Composing Classes
Roles Vs Traits

Fernando Barbosa[1] and Ademar Aguiar[2]
[1]*Escola Superior de Tecnologia, Instituto Politécnico de Castelo Branco, Av. do Empresário, Castelo Branco, Portugal*
[2] *INESC TEC and Faculdade de Engenharia da Universidade do Porto, Rua Dr. Roberto Frias, Porto, Portugal*

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

Abstract: Code replication has significant drawbacks in system maintenance. Code replication can have its origins in the composition limitations of the language. Several proposals have tried to overcome these limitations. A popular one is traits. However, traits do not support state or visibility control. Static roles are also a way of composing classes that has the benefits of traits and offers state, visibility control and other advantages as block renaming. We compare both approaches on how they are used to compose classes, and how they can be used to reduce code replication caused by composition limitations. As a case study we will compare how both approaches can reduce code replication by detecting and removing code clones within the JHotDraw framework. Results show that roles are capable of reducing a larger amount of replicated code than traits.

1 INTRODUCTION

Code clones, identical blocks of code, are a hint that the system needs to be refactored (Fowler, 1999). However code clones appear in most systems, specially in large ones (Mayrand et al., 1996) (Baxter et al., 1998). Code clones impair maintenance and evolution of a system (Baxter et al., 1998). One 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 evolution, comprehensibility and cost (Roy and Cordy, 2007).

One origin of clones is the lack of composition mechanisms (Mayrand et al., 1996; Baxter et al., 1998; Roy and Cordy, 2007). This makes it harder to deal with crosscutting concerns - concerns that a class must deal with but are not its main concern. When dealing with the same concern classes tend to use similar code. This is more frequent in languages without multiple inheritance, but multiple inheritance has so many practical problems that it has been left out of recent languages, like Java and C#.

Some clones could be avoided if a language had other composition mechanisms. Several proposals are available, like multiple inheritance, mixins (Bracha and Cook, 1990), traits (Ducasse et al., 2004; Scharli et al., 2003), features (Apel and Kästner, 2009) and aspects (Kiczales et al., 2001).

Traits can be seen as a set of methods that provide common behaviour. When a class uses a trait its methods are added to the class. The class also provides glue code to compose the several traits. Traits cannot store state. State is maintained by the class that uses the trait.

When a class plays a role the role methods are added to the class interface. Thus an object's behaviour is defined by the composition of all roles its class declares to play. A class can configure the role to its needs by configuring types and methods names. Roles support state and visibility control.

Composing classes using traits or roles can minimize the code replication due to limitations of the composition mechanism. To assess this we conducted an experiment to account how both approaches could be used to remove the replicated code found in the JHotDraw Framework. We briefly present the two approaches then compare them showing how they deal with conflict resolution, composition order, etc.

We identified code clones using a clone detecting tool, and grouped them according to their concerns. We then tried to develop a role and a trait for each concern, thus removing the clones. We developed roles for nearly all detected concerns, but couldn't do the same for traits.

Barbosa F. and Aguiar A..
Composing Classes - Roles Vs Traits.
DOI: 10.5220/0004424000630073
In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2013), pages 63-73
ISBN: 978-989-8565-62-4

We can summarize our paper contributions as: a comparison of roles and traits features, ways of reducing replicated code using traits and roles; a comparison of how roles and traits tackle the problem of reducing duplicated code and identifying which clones they can eliminate; a case study showing how each approach reduces replicated code in an open source system.

This paper is organized as follows. Section 2 presents Traits and Section 3 presents roles. In Section 4 we compare the two approaches. Section 5 shows how to remove clones using roles and using traits and section 6 presents the JHotDraw framework case study. Related work is presented in section 7 and section 8 concludes the paper.

2 TRAITS IN A NUTSHELL

Traits are units of code reuse and a class can be constructed using several traits (Ducasse et al., 2004; Scharli et al., 2003). Traits have a flattening property: a class can be seen indifferently as a collection of methods or as composed by traits. The fact that the class can be seen as a whole promotes understanding and the fact that it can be composed promotes reuse.

In Traits a class can be constructed by using inheritance and by adding traits. The class must supply all state variables and glue code. The glue code is the set of methods that the trait requires the class to provide (for example, accessor methods for the state variables). Thus a class can be decomposed into a set of coherent features and the glue code connects the various features together.

According to (Ducasse et al., 2004), Traits have the following properties:

– A trait provides methods that implement behaviour
– A trait requires a set of methods that serve as parameters for the provided behaviour.
– Traits do not specify state variables, and methods provided by traits never access state variables.
– Classes and traits can be composed from traits.
– The composition order of traits is irrelevant.
– Conflicting methods must be explicitly resolved.
– Trait composition does not affect the semantics of a class: the meaning of the class is the same as it would be if all of the methods obtained from the trait(s) were defined directly in the class.
– Similarly, trait composition does not affect the semantics of a trait.

A class can redefine its superclass's and its trait's methods. Conflicts arise when unrelated traits have methods with the same signature. The conflict must be solved explicitly by redefining the conflicting method in the class. The conflict is thus resolved locally. To access the conflicting methods Traits support aliases. It works by giving an alias to a method so it can be used in the class without trouble. To prevent conflicts from occurring in the first place traits also support the exclusion of methods.

Some attempts to bring traits into Java-like languages have been made (Quitslund and Black, 2004; Smith and Drossopoulou, 2005). To compare the Trait approach to the Role approach we used Chai (Smith and Drossopoulou, 2005). Since Chai is an extension to the Java language and we intend to use our JavaStage language, that is also an Java extension, we can argue that the differences between the final code is due integrally to each approach and not to the underlying language. The traits examples in this paper are presented using the Chai syntax and derive from the example shown in (Smith and Drossopoulou, 2005).

Figure 1 shows trait declaration in Chai and its use by classes. We can see requirement of methods

```
class Circle {
  int radius;
  int getRadius() { ... }
}
trait TEmptyCircle {
  requires {  void drawPoint(int x, int y);
              int getRadius();  }
  void draw() { ... }
}
trait TFilledCircle {
  requires {  void drawPoint(int x, int y);
              int getRadius();  }
  void draw() { ... }
}
```

```
trait TScreenShape {
  void drawPoint(int x, int y) {...}
}
trait TPrintedShape {
  void printPoint(int x, int y){...}
}
class ScreenEmptyCircle extends Circle
            uses TEmptyCircle,TScreenShape { }
class PrintedFilledCircle  extends Circle
             uses TFilledCircle,TPrintedShape {
    alias { void printPoint(int x, int y)
              from TPrintedShape as
            void drawPoint(int x, int y)  }
}
```

Figure 1: Trait example (adapted from (Smith and Drossopoulou, 2005).

in the TEmptyCircle: it offers a draw method and requires the class to provide the drawPoint and getRadius, with the specified signature. The same methods are also required by TFilledCircle. The code also shows a Circle class, representing a circle, and two subclasses composed by traits and that inherit from Circle. The ScreenEmptyCircle class is an empty circle that can be drawn in the Screen, so it uses TEmptyCircle and TScreenShape. The methods required by TEmptyCircle are supplied by Circle and TScreenShape, so ScreenEmptyCircle does not need to provide them itself. PrintedFilledCircle is a filled circle than can be printed in a printer, so it inherits from Circle and uses TFilledCircle and TPrintedShape. TFilledCircle required methods are supplied by Circle and TPrintedShape. In the TPrintedShaped case the class needed to alias the trait method for the required name.

For more information on Traits we refer to (Ducasse et al., 2004; Scharli et al., 2003) and for the Chai language syntax we refer to (Smith and Drossopoulou, 2005).

3 ROLES IN A NUTSHELL

We use roles as a basic construct from which we can compose classes. Roles provide the basic behaviour for concerns that the classes must deal with but are not their main concern. Thus we can better modularize the construction of classes. We must mention that we use roles statically as defined by Riehle in (Riehle, 2000) where he uses them as static entities for modelling purposes. We do not use roles as dynamic entities that can be attached or detached from an object at runtime. Since there is much work on the use of dynamic roles (Steimann, 2000; Tamai et al., 2007; Herrmann, 2005) this must be mentioned to avoid confusion.

To program with roles we use JavaStage, an extension to Java (Barbosa and Aguiar, 2013). Examples in this paper use the JavaStage syntax. Figure 2 shows the role version of the trait example of Figure 1.

A role may define methods and fields including access levels. A class can play any number of roles, and can even play the same role more than once. A class playing a role is a player of that role. When a class plays a role all the non private methods of the role are added to the class. To play a role the class uses a plays directive and gives the role an identity. To refer to the role the class uses its identity. Roles can inherit from roles and can also play other roles.

A role may require the player to have specific methods. Those methods are stated in a requirement list, which indicates who must supply the method and the method signature. The Performer keyword indicates that the supplier is the player. Performer is used within a role as a place-holder for the player's type. This enables roles to declare fields and parameters of the type of the player.

JavaStage has a method renaming mechanism that allows the renaming of methods with a simple configuration. Each name may have three parts: a configurable one and two fixed. Both fixed parts are optional. The configurable part is bounded by #, like in the example: fixed#configurable#fixed

The name configuration is done by the class playing the role in the plays clause. To play the role the class must define all configurable methods.
It's possible to declare several versions of a method using multiple definitions of the configurable name. This way, methods with the same structure are defined once.

```
class Circle {
  int radius;
  int getRadius() { ... }
}
role EmptyCircle {
  requires Performer implements
                void #draw#(int x, int y);
  requires Performer implements int getRadius();
  void draw() { ... }
}
role FilledCircle {
  requires Performer implements
                void #draw#(int x, int y);
  requires Performer implements int getRadius();
  void draw() { ... }
}
```

```
role ScreenShape {
  void drawPoint(int x,int y){ ... }
}
role PrintedShape {
  void printPoint(int x,int y){  ...}
}
class ScreenEmptyCircle extends Circle {
  plays EmptyCircle(
           draw = drawPoint ) emptyCircle;
  plays ScreenShape screenShp:
}
class PrintedFilledCircle extends Circle {
  plays FilledCircle(
           draw = printPoint ) fillCircle;
  plays PrintedShape;
}
```

Figure 2: Role example, equivalent to the traits' example in Figure 1.

Role members have all the visibility control available to classes and a protected role member is accessible to its players and subroles. A protected class member is also accessible to roles. A class can reduce the visibility of the role members. If a class uses protected in the plays clause then all the public role methods are imported to the class as protected.

Class defined methods always take precedence over role methods and role methods take precedence over inherited methods. Conflicts may arise when a class plays roles that have methods with the same signature. When conflicts arise the compiler issues a warning. Developers can handle the conflict by redefining that method and calling the intended method. This is not mandatory because the compiler uses, by default, the method of the first role in the plays clause order.

JavaStage supports role constructors but does not allow direct role instantiation. For more information on roles and JavaStage we refer to (Barbosa and Aguiar, 2013)

4 A COMPARISON BETWEEN ROLES AND TRAITS

For comparing roles and traits we follow a few key points that both approaches must deal with and describe how each handled the situation.

Unit of Composition. In roles the unit of composition is the role while in traits it is the trait.

Inheritance. Roles and traits are targeted for single inheritance languages so there is no multiple inheritance support. Roles can play other roles and traits can use other traits. Both approaches also support a class using the same unit several times. In a class, to access the features of the superclass both approaches use the super keyword. In a role, however, the super keyword refers to the super role, as roles can inherit from other roles. In a trait it refers to the superclass of the composing class.

State Support. Roles can have state and it does not cause any conflict because to access role state the class must use the role identity thus no conflicts arise. Traits do not support state. Proposals to solve this introduced a significant complexity to the trait model and encapsulation problems (Cutsem et al., 2009). When modelling a concept we, often, need to express state. For example, to model a container we need a structure for storage. Forcing the composing class to supply that structure is rather breaking the container's encapsulation.

Conflict Resolution. Both approaches follow the same rules for method overriding. The class overrides methods from roles/traits and roles/traits override the class inherited methods. Conflicts may arise when methods with the same signature are provided by more than one unit. In traits the conflict must be resolved explicitly while in roles the method of the first played role is used (there is a compiler warning). In both cases it is the class composer that decides which method to use. In traits he can choose to exclude some methods so there is no conflict or he can redefine the method and use aliases to refer to each of the conflicting methods. In roles there is no exclusion and the class composer must redefine the conflicting method if he wishes to override the rule of using the method of the first role.

Composition Order. The order in which traits are composed is symmetric so order of composition is irrelevant. The same applies for the roles when there are no conflicting methods. When there are conflicting methods the order of the plays will dictate which method is used. This, however, is not mandatory as discussed in the previous topic.

Method Renaming vs Aliases. There is a fundamental difference between aliases in traits and method renaming in the roles. The traits aliases are used only by the class for distinguishing conflicting methods, the class interface is not affected. In roles the renaming affects the class interface. This means that a class may be able to tailor its interface to suit its needs and not be limited by the role interface. The renaming mechanism of the roles also allows renaming several methods in one go, while aliases in traits are made one by one. Roles renaming scheme can provide multiple versions of a method. Traits aliases can be applied to any method, while on roles only the configurable methods can be configured.

Flat and Composite View. Both approaches support a flat view of the class as well as a composite view. Thus a class can be seen as a set of methods, the flat view, or as being composed by several units of composition, the composite view. The class interface in both views is exactly the same. The main difference between the two is that a trait method is seen just like a class method, and a role method is always a role method and each reference to other methods will always refer to role methods. For example, suppose a trait that defines the methods foo and bar, where bar calls foo. If the class overrides the foo method then the trait bar method will call the foo method on the class not on the trait. The same situation is handled differently by roles. If the method bar of the role is called then it will call the foo method on the role and not on the class. For a role method to call a class method it must do it

explicitly using the perfomer keyword.

Visibility control. Traits have no visibility control. Freezable traits (Ducasse et al., 2007) compensate this by allowing classes to freeze/unfreeze methods, i.e., declare a method as private (freeze) or making it public (defrost). But there is no way to express access constraints between class and trait. For example, fields should be accessed directly only by the owner's code. Traits do no support this. Roles on the other hand support all Java access levels, so a specific interface between role and class is possible.

Stating Requirements. The use of generic types is a useful feature in most languages, especially for dealing with object collections. Traits can require methods from the class that uses them, but cannot impose restriction on generic types it interacts with. The requires statement of roles indicates the method signature and which type it is required from. This allows roles not only to require methods from the class but also from other collaborators types.

5 REMOVING CLONES

We want to assess if the extra units of composition roles and traits provide are capable of reducing code clones. To remove code clones refactorings (Fowler, 1999) are normally used. The ones most used for removing code clones are: Extract Method; Pull Up Method, Pull Up Field, Extract Superclass, Extract class and Form Template Method (Fanta and Rajlich, 1999; Komondoor and Horwitz, 2000) (Higo et al., 2004).

We identified three clone types where roles or traits can be applied to remove code clones that fall outside the scope of these refactorings or produce better results. The clone types all have method granularity, so if actual clones do not have method granularity other refactorings must be used. Clone types are: Clones with identical code; Clones with similar code but using different types; Clones with similar code but using different method names with or without different types

Clones with Identical Code. These clones have identical methods and/or fields. This could be handled by the Extract Class refactory, but we argue that this is one situation where roles/traits produce better results. Extract Class forces the original class to provide delegate methods to the newly created class. With roles and traits those methods are not required. We put the code in the role/trait and then compose the class using it.

The application of roles and traits is shown in Figure 3. The figure represents two classes with replicated code. Both classes have different, unrelated, superclasses so Pull Up Method and Extract Superclass cannot be used. The replicated code was placed in a trait and a role.

Both solutions are similar, the difference is that roles support state so they do not require the getX and setX methods and can even provide them. Traits require the class to supply those methods.

Clones with Similar Code but using Different Types. These could be handled by Extract Class, using type parameters. For example we can build a Company class that manages workers and we can build a PolyLine that stores points. Both classes will have code for adding and removing workers/points, so there will be replicated code between them, only

```
class A extends SuperA {
  private int x;
  int getX() { return x; }
  void setX(int x){this.x=x;}
  void foo( ) {  // more code
    x += 14;
  }
  void bar() { ... }
}
class B extends SuperB {
  private int x;
  int getX() { return x; }
  void setX(int x){this.x=x;}
  void foo( ) { // more code
    x += 14;
  }
  void bar() { ... }
}
```

```
trait TOne {
  requires{ int getX();
            void setX( int x );}
  void foo( ) { // more code
    setX( getX() + 14 );
  }
  void bar() { ... }
}
class A extends SuperA uses TOne{
  private int x;
  int getX() { return x; }
  void setX( int x){this.x = x;}
}
class B extends SuperB uses TOne{
  private int x;
  int getX() { return x; }
  void setX( int x){this.x = x;}
}
```

```
role ROne {
  private int x;
  int getX() { return x; }
  void setX(int x){this.x=x;}
  void foo( ){ // more code
    x += 14;
  }
  void bar() { ... }
}
class A extends SuperA {
  plays ROne r1;
}
class B extends SuperB {
  plays ROne r1;
}
```

Figure 3: Removing identical clones using roles and traits.

the stored type is different. We can create a unit responsible for this management. We show this example in Figure 4. For simplicity and space we used arrays and do not show the management code.

There is a limitation in traits that may render a solution impossible: traits cannot require methods from other sources other than the class that uses them. A possible example is the Observer pattern (Gamma et al., 1995), where subjects maintain a list of observers and notify them when changes occur using an update method. The observer management is similar to the container problem just described, so the same solution can be applied. The problem lies in calling the update method. For calling this method the Trait must specify the type of the observer otherwise it cannot call a method on it. Roles can solve this by requiring the Observer type to implement an update method, as shown in Figure 5, where a Figure class notifies FigureObservers whenever it is changed. The solution reuses the container role and just adds the notify method.

Clones with Similar Code but Using Different Method Names with or without different Types. These clones have identical code but the names of the methods are not identical. The used types may also be different. For example we could change the Company and Polyline example and change them so that each had different names. The company would have addWorker and removeWorker methods while PolyLine would have addPoint and removePoint.

Traits aliases do not cope with these changes as they only affect the methods internally. With traits we would have to uniform the methods names and then apply the previous topic solution. We show how this situation is handled by roles in Figure 6. The Company class also shows how we can use the multiple method version to produce an addWorker and an addEmployee method.

6 CASE STUDY

To compare how roles and traits are capable of reducing code replication we applied both to the JHotDraw framework. The framework defines the basic structure for a GUI based editor with tools, different views, user-defined graphical figures, etc.

6.1 Case Study Setup

We searched for replicated code with CCFinderX (Kamiya et al., 2002), an established clone detection tool used in aspect mining works (Ceccato et al., 2005).

We filtered clones inside the same file (same class), thus eliminating clones that could use Extract

```
trait TContainer<T>{
  requires {
    T[] getAll();
  }
  void add(T t){...}
  void remove( T t){
     ...
  }
}
class Company uses
 TContainer<Worker>{
  Worker arr[];
  Worker[] getAll(){
     return arr;
  }
}
class PolyLine uses
  TContainer<Point>{
  Point arr[];
  Point[] getAll() {
     return arr;
  }
}
```

```
role Container<T> {
  private T arr[];
  void add( T t ){...}
  void remove( T t){...}
  T[] getAll() {
    return arr;
  }
}

class Company {
 plays Container<Worker>
  cWorker;
}

class PolyLine {
  plays Container<Point>
    cPoint;
}
```

Figure 4: Removing identical clones with different types using roles and traits.

```
role Subject<T> extends Container<T>{
  requires T implements void update();
  void notify( ) {
    for( T t : getAll() )
      t.update();
  }
}
class Figure {
  plays Subject<FigureObserver> figSubject;
}
```

Figure 5: Defining requirements on collaborators types.

```
role Container<T> {
  private T arr[];
  void add#Thing#( T t ) { ... }
  void remove#Thing#( T t ) {...}
}
class Company {
  plays Container<Worker>( Thing = Worker,
        Thing = Employee ) cWorker;
}
class PolyLine {
  plays Container<Point>( Thing = Point
                        ) cPoint;
}
```

Figure 6: Removing identical clones with different methods and types using roles.

Method or similar. We want to assess traits and roles capability to reduce the code clones derived from compositional limitations, so we only want clones that are not removable with traditional refactorings.

The first result included 271 clones, reduced to 146 after filtering. These were manually inspected. 41 false clones were removed leaving a final 105 sets. Some clones only had similar structures, but as they focused on the same concern we did consider them. This will explain some unresolved concerns.

We grouped clones according to their concerns. This helped us decide which role/trait to develop. We identified 42 concerns, but 5 were removed (2 could be easily refactored, 1 was deprecated code and 2 were classes pending substitution).

We've decided not to change any class interface or any concern implementation, so the framework is unchanged. This can restrict roles/traits development but we want to assess how we can reduce code replication, not redesign the framework. Roles were developed and compiled with JavaStage (Barbosa and Aguiar, 2013) and traits developed with Chai (Smith and Drossopoulou, 2005).

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 Traits, and the ratio between the various solutions.

LOC are a good measure on the effort that each approach requires, because both use Java as the underlying language, the syntax of both solutions is analogous and we made an effort to uniform the LOC count. We counted as LOC the requirements statements that traits and roles use. We also counted as LOC the roles' plays directive and the traits' uses directive. This overhead can lead a small clone to have more LOC in the solution than in the original form but the fact that there is no clone gives the system a great modularity advantage.

In the cases where roles removed clones and traits did not, the table shows the roles features that allowed them to remove the clone. For the concerns that neither technique worked it states the reason why they failed.

6.2 Results Analysis

Table 1 shows that from the 38 concerns only 8 (21%) concerns were not resolved with roles. Traits failed to resolve 15 (39%) concerns. It also shows that traits were not able to resolve the clones that roles could not. The final outcome is better than these numbers indicate as we will discuss.

We can see from table 1 that roles never had worst results than traits, succeeding in 7 concerns where traits failed and fared better in 13 more concerns. This indicates that roles are, at least, as good as traits in reducing replicated code.

Concerns resolved with Roles and Traits. Comparing the LOC ratio of both approaches, in those concerns both resolved, one finds that in average roles only have 83% of the traits code and 68% of the original code, so the effort of developing the role system seems smaller.

In 6 concerns we were able to reuse roles from the role library developed in (Barbosa and Aguiar, 2012). From those, 3 are solvable with traits but we had to develop a special trait for each concern and could not reuse them from a library. This explains the great difference in LOC in these concerns.

Supporting state is the role feature responsible for the fewer code used in roles. The class instead of having to declare each field and provide getters and setters would have the field and methods defined in the role. This is no small advantage not only in LOC but also in terms of abstraction and encapsulation.

The multiple method version also enabled roles to have less code, because a single method definition can provide several methods.

Concerns resolved by Roles Only. Roles were able to resolve 7 concerns that traits did not. From these, 3 used roles from the library. Requiring and renaming methods from other participants is the feature that enables roles to solve more clones.

From all the concerns roles resolved, two exhibit a higher LOC than the original implementation: "Handle creation" and the "Polygon locator".

The "Handle creation" concern deals with the creation of handles for each figure. We placed the creation of the handles in a handle creator class that has a method for the handle creation for each class. That and the role overhead lead to more lines of code than the original implementation. But the role has an advantage over the original code: it can dynamically change the handle creator.

The "Polygon Locator" uses an anonymous class. In JavaStage roles cannot be applied to anonymous classes so we had to develop an inner class to play that role and then use it.

Unresolved concerns. A surprising result is that for the 2 concerns with the most clone sets and class involved neither technique works. This is because they are clones in the structure and not on the code itself. The "Creating undo activity" creates an UndoActivity object for each tool and command. Each has an UndoActivity inner class. Because inner class constructors have different parameters in

Table 1: Identified concerns with the number of associated clones and affected classes. It also shows the LOC for each approach and respective ratios.

*= reused role from library, br= block renaming, g= generics, mv= multiple versions, rp= requires from participant, s= state

	Concern	clone #	class #	Original LOC	Roles LOC	Roles/ Original	Traits LOC	Roles/ Traits	Traits/ Original
Resolved by Roles and Traits	Drawing Handles	8	15	64	40	63%	40	100%	63%
	Setting up the undo activity before executing a Command	2	8	56	44	79%	44	100%	79%
	BringToFront/SendToBack Commands	1	2	20	12	60%	12	100%	60%
	Handle creation	11	20	70	87	124%	87	100%	124%
	Drawing polygons	1	2	12	11	92%	11	100%	92%
	Palette Listener	1	2	20	17	85%	17	100%	85%
	DisplayBox persistence	2	5	35	12	34%	12	100%	34%
	DisplayBox handling	6	8	58	29	50%	60	48%	103%
	DesktopListener Subject	2	3	63	45	71%	55	82%	87%
	Changing connections	3	3	98	53	54%	65	82%	66%
	Finding connectable figure	1	3	98	53	54%	65	82%	66%
	Testing command executability	5	7	14	14	100%	15	93%	107%
	Floating text holder	2	2	47	36	77%	47	77%	100%
	DrawingViewListener Subject	2	4	63	26*	41%	47	55%	75%
	Setting text in a text Figure	2	2	36	22	61%	32	69%	89%
	Enumerator	1	3	33	11*	33%	37	30%	112%
	Figure Listener that resends notifications	2	3	35	23*	66%	37	62%	106%
	Menu enabling	1	2	20	14	70%	14	100%	70%
	Version control	1	2	12	9	75%	9	100%	75%
	Selected button manager	1	2	18	12	67%	16	75%	89%
	Text attributes management	2	2	206	120	58%	149	81%	72%
	Updating DrawingView Strategy	1	2	29	26	90%	32	81%	110%
	Connection insets computing	1	3	10	7	70%	7	100%	70%
Resolved only by Roles								**Roles features**	
	Undo/Redo Commands	1	2	32	31	97%		br rp g	
	Changing connection handles	1	2	20	19	95%		br rp g	
	Polygon and PolyLine Handles	3	2	32	28	88%		rp	
	Tools and Commands Dispatchers	6	4	89	32*	36%		br rp	
	Figure/Handle and Enumerator	1	2	33	2*	6%		br rp	
	Polygon locator	1	2	13	20	154%		rp	
	Drawing editor	1	3	54	28*	52%		mv br rp	
Unresolved				**Reason**					
	Desktop initial configurations	1	2	required too much configuration					
	Persistence (read/write)	3	6	similar but not quite identical code					
	UndoActivity	13	24	Undoactivity inner classes constructors					
	Creating UndoActivity	14	18	after other roles was just a line of code					
	Handle manipulation starting action	3	5	required 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					

number and types, roles and traits could not resolve this concern. Another example is the "Handle manipulation starting action": code is similar but not identical: methods have different parameters.

Another example is "Persistence": because figures must be streamed they have a write and read methods with similar, but not identical, structures.

Another unresolved concern is "DrawingView listener". The replicated code is redefining the original method for performance issues.

The unresolved "Desktop Initial configuration" deals with a Desktop's panel initialization. Each initialization is similar so we could configure a role/trait for each. But it would be easier to know how to configure the scroll pane.

Other unresolved concern was a single line like getSomeObject().doSomething(). The first method returns different objects that call different methods, so role configuration would take more LOC.

We would count only 4 unresolved concerns if we had not considered some concerns as clones.

6.3 Threats to Validity

We only considered a single system. However, the discussion in section 5 hints that results from other systems would also have roles performing better than traits. We need to do the same test with more systems to fully assess this.

The clone detection settings can affect the clones detected which would lead to different concerns. But we needed to reduce the amount of clone sets to a manageable number or there would be a greater number of false clones. We even used less than the limit of 30 tokens recommended in (Kamiya et al., 2002) to limit false clones. So we believe that our settings provided a good number of clones/concerns that make this case study results valid.

There could be biased results from having the same developers doing the role and traits approach. After all the authors are more experienced in roles than in traits. This could bias the results towards roles. To prevent this, we opted to base the study on clone detection and not on developing an alternate system from scratch.

The effort used to develop each approach was not taken into account. For that we would need to assess how teams of developers, each using a different approach, tackled the same problem. While the LOC number gives a hint on the effort required, and here roles have an advantage, it does not tell the entire story as already mentioned. However it does give some insights. One insight is that the use of state reduces the effort to develop roles by reducing the amount of glue code one must write to use traits. This also gives roles a better modelling capability because a role can model a concept that has state and behaviour as opposed to the traits' behaviour only modelling. We also reused roles from our role library, but traits equivalent could not be placed in a traits library, which means that the effort of developing roles was less than that of traits.

7 RELATED WORK

There are a number of dynamic role approaches like Object Teams (Herrmann, 2005), EpsilonJ (Tamai et al., 2007) and PowerJava (Baldoni et al., 2007). These are known for their capability to attach and detach roles from objects at runtime, something that (Cutsem et al., 2009) also supports for traits. ObjectTeams introduces 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 type. PowerJava has a similar concept – the institution. When an object wants to interact with an institution it must assume one of the roles the institution offers. In EpsilonJ roles are also defined as inner classes of a context. Roles are assigned to an object via a bind directive. It uses a requires directive similar to roles and traits. It also offers a replacing directive to rename methods names.

Feature Oriented Programming (FOP) (Apel and Kästner, 2009) decomposes the system into features. FOP relies on a step-wise refinement of applications by adding new features or refining existing ones. To compose a system we just state which features it has. The composition is made automatically with tool support, like AHEAD (Batory et al., 2004). This is a more powerful technique than roles or traits. AHEAD uses several tools for composing the code and extra files for configuring the composition step. Roles/Traits are programming languages that statically compose classes using only source code. AHEAD can be used to compose classes. For example, we can develop a class that defines the basic behaviour of a class, undistinguishable from a normal Java class, except that it has a feature keyword indicating to which feature it is associated to. We can then construct several refinements to that class. Each refinement indicates the added feature and the class it refines.

Package Templates (PT) (Krogdahl et al., 2005) use traditional java packages with a twist. Classes

defined in these packages are only directly available when the package is instantiated. When instantiated the classes can be tailored to the context of use by: getting additions; elements can be renamed; type parameters are given actual types. This tailoring is similar to roles as roles also support renaming and type parameters. PT may also impose restrictions on the various types via a constraints declaration that resembles roles requirement list. Classes in a PT can be merged with classes from other PT and can be used more than once in the same merging (like roles/traits can be used multiple times). The main difference between PT and roles is that PT, like traits, rely on inheritance to do the merging and roles rely on inner classes. Name clashes are resolved via renaming, which can be applied to fields and methods. The renaming cannot be used on the constraints. JavaStage on the other hand allows the renaming of required methods

Aspect-Oriented Programming is an approach that tries to modularize crosscutting concerns (Kiczales et al., 2001). AOP defines pointcuts to identify points in the executing program that may trigger a different execution path and advices that indicate the new execution path. While the modularization of crosscutting concerns is the flagship of AOP several authors disagree (Steimann, 2000; Przybyłek,v2011). The effects of pointcuts and advices, especially when several aspects have similar pointcuts, may be unpredictable. Thus simple changes in the class code can have unsought effects (Kästner et al., 2007).

The obliviousness feature of AOP means that a class is aspect unaware so aspects can be plugged or unplugged as needed. But it introduces problems in comprehensibility (Griswold et al., 2006). To understand the system we must know the classes and which aspects affect each class. This is a major drawback when maintaining a system, since the dependencies aren't always explicit and there isn't an explicit interface between both parts.

With roles/traits all dependencies are explicit and the system comprehensibility is increased (Riehle, 2000). Roles do not have the obliviousness of AOP because the class is aware of the roles it plays.

Caesar (Mezini and Ostermann, 2003) uses aspect technology to modularize crosscutting concerns and enhance reuse of aspects leading to a greater reduction of repeated code. Caesar uses an Aspect Collaboration Interface that decouples aspects binding and implementations by defining them in a separated module. Caesar does not allow method renaming.

Jiazzi (McDirmid et al., 2001) is based on Units (Flatt, 1998) and aims at building systems out of reusable components integrated with the language. Jiazzi has two types of units: Atoms (composed by java classes) and Compounds (composed by atoms or other compounds). Jiazzi supports the addition of features to classes without editing their source code. Roles/Traits could be used within Jiazzi to specify these new features. A trait/role could be used to add the same behaviour for different classes in the same unit, or for the same class but in different units.

8 CONCLUSIONS

We compared how the role and traits approaches deal with the composition problems that they aim to diminish by doing a study on how each can reduce replicated code, especially the replicated code that derives from lack of compositional mechanisms in single inheritance languages.

The outcome of the study showed that roles are more reusable than traits, because roles support state, have a renaming mechanism that tunes them to the class purpose and can even provide several versions of a method in a simple way. We validated our approach developing roles for the JHotDraw framework and eliminated nearly all duplicated code. Doing the same test for traits showed that they cannot eliminate all the clones roles were capable of. We even reused some roles from our role library showing that they are really reusable.

REFERENCES

Apel, S., Kästner, C. (2009): An Overview of Feature-Oriented Software Development, in Journal of Object Technology, vol. 8, no. 5, July–August 2009.

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

Barbosa, F. and Aguiar, A. (2012). Roles as Modular Units of Composition. In 7th International Conference on Evaluation of Novel Approaches to Software Engineering, June, 29-30, Wroclaw, Poland.

Barbosa, F. and Aguiar, A. (2013). Using Roles to Model Crosscutting Concerns. In Aspect Oriented Software Devlopment (AOSD3), March 24–29, Fukuoka, Japan.

Batory, D., Sarvela, J. N. and Rauschmayer, A., Scaling Step-Wise Refinement. IEEE TSE, 30(6), 2004.

Baxter, I., Yahin, A. Moura, L., Sant'Anna, M. and Bier. L. (1998). Clone Detection Using Abstract Syntax Trees. In Proc. of Int. Conf. on Software Maintenance.

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

Ceccato, M., Marin, M., Mens, K., Moonen, L, Tonella, P. and Tourwe, T (2005). A qualitative comparison of three aspect mining techniques, *Proc. of the Inter. Workshop on Program Comprehension*, Washington.

Cutsem, T., Bergel, A., Ducasse, S. and Meuter, W. (2009): Adding State and Visibility Control to Traits Using Lexical Nesting, *Proc. of ECOOP 2009*, Italy.

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

Ducasse, S., Wuyts, R., Bergel, A., and Nierstrasz, O. (2007): User-changeable visibility: Resolving unanticipated name clashes in traits. *In Proceedings OOPSLA*, New York, NY.

Fanta, R., Rajlich, V. (1999): Removing Clones from the Code. Journal of Software Maintenance: Research and Practice, Volume 11(4):223-243.

Flatt, M. and Felleisen, M. (1998) Units: Cool modules for HOT languages. *In Proc. of PLDI*, May 1998.

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

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

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

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

Higo, Y., Kamiya, T., Kusumoto, S., Inoue. K. (2004): Refactoring Support Based on Code Clone Analysis. *In Proceedings of the 5th International Conference on Product Focused Software Process Improvement (PROFES'04)*, Kansai Science City, Japan.

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.

Kästner, C., Apel, S., Batory, D., 2007: A Case Study Implementing Features using AspectJ. In: *11th Inter. Conference of Software Product Line*, Kyoto, Japan.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold. W. G., (2001): An overview of AspectJ. In proceedings of *ECOOP 2001*, Budapest, Hungary.

Komondoor, R. and Horwitz SS. (2000): Semantics-Preserving Procedure Extraction. *In Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pp. 155-169, Boston, MA, USA.

Krogdahl, S., Møller-Pedersen, B., Sørensen, F. (2005): Exploring the use of Package Templates for flexible reuse of Collections of related Classes, *in Journal of Object Technology*, vol. 8, no. 7.

Mayrand, J., Leblanc, C. and Merlo, E. Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics. *In Proc. of the International Conference on Software Maintenance*, 1996.

McDirmid, S., Flatt, M. and Hsieh, W.C. Jiazzi: new-Age Components for Old-Fashioned Java", *OOPSLA 2001*.

Mezini, M. and Ostermann, K. 2003. Conquering Aspects with Caesar. *In Proc. of AOSD 2003*, pages 90 – 99.

Przybyłek, A. /2011). Systems Evolution and Software Reuse in Object-Oriented Programming and Aspect-Oriented Programming , *TOOLS 2011, LNCS 6705*.

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

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

Roy, C. and Cordy, J. (2007) A Survey on Software Clone Detection Research. Tech. Report 2007-451, School of Computing, *Queen's University at Kingston*.

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

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

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

Tamai, T., Ubayashi, N., and Ichiyama, R., (2007): Objects as Actors Assuming Roles in the Environment, in Software Engineering For Multi-Agent Systems V, Lecture Notes In Computer Science, vol. 4408. *Springer-Verlag*, Berlin.