleInstance) rconst.newInstance(this, player));
    }
    else
    {
        RoleContext result = persister().load(roleContextName(player), this);
        result.instance().setPlayer(player);

       return result;
```

Figure 7. A new role instance is created only in case that the `RoleLocator` is not able to retrieve a role instance of the right type. The search is done in the player, or in the persistence mechanism.

```
public List<RoleAction> getActions(){
    Method[] methods = getClass().getMethods();
    List<RoleAction> result = new ArrayList<RoleAction>();
    for (Method method : methods)
    {
        if (method.isAnnotationPresent(RoleMethod.class))
        {
            RoleMethod annotation =
                (RoleMethod) method.getAnnotation(RoleMethod.class);
            result.add(new RoleAction(method.getName(), annotation.description()));
        }
    }
    return result;}
```

Figure 8. In `RoleInstance` is defined `getActions()`, which returns the whole list of the possible powers offered by the role.

a method is first called an instance of the outer class is created, an instance of the role type of the user is created and associated with the user in the session. In Figure 5, the configuration file of Spring is illustrated: the request of creating a market instance from Struts is not mapped on a `Market` instance, but on the bean `marketRoleLocator`. `marketRoleLocator` is associated with a factory `RoleLocator`, which specifies the creation logic of roles and institutions. The institution `Market` is created by Spring only once (see the scope `singleton`), while an instance of `RoleLocator` is created each time a new user logins (see the scope session) and the unique `Market` instance is passed to its constructor. In this way, all role instances created by `RoleLocator` are linked to the same outer class instance `Market`. After the `RoleLocator` instance is created the first time, then for each user message the method `instance` is invoked. In this way, when a method is invoked on the institution class (the web application), Spring via the `RoleLocator` returns the role instance rather than the institution instance and Struts invokes the method on this role instance, thus adapting the method to the user. The role instance is found in the session according to the role type the user logged into.

This behavior is described in Figure 6. While the constructor of `RoleLocator` only assigns to its state the institution instance, the `instance` method looks up the right role instance related to the user and returns it. Since the player of a role is not part of the system, but a client communicating with it, it is represented by an instance of the class `Player` or of a refinement of it in case the player has to satisfy some requirements (see Section 4).

With the player are associated all the information about all the roles played by it in which institution, and, in particular, the active role, to which the method invocations are dispatched (this opens the possibility for a player to switch to one role from the other in the same session). The `Player` instance represents the user session in a structured way:

```
HttpSession session =
   ServletActionContext.getRequest().getSes
sion();
session.putValue("USER", player);
```

The role instances associated with the player represent the state of the interaction with the web application. The role instance is first created when the `RoleLocator` associated at the session level is not able to retrieve a role instance of the right type in the player or in the persistence mechanism maintaining previously played roles (see Figure 7).

`RoleContext` and `RoleInstance` are wrappers around the proper instance of a role type inner class of an institution. In particular, `RoleContext` maintains the different instances of the roles an agent can play in the different institutions on a server. The same role instance is passed to the JSP code as result of the work of Struts. This is of particular importance, since the JSP code is likely to use or print some of the variables or to

```
<%@ taglib prefix="s" uri="/struts-tags" %>

 <html>
    <body>
        <h2>User <s:property value="login" /></h2>
        <h2>Role <s:property value="role" /></h2>
        <h2>Number of orders of the store <s:property value="numberOfOrders" /></h2>
        <h2>Number of own orders <s:property value="privateOrders" /></h2>
        <h2>Number of products in the cart <s:property value="prodsInCart" /></h2>
        <a href="<s:url action="send" />">Send a new order</a><br>

        <s:iterator id="action" value="actions">
          <s:set name="description" value="${action.description}" />
          <a href="<s:url action="${action.url}" />"><s:property value="description" /></a><br>
        </s:iterator>

        <a href="<s:url action="logout" />">logout</a>
    </body>
</html>
```

Figure 9. The JSP code returning the HTML page after the execution of `sendOrder` method. The variables in the Struts tags refer to the role instance returned by Spring.

invoke methods of the role instance.

For example the Struts code

```
<s:property value= "privateOrders">
```

in the JSP code of Figure 9 returns the value of the privateOrders variable in a role instance or, if it is not defined in the role class, the value of the corresponding variable in the outer class (the web application). In the JSP code, referring to the role instance via the session is particularly clumsy, while the Struts and Spring combination offers an elegant solution, returning the role instance whose identification is delegated to the factory in Spring. This allows expressing the principle that there is no direct interaction with an institution, but it is always mediated via a role (see item 3 in Section 1): thus the same method can have different meanings depending on the role it is dispatched to and different roles can have different methods.

Moreover, since an invocation of a method is redispatched to a role instance, this role instance works as a structured session: the variables of the role instance represent the state of the interaction between client and application in a structured way including type checking, thus avoiding the perils of traditional session handling via hashtables and casting (item 5). At the same time the inner-outer class visibility rules in Java allows a role to have access both to the private variables of the outer class and of the sibling roles, thus allowing the coordination among different users without resorting to sessions or global variables (items 6, 7).

A further advantage of representing sessions as role instances is in case it is necessary to make them persistent after the logout of the user or a reboot of the server: persistency mechanisms of Java can be used to store the role instance on a database without further elaboration (item 8). Since the actions which can be invoked by the user are all included in role class, it is possible by reflection to have at disposal the list of all possible actions of the user (item 4). In Figure 8 it is represented the code which from the role type extracts the methods which should become actions of the client web interface. The `getActions` methods exploits the `@RoleMethod` annotation in the role types (see Figure 3). This information is used both to generate menus in the html pages generated by JSP programs and in the exception handling: before invoking the method specified in a get or post message on a role, its existence can be verified. Thus the fact that a user is authorized to invoke a method is modelled by the membership of the method to the role class, without having to add further mechanisms like annotations, which are, instead, used to automatically create the XML files for Struts configuration (item 1, see Figures 4 and 9).

4. FROM USERS TO SERVICES

In case of a web application for a user it is not apparent the interaction between the role and its player, one of the basic characteristics of the powerJava definition of roles.

Instead, in case of web applications interacting with each other by providing services, the two way interaction between player and role can be exploited. E.g. consider the case of a web application like the market example above which needs further credentials to perform an action requested by the client service. Before sending a reply back to the requesting server, the requested method during its execution invokes on the player a `getCredentials` method.

The interaction between the player and role is modelled by invoking methods from the client interface and the retrieval of the role instance from the `Player` instance made by `RoleLocator` invoked by Spring. In powerJava this was made by means of role cast before role method invocation.

In powerJava the interaction between the role and the player is done via method invocation on the value of the special variable that which is initialized at the creation of the role instance with a reference to the player objects (see Figure 3). The methods which can be invoked are specified by an interface which must be implemented by the player: the requirements needed to play a given role.

In case of web services, there is obviously no interface which can be implemented by the player, which is a server providing services on another machine. The requirements correspond to the services which can be invoked on that server. However, since in the role the services must be invoked and the services are expressed as methods, the requirements interface of powerJava has a double role: as a declarative specification of the services which should be provided by the player server, and for type checking the requirement method calls inside a role (see Figures 2, 3).

This method cannot be obviously remotely invoked on the player server: it must be rather translated in the invocation of a service. This can be done via a stub which for each method sends to the player server a message requesting the service and returns the reply of the server.

```
public class CertClient extends Player
  implements Certified {
    private Certified stub;
    public CertClient(Certified stub)
    {
        this.stub = stub;
    }
    public String getCredentials()
    {
        return stub.getCredentials();
    }
}
```

Figure 10. A possible implementation for the CertClient, which is a Certified Player.

The stub has to implement the requirement interface to allow the type checking to verify the requirement invocation in the role class (see Figure 10). However, the stub can be created automatically by the powerJava precompiler.

The player, once playing the desired role, can use its powers to operate with the organization. In our example (see Figure 3), the player for the `Administrator` role is able to send an order, and to add a new product to the market (using `sendOrder` and `addProduct` powers), while the player for the `Customer` role can only send an order (by means of the `sendOrder` power), but its sending operation is quite different from the `Administrator`'s one.

5. CONCLUSIONS

In this paper we show our framework for introducing the powerJava mechanism for roles and organizations into web applications. In our vision, roles and organizations in web applications can solve a number of issues that developers have to take in account when programming their pages, like keeping the state of the interaction between users and web site, or giving different behaviors to the actions of different types of user.

The same methodology has been used to introduce roles in distributed systems based on the multi-agent methodology: a framework Jade-based [4], named powerJade, has been implemented [3], following a similar approach to the one used for this work. powerJade offers classes to implement roles, players, and organizations as agents in Multi Agent Systems, considering roles as sets of behaviours that agents playing them have to assume. Players interact with organizations by means of roles; all the communication issues are solved using messages, which allows interactions between agents active on different platforms. There is a precise protocol of messages which agents have to follow to play a role inside an organization.

First, the candidate player has to initiate an enactment protocol (see [5]), via messages, with the organization offering the desired role. The organization, if it considers the agent authorized to play the role, returns to the candidate player a list of specifications about the powers and requirements of the requested role; otherwise, it denies to the player to play the role. The player, if considered reliable, decides whether to respond to the organization that it can play the role (and this results in the creation of a new role instance, associated to the player) or not (and the protocol ends without any role instance creation).

Also the interaction between a player and a role instance associated to it is regulated by some protocols: the request (from the role to the player) for the execution of a requirement, the invocation of a power by the player, and the request of the role to invoke a power.

In this context, we have to remember that only the player is an autonomous agent, while the role and the organization aren't. This is the reason for which, in the first considered protocol, a request for a requirement is done by the role (normally after a call for a power done by the player), but the player can also decide of not to execute it, for any reason.

Future works is introducing relationships in web applications by means of roles, like it is done in object oriented languages [2]. Moreover, in this paper we do not address yet the link between arguments of methods and values passed by clients via forms, even if Struts seems again a promising framework, since it automatically deal with methods for dealing such values and initializing variables.

6. REFERENCES

[1] M. Baldoni, G. Boella and L. van der Torre, "Interaction between Objects in powerJava", 2007, pages 7-12, volume 6, number 2, Journal of Object Technology

[2] M. Baldoni, G. Boella and L. van der Torre, " The Interplay between Relationships, Roles and Objects", Proceedings of FSEN'09. LNCS Springer, 2009

[3] M. Baldoni, G. Boella, V. Genovese, R. Grenna, and Leendert van der Torre, "How to Program Organizations and Roles in the JADE Framework" pages 25-36, Proceedings of MATES'08, LNCS Springer 2008.

[4] F. Bellifemine, A. Poggi, G. Rimassa, "Developing multi-agent systems with a FIPA-compliant agent framework", Software - Practice And Experience. 31(2) pages 103–128. 2008.

[5] M. Dastani, B. van Riemsdijk, J. Hulstijn, F. Dignum, and J.-J. Meyer. "Enacting and deacting roles in agent programming", 2004, pages 189-204 Proceedings of AOSE'04.

[6] T. Reenskaug, "The Model-View-Controller (MVC) - Its Past and Present" 2003, JAOO, Arhus

[7] R. Sandhu, E. Coyne, H. Feinstein and C. Youman,"Role-Based Access Control Models", 1996, pages 38-47, volume 2, IEEE Computer