Reusable Roles, a test with Patterns

Fernando Sérgio Barbosa
Escola Superior de Tecnologia de Castelo Branco
Avenida do Empresário
600-035 Castelo Branco, Portugal
fsergio@ipcb.pt

Ademar Aguiar
Faculdade de Engenharia da Universidade do Porto
Rua Dr. Roberto Frias
4200-465 Porto, Portugal
ademar.aguiar@fe.up.pt

ABSTRACT

Although roles have been around for a long time they have not yet reached mainstream programming languages. The variety of existing role models may be a limiting factor. We believe that for roles to be widely accepted they must enhance code reuse. An outcome would be a library of roles. We present and discuss what we feel are the characteristics that a role model must have to enable reusable and player-independent roles. In this paper we present our role model and JavaStage, a role language that extends Java, with examples of reusable roles. Finally, we present our steps towards the building of a role library, by presenting the roles developed from the analysis of the GoF Design Patterns. The results obtained, we developed roles for 10 of the 23 GoF patterns, are promising.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.13 [**Software Engineering**]: Reusable Software

General Terms

Design, Languages

Keywords

Roles, Modularity, Libraries, Design patterns

1. INTRODUCTION

Ever since the role concept has been introduced in software by Bachman and Daya [3, 2], the research has flourished and several role models have been proposed. But the definitions, modeling ways, examples and targets are often different [29, 9]. Many languages that support roles have emerged. These can be divided into extensions to already existing languages or completely new ones. Regardless of these efforts, roles have not reached mainstream languages and are not used by many programmers.

We believe that the several existing models may come as a factor in role non acceptance, but the main obstacle may be that programmers do not see role benefits in practice. The research on roles has focused largely on the dynamic nature of roles [11, 31, 4], modeling with roles [24, 25], and on modeling relationships [23, 17]. Dynamic role approaches focus on expanding an existing class to be used in a new context, so they develop roles for the classes that need to be expanded. This may explain why roles use a playedBy clause that states which class can play them. Because of this focus on the classes the reusability of roles has been neglected. We argue that this playedBy clause is limiting role reusability, because a role could be used by another, unrelated, class. But roles must know something of its intrinsic (object that it is attached to), like the set of methods it offers. We propose a different way of stating these restrictions on the role without linking it to a specific class. We will move the focus from the class to the role as we try to maximize its reuse. We believe that if developers can use roles to write less code then they will more easily adopt them.

A role provides a view of the object in a collaboration, so methods names must make sense in that context. In similar collaborations, however, where we could reuse the same role those names would probably be inadequate. A name of a method must clearly state its purpose but for a full purpose role those names are difficult to achieve. To overcome this we developed a renaming mechanism that allows an easy configuration so that the methods' names for a role may be tuned for a particular collaboration. The proposed renaming mechanism does not apply only to the role methods but also to the participants' methods needed by the role.

To achieve code reuse we'll use the principles of modularity as guidelines. Out intention is to develop roles that can be seen as modules, meaning they are truly generic and reusable. We argue that developing roles independently from their intrinsics is a key factor that will allow a role to be fully reusable. In order to express our approach we propose an extension to the Java language - JavaStage. We will present JavaStage with a few examples of generic, fully reusable roles, such as a Subject role from the Observer design pattern or a Singleton role from the Singleton design pattern [8].

If roles are reusable then it should be possible to build a library of roles. We started our role library with the analysis of the Gang of Four (GoF) design patterns [8]. Design Patterns are a good starting point because they are used in many frameworks and so represent a lot of real code. If we can create roles for these patterns then our approach is likely to have impact on many of today frameworks and applications. Results are promising as we developed roles for 10 of the 23 patterns. Even if some are limited in features others cover nearly all aspects of the pattern.

We can summarize our contributions with this paper as:

- a new role model, where the class states the role it plays instead of the role stating which classes play it
- a role language - JavaStage - that supports a powerful renaming mechanism
- development of a role library based on the GoF patterns

This paper is organized as follows. The next section is reserved for a brief description of roles. Section 3 is dedicated to the problem of building reusable roles. In section 4, we present some role characteristics and problems associated to roles and relate them to our approach. In section 5, we present our approach by presenting the JavaStage role model with a few examples. Design patterns were the starting point of our role library and we discuss our roles derived from them in Section 6. Related work is presented in section 7, and section 8 concludes the paper.

2. ROLES

To deal with the complexities of any problem we normally use abstractions. In software, abstractions try to model concepts present in the problem domain. In Object-Oriented (OO) languages classes are the usual abstraction mechanism. A concept can be specialized to a more specific concept. An animal can be specialized to a mammal or a reptile, or even to a quadruped or a biped, depending on the focus of analysis. This specialization is, in OO systems, modeled by inheritance.

But, as a simplification of the problem, OO modeling cannot capture all the dynamic present. OO systems are founded in the Aristotelian view of the world with ideas (classes) and phenomena (objects). Each phenomenon is a manifestation of an idea: a particular chair is a manifestation of the idea of a chair. Ideas and phenomena do not exist with one to one correspondence, though. A phenomena can be classified in several different ideas: 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 programming terms it is not possible to accommodate all possible views of a phenomenon with a single idea as we cannot foresee all its uses. Thus objects need to evolve overtime.

Steimann [29] states that Lodwick was the first to break with the Aristotelian vision according to which the nouns of a language govern its structure and meaning. 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. This calls for other modeling construct than classes. One that can model the roles some phenomena plays when inside a given context. One such construct are roles.

The role concept for software use is considered to have been first introduced in the work of Bachman and Daya in [3] and [2]. Sowa [28] introduced a distinction between natural types and role types. Natural types are related to the essence of an entity (a class in OO languages) and roles types are the characteristics that depend on an accidental relationship to other entities. This is a distinction that is still used today by many role models.

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 [14]. 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. Furthermore perspectives can change dynamically. This means that an object’s set of methods may have additive and subtractive properties [14]. Thus, roles allow objects to evolve over time.

If roles allow such evolution of objects a question may occur: are classes a superfluous concept or do roles need classes? An argument for classes and roles is that classes represent what is static and roles describe what is dynamic. A class defines an entity, while a role only refines an entity in a certain context. Another argument is that with both concepts separated we can have separated inheritance hierarchies of classes and roles [14].

3. REUSING ROLES

This section is dedicated to what we believe are the key factors that will enhance role reuse. Every role must have an intrinsic, or a player, that assumes the role. Independent evolution of roles and intrinsics must be ensured if we want roles to be reusable. Another factor is role configuration, so that a role can be tuned for a particular player class.

Independent evolution of roles and players calls for the use of modularity principles. If we treat a role as a module and the player as another module then we can strive for a greater independence between them.

It is our purpose to build reusable roles. Does this mean that previous role languages do not allow reusable roles to be built? No, but we believe that our approach takes role reusability a step forward. With previous proposals roles are bounded to a player and several restrictions apply because of their dynamic nature. By using roles as a static construct we can use them as a building block for classes and don’t have to deal with dynamic constraints (see section 4.3 for more details). By letting classes declare which roles they play we can have some configuration of the role to be made by the class, most notoriously method names and data types, and not force roles to adapt to their players.

3.1 Modularity

Modularization [20] is one of the most important concepts in software development. Breaking a system into modules allows the independent development of each module. This shortens the development time as each team may develop their assigned modules simultaneously. Independent development also enables the modification of a module, even a drastic one, without any change to other modules. The scaling property of modules allows building one module using other modules. Another advantage of modularization is comprehensibility because one can study the system one module at a time. There are numerous advantages of modularization like enhanced error tracing and fixing, reduced system compiling time, etc, but the one that we, as developers, treasure most is the high reusability of modules. This allows the development of libraries which in turn reduces the amount of code one must write in order to build a system and with extreme benefits in system reliability (assuming libraries have been thoroughly tested).

A module has an interface and an implementation. The implementation is the way the module is built. A key concept in modularization is encapsulation. When a module is well encapsulated changes in that module do not affect any other module. The interface defines how clients interact with the module. Since this is what clients see and use it shouldn't change much along the module life-cycle as clients must be aware of the changes and in turn change their implementation accordingly.

Modules interact with each other but some modules are more tightly connected than others. The intra-modules interactions are more intense than inter-modules interactions. Intra-modules interactions may require a specialized interface that other modules don't need, and shouldn't know of. To cope with this, most languages declare different levels of access to the modules members. In typed OO programming languages, where the smallest module is the class, there are, at least, 3 levels of access: private, protected and public.

3.1.1 Problems with modularization

Modularity is a very important concept but how do we decompose a system into modules? A definitive answer to this question is still missing. There are numerous decomposition techniques, each focusing on a particular view. The most successful one today is Object Oriented Decomposition where the system is decomposed into classes and objects. It is argued that a single decomposition strategy is not enough to adequately capture all the systems' details [32], and the quest for another, more universal, decomposition strategy is still going on.

A consequence of using a single decomposition strategy is the crosscutting concerns that appear when several modules must deal with the same problem, even if it is outside their main concern, because one cannot find a single module responsible for it in the light of that decomposition strategy. This leads to scattered, replicated code. The consequences of this are the exact opposite of the benefits of modularizations. Because a module must deal with a part of a problem that is spread by several others, changes to that code may affect other modules, and most times do. This affects independent development. Development and maintenance are a nuisance too because changes in the code needs to be done in all modules transversely.

There are several attempts to eliminate crosscutting concerns from OO programs, most notably Aspect Oriented Programming [12]. We will contribute with another view of the problem by using roles as a smaller module than the class. This way we can put the crosscutting concern in a role, or a set of roles, and the classes would play those roles. Any changes to the concern are limited to the roles thus greatly improving maintenance and reducing change propagation, or in other words, the crosscutting concerns become more modular.

3.1.2 Roles as modules

In order to develop roles independently from its players we must follow modularity rules and treat a role as one module and the player as another. Thus roles must provide an interface, ensure encapsulation and have to be developed independently from other modules. It can be argued that modularization is a matter of design and not a language feature [14] and that we can have modularization in languages that do not provide encapsulation. While this is true, language design can enhance the use of modularization and favor good programming practices. So in a language that strives for modularization these are desirable features and we followed them in our approach.

Providing an interface is straightforward if we allow roles to be first class entities in a language that supports them. Ensuring encapsulation and independent development raises a few issues. We must consider the fact that a role only makes sense when "played" by a class. Does this mean that the class playing the role has access to the role members and vice-versa? If this was so then the role could not be developed independently from the classes that play it, because any change in the role implementation could cause changes in the class. The same holds if we grant the role with access to the class members. Then changes in the class may force changes in the role. From this discussion we can see that roles and classes have to be independent and rely solely on interfaces. Because roles and classes have a special relationship, different from a super subclass relationship, it may be the case that we need a special role-class interface. For that we could use yet another access level, or redefine the meaning of the protected level to include the role-class relationship. We will discuss our option when we present our role model.

3.2 Configuration of a role

A role only makes sense in a collaboration so it must be tuned for that particular use. If we want to reuse roles them each role must be adaptable to different collaborations and contexts. We must therefore allow players classes to configure the role to their suit their specific needs.

3.2.1 The need for a renaming mechanism

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 where 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 specifies an interface instead of a class it is possible for the role can be played by different classes, but even this is not enough. A hindrance to this reuse is the name of the methods that are specific to an interaction. As an example, the Observer pattern [8] describes an interaction between subjects and observers that is present in many different systems with only minor changes, most notably the names of the methods to register an observer with a subject and the methods used by the subject to notify its observers. A Subject role for a MouseMotionListener instance of the pattern would define methods like addMouseMotionListener, or removeMouseMotionListener. That role could not be reused for an instance of the same pattern but for a

KeyListener which uses methods like addKeyListener or removeKeyListener. Because a method name must somehow indicate its functionality, using a generic name like addListener would not be correct as it would reduce the comprehensibility of the code. Another major drawback is that it would limit the class to play only one subject role. Considering this, a renaming mechanism would expand the reusability of the role to several situations. Of course some restrictions must apply, because a class that plays a role must ensure a specific interface, but that interface should be configurable, at least in what respects to methods names.

Some languages [31] use a "rename" clause that allow player classes to rename methods. If the role interface is big then this task is tedious and error prone. We need a more expedite way of doing this.

There is also the problem of the role calling the player methods. Again method names are important. In the Subject role each subject has a method that calls the observer's update method. In the Java-like implementation of the pattern such method is not called update but, instead, are used several methods like mousePressed, mouseReleased, etc. The "rename" clause is not usable here because the number of methods that get called varies between instances of the subject role.

We need a mechanism that allows fast renaming for both role methods and methods that are called by the role.

3.2.2 Configuring data types

Sometimes the role may perform the same actions in different contexts but differ on the data types used in each particular context. For example the Subject role uses MouseListener or KeyListener objects. To cope with this, roles must provide a way to configure data types, if the language does not handle it. For example, C++ supports type configuration using templates and Java by using generics.

3.3 Summary

To summarize, for roles to be fully reusable then they must:

- provide an interface
- ensure encapsulation
- have to be developed independently from its players
- provide a method renaming mechanism that enables the role to be played by any class that fulfills some requisites
- Configure data types

4. SUPPORTING ROLES

There are several languages that support roles, and many role models. Each one chooses a set of role characteristics that suits its needs. A complete description of roles characteristics is therefore lacking if we take a language only. This section tries to present some properties of roles and practical considerations and was based on [9]. We will discuss our role model using the guidelines of this section.

4.1 Classes playing roles

A class may have restricted role-playing capabilities ranging from the number of roles and the type of roles it can play. In some languages a class must have at least one role attached, such as Fibonnaci [1] where a class is a null-object. A class may also play a fixed number of roles but it is not common. More normally a class may play an undetermined number of roles. Pernicci [22] defines a class as consisting in one to many role descriptions, where the first role description is equivalent to a "normal" class.

Even if a class may play several roles it may impose restrictions on the roles it can play. Playing the same role more than once is prohibited in languages that rely on role type for method dispatching. Other languages like Object-Teams [11] allow a class to play the same role several times but in different contexts.

A class can have conjunctive or disjunctive attachment of roles. In conjunctive attachment roles are like wrappers that cover the intrinsic (attached object). Attaching another role means wrapping the previous role with the new role. In disjunctive attachment, a role may be attached directly to the object even if it already plays other roles. Figure 1 shows the two kinds of attachment.

Classes conceptually cannot inherit from roles [15]. Roles are usually written for a class, or its subclasses. Abstract roles are not written for a particular class, but designed to be the superrole of a more specific, class-targeted role.

Typically, roles can be attached to only one object, but some use multiroles: roles that can be attached to several objects.

Visibility of roles is also important. Normally roles are public because roles usually model object interactions but there can also be private, and protected, roles. Here the notion of private can vary: a role can be private to the object it is attached to or private to the class that defines it, if it is defined inside a class.

Limits on the number of role instances can also apply. An instance of a role can also be attached only to a single object but several instances of the same role can be attached to the same object (if a class can play multiple times the same role). Then there is the possibility of a role having a single instance that when attached to a new object it is automatically detached from the previous object.

In our approach a class imposes no limit on the number of roles it plays nor on the number of times it plays the same role. However, due to its static characteristics, it does not support multiroles and the attachment is done on a class by class basis rather than on object by object basis.

4.2 Roles playing roles

A role may also play other roles. The same thought for classes apply: roles can play no roles, play a fixed or an unlimited number of roles. Like classes the type of roles a role can play may vary. We do support roles to play other

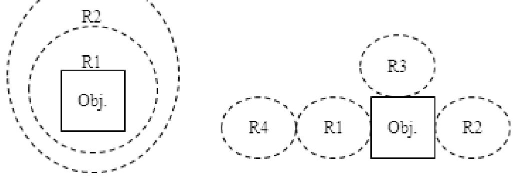

Figure 1: Examples of Conjunctive role attachment (left) and Disjunctive role attachment (right)

roles and the same restrictions for classes apply.

4.3 Role dynamics

Roles can be taken on by the object or imposed by the outside. When a role is imposed it is automatically attached when the object enters a specific context. Role movements reflect the dynamicity of the real world, but it poses many practical problems. The major problems with role movement are the dynamic situations that may arise.

The example in Kristensen and Osterbye [16] refers to the role Mayor. When a person is elected Mayor it assumes the role and when another person is elected the role gets transferred. But if we move the role then the person that played the role of mayor does not recall being a mayor. It is possible that other clients knew the person as a mayor and after the move they now know the new mayor and don't remember the other person. There are cases in which only the functionality of the role is important and not who plays it. In these cases the role may be moved freely without the previous problems (it may have others, though).

Moving a role must also take into account if a role is currently executing or not. If the role is not executing then it may be safe to move it. If roles have identity then clients may have the knowledge that a role is attached to a specific object and moving the role may invalidate that knowledge. The same can be true if a client does not know the specific object but depends on a particular state of that object, when moving a role to other object the state may be changed without the client knowing it. Kniesel [13] presents other reasons to invalidate roles from moving. If a role is executing, moving it it raises a all new kind of problems. When the role resumes its execution the environment has changed! Some solutions involve letting the role finish its execution and then perform the movement, but that may not always be possible (it may even lead to deadlocks).

Moving roles must consider the fact that roles may play other roles. Thus when moving a role it makes sense to move along all the roles it plays. But that may be untrue if the target intrinsic have restrictions on the roles it may play. It may play role r1, but may not play role r2. If role r2 is attached to role r1 then the move fails. What to do? Not move role r1 or detach role r2 from r1 and then perform the move? The correct way depends on the concrete case, even though a default behaviour can be enforced. Either way new problems arise. Detaching role r2 may not be possible or if it is detached should it be replaced by another role?

When we move a role all the references to that role must be updated. A notification method should also be called to prevent the problems mentioned when a client depends on a specific object or specific object state.

Because of the many problems involved many languages forbid the movement of roles. Once a role has been attached it cannot move. This increases code predictability [7].

A similar problem with moving roles is removing roles. The main problem is the past being removed with the role. When a student ceases to be a student it has nevertheless a record of grades, etc. When the role is detached it is as if the object never has been a student. There is also the fact that clients have the references to the role. The school still has references to the student's role. It may depend on the role to know the name of the student. If the role is detached then the school loses this information.

A solution for role removing is role replacement. We could replace the student role with a graduate role that stores the record information. But that is done on a case by case scenario. There are also all the problems with role movement to be dealt with, like removing a role when it is executing and so forth. Another solution is not to allow removal of roles. When attached it remains attached and only terminates when the object terminates: it is a life role.

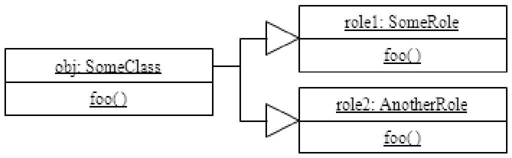

Figure 2: An object playing two roles with overlapping members

Solutions for dynamic situations are very difficult to obtain because there are infinite possibilities. Since we are focusing on code reuse and class composition in a static context we dismissed all role dynamics. This means that when an object is created all its role instances are created and attached to it and are destroyed when the object is destroyed.

4.4 Subtypes or supertypes

Controversy exists whether roles are considered as subtypes or supertypes of its intrinsic. We can see why in the following paradox: at compile time a role may be considered a supertype because its concept is wider than that of its intrinsic (a costumer may be played by a person or a company) but at run-time the opposite holds (not every person plays the role of costumer). The subtype view is the more consensual view, though.

Another view states that roles are an unrelated type of their intrinsic. This is the view adopted by many role models that are implemented as design patterns. This is the view we adopted: a role only adds to the class interface and a role instance does not replace the class instance in any situation or vice-versa.

A role could extend a class, even though they are conceptually different (due to our role configuration techniques we do not allow that). Roles can extend other roles but some restrictions apply. If a role is an abstract role there are no restrictions. If a role has an intended intrinsic this restricts the subrole as the subrole intrinsic specification cannot be of a reduced type compared to the superrole intrinsic. Here reduced is used in the sense that the type cannot have fewer properties. A subrole may, on the other hand, extend the intrinsic specification.

4.5 Defining properties in roles

If we allow roles to have fields and methods this will eventually lead to name collisions. This happens if an object has two attached roles with the same methods (or fields) like that shown on Figure 2. This is bound to happen if the same role is attached more than once to the same object. If this is an intended procedure than bookkeeping of the fields must be ensured. If it is not intended then accidental attachment of the same role has to be prevented. The solution to these problems is language specific and even role specific.

With methods there is another question: the role methods override the object methods or vice-versa? The answer may depend on whether the relation between the role and intrinsic is of a super or sub type nature. It may also depend on the language: C++ allows only the overriding of virtual methods, in Java all methods may be overridden, except those declared final.

4.6 Method call

When entity A sends a message to some entity B what happens? The answer depends on the role model being used. First it must be known to which entity A is referring to. If A is referencing a role (as in most role languages) then A is calling a method in the role. If A is referencing the object then the method must be dispatched to the role by some mean. This can take a very complicated route because a role may be playing another role.

What happens when several roles define the same method? Take for instance, the example shown in Figure 2: when we call `foo()` on object `obj` which methods gets called as there are 3 possibilities? Some languages call all the methods, others have some mechanism to select one of them. The problem with calling all the methods is the return value. ObjectTeams [11] for example leaves the result undefined. Some mechanisms select the last attached role, we, for instance, select the first declared role. Others techniques force the caller to disambiguate the call using casts. This, of course, forces the obj client to know in advance the roles that object plays which can be considered a break in the class encapsulation. In the given example we would use one of the following lines, depending on which version to call:

```
((SomeClass)obj).foo();
((SomeRole)obj).foo();
((AnotherRole)obj).foo();
```

There are other considerations such as: may a role call its intrinsic methods or vice versa? Can a role place calls on other roles? What if a role and an intrinsic have the same methods? Note that this situation is different from a method called from an external entity. Some languages define a reference self to contrast with the this reference. The this reference indicates the intrinsic while self references the role, thus disambiguating the call (we use a performer reference with the same meaning but for other purposes). Some impose a restriction on the intrinsic calling its role's methods. It does assume that the intrinsic knows its roles and may depend on them. In dynamic situations this cannot be allowed as the role may not be there. An alternative way is for the intrinsic to query its roles for the presence of a particular one and then call its methods. In a static context the role may be assumed to be there so this does not really pose a question in our role model.

A recurrent question, still unanswered, is role identity. Some argue that roles have no identity [15, 16]. Considering that roles are not independent entities then roles should not have an identity. The object and its roles are referenced as one. If roles can evolve objects they cannot modify its identity otherwise it ceases to be the same object. To identify the role to be addressed casts are used. One example of such an approach is the previous code. This, however, forbids multiple roles of the same type.

Others argue that roles do have an identity that is different from its intrinsic [18, 33]. When a role has identity the class can have multiple instances of the same role and distinguish each one. If the role identity is independent of the intrinsic it may lead to an equivalent of object schizophrenia when a class plays several roles or is engaged in several collaborations. To solve the problem of the foo() method we would use one of the following lines, depending on which version to call:

```
obj.foo();
obj.role1.foo();
obj.role2.foo();
```

There are also those, like we, to whom roles share the same identity with its intrinsic and also have one that distinguishes it from other roles present in its extent [27]. This usually calls for the use of methods to compare identities in dynamic contexts.

4.7 Mixin roles and class members

A role models a particular view of an object so it must have either state or behavior or both, otherwise they would be comparable to interfaces [30]. But if roles have state and behavior then how do they relate to the state and methods of the intrinsic? If roles add their properties to the intrinsic properties other questions like name collisions, method calling, etc, arises.

4.7.1 Name collisions

Name collision between role fields and player fields may occur. If an intrinsic is allowed to play the same role multiple times then name collision is inevitable.

Member name collision is resolved if roles are allowed to have identity (whether or not independent). Inside an intrinsic, to access a role member we must precede it by the role identity, otherwise a class member is used. Inside the role, the member is assumed to be the role member (as in the section 4.6). If the role needs to disambiguate between an intrinsic member or a role member it must use a this or self/performer reference in those cases that support it.

Playing the same role multiple times is also resolved when roles have identity, because it suffices to use the correct identity to select which version of the role to use.

4.7.2 Method calling

Method calls are different from field access. Method calls are made by the intrinsic clients that, in most cases, are unaware of the roles the object is playing. Are role methods added to the intrinsic interface or accessed via a role reference? If accessed by a role reference then clients must know about the roles that the object plays. If they are added to the intrinsic interface then clients may be unaware of the roles. Forcing clients to use the role identity should be kept to a minimum because the less the client knows about the class and the roles it plays the better it will respond to changes in that class. An exception can be made when an object plays the same role multiple times, then the clients need to specify which role to use.

4.7.3 Method Overriding

If a class and roles methods are identical then do roles methods override or are overridden by class methods? Allowing role methods to override class methods means that a role can alter the normal behavior of an object, which is what dynamic roles are about. But a class may want to play a role and yet retain some aspects of its own behavior. We must also distinguish between a mandatory implementation for a role method of a default implementation, very much like in inheritance hierarchies (final methods in java, no virtual functions in C++). In our role model role methods are added to the class interface but are overridden by class methods.

5. JAVASTAGE ROLE MODEL

When we started developing our role model for JavaStage the main goal was to enhance code reuse while maintaining the model as close as possible to OO decomposition. 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.

Our view of roles is somewhat different from others as we focus more on the static nature of roles, as used by [25], rather than on its dynamic nature as seen in PowerJava [4] and ObjectTeams [11]. We also tried to stick as close to the OO model as possible and the syntax additions at the minimum. We, therefore, dismissed the ObjectTeams' Context concept and the PowerJava's Institution concept as they introduce greater complexity to the model.

An advantage of the Context concept is the contextualization of the interactions. We can argue that, for example, the observer's subject role is useful only inside a subject-observer relationship and in the Context both roles are objectively identified and without the Context the role of the observer is somewhat abstract. It also states that an object only plays a role inside that context, and whenever it enters the context it gets the role attached and when leaving the context the roles is detached.

Without a Context we are forced to check which objects play the role of the observer and when that interaction takes place. But contextualization comes with a complex design that strays from the OO path, making it harder to understand. Also the Context concept is useful in dynamics environments that we do not address. In our static environment its advantages are diminished, it only has contextualization, but still retains its complexity. We believe that the small changes we introduced in the OO decomposition will be better accepted by the developer community than a completely new concept. The small learning curve aims also to be an advantage towards that acceptation.

Our goal is to make roles reusable thus contributing to diminish code replication. To validate our approach we show, in Section 6, that the development of a role library is possible using our role model.

If we want 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. Java was adopted because it is a widely used language and because Sun's Java compiler code is open source thus facilitating changes to the language. We named our extension JavaStage.

We will present our role model based on the discussion from the two previous sections and with the presentation of some examples. Some points of the discussion in section 4 do not apply to our model because they deal with problems of role dynamics and we deal only with the static nature of roles as seen in [25]. To support role dynamics we would need to complicate the model and introduce new concepts like contexts (as in [4] and [11]) which was not our goal.

5.1 Roles as Modules

JavaStage supports roles as a first class identity. The syntax to declare a role is similar to that of a class. Roles can declare fields and methods like a class. When we want a class to play a specific role we use the plays directive. This is contrary to the general rule where it is the role that states which classes plays it. As discussed in section 3, this is restricting the reuse of the role, so in JavaStage it is the class that states the roles it plays.

```
public role Singleton {
    requires Performer implements Performer();


    private Performer single = null;

    public Performer getInstance( ){
        if( single == null )
            single = new Performer();
        return single;
    }
}

class ASingleton {
   plays Singleton rSingle;

   roleprotected ASingleton() { ... }
}
```

Figure 3: Code of the singleton role (adapted) and a player class.

In JavaStage all access levels of Java are supported and an additional one was added: roleprotected. The roleprotected level gives access to the role members by the player and to the player's members by the role. The protected level could be used but we chose to introduce a new access level to clearly state the player-role interface.

Figure 3 shows a Singleton role (adapted from our library) and a class playing that role. The Singleton role is reusable. Because the singleton role must call the player's constructor we need to declare the constructor as roleprotected. If we used the protected level then subclasses could call the superclass constructor and thus break the Singleton pattern. The roleprotected level, however, allows other roles to create instances of the class also breaking the Singleton pattern. The difference between the two approaches is that a class knows nothing about subclasses and cannot control subclassing (apart from final classes) whereas in JavaStage it is the class that states which roles it plays. Therefore, a class may choose not to play a role if it breaks its singleton nature.

In JavaStage roles provide an interface, have an implementation and provide encapsulation. Roles and classes are thus completely independent modules and may be independently developed.

5.2 Role identity

In our approach a role has an identity. The identity is given by the player in the plays clause. That identity is associated with the player identity. As can be seen in Figure 3 the identity of the role inside the ASingleton player is rSingle. Whenever the player wants to access role fields, or methods, it must use the syntax

```
rSingle.someRoleMember
```

If the role is public then that role identity is accessed just like any class member as shown below:

```
ASingleton s = ASingleton.getInstance();
s.rSingle.someRoleMember;
```

This allows clients, which have knowledge of the roles the class plays, to select the role they want. This is useful when the class plays several roles that have overlapping methods.

5.3 Mixin roles and class members

When a class plays a role the role methods are added to the class interface. If there is a name collision between role and class methods then the class methods are used if defined in that class. Inherited methods are overridden by role methods.

A new rule must be added because a class may play two, or more, roles that have overlapping methods. In such a case, the methods added to the class interface are those of the first role, following the sequence of the play clauses. This is, of course, the default action, but developers may change it simply by redefining the method and explicitly call the intended role method.

Our approach allows roles to have state and behavior. Name collisions between different role members and between role members and player members are resolved by using role identity.

To access its player members the role uses the `performer` keyword. Each instance of a role has a performer reference that points to the player instance. To represent the type of the performer, inside the role, JavaStage declares a `Performer` type. Because we don't know that type in advance this is used as a placeholder. It is used for storing references to objects or objects used as a parameter in some method. We could use a generic for this purpose but then the class would have to configure the role with its own type and that feels unnecessary. It is also used in the method renaming strategy discussed below when we state that the performer must implement some method.

5.4 Method renaming strategy

A name of a method must clearly state the purpose of the method but when creating a full purpose role those names are difficult to achieve. See the discussion in section 3 for details. To overcome this restricting factor we developed a renaming method that allows an easy configuration of the names. Each name may have two parts: a configurable one and a fixed one. The configurable part is bounded by a # as shown next.

```
fixed#configurable#fixed
```

The fixed part is optional thus leaving the name of a method to be fully configurable by the class. The configuration of the name is done by the class that plays the role in the plays clause as shown in Figure 4.

Figure 4 shows excerpts of the Container role that defines the role of a container and two classes that play that role: Company and PolyLine. Companies must keep track of workers, and allow the addition and removal of workers. Polylines are composed of points so a PolyLine needs to allow adding and removing points.

An advantage of this renaming strategy is the possibility to rename several methods with one single clause, as seen in the Container example and illustrated in Figure 5 which represents the interface that a client of the Company class

```
public role Container<T> {
    private List<T> content;

    public Container( ){...}
    public Container( List<T> store ){...}
    public void add#Thing#( T t ){ ... }
    public void remove#Thing#( T t ){ ... }
    public void remove#Thing#( int idx ){...}
    public boolean contains#Thing#( T t ) {...}
    public void clear#Thing#s( ){ ... }
    public T get#Thing#( int idx ){ ... }
    public Collection<T> getContent(){ ... }
}

class Company {
   plays Container<Worker>(
        Thing = Worker
     ) workers;
}

class PolyLine {
   plays Container<Point>(
       Thing = Point
     ) points;
}
```

Figure 4: Excerpt of the Container role and two classes playing that role.

sees. With a rename clause we would need several lines of code to achieve the same effect.

A role may also call player methods or any other entity methods. In order to maximize role reuse we cannot force players to name the methods as stated by the role. Taking the example in section 3.2.1, a MouseListener subject may call mousePressed or mouseReleased methods on the listeners. Such names are useful only in the MouseListener context. A Subject role that is to be reused cannot make assumptions on the methods called. Callback methods are also to avoid because they are awkward to use and need some amount of configuration.

The proposed solution to the called method names is to extend the configurable method renaming strategy to include these methods. Thus we can write a role method like

```
public void doSomething( ) {
   performer.do#action#( );
}
```

The method name is the only thing that can be made configurable, method parameters and return type cannot, because the method's signature must be previously known to

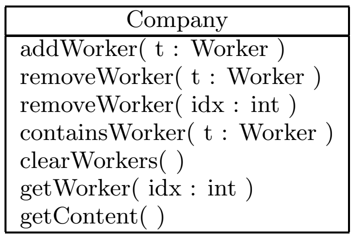

Figure 5: The interface of the Company class as seen from its clients

```
public role Subject<ObserverType,EventType> extends Container<ObserverType> {
  requires ObserverType implements void #Event.notify#( EventType event );

  public void fire#Event#( EventType e ){
    for( Iterator<ObserverType> i = getContent().iterator(); i.hasNext();  )
      i.next().#Event.notify#( e );
  }
}

public class SomeClass {
  plays Subject<MouseListener, MouseEvent>(
              Thing = MouseListener,
              Event = MousePressed,    Event.notify = mousePressed,
              Event = MouseReleased,   Event.notify = mouseReleased
             ) mouseListenerRole;

  plays Subject<MouseMotionListener, MouseEvent>(
              Thing = MouseMotionListener,
              Event = MouseDragged,    Event.notify = mouseDragged,
              Event = MouseMoved,      Event.notify = mouseMoved
             ) mouseMotionListenerRole;
}
```

Figure 6: The Observer pattern Subject role and an excerpt of a class playing that role twice.

statically compile the role. In some situations the method signature, apart from the name, could be inferred, but we cannot guarantee that for all cases (most notably for return types). We need, therefore, some way to state the required method signature. We propose to use a requires statement in which is stated who must supply the method, its return type, configurable name and parameters. The supplier must be the player (using the Performer type) or a generic type. Other types are not allowed: if we know the type we know its methods and do not need to configure them. Method renaming is valid only for those types that we do not know in advance.

The requires statement has the following form:

```
requires implementer implements
  returnType config#name#( params );
```

We added yet another feature to method renaming: the possibility to declare multiple versions of a method. For this we just have to define multiple times the configurable name in the class. For example if the Company class declared the Container role with

```
plays Container<Worker>(
         Thing = Worker,
         Thing = Employee ) workers;
```

Then it would have addWorker and addEmployee methods. The same applies to the other methods as well.

This feature is even more powerful if used together with the feature of calling methods. Then with just a name definition we can have several method versions. For this to work 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. The syntax is as follows:

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

In a player if we used the configurations

```
Method = Foo,  Method.inner = Foo,
Method = Bar,  Method.inner = Bar
```

then the player would have the roleFoo and roleBar methods, and each would call calledFoo and calledBar respectively.

With these features we can write the Subject role shown in Figure 6. The figure also shows a class playing the roles of MouseMotionListener and MouseListener. Configuring both roles is a simple case of renaming strategies. The code in Figure 6 would be smaller had we written a MouseListenerSubject role and a MouseMotionListenerSubject role based on the Subject role. Those roles would play the Subject role using the same configurations used in SomeClass. These roles could then be reused by all classes that wanted to play them instead of redefining the Subject role again and again. We left them like this for demonstration purposes.

There is one more feature in our renaming strategy: the use of the class directive. When class is used as a configurable part of a role method name it will be automatically replaced by the name of the player class when mixed in the class code. This is useful in inheritance hierarchies because we simply need to place the plays clause in the superclass and each subclass gets automatically 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 Element must provide an accept method that takes a Visitor as an argument. The Visitor must declare a visit method for each Element type. Each Element then calls a specific method on the Visitor. The Visitor's methods usually follow a name convention in the form of visitElementType. We used that property in our VisitorElement role, as shown in Figure 7. This example shows how it could be used in a Figure hierarchy where figures are the Elements. It also shows that the Figure subclasses don't have any pattern code, because they will get automatically an acceptVisitor method that will call the correct visit method.

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

  public void accept#visitor#( VisitorType v ){
    v.visit#visitor.class#( performer );
  }
}

public class Figure {
plays VisitorElement<FigureVisitor>
          ( visitor = Visitor ) visit;
  // ... rest of class code
}

public class LineFigure extends Figure {
    // no Visitor pattern code
}

  public interface FigureVisitor {
      void visitFigure( Figure f );
      void visitLineFigure( LineFigure f );
      void visitTextFigure( TextFigure f );
}
```

Figure 7: The VisitorElement role, a class Figure that plays the role, a subclass from the Figure hierarchy and a Visitor.

5.5 Roles and multiple inheritance

Static roles can be used as a way to overcome multiple inheritance problems or a way of supporting multiple inheritance in languages that do not support it. Problems with multiple inheritance 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 like Java. In Java, when we need a class to be part of two or more hierarchies we must use Interfaces. Interfaces can only declare constants and methods signatures and cannot have state or default method implementations. This may result in the duplication of a default implementation in classes that implements the same interface while inheriting from different superclasses.

We can use roles to provide a default implementation for any class hierarchy. Therefore to emulate multiple inheritance a class would implement the various interfaces and play the role with the default implementation for each interface. There are no ambiguities between which methods to call as JavaStage uses the sequence of the plays clause to generate the class methods. If some method other than the provided by JavaStage is intended the role identity can be used to call the appropriate method. This can be done by the class itself or by the client if it knows about the roles.

In Java it is common to start a inheritance hierarchy with an interface and then a superclass that provides the default behavior for that hierarchy. We argue that the default implementation should be provided by a role and then the default class would play that role. This way we can reuse the basic behavior whenever there is a need to, thus preventing a possible use of multiple inheritance as seen from the previous discussion. This proposal is depicted in Figure 8. In the example there is a Figure hierarchy that has an interface and a role as top elements. The DefaultFigure class implements the interface and plays the role. All its subclasses inherit this default behavior. 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 behavior every DefaultFigure subclass has.

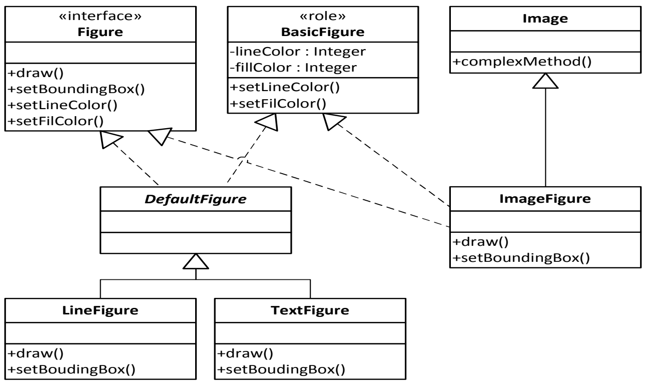

Figure 8: Example of a Figure hierarchy with both an interface and a role as top elements

5.6 Other issues

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 since roles methods are added to it. There are in fact several reasons for not doing so. One is the role identity which, purposely, resembles an object declaration. Yet another reason is the naming configuration, which would clutter that declaration. A final reason is that roles may have non default constructors and we need a way to pass parameters to them. It would appear awkward to do that in an implements-like declaration. Role initialization is sometimes required. For example, the Container role from Figure 5 uses an ArrayList by default, but with a constructor we can let the player choose which container to use. The use of constructors is supported by our approach. The syntax is:

```
plays role( nameConfiguration )
      roleID( roleConstructorParams );
```

Roles and interfaces are somewhat related [30] so we could let roles define a type and hence have the polymorphic behavior interfaces have. We could write code that would work with any class that plays that role. Our method renaming strategy, however, forbids this because the actual interface a role provides is configured by the class and not by the role itself. Roles with no configuration could in fact define a type but its implications have not been studied yet and are relegated to future work.

5.7 The inversion of the play declaration

Like previously stated, in traditional approaches the role declares its players. We did it the other way around: the class declares which roles it plays.

One reason to use the "played by" clause in the role is to impose a constraint on the classes that may play that role, to ensure that the player has a certain interface (or structure) that the role uses. 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 as explained in 3.2.1, when we account for methods names this is a limiting factor. The inversion of declaration from the role to the class allowed us to provide a renaming mechanism that configures the role as it should be used by the class. The role must, nevertheless, impose some restrictions on the player interface when it needs to communicate with it. In our case those restrictions are imposed via the ”requires” statement. This statement also allows us to impose restrictions not only in the player itself but also on other objects that are part of the interaction. For example, in our subject role for the observer pattern, we can configure which interface the observer must implement.

Another reason for the ”played by” clause to be used inside the role is the dynamic nature of many role languages. In a dynamic context the class cannot declare which roles it plays because that is known only at run-time in an object by object basis. Since we are in a static context we know which roles the class has to play throughout the system and this is not a problem. We do not intend to support a dynamic context, as our goal is code reuse, but our approach is not a limiting factor in this situation either. With a Context-like concept we could easily make the role configuration in the context. This way classes have no knowledge of roles and vice-versa, leaving the coupling to be made by the Context. As mentioned in the begging of section 5 the Context concept has many advantages in dynamic situations and this is another one.

In static contexts it is also complex for the ”playedBy” clause to be used in a role instead of a class. It means that all the classes that fulfill the ”playedBy” clause get that role attached, even if they do not want to play it? That would need the class to do some configuration. To deal with the problem of playing the same role more than once we would need to let the class do some configuration too. So why not let the class configure everything? This way the configuration is done in a specific place. We believe that inverting the play clause is a much simpler and expedite way of configuring a role.

6. TOWARDS A LIBRARY OF ROLES

In order to prove the validity of our approach we developed a compiler for the JavaStage specifications, based on the javac compiler. The ultimate goal is to develop a library of roles with this tool.

Our library starting point was the analysis of the 23 GoF patterns [8]. Design Patterns are a good starting point because they are used in many frameworks and so represent a lot of real world code. If we can create roles for these patterns this means our approach is likely to have impact on many of today frameworks and applications.

6.1 Roles in Design Patterns

Each pattern defines a number of participants that collaborate with each other to carry out their responsibilities. Some participants in these patterns can be seen as roles while others cannot. This distinction is made in [10] by considering the roles superimposed or defining, even though their concept of roles differs somewhat from ours.

A defining role is a role that completely defines the class, that is, the class has no other concern besides its participation in the pattern. Such an example is the State hierarchy in the State pattern. In this pattern there is an object - the Context - that alters its behavior when its internal state changes. The pattern proposes the State, that is a class (or an interface) that defines the behavior associated with a particular Context. Each possible state is then implemented by a subclass (or a class implementing the interface). The Context object has a state object into which it delegates state-specific requests. When the Context object changes state it actually changes the state object. Each state subclass has no other concern than performing the actions the object forwards to it when in that state.

A superimposed role is a role that is assigned to classes that have other concerns outside their participation in the pattern. In the Chain of Responsibility pattern, for example, the Handler role is superimposed in every participant that is a link in the chain. It has to either handle a request or forward it to its successor. This behavior is not the main concern of the class but is has to perform these actions just because it is part of a Chain of Responsibility.

For each pattern we took the roles played by each participant and focused on similar code between instances of the pattern to find reusable code. A goal we expected to reach with our approach is a better modularization in some of the patterns.

We present our results by groups of patterns. The patterns were grouped by the similarities between implementation or between similar problems presented. We also grouped the ones already described in the latter sections.

For those patterns for which we develop a role we also implemented a sample scenario that illustrated its use, but those samples will not be discussed here, except the sample for the Observer pattern that will be used as an indicator of role player independency. The results presented here show that it is possible to build reusable roles that are independent from their players.

6.1.1 Builder, Façade, Interpreter, Iterator, Mediator, Template Method

These patterns showed no common code between instances. That is because they are highly dependent on the nature of the problem. For example, the way a language is to be interpreted depends only on the language itself, so we need a purposely built interpreter for each language. An iterator is developed for a concrete aggregate and every aggregate has a unique way to traverse.

A special case is Mediator: even if it is developed for a particular problem it usually uses the Observer pattern as a way to connect Colleagues to the Mediator, so the Subject role would be reused here. Nevertheless no similar code between instances was found.

6.1.2 Abstract Factory, Adapter, Bridge, Decorator, Command, Strategy

The code for each of these patterns is very similar between instances but, nevertheless, we could not write a role for any of them. For example, many abstract factories simply have methods that are basically a return statement and the creation of an object. However the way the object is created and its type are specific to that method alone. The Adapter pattern instances are also similar in the way that the Adapter forwards calls to the Adaptee, but when to make such calls and, even more limiting to our role approach, the call parameters and return types vary with each instance, and even with each adapted method. Roles could be used in this pattern as a way to emulate multiple inheritance as shown in Figure 8. A common feature between instances of the Adapter, Decorator and some Strategy instances is that a reference to the Adaptee (or ConcreteComponent or selected Strategy) is held by the Decorator (or Decorator or Context) and sometimes that reference needs to change. We could write a role that provides such reference management but its practical usefulness is very low: with current IDE development we just need to declare the reference and automatically generate the corresponding setters and getters.

6.1.3 Flyweight, Proxy, State

The roles developed for these patterns are limited in their actions and are basically management methods. Even so we find these roles useful because they provide the basic behavior for the pattern and the developers need to focus only in the specifics of their instance.

Flyweight depends on small sharable objects that the client manipulates and on a factory of flyweights that creates, manages and assures the sharing of the flyweights. The concrete flyweights and their interface are instance dependent and are not open to reusability between instances. Many flyweight factories share a common behavior: verify if the required flyweight already exists and, if so, return it or, otherwise, create it, store, and then return it. We developed a role for the flyweight factory that relies on a map to manage the flyweights and also supplies the management method. The flyweight creation method is the only instance specific method the factory needs so we require the player to supply such a method. The types of the flyweights are defined using generics and the methods names are configured using the rename strategy.

Proxy provides a surrogate or placeholder for another object to control access to it. The real subject is placed inside one object, the Proxy, which controls access to it. Some operations are dealt by the proxy itself, while others are forwarded to the subject. Which methods are forwarded or handled by the proxy are instance dependent as is the creation of the subject, but the forwarding mechanism and checking if the subject is created or accessible is somewhat similar between instances. We developed a proxy role that provides this similar behavior. It stores the reference to the subject and provides the method to check the existence of the subject and triggers its creation otherwise.

The State pattern allows an object to alter its behavior when its internal state changes. There are almost no similarities in this pattern because each instance is specially made for the task at hand. The only similarity between several instances is the state change mechanism, which can be made in one method. We therefore developed a role that is responsible for the state transitions and keeping the current state. The state change method terminates the actual state before updating to, and starting, the new state. The name of the state change method and the state terminating and starting methods are configured by the rename feature, while the state class is configured by generics.

6.1.4 Factory Method, Prototype

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

The Factory Method defines an interface for creating an object, but let subclasses decide which class to instantiate. The normal implementation of this pattern is clearly instance dependent and provides no common code between instances. There is, however, a variation whose purpose is to connect parallel class hierarchies. In this variation each class belonging to a hierarchy delegates some responsibilities to a corresponding class belonging to another hierarchy. For this variation there is one similarity between instances that we can explore: each class has a method that creates the corresponding object. What we did was to put the creation of the product in a creator class. The creator class provides methods that create the required product, one method for each product. Then the classes just have to call the corresponding method in the creator. An advantage of this solution is the modularization of the pattern in which the correspondence between classes is made in a separate class rather than on a class by class basis. Future additions and changes are made in this class only. Because the creation process is now delegated to a single class this means that as an extra advantage we can change the creator dynamically.

We developed a role that allows the specification of the factory method that creates the object of the corresponding class. The method uses the class directive in the renaming feature which allows the plays reference to be made only in the top class of the hierarchy. On the other hand the use of the class directive implies that the creator must rely on method naming conventions. This is, however, a small price to pay for the extra modularity gained.

In a sample implementation we simulated a Figure hierarchy where each figure has a specific manipulator. The Figure class plays the FactoryMethod role where it defines that the product created is of the type FigureManipulator and the creator is a ManipulatorCreator object. Only the Figure class has the plays declaration, as stated above, but each subclass has it own createManipulator method that redirects the call to the ManipulatorCreator. This is the class responsible for the creation of the correct Manipulator for each subclass. As said above, if we wish to change the way manipulators are created or even which manipulator is created for each subclass we do it in this class alone and we need not to change the Figure subclasses. Figure 9 illustrates this example.

For FactoryMethod we also developed a role that has a fixed Creator instead of a dynamic one for those cases where this extra functionality isn't required.

The Prototype pattern specifies the kind of objects to create using a prototypical instance, and creates new objects by copying this prototype. This pattern relies on the prototype class to have a clone method that produces an identical copy of the object. While every class has its own mechanism for cloning its objects it may not be sufficient because of the deep copy vs shallow copy problem. The clone method may do just a shallow copy where a deep copy is needed, or vice-versa. If the client, when it really matters, could choose how the copy is made it would be more pertinent. For this we developed a role that moves the creation of the copy to another class. This mechanism is similar to the one used in our solution to the FactoryMethod pattern role. This means that the new class is responsible for creating the copies of all

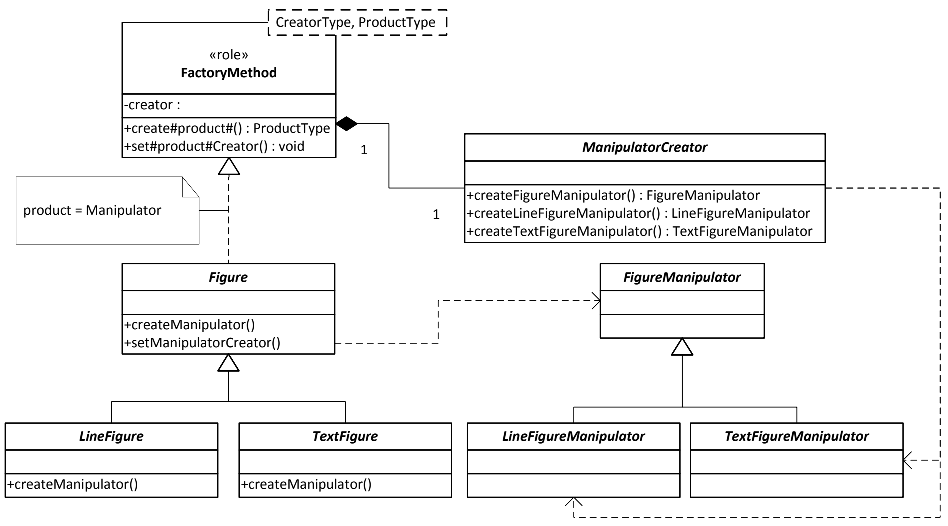

Figure 9: The use of the FactoryMethod role to relate a Figure subclass to the corresponding FigureManipulator

classes that may be used as prototypes and thus may choose how to make the copy. Because it uses the class directive the subclasses of Prototype don't need to declare the clone method.

6.1.5 Singleton, Composite, Observer, Visitor

These patterns were discussed along the presentation of the role model, except for Composite, which reuses the Container role already presented.

The Composite pattern composes objects into tree structures to represent part-whole hierarchies. Each composite must maintain a collection of child components and implement the operations defined by the component hierarchy. There are common operations for all Composite instances: the management of the children (like addChild, removeChild, ...). The operations defined by the component hierarchy are instance dependent and are not suitable for generalization, even thought most of their implementations is the traversal of the children collection and performing the corresponding operation on each child. A map function, popular within functional programming, would be useful here. For this pattern we didn't develop a particular role, but we reused the Container role (Figure 5), which takes care of children management in the Composite role

6.1.6 Chain of Responsibility

The purpose of the Chain of Responsibility pattern is to avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Each object is chained to another and the request is passed along the chain until an object handles it. The implementation of this pattern, in Java, often involves the use of a reference to the successor and the code to handle or pass the request. The specific code to each instance relates to how the request is handled and how each handler determines if it can or cannot handle the request. There are some implementations that require no request information to be passed, that is, the request method has no parameter. There are also implementations in which the request method returns a value. To accommodate these variations we developed a role for each.

In our roles the names of the methods are configured using the renaming feature. The types of the handlers, the request (in those roles that use it) and return type (where in use) are defined by generics. The roles also define the get and set methods for the successor, which are also configurable via renaming.

6.1.7 Testing role player independency

In order to asses if our roles are indeed 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 the corresponding DSM as an example of that work.

For the Observer role we developed a Flower subject that must notify its observers when it opens. For that we developed a FlowerObserver that has the flowerOpened(FlowerEvent e) method. As an observer we developed a Bee class that, when notified, will print a message that the bee is seeing an open flower. The Flower class plays the FlowerSubject role, which is the Subject role configured to this particular scenario. Figure 10 shows the code for both role and class. 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 a good programming practice to do so.

From that sample we obtained the DSM of Figure 11. 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. That would hold even if we didn't declare the FlowerSubject role and used the direct configuration as discussed before. 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

```
public role FlowerSubject {
  plays Subject<FlowerObserver,FlowerEvent>(
    Thing=FlowerObserver,
    Event=Open,  Event.notify=flowerOpened ) sbj;
}

public class Flower {
   plays FlowerSubject flwrSubject;
   private boolean opened = false;

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

Figure 10: The FlowerSubject role and the Flower class from our subject role sample.

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 a role version of the JHotDraw Framework[1].

6.1.8 Conclusion

From our study there are a few patterns that do not gain from the use of roles, namely: Builder, Façade, Iterator, Mediator, Memento, Strategy and Template Method. These roles are quite instance specific and the classes built for their implementation are dedicated and are not reusable outside that pattern. There are a few patterns that could benefit from the use of roles like a way to emulate multiple inheritance and to provide a default code implementation to some operations done in a class inheritance hierarchy. These are the Abstract Factory and specially Decorator. This is also valid for the State if states share some common code.

We also found some similar code between instances that we could not isolate and put into a generic 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. Arranging support for such variations would make the role heavily configurable and the configuration alone would be more complex than to write the code in the first place.

We developed roles for a total of 10 patterns out of 23, which is a good outcome, especially because every role has a high reusability factor. We believe that our Subject role, for example, shall be useful for a large number of Observer instances. There are also additional advantages in some roles, like a better modularity, in the Factory Method and Prototype. Other roles are limited in their actions, like Factory Method, which addresses a particular variation, but is highly reusable for that purpose.

[1]www.jhotdraw.org

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 11: Dependency Structure Matrix for the Observer role sample.

6.2 Roles in Frameworks

The next step is to analyze several frameworks in order to access whether our roles are really reusable and at the same time find more generic roles to complete our library. Even though we are in an initial stage of this study some of our roles have proved to be quite useful, namely Observer and State, which we already successfully used in the JHotDraw Framework.

7. RELATED WORK

There are various proposed extensions to the Java language, and others, to support roles but none has our renaming feature or aims at building a library of roles. Most are concerned with the dynamic nature of roles. Other approaches include support for relationships rather than roles.

A well-known approach is the role object pattern [5, 7]. 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. This approach, however, does not account for the static part of roles which is our main concern.

Object Teams [11] 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 [31] 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 [4] 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.

Traits [26] offer a way of composing software that are somewhat similar to Mixins [6]. A trait is the primitive unit of code reuse, like roles in our approach, 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. The flat property of classes in traits 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 we allow roles to have state and traits aren't allowed to have state. The trait's solution for the Container role would have to rely on the class to provide the storage for the elements. In traits we can provide aliases for methods, similar to our renaming mechanism, 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.

The remainder of this section is dedicated to approaches that deal directly with relationships rather than roles.

In [23] authors use Scala as a programming language to develop a library to describe roles and relationships. Traits are used to insert state and behavior in objects that participate in a relationship.

Neslson et al. [17] builds a library to support roles and relationships but using the Java language. They use tuples has the main construct. Relationships are implemented as a mutable set of tuples.

Pearce and Noble [21] use aspect technology to build a library of aspects that support relationships. The approach has the advantage to place the relationship code outside the participants and offers the possibility to dynamically add participants.

The C# language gets the NOIAI class library for representing relationships in [19]. Role access is similar to field access, much like our approach. But to a given object we access not the role it plays but the role of the other end of the association. To access the employer of the worker john we must use the john.employer syntax. In our approach it would be john.getEmployer( ) which we believe is more natural.

8. CONCLUSION

For roles to reach mainstream languages developers must have strong reasons to use them. We believe that a library of reusable roles is a good starting point in capturing the developers' attention. This is achievable only if roles can be manipulated like any other module in software. To reach this goal we presented a set of guidelines that role support must follow to allow the creation of such a library.

We presented a Java extension, JavaStage, which follows these guidelines and discussed how it could be used to develop a library or roles. Our work with the GoF Design patterns showed that JavaStage has a great potential for developing reusable roles and that it is possible to develop useful and fully reusable roles.

9. ACKNOWLEDGEMENTS

We would like to thank our shepherd for the useful comments and guidance. We also want to thank the reviewers in the writers' workshop for their valuable insights and suggestions.

10. REFERENCES

[1] A. Albano, G. Ghelli, and R. Orsini. Fibonacci: a programming language for object databases. *The VLDB Journal*, 4:403–444, July 1995.

[2] C. W. Bachman. The role data model approach to data structures. In *ICOD'80*, pages 1–18, 1980.

[3] C. W. Bachman and M. Daya. The role concept in data models. In *Proceedings of the third international conference on Very large data bases - Volume 3*, VLDB '1977, pages 464–476. VLDB Endowment, 1977.

[4] M. Baldoni, U. Studi, and T. Italy. Interaction between objects in powerjava. *Journal of Object Technology*, 6:7–12, 2007.

[5] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. Role object. *Pattern Language of Program Design 4*, pages 15–32, 2000.

[6] 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.

[7] M. Büchi and W. Weck. Generic wrappers. In *Proceedings of the 14th European Conference on Object-Oriented Programming*, ECOOP '00, pages 201–225, London, UK, 2000. Springer-Verlag.

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

[9] K. B. Graversen. *The nature of roles—A taxonomic analysis of roles as a language construct*. PhD thesis, IT University of Copenhagen, Denmark, 2006.

[10] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '02, pages 161–173, New York, NY, USA, 2002. ACM.

[11] S. Herrmann. *Programming with Roles in ObjectTeams/Java*. 2005.

[12] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[13] G. Kniesel. Objects don't migrate! – perspectives on objects with roles. Technical report IAI-TR-96-11, ISSN 0944-8535, CS Dept. III, University of Bonn, Germany, nov 1996.

[14] B. B. Kristensen. Object-oriented modeling with roles. In *Proceedings of the 2nd International Conference on Object-Oriented Information Systems*, pages 57–71. Springer-Verlag, 1995.

[15] B. B. Kristensen. Architectural abstractions and language mechanisms. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*, APSEC '96, pages 288–, Washington, DC, USA, 1996. IEEE Computer Society.

[16] B. B. Kristensen and K. Osterbye. Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2:143–160, December 1996.

[17] S. Nelson, J. Noble, and D. J. Pearce. *Implementing first-class relationships in java*. 2008.

[18] E. Odberg. Category classes: flexible classification and evolution in object-oriented databases. In *Proceedings of the 6th international conference on Advanced information systems engineering*, CAiSE '94, pages

406–420, Secaucus, NJ, USA, 1994. Springer-Verlag New York, Inc.

[19] K. Osterbye. Design of a class library for association relationships. In *Proceedings of the 2007 Symposium on Library-Centric Software Design*, LCSD '07, pages 67–75, New York, NY, USA, 2007. ACM.

[20] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15:1053–1058, 1972.

[21] D. J. Pearce and J. Noble. Relationship aspects. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 75–86, New York, NY, USA, 2006. ACM.

[22] B. Pernici. Objects with roles. In *Proceedings of the ACM SIGOIS and IEEE CS TC-OA conference on Office information systems*, COCS '90, pages 205–215, New York, NY, USA, 1990. ACM.

[23] M. Pradel. Explicit relations with roles - a library approach. Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL) at OOPSLA '08, 2008.

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

[25] D. Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich, Zürich, Switzerland, 2000.

[26] N. Schärli, S. Ducasse, O. Nierstrasz, and A. P. Black. Traits: Composable units of behaviour. *Lecture Notes in Computer Science*, 2743:248–274, 2003.

[27] P. Sousa, A. R. Silva, A. R. Silva, and J. A. Marques. Object identifiers and identity: a naming issue. In *In International Workshop on Object Orientation in*. IEEE Press, 1995.

[28] J. Sowa. *Conceptual Structures: Information Processing in Mind and Machine*. Addison-Wesley, 1984.

[29] F. Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data Knowl. Eng.*, 35:83–106, October 2000.

[30] F. Steimann. Role = interface: a merger of concepts. *Journal of ObjectOriented Programming*, 14(4):23–32, 2001.

[31] T. Tamai, N. Ubayashi, and R. Ichiyama. Software engineering for multi-agent systems v. chapter Objects as Actors Assuming Roles in the Environment, pages 185–203. Springer-Verlag, Berlin, Heidelberg, 2007.

[32] P. Tarr, H. Ossher, W. Harrison, S. M. Sutton, and Jr. N degrees of separation: Multi-dimensional separation of concerns. pages 107–119, 1999.

[33] E. Truyen. *Dynamic and context-sensitive composition in distributed systems*. PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, November 2004.