Roles and Coordination in powerJava

M. Baldoni, G. Boella

Dipartimento di Informatica, Università degli Studi di Torino (Italy)

and **L. van der Torre**

SEN3 - CWI Amsterdam and TU Delft (The Netherlands)

Abstract

In this poster we apply the **role metaphor** to coordination.
Roles are used in sociology as a way for structuring organizations and for coordinating their behavior. In our model, the distinguishing features of roles are their **dependence on an institution**, and the **powers** they assign to **the players of roles**. The institution represents an **environment** where the different components interact with each other by using the powers attributed to them by the roles they play, even when they do not know each other. The interaction between a component playing a role and the role is performed via precise interfaces stating the **requirements** to play a role, and which **powers** are attributed by roles. Roles encapsulate their players' capabilities to interact with the institution and with the other roles, thus achieving separation of concerns between computation and coordination.
The institution acts as a coordinator which manages the interactions among components by acting on the roles they play, thus achieving a form of exogenous coordination. As an example, we introduce the role construct in the **Java programming language**, providing a **precompiler** for it.

2. Introducing roles in Java

Our proposal is to define the role construct as a sort of **double face interface** which allows the connection of a player to an institution. The interface is double in that it specifies:

1. The methods required to a class playing the role (**requirements**). In order to play a role, a class must offer some methods. These are specified in the role by an interface.

2. The **methods offered** to objects playing the role (**powers**). An object of a class, offering the required methods, plays the role: it is empowered with new methods as specified by this part of the role definition. These methods are powers since roles are implemented in Java as **inner classes of an institution** (another class). Since methods in inner classes see the private fields of the outer class containing them and of other inner classes, the methods are able to **access the private state** of the institution and of the sibling roles.

This double face pervades the life of a role: **first**, a role is **defined** with its requirements and powers, **second** its powers are **implemented** in an inner class of the institution, which connects a role with a player satisfying its requirements, and, **third**, the inner class implementing the role is **instantiated** passing to the constructor an instance of an object satisfying the requirements as well as an instance of the institution.

Requirements of a role in Java correspond to the notion of interface, specifying which methods must be defined in a class playing the role. As for interfaces, this mechanism of partial descriptions allows the **polymorphism** necessary for a role to be played by different classes.

1. Properties of roles

Our notion of roles is based on the **ontological analysis** of roles by Boella & van der Torre 2004. The **defining properties** of roles are:

1. Foundation: an **instance of a role** must always be associated with an **instance of the institution** it belongs to besides being associated with an **instance of its player.**

2. Definitional dependence: The **definition of the role** is given **inside the definition of the institution** it belongs to.

3. Institutional empowerment: the **actions defined for the role** in the definition of the institution **have access to the state and actions** of the institution and to the other roles' state and actions: they are powers.

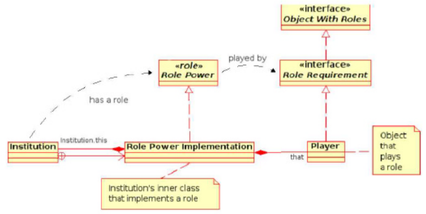
Moreover, following Steimann 2000, roles can be **played by different kinds of actors.** For example, the role of customer can be played by instances both of person and of organization, i.e., two classes which do not have a common superclass.

3. Roles and coordination

It is possible to draw a comparison between roles and the **IWIM** (Idealized Worker Idealized Manager, Papadopoulos & Arbab 1998) model. **Components playing roles** are the **workers** carrying out the computation, while the **institutions** are the **managers** coordinating the other processes. Institutions (managers) do not directly manipulate components (workers), but they **coordinate them through the roles** they play, which represent the state of the components inside the institution. Symmetrically, the components do not interact with other components, but they **interact** with the institution and with the other roles **only through the powers** offered by the roles they play.

Roles give to their players the powers to interact with the other roles and the institutions. Powers, which are modelled as methods, are defined by the institution, so components are not required to know their implementation, while they can be invoked on them when acting in a role. By means of roles it is possible to **connect the output of a component to the input** of another component without requiring them to be aware of the connection, and to encapsulate the modalities of this connections, like concurrency synchronization. In this way we achieve the **separation of concerns**: **components** which act as the primary **unit of computation** in a system, and **institutions** which **specify interaction** and **communication patterns** by means of roles.

Since **powers** are methods inside and defined by the institution they have the possibility to **access both the institution and the other roles** in it: hence, they can also **reconfigure** the interaction between the componentsplaying the role. In the same way, the institution itself can modify these interconnections, thus achieving an **exogenous coordination** of the components composing the system.



A UML diagram of the notion of role.

4. powerJava syntax

```
roledef ::= "role" identifier ["extends" identifier*]
  "playedby" identifier interfacebody

roleimplementation ::=
  [public | private | ...] [static] "class" identifier
  ["realizes" identifier] ["extends" identifier]
  ["implements" identifier*]  classbody

keyword ::= that | ...

rcast ::= (expr.identifier) expr
```

5. Role Casting

Role's methods can be **invoked** from their players, given that **the player is seen in its role**. To do this, we use the Java idea of casting with a difference: the object is not casted to a type. **Role casting** is done in powerJava by **casting the player of the role** to the **role implementation** we want to refer to: e.g.: **((school.StudentImpl) chris). getName()**

Type casting in Java allows to see the same object under different perspectives while maintaining the same structure and state. In contrast, **role casting views an object as having a different, even if related, state and and different behaviors**. This is because, it conceals a **delegation** mechanism: the player instance hiddenly delegates the **role instance** the execution of the method. The delegated object can access the state of the institution via its powers.

6. References

M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct. In: Procs. of MTCoord'05, 2005.
Boella, G., van der Torre, L.: Organizations as socially constructed agents in the agent oriented paradigm. In: Procs. of ESAW'04, 2004.
Papadopoulos, G. and F. Arbab, Coordination models and languages, Advances in Computers 46, 1998.
Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data and Knowledge Engineering 35, 2000.

```
interface StudentReq // Student's requirements
  {  String getName(); }

role Student playedby StudentReq  // Student's powers
  {  String getName();
     void takeExam(int examCode, HomeWork hwk);  }

class School { // institution defining the StudentImpl role
    private int [ ][ ] marks;
    private String schoolName;

  class StudentImpl realizes Student { // Role as inner class
      private int studentID;
      public  void  takeExam(int  examCode;  HomeWork  hwk)
          {  marks[studentID][examCode]=evalHomeWork(hwk); }
      public String getName()
          {  /* "that" is the player of the role
             schoolName a private variable of the institution */
             return that.getName()+",student at "+ schoolName;  }
  }
}

// a possible player of the role
class Person implements StudentReq {
    private String name;
    String getName() {  return name; }
}

// An example main
public static void main(String[] args) {
   Person  chris = new Person("Christine");
   School  harvard = new School("Harvard");
   harvard.new StudentImpl(chris); // chris plays the role
   String x=((harvard.StudentImpl) chris).getName(); // Role cast
    ((harvard.StudentImpl) chris).takeExam(...,...); // Role cast
  }
```

Translation in Java

```
interface StudentReq // Student's requirements
  {  String getName();  }

interface Student { // Student role definition
   String getName();
   void takeExam(int examCode, HomeWork hwk); }

class School {
   private int [ ] [ ] marks;
   private String schoolName;

  class StudentImpl implements Student { // Role as inner class
     StudentReq that; // Added by JavaRoleParser
     public StudentImpl (StudentReq that) { // Binding of player
       this.that = that;  // Added by JavaRoleParser
       ((ObjectWithRoles)this.that).setRole(this, School.this);  }
     // Role's fields and methods ...
  }

interface ObjectWithRoles { // Objects which can play roles
   public void setRole(Object pwr, Object inst);
   public Object getRole(Object inst, String pwr); }

class Person implements StudentReq, ObjectWithRoles {
   private java.util.Hashtable roleslist=new java.util.Hashtable();
   public void setRole(Object pwr, Object inst) {
      roleslist.put(inst.hashCode() +
              pwr.getClass().getName(), pwr);  }
   public Object getRole(Object inst, String pwr) {
      return roleslist.get(inst.hashCode() +
                  inst.getClass().getName() +  "$" + pwr);  }
  ... } // Player's fields and methods

public static void main(String[] args) {
   Person chris = new Person("Christine");
   School harvard = new School("Harvard");
   harvard.new StudentImpl(chris);
   String x = ((School.StudentImpl) // Translation of role cast
              chris.getRole(harvard, "StudentImpl")).getName());
   ((School.StudentImpl)
        chris.getRole(harvard, "StudentImpl")).takeExam(...,...);  }
```