SCALA ROLES
A Lightweight Approach towards Reusable Collaborations

Michael Pradel[1*] and Martin Odersky[2]
[1] *TU Dresden, 01062 Dresden, Germany*
[2] *EPFL, 1015 Lausanne, Switzerland*

Keywords: Object-orientation, programming languages, roles, collaborations, reuse, dynamic proxy.

Abstract: Purely class-based implementations of object-oriented software are often inappropriate for reuse. In contrast, the notion of objects playing roles in a collaboration has been proven to be a valuable reuse abstraction. However, existing solutions to enable role-based programming tend to require vast extensions of the underlying programming language, and thus, are difficult to use in every day work. We present a programming technique, based on dynamic proxies, that allows to augment an object's type at runtime while preserving strong static type safety. It enables role-based implementations that lead to more reuse and better separation of concerns.

1 INTRODUCTION

Software objects represent real-world entities. Often, those entities fundamentally change during their lifetime. Also, there may be different views on an object depending on the context. One way to express this in a programming language is by *dynamically adapting the type* of objects by enhancing and reducing the set of visible fields and methods.

Another issue tackled in this paper is *reuse*, one of the major goals in software engineering. Reuse is the process of creating software systems from existing software rather than building it from scratch. In object-oriented programming languages, classes abstract over sets of objects according to common properties and behavior. Though, relations between objects are usually manifold. In a traditional approach, this is reflected by a complex network of interconnected classes. As different contexts in which objects collaborate are not explicit in a purely class-based approach, reusing a particular object collaboration is difficult.

The software modeling community has been discussing since long a promising solution: *role modeling* or *collaboration-based design* (Reenskaug et al., 1996; Riehle, 2000; Steimann, 2000). The main idea is that objects play *roles*, each describing the state and behavior of an object in a certain context. In other words, a role provides a particular view on an object. Similar to objects, roles can be related to each other, for instance by references, field accesses, and method calls. A number of related roles form a *collaboration*. As objects can be abstracted by classes, roles are abstracted by *role types*, that can be thought of as partial classes. A typical example is an instance of a class `Person` that may play the role of a `Professor` (related to a role `Student`) and the role of a `Father` (related to `Child` and `Mother`). We can separate both concerns into two collaborations `University` and `Family`. Roles and collaborations permit to explicitly describe interwoven relations of objects, and hence, provide an interesting reuse unit orthogonal to classes.

While roles are accepted in modeling (see for example *UML collaborations* (Object Management Group OMG, 2007)), they are rare in today's programming languages. Most of the proposed solutions (Bäumer et al., 1997; Herrmann, 2007; Smaragdakis and Batory, 2002) either do not fully conform to the commonly used definition of the role concept (Steimann, 2000) or require extensive changes in the underlying programming language. The latter makes them hard to employ in every day work since existing tools like compilers and debuggers cannot be used. In this work, we propose a lightweight realization of roles in the Scala programming language (Odersky, 2008). It can be realized as a library, that is, without any language extension.

*This work has been accomplished during a stay of the first author at the second author's lab.

Pradel M. and Odersky M. (2008).
SCALA ROLES - A Lightweight Approach towards Reusable Collaborations.
In *Proceedings of the Third International Conference on Software and Data Technologies - PL/DPS/KE*, pages 13-20
DOI: 10.5220/0001873700130020

Listing 1 gives a first glimpse of our solution. In the first line, a class `Person` is instantiated. The contexts in which the person occurs are represented by collaborations, that are instantiated in lines 4 and 5. In the following, the object is accessed playing the roles of a professor and a father (lines 8 and 11), as well as without any role (line 14). More details on our approach and other examples follow in Sections 3 and 4.

```
1  val p = new Person("John")
2
3  // collaborations are instantiated
4  val univ = new University{}
5  val fam = new Family{}
6
7  // person in the university context
8  (p as univ.professor).grade_students()
9
10 // person in the family context
11 (p as fam.father).change_diapers()
12
13 // person without any role
14 p.name   // "John"
```

Listing 1: A person playing different roles.

The major thrust of this paper is to show that programming with roles is feasible in a lightweight fashion. More specifically, our contributions are the following:

- A programming technique for roles that might be applicable to most object-oriented programming languages. It is based on compound objects managed by dynamic proxies.
- A role library for Scala that allows to dynamically augment an object's type while preserving strong static type safety.
- A novel reuse unit, dynamic collaborations, that captures the relations of objects in a certain context.

The following section summarizes the idea of roles and collaborations in object-oriented programming languages. Our approach is explained in detail in Section 3, followed by Section 4 giving concrete examples of augmenting types and reusing the Composite design pattern as a collaboration. Finally, a short overview of similar approaches is given in Section 5.

2 OBJECTS AND ROLES

There are many different definitions of roles in the literature (Guarino, 1992; Kristensen and Osterbye, 1996; Reenskaug et al., 1996; Riehle, 2000). Steimann (Steimann, 2000) gives a comprehensive overview and presents a list of role features, whereof the following are the most essential.

- A role comes with its own properties and behavior.
- Roles are dynamic, that is, they can be acquired and abandoned by objects at runtime. In particular, a role can be transferred from one object to another.
- An object can play more than one role at the same time. It is even possible to play two roles of the same role type multiple times (in different contexts).
- The state and the members of an object may be role-specific. That is, binding a role to an object can change its state as well as its fields and methods.

Roles are a relative concept, in the sense that a role never occurs alone, but always together with at least one other role. Related roles form a collaboration as shown in Figure 1, where a role drawn on top of an object indicates that the object plays the role. Possible relations between roles, such as references, field accesses, and method calls, are abstracted by a connecting line.

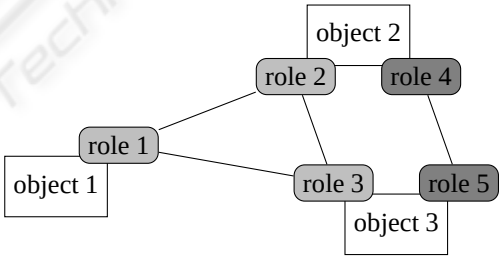

Figure 1: Roles 1 to 3, as well as roles 4 and 5 form two distinct collaborations that can describe independent concerns.

For a concrete example of the problem collaborations aim to solve, consider a graphics library containing a class `Figure` and two subclasses `BorderFigure` and `TextFigure` (Listing 2). These classes contain members representing properties like colors or the text in a `TextFigure`. Furthermore, we want to nest figures, for instance putting a `TextFigure` inside a `BorderFigure`. This can be realized with the Composite pattern (Gamma et al., 1995) which creates a tree-like data structure while allowing clients to treat individual objects and compositions of objects uniformly.

The code related to the Composite pattern is highlighted in Listing 2. We argue that, instead of being added to the figure classes, it should be extracted into

```
1  class Figure {
2    var bgColor = white
3
4    def addChild(f: Figure)
5    def removeChild(f: Figure)
6    def getParent: Figure
7    protected def setParent(f: Figure)
8  }
9
10 class BorderFigure extends Figure {
11   var borderColor = black
12
13   def addChild(f: Figure) = { /* ... */ }
14   // implementations of other
15   // abstract methods
16 }
17
18 class TextFigure extends Figure {
19   var text = ""
20
21   // implementations of abstract methods
22 }
```

Listing 2: A figure hierarchy implementing the Composite design pattern.

a collaboration. This approach has two main advantages:

- *Separation of concerns.* Figures have a number of inherent properties (in our example colors and text) that should be separated from the concern of nesting them.
- *Reuse.* Instead of implementing the pattern another time, we can reuse an existing implementation and simply attach it where needed.

Moving the highlighted code into supertypes is a reasonable solution in some cases. A role-based approach, however, provides more flexibility since behavior can be attached at runtime to arbitrary objects. Consequently, it can be applied without changing the type hierarchy of the graphics library, and even without access to their source code.

We propose to implement the Composite pattern as a collaboration with two roles `parent` and `child`. As a result, to access an instance of one of the figure classes being part of a composite, one can simply attach the parent or child role to it. As we will explain in more detail in the following sections, this is realized with the `as` operator in our approach. For instance, `someFigure as composite.parent` enhances the object `someFigure` with members related to being a parent in a composite.

3 COMPOUND OBJECTS WITH DYNAMIC PROXIES

The purpose of this paper is to show how roles and collaborations can be realized in programming. This section explains our solution conceptually and shows details of our implementation in Scala that may be interesting for similar implementations in other languages. Our approach benefits from flexible language features of Scala, such as implicit conversions and dependent method types, and from the powerful mechanism of *dynamic proxies* in the Java API. The latter can be used since Scala is fully interoperable with Java. However, the described programming technique is not limited to a specific language. As we will argue at the end of this section, the essence of it may be carried over to other languages than Scala.

A major question is whether to implement roles with objects or as a first-class language construct. We opted for a lightweight solution that realizes roles as objects which are attached to core instances dynamically. One advantage is that the underlying programming language need not to be changed and existing tools like compilers can be used without modifications. However, this leads to the problem of having multiple objects where only one is expected by programmers. For instance, a person playing the role of a professor is represented by one core object of type `Person` and one role object of type `Professor`. Issues arising from that situation have been summarized as *object schizophrenia* (Harrison, 1997). The main problem to resolve is the unclear notion of object identity that can, for instance, lead to unexpected results when comparing objects.

We argue that one can avoid such confusion by regarding a core object with temporarily attached role objects as a *compound object*, as depicted in Figure 2. The compound object is represented to the outside by a dynamic proxy that delegates calls to the appropriate inner object.

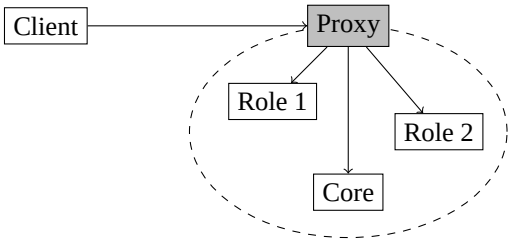

Figure 2: The proxy intercepts calls to the compound object and delegates them via an invocation handler.

A dynamic proxy is a particular object provided by the Java API. Its type can be set dynamically when creating it through a list of Java interfaces. Internally,

dynamic proxies are realized by building and loading an appropriate class file at runtime. The behavior of the proxy is specified reflectively by an *invocation handler*, that is, an object providing an `invoke` method that may delegate method calls to other objects.

We use a dynamic proxy to represent a role-playing object. Thus, its type is made up of the core object's type and the types of the role objects that are currently bound to it. The invocation handler of the proxy has a list of inner objects, one core object and arbitrary many role objects, and delegates calls to the responsible objects. Policies mapping method calls to inner objects may be specified if needed. A simple default is to reflectively delegate to role objects whenever they provide the required method and to the core object otherwise, such that roles may override the behavior of their core object.

Managing the compound object, creating a dynamic proxy with the appropriate type, and configuring the invocation handler is hidden from the user through a single operator called `as`. The expression `object as role` allows to access an object playing a certain role by temporarily binding `role` to `object`.

In Scala, all method calls can be written as infix operators. Hence, `object as role` is equivalent to `object.as(role)`. However, we want to bind roles to arbitrary objects, that is, we cannot assume `object` to provide an `as` method. In contrast, we can easily add the inverse (but less intuitive to use) method `playedBy` to role objects. The trick is to use Scala's implicit conversions (Odersky et al., 2008) to turn a method call `object.as(role)` into `role.playedBy(object)`. An implicit conversion is a special method inserted by the compiler whenever an expression would not be otherwise type-correct.

```
1 implicit def anyRef2HasAs[Player <: AnyRef]
2     (core: Player) =
3   new HasAs[Player](core)
4
5 class HasAs[Player <: AnyRef]
6     (val core: Player) {
7   def as(r: Role[Player]) = r.playedBy(core)
8 }
```

Listing 3: An implicit conversion that adds the method `as` to arbitrary objects.

The implicit conversion in lines 1 to 3 of Figure 3 wraps a core object into an instance of `HasAs`, a class providing the required `as` method. The method `anyRef2HasAs` has a type parameter `Player` which is inferred from the type of the argument `core`. `Player` is restricted by the upper bound `AnyRef`, Scala's equivalent to Java's `Object`. The `as` method simply calls `playedBy` on the role object (line 7). This returns a proxy object having the type of the core object, `Player`, extended with the type of the role object, `r.type`. As a result, roles can be dynamically attached to arbitrary objects writing `object as role`, which gives type-safe access to core and role members.

An interesting detail to note here is that the return type of `as` is a *dependent method type*, an experimental feature of Scala. The return type of `playedBy`, and consequently also of the `as` method, is `Player with r.type`. Hence, the return type of the method depends on the value of its argument `r`.

3.1 Object Identity

Another issue is to provide a clear notion of *object identity*. We argue that the identity of an object should be the same independent of whether roles are attached or not. For instance, a person being a father is still the same person, and consequently, object identity should reflect this. To clarify the problem, consider comparing objects and role-playing objects. Four kinds of comparison are possible:

```
(1)            object == (object as role)
(2)   (object as role) == object
(3)   (object as role) == (object as role)
(4)  (object as role1) == (object as role2)
```

To achieve (seeming) object equality between objects and role-playing objects, we modify identity-related methods of dynamic proxies. We use the fact that `==` and the `equals` method are equivalent in Scala. That is, the expressions `x == y` and `x.equals(y)` give the same result. We define `equals` and `hashCode` of proxies such that they map to the implementation of the core object, and, in case the right-hand operator of `==` is a proxy as well, compare with its core object. Although this solves the problem for expressions 2 to 4, it unfortunately does not for expression 1 since we cannot modify the `equals` and `hashCode` methods of arbitrary objects using a library approach. One possible solution would be to require the type of core objects to inherit from a type `RolePlayer` which contains an adapted `equals` methods. If the argument of `equals` is a proxy, it would compare with its core object, and otherwise fall back to the default implementation of `equals`. However, this makes adding roles to arbitrary objects impossible. Finding a satisfactory solution for this issue remains as future research.

So far, we have shown how arbitrary objects can be dynamically enhanced with new functionality by binding roles to them. Our solution provides strong static type safety, that is, only members of either the core object or a role object can be accessed without

a type error. Moreover, obstacles arising from object schizophrenia can be solved with a compound object represented by a dynamic proxy and an adapted notion of object identity.

3.2 Collaborations with Nested Types

In the following, we delve into another important aspect of roles, namely grouping them into collaborations. Per definition, roles do not occur alone, but describe the behavior of an object in a certain context. This context is given by other roles yielding a set of related roles called collaboration.

One of the basic principles of Scala is nesting of types, allowing related types to be grouped and extended together by extending their outer type. Following this principle, a collaboration is presented by an outer trait[2] whose inner traits represent its roles. Our role library provides a trait `Collaboration` that concrete collaborations must extend. It contains an inner trait `Role` that must be extended to define concrete role types. Most of the details like the `playedBy` method creating the dynamic proxy, are implemented in these base traits, such that collaboration developers do not have to bother with it. Listing 4 shows a simplified version of the `Collaboration` trait.

```
1  trait Collaboration {
2    trait Role[Player <: AnyRef] {
3      def playedBy(core: Player):
4          Player with this.type = {
5        val handler =
6          new InvocationHandler(this, core)
7        createProxy(core, handler)
8          .asInstanceOf[Player with this.type]
9      }
10
11     private def createProxy
12         (core: Player,
13          handler: InvocationHandler) = {
14       val interfaces: Array[Class] =
15         getInterfaces(this, core)
16       Proxy.newProxyInstance
17         (core.getClass.getClassLoader,
18          interfaces, handler)
19     }
20   }
21 }
```

Listing 4: The collaboration trait that concrete collaborations extend.

At first, we create an invocation handler (line 6). It realizes the delegation of all incoming calls to either the core object or the role object and adds special treatment for the methods `equals` and `hashCode`. Then, `createProxy` is called. It reflectively retrieves Java interfaces for the core object and the role object (line 15) and instantiates a dynamic proxy via the Java API. The proxy has the type of the core object, `Player`, and of the role, `this.type`. Since `newProxyInstance` simply returns a `java.lang.Object`, we down-cast the proxy to `Player with this.type` before returning it to the user (line 8). This cast can be done safely as we configured the proxy to have exactly this type. The benefit is that user code can be type-checked and no further casts are necessary.

In this section, we have shown how objects with dynamically changing types can be expressed with dynamic proxies. Furthermore, the flexibility of Scala permits to introduce a convenient syntax (the `as` operator), and hence, hide unnecessary details from the user. Collaborations can be expressed with nested types.

Although we implement our approach in Scala, we argue that its essence depends only on a small set of language features, and hence, can be transferred to other programming languages as well. There are two basic ingredients: first, one requires a way to dynamically create proxies whose type and implementation can be specified reflectively. Second, a notion of inner types is needed for collaborations. With minor adaptations, they can be realized with inner classes like those of Java. Other language features we used for our implementation, such as implicit conversions and dependent method types, help in providing a convenient syntax for using roles, but are not absolutely necessary.

4 SCALA ROLES IN ACTION

While the above is part of a library, the following section explains our approach from the perspective of collaboration developers and users. Collaboration-based programming has two major benefits. First, binding roles to objects and accessing them type-safely leads to a kind of *type dynamism*. It allows to enhance and reduce the set of visible members of objects at runtime without accessing its source code. Second, collaborations provide a *reuse unit* orthogonal to classes by encapsulating the behavior of multiple related objects. They focus on one specific aspect of a program, and thus, extract related code fragments. This leads to better separation of concerns and, possibly, reuse of a collaboration in different contexts. In this section we give concrete examples illustrating both benefits.

[2] A trait is a type similar to a class, however, providing a safe form of multiple inheritance. Traits can be thought of as interfaces with an implementation. See (Odersky et al., 2008) for further details.

4.1 Persons and their Roles

A classical example for roles are persons behaving specifically depending on the context, in other words, persons having different roles. Consider, for instance, the relation between a student at a university and his supervisor. The student gains motivation when being advised by the supervisor and wisdom when working, where the amount of gained wisdom depends on the student's current motivation. Assuming we have a class `Person` that may also occur in other contexts, supervisor and student can be modeled as roles of it. Listing 5 shows a collaboration with two roles `supervisor` and `student`, that are instances of the role types `Supervisor` and `Student`.

```
1  trait ThesisSupervision
2      extends Collaboration {
3    val student = new Student{}
4    val professor = new Professor{}
5
6    trait Student extends Role[Person] {
7      var motivation = 50
8      var wisdom = 0
9      def work = wisdom += motivation/10
10   }
11   trait Supervisor extends Role[Person] {
12     def advise = student.motivation += 5
13     def grade =
14       if (student.wisdom > 80) "good"
15       else "bad"
16   }
17 }
```

Listing 5: A collaboration describing the relation between a student and a supervisor.

A concrete collaboration must extend the abstract trait `Collaboration`. Doing so, it inherits the inner trait `Role` that can be extended by concrete roles. `Role` takes a type parameter that specifies the type of possible core objects playing the role. For roles that may be bound to arbitrary objects, `AnyRef` can be passed.

Listing 6 depicts how to use a collaboration. Paul supervises Jim's master project, and is, in his role as a PhD student, himself supervised by Peter, a professor. To use a collaboration, it must be instantiated (lines 8 and 9). Persons are accessed playing a certain role with the `as` operator. A role must be qualified with a collaboration instance. The main benefit of instantiating collaborations is that roles may be used multiple times in different contexts.

Note that Paul plays different roles in the example. While he occurs as Jim's supervisor in line 12, he takes the role of a student in line 14. In each case, paul (seemingly) has a different type, and thus, offers different members. As our solution provides

```
1  // a master student
2  val jim = new Person("Jim")
3  // a PhD student
4  val paul = new Person("Paul")
5  // a professor
6  val peter = new Person("Peter")
7
8  val master = new ThesisSupervision{}
9  val phd = new ThesisSupervision{}
10
11 (jim as master.student).work
12 (paul as master.supervisor).advise
13
14 (paul as phd.student).work
15 (peter as phd.supervisor).grade
16 (peter as phd.supervisor).name
```

Listing 6: Usage of the `ThesisSupervision` collaboration. The person Paul plays different roles depending on the context.

strong static type safety, calling the method `advise` on `(paul as phd.student)` would result in a type error during compilation.

It is also noteworthy that role-playing objects still have the type of their core object. For instance, in line 16, the field `name` that is defined in `Person` can still be accessed. Furthermore, a role itself can always access its current core object using a method `core`. Hence, the implementation of the student role can, for example, access the student's name via `core.name`.

The state of the roles in a concrete collaboration instance is preserved between different uses of the `as` operator. These *stateful roles* allow transferring a role and its state from one core to another. For instance, suppose Peter retires as a professor at the end of Listing 6. A new professor can take over the supervision of Paul by binding the supervisor role to him: `newProf as phd.supervisor`.

4.2 Composite Design Pattern

The above example illustrates how roles enable runtime type enhancements. The second part of this section focuses on another benefit of roles and collaborations, namely reuse. In particular, we show how the Composite design pattern (Gamma et al., 1995) can be represented as a reusable collaboration.

Expressed in terms of roles, the pattern essentially consists of a parent role and a child role (Riehle, 1997). Let us define a collaboration similar to that of Listing 5 with two roles `parent` and `child` providing the functionality of a composite, that is, methods `addChild`, `removeChild`, etc. Having exactly one instance of each role in the collaboration would imply dealing with a new collaboration instance for each parent-child relation between two objects. As a more convenient solution, we propose *role mappers*, helper

objects creating a new role instance whenever a new core object requires a certain role. Before delving into details of its implementation, Listing 7 shows how to use the Composite collaboration.

```
1  class Figure {
2    var bgColor = white
3  }
4  class BorderFigure extends Figure {
5    var borderColor = black
6  }
7  class TextFigure extends Figure {
8    var text = ""
9  }
10
11 val f1 = new Figure
12 val f2 = new TextFigure
13 val f3 = new BorderFigure
14 val f4 = new TextFigure
15
16 val c = new Composite[Figure]{}
17 implicit def figure2parent(f: Figure) =
18   f as c.parent
19 implicit def figure2child(f: Figure) =
20   f as c.child
21
22 f1.addChild(f2)
23 f1.addChild(f3)
24 f3.addChild(f4)
25
26 f1.getChild(0) // f2
27 f4.getParent   // f3
```

Listing 7: With the Composite collaboration, figures can be treated as members of a composite without containing the implementation of the pattern.

In contrast to Listing 2, the figure classes do not contain any source code for figures being a composite (lines 1 to 9). Instead, we instantiate the Composite collaboration in line 16 and parametrize it with the desired type of core objects. To enhance the readability of the following code, two implicit conversions are defined in lines 17 and 19. They cause figures to be converted into figures playing either the parent role or the child role. Hence, the figures can be used as if they contained composite members, such as `addChild`. Alternatively, we could also use the `as` operator explicitly, for instance, writing `(f4 as c.child).getParent` in line 27.

Contrary to the thesis supervision example, there exist an arbitrary number of instances of each role type in the Composite collaboration. Instead of instantiating roles statically as in Listing 5, there are two role mappers `parent` and `child`. A role mapper deals with the binding between core objects and role instances, creating new role instances on demand and reusing existing ones for already known core objects. Similarly to a role, a role mapper provides a `playedBy` method. Hence, using the same syntax as for roles with a fixed number of instances per collaboration, role mappers allows for arbitrary many of them. Whether to use roles or role mappers is a design decision of collaboration developers.

5 RELATED WORK

There are a couple of other interesting approaches towards implementing roles. The Role Object pattern (Bäumer et al., 1997) describes a design that splits one conceptual object into a core object and multiple role objects, each enhancing the core object for a different context. Core classes and role classes extend a common superclass that clients deal with. Clients add (remove) roles to (from) a core object by calling appropriate methods on it and passing a role descriptor as argument. We developed the ideas of the Role Object pattern focusing on two major drawbacks: first, clients can only dynamically detect if a core object provides a certain role and if so, must down-cast the role object before invoking role-specific methods. Hence, instead of having static type safety, programmers need to deal with runtime checks. Second, the role object pattern suffers from the problem of object schizophrenia; thus, clients must pay attention to not rely on object identity.

Steimann proposes two independent type hierarchies, one for classes (called *natural types)* and one for role types, and presents a role-oriented modeling language formalizing his approach (Steimann, 2000). We do not adopt this idea mainly for pragmatic reasons, since its realization in an existing programming language demands substantial changes to the type system. Also, two strictly separated type hierarchies contradict one of the properties of roles in (Steimann, 2000), namely roles playing roles.

A recent and very inspiring work on roles is ObjectTeams/Java (Herrmann, 2007). This is an extension of Java adding first-class support for roles, role types, and collaborations (called *teams*). Collaborations are represented as special classes, *team classes*, whose inner classes are considered to be role types. The problem that role objects do not directly conform to the type of their core objects is solved by *translation polymorphism*, an implicit type-safe conversion between role instances and their core instances (Herrmann et al., 2004).

Roles can also be realized with aspect-oriented programming (Kiczales et al., 1997). Kendall analyzes how to implement role models with aspects and compares it with object-oriented approaches (Kendall, 1999). In (Hannemann and Kiczales, 2002), design patterns are implemented

with AspectJ, leading to similar benefits as our approach, namely reusability and better separation of concerns.

6 CONCLUSIONS

In this paper we propose a programming technique that enables expressing roles and collaborations. It is lightweight in the sense that no changes to the underlying language are necessary. Instead, dynamic proxies solve the problem of representing multiple objects as one compound object. Our approach allows to widen the set of members of an object at runtime, or in other words, to dynamically augment its type. Nevertheless, user code can be statically type-checked. We provide a Scala library as proof-of-concept and show how extracting concerns into collaborations supports reuse.

Future work will include applying our approach to a larger project in order to verify its usefulness and gain more insight about roles in programming languages. Another open task is a role-based library of reusable implementations of design patterns and other recurring object collaborations. Moreover, other role features should be investigated further, such as roles restricting access to its core object and restrictions on the sequence in which roles may be acquired and relinquished.

ACKNOWLEDGEMENTS

We would like to thank Prof. Uwe Aßmann for inspiring this work and the anonymous reviewers for their comments and suggestions.

REFERENCES

Bäumer, D., Riehle, D., Siberski, W., and Wulf, M. (1997). The Role Object pattern. In *Proceedings of the Conference on Pattern Languages of Programs (PLoP '97)*.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Guarino, N. (1992). Concepts, attributes and arbitrary relations: some linguistic and ontological criteria for structuring knowledge bases. *Data Knowl. Eng.*, 8(3):249–261.

Hannemann, J. and Kiczales, G. (2002). Design pattern implementation in Java and AspectJ. *SIGPLAN Not.*, 37(11):161–173.

Harrison, W. (1997). Homepage on subject-oriented programming and design patterns. http://www.research.ibm.com/sop/sopcpats.htm.

Herrmann, S. (2007). A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207.

Herrmann, S., Hundt, C., and Mehner, K. (2004). Translation polymorphism in Object Teams. Technical report, TU Berlin.

Kendall, E. A. (1999). Role model designs and implementations with aspect-oriented programming. *SIGPLAN Not.*, 34(10):353–369.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 1997)*, pages 220–242.

Kristensen, B. B. and Osterbye, K. (1996). Roles: conceptual abstraction theory and practical language issues. *Theor. Pract. Object Syst.*, 2(3):143–160.

Object Management Group OMG (2007). OMG Unified Modeling Language (OMG UML), superstructure, v2.1.2.

Odersky, M. (2008). *Scala Language Specification*. Version 2.7, http://www.scala-lang.org/docu/files/ScalaReference.pdf.

Odersky, M., Spoon, L., and Venners, B. (2008). *Programming in Scala, A comprehensive step-by-step guide*. Artima.

Reenskaug, T., Wold, P., and Lehne, O. A. (1996). *Working with Objects, The OOram Software Engineering Method*. Manning Publications Co.

Riehle, D. (1997). Composite design patterns. In *OOPSLA '97: Proceedings of the 12th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 218–228, New York, NY, USA. ACM Press.

Riehle, D. (2000). *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zürich.

Smaragdakis, Y. and Batory, D. (2002). Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255.

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