Modeling Crosscutting Concerns with Roles

Fernando Sérgio Barbosa
Higher School of Technology
Polytechnic Institute of Castelo Branco
Castelo Branco, Portugal
fsergio@ipcb.pt

Ademar Aguiar
Department of Informatics Engineering
Faculty of Engineering of University of Porto
Porto, Portugal
ademar.aguiar@fe.up.pt

***Abstract*—Modularization allows the development of independent modules and their reuse. However a single decomposition strategy cannot neatly capture all the systems concerns. Thus some concerns are spread over several modules – the crosscutting concerns. To cope with this we need to have other class composition techniques than those available in traditional Object Oriented programming languages. One of such compositions is roles. If roles are used to compose classes and if a role models a crosscutting concern, then the concern is limited to the role and not spread over several classes. To validate this approach we conducted a case study. In the case study crosscutting concerns were identified in a system using a clone detection tool and roles were developed to model those crosscutting concerns. Results show that this approach reduces significantly the spreading of crosscutting concerns code.**

Roles, Crosscuting concerns, Code clones

I. INTRODUCTION

Modularization [1] is one of the most important concepts in software development. Decomposing a system into modules allows the independent development of each module. This shortens development time and allows the modification of a module without changing other modules.

But a single decomposition strategy cannot capture all possible views of a module.[2]. We could use multiple inheritance, but it has so many practical problems that it has been left out of recent programming languages. Even if we could use multiple inheritance, there are always concerns that cannot be adequately decomposed using a single decomposition strategy [3], and end up scattered among the various modules. These are called the crosscutting concerns.

A consequence of crosscutting concerns is replicated code. When classes must implement a crosscutting concern developers tend to copy-paste the code that deals with it [4]. Thus the presence of code clones in a system is an indicator that there are crosscutting concerns in that system [5].

An obvious problem of code clones is the increased system size. But code clone also impairs system's maintenance and evolution [6]. A particular problem is the inconsistence in updating, where a bug in a code block is propagated to all its clones, and is fixed in most but not all occurrences. Code clones also have negative effects in program comprehensibility, evolution, cost and may be an indicator of design flaws [7].

To prevent such consequences we need to use other decomposition techniques. Several proposals are available, like inheritance, mixins [8], traits [9], features [10], aspects [11] and roles [2][12]. We believe that if we explore the way roles can be used to compose classes we will find that roles are capable of modeling crosscutting concerns.

There are many definitions of the role concept in the literature [2][12] but we are interested in using roles as components of classes. For that purpose we use the role definition used by Riehle [12], where roles are an observable behavioral aspect of an object. We can use roles to compose classes, meaning that an object's behavior is defined by the composition of all roles it plays.

For modeling crosscutting concerns with roles we place that crosscutting concern in a role and all classes that deal with that concern just have to play the role. This way the role encapsulates the concern code and prevents code clones.

To validate these ideas we conducted a case study with the JHotDraw framework. In this case study we used a clone detection tool to identify code clones in the framework. We identified crosscutting concerns by aggregating clones that deal with the same concern. Each crosscutting concern was analyzed and, whenever possible, a role that deals with that concern was developed using JavaStage [13], an extension to the Java language that supports roles. The results of the case study indicate that roles can in fact be used to model crosscutting concerns and reduce code clones from a system.

This paper is organized as follows. The next Section presents role modeling and how it can address crosscutting concerns. Section III presents the JHotDraw case study and its results. Related work is presented in Section IV, and Section V concludes the paper.

II. MODELING WITH ROLES

Role modeling using static roles was used as an integral part of the OORam method [14] and by Riehle in [12]. We took these modeling approaches into the programming level using roles as blocks for composing classes. To support roles we developed the JavaStage language. We will not discuss JavaStage but refer the reader to [13]. JavaStage extends java but our approach may apply to other single-inheritance languages and even in multiple inheritance languages.

In JavaStage a role is a first class entity, so it can be described using an appropriate type specification. A class that plays a role type acts according to the role type specification. Classes may act according to several different role types. Thus, different clients may have different views on a class instance.

A class represents a domain abstraction, its properties and behavior. But, in JavaStage, a class also defines which roles it plays and how they are composed. The union of the operations defined in the class and the operations defined in the roles constitutes the class interface, and the composition of all role types constitutes the type of the class. This is to say that the class interface is the union of the role interfaces [15]. Because a class may be viewed as a class that plays only one role then this model is a canonical extension of the object model [16]. It means that existing software can be integrated into the role model without changes.

We could achieve the same effect by using multiple inheritance, defining each role in a separate class. The composing class would inherit from all classes. The use of multiple inheritance, however, has many problems. These come mostly by name collisions when a class inherits from two or more superclasses that have equally named methods or fields and duplicated code when a class inherits twice from the same superclass – the classic diamond problem.

In JavaStage, roles have features like a powerful renaming mechanism that allows classes to tailor methods' names for their specific situation; the possibility to play the same role more than once and the possibility to define multiple versions of a method [13]. Roles can inherit from roles and can play other roles thus giving developers a big range of modeling options.

To exemplify role modeling we present in Figure 1 the class diagram of a simplified graphical user interface (GUI) framework based on Java AWT/Swing frameworks. Framework classes represent the widgets (or components) that usually appear in a GUI, like windows, buttons, menus, toolbars, etc. Some components may own other components: a window may own several toolbars, and a toolbar may own several buttons. Some clients may be interested on knowing when the mouse is hovering a component so the Observer pattern is used. Other instances of this pattern are used as clients may be interested in other user's actions or if a component has lost focus, etc. A component has a collection of properties that specifies the way it should de drawn. Properties are represented by name-object pairs, where name is the property and object represents the property's value.

Clients that are interested in knowing if a component has lost focus are not interested in drawing a component or if the user clicked it with the mouse. For those clients components assume the role of FocusSubject and only those operations related to that role are of interest. For those clients that want to know about the actions the user performs with the mouse it plays the role of MouseSubject. Clients may set or read properties of the component so, for these, the component plays the role of PropertyProvider. The CompositeParent role is responsible for managing a collection of children.

The mentioned roles are depicted on the upper right side of Figure 1, where we also show the role associations and the revised class diagram, now using the roles.

A. *Role modeling advantages*

Role modeling comes with several advantages in terms of reuse, comprehension, development and documentation [12]. When a class is described as a set of roles it helps separating the various ways in which a class is used. This means that

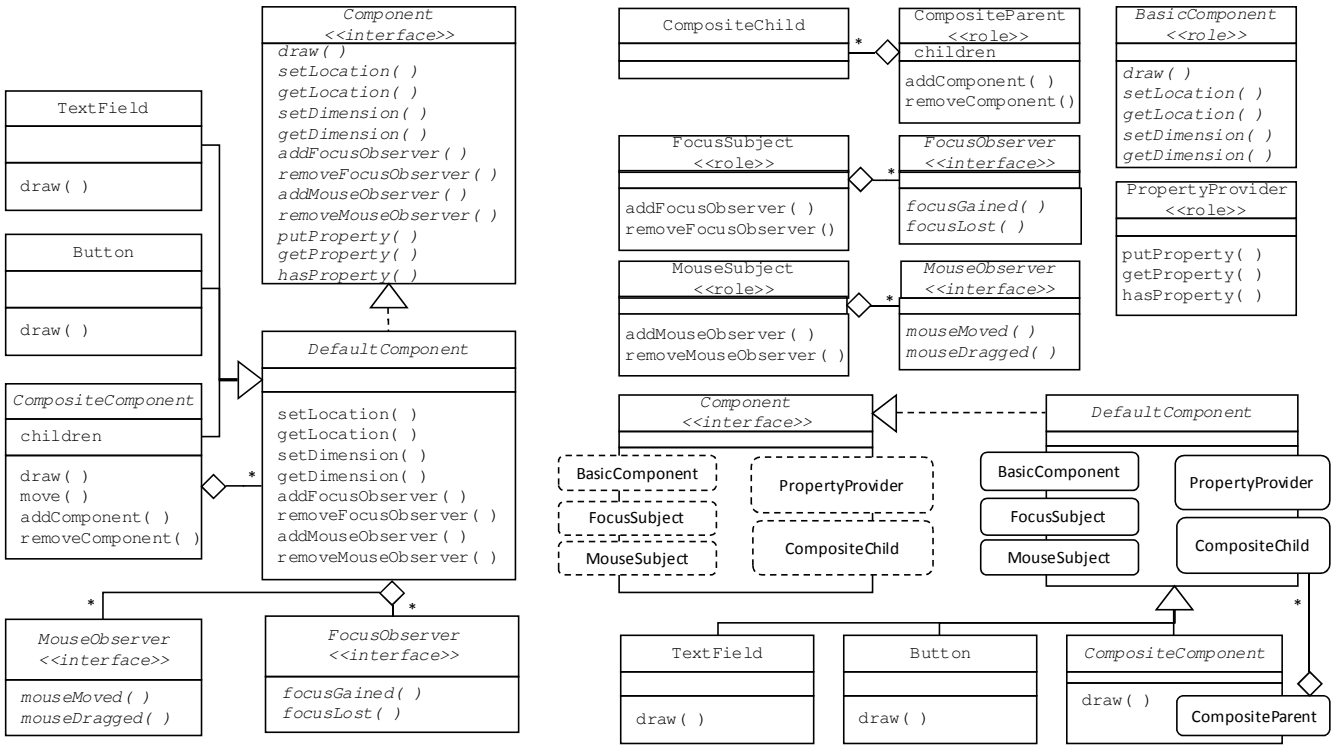

Figure 1. Example of modelling with roles. On the left: class diagram of the Component Framework. On the right: Roles and their relations in the Component Framework, and the revised class diagram of the Component Framework, now with roles. Rounded rectangles identify roles played by the class. Dashed round rectangles represent the interface provided by the role

the documentation can be done in these terms. That helps clients to better understand and use the class and focus on whichever aspect they are interested in. Designing the class can also be done in role terms, thus developers are able to focus only in one aspect of the class. This enables independent development of a class with all its benefits in terms of reduced development time and complexity.

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

Role modeling also allows for better understanding using previous experiences. When a developer knows how to use roles that have a relationship in a system, then when he encounters different roles with similar relationships the past experience will allow a better understanding. One such example is the use of the Observer pattern. When experienced with a FocusSubject and how it works with a FocusObserver to use the MouseSubject and a MouseObserver is much simpler and straightforward.

B. Modeling Crosscutting Concerns

Crosscutting concerns are those concerns that appear when several modules must deal with the same problem because one cannot find a single module responsible for it in the light of a decomposition strategy. This leads to scattered, replicated code. Its consequences are the opposite of the benefits of modularizations. Since a module deals with a part of a problem that is spread over other modules, changes to that code may affect those modules. This affects independent development. Maintenance is impaired too because changes in the code needs to be done in all modules transversely.

Because a role is smaller composition unit than a class we can put the crosscutting concern in a role, or a set of roles, and the classes that have the crosscutting concern play those roles. Any changes to the crosscutting are limited to the roles thus greatly improving maintenance and reducing change propagation, or in other words, the crosscutting concerns become more modular.

Even the simplified GUI framework shows several examples. The component's main concept is not to act like a Focus Subject or a Mouse Subject but it has those roles superimposed on it. With roles we were able to extract those concerns from the class, thus reducing the scattering of code. Furthermore those roles are reusable whenever we need a class to address any of those concerns, even if it is not part of the Component hierarchy. We can also argue that being a PropertyProvider is not the component's main concern. It assumes that a property is identified by a name and that name is a String. It would be more reusable if it used generics for the property type. We can also use generics to specify the value type instead of type Object. After a closer look, the property provider is in fact a map that maps keys to values. We could reuse a map implementation if we inherited from a Map class, but that would be conceptually wrong. Our class is not a map, it plays the role of a property map.

Figure 2 shows the code for that mapper role and the code for a Component class playing the PropertyProvider role and also of a Figure class that also plays the same role, but for figure properties like line color, line width, etc. In both cases the map uses string as keys and objects as values and in both cases the methods are getProperty, putProperty and hasProperty, as defined by the configuration, but they could use different key/values types and methods' names.

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

  public ValueType get#Thing#( KeyType name ){
    return map.get( name );
  }
  void put#Thing#( KeyType name, ValueType value){
    map.put( name, value );
  }
  public boolean has#Thing#( KeyType name ){
    return map.containsKey( name );
  }
}

class DefaultComponent implements Component {
  plays Mapper<String,Object>(
            Thing = Property ) mapper;
}
class DefaultFigure implements Figure {
  plays Mapper<String, Object>(
            Thing = Property ) mapper;
}
```

Figure 2. Definition of the Mapper role with configurable methods and two classes playing the role

III. CASE STUDY

A. Case Study Subject

To asses how roles are capable of modeling crosscutting concerns we applied them to the JHotDraw framework. JHotDraw is a Java GUI framework for technical and structured Graphics. JHotDraw is structured around four main inheritance hierarchies. These hierarchies reflect the main classes used in the framework. These are the Figures, Views, Tools and Handles.

JHotDraw has been used in works for the detection of crosscutting concerns for aspect mining [17] so it is a suitable candidate for this study, where we want to assess how roles handle those crosscutting concerns.

B. Case Study Setup

We searched for replicated code using CCFinderX [18] an established clone detection tool used in the aspect mining works [17]. We used the standard options of CCFinder.

We are interested in crosscutting concerns, so we are interested only in clones that are not solvable with traditional refactorings [19]. One of such refactorings is the Extract Method that usually deals with code inside a unique class. So to filter out such clones we only considered clones that appeared in, at least, two files. This also filter clones that do not deal with crosscutting concerns as a concern must be present in at least two classes to be considered a crosscutting

concern. For simplicity and space reasons we will refer crosscutting concerns simply as concerns.

The first output included 271 clone sets. After filtering we ended up with 146 clones. After a manual inspection 41 false clones were removed leaving a final 105 sets. Some clones are not really identical, but as they focused on the same concern so we did not remove them. This will account for some of the unresolved concerns.

We grouped clones according to the concern they dealt with. We identified a total of 43 concerns. From those 43 concerns we removed 5 because 2 could be resolved by refactoring alone, 1 was deprecated code and 2 were classes pending substitution.

After this selection we again analyzed the clones and, if possible, we've built a role encapsulating the concern. We've decided not to change any class interface so the overall framework is unchanged. We've also set a rule not to change the concern implementation to retain the author's intent. Only minor changes were allowed, as they wouldn't compromise it. We only developed roles that respect the role concept. We detected some clones that could be removed using a different inheritance hierarchy. We did not use roles to reduce that replicated code, because changing the inheritance hierarchy was a better solution

Roles were developed with JavaStage. The JavaStage compiler and the developed JHotDraw framework, can be found at http://www.est.ipcb.pt/pessoais/fsergio/javastage.

C. *Case Study Results*

Results are shown on Table 1. For each concern it shows how many clones were associated and how many classes were affected. It also shows the number of lines of code (LOC) that the clone had, the lines of code that were used by Roles and the ratio between them. For the concerns where roles failed it states the reason why they failed.

We can see that from the 38 concerns only 8 (21%) were not resolved with roles. This seems to indicate that roles are suited to model crosscutting concerns. The final outcome is better than these numbers indicate as we will discuss.

LOC are a good measure on the effort that each approach requires but it is not a good measure on how the modularity issues are handled. One can write more lines of code but if the resulting system is more modular it is a better system.

We counted as LOC the requirements statements that roles must declare. We also counted as LOC the roles' plays directive. Assume one concern that presents 8 lines of replicated code in each class which could be resolved with a simple role. We would expect this role to have the same 8 LOC. That is not so because we do not count the class declaration as a clone LOC but count the role declaration as a solution LOC. Roles may also require methods, so these requirements are counted as LOC. Thus for the 8 LOC clone the role would have 1 more fixed, 1 more for each player and 1 more for each requirement. If the role requires 3 methods and the clone appears in two classes then the clone has 16 LOC and the role solution would count 14 LOC. That may not seem a great improvement but LOC do not account for the modularity and maintenance issues. Removing the clone gives the system a great advantage in modularity terms.

TABLE I. IDENTIFIED CONCERNS WITH THE NUMBER OF ASSOCIATED CLONES AND AFFECTED CLASSES. IT ALSO SHOWS THE LOC FOR EACH APPROACH AND RESPECTIVE RATIOS.

Concern	clone #	class #	Original LOC	Roles LOC	Roles / Original
Drawing Handles	8	15	64	40	63%
Setting up the undo activity before executing a Command	2	8	56	44	79%
BringToFront/SendToBack Commands	1	2	20	12	60%
Handle creation	11	20	70	87	124%
Drawing polygons	1	2	12	11	92%
Palette Listener	1	2	20	17	85%
DisplayBox persistence	2	5	35	12	34%
DisplayBox handling	6	8	58	29	50%
DesktopListener Subject	2	3	63	45	71%
Changing connections	3	3	98	53	54%
Finding connectable figure	1	3	98	53	54%
Testing command executability	5	7	14	14	100%
Floating text holder	2	2	47	36	77%
DrawingViewListener Subject	2	4	63	26*	41%
Setting text in a text Figure	2	2	36	22	61%
Enumerator	1	3	33	11*	33%
Figure Listener that resends notifications	2	3	35	23*	66%
Menu enabling	1	2	20	14	70%
Version control	1	2	12	9	75%
Selected button manager	1	2	18	12	67%
Text attributes management	2	2	206	120	58%
Updating DrawingView Strategy	1	2	29	26	90%
Connection insets computing	1	3	10	7	70%
Undo/Redo Commands	1	2	32	31	97%
Changing connection handles	1	2	20	19	95%
Polygon and PolyLine Handles	3	2	32	28	88%
Tools and Commands Dispatchers	6	4	89	32*	36%
Figure/Handle and Enumerator	1	2	33	2*	6%
Polygon locator	1	2	13	20	154%
Drawing editor	1	3	54	28*	52%
			Reason		
Desktop initial configurations	1	2	Too much configuration		
Persistence (read/write)	3	6	Similar but not identical code		
UndoActivity	13	24	UndoActivity inner classes declaration and constructor		
Creating UndoActivity	14	18	After other roles was just a line of code		
Handle manipulation starting action	3	5	Too much configuration		
Point is inside Figure	3	6	code too small		
DrawingView Listener	1	2	perfomance issues		
Mouse motion handling	1	2	code too small		

* = reused from library

1) *Modelled Concerns*

Roles succeeded in 30 (79%) of the 38 concerns. This indicates that roles are capable of reducing replicated code and modeling crosscutting concerns. Comparing the LOC ratio, one finds that, in average, roles only have 68% of the original code, so the effort of developing the role system is smaller. Taking the absolute LOC value, the original system has 1390 LOC and roles have only 883 LOC. This means that roles reduced the replicated code in 36,5%.

In 6 concerns we were able to reuse/place roles from a role library [20] we developed to capture the basic behavior of the Gang of Four design patterns [21]. This explains the great difference in LOC in these concerns. From all the concerns roles resolved, two exhibit a higher number of LOC than the original implementation.

The "Handle creation" concern deals with the creation of handles for each figure. We moved the handle creation to a handle creator class and the role class methods on that class. Since some clones only have similar code we had to reproduce every method in this creator class. The class code, plus the code original classes use to play the role and the definition of the role leads to more lines of code than the original implementation. But the role has one advantage: it can dynamically change the handle creator.

The "Polygon Locator" is responsible for returning a point inside a polygon given a point index. It is used in two classes but one of them uses an anonymous class. Currently JavaStage's roles cannot be applied to anonymous classes so we had to develop an inner class. This code and the role configuration lead to a higher LOC, because the original code size was not enough to compensate for this overhead.

2) Unresolved concerns

A surprising result is that the 2 concerns with the most clone sets and class involved are unresolved with roles. This is due to the nature of the clones. They are clones only in the structure and not on the code itself. The "Creating undo activity" concern creates an undo activity object for each of the various tools and commands supported by the framework. Each tool class has an UndoActivity inner class hence the undo activity creation is just a line of code instantiating an object of the respective inner class. Because each inner class constructor has different parameters in number and types, roles could not resolve this concern. UndoActivity concern clones are due to the inner classes, because they all have the same name and constructors with the same structure, even if not equal. Another example of such a concern is the "Handle manipulation starting action": code is similar but not quite identical and most code would disappear with refactoring.

Another example is "Persistence": because figures must be streamed they have a write and read method with similar structure, but not quite identical code. We reduced this duplicated code with our DisplayBoxed role, though.

Another unresolved concern is the "DrawingView listener". An overriding method is redefining the original, allegedly for performance issues we failed to understand.

One unresolved clone, "Desktop Initial configuration", dealt with a Desktop's panel initialization, which initializes panel titles and adjusts a scrollPane. Each possible initialization is similar so we could configure a role for every way a scroll pane is configured and then reuse them. But knowing each possible role would require more effort than to know how to configure the scroll pane.

The other unresolved concerns were a single line in the form of `return getSomeObject().doSomething()`. Since the first method returns different objects that in turn call different methods, role configuration would be harder than writing the code itself.

Had we not considered some of these concerns as crosscutting concerns, we would count only 4 as unresolved.

3) Threats to Validity

One threat to this study results is that we only considered a single system. For results to be more decisive we might need to do the same test with more systems. Nevertheless the nature of roles allows us to say, with some confidence, that results for other systems would not be that different.

The clone detecting settings can also affect the detected clones that would lead to different concerns. That and the removal of clones from the same file could have removed important clones. However, we would need to reduce the amount of clone sets to a manageable number. We even go under the limit of the minimum 30 tokens recommended in [18] for limiting false clones. So while different settings would result in some different clones we believe that our settings provided a good result in detecting meaningful concerns.

IV. Related Work

Feature Oriented Programming (FOP) decomposes the system into features [10], which are the main abstractions in FOP during design and implementation. Features reflect user requirements and incrementally refine each other. FOP relies on a step-wise refinement of applications by adding new features or refining existing ones. FOP is mainly used in Software Product Lines and program generators. In FOP, Mixins are used to implement features [8]. Each mixin layer contains the code each class needs for a given feature and are composed into a static component. Roles can be used instead of mixins, as they offer more ways of configurations and don't have mixins limitations like a linear composition order.

Aspect-Oriented Programming is another approach that tries to modularize crosscutting concerns [11]. But AOP is not close to OO and requires learning many new concepts. And while the modularization of crosscutting concerns is the flagship of AOP several authors disagree [22][23]. Concepts like pointcuts and advices are not easy to understand. The effects of these constructs are also more unpredictable than any OO concept. A particular one is the fragile pointcut. This problem arises when simple changes made to a method code make a pointcut either miss or incorrectly capture a joint point thus incorrectly introducing or failing to introduce the necessary advice. Thus simple changes in the class code can have unsought effects [24].

The obliviousness feature of AOP means that a class is aspect unaware so aspects can be plugged or unplugged as needed. But it also introduces problems in comprehensibility [25]. To fully understand the system we must not only know the classes but also have to know the aspects that affect each class. This is a major drawback when maintaining a system, since the dependencies aren't explicit and there isn't an explicit interface between both parts. With our approach all dependencies are explicit and the system comprehensibility is increased when compared to the OO version [26]. We do not have the obliviousness of AOP as the class knows and is aware of the roles it plays. But any changes to the class code are innocuous to the role, as long as their contract is fixed.

We do not believe our approach can replace AOP. They are different and approach different problems. We believe that for static concerns our approach is more suitable while AOP is better suited for (un)pluggable concerns.

Traits [6] offer a way of composing software that is somewhat similar to Mixins [6]. A trait is the primitive unit of code reuse, like our roles, which means that only traits can

be used to compose classes. Traits can also be used to compose other traits. But traits only provide methods and not state and access levels. A class composed with traits can be seen either as a flat collection of methods or as being composed by traits. This flat property means that the code inside the trait can be seen as the code inside the class, for example, a super reference inside the trait code refers to the superclass of the class that uses the trait. In our approach we can also see a class as simply a set of methods, forgetting that it plays a role, but we have not this flat property, as a super reference in a role refers to the superrole.

V. CONCLUSION AND FUTURE WORK

We have presented a new way of modeling crosscutting concerns. Using roles we have a finer grain composition technique that allows the crosscutting concerns to be composed into the classes without its code being placed in the class itself.

We modeled crosscutting concerns by developing a role that addressed it. The crosscutting concern's code is therefore limited to the role. To better model those concepts roles support state and visibility control. Classes play the role and acquire the role behavior. Changes to the concern implementation are limited to the role.

We validated our approach developing roles for the JHotDraw framework and eliminated nearly all of the existing crosscutting concerns that exhibited duplicated code. We even reused some roles from our role library showing that they are really reusable.

For future work we are developing a role version of the Sun's java compiler and the Spring framework, using JavaStage. Results so far are promising as we already reused some of our library roles, like an Observer and Visitor. The use of these roles in those case studies can eliminate a great amount of duplicated code.

REFERENCES

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

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

[3] Tarr, P., Ossher, H., Harrison, W. and Sutton Jr., S. M. (1999), N degrees of separation: multi-dimensional separation of concerns, Proceedings of the 21st international conference on Software engineering, New York, NY, USA

[4] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin, An ethnographic study of copy and paste programming practices in oopl, Proceedings of the 2004 International Symposium on Empirical Software Engineering (Washington, DC, USA), ISESE '04, 2004, pp. 83–92.

[5] Bruntink, M. van Deursen, A. van Engelen, R. Tourwé, T., On the use of clone detection for identifying crosscutting concern code, IEEE Transactions on Software Engineering, Vol. 31, No. 10, (2005)

[6] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do Code Clones Matter? In Proc. Int. Conf. on Software Engineering, pages 485–495. IEEE Computer Society, 2009.

[7] C. Roy and J. Cordy. A Survey on Software Clone Detection Research. Technical Report 2007-451, School of Computing, Queen's University at Kingston, 2007.

[8] G. Bracha, and W. Cook. Mixin-Based Inheritance. In Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications / Proceedings of the European Conference on Object-Oriented Program-ming, pages 303–311, 1990. Ottawa, Canada.

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

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

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold. An overview of AspectJ. In proceedings of ECOOP 2001, Budapest, Hungary, (LNCS, vol. 2072), Springer; 327–335, 2001

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

[13] Barbosa, F. and Aguiar, A. (2012). Modeling and Programming with Roles: Introducing JavaStage, In the 11th International Conference on Intelligent Software Methodologies, Tools and Techniques (SoMeT_12), Genoa, Italy, to appear.

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

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

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

[17] Ceccato, M., Marin, M., Mens, K., Moonen, L, Tonella, P. and Tourwe, T. A qualitative comparison of three aspect mining techniques, Proceedings of the 13th InternationalWorkshop on Program Comprehension (Washington, DC, USA), IWPC '05, 2005, pp. 13–22

[18] Kamiya, T., Kusumoto, S. and Inoue, K. (2002), Ccfinder: a multilinguistic tokenbased code clone detection system for large scale source code, IEEE Trans. Softw. Eng. 28, no. 7.

[19] Fowler, M., (1999), Refactoring: Improving the design of existing code, Addison-Wesley, Boston, MA, USA.

[20] Barbosa, F. and Aguiar, A. (2011). Generic roles, a test with patterns In 18th Conference on Pattern Languages of Programs, PloP 2011 Oct 21-23, Portland, OR, USA

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

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

[23] Przybyłek, A.(2001). Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming , J. Bishop and A. Vallecillo (Eds.): TOOLS 2011, LNCS 6705, pp. 163–178.

[24] Kästner, C., Apel, S., Batory, D., 2007: A Case Study Implementing Features using AspectJ. In:11th International Conference of Software Product Line Conference (SPLC 2007), Kyoto, Japan

[25] Griswold, W.G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H., 2006: Modular Software Design with Crosscutting Interfaces. IEEE Software 23(1), 51–60 (2006)

[26] Riehle, D. and Gross, T. 1998. Role Model Based Framework Design and Integration." In Proceedings of the 1998 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98). ACM Press