ROLES AS MODULAR UNITS OF COMPOSITION

Fernando Barbosa[1] and Ademar Aguiar[2]
[1]*Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Av. do Empresário, Castelo Branco, Portugal*
[2]*Departamento Informática, Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, Porto, Portugal*
fsergio@ipcb.pt,_ ademar.aguiar@fe.up.pt

Keywords: Modularity, Inheritance, Roles, Composition, Reuse.

Abstract: Object oriented decomposition is the most successful decomposition strategy used nowadays. But a single decomposition strategy cannot capture all aspects of a concept. Roles have been successfully used to model the different views a concept may provide but, despite this, roles have not been used as building blocks. Roles are mostly used to extend objects at runtime. In this paper we propose roles as a way to compose classes that provides a modular way of capturing and reusing those aspects that fall outside a concept's main purpose, while being close to the OO approach. We present how roles can be made modular and reusable. We also show how we can use roles to compose classes using JavaStage, a java extension that support roles To validate our approach we developed generic and reusable roles for the Gang of Four patterns. We were able to develop reusable roles for 10 out of 23 patterns, which is a good outcome.

1 INTRODUCTION

To deal with the complexities of any problem we normally use abstractions. In Object-Oriented (OO) languages classes are the usual abstraction mechanism. Each class represents a specific concept. A single decomposition technique however cannot capture all possible views of the system (Tarr,1999) and each concept may be viewed differently depending on the viewer: a river may be a food resource to a fisherman, a living place to a fish, etc. Roles can accommodate these different views.

Roles were introduced by Bachman and Daya (Bachman,1977) but several role models have since been proposed. But the definitions, modeling ways, examples and targets are often different (Graversen, 2006)(Steimann,2000). The research on roles has focused largely on its dynamic nature (Herrmann, 2005)(Baldoni,2007)(Tamai,2007), modelling with roles (Riehle,1998) and relationships (Pradel,2008).

Role modelling, by decomposing the system into smaller units than a class, has proved to be effective, with benefits like improved comprehension, documentation, etc (Riehle,2000). However, no language supports such use of roles. To overcome this fact we'll focus our role approach in class composition and code reuse.

We propose roles as a basic unit we can compose classes with. A role defines state and behaviour that are added to the player class. Roles provide the basic behaviour for concerns that are not the class's main concern, leading to a better modularization. A class can then be seen either as being composed from several roles or as an undivided entity.

To maximize role reuse we'll use modularity principles as guidelines. We intend to develop roles as modular units, making them reusable. We argue that developing roles independently their players will make them much more reusable. To express our ideas we created JavaStage, an extension to Java. We will use JavaStage in the examples so we will give a brief introduction so examples are clear.

To show that roles can be reusable modules we show that it is possible to build a role library. We started our role library with the analysis of the Gang of Four (GoF) design patterns (Gamma,1995). We were able to develop generic roles for 10 patterns.

We can summarize our paper contributions as: a way of composing classes using roles as modular units; a java extension that supports roles; a role library based on the GoF patterns.

This paper is organized as follows: Section 2 gives a brief description of decomposition problems. Section 3 discusses how to enhance role reuse. Section 4 shows how to use roles to compose classes using JavaStage. Our roles for the GoF patterns are debated in Section 5. Related work is presented in Section 6, and Section 7 concludes the paper.

2 DECOMPOSITION PROBLEMS

How do we decompose a system? There still isn't a definitive answer and there are many decomposition techniques. The most used today is Object Oriented Decomposition, but some argue that a single decomposition strategy cannot adequately capture all the system's details (Tarr, 1999).

Consequences of using a single decomposition strategy are crosscutting concerns. They appear when several modules deal with the same problem, which is outside their main concern, because one cannot find a single module responsible for it. This leads to scattered, replicated code.

Because a module must deal with a problem that is spread by several others, changes to that code will, quite probably, affect other modules. Independent development is thus compromised, evolution and maintenance are a nightmare because changes to a crosscutting concern need to be done in all modules.

We will tackle this problem by using roles as a building block for classes. We put the crosscutting concern in a role and the classes play the role. Any changes to the concern are limited to the role, improving maintenance and reducing change propagation. The crosscutting concerns become more modular.

2.1 Multiple Inheritance

To overcome decomposition restrictions some languages use multiple inheritance. But multiple inheritance also has multiple problems, caused mostly by name collisions when a class inherits from two or more superclasses that have methods with the same signature or fields with the same name. It can even occur when a class inherits twice from the same superclass – the diamond problem. Different languages provide different solutions (virtual classes in C++) and others simply forbid it like Java.

Java uses interfaces when a class must be seen as conforming to another type. Interfaces only declare constants and methods signatures so they have no state or method implementations. This may result in the code duplication in classes implementing the same interface but with different superclasses.

It is usual to start an inheritance hierarchy with an interface and then a superclass providing the default behaviour for that hierarchy. We argue that the default implementation should be provided by a role and the superclass plays that role. This way we can reuse the basic behaviour whenever we need to, thus preventing the use of multiple inheritance. This is depicted in Figure 1. The example shows a Figure hierarchy with an interface and a role at the top. The DefaultFigure class implements the interface and plays the role. All its subclasses inherit this default behaviour. The ImageFigure, a subclass from another hierarchy, also becomes part of the Figure hierarchy by implementing the Figure interface. It also plays the BasicFigure role so it has the same default behaviour every DefaultFigure subclass has.

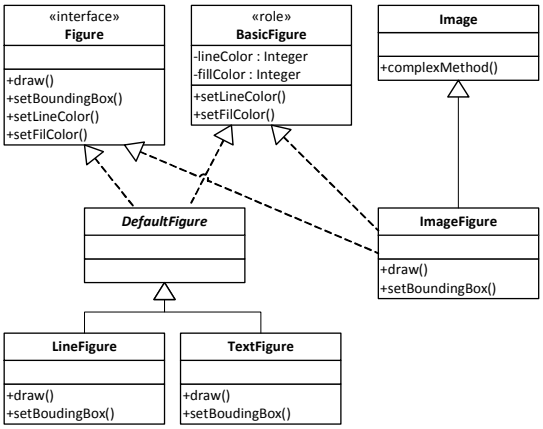

Figure 1. Example of a Figure hierarchy with both an interface and a role as top elements

2.2 Aspect Oriented Programming

There are other attempts to remove crosscutting concerns, like Aspect Oriented Programming (AOP) (Kickzales,2001). 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 (Steimann, 2006)(Przybyłek,2011).

Concepts like pointcuts and advices are not easy to grasp, and their effects are more unpredictable than any OO concept. A particular one is the fragile pointcut (Koppen,2004): simple changes in a method can make a pointcut either miss or incorrectly capture a joint point thus incorrectly introducing or failing to introduce the required advice.

AOP obliviousness (Filman,2000) means that the class is unaware of aspects and these can be plugged or unplugged as needed. This explains why some dynamic role languages use AOP. But it also brings comprehensibility problems (Griswold, 2006). To fully understand the system we must know the classes and the aspects that may affect them. This is a major drawback when maintaining a system, since the dependencies aren't always explicit and there isn't an explicit contract between both parts.

With roles all dependencies are explicit and the system comprehensibility is increased compared to the OO version (Rielhe,1998). Roles do not have AOP obliviousness because the class is aware of the

roles it plays. Any changes to the class do not affect the role, if the contract between them stays the same.

Our approach does not replace AOP. They are different and approach different problems. We believe that for modelling static concerns our approach is more suitable while AOP is better suited for pluggable and unpplugable concerns.

2.3 Traits

Classes' composition using alternatives to multiple inheritance have been proposed such as mixins (Bracha,1990)(Bracha,1992) and traits (Scharli, 2003)(Ducasse,2004)(Black,2004). Traits have one advantage over mixins and single inheritance: the order of composition is irrelevant. Traits have first appeared in smalltalk but some attempts have been made into bringing traits in to java-like languages (Quitslund,2004)(Smith,2005).

Traits can be seen as a set of methods that provide common behaviour. Traits are stateless, the state is supplied by the class that uses it and accessed by the trait through required methods, usually accessors. Trait's methods are added to the class that uses them. The class also provides glue code to compose the several traits it uses.

Traits don't have visibility control, meaning that a trait cannot specify an interface and all trait methods are public, even auxiliary ones. Since traits cannot have state then this is a minor problem, but it does limit a class-trait interface.

Traits have a flattening property: a class can be seen indifferently as a collection of methods or as composed by traits, and that a trait method can be seen as a trait method or as a class method.

In our approach a class can be seen as being composed from several roles or as an undivided entity. This is not to be confused with the flattening property of traits. A super reference in a trait refers to the super of the class that uses the trait, while a super reference in the role refers to the super role.

Our roles can have state, visibility control and their own inheritance hierarchy while traits cannot. In our approach the order of role playing is also irrelevant except for a specific conflict resolution, but it is so to facilitate development and can be overridden by the developer or by the compiler.

3 REUSING ROLES

This section is dedicated to what we believe are the factors that will enhance role reuse: independent evolution of roles and players, role configuration, and roles being used as components for classes.

3.1 Roles as Modules

Modularization (Parnas,1972) is one of the most important notion in software development. Breaking a system into modules allows each module to be independently developed, shortening development time. Other advantages are better comprehensibility, enhanced error tracing, etc, but the one developers treasure most is the modules' high reusability. It allows library development and libraries reduce the amount of code one must write to build a system.

A key concept is encapsulation. When a module is encapsulated changes in the module, even drastic ones, do not affect other modules. A module has an interface and an implementation. The interface defines how clients interact with the module, and it shouldn't change much along the module life-cycle as clients must be aware of the changes and change their own implementation accordingly.

Modules interact with each other but intra-modules interactions are more intense than inter-modules interactions. Intra-modules interactions may require a specialized interface. To cope with this, most languages declare different levels of access, usually 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. Thus roles must provide an interface and ensure encapsulation.

Providing an interface is simple if we use roles as first class entities. Encapsulation and independent development raises a few issues. We must consider that roles only make sense when played by a class. But classes cannot have access to role members and vice-versa. If they did roles and classes could not be developed independently, because any change to the role structure could cause changes in the class and vice-versa. Therefore roles and classes must rely solely on interfaces.

3.2 Dropping the playedBy clause

Many role approaches focus on the dynamic use of roles: extending objects by attaching roles. Roles are usually bounded to a player by a playedBy clause that states who can play the role. In dynamic situations where roles are developed for extending existing objects this is acceptable, even if it restricts role reuse, but not in static situations.

Using an example derived from (Ingesman,2011) we show, in Figure 2, a Point class representing a 2D cartesian coordinate and a Location role that

provides a view of Point as a physical location on a map. We also present a PolarPoint class that represents a coordinate in polar coordinates. The role could apply to both classes but the playedBy forbids it as these classes are not related. Making one a subclasse of the other would violate the "is a" rule of inheritance. We could use a Coordinate interface with getX and getY methods with both classes implementing that interface. This cannot be done in a dynamic context where both classes are already developed and cannot be modified.

Our purpose is to use roles as building blocks and not for extending objects. This is a totally different way of viewing role-class relationships. Our roles are meant to be used to compose classes so roles are developed without knowledge of all classes that can play them. Thus using the playedBy clause would limit role reusability. In the example, if we develop a role for both classes the role must state that it needs the player to have getX and getY methods. Some form of declaring these requirements must be used but not by using a playedBy clause.

```
class Point {
  int x, y;
  Point(int x, int y){this.x= x; this.y= y;}
  int getX( ) { return x; }
  int getY( ) { return y; }
}
class PolarPoint {
  int r;  double beta;
  PolarPoint(int r, double b) {
    this.r = r; beta = b; }
 int getX(){return (int)(r*Math.cos(beta));}
 int getY(){return (int)(r*Math.sin(beta));}
}
role Location playedBy Point {
  string getCountry() {
    int x = performer.getX();
    int y = performer.getY();
    // converting point to a country name
    String country = "PT";
    return country;
  }
}
```

Figure 2. A Point class, a Location role playable by it and a PolarPoint class that could also play the Location role.

3.3 The need to rename methods

Methods names are specific to an interaction. For example, Observer (Gamma,1995) describes an interaction between subjects and observers. It is used in many systems with minor changes, usually the methods used to register an observer with a subject and the update methods used by the subject to notify its observers. A Subject role for a MouseListener instance of the pattern would define methods like addMouseListener, or removeMouseListener. That role could not be reused for a KeyListener instance which uses methods like addKeyListener or removeKeyListener.

A method's name must indicate its purpose, so a name like addListener reduces comprehensibility, and can limit the class to play only one subject role. Thus, renaming methods expands role reusability. A class that plays a role must ensure a specific interface, but that interface should be configurable, at least in what respects to method names.

Some languages (Tamai,2007) 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. For example, each subject has a method that calls the observer's update method. In the Java AWT implementation of the pattern there are several methods like mousePressed, mouseReleased, etc. The rename clause 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.

3.4 Summary

For roles to be fully reusable then they must provide an interface; ensure encapsulation; be developed independently from its players; state requirements player must fulfil; provide a method renaming mechanism that enables the role to be configured by the player

4 COMPOSING CLASSES USING ROLES

To support roles we developed JavaStage, an extension to Java. Examples in this paper have been compiled with our JavaStage compiler. We will not discuss JavaStage's syntax in detail but it will be perceptible from the examples and we will explain it briefly so that examples are understandable.

4.1 Declaring Roles

A role may define fields, methods and access levels. A class can play any number of roles, and can even play the same role more than once. We refer to a class playing a role as the player of that role.

When a class plays a role all the non private methods of the role are added to the class. To play a

role the class uses a plays directive and gives the role an identity, as shown in Figure 3. To refer to the role the class uses its identity.

As an example we will use the Subject role from the Observer pattern. Consider a Figure in a drawing application. Whenever the Figure changes, the drawing must be updated so the figure plays the role of an observer's subject. Being a subject is not the Figure main concern so it's wise to develop a subject role, shown in Figure 3, to capture that concern and let figures play it. In the code we omitted access modifiers for simplicity, but they should be used.

4.2 Stating Role Requirements

A role does not know who will be its players but may need to exchange information with them 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 the role interacts with. The list states the method owner and the method signature. 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. This is shown in Figure 4 which shows a singleton role.

4.3 Method renaming

We developed a renaming mechanism, to enhance role reuse and facilitate role configuration, which allows method names to be easily configured. Each name may have three parts: one configurable and two fixed. Both fixed parts are optional so the name can be fully configurable by the player. The configurable part is bounded by # as shown next.

```
fixed#configurable#fixed
```

The name configuration is done by the player in the plays clause as depicted in Figure 5. To play the role the class must define all configurable methods.

We can take our figure subject role and make it more generic with this renaming mechanism. In Figure 5 we show how we can use method renaming to make our subject role more generic. It also shows a class playing that role as a FigureObserver subject and as a FigureHandlerObserver subject.

```
role FigureSubject {
 Vector<FigureObserver> observers =
              new Vector<FigureObserver>();
 void addFigureObserver( FigureObserver o){
       observers.add( o );
 }
void removeFigureObserver(FigureObserver o){
      observers.remove( o );
 }
 protected void fireFigureChanged( ){
    for( FigureObserver o : observers )
      o.update( );
 }
}
class DefaultFigure implements Figure {
 plays FigureSubject figureSbj;

 void moveBy(int dx, int dy) {
   // code for moving the figure
   // firing change, using role identity
   figureSbj.fireFigureChanged();
 }
}
```

Figure 3. A Figure subject role for an instance of the observer pattern and a class playing it.

```
public role Singleton {
  requires Performer implements Performer();
  private static Performer single = null;
  public static Performer getInstance( ){
    if( single == null )
        single = new Performer();
    return single;
  }
}
```

Figure 4. A Singleton role requiring its player to have a default constructor.

```
public role GenericSubject<ObserverType> {
 requires ObserverType implements
                    void #Fire.update#();
 public void add#Observer#( ObserverType o){
   observers.add( o );
 }
 protected void fire#Fire#( ){
   for( ObserverType o : observers )
     o.#Fire.update#( );
 }
}
class DefaultFigure implements Figure {
  plays GenericSubject<FigureObserver>
    (Observer = FigureObserver,
     Fire = FigureChanged,
     Fire.update = figureChanged
    ) figureSbj;
  plays GenericSubject<FigureHandleObserver>
     ( Observer = FigureHandleObserver,
       Fire = FigureHandleChanged,
       Fire.update = figureHandleChanged
     ) figHandleSbj;
  public void moveBy(int dx, int dy) {
    figureSbj.fireFigureChanged();
  }
}
```

Figure 5. The generic subject role now with configurable methods (in bold) and a class playing that role twice

4.4 Multiple versions of a method

It's possible to declare several versions of a method using multiple definitions of the configurable name. Methods with the same structure are defined once.

We can expand FigureObserver to include more update methods to specify which change occurred, like figureMoved. Such plays clause would be:

```
plays GenericSubject<FigureObserver>
  ( Fire = FigureChanged,
    Fire.update = figureChanged,
    Fire = FigureMoved,
    Fire.update = figureMoved,
    Observer = FigureObserver  ) figureSbj;
```

4.5 Making use of name conventions

Another feature of our renaming strategy is the class directive. When class is used as a configurable part it will be replaced by the name of the player class. This is useful in inheritance hierarchies because we just need to place the plays clause in the superclass and each subclass gets a renamed method. It does imply that calls will rely on name conventions.

One such case is the Visitor pattern. This pattern defines two roles: the Element and the Visitor. The Visitor declares a visit method for each Element. Each Element has an accept method with a Visitor as an argument that calls the corresponding method of the Visitor. Visitor's methods usually follow a name convention in the form of visitElementType. We used this property in our VisitorElement role, as shown in Figure 6. The example shows it being used in a Figure hierarchy with figures as Elements. It also shows that Figure subclasses don't have any pattern code, because they will get an acceptVisitor method that calls the correct visit method.

4.6 Roles playing roles or inheriting from roles

Roles can play roles but can also inherit from roles. 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.

For example managing observers is a part of a more general purpose concern that is to deal with collections. 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.

If the FigureSubject role can be played by several classes then we'll create a FigureSubject based on GenericSubject. Because we need to rename the role methods the FigureSubject role must play the generic Subject role and define all its methods. DefaultFigure would then use FigureSubject without any configuration.

Both situations are depicted in Figure 7.

```
role VisitorElement<VisitorType> {
 requires VisitorType implements
   void visit#visitor.class#( Performer t );

  void accept#visitor#( VisitorType v ){
    v.visit#visitor.class#( performer );
  }
}
class DefaultFigure {
  plays VisitorElement<FigureVisitor>
            ( visitor = Visitor ) visit;
  // … rest of class code
}
class LineFigure extends DefaultFigure {
    // no Visitor pattern code
}
interface FigureVisitor {
    void visitLineFigure( LineFigure f );
    void visitTextFigure( TextFigure f );
    //…
}
```

Figure 6. The VisitorElement role, a class Figure that plays the role, a subclass from the Figure hierarchy and the Visitor interface.

```
role GenericContainer<ThingType> {
  Vector<ThingType> ins =
                    new Vector<ThingType>();
  void add#Thing#( ThingType t )  {
    ins.add( t );
  }
  void insert#Thing#At(ThingType t,int idx){
    ins.insertElementAt( t, idx );
  }
  protected Vector<ThingType> get#Thing#s(){
    return ins;
  }
}
role GenericSubject<ObserverType>
     extends GenericContainer<ObserverType>{
  requires ObserverType implements
                    void #Fire.update#();
  protected void fire#Fire#( ){
    for( ObserverType o : get#Thing#s() )
         o.#Fire.update#( );
  }
}
role FigureSubject {
  plays GenericSubject<FigureObserver>
    ( Fire = FigureChanged,
      Fire.update = figureChanged,
      Fire = FigureMoved,
      Fire.update = figureMoved,
      Thing = FigureObserver ) figureSbj;
}
class DefaultFigure implements Figure {
  plays FigureSubject figureSbj;
}
```

Figure 7. Role inheritance and role playing roles.

4.7 Conflict resolution

Class methods have 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. The conflict can be resolved by redefining the 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 order and role methods override inherited methods. This may seem a fragile rule, but we believe it will be enough for most cases. 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. He can solve the situation as he wishes and not as imposed by the role or superclass' developers.

5 TOWARDS A ROLE LIBRARY

5.1 Roles in Design Patterns

To start our role library we analysed the 23 GoF patterns (Gamma,1995). They are a good starting point because of its wide use. If we create roles for these patterns then our approach will have impact on many of today frameworks and applications.

Each pattern defines a number of collaborating participants. Some participants can be seen as roles while others cannot. This distinction is made in (Hannemann,2002) by considering the roles defining or superimposed. For each pattern we took the roles of each participant and focused on similar code between pattern instances to find reusable code. We present our results by groups of patterns. They were grouped by similarities between implementation or problems. We've built a sample scenario for each pattern but will not discuss them, due to space constraints.

5.1.1 Singleton, Composite, Observer, Visitor.

Singleton, Observer and Visitor were already discussed. Composite uses the Container role. Each composite maintain a collection of child components and implements the operations defined by the component hierarchy. Children management is common between instances, so we reused the Container role. Component operations are instance dependent and not suitable for generalization, even if they mostly consist in iterating through the children and performing the operation on each child.

5.1.2 Factory Method, Prototype.

With these patterns we developed roles that provide a greater modularity and dynamicity not present in traditional implementations. The use of the class directive for renaming is common to these roles.

Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate. Implementation of this pattern is instance dependent. There is, however, a variation whose purpose is to connect parallel class hierarchies: each class from a hierarchy delegates some tasks to a corresponding class of another hierarchy. Each class has a method that creates the corresponding class object (product). We moved the creation of the product to a creator class, which provides methods to create all products, one method each. Classes just call the right method in the creator. One advantage is the modularization of the pattern as the association between classes is made in a single class not on a class by class basis. Future changes are made to the creator class only. Because the creation process is in a single class we can dynamically change the creator. We developed a role that allows the specification of the factory method that creates the object of the parallel class. The method uses the class directive so the plays clause is used only in the top class. It implies the use of naming conventions, but that is a small price to pay for the extra modularity. We also developed a role with a fixed creator, when dynamic creators aren't needed.

The Prototype pattern specifies the kind of objects to create using a prototypical instance, and creates new objects by cloning this prototype. The prototype class has a clone method that produces a copy of the object. Every class has its own cloning method but it may not be sufficient because the clone method may do a shallow copy where a deep copy is needed, or vice-versa. The client should choose how the copy is made. We developed a role that moves the creation of the copy to another class, as we did for FactoryMethod. That class is now responsible for creating the copies of all classes used as prototypes and thus may choose how to make the copy. Because it uses the class directive Prototype subclasses don't need to declare the clone method.

5.1.3 Flyweight, Proxy, State, Chain of Responsibility

Roles developed for these patterns are basically management methods. They are useful as they

provide the basic pattern behaviour and developers need to focus only on the specifics of their instance.

Flyweight depends on small sharable objects that clients manipulate and on a factory of flyweights that creates, manages and assures the sharing of the flyweights. The concrete flyweights are distinct but many flyweight factories have a common behaviour: verify if a flyweight exists and, if so, return it or, if not, create, store and then return it. Our flyweight factory role manages the flyweights. Players supply the flyweight creation method.

In Proxy a subject is placed inside one object, the proxy, which controls access to it. Some operations are dealt by the proxy, while others are forwarded to the subject. Which methods are forwarded or handled are instance dependent as is the creation of the subject. Forwarding and checking if the subject is created or accessible is fairly similar between instances. Our proxy role stores the subject reference and provides the method that checks if the subject exists and triggers its creation otherwise.

The State pattern allows an object to alter its behaviour when its internal state changes. There are almost no similarities in this pattern because each instance is unique. Our role is responsible for keeping the current state and for state transitions. The state change method terminates the actual state before changing to, and starting, the new state.

Chain of Responsibility avoids coupling the sender of a request to its receiver. Each object is chained to another and the request is passed along the chain until one handles it. Implementations of this pattern often use a reference to the successor and methods to handle or pass the request. Each instance differs in how the request is handled and how each handler determines if it can handle the request. Some implementations use no request information, others require some context information and in others the request method returns a value. We developed a role for each variation.

5.1.4 Abstract Factory, Adapter, Bridge, Decorator, Command, Strategy.

The code for these patterns is very similar between instances but we could not write a role for any. For example, many abstract factories have methods with a return statement and the creation of an object. However the object's type and how it is created are unique. Adapter instances are similar in the way the Adapter forwards calls to the adaptee, but the call parameters and return types vary for each method.

5.1.5 Builder, Façade, Interpreter, Iterator, Mediator, Memento,Template Method.

These patterns showed no common code between instances, because they are highly dependent on the nature of the problem. For example, an iterator is developed for a concrete aggregate and every aggregate has a unique way to traverse.

5.2 Summary

We developed roles for a total of 10 patterns out of 23, which is a good outcome, especially because every developed role is reusable in several scenarios. We believe that our Subject role, for example, will be useful for a large number of Observer instances. There are also additional advantages in some roles, like a better modularity in Factory Method and Prototype. Some roles are limited in their actions, like State but are highly reusable, nevertheless.

From our study there are a few patterns that do not gain from the use of roles. These roles are quite instance specific and the classes built for their implementation are dedicated and are not reusable outside the pattern. There are a few patterns that could benefit from using roles to emulate multiple inheritance and provide a default implementation to operations done in a class inheritance hierarchy, like Abstract Factory and Decorator. We also found similar code between instances that we could not put into a role. This was the case of patterns that forwarded method calls, like Adapter, Decorator and Proxy. However the variations were not supported by roles because they were in the methods return type and parameters types and number.

5.3 Testing Role-player Independency

In order to asses if our roles are independent of their players we took the sample scenarios that illustrated its use and built a dependency structure matrix (DSM) for each. We use our sample of the Observer role and its DSM as an example of that work.

For an Observer sample we developed a Flower class that notifies its observers when it opens, as shown in Figure 8. Flower plays the FlowerSubject role, which is the Subject role configured to this particular scenario. As an observer we developed a Bee class that when notified prints a message saying it is seeing an open flower. The code for the bee, observer interface and the flower event are not shown for simplicity. The FlowerSubject role is not really necessary as the Flower could configure the Subject role directly but it is good practice to do so.

From that sample we obtained the DSM of Figure 9. Here we can find that there is no dependency between the Subject role and the Flower class and that the FlowerSubject depends only on the Subject role and not vice-versa. If we group the classes into modules as shown in the figure we can see that the module where the role is included does not depend on any other module. It shows that the flower module is dependent from the role module via the role. It also shows that the Flower module does not depend on its concrete observers, as expected from the observer pattern. The Subject role is therefore independent of its players as could be inferred from the use of the subject role in a total of 3 examples in this paper alone. We may also add that we also used that same role in the JHotDraw Framework.

```
public role FlowerSubject {
 plays Subject<FlowerObserver,FlowerEvent>(
  Thing=FlowerObserver,
  Fire=Open,Fire.update=flowerOpened) sbj;
 }
}
public class Flower {
   plays FlowerSubject flwrSubject;
   private boolean opened = false;

   public void open(){
     opened = true;
     fireOpen( new FlowerEvent( this ) );
   }
}
```

Figure 8. The FlowerSubject role and the Flower class from our subject role sample.

Name		1	2	3	4	5	6	7	8
EventType	1								
ObserverType	2	1							
Subject	3	1	1						
FlowerEvent	4							1	
FlowerObserver	5				1				
FlowerSubject	6	1			1	1			
Flower	7				1		1		
Bee	8				1	1		1	

Figure 9. DSM of the Observer role sample.

6 RELATED WORK

To our knowledge there's never been an attempt to implement roles as static types and as components of classes. Riehle (Riehle,2000) lays the foundations for role modeling using static roles. He proved role usefulness in the various challenges frameworks are faced with, like documentation, comprehensibility, etc. He does not propose a role language, but simply explains how roles could be used in some languages.

Chernuchin and Dittrich (Chernuchin,2005B) use the notion of natural types and role types that we followed. They also described ways to deal with role dependencies which we didn't consider as it would introduce extra complexity to the role language. They suggest programming constructs to support their approach but no role language has emerged.

Chernuchin and Dittrich (Chernuchin,2005) compared five approaches for role support in OO languages. They were multiple inheritance, interface inheritance, the role object pattern, object teams and roles as components of classes. They used criteria such as encapsulation, dependency, dynamicity, identity sharing and the ability to play the same role multiple times. Roles as components of classes compared fairly well and the only drawback, aside dynamicity, was the absence of tools that supported it. With JavaStage that drawback is eliminated.

Object Teams (Herrmann,2005) is an extension to Java that uses roles as first class entities. They introduce the notion of team. A team represents a context in which several classes collaborate. 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 (Tamai,2007) 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 as-signed to an object via a bind directive. EpsilonJ uses a requires directive 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 (Baldoni,2007) 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.

7 CONCLUSIONS

We presented a way of composing classes using roles. With roles we are able to capture the concerns that are not the class main concern and modularize them. We presented an, hitherto missing, language that supports roles as components of classes and showed how we can use it to compose classes. Moreover we showed that roles can be made

reusable to a great extent. The result was the development of generic roles for 10 GoF patterns.

REFERENCES

Bachman, C. W., Daya, M., (1977): The role concept in data models, in Proceedings of the 3rd International Conference on Very Large Databases 464–476.

Baldoni, M., Boella, G. van der Torre, L., (2007): Interaction between Objects in powerJava, journal of Object Technologies 6, 7 - 12.

Black, A. and Scharli, N. (2004) Programming with traits. In Proceedings of the Interna-tional Conference on Software Engineering 2004, May 2004.

Bracha, G. and Cook, W. (1990): Mixin-Based Inheritance. In Proceedings of the OOPSLA/ECOOP, pages 303–311, Ottawa, Canada. ACM Press.

Bracha, G. (1992): The programming language jigsaw: mixins, modularity and multiple inheritance. PhD thesis, University of Utah.

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.

Chernuchin, D., Lazar, O. S., and Dittrich, G., (2005) Comparison of Object-Oriented Approaches for Roles in Programming Languages, Papers from the 2005 Fall Symposium, ed.

Devanbu, P.; Batory, B.; Kiczales, G.; Launchbury, J.; Parnas, D.; Tarr, P. (2003); "Modularity in the new millennium: a panel summary", Proc. of the 25th International Conference on Software Engineering

Ducasse, S., Schaerli, N., Nierstrasz, O., Wuyts, R. and Black, A. (2004): Traits: A mechanism for fine-grained reuse. In Transactions on Programming Languages and Systems.

Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns at OOPSLA (2000)

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

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)

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

Hannemann J., Kiczales G. 2002. Design Pattern Implementation in Java and AspectJ. Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications, Seattle, USA.

Herrmann, S., (2005): Programming with Roles in ObjectTeams/Java. AAAI Fall Symposium: "Roles, An Interdisciplinary Perspective".

Ingesman, M. D., Ernst, E. 2011. Lifted Java: A Minimal Calculus for Translation Polymorphism, in Proceeding of the International Conference on Objects, Models, Components and Patterns, Zurich, Switzerland

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, 200

Koppen, C., Störzer, M.: PCDiff, 2004: Attacking the fragile pointcut problem. In: European Interactive Workshop on Aspects in Software, Berlin, Germany

Parnas, D. L., (1972): On the criteria to be used in decomposing systems into modules. Commun. ACM 15, 12, Dec. 1972, 1053-1058

Pradel, M., (2008): Explicit Relations with Roles - A Library Approach. Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL)

Przybyłek, A.. Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming , J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705, pp. 163–178, 2011.

Quitslund, P. and Black, A. (2004): Java with traits - improving opportunities for reuse. In Proceedings of the 3rd International Workshop on Mechanisms for Specialization, Generalization and inheritance

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

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

Scharli, N., Ducasse, S., Nierstrasz, O. and Black, A. (2003): Traits: Composable units of behavior. In Proceedings of ECOOP 2003, volume 2743 of Lecture Notes in Computer Science. Springer.

Smith, C. and Drossopoulou, S. (2005): Chai: Traits for Java-like languages. In Proceedings of ECOOP 2005.

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

Steimann, F., 2006, The paradoxical success of aspect-oriented programming", in OOPSLA '06, Proceedings of the 21st Annual Conference on Object-Oriented Programming Languages, Systems, and Applications

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, 185-203

Tarr, P. L., Ossher, H., Harrison, W. H., and S. M. S. Jr. 1999. N degrees of separation: Multi-dimensional separation of concerns. In International Conference on Software Engineering.