An implemetation of roles as affordances: powerJava

Erik Arnaudo, Matteo Baldoni, Guido Boella, Valerio Genovese, and Roberto Grenna

Abstract. **This document shortly describes powerJava, a Java extension which provides the instructions to manage roles. After defined the environment in which we have worked, we will discuss the language's new instructions and we will show an example.**

I. Something about roles

Object orientation is a leading paradigm in programming languages, modeling, knowledge representation and databases. When we think to an *object*, we do it in terms of *attributes* and *methods*, and if we refer to *object interaction*, we do it in terms of *public attributes* and *public methods*: these are the only ways to realize it!

In computer science literature, other kinds of interaction between entities have been proposed at levels higher than programming languages. We can properly speak about *sessions*, which contain the interaction state (as at web services level), but a very important concept is that of *role*. Steimann [1] provided an interesting role representation, giving three types of "role as": *roles as named places*, *role as specialization and/or generalization*, *roles as adjunct instances*. Our approach is to consider *roles as affordances*; we consider roles as instances having a different identity respect to the players that play them. Inspired by research in cognitive science, this view sees the properties (attributes and operations) of an object as something not independent from whom is interacting with it. In this way, an object "affords" different ways of interaction to different kinds of objects.

The notion of "affordance" has been made popular by Norman [3] (p. 9): "The term affordance refers to the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used. A chair affords ('is for') support, and, therefore, affords sitting."

How can we use the concept of "affordance" to introduce new modeling concepts in object oriented knowledge representation? The affordances of an object are not isolated, but they are associated with a given specie. So we have to consider sets of affordances. We will call a *role type* the different sets of interaction possibilities, the affordances of an object, which depend on the class of the interactant manipulating the object: the *player* of the role. To manipulate an object it is necessary to specify the role in which the interaction is made.

A given role type can be instantiated, depending on a certain player of a role (which must have the required properties), and the *role instance* represents the state of the interaction with that role player. Just to better explain the possible use of roles as affordances, we introduce the following figures, which introduce different ways to interact with an object through the roles it offers.

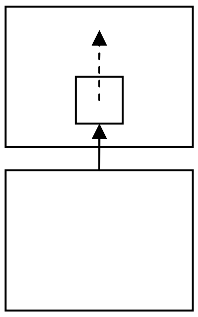

Figure 1 - An object interacts with another one by means of the role offered by it.

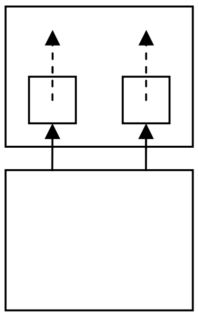

Figure 2 - An object interacting in two different roles with another

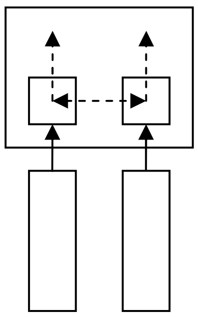

Figure 3 - Two objects which interact with each other by means of the roles of another object.

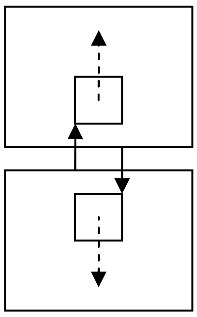

Figure 4 - Two objects interact with each other, each one playing a role offered by the other.

The idea behind affordances is that the interaction with an object does not happens directly with it by accessing its public attributes and invoking its public operations. Rather, the interaction with an object happens via a role: to invoke an operation, it is necessary first to be the player of a role offered by the object the operation belongs to. The roles which can be played depend on the properties of the role player (the requirements), since the roles represent the set of affordances offered by the object.

II. POWERJAVA AND ITS ENVIRONMENT

In order to translate the roles as affordances approach, we realize *powerJava*.

What we did, simply was an extension of Java 1.4 grammar, implemented using *JavaCC* (Java Compiler Compiler), which is a parser generator that returns in output Java code.

This parser generator is a program which reads a grammar specification and converts it in a Java program (parser) that can recognize this grammar.

JavaCC offers other tools too. For example, *JJTree*, that realizes (before obtaining the parser) a tree from the grammar written by the user. By JJTree exists the possibility of inserting code (written in Java, obviously), to “drive” the behavior of the parser in some cases. It defines the *Node* interface too. The Node interface is the interface that each node of the tree has to implement, and it offers some methods for setting the node parent or its sons.

JavaCC is a top-down parser, but JJTree generates the tree in a bottom-up mode, using a stack to contain the nodes after creating them.

The idea is that the tokens are serialized in a chain, but from each of them is possible to reach its parents (father, grandfather and so on). In this way it’s possible to manage many events, and it’s clear that for each new token it’s possible to define the correct behavior.

The token concept (together with, for example, the *skip* concept) is something of very powerful, that collocates JavaCC at an higher level than other parsers.

JavaCC generate a LL(1) parser, but in some points of the grammar the parser can works as a LL(k) one: defining some *lookaheads* it’s possible to manage expressions made by k words. In other cases, the parser works as a LL(1), with all obvious advantages in terms of performances.

The use is very simple. The grammar files have a .jj or .jjt extension. When we start from a .jjt file, we have simply to perform:

```
C:\>JJTree filename.jjt
```

The next step is to execute:

```
C:\>JavaCC filename.jj
```

which generates all the .java files needed, including our parser. All that we have to do is to compile all the .java files.

Before continuing, we have to say that we can customize a lot of the generated classes, like SimpleNode.java, or UnparseVisitor.java. “Visitor” is the pattern that JavaCC uses for visiting the nodes.

After this short description of the environment, in *Figure 5* is shortly described the extension of the grammar syntax to obtain powerJava.

The system requirements to use powerJava environment are very simple: you’ve only to have JDK 1.4 or later on your pc.

Please note that the name powerJava is due to the fact that we call the methods offered by roles “powers”, because they offer the possibility to modify the private state and access private methods of the institution which defines them, and the state of the other roles defined in the same institution.

III. AN EXPLENATORY EXAMLPE

To make an example (see [4]), let’s suppose to model a class *Printer*. The interaction possibilities offered by the class

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

classdef ::= ["public"|"private"|...] "class" identifier
["extends" identifier] ["implements" identifier*] classbody

classbody ::= "{" fielddef* constructors* methoddef* roleimpl* "}"

roleimpl ::= “class” identifier_1 "realize" identifier_2 rolebody

rolebody ::= "{" fielddef* constructors* methoddef* "}"

rcast ::= (expr.identifier) expr

keyword ::= that | ...
```

Figure 5 - The first extension of the Java (1.4) syntax in powerJava.

are different and depend on which objects invoke its methods. For example, some objects have more privileges than other ones, and thus they can invoke methods which are not available to other objects interacting with the same printer. Moreover, some methods keep track of the interaction with each specific object invoking them. For example, print counts the number of pages printed by each object invoking it to check whether the quota assigned to the object is respected. However, objects with more privileges do not have a quota of printed pages.

The *Printer* can be seen as an institution which supplies two different roles for interacting with it (the set of methods a caller can invoke): one role of normal *User*, and the other role of *SuperUser*. The two roles offer some common methods (roles are classes) with different implementations, but they also offer other different methods to their players (and there is no direct way to interact with the *Printer*). For example, *Users* can print their jobs and the number of printable pages is limited to a given maximum; thus, the number of pages is counted (the role associates new attributes with the player): each *User* should be associated with a different state of the interaction (the role has an instance with a state which is associated with its player). The *User* can print since the implementation of its methods has access to the private methods of the *Printer* (the methods of the *User* access the private attributes and operations of another object, the institution). *SuperUsers* have the method print with the same signature, but with a different implementation: they can print any number of pages; moreover, they can reset the page counter of *Users* (a role can access the state of another role, and, thus, roles coordinate the interaction).

A role like *SuperUser* can access the state of the other *User* roles and of the callee object (the institution *Printer*) in a safe way only if it encapsulated in the institution *Printer*. Thus the definition of the role must be given by the same programmer who defines the institution (the class of the role belongs to the same institution class namespace, or, in Java terminology, it is included in it).

In order to interact as *User* or *SuperUser* it is necessary to exhibit some requested behavior. For example, in order to be a User a caller object must have an account (it must be an *Accounted*), which is printed on the pages (returned by a method offered by the player of the role). A *SuperUser* can have more demanding requirements.

Finally, a role *User* can be played only when there is an instance of *Printer* and an instance of a class implementing *Accounted* which can play the role.

In the following figure there is the code of our example.

First, we have to import package to manage roles (Figure 6-[1]), then, we define the class for the *Login* (Figure 6-[2]) and the class for the *Job* (Figure 6-[3]).

User (Figure 6-[5]) and *SuperUser* (Figure 6-[6]) are roles, both played by an *AccountedPerson* (Figure 6-[4]).

```
import it.unito.di.javarole.*;                              [1]
class Login                                                 [2]
{
        private String ID;

        public Login(String ID)
        {
                this.ID = ID;
        }
}
class Job                                                   [3]
{
        private int ID;
        private int numberOfPages;

        public Job()
        {
                this.ID = -1;
                this.numberOfPages = 1;

        }

        public Job(int ID, int n)
        {
                this.ID = ID;
                this.numberOfPages = n;
        }

        public int getID()
        {
                return this.ID;
        }

        public int getNumberPages()
        {
                return this.numberOfPages;
        }
}
interface AccountedPerson                                   [4]
{
        Login getLogin();
}
role User playedby AccountedPerson                          [5]
{
        int print(Job job);
        int getPrintedPages();
}
role SuperUser playedby AccountedPerson                     [6]
{
        int print(Job job);
        int getTotalPages();
}
class Person implements AccountedPerson                     [7]
{
        private Login login;

        public Login getLogin()
        {
                return login;
        }

        public void setLogin(String ID)
        {
                login = new Login(ID);
        }
}
```

Figure 6 – Our example's code – Part 1 of 3

Then we have to write the code for the *Person* class (Figure 6-[7]), which implements *AccountedPerson*; what we have now to do is to write our institution class: *Printer* (Figure 7-[9]). In *Printer* we write two inner classes, *U* (Figure 7-[9]) and *S* (Figure 7-[10]), which are the roles offered by it.

(Figure 8-[11]) showes the code for the main, in which we can also see the .transfer method (Figure 8-[12]). In last build of our environment, in fact, we have also implemented methods to transfer and to remove a role. When Sergio transfers his *SuperUser* role to chris, he will irremediably lose it. In this way, he is having no role, so that an opportune exception will be thrown if he would to play it.

Once we've written the code, we simply have to compile (better, pre-compile) it, in order to obtain a standard Java program. To do it, it's enough to run:

```
C:\>java JavaRolePreCompiler <MyFile.java> TargetFile.java
```

where MyFile.java is the source we've written few minutes ago, while TargetFile.java is the file that we will compile once we have it. It's a good choice to name the class that contain the main with TargetFile name.

Let's suppose that we saved our file as Test1.java, so we have to write:

```
C:\>java JavaRolePreCompiler <Test1.java> TestOne.java
```

Now we can compile TestOne.java, obtaining the executable:

```
C:\>javac TestOne.java
```

And, finally, we can execute:

```
C:\>java TestOne.java
```

You can download this (really working!) example (within all the work environment) from the website http://www.powerjava.org.

IV. JUST AN IDEA OF THE PRE-COMPILING RESULT

Only for give a track about the target of pre-compiling operations, we write an interesting example involving the *User* role.

In Figures 9, 10, 11 you can see the original code (before pre-compiling) and the pre-compiled code (in italic).

First, let's consider the *User* role definition (Figure 9-[13]) and it's corresponding pre-translation (Figure 9-[14]).

Following, we focus on the the class *Printer* (Figure 9-[15], 10-[16]), and in a particular way to those variables and structures added by the pre-compiler. We can see the HashTable rolelist (Figure 10-[17]), that will contain the class offering role definitions.

We defined an inner class *U* (Figure 10-[18]), realizing *User*, inside class *Printer* (which translation is in Figure 10-[19] – Figure 11).

It's very interesting to note the use of keyword *that*. It refers to *that* object is playing the role at issue, and it's used only in role implementation. An example is the invocation of *that.getLogin()* as a parameter of the print method in the previous code.

```
class Printer                                             [8]
{
  private int totalPrintedPages = 0;
  private int MAX_PAGES_USER = 100;
  private void print(Job job, Login login)
  {
    System.out.println("Printed job " + job.getID());
    totalPrintedPages += job.getNumberPages();
  }
  class U realizes User                                   [9]
  {
    int counter = 0;

    public U(){}
    public U(int i){}
    public int print(Job job)
    {
      if (counter > MAX_PAGES_USER)
      //throw new IllegalPrintException();
      {
        System.out.println("Too many pages printed!");
        return 0;
      }
      else
      {
        counter += job.getNumberPages();
        Printer.this.print(job, that.getLogin());
        return counter;
      }
    }
    public int getPrintedPages()
    {
      return counter;
    }
  }
  class S realizes SuperUser                              [10]
  {
    public int print(Job job)
    {
      Printer.this.print(job, that.getLogin());
      return totalPrintedPages;
    }
    public int getTotalPages()
    {
      return totalPrintedPages;
    }
  }
}
```

Figure 7 – Our example's code – Part 2 of 3

```
public class TestOne                                              [11]
{
  public static void main(String[] args)
  {
    Job j1 = new Job(1, 57);
    Job j2 = new Job(2, 160);
    Job j3 = new Job(3, 94);
    Job j4 = new Job(4, 211);
    Printer hp8100 = new Printer();
    Person chris = new Person();
    Person sergio = new Person();
    hp8100.new U(chris);
    hp8100.new S(sergio);
    ((hp8100.U)chris).print(j2);
    ((hp8100.S)sergio).print(j4);
    ((hp8100.U)chris).print(j1);
    ((hp8100.S)sergio).transfer(chris);                           [12]
    ((hp8100.U)chris).print(j1);
    ((hp8100.S)sergio).print(j3);
    System.out.println("Chris printed "
+ ((hp8100.U)chris).getPrintedPages() + " pages.");
    System.out.println("The printer printed "
+ ((hp8100.S)sergio).getTotalPages() + " pages.");
  }
}
```
Figure 8 – Our example's code – Part 3 of 3

V. Future developments

Our next goal is to implement features that make powerJava able to model all the cases represented in *Figures 1-4*; we will work to implement something of more collaborative too, and we can't exclude to try interactions between powerJava and other environments. We will surely work focusing on relations, as defined in [5] and [6], and sessions, as defined in [7].

```
role User playedby AccountedPerson              [13]
{
  int print(Job job);
  int getPrintedPages();
}

interface User                                  [14]
{
  int print(Job job);
  int getPrintedPages();
}

class Printer                                   [15]
{
  private int totalPrintedPages = 0;
  private int MAX_PAGES_USER = 100;
…
…
}
```
Figure 9 – Pre-compiling example – Part 1 of 3

```
class Printer implements ObjectWithRoles                                  [16]
{
  private java.util.Hashtable rolelist = new java.util.HashTable();  [17]
  private int totalPrintedPages = 0;
  private int MAX_PAGES_USER = 100;
...
...
}

class U realizes User                                                     [18]
{
  int counter = 0;

  public U(){}
  public U(int i){}

  public int print(Job job)
  {
    if (counter > MAX_PAGES_USER)
//throw new IllegalPrintException();
    {
      System.out.println("Too many pages printed!");
      return 0;
    }
    else
    {
      counter += job.getNumberPages();
      Printer.this.print(job, that.getLogin());
      return counter;
    }
  }

  public int getPrintedPages()
  {
    return counter;
  }
}

class U implements ObjectWithRoles, RoleInterface, User                   [19]
{
  private AccountedPerson that;
  public void destroy()
  {
    ((ObjectWithRoles)this.that).removeRole(this, Printer.this);
    this.that = null;
  }
  public void transfer()
  {
    ((ObjectWithRoles)this.that).removeRole(this, Printer.this);
    this.that = (AccountedPerson)req;
    ((ObjectWithRoles)this.that).setRole(this, Printer.this);
  }
  ... // Code defining the possibility for
  ... // the class to offer roles
  int counter = 0;
```
Figure 10– Pre-compiling example – Part 2 of 3

```
  public U(AccountedPerson that)
  {
    int alreadyPresent = 0;
    try
    {
      if(((ObjectWithRoles)that).getRole(Printer.this, “U”) != null)
      alreadyPresent = 1;
    }
    catch (Exception e){}
    if (alreadyPresent == 1)
      this.that = (AccountedPerson)
((ObjectWithRoles)that).getRole(Printer.this, “U”);
    else
      this.that = that;
    ((ObjectWithRoles)this.that).setRoles(this.Printer, this);
  }
  public U(int i)
  {
    int alreadyPresent = 0;
    try
    {
      if(((ObjectWithRoles)that).getRole(Printer.this, “U”) != null)
        alreadyPresent = 1;
    }
    catch (Exception e){}
    if (alreadyPresent == 1)
      this.that = (AccountedPerson)
((ObjectWithRoles)that).getRole(Printer.this, “U”);
    else
      this.that = that;
    ((ObjectWithRoles)this.that).setRoles(this.Printer, this);
  }
  public int print(Job job)
  {
    if (this.that == null)
throw new RunTimeException(“Reference to that is null”);
    if (counter > MAX_PAGES_USER)
//throw new IllegalPrintException();
    {
      System.out.println("Too many pages printed!");
      return 0;
    }
    else
    {
      counter += job.getNumberPages();
      Printer.this.print(job, that.getLogin());
      return counter;
    }
  }

  public int getPrintedPages()
  {
    if (this.that == null)
throw new RunTimeException(“Reference to that is null”);
      return counter;
  }
}
```

Figure 11 – Our example’s code – Part 3 of 3

REFERENCES

[1] F. Steimann. “On the representation of roles in object-oriented and conceptual modeling”. *Data & Knowledge Engineering*, 35:1 (2000) 83-106

[2] M. Baldoni, G. Boella, and L. van der Torre. “Modelling the interaction between objects: roles as affordances”. In *J. Lang, F. Lin, and J. Wang, editors, Knowledge Science, Engineering and Management: First International Conference*, KSEM, volume 4092 of LNCS, pages 42-54, Guilin City, China, August 5-8 2006. Springer.

[3] D. Norman. “The Design of Everyday Things”. *Basic Books*, New York (2002)

[4] M. Baldoni, G. Boella, and L. van der Torre. “Interaction between Objects in powerjava.” *Journal of Object Technology, Special Issue OOPS Track at SAC* 2006, 6(2), 2007.

[5] M. Baldoni, G. Boella, and L. van der Torre. “Relationships Define Roles, Objects Offer Them.” *Roles07 workshop at ECOOP*, pages 4-14.

[6] M. Baldoni, G. Boella, and L. van der Torre. “Relationships meet their roles in object oriented programming.” *Procs. of the 2nd International Symposium on Fundamentals of Software Engineering 2007 Theory and Practice* (FSEN ‘07).

[7] V. Genovese. “A Meta-model for Roles: Introducing Sessions.” *Roles07 workshop at ECOOP*.

Eric Arnaudo is student at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy.

Matteo Baldoni is Associated Professor at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, Torino, 10149, Italy. E-mail: baldoni@di.unito.it.

Guido Boella is Associated Professor at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy. E-mail: guido@di.unito.it.

Valerio Genovese is student at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy.

Roberto Grenna is PhD student at the Dipartimento of Informatica, Università di Torino, Corso Svizzera 185, 10149, Torino, Italy E-mail: grenna@di.unito.it.