Refactoring to Static Roles

Fernando Barbosa
Escola Superior de Tecnologia
Instituto Politécnico de Castelo Branco
Castelo Branco, Portugal
fsergio@ipcb.pt

Ademar Aguiar
INESC TEC and Departamento Informática
Faculdade de Engenharia da Universidade do Porto
Porto, Portugal
ademar.aguiar@fe.up.pt

Abstract— **Roles can be used to overcome some composition limitations in Object Oriented Languages and contribute to a better code reuse, reducing code replication and improve code maintenance. Therefore, the refactoring of legacy code to roles is an important step in maintaining and evolving this code. In this paper, we present refactorings to convert a system to roles We also present some refactorings that enable roles to be even more reusable.**

Keywords- roles; refactoring; code reuse; code maintenance

I. INTRODUCTION

The "tyranny of the dominant decomposition" states that a single decomposition strategy cannot capture all possible views of a system [1], so there are always concerns that cannot be adequately decomposed and are scattered among the various modules. Several decomposition alternatives have been proposed: mixins [2], traits [3], features [4], aspects [5] and both dynamic [6][7] and static roles [8].

We use static roles as defined by Riehle in [6]. We do not use roles as dynamic entities that can be attached or detached from objects. There is much work on dynamic roles [7][9] [10] so this is mentioned to avoid confusion. Our static roles model concerns that are a subset of a class responsibility: those that are not the class main concept. Roles compose classes by adding their code to the class. The class interface can be seen as a whole or as a union of all the methods offered by the roles it plays. To program with roles we use JavaStage, an extension to Java. For more information on static roles and JavaStage we refer to [8].

Our experience with the use of static roles showed that they provide better decomposition when compared to class decomposition [8]. We would improve legacy systems if we make them use roles. The use of roles would provide a better way to reuse code, eliminate code replication, enhance the systems' modularization and easy maintenance.

Refactoring [11][12] is program transformation where the program maintains its behavior but is improved in non-functional qualities like readability, reuse or changeability. We can use refactorings as a way to transform a system without roles into a system with roles. There is not, however, a catalogue for role related refactorings. To fill this gap we present, in this paper, a collection of role related refactorings.

In these refactorings we use the JavaStage language [8] because it is backward compatible with Java and JVM compliant. Existing systems can be upgraded to roles in a transparent way to their users.

The refactorings were developed using our experience using roles to reduce code replication in several systems, including those referred in [8], and also when developing design patterns using roles [13]. The proposed refactorings may not be complete but they provide a starting point for a role refactoring catalogue. Our experience has been transforming existing systems into roles and not developing and maintaining systems with roles, so there may be some refactorings that only deal with roles yet to be discovered.

The rest of the paper is organized as follows: Section II shows a refactoring example. Section III presents the advantages of refactoring to roles. Section IV presents the proposed refactorings. Section V deals with related work and Section VI concludes the paper.

II. A FIRST EXAMPLE

Consider the example of Figure 1 which shows an excerpt of an Abstract Figure class that is a superclass for all the figures in a drawing application. The figure must warn the view whenever it is changed so the view can be updated. An Observer [14] is used for this purpose. We can argue that being a subject is not the class's main concern. From this we can say that the code from lines 7 to 20 should not be in the class. We can put that code into a FigureSubject role by using Extract Role. The outcome is shown in Figure 2.

A role may define methods and fields with access levels (lines 11-26). To play a role the class uses a plays directive and gives the role an identity (line 2). A class playing a role is called a player of the role. When a class plays a role all the non private methods of the role are added to the class.

Looking at the role we can see that to use it in other situations we could just use another observer type. Ignoring methods names, for now, we could apply the Replace Type with Generic refactoring and build the role in Figure 3.

We can observe that what prevents this role from being reusable for other instances of the observer pattern are the methods names. The methods that require the use of Make Method Name Configurable are the methods that add and remove observers, the fire methods and the update methods.

The JavaStage language allows the configuration of a method name. It can also require certain collaborators to have specific methods. These features are used in the Make Method Name Configurable refactoring.

Each configurable method name may have three parts: a configurable one and two fixed (optional). The configurable part is bounded by # as in fixed#config#fixed. Configuration is done by the class playing the role in the plays clause.

 ISBN: 978-1-61208-304-9

```
1 public class AbstractFigure implements Figure {
2   private Color color;
3   public void moveBy(int dx, int dy) {
4      fireFigureMoved( );  }
5   public void setColor( Color c ){
6      fireFigurePropertyChanged( );   }
7   private Vector<FigureObserver> observers =
8                  new Vector<FigureObserver>();
9   void addFigureObserver(FigureObserver o){
10     observers.add( o ); }
11  void removeFigureObserver(FigureObserver o){
12     observers.remove(o);}
13  protected void fireFigureMoved( ){
14     for( FigureObserver o : observers )
15        o.figureMoved( );
16  }
17  protected void fireFigurePropertyChanged( ){
18     for( FigureObserver o: observers )
19        o.figurePropertyChanged( );
20  }
21}
```

Figure 1 An excerpt of an AbstractFigure class doing work outside its main concern

```
1 public class AbstractFigure implements Figure {
2   plays FigureSubject figSubject;
3   private Color color;
4   public void moveBy(int dx, int dy) {
5      fireFigureMoved( );
6   }
7   public void setColor( Color c ){
8      fireFigurePropertyChanged( );
9   }
10}
11public role FigureSubject {
12  private Vector<FigureObserver> observers =
13                 new Vector<FigureObserver>();
14  void addFigureObserver(FigureObserver o){
15     observers.add( o ); }
16  void removeFigureObserver(FigureObserver o){
17     observers.remove(o); }
18  protected void fireFigureMoved( ){
19     for( FigureObserver o : observers )
20        o.figureMoved( );
21  }
22  protected void fireFigurePropertyChanged( ){
23     for( FigureObserver o : observers )
24        o.figurePropertyChanged( );
25  }
26}
```

Figure 2 The class from Figure 1 now refactored to roles.

```
1 role Subject<ObserverType> {
2   private Vector<ObserverType> observers =
3                  new Vector<ObserverType>();
4   void addObserver(ObserverType o){
5      observers.add( o ); }
6   void removeObserver(ObserverType o){
7      observers.remove(o); }
8   // ...
9 }
```

Figure 3 The role from Figure 2 using other types of observers

JavaStage has a multiple method version feature. It is possible to declare several versions of a method using multiple definitions of the configurable name. Methods with the same structure are defined once. Using these features we can develop the role and class that are depicted in Figure 4.

III. REASONS TO REFACTOR TO ROLES

In this section we present the advantages of refactoring a system to roles.

1) Refactor to Reuse Code. Delegation and inheritance may be used to reuse code. A class represents a concept others reuse by using instances of the class. If some classes have a common behavior we put that behavior in a class and make those classes inherit from it. However, with single inheritance, classes that are part of another hierarchy cannot reuse the common behavior. Multiple inheritance has many problems so many recent languages do not support it.

If we place the common behavior in a role we can reuse that role whenever we need, since they have not the multiple inheritance problems neither have single inheritance limitations. A class can play many roles and even play the same role more than once without duplicated field conflicts. The fact that roles are tailorable for a particular task, due to method renaming and type configuration allows a wider range of reuse not available with inheritance or delegation. The GenericSubject role shows how reusable a role can be.

2) Refactor to Remove Code Clones. Programmers sometimes reuse solutions by copying code and modifying it to fit a new purpose. This leads to code cloning as several fragments of a system will be identical or very similar. This can have immediate advantages like reduced development time, but in the long run a system with code clones is more difficult to maintain [15][16] and more error prone [16].

Code clones can be eliminated by better design [17] or refactoring [11][18][19]. Traditional refactoring used to deal with clones are**:** Extract Method, Pull Up Method, Extract Superclass, Extract Class and Form Template Method. We extend these refactorings by proposing to refactor to roles.

To eliminate duplicated code using roles we need to develop a role providing the replicated behavior. This way a

```
1 role Subject<ObserverType> {
2   requires ObserverType implements
              void  #Event.update#( );
3   Vector<ObserverType> observers =
              new Vector<ObserverType>();
4   public void add#Observer#(ObserverType o){
5      observers.add( o ); }
6   void remove#Observer#(ObserverType o) {
7      observers.remove( o ); }
8   protected void fire#Fire#( ){
9      for( FigureObserver o : observers )
10        o.#Fire.update#(  );
11  }
12}
13public class AbstractFigure implements Figure {
14  plays Subject<FigureObserver>
15  ( Fire=FigureMoved, Fire.update=figureMoved,
16    Fire = FigurePropertyChanged,
17    Fire.update = figurePropertyChanged
18    Observer = FigureObserver ) figSubject;
19  private Color color;
20  public void moveBy(int dx, int dy) {
21     fireFigureMoved( );
22  }
23  public void setColor( Color c ){
24     fireFigurePropertyChanged( );
25  }
26}
```

Figure 4 A subject role and an AbstractFigure class playing it.

 ISBN: 978-1-61208-304-9

class does not need to replicate the code, just play the role.

3) Refactor to Enhance Modularization. A single decomposition strategy cannot adequately capture all the system's details [1]. The result are crosscutting concerns, that appear when several modules deal with the same problem because one cannot find a single module responsible for it. This leads to replicated code as each class must implement the code on its own.

With roles however, we can place the crosscutting concern in a role. The concern is thus neatly modeled. Because there is a role-player interface they can be seen as independent modules. Roles are used to compose classes but they are also independent of the classes so we can argue that roles provide a better modularization.

4) Refactor to Ease Maintenance. If a module deals with a problem that is spread by several others then changes to the code will, probably, affect other modules. Independent development is compromised. Evolution and maintenance are a nightmare because changes to that code needs to be done in all modules. If a role is used to model that concern then all changes are made in the role alone.

IV. ROLE REFACTORINGS

This section presents the role refactorings we propose. We present in tables 1 and 2, for each refactoring, the name, a summary of the situation in which the refactoring is useful and a summary of the recommended actions.

We grouped the refactorings in two categories: refactorings to extract concerns into roles (shown in table 1) and refactorings to improve role reuse (shown in table 2). We recommend that the role extraction refactorings should be used first. After the role is in place it is easier to find how we can refactor it to make it more reusable. We can also detect that some roles are similar and refactoring them so they become identical and we can leave just one.

We will present, for each refactoring, a motivation and a discussion of the mechanics. Due to space constraints we cannot present the full details but will cover the main problems and variations. We do not state where to compile and test and rely that readers are aware that these steps are crucial in refactoring. Also due to space constraints we will not present step by step snippets of code or even code samples for each refactoring but will present examples that show how several refactorings are used.

A. *Refactorings to extract concerns to roles*

These refactorings are intended to extract concerns to roles so classes can deal with their main concern only. There are top level refactorings like Extract Role and low level ones as Move Method Between Class and Role.

1) Extract Role. We use this refactoring whenever we feel that a class is doing work that falls outside the class main concern. The motivation is thus the same as for the Extract Class from [11].

The mechanics are simple: Create a role with a name that indicates the concern it deals with; Move each field and method that are related to that concern to the role by using Move Field From Class to Role and Move Method From Class to Role; Make the class play the role.

a) Extract Role vs Extract Class. Extract Class can be replaced by Extract Roles. This way classes do not need to create delegation methods, just play the role. Which one to use depends on the code nature. If it is a standalone concept it should be put into a class, otherwise it should be put into a role. This follows the role definition that a role is an observable behavioral aspect of a class. In Figure 1 the code reflects only a partial behavior, an entity that maintains an observer list and informs them, so role use is better.

b) Extract Role vs Extract Superclass. This refactoring could be used instead of Extract Superclass. Again the decision is based on the concept the code represents. If it is better modeled by a class and inheritance is adequate then Extract Superclass should be used. If the concept is better modeled by a role then Extract Role should be used. Extract Superclass forces classes to be in an inheritance hierarchy. In contrast, Extract Role does not require player classes to be related. On the other hand, Extract Superclass can take advantage of polymorphic code and roles cannot.

2) Move Method from Class to Role. Moving a method to a role is different than moving a method to a class, so we included this refactoring. When a class plays a role it obtains the role methods, thus we do not need delegate methods. Figure 2 shows the outcome of this refactoring for the add, remove and fire methods.

The simplified mechanics are: Apply Move Method to the method always removing the delegate method; If the method makes references to the player object replace that object with the performer keyword; For each method that is called on the player place it in the requirements list.

3) Move Field Between Class and Role. As with moving methods, moving a field to a role is somewhat different from moving it between classes so we decided to include a new refactoring. The main difference is that a role cannot access the player fields nor the class can access role fields.

The simplified mechanics are: If the field is used by the class from which it is being moved then use Encapsulate Field; Use Move Field on the field and Move Method on the getters and setters. Figure 2 shows the outcome of this refactoring for the vector of observers.

4) Replace Superclass with Role. Inheritance is a good way to get a default implementation for a concern. But this cannot be used just for code reuse, the classes must have something in common than just code. The benefits of reusing implementations, however, are so great that inheritance is used just the same. Roles provide another way of reusing implementations and can be used in this situation. Figure 5 shows such an example.

For this refactoring the simplified mechanics involve: Use Make Class a Role on the superclass; Replace every extends for the superclass with a plays for the role.

5) Make Class a Role. When a class only provides behavior meant to be used by other classes then it is not a class but a role (see examples in Figure 6). We use Make Class a Role by: creating a new role; Copy the code of the class into the role; If the code has references to the client type then make them refer to the Performer type; If the code

Table 1. SUMMARY OF REFACTORINGS TO EXTRACT CONCERNS TO ROLES

Refactoring name	Situation summary	Typical Action Summary
Extract Role	You have a class doing work outside its main concern	Create a role and move the relevant fields and methods to the new role
Move Method from Class to Role	A method is used or using more features from a role than the class on which it is defined	Create a new method with a similar body in the role and remove it from the class
Move Field Between Class and Role	A field is used by a role or class more than it is used by the class or role on which it is defined	Create a new field in the role or class, encapsulate it and change the class or role to access the field trough methods
Replace Superclass with Role	A superclass is used by its subclasses for reuse purposes only	Create a new role with similar code of the class and make subclasses play the role instead of inheriting from the class
Make Class a Role	You have a class that represents only a partial behavior	Make the class a role
Replace Delegation with Role Playing	A class has a number of delegating methods to another class	Create a role with similar code of the delegated class. Make the delegating class play the role instead.
Inline role	You have a role that only one class plays	Move the role code into the class
Move Method from Role to Class	A method is used or using more features from a class than the role on which it is defined	Create a new method with a similar body in the class and remove it from the role
Replace Role Playing with Superclass	Classes that are related trough inheritance are using role playing instead	Create a class that plays the role and make subclasses inherit from the class.

Table 2. SUMMARY OF THE REFACTORINGS TO IMPROVE ROLE REUSE

Refactoring name	Situation summary	Typical Action Summary
Replace Type with Generic	A role is bound to a type but could be used with another type as well	Turn the type into a generic and instantiate the type when playing the role
Make Method Name Configurable	A method name is too general to be of use in several instances of a role	Use the renaming scheme to provide a configurable name and let players configure its name.
Rename Role Method	The name of a role method does not reveal its purpose	Change the name of the method
Name a Configurable Method	A method name is configurable when it should be fixed	Remove the configurable part of the name and give the method a suitable name
Replace Generic with Type	A generic type is used in the role but players always use the same concrete type	Replace the generic type with the concrete type

has references to the client object then substitute those references to performer. For every client method referred to in the role add it to the requirements list.

6) Replace Delegation with Role Playing. A class may be used by others just to provide an implementation for some features, where the client class just delegates the job. We can have the same effect by placing the implementation in a role and the client playing the role (see Figure 6). The mechanics for this are: If the class is used in this way by all clients then use Make Class a Role; If the delegated class is used in another way by other clients consider using Extract Role on the delegated class to extract its behavior into a role; Make the class play the new role; Remove all references to the class; Remove all delegate methods.

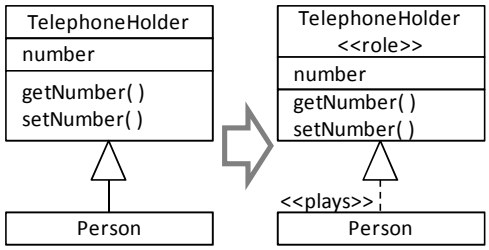

Figure 5 Replace Superclass with Role

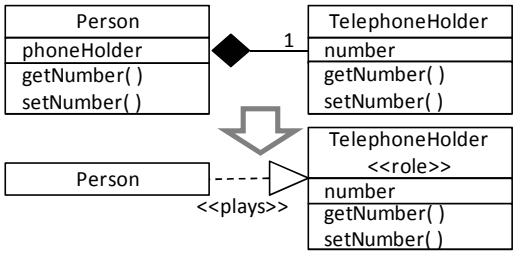

Figure 6 Replace Delegation with Role Playing

7) Inline role. A class that plays a role has become a more suitable implementation as its concern has evolved to include that of the role, or the role is played by just one class and has an insignificant amount of behavior.

We can Inline Role by: Copying every field and method from the role to the class; If a role field has the same name of a class field Rename one so that there is not a name clash; If a class method has the same signature of a role method then do not copy that method from the role, except if the class explicitly calls that method, in which case you must Rename the role method so there is not a name clash; Delete the plays clause; Delete the role.

8) Move Method from Role to Class. This is different from Move Method From Class to Role, because of the steps involved: If the method is configurable them use the refactoring Name a Configurable Method first; If the method uses generics in the role but not on the class apply Replace Generic with Type; Apply Move Method to the method; If the method makes references to role fields use accessor methods. If the method calls other role methods make the calls explicit by using the role identity.

9) Replace Role Playing with Superclass. Classes that play the same role may be related by inheritance instead. The mechanics are: Verify if classes that use the same role using the same configurations should be related by

inheritance; Create a new Class with a suitable name; Make the new class play the role using the same configurations as it subclasses; Make all classes extend the new class; Remove the plays in all subclasses.

B. Refactorings to Improve Role Reuse

We can make a role more reusable if we can expand its possible players, whether by making the types it uses more general or by making its methods configurable by the player.

1) Replace Type with Generic. Generics can be used as a place holder for the real type. The real type is defined when the code is actually being used. We suggest that if a type used by a role can be replaced by a generic it should.

Problems arise when we intend to call methods on those generic types. Java can bound a generic to certain types. For example, class Sample<T extends SuperType>, bounds T to be a subclass of SuperType. The problem when using roles is that these boundaries can be restricting. For example, in an Observer subject observers can be of any type and their interfaces are different. Roles have a requirements list that takes care of this problem.

We recommend the use of generic types instead of a concrete type. This is how to do it: Identify which types can be made generic so the role may be more reusable; Substitute each type by a generic; For each method that is called on the generic type place it in the requirements list; If the method name is not general use Rename Role Method or consider using Make Method Name Configurable.

An example is presented in Figure 3, where the FigureSubject role of Figure 2 has its FigureObserver type replaced by the generic ObserverType.

2) Make Method Name Configurable. Meaningful method names can be difficult to achieve. Role developers do not know the concrete context where the role will be used so use names that are generic. The player developer knows which names would fit the concrete use but cannot rename them because it could break other players.

To Make a Method Name Configurable: Identify which part of the method is more likely to change; Consider other methods that may have similar name parts so they can all be altered with this refactoring; Give a suitable configurable part for each method; If a method is used by the role then rename it in every place it is called and in the requirements list; The role name may also be Renamed to accommodate its wider use; Configure each player of the role so that they give the configurable methods the same names as before.

An example is presented in Figure 4. The FigureSubject role from Figure 2 is renamed to Subject and its methods are made configurable so players can choose a proper name for the add and remove methods and for each fire method.

3) Rename Role Method. Renaming a role method is trickier than renaming a class method, because the name may be configurable. If a name is not configurable then the mechanics of renaming the method is equal to Rename Method, with the difference that we must check every client of every player class. When the method is configurable the renaming is done thus: If the renaming affects only the configurable part then change it in all the role code that uses the method; Change the configurable part in all the plays clauses for that role. If the renaming affects the fixed part of the name replace it in every occurrence in the role. For each player class check if the renamed method is overridden and if it is decide if the class should not rename its own method; Change each client of each player to use the new name.

4) Name a Configurable Method. Configurable role methods allow the method name to be adequate in several situations, but sometimes we can make names configurable where a single name is suitable for every use. To Name a Configurable Method: Check if all players use the same name for the method or the name suits all players; Check if any player uses a multiple version of this method; In the role rename the method to the fixed name; In the role update all references to the method with the new name; In each player delete the configuration of the method from the plays clause if this was the only method to use that configurable part.

5) Replace Generic with Type. When developing generic roles we know we overdue the use of generics if all clients use the same concrete type. This refactoring makes the role simpler to use. The mechanics are: Replace all occurrences of the generic with the concrete type in the role; If the generic has entries in the requirement list delete them; In each player remove the instantiation of the concrete type.

V. RELATED WORK

There is much work related to Object Oriented refactorings [11][12] and we adapted some of those to roles, but to our knowledge there is no published work that concerns refactoring to roles. This includes the works of dynamic roles and not just static roles. Static roles have been used in the work of VanHilst and Notkin in [20] where they proposed to use roles in the C++ language. Dynamic role approaches as EpsilonJ [21] and PowerJava [10] have been around for a while but no refactorings to dynamic roles have been published. We believe that our adaptation of code smells to roles can also benefit these role related approaches.

Object Teams in its project home page [22] mentions the adaptation of Extract Method, Move Method, Pull Up, Pull Down and Rename to the objects teams specific relationships (implicit role inheritance, team nesting, role-base bindings and method bindings). They also support new role related refactorings like Extract Callin and Inline Callin. But, there is not a presentation or mechanics of these refactorings.

The role object pattern [23] is used for representing objects that expose different properties in different contexts. Steimann and Stolz [24] describe a way to refactor code to this pattern that provides lightweight role objects with a leaner code than the previous approaches. They also softened the preconditions on when to apply the refactoring.

There are other approaches to class compositions, like Traits [3], Multi-dimensional separation of concerns [1]. Package Templates (PT) [25], Caesar and its Virtual classes [26], Jiazzi and its Units [27]. To our knowledge none of these approaches tackled the problem of refactoring legacy code. We consider Traits to be the most related approach to static roles, as we can see a trait as a role without state. We believe, therefore, that some role refactorings can be used in Traits, namely Extract Role could be used as an Extract Trait

 ISBN: 978-1-61208-304-9

as long as we removed the part related to moving fields and replaced it with Encapsulate Field.

Feature Oriented Programming (FOP) decomposes the system into features [3]. Features reflect user requirements and incrementally refine each other. In [28], Liu et al propose a theory of Feature Oriented Refactoring (FOR), which is the process of decomposing a program into features, thus recovering a feature based design and giving it an important form of extensibility. Since a feature's implementation can vary between systems, the authors developed an algebraic theory of FOR that exposes the highly regular structure that features impose on programs. They also supply a methodology and a tool based on the theory. This work, however, can be applied only to FOP.

Aspect-Oriented Programming as used in AspectJ [5] is an approach that tries to modularize crosscutting concerns. There is work on refactorings systems to aspects [29]. Due to the renaming capability of JavaStage we can include some refactorings related to method names, while in AOP we cannot.

VI. CONCLUSIONS AND FUTURE WORK

We showed that refactoring a system to roles brings benefits to the system like a higher reusability, better modularization, among others.

We proposed a series of refactorings based on our studies with converting OO systems to roles and design pattern implementation using roles. These refactorings provide a way to convert legacy code to role code. Some refactorings deal with the problem of making the role more general purpose thus enhancing code reuse.

For future work we intend to develop a tool to give these refactorings some automatic support. We also intend to carry on our studies concerning role development so we can discover new refactorings that involve role development and use, not just upgrading roles and refactoring to roles. This will contribute to a more complete role refactoring catalogue.

REFERENCES

[1] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton Jr., "N degrees of separation: multi-dimensional separation of concerns", Proc. International Conference on Software Engineering, 1999, pp. 107-119.

[2] G. Bracha and W. Cook, "Mixin-based inheritance". Proc. OOPSLA/ Proc. ECOOP, 1990, pp 303-311.

[3] S. Ducasse, N. Schaerli, O. Nierstrasz, R. Wuyts and A. Black, "Traits: a mechanism for fine-grained reuse". Transactions on Programming Languages and Systems. vol. 28, no. 2, March 2006, pp. 331-388.

[4] S. Apel and C. Kästner, "An overview of Feature-Oriented Software Development", in Journal of Object Technology, vol. 8, no. 5, July–August 2009, pp 49–84.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold, "An overview of AspectJ". Proc. ECOOP 2001, pp 327-353.

[6] D. Riehle and T. Gross, "Role model based framework design and integration", Proc. OOPSLA '98. 1998, pp. 117-133.

[7] F. Steimann, "On the representation of roles in object-oriented and conceptual modeling", Data & Knowledge Engineering, vol. 35, no. 1, 2000, pp 83–106.

[8] F. Barbosa and A. Aguiar, (2013), "Using roles to model crosscutting concerns", Proc. Aspect Oriented Software Development (AOSD13), March 2013, pp 97-108.

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

[10] M. Baldoni, G. Boella and L. van der Torre, "Interaction between objects in power-Java", Journal of Object Technologies, vol 6, 2007, pp 7 – 12.

[11] M. Fowler. "Refactoring – Improving the Design of Existing Code", Addison Wesley, 2000.

[12] W. Opdyke, "Refactoring Object-Oriented frameworks", Ph.D. thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[13] F. Barbosa and A. Aguiar, "Generic roles, a test with patterns" Proc. Pattern Languages of Programs, 2011.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, "Design Patterns: elements of reusable Object-Oriented software", Addison-Wesley, 1995.

[15] I. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees", Proc. of the Int. Conf. on Software Maintenance, Nov. 1998, pp 368-377

[16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?", Proc. Int. Conf. on Software Engineering, IEEE Computer Society, 2009, pp 485-495

[17] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis, "Measuring clone based reengineering opportunities". Proc. International Software Metrics Symposium, Nov. 1999, pp 292-303

[18] R. Fanta and V. Rajlich, "Removing clones from the code". Journal of Software Maintenance: Research and Practice, vol. 11, no 44, August 1999, pp 223-243.

[19] Y. Higo, T. Kamiya, S. Kusumoto, and K. Inoue. "Refactoring support based on code clone analysis". Proc. International Conference on Product Focused Software Process Improvement, 2004, pp 220-233.

[20] M. VanHilst and D. Notkin, (1996) "Using role components to implement collaboration-based designs". Proc. OOPSLA '96, 1996, pp 359-369.

[21] T. Tamai, N. Ubayashi, and R. Ichiyama, "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, 2007, pp 185-203

[22] http://www.eclipse.org/objectteams/features.php, last access in Jan. 2013.

[23] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf, "The role object pattern", Proc. PLoP1997, 1997. pp 15-31

[24] F. Steimann and F. U. Stolz,. "Refactoring to role objects", Proc. International Conference on Software Engineering, 2011, pp 441-450.

[25] S. Krogdahl, B. Møller-Pedersen, and F. Sørensen, "Exploring the use of Package Templates for flexible re-use of Collections of related Classes", Journal of Object Technology, vol. 8, no. 7, Nov. – Dec. 2005, pp 59-85.

[26] E. Ernst, K. Ostermann, and W. R. Cook. "A virtual class calculus", Conference record of the 33rd Symposium on Principles of Programming Languages. 2006, pp 309-330.

[27] S. McDirmid, M. Flatt, and W.C. Hsieh, "Jiazzi: new-age components for old-fashioned Java", Proc. OOPSLA 2001, pp 211-222.

[28] J. Liu, D. Batory, and C. Lengauer, "Feature oriented refactoring of legacy applications". Proc. Inter. Conference on Software Engineering (ICSE '06), 2006, pp 872-881.

[29] M. Monteiro and J. Fernandes. "Towards a catalog of aspect-oriented refactorings". Proc. Int. Conf. on Aspect-Oriented Software Development, 2005, pp 111–122.

 ISBN: 978-1-61208-304-9