Modeling and Programming with Roles: Introducing JavaStage

Fernando Sérgio Barbosa [a,1] and Ademar Aguiar [b]
[a] *Escola Superior de Tecnologia do Instituto Politécnico de Castelo Branco*
[b] *Faculdade de Engenharia da Universidade do Porto*

Abstract. Roles are not a new concept, but they have been used in two different ways: as modeling concepts in a static view and as instance extensions in a dynamic view. For these views only the dynamic offers supporting languages. The static view, although proving the utility of roles in modeling, does not offer a programming language that allows developers to use roles all the way from modeling to programming. We try to overcome this by presenting our role language JavaStage, based on the Java language. We do this by designing and implementing a simple framework and then compare the results with its OO equivalent. Our results show that static roles are in fact useful when used in code and that JavaStage features expand role reuse.

Keywords. Roles, Modeling, Code Reuse, Object Oriented, Single Inheritance Languages.

Introduction

To deal with the complexities of any problem we use abstractions. In Object-Oriented (OO) languages classes are the usual abstraction mechanism. Each class represents a concept from the problem domain. A concept can be specialized to a more specific concept. This specialization is, in OO systems, modeled by inheritance.

OO however cannot capture all possible views of the same object. Each existing concept may be viewed differently depending on the viewer: a river may be viewed as a food resource by a fisherman, a living place by a fish, a transport route for boats, etc. In order to accommodate these different views roles have emerged. An object therefore plays a role in a specific context. As an example the act of murder has the roles "murderer" and "murdered". These names define the individuals involved in the murder context. Outside this context the individuals have their own proper names.

Roles represent the behavior of an object with respect to a specific object collaboration task. As we have seen, objects behave in different ways when acting in different contexts. Therefore, in each context the object plays a different role. This introduces the notion of multiple perspectives [1]. The role is determined by the perspective the client holds on the object that plays the role. The perspective is a set of the properties of the object, modeled by a set of methods. Other objects in the collaboration can access the selected set of methods.

[1] Corresponding Author. Fernando Sérgio Barbosa, Escola Superior de Tecnologia, Av do Empresário, 6000-035 Castelo Branco, Portugal; E-mail: fsergio@ipcb.pt

There are many different role models [2][3][4] and interpretations on the role theme. Steinmann in [5] gives a good overview on the subject while identifying 15 features associated with roles. Graversen in [6] provides a role taxonomy that expresses the variability points of role semantics. It also contains a feature model that describes in detail properties of roles and how they can vary among role models.

A major part of the research focus on the dynamic aspects of roles. This dynamic aspects includes the attachment and detachment of roles from objects at runtime. Even though this can in fact extend the use of objects it also adds an extra complexity, both to program comprehension and development. Role composition problems add even further to that complexity, as some roles may be conflicting with each other.

Dynamic role approaches focus on expanding instances of an existing class to be used in a new context, so they develop roles for the classes that need to be expanded. Because of this focus on the classes the reusability of roles has been neglected in these languages. We will move the focus from the class to the role. While dynamic roles are a very important part of role usage we feel that the static nature of roles can also be important. Here static means that in a class all the roles it plays are active and no roles can be attached or detached at runtime.

Static roles have been used for modeling [2][7][8] and while the benefits (code reuse, comprehension, development and documentation) of these modeling techniques are demonstrated no programming language that deals only with the static nature of roles has yet appeared. Even the languages that deal with the dynamic aspects of roles neglect this static nature. This is a gap in programming languages that we try to bridge with JavaStage, a language extension to Java that uses roles as a first class construct. It is a goal of this paper to show that static roles can be worthy to use in all stages of the development process.

As an introduction to role modeling we will present a small framework. First the framework is presented using OO principles only. After the introduction of roles the framework will be redesigned using role modeling. In this we also present some role advantages that we expect to obtain from implementing the example using roles.

In this paper we present the JavaStage language so we present its syntax and features using as a running example the framework used in the modeling section. We will introduce JavaStage as we develop the roles found in the role version of the framework. Besides supporting roles JavaStage has these key features: a class can play any number of roles; a class can play the same role multiple times; roles can play other roles, role identity and a powerful renaming scheme.

The implementation of the framework will serve as the base for the discussion whether static roles may play an important part in software development and reuse of code. Although this is a preliminary study, based on a simple framework, our results show that roles are reusable to a high extent, especially if we use some key features of JavaStage. Other issues like multiple inheritance and the use of roles in design patterns will also be covered.

The rest of the paper is organized as follows. Section 1 presents the sample framework and its OO design as well as its role-based version. In section 2 are explained the guidelines used in developing the JavaStage language. In section 3 we introduce the JavaStage language extension using the sample framework, thus showing how to build the framework with JavaStage. Design decisions and the considerations on the implementation of the JavaStage language are discussed in Section 4. The use of roles in the example is discussed in section 5. Related work is presented in section 6 and section 7 concludes the paper.

1. Role Modeling

To present role modeling we will use a simplified graphical user interface (GUI) framework based on the Java AWT/Swing frameworks. The elements present in the framework represent all the widgets (referred to as components) that usually appear in a GUI, like windows, buttons, menus, toolbars, scrollbars, etc. Some components may own other components. For example, a window may own several toolbars, and a toolbar may own several buttons. This is a large framework but we will focus only on some of the elements and the basic structure of the framework. We will first present the OO version of the framework and then the version extended with roles.

1.1. OO Version

The main concepts in the GUI framework are the components, therefore it is natural to create classes that model each component. As all the components share a common concept, and may even share code and behavior, an inheritance hierarchy is created. Some components may contain other components, like the toolbar or even the window so the Composite pattern [9] was used. Since developers may be interested on knowing when the mouse is hovering a component the Observer pattern is used. Other instances of the Observer pattern are used as developers may be interested in other user's actions or if a component has lost/gained focus, etc. A component also has a collection of properties that specifies the way it should de drawn. Properties are represented by means of name-object pairs, where name is the property and object represents the value of the property. Figure 1 shows a possible class diagram of this framework.

1.2. Role Modeling Principles

Role modeling using static roles was used as an integral part of the OORam method [7] and by Riehle in [10] and [8]. In role modeling the notions of natural types and role types [11] as used in [12] distinguish between classes and roles.

A role type describes the view one object holds of another object. A role type is a type, so it can be described using an appropriate type specification mechanism. An object that conforms to a role type, acts according to the type specification, which is to say that the object plays that role. At any given time, an object may act according to several different role types. Thus, different clients may have different views on an object. Also, different objects may provide behavior specified by a common role type.

A natural type is a class and it represents a domain abstraction, its properties and behavior, much like in pure OO terms. But a class also defines which roles it plays and how they are composed. A class therefore defines which roles their instances may play. The union of the operations defined in the class and the operations defined in the roles constitutes the class interface, and the composition of all role types constitutes the type of the class. This is to say that the class interface is the union of the role interfaces. This is also argued in [13]. Because a standard OO class may be viewed as a class that plays only one role then this model is a canonical extension of the object model [12]. It also means that existing software can be integrated into the new model without changes.

We could achieve the same effect by using multiple inheritance, defining each role in a separate class. The composing class would just need to inherit from all classes. This situation is analyzed and compared to other role approaches in [14]. The use of multiple inheritance, however, has many problems. These come mostly by name

collisions when a class inherits from two or more superclasses that have equally named methods or fields and duplicated code when a class inherits twice from the same superclass – the classic diamond problem. Different languages provide different solutions to this problem as others simply forbid it. For these languages this is not a possible solution. As we will see later multiple inheritance also doesn't covers all situations that JavaStage does.

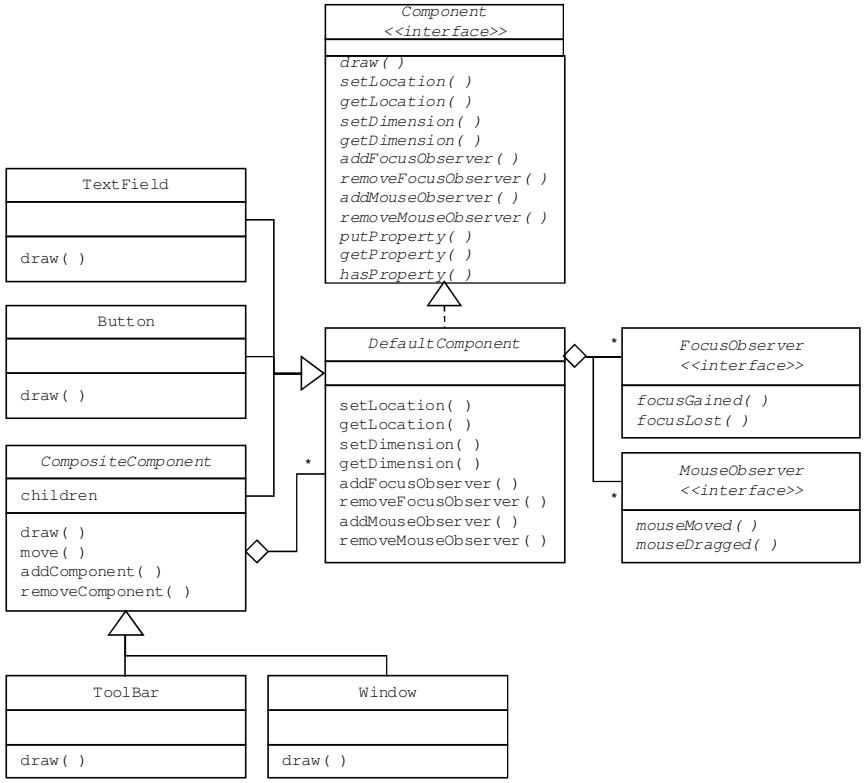

Figure 1. Class diagram of the Component Framework.

1.3. Role Model Version

Consider again the Component Framework. Clients that are interested in knowing if a component has lost or gained focus are not interested in drawing a component or if the user clicked it with the mouse. For those clients components assume the role of FocusSubject and only those operations related to that role are of interest. For those clients that want to know about the actions the user performs with the mouse on the component it plays the role of MouseSubject. Clients may set or read properties of the component so, for these, the component plays the role of PropertyProvider. The CompositeParent role is responsible for managing a collection of children and so it plays the role of a Container when clients wish to add or remove components to it. Finally the DefaultComponent class is responsible for the definition of the basic behavior of a graphical component. We can argue that this is also a role the class plays, even though it has no direct clients because it is an abstract class. Because subclasses

inherit all roles the superclass plays it will also inherit this same basic behavior, so the clients of subclasses are also clients of the superclass.

The mentioned roles are depicted in Figure 2, where we also show the role associations. The roles that do not offer any implementation are called no-operation roles [8] and are used as a handle for a role played by an object. These are the cases of the ComponentClient and PropertyClient. The BasicComponent role does not impose any behavior on its clients so any object may play the role of ComponentClient. The same is true for the PropertyClient.

With the created roles we can build a revised class diagram. This is shown in Figure 3. CompositeParent subclasses are not shown to save space as they have not changed from the original diagram. The client roles are also not shown, to save space, as their relationships with the framework roles are depicted in Figure 2.

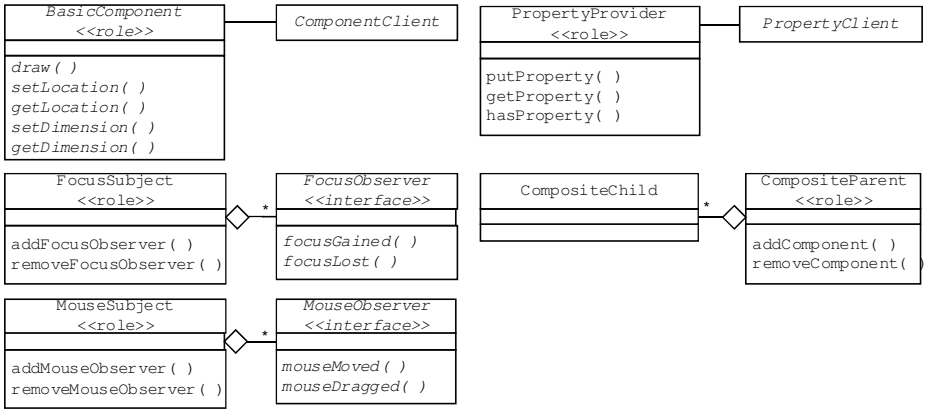

Figure 2. Roles and their relations in the Component Framework.

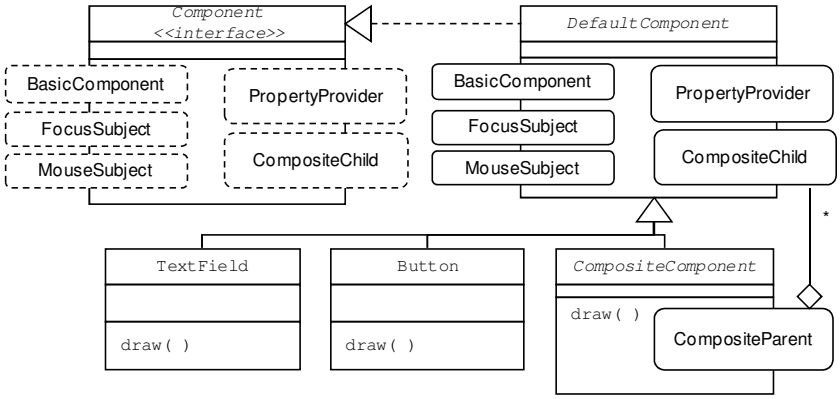

Figure 3. The revised class diagram of the Component Framework, now with roles. Rounded rectangles identify roles played by the class. Dashed round rectangles represent the interface provided by the role.

1.4. Role Modeling Advantages

Role modeling comes with several advantages in terms of reuse, comprehension, development and documentation [8]. When a class is described as being composed of

several roles it helps separating the various ways in which a class is used. This means that the documentation may also be done in these terms. That helps clients to better understand and use the class and focus in whichever aspect they are interested in. Designing the class can also be done in terms of its roles, thus developers are able to focus only in one aspect of the class. This enables independent development of a class with all its benefits in terms of reduced development time and complexity.

As seen in Figure 2 class relationships are reduced to role relationships. Because roles focus in a particular view of a class we need not to understand the target class in its whole. This facilitates the understanding and development of these relationships. Whenever needed the broader perspective can also be used. Role modeling allows for a transition between the role level and the class level, without losing any information.

The use of role modeling may also allow for better understanding using previous experiences. When a developer encounters roles that have a relationship in a system and he learns how to use them then when he encounters different roles that have the same or similar relationship the past experience will allow a better understanding. Because it is possible to have very different classes play the same role, or similar, these situations are more frequent, and so more experience will be gained, than it would be in a class to class relationship. One such example is the use of the Observer pattern. When experienced with a FocusSubject and how it works with a FocusObserver to use the MouseSubject and a MouseObserver is much simpler and straightforward.

The developed roles may also be reused by other classes that play the same role. The FocusSubject, or the MouseSubject are both reusable in other contexts. The PropertyProvider is another role that is reusable outside the context of the GUI. A word processor could use it for storing the properties of a paragraph, a line or a character (font, size, margins, etc). A figure framework could reuse it to store the properties of a graphical figure (line width, line style, etc).

2. Role Implementation Guidelines

There are several languages that support roles, and many role models, each choosing a set of role characteristics that suits its needs, and a good overview on these can be found in [6]. However, to our knowledge, there's never been an attempt to implement roles as static types and as components of classes. Chernuchin and Dittrich [14] compared five approaches for role support in OO languages. For roles as components of classes they reported that there are no tools or languages to support it, even if they scored well in the comparison. In this paper we show how JavaStage supports roles.

In this section we present the guidelines used in developing the JavaStage role model. Roles have not reached mainstream languages, maybe because developers do not see role benefits in programming, even if they see role benefits in design. We believe that if roles are used to write less and more modular code they will be more easily adopted. Therefore our major guideline when developing our role language is enhancing code reuse while retaining the role modeling advantages.

Modularization [15] is one of the most important concepts in software. A key concept in modularization is encapsulation. A well encapsulated module can be changed, even drastically changed, without affecting other modules. A module has an interface and an implementation. The interface defines how clients interact with the module, so it shouldn't change much along the module life-cycle as clients must be aware of the interface changes and change their own implementation accordingly.

Modules interact with each other via interfaces. Intra-modules interactions are more intense than inter-modules interactions so intra-modules interactions may require a specialized interface. To cope with this, most languages declare different levels of access. There are, usually, 3 levels of access: private, protected and public.

To maximize role reuse we have to enable the independent evolution between roles and players. If we treat a role as a module and the player as another module then we can strive for a greater independence between them. However we must consider that a role only makes sense when played by a class. The role will add to the class interface, like a superclass, and may require the class to provide some interface. This may call for a specific role-class interface. What cannot be done is to grant access to the role members by the class or to the class members by the role. If so any change in the class could lead to changes in the role and vice-versa. Therefore roles and classes must rely solely on interfaces.

When we started developing our role model we tried to maintain the model as close as possible to the OO paradigm. We believe that this approach is more likely to get the acceptance from the OO community than a model that would introduce many new concepts. To bring roles into mainstream languages the easiest way is to extend an already existing programming language so we opted to introduce extensions to the Java language. Language extensions should also be kept to a minimum, for the same reasons. Java was adopted because it is a widely used language and because the Java compiler is open source thus facilitating development of a compiler to the language extension.

Traditionally roles declare which classes can play them, whether by name or by interface, using a "played by" clause or similar. We believe that declaring predefined players is a great limitation in the reusability of roles. The same role used for a class could be reused for another class were not for the fact that the role developer did not foresee all its possible uses. One can argue that a role only makes sense in an interaction between classes and thus restrict the player classes to the ones involved in that interaction. However the same role could be reused in another, similar, interaction but with different players. If the "played by" clause specified an interface instead of a class it would be possible for the role to be played by many different classes but even this is not enough as we will explain later. Of course some restrictions must apply, because a class that plays a role must ensure a specific interface. Some form of declaring these requirements must be used but not by using a playedBy clause. We propose a requirement list. Each role must state which methods it requires the intrinsic, or any other participant, to have.

Methods' names are specific to an interaction. For example, the Observer pattern [9] describes an interaction between subjects and observers used in many different systems with minor changes, most notably the registration methods used to register an observer with a subject and the update methods used by the subject to notify its observers. In our example framework the Component plays the Subject role for a FocusObserver and for a MouseObserver. The focus subject defines methods like addFocusObserver and removeFocusObserver while the mouse observer defines methods like addMouseObserver, and removeMouseObserver.

We cannot use a single role to model both subjects, unless we use a generic name like addObserver. However, a method's name must indicate its functionality, so using such a name would reduce code comprehensibility. This could also limit the class to play only one subject role. Thus, renaming methods expands role reusability. Of course some restrictions apply, because a class playing a role must ensure a specific interface, but the interface should be configurable, at least in what respects to method names.

Some languages [16] use a “rename” clause that allow classes to rename a role method. If the role interface is big this task is tedious and error prone. We need a more expedite way of doing this.

Roles also interact with other objects. Again method names are important. Each subject has a method that calls the observer’s update method. In our example the focus subject calls the focusGained and focusLost methods on the observer while the mouse subject calls mouseMoved and mouseDragged. The rename clause, as used traditionally, is not usable here because it applies only to the role methods. We need a mechanism that allows fast renaming for role methods and methods called by the role.

To summarize, our guidelines when developing JavaStage, were:

- treat roles as modules, so roles must have an interface, state and behavior
- roles and classes communicate via interfaces
- introduce few language extensions
- replace the playedBy clause by a requirement list
- provide a method renaming mechanism that enables the role to be played by any class that fulfills some requisites

3. Introducing JavaStage

JavaStage is an extension to the Java language that supports roles as a first class identity and uses OO principles. We tried to introduce as few extensions to the language as possible and we only introduced 3 new keywords (role, plays and requires), which are responsible for the definition of a role, the requirements list and declaring that a class plays a role. We also introduced a renaming mechanism that makes use of a special character: #. The JavaStage syntax is presented in Figure 4.

```
type_decl ::= (role_decl | class_decl | … ) ";"
role_decl ::= {class_modifier} "role" identifier ["extends" class_name]
              "{" role_body "}"
role_body ::= ( requires | class_body )
requires ::= "requires" type "implements"
             ( (type method_decl) | constructor_decl) [throws] ";"
class_body ::= ( plays | … )
plays ::= {access_modifier} "plays" class_name [configs]
          identifier [role_params] “;”
configs ::= "(" config {"," config} ")"
config ::= identifier "=" identifier
role_params ::= "(" args_list ")"
```

Figure 4. The extension of java syntax in JavaStage.

3.1. Declaring Roles

The syntax to declare a role is similar to that of a class. Roles can declare fields and methods just like a class (see Figure 5). When we want a class to play a specific role we use the plays directive. Once again we mention that in most role models it is the

role that states which classes plays them. As discussed in section 2, this is restricting the reuse of the role, so in JavaStage it is the class that states the roles it plays.

Role members have all the visibility control available to classes. We extended the protected level to include the role-class relationship. A protected role member is accessible to its players and subroles. A protected class member is also accessible to roles. In JavaStage roles provide an interface, have an implementation and supports encapsulation. Roles and classes are thus completely independent modules and may be independently developed.

3.2. Playing Roles

To play a role the class uses a plays directive and gives the role an identity, as shown in Figure 6. To refer to the role the class uses its identity.

```
public role PropertyProvider {
  private HashMap<String,Object> properties =
                                  new HashMap<String,Object>();
  public Object getProperty( String name ){
    return properties.get( name );
  }
  public void putProperty( String name, Object value ){
    properties.put( name, value );
  }
  public boolean hasProperty( String name ){
    return properties.containsKey( name );
  }
}

public role FocusSubject {
  private Vector<FocusObserver> observers =
                                  new Vector<FocusObserver>();
  public void addFocusObserver( FocusObserver obs ){
    observers.add( obs );
  }
  public void removeFocusObserver( FocusObserver obs ){
        observers.remove( obs );
  }
  protected void fireFocusGained( FocusEvent e ){
    for( FocusObserver obs : observers )    obs.focusGained( e );
  }
}
```

Figure 5. Definition of the PropertyProvider and FocusSubject role (first version).

```
public class DefaultComponent implements Component {
    plays PropertyProvider propertyProvider;
    plays FocusSubject focusSbj;
    plays MouseSubject mouseSbj;
    plays BasicComponent basicComp;

    public void draw( Graphics g ){
       Border b = (Border) propertyProvider.getProperty( "Border" );
       if( b != null ){ b.draw( g ); }
       //...
    }
}
```

Figure 6. Definition of the DefaultComponent class (first version).

When a class plays a role all the non private methods of the role are added to the class interface. Thus a class can be seen either as being composed from several roles or as an undivided entity. This means that for clients of the class DefaultComponent the representations of the class in Figure 1 and Figure 3 are equivalent.

A class can reduce the visibility of the exported role members. If a class uses protected in the plays clause then all the public role methods are exported as protected. This way a class can use roles to provide an interface for their subclasses only. A class cannot change a single member's visibility.

3.3. Stating Role Requirements

A role does not know who their players might be but may need to exchange information with its player, so it must require the player to have a particular interface. We do that using a requirements list. The list can include required methods from the player but also required methods from objects which the role interacts with. In the list the role states the method owner and the method signature. The requires statement has the following syntax:

```
requires supplier implements methodSignature;
```

To indicate that the owner is the player we use the Performer keyword. Performer is used within a role as a place-holder for the player's type. This enables roles to declare fields and parameters of the type of the player.

With this requirement list we could develop a generic subject role that requires observers to have an update method. Note that the observer type does not need to implement a specific interface.

```
public role GenericSubject<ObserverType> {
  requires ObserverType implements void update();

  private Vector<ObserverType> observers = new Vector<ObserverType>();

  public void addObserver( ObserverType o ){   observers.add( o );  }
  public void removeObserver( ObserverType o ){  observers.remove( o ); }
  protected void fireChanged( ){
    for( ObserverType o : observers )  o.update();
  }
}
```

Figure 7. A generic subject role requiring its observers to implement an update method (first version).

3.4. Renaming role methods

Take the PropertyProvider example. It assumes that a property is identified by a name and that name is a String. It would be more reusable if it used generics for the property type. We can also use a generic type to specify the value type instead of type Object. After a closer look, the property provider is in fact a map that maps keys to values. We could reuse a map implementation if we inherited from a Map class, but that would be conceptually wrong. Our class is not a map: it plays the role of a property map. The right way is to develop a role that is a Mapper. The only thing that prevents this are methods names. A more general Mapper cannot use names like getProperty or hasProperty, because these are only associated with properties.

We propose a renaming method that allows an easy configuration of methods' names. Each name may have three parts: a configurable one and two fixed. Both fixed parts are optional thus leaving the name of a method to be fully configurable by the class. The configurable part is bounded by # as shown next.

```
fixed#configurable#fixed
```

The configuration of the name is done by the class that plays the role in the plays clause, using the syntax:

```
plays roleType( configurable = nameToUse) roleId;
```

We can then build our Mapper role using this renaming strategy (see Figure 8).

```
role Mapper<KeyType, ValueType> {
   private Map<KeyType,ValueType> properties;

   ValueType get#Thing#( KeyType name ){return properties.get( name );}

   void put#Thing#( KeyType name, ValueType value ){
     properties.put( name, value );
   }
   boolean has#Thing#( KeyType name ){
     return properties.containsKey( name );
   }
}
```

Figure 8. Definition of the Mapper role, that replaces the PropertyProvider role, with configurable methods (second version).

For the PropertyProvider we would use the following plays clause to configure the Mapper:

```
plays Mapper<String,Object> (Thing = Property)  propertyProvider;
```

The DefaultComponent class retains its original behavior but we now have a role that is more generic and reusable than the one we started with. In fact the role is able to be reused in all situations that the PropertyProvider would be used and many more.

We used the same approach to the CompositeParent and came up with a Container role. We must state the fact that reusing the Mapper role for the PropertyProvider role is an implementation choice and does not invalidate the modeling design or even the documentation.

3.5. Providing Multiple Versions of a Method

It's possible to declare several versions of a method using multiple definitions of the configurable name. This way methods with the same structure are defined only once. For this feature to be used we must use a configurable called method inside a configurable role method. We must name the called method after the method it is called from. This is done using a dot name, where the configuration name before the dot is the configuration name of the outer method.

```
void role#Method#( ){
    performer.called#Method.inner#( );
}
```

This way the compiler knows that both methods are to be used together and can check if one configuration name has the same number of configurations of the other and it also checks that they are defined sequentially in the plays clause:

We could then build our generic subject role using this renaming strategy, so it no longer relies on an update method. We can also configure the addObserver method so we can use an appropriate name. We also added an event parameter to the update method. Our enhanced version of the role is shown in Figure 9.

```
public role GenericSubject<ObserverType, EventType> {
  requires ObserverType implements void #Fire.update#( EventType e );

  public void add#Observer#( ObserverType o ){ observers.add( o );  }
  public void remove#Observer#( ObserverType o ){ observers.remove( o );}
  protected void fire#Fire#( EventType e){
      for( ObserverType o : observers)  o.#Fire.update#( e );
  }
}
```

Figure 9. Definition of the generic subject role (second version) now with configurable methods.

We can now use the generic subject role to be used either as a focus subject or as a mouse subject, providing all the notification methods. The focus subject would be configured like:

```
plays GenericSubject<FocusObserver,FocusEvent>
  ( Fire = FocusGained, Fire.update = focusGained,
    Fire = FocusLost,   Fire.update = focusLost,
    Observer = FocusObserver ) focusSbj;
```

```
public role GenericContainer<ThingType> {
 private Vector<ThingType> ins = new Vector<ThingType>();

 public void add#Thing#(ThingType t){ ins.add(t);}
 public void remove#Thing#(ThingType t){ ins.remove(t);}
 protected Vector<ThingType> get#Thing#s(){ return ins; }
}

role GenericSubject<ObserverType,EventType> extends
                                        GenericContainer<ObserverType>{
  requires ObserverType implements void #Fire.update#( EventType e );
  protected void fire#Fire#( EventType e ){
    for( ObserverType o : get#Thing#s( ) )  o.#Fire.update#( e );
  }
}

public role FocusSubject {
  plays GenericSubject<FocusObserver,FocusEvent>
        (Fire= FigureChanged, Fire.update= figureChanged,
         Fire= FigureMoved,   Fire.update = figureMoved,
         Fire= FigureRemoved, Fire.update= figureRemoved,
         Thing = FigureObserver ) figureSbj;
}
```

Figure 10. Roles extending roles and roles playing roles.

3.6. Roles Playing Roles or Inheriting from Roles

Roles can play roles but can also inherit from roles. A role cannot inherit from a class and vice-versa. When a role inherits from a role that has configurable methods it cannot define them. When a role plays another role it must define all its configurable methods, but can have its own configurable methods.

For example managing observers is a part of a more general purpose concern that is related to containers. We can say that the subject role is an observer container and develop a generic container role and make the subject inherit from the container. We can therefore reuse the container role already mentioned for the CompositeParent. We can also develop a FocusSubject that plays the GenericSubject role (see Figure 10).

3.7. Role Constructors

We may need to parameterize roles. In our container role we may want the container to be an ArrayList instead of a Vector, as shown in Figure 11. We support role constructors but we do not allow role instantiation.

```
public role GenericContainer<ThingType> {
 private List<ThingType> ins;

 public GenericContainer( ){  ins = new Vector<ThingType>(); }
 public GenericContainer( List<ThingType> container ){ ins = container; }
 // ...
}

public class CompositeComponent extends DefaultComponent {
  plays GenericContainer<Component>( Thing = Component )
            components( new ArrayList() );
  // ...
}
```

Figure 11. Final version of the Container role now supporting constructors. The CompositeComponent plays the GenericContainer role configuring it to use an ArrayList as the container.

4. Role Implementation and Design Decisions

To support JavaStage we developed a compiler based on the javac compiler. This was possible because the java compiler code is open source. We extended it to comply with the JavaStage syntax. Because it is a full java compiler and not a preprocessor it can be used to compile java code with or without roles in an transparent way. JavaStage's roles are therefore compatible with Java, and do not need any preprocessing. The generated code is plain java bytecodes so it is does not need any role aware JVM.

4.1. Implementation of JavaStage

To support roles we opted for a version using inner classes. This not only supports all our options for the approach but also ensures that no performance penalty is introduced while maintaining the final code executable in existing virtual machines.

When a class plays a role the role code is copied to the class as an inner class. Figure 12 shows an excerpt of how the DefaultComponent class, playing the

GenericSubject configured to a focus subject, would look like. We can see that a role is used as an object inside the class. This allowed roles to have constructors. However no one can directly instantiate a role, as roles aren't meant to have instances.

```
public class DefaultComponent implements Component{
  public void draw( Graphics g ){
       Border b = (Border) propertyProvider.getProperty( "Border" );
       if( b != null ){   b.draw( g );  }
  }

  private class GenericContainer#focusSbj {
    private java.util.Vector<FocusObserver> ins =
                  new java.util.Vector<FocusObserver>();
    public void addFocusObserver(FocusObserver t){ ins.add( t );   }
    protected java.util.Vector<FocusObserver>
          getFocusObservers(){  return ins;   }
  }

 private class GenericSubject#focusSbj extends GenericContainer#focusSbj{
     protected void fireFocusGained( FocusEvent e){
       for(FocusObserver o : getFocusObservers() ) o.focusGained( e );
     }
     // ... other fires
  }

  private GenericSubject#focusSbj focusSbj  =
                new GenericSubject#focusSbj();
  public void addFocusObserver( FocusObserver t){
     focusSbj.addFocusObserver( t );
  }
  protected void fireFocusGained( FocusEvent e ){
     figureSbj.fireFocusGained( e );
  }
}
```

Figure 12. Excerpt of how the AbstractFigure class would look after the role is added to the class.

The # in the inner class name ensures there isn't a name clash between synthetic classes and written code. The role identity in the class name ensures that no conflicts arise when playing the same role twice (e.g. the class name for the FocusSubject role is GenericSubject#focusSbj and for a MouseSubject would be GenericSubject#mouseSbj).

Role methods are copied to the class interface and call the corresponding method on the role object (see addFocusObserver). This apparently introduces a redirection but it is easily solved by making the compiler call the role method directly when the class method is called. No performance loss is therefore introduced.

4.2. Role Identity

Our roles have an identity, associated with the player, given by the player in the plays clause. When the player access role members it uses this identity. If the role is public then its identity is accessed just like any class member. Class clients can then select the role they want. However, we consider that roles should not be public, for the same reasons fields should not be public.

Role identity is also used to distinguish between roles when resolving a conflicting method. We use the identity to specify which role we want to access. This is better than

using class names, as in C++, because we can change the role hierarchy and still be able to maintain the code unchanged. Another reason to provide roles with an identity is to support state. Since each role field must be accessed using the role identity there never is a name conflict between fields of different roles.

4.3. The plays clause

Shouldn't the plays clause be considered equivalent to the extends or implements clauses and be placed accordingly? After all it does have impact in the class interface. There are in fact several reasons for not doing so. One is the role identity which, purposely, resembles an object declaration (see the implementation section). Yet another reason is the naming configuration, which would clutter that declaration. A final reason is role initialization, as roles may have non default constructors. It would be awkward to do these configurations in an implements-like declaration.

4.4. Conflict resolution

Class defined methods always take precedence over role methods. Conflicts may arise when a class plays roles that have methods with the same signature or when an inherited method has the same signature of a role method. When conflicts arise the compiler issues a warning. Developers can handle the conflict by redefining that method and calling the intended method. This is not mandatory because the compiler uses, by default, the method of the first role in the plays clause order. Role methods also override inherited methods. This may seem like a fragile rule, but we believe that for most situations it will be enough. We argue that even if a conflicting method is later added to a role the compiler does issue a warning so the class developer is aware of the situation and can solve it. The important part is that the class developer can solve the situation as he wishes and not as imposed by the role or superclass' developers.

4.5. Role as types

We could let roles define a type and write code that would work with any class that plays the role. Our renaming strategy, however, forbids this because the actual interface a role provides is configured by the class. Roles that do not use renaming could define a type. In our approach making a role a type is simple, however we still haven't explored if there are advantages in considering roles as types. We defer this decision to future work.

4.6. Requirements listing

Requirements in most role languages are made by a playedBy clause, in our case restrictions are imposed via the requirement list. This list allows us to provide a renaming mechanism that tailors the role to the class. This also allowed us to impose restrictions on other classes that are part of the interaction, like the observer interface in the subject role.

5. Example Discussion

We succeeded in implementing all the roles used in the design so it seems the gap between design and implementation is filled by JavaStage. In this section we also discuss some of the role advantages we feel could be inferred from this simple example.

5.1. Code Reuse

The code reuse of roles showed real promises, and somewhat exceeded expectations, because we were able to develop roles that have a wider field of application than this framework. The GenericSubject role is capable of being used in several scenarios, even scenarios not related to Components. The Mapper and the Container are also reusable in many different contexts as discussed in the implementation section.

We believe that a major factor in role reuse is that in our approach roles do not define a specific player. If the role does not require a player, or clients, to have a certain interface it means that the role is reusable in unforeseen scenarios. Also the requirements the role imposes on the player/clients are configurable making it open for tailoring to a variety of contexts widening the domain of possible usages. Of course the renaming mechanism also plays an important part in this reuse.

5.2. Context Influences Method Names.

Another observation we did in this study is the way in which the context of use of a role influences method names. We found roles in different contexts to be basically the same except for method names and the types of arguments. This is quite visible in the GenericSubject, Mapper and Container. In the GenericSubject the type of observers influence the parameter types but also methods' names (addMouseObserver, addFocusObserver, fireMouseMoved, …). In the Container the type of the content also influences the parameter types and methods' names. We used the container in the CompositeParent and tailored it to use methods like addComponent, and also used it in the subject now tailored to use methods like addFocusObserver.

From this example we can reason that for roles to adapt to different contexts they must have a renaming mechanism. Some role languages offer a rename mechanism but none has such a powerful renaming mechanism as JavaStage. The rename clause of such languages is used on a method by method basis. Modifying an entire interface is hard. None of the languages has the multiple method versions feature of JavaStage.

5.3. Roles and Multiple Inheritance

Roles can also emulate multiple inheritance for those languages that do not support it. Java uses interfaces, but they can only declare constants and method signatures and cannot have state or default method implementations. This may result in the duplication of a default implementation in classes that implement the same interface while inheriting from different superclasses.

In Java it is common to start an inheritance hierarchy with an interface and then a superclass that provides the default behavior for that hierarchy, like we did in the GUI framework (see Figure 1). We argue that the default implementation should be provided by a role and that the default class plays that role. This way we can reuse the basic behavior whenever there is a need to. To emulate multiple inheritance a class

implements the various interfaces and plays the role with the default implementation for each interface. Possible ambiguities are resolved as explained in section 4.4.

Our roles can also be used in multiple inheritance languages. A limitation of multiple inheritance is the fact that we cannot use it to play the same role multiple times as we can only inherit once from each superclass: if a class inherited from GenericSubject for a focus subject it could not inherit again for a mouse subject. With roles this is possible. The renaming mechanism enhances this by allowing the class to properly configure methods' names, which is not possible in multiple inheritance.

5.4. Reusing Design Patterns Code.

Finally we observed that one of the roles we reused, Subject, is a participant in an instance of a well-known design pattern: Observer [9]. The role ComponentParent also participates in an instance of the Container pattern. Because these roles are part of a design pattern and we reused them, it opens the question if whether or not roles can be used to reuse actual pattern code.

Design patterns are known for offering flexible solutions to reoccurring software development problems. Each pattern is applied in a given context and defines a number of participants that collaborate with each other to carry out their responsibilities. The pattern also defines the ways in which the participants collaborate. These participants can be seen as roles. Design patterns are one of the most successful abstraction tools that developers have. Despite the fact that patterns are reusable the code that implements them mostly is not [17].

There are some attempts to reuse pattern code [18][19]. These attempts however use dynamic solutions. For static versions there are tools that allow the instantiation of patterns [20][21]. We think its better not to depend on tools but rather on language features, even if this is not consensual [22]. Our approach hints that roles can be used as a way to statically reuse pattern code. So assess this we did a study in which we tried to implement roles for each one of the 23 Gang of Four design patterns [9]. As a result of that study [23] we developed roles for 10 of those patterns which showed that it is possible to create generic roles.

5.5. Development Time and Program comprehensibility

Our simple study does not allow us to achieve definite conclusions about development time or code comprehensibility. For these we must rely on the studies by Riehle in [8][10]. We can nevertheless speculate that if we placed roles like GenericSubject, Container and Mapper, in a library of roles the development of this framework would be faster.

Such a library could lead to a situation where most of the instances of the observer pattern used the GenericSubject role. This would make this role very popular and hence all the code that used it would be easily comprehended and apprehended by anyone familiar with the role. The same goes for the other developed roles. We can further extend this to the point that roles can uniform the way some code is written so it will eventually become more and more easy to develop and understand.

5.6. Language Usage

This study did not take into account the difficulty in learning the JavaStage language. For this we would need to use a group of developers. This will be subject of another study. Nevertheless we believe that the few extensions that JavaStage introduces are simple to understand and do not pose great difficulties. The renaming mechanism with the # use is the feature that may raise more comprehension problems. In our opinion, however, its great usefulness makes for this extra complexity.

5.7. More Examples of use

Our example framework is somewhat limited and very simple, but we found it was useful to provide an overview on role modeling and to introduce JavaStage. To assess the usability of JavaStage we conducted several other studies, waiting publication, with larger frameworks. For example we developed a version of the JHotDraw[2] framework with roles. For this we detected code clones and tried to reduce these clone by applying roles. We were able to reduce almost all the duplicated code in a quite satisfactory way. That study also showed that the use of the renaming mechanism and the requirements list are indeed very useful. The code for the example presented in this paper, the mentioned JHotDraw framework version, as well as the JavaStage compiler are available at http://www.est.ipcb.pt/pessoais/fsergio/javastage.

6. Related Work

We divided related work into 3 subsections. The first subsection is related directly to our work, that is, static role approach, the second is dedicated to the dynamic role approach and the third is dedicated to other forms of composition.

6.1. Static roles

There are not other languages that support static roles. Riehle in [8] showed how roles can be successfully applied to frameworks in the various challenges frameworks arise, like documentation, comprehensibility, etc. But no role language ir proposed, there's only a description on how roles could be used in several languages. In Java he uses interfaces to define roles but the role code is written in the class itself, making role reuse impossible. We extended his work by taking roles to the implementation stage, thus taking the advantages of roles all the way from modeling to final code.

Chernuchin and Dittrich [12] described ways to deal with role dependencies. We didn't considered them as we believe that it would introduce further complexity to the language and its benefits do not make up for that extra complexity. Their role model is somewhat similar to ours. Even though they suggest programming constructs to support their approach no role language or tool has emerged.

Chernuchin et. al [14] compared multiple inheritance, interface inheritance, the role object pattern, object teams and roles as components of classes for role support in OO languages. They used criteria such as encapsulation, dependency, dynamicity,

[2] www.jhotdraw.org

identity sharing and the ability to play the same role multiple times. Roles as components of classes fared well in the comparison and the only drawback, aside dynamicity, was the lack of support from tools and languages. JavaStage fills that need.

6.2. Dynamic roles

A well-known approach to role dynamics is the role object pattern [24][25]. This pattern is very flexible because it allows the addition and removal of role objects to a so called core object. Encapsulation and role hierarchy are supported. Also supported is the ability to play the same role several times. It is of a great complexity so it is hard to apply and comprehend. When there are just a few roles its complexity overcomes its usefulness. It is also not recommended when roles have many interdependencies.

Object Teams [26] is an extension to Java that uses roles as first class entities. They also introduce the notion of team. A team represents a context in which several classes collaborate to achieve a common goal. Even though roles are first class entities they are implemented as inner classes of a team and are not reusable outside that team. Roles are also limited to be played by a specific class.

EpsilonJ [16] is another java extension that, like Object Teams, uses aspect technology. In EpsilonJ roles are also defined as inner classes of a context. Roles are assigned to an object via a bind directive. EpsilonJ uses a requires directive that is similar to ours. It also offers a replacing directive to rename methods names but that is done on an object by object basis when binding the role to the object.

PowerJava [27] is yet another java extension that supports roles. In PowerJava roles always belong to a so called institution. When an object wants to interact with that institution it must assume one of the roles the institution offers. To access specific roles of an object castings are needed. Roles are written for a particular institution, therefore we cannot reuse roles between institutions.

6.3. Other related approaches

Traits [28] offer a way of composing software that are somewhat similar to Mixins [29]. A trait is the primitive unit of code reuse, like our roles, which means that only traits can be used to compose classes. Traits can also be used to compose other traits. Furthermore a class composed with traits can be seen either as a flat collection of methods or as a being composed by traits. This flat property means that the code inside the trait can be seen as the code inside the class, for example, a super reference inside the trait code refers to the superclass of the class that uses the trait. In our approach we can also see a class as simply a set of methods, forgetting that it plays a role, but we have not this flat property, as a super reference in a role refers to the superrole.

Like our approach traits provide methods for the class and may require the class to provide some methods. A significant difference between our approach and traits is that roles can have state and traits cannot. The trait's solution for the Container role would have to rely on the class to provide the element’s storage. In traits we can provide aliases for methods but it is done on a method by method basis, while our renaming strategy enables multiple renaming. We can also rename methods called by the role, while in traits we can only alias methods from the role itself.

Feature Oriented Programming (FOP) decomposes the system into features [30], so they are the main abstractions during design and implementation. Features reflect user requirements and incrementally refine each other. Because features are the

distinguishing product characteristics of Software Product Lines, FOP is mainly used for SPL and program generators. Mixin layers [31] are used to implement features. Each mixin layer contains the code for the role(s) that each class plays in a given feature and composes them in a static component. Roles can be used instead of mixins, as they offer more ways of configurations and don't have mixins limitations like a linear composition order.

Aspect-Oriented Programming is an approach that tries to modularize crosscutting concerns [32]. However AOP is not close to OO and requires learning many new concepts. And while the modularization of crosscutting concerns is the flagship of AOP several authors disagree [33][34]. Concepts like pointcuts and advices are not easy to understand, and their effects are more unpredictable than any OO concept. A particular one is the fragile pointcut [35]. This problem arises when simple changes made to a method code makes a pointcut either miss or incorrectly capture a joint point thus incorrectly introducing or failing to introduce the necessary advice.

The obliviousness feature of AOP [36] means that a class is aspect unaware so aspects can be plugged or unplugged as needed. This somewhat resembles dynamic roles and explain why AOP is used in dynamic role languages. But it also introduces problems in comprehensibility [37]. To fully understand the system we must not only know the classes but we also have to know the aspects that affect each class. This is a major drawback when maintaining a system, since the dependencies aren't always explicit and there isn't an explicit interface between both parts.

With our approach all dependencies are explicit and the system comprehensibility is increased when compared to the OO version [10]. We do not, however, have the obliviousness of AOP because the class knows and is aware of the roles it plays. That way any changes to the class code are innocuous to the role, as long as the contract between them stays the same. As a final point we do not believe that our approach can replace AOP. They are different and approach different problems. We believe that for modeling static concerns our approach is more suitable while AOP is better suited for pluggable and unpluggable concerns.

Caesar [38] uses aspect technology to modularize crosscutting concerns and enhance reuse of aspects leading to a greater reduction of repeated code. Caesar uses an Aspect Collaboration Interface that decouples aspects binding and implementations by defining them in a separated module. Caesar does not allow method renaming. We can compare our Subject role with the subject role in [38]: our role subject has fully configurable methods names while in Caesar all subjects must have an addObserver method. In Caesar for a class to be a subject for two different actions (e.g. MouseListener and FocusListener) we must define a binding class for each, while with roles we can do the configuration in the class alone. Lastly Caesar's observers can have only one notification method, named notify, whereas in ours we can define several notification methods each with a meaningful name.

7. Conclusion and Future Work

In this paper we presented how to model a simple framework using static roles and how to implement that same framework with JavaStage. This is the first language to tackle the static role use.

We presented JavaStage as a language supporting role that is an extension to java that can be used to develop java programs with roles. To implement JavaStage we

changed the javac compiler to support its syntax and role model in a way that it makes the use of roles fit nicely in traditional java code. It compiles plain java code and java code with roles in a transparent way. No precompiler is necessary and no other tool is required to support roles.

Results from this preliminary study seem to indicate that the expected benefits of roles correspond to real and practical benefits. Role benefits included reuse of roles, reduced development time and enhanced comprehensibility, among others. We showed that we could develop roles that are capable of being reused in a variety of different scenarios. This is a consequence of the JavaStage requirements list and the renaming mechanism both for role method names as well for player/clients methods names. Reduced development time follows directly from the code reuse. With roles libraries development time is therefore reduced, and we argued that a role library is possible. Enhanced comprehensibility is more difficult to show in such a simple example but, nevertheless, we feel that using a library will reduce code complexity thus making it more understandable.

Roles also proved to be effective in emulating multiple inheritance, with all its advantages (and more) and without its problems.

As a side effect our work hinted that roles could be used to implement traditional design patterns roles that are reusable, thus contributing to many of today frameworks. This study was debated in other work, so they are not discussed in depth here. Future work also includes the redesign of real frameworks using roles in its implementation. With this work we showed that an OO system gains by using roles.

References

[1] Kristensen, B. B., (1996): Architectural abstractions and language mechanisms, Asia Pacific Software Engineering Conference.

[2] Kristensen, B. B., (1995): Object-oriented modeling with roles, in Proceedings of the 2nd International Conference on Object-Oriented Information Systems, Springer-Verlag, pp. 57–71.

[3] Kristensen, B. B. and Østerbye, K.. (1996): Roles: Conceptual abstraction theory & practical language issues, Theory and Practice of Object Systems 2(3): 143–160.

[4] Reengskaug, T. 2007: Roles and classes in object oriented programming. In: Roles 2007. Proceedings of the 2nd Workshop on Roles and Relationship in Object Oriented Programming, Multiagent Systems, and Ontologies

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

[6] Graversen, K. B., (2006): The nature of roles - A taxonomic analysis of roles as a language construct, Ph. D. Thesis, IT University of Copenhagen, Denmark

[7] T. Reenskaug, P. Wold, and O. A. Lehne. Working with objects - the OOram software engineering method. Manning, 1996.

[8] Riehle, D. 2000. Framework Design: A Role Modeling Approach, Ph. D. Thesis, Swiss Federal Institute of technology, Zurich.

[9] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., (1995): Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[10] Riehle, D. and Gross, T. 1998. Role Model Based Framework Design and Integration." In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications

[11] Sowa, J., (1984): Conceptual Structures: Information Processing in Mind and Machine. Addison Wesley.

[12] Chernuchin, D., and Dittrich, G. 2005. Role Types and their Dependencies as Components of Natural Types. In 2005 AAAI Fall Symposium: Roles, an interdisciplinary perspective.

[13] Steimann, F., (2001): Role = interface: a merger of concepts, Journal of Object-Oriented Programming 14(4): 23–32.

[14] Chernuchin, D., Lazar, O. S., and Dittrich, G., Comparison of Object-Oriented Approaches for Roles in Programming Languages, Papers from the 2005 Fall Symposium, ed. Guido Boella, James Odell, Leendert van der Torre, and Harko Verhagen. Technical Report FS-05-08. American Association for Artificial Intelligence, Menlo Park, California.
[15] Parnas, D. L., (1972): On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12, Dec. 1972, 1053-1058
[16] Tamai, T., Ubayashi, N., and Ichiyama, R., (2007): Objects as Actors Assuming Roles in the Environment, in Software Engineering For Multi-Agent Systems V: Research Issues and Practical Applications, Lecture Notes In Computer Science, vol. 4408. Springer-Verlag, Berlin, Heidelberg,
[17] Soukup, J. 1995. Implementing Patterns. In: Coplien J. O., Schmidt, D. C. (eds.) Pattern Languages of Program Design. Addison Wesley
[18] Hannemann J., Kiczales G. 2002. Design Pattern Implementation in Java and AspectJ. Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002), Seattle, USA, ACM press, pp. 161-173.
[19] Rajan, H. 2007. Design pattern implementations in Eos. In Proceedings of the 14th Conference on Pattern Languages of Programs (Monticello, Illinois, September, 2007). PLOP '07. ACM, New York.
[20] Phattarasukol, S. and Sang, D. 2005. PatternStudio: a tool for design pattern management. In Companion To the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05.
[21] Budinsky, F. J., Finnie, M. A., Vlissides, J. M., and Yu, P. S. 1996. Automatic code generation from design patterns. IBM Syst. J. 35, 2 (May. 1996), 151-171.
[22] Chambers, C., Harrison, B., and Vlissides, J. 2000. A debate on language and tool support for design patterns. In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (Boston, MA, USA, January 19 - 21, 2000). POPL '00. ACM, New York, NY.
[23] Barbosa, S. and Aguiar, A. (2011). Generic roles, a test with patterns In 18th Conference on Pattern Languages of Programs, PloP 2011 Oct 21-23, Portland, OR, USA.
[24] Bäumer, D., Riehle, D., Siberski, W. and Wulf, M. (2000): Role Object, in Pattern Languages of Program Design 4, Massachusetts: Addison-Wesley, 2000. Chapter 2, page 15-32.
[25] Fowler, M. (1997): Dealing with Roles, Proc. of the 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, USA, Sept. 2-5.
[26] Herrmann, S., (2005): Programming with Roles in ObjectTeams/Java. AAAI Fall Symposium: "Roles, An Interdisciplinary Perspective".
[27] Baldoni, M., Boella, G. van der Torre, L., (2007): Interaction between Objects in powerJava, journal of Object Technologies 6, 7 - 12.
[28] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. Lecture Notes in Computer Science, 2743:248–274, 2003.
[29] G. Bracha and W. Cook. Mixin-based inheritance. In Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications, OOPSLA/ECOOP '90, pages 303–311, New York, NY, USA, 1990. ACM.
[30] Apel, S., Kästner, C. (2009): An Overview of Feature-Oriented Software Development, in Journal of Object Technology, vol. 8, no. 5, July–August 2009,pages 49–84
[31] Smaragdakis, Y. and Batory, D., Mixin Layers (2002): An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. ACM TOSEM, 11(2).
[32] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold. W. G., (2001): An overview of AspectJ. In proceedings of ECOOP 2001, Budapest, Hungary, (LNCS, vol. 2072), Springer; 327–335
[33] Steimann, F., The paradoxical success of aspect-oriented programming", in OOPSLA '06, Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (2006) 481–497
[34] Przybyłek, A.(2001). Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming , J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705.
[35] Koppen, C., Störzer, M.: PCDiff, 2004: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany
[36] Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns at OOPSLA (2000)
[37] Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H., 2006: Modular Software Design with Crosscutting Interfaces. IEEE Software 23(1), 51–60 (2006)
[38] Mezini, M. and Ostermann, K. 2003. Conquering Aspects with Caesar. In Proc. of AOSD 2003.