Explicit Relationships with Roles - A Library Approach

Michael Pradel *
EPFL, Switzerland
michael@binaervarianz.de

Abstract

In today's object-oriented programming languages the relations between objects are often implicit and scattered over multiple classes. This creates a gap between design, where relationships are more explicit, and implementation. Programs become not only harder to understand but also more difficult to modify. We present a programming technique that allows programmers to make relationships explicit by assigning roles to participating objects and dynamically mixing new fields and methods into existing objects with delegation semantics. Our approach builds upon commonly available language constructs and is implemented as a Scala library, such that it can be easily adopted in practical settings.

Categories and Subject Descriptors D [*3*]: 3—Classes and objects

Keywords Object-orientation, programming languages, roles, relationships, Scala

1. Introduction

In object-oriented programming, objects encapsulate state and behavior. Classes abstract over sets of similar objects and make such similarities explicit. However, objects do not only exist on their own but are related to each other in manifold and complex ways. Unfortunately, these relations are implicit in most programming languages and hidden behind references, method calls, etc.

A promising approach to reducing the complexity of a network of interrelated objects is to make their relationships explicit. Two ways to achieve this goal have been proposed: On the one hand, one can use commonly available programming language constructs and encode relationships with classes and objects [NG95]. This approach is limited, though, since the syntactic and semantic expressiveness is bounded by the underlying language. On the other hand, there are approaches towards new programming languages with relationships as a first-class construct [Rum87, BW05, BGE07]. While providing extensive support for relationships, they are hard to adopt in production environments and have not found their way into mainstream languages yet.

We aim at striking a balance between both approaches by proposing a library for the Scala programming language [OSV08], which allows for explicitly describing relationships. While still being compatible with the underlying language, its expressive power goes beyond simple class-based encodings. Scala provides a set of powerful basic language constructs that enables highly-expressive extensions in form of libraries, for example, [EMO06, HO06].

Instead of using first-class language constructs for relationships and roles, we have to express them with existing constructs. Our approach represents relationships by classes that are instantiated at runtime. Objects that participate in a relationship each play a *role*. Roles are specified as inner traits of a relationship class and provide fields and methods (members).[1] When an object participates in the relationship, the members of the role it plays are dynamically mixed into the members of the object using delegation semantics. Thus, being part of a relationship enhances existing objects with new functionality that is only relevant for this relationship. Internally, an object and its role are two separate objects, which are combined by a dynamic proxy.

The following code snippet gives a first idea of how our solution looks like:

```
val mary = new Person{}
val comp = new Company{}

val e = new Employment(mary, comp)
e.salary = 1200

e.employee.work()  // type Person with Employee
e.employer.pay()   // type Company with Employer
```

We instantiate a relationship `Employment` and pass a person `mary` and a company `comp` as its participants. The relationship instance `e` has a field to store the salary. We can

* Work performed at LAMP/EPFL. Presentation support provided, in part, by ETH Zurich and by the author.

[1] In Scala, a trait is similar to a class, but provides a safe form of multiple inheritance (mixin composition) [Ode08].

access the participants of the relationship as fields of the relationship instance, where the field names `employee` and `employer` designate their roles. Accessing members of a participant can be statically type checked, since its type is that of the participating object and that of the role. For example, `e.employee` has the type `Person with Employee`, where `Employee` is an inner trait of `Employment`. We elaborate on the example in the remainder of the paper.

Our approach provides the typical benefits of explicit relationships without requiring a special-purpose language. The overall structure of software becomes easier to understand because entities and the relationships between them are described separately. Moreover, we obtain high code locality, because a relationship encapsulates conceptually related code fragments instead of scattering them over multiple classes. This facilitates modifications of existing software, as one can adapt one concern of an application by changing the program at only one location.

We show in this paper how far a library approach can go in making relationships explicit. Our main contributions are the following:

- A programming technique for explicitly describing relationships and their roles, which uses only commonly available programming language features (Section 3 and Section 4).
- Dynamic mixin of relationship level members into participants with delegation semantics similar to object-based languages (Section 3).
- Implementation as a Scala library with different variations of our approach.

Preliminary results of this work have been presented in [PO08].

2. Roles, Relationships, and Collaborations

This section gives the necessary background by shortly introducing roles, relationships, and collaborations. Moreover, we introduce a running example for this paper.

Our running example is about people and their professional life. Suppose we want to describe a couple of general features of persons, such as name, date of birth, etc. Furthermore, we want to express how people relate to each other as employees and employers, where employees work and keep track of their working hours, for which employers pay them. In real life, persons may be employed at different employers during their lifetime. Hence, we want to be able to modify these relations dynamically. Moreover, our software may cover other, unrelated aspects of persons, such as their families. Thus, we want to keep the employment concern separate from the rest.

One solution for such a problem is the use of *roles*. The notion of roles that we use in this paper stems from conceptual modeling. It has been brought to object-oriented software by Reenskaug [RWL96]. A role describes the behavior of an object in a certain context. Roles are bound to objects at runtime, in which case we say that an object *plays* a role. A comprehensive summary of the various definitions of the term role is given in [Ste00], which presents 15 features of roles. The following summarizes the most essential features:

1. A role comes with its own properties and behavior.
2. 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.
3. 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).
4. 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.
5. Roles depend on relationships, that is, a role is only meaningful in the context of a relationship and cannot exist without.

There are different notions of relationships with respect to object-oriented programming languages. For example, relationships are directly mapped to mathematical relations in [BGE07]. In this paper, we use relationships as a more general term, referring to a specific context in which a number of objects interact with each other. We say that two objects interact if, for example, one has a reference to the other, or if one is accessing a member of the other.

We can make this general kind of relationships explicit with *collaborations*. A collaboration is a set of related roles that covers one concern of an application. Returning to our example, we can use two roles `Employee` and `Employer`. Both roles together form an `Employment` collaboration, which itself may also contain members, for example, to store the salary.

3. Dynamic Mixin of Role Members

When objects participate in a relationship, their state and behavior may be changed or extended for this specific context. We allow such adaptations by assigning a role to each participant. A role is defined as a set of fields and methods which are mixed into those of the participating object. For example, the following defines the employee role:

```
trait Employee extends Role[Person] {
  var hoursWorked = 0
  var money = 0
  def work = hoursWorked += 8
}
```

A trait becomes a role by extending `Role` from our library. `Role` takes a type parameter to specify the type of objects that may play the role (`Person` in the example). To allow for binding a role to arbitrary objects, `AnyRef`, Scala's equivalent to Java's `Object`, can be passed. By restricting

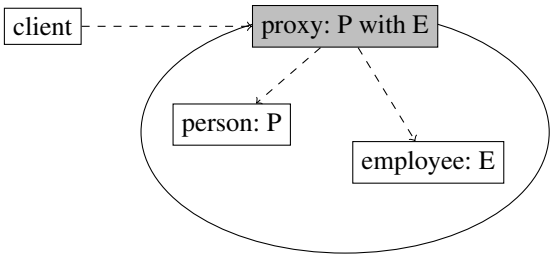

Figure 1. An object (*person*) and its role (*employee*) are represented by a dynamic proxy (dashed arrows depict references). The proxy has the compound type of both objects (*P with E*).

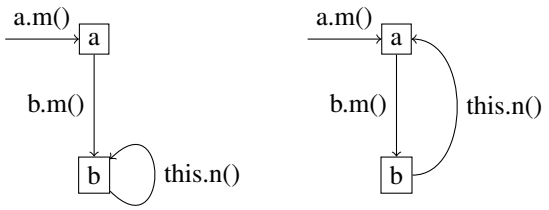

Figure 2. Forwarding (left) versus delegation (right). In case of delegation, `this` refers to the originally called object (arrows depict method calls).

the type of objects that are allowed to play a role, programmers of the role can type-safely refer to fields and methods of the participating object via a member `core` inherited from `Role`. For instance, we could refer to a person's name in the above definition of `Employee` with `core.name`.

Adding members to objects at runtime is a known feature in dynamically typed languages, such as Ruby or Python. Our approach allows for adding members dynamically and, at the same time, statically type checking access to them. That is realized by representing roles by objects and hiding the role-playing object and its role behind a dynamic proxy (Figure 1). The type of a dynamic proxy can be specified dynamically upon its creation. For example, to attach a role of type `Employee` to an object of type `Person`, we would create a proxy with the compound type of `Employee` and `Person`, that is, a type containing all the members of `Employee` and `Person`. Dynamic proxies are, for instance, provided by the Java API and can be used in Scala thanks to its Java compatibility.

One restriction of our library is that roles can only be bound to instances of traits (and not classes). Hence, `Person` must be a trait in the above example. The reason is that one requires interfaces for setting the type of a dynamic proxy. Scala's traits are always accompanied by an interface, whereas generating interfaces for arbitrary classes is difficult. As traits provide most of the features of classes in Scala, though, this restriction is not an issue in most applications.

3.1 Delegation Semantics for Roles

Adding members to an object is not enough, though. A dynamic proxy treats method calls reflectively, for example, by calling corresponding methods of one of the objects that the proxy hides. The semantics of these method invocations are worth a closer look. Imagine a situation where a role should override behavior of the role-playing object. For instance, persons may have a method `pickUpPhone` that calls a method `greet`, where `greet` is modified by the employee role. If the proxy simply forwards a call to `pickUpPhone` to the person, the overridden `greet` will never be called, because the person object calls `this.greet`.

The above issue has been described as the *self problem* [Lie86]. The crucial question is whether we use *forwarding* or *delegation* when the proxy invokes a method of one of the objects that it hides (Figure 2). Forwarding is the default behavior in class-based languages, where the control flow is passed to the callee until the method returns. In contrast, object-based languages, such as Self [US87], provide delegation. That is, they change the value of `this` in the callee, such that `this` always refers to the originally called object, hence, in our case the proxy.

To obtain delegation behavior, we make use of the way Scala translates traits into Java bytecode.[2] Consider the following, very simple trait containing one method:

```
trait T {
  def meth = 23
}
```

The Scala compiler will translate it into two parts: First, we get a Java interface, which represents the type of the trait and includes abstract versions of its members.

```
public interface T {
  public abstract int meth();
}
```

Second, the compiler creates an abstract class with implementations of the trait's methods. For each method, there is a static method which is called whenever a method of the trait is invoked in a Scala program. Since static methods are not bound to a particular instance, we have to specify the value of `this` to be used inside the method body. Therefore, each of those static methods takes an additional parameter `$this`.

```
public abstract class T$class {
  public static int meth(T $this) {
    return 23;
  }
}
```

[2] For ease of understanding, we give the corresponding Java source code here.

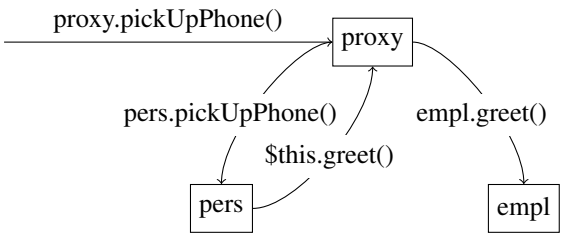

Figure 3. Method dispatch inside role-playing objects has delegation semantics; `$this` always refers to the proxy and is used instead of `this`.

The translation scheme for traits allows for simulating delegation. The dynamic proxy is accompanied by an *invocation handler*, which reflectively deals with method calls to the proxy. Our library provides a default invocation handler with the following policy:

1. If the method to call is implemented by the role, invoke the corresponding static method and pass the proxy as value for `$this`.
2. Otherwise, invoke the static method that corresponds to the role-playing object, also passing the proxy as value for `$this`.

Since we set the type of the proxy to the compound type of the role and the role-playing object, we can guarantee that each method that is called is actually implemented in one of them. Consequently, the invocation handler is always able to delegate method calls.

As a result of the above policy, a role can override behavior of the participating object. Reconsider the above example of the employee role that overrides the `greet` method of persons (Figure 3). A call to `pickUpPhone` will be dispatched by the proxy to the static method containing `pickUpPhone`'s implementation, passing the proxy as value for the parameter `$this`. The call to `greet` is, thus, made again on the proxy, which can dispatch it to the employee role .

4. Making Relationships Explicit with Collaborations

This section shows how we can use the role encoding presented in Section 3 to implement relationships. We present *collaborations*, that is, sets of related roles that encapsulate relationships and the relationship-specific behavior of participating objects. We illustrate different variations of the concept with examples and discuss their properties.

Generally, we represent collaborations by using classes whose inner traits are considered to be roles. Roles cannot occur without a collaboration in our approach, according to Steimann's feature *roles depend on relationships* (see Section 2). Listing 1 shows the basic structure of a collaboration that describes the relation between employees and employers. These are described with two role types `Employee` and `Employer`. In line 2 (and 3), we bind an instance of the `Employee` (and the `Employer`) role to a person (and a company) passed as a constructor argument to the collaboration. We therefore use the `as` operator from our library. Calling `o as r`, where `r` is a reference to a role, yields the object `o` playing the role `r`. Internally, a dynamic proxy is created that hides both the object and the role instance. More details on the implementation of the `as` operator are given in [PO08].

4.1 Fixed Roles

A collaboration encapsulates a set of related roles. Our experiments have shown that different variations of this general concept are useful. One criterion is the binding between roles and objects. While some roles are transient and may change their underlying objects regularly, others are of a more permanent nature and belong to one participant during the entire lifetime of the role. We reflect this distinction between *transient roles* and *fixed roles* in our library.

Listing 1 is an example of a collaboration with two fixed roles. The participants of a fixed collaboration are passed as constructor arguments when the collaboration is instantiated:

```
val e = new Employment(mary, comp)
```

Once a collaboration instance is created, the participants cannot change as long as the collaboration is used. We can access a participating object in its role using the role name qualified by the collaboration instance:

```
e.employee.work()
e.employer.pay()
```

4.2 Transient Roles

While fixed roles are useful if the relationship has a stable set of participants, we may want a more flexible mechanism in some cases. Consider, for example, a company with a high fluctuation of employees, where a new person arriving should simply take over where another stopped working.

This scenario can be expressed with *transient roles*, which are only temporarily bound to objects. Instead of using the `as` operator once during the initialization of a collaboration instance, users of the relationship call it explicitly for transient roles. In this manner, a role can be seamlessly transferred from one object to another, retaining the state of the role. Since the participants are not fixed, they are not given in the constructor:

```
val e = new Employment()
```

Instead, roles are explicitly bound to objects using the `as` operator. This time references to a role must be qualified with a collaboration instance:

```
(mary as e.employee).work
(paul as e.employee).work
```

The implementation of the corresponding collaboration differs only slightly from Listing 1. All we have to do is

```
1  class Employment(p: Person, c: Company) extends Collaboration {
2    val employee = p as new Employee{}
3    val employer = c as new Employer{}
4
5    trait Employee extends Role[Person] {
6      var hoursWorked = 0
7      var money = 0
8      def work = hoursWorked += 8
9    }
10
11   trait Employer extends Role[Company] {
12     def pay = {
13       employee.money += employee.hoursWorked * 10
14       employee.hoursWorked = 0
15     }
16   }
17 }
```

Listing 1. A collaboration with two fixed roles `employee` and `employer`.

removing the constructor parameters and changing lines 2 and 3 into:

```
val employee = new Employee{}
val employer = new Employer{}
```

Note that using transient roles for the roles as defined in Listing 1 may cause unexpected behavior. Since the state of a role is preserved when the role is transferred to another object, the next person playing the employee role continues with the value of `hoursWorked` increased by `mary` and `paul`. However, transient roles are, for instance, useful if the employee role contains state such as the intermediate result of a project. In this case, a new employee can continue using the results of his predecessor.

4.3 Arbitrary Many Role Instances

In the above examples, the number of role instances per collaboration instance is fixed. However, some relationships may contain arbitrarily many participants. For instance, let us consider the case where multiple employees should be captured in one relationship. If the number of employees is not statically known, we can use *role mappers*. From the user's point of view, role mappers provide exactly the same syntax as transient roles. When binding a role to an object with `as`, though, new role instances are created dynamically whenever an object participates that has not played that role yet.

Using an `Employment` collaboration with a role mapper for the employee role, the following creates two role instances:

```
(mary as e.employee).work
(paul as e.employee).work
(mary as e.employee).work
```

In the third line, the role instance from the first line, including its state, is reused and again attached to `mary`.

To use a role mapper the collaboration developer must replace the role instantiation by a role mapper object. Modifying Listing 1 such that the employee role becomes a role mapper results in Listing 2. We create a role mapper as an `object` that extends `RoleMapper` from our library (line 2).[3] It takes the type of the participating object (`Person`) as well as the type of the role (`Employee`) as type parameters. The employer role is now implemented as a transient role (line 5). Consequently, both roles defined by the collaboration can be used with the same syntax, namely the `as` operator. In the `Employer` role, we have to change the `pay` method, such that it gets the employee that should be paid as an argument (line 12). This is necessary since multiple employees may participate in the collaboration.

4.4 Querying

Role mappers allow for arbitrarily many participants playing a certain role in one collaboration instance. We support querying the set of participants with Boolean functions. For example, one may be interested in employees that have worked for more then a certain amount of hours. This can be queried with:

```
e.employee.query(x => x.hoursWorked > 100)
```

The result will be an iterator over the set of all objects that play the employee role and that fulfill the predicate `hoursWorked > 100`.

4.5 Multiplicities

In modeling, one can usually specify multiplicities for relationships. We allow for setting an upper bound for the number of participants bound by a role mapper. The role mapper specification may, for example, include:

[3] The `object` keyword in Scala makes the Singleton pattern [GHJV95] part of the language. It creates a class and its one and only instance.

```
class Employment extends Collaboration {
  object employee extends RoleMapper[Person, Employee] {
    override def createRole = new Employee{}
  }
  val employer = new Employer{}

  trait Employee extends Role[Person] {
    // same as in Listing 1
  }

  trait Employer extends Role[Company] {
    def pay(e: Employee) = {
      e.money += e.hoursWorked * 10
      e.hoursWorked = 0
    }
  }
}
```

Listing 2. The employment collaboration with a role mapper `employee`. It allows for arbitrarily many instances of the employee role.

```
override def max = 5
```

As a result, calling `as` with more than five different objects produces a runtime exception. A convenient way of specifying multiplicities that can be statically checked within our approach is an open issue.

5. Discussion

We use a lightweight approach by providing explicit roles and relationships as a library instead of changing the underlying language. Surprisingly, we can nevertheless provide many benefits of languages with first-class relationships or first-class roles. Having a low cost of adoption, we believe that our proposal is a good candidate to gain practical experience with relationships and roles in programming.

One issue that is not solved so far is how to remove objects from a relationship. We do not provide an explicit removal operator, because it could lead to inconsistencies due to aliasing. Consider the following example, where the employee role is implemented with a role mapper:

```
val x = paul as e.employee
val y = bill as e.employee

e.employee.query(true)  // 2 employees
// remove paul from the collaboration
e.employee.query(true)  // 1 employee

x.work // inconsistency
```

By storing `paul as e.employee` into `x`, we can still access `paul` in the employee role, even after explicitly removing him from the collaboration. This would contradict the result of querying the collaboration, where `bill` is not present anymore. Unfortunately, we cannot detect such situations statically (throwing a runtime exception would be possible, though). To protect users from such inconsistencies, we do not provide a removal mechanism for role mappers. As a result, objects added to a relationship belong to it until the collaboration instance is removed by garbage collection. Finding a safe way of removing relationships remains as future work.

Rumbaugh argues for first-class relationships because they "abstract [..] the high-level static structure of the system" [Rum87]. In our approach, the information about which classes can be related via which collaborations is specified by the type parameter of `Role` that constrains the objects playing that role into a certain type. As a result, it is trivial to detect statically that persons may be related via an employment relationship.

Our approach builds upon commonly available constructs of object-oriented languages. In other languages that provide inner classes and dynamic proxies, such as Java, similar implementations are possible. Our implementation makes use of a number of particularities of Scala. First, the `as` operator is implemented using infix notation for method calls and implicit conversions, language features not provided by Java.[4] Second, we benefit from Scala's translation scheme for traits to realize delegation (as explained in Section 3). Nevertheless, we believe many of the ideas of our work to be transferable to other languages with minor adaptations.

In Section 2, we mentioned essential features of roles given by [Ste00]. Our approach provides these features as follows:

1. A role comes with its own properties and behavior, since roles are defined as traits and provide fields and methods (Section 3).

[4] An implicit conversion is a special function inserted by the compiler whenever an expression would not be otherwise type-correct.

2. Roles are dynamic and can be acquired and abandoned at runtime using transient roles and the `as` operator. A role `r` can be transferred from one object to another by using `.. as r` with different objects as shown in Section 4.2.
3. An object can play more than one role at the same time by using `as` multiple times. For example, an employee can take the role of a workaholic with `(paul as employee) as workaholic`. For playing two roles of the same role type in different contexts one must use two different collaboration instances. For example, being employed with two different companies can be expressed as follows:

```
(paul as e1.employee).work
(paul as e2.employee).work
```

   By adding both roles together with

```
(paul as e1.employee) as e2.employee
```

   the members of the second role override those of the first.
4. The state and the members of an object may be role-specific and a role can override members of an object as discussed in Section 3.
5. Roles depend on relationships, since roles are defined inside collaborations. At runtime, roles have to be qualified with a collaboration instance.

6. Related Work

This work draws on other approaches to making roles and relationships more explicit in object-oriented programming languages. Rumbaugh was the first to propose relationships (relations) as a first-class construct, aiming at externalizing otherwise scattered references into a symmetric construct [Rum87].

In [BW05], a prototypical programming language with first-class relationships, *RelJ*, is formalized. The authors stress a notion of relationship inheritance different from class inheritance, where at most one relationship instance between two objects may exist, while sub-relationships delegate to super-relationships. Balzer et al. build upon [BW05] and develop a relational model based on mathematical relations to describe object collaborations in a declarative way [BGE07] . Members can be added to objects that participate in a relationship (*membership interposition*). Furthermore, the authors investigate sophisticated relationship invariants.

Implementing relationships with existing language constructs is also proposed in [NG95]. The authors propose the use of relationship objects that either keep references to participating objects and forward calls to them, or are dependents of these objects and get updated via the Observer pattern [GHJV95]. Østerbye presents a library for expressing association relationships in C# [Øs07], which makes heavy use of runtime reflection on type parameters. It defines roles as fields in the core classes that relate to it. The library provides both a relationship-centric view and a role-centric view, and facilitates to switch from one to the other.

An aspect library, expressing relationships as separable, cross-cutting concerns, is presented in [PN06]. The authors distinguish static relationships, implemented with inter-type declarations, and dynamic relationships. Similar to our approach, the latter allows for dynamically adding objects to relationships when needed. However, the paper considers only structural aspects of relationships; changing the behavior of participants in a dynamic relationship is not supported. Our work also relates to the aspect-oriented paradigm in general, since a role can intercept the methods of an object and change its behavior without modifying the object's source code.

In the remainder of this section, we present related work that focuses on roles rather than relationships. The Role Object pattern [BRSW00] describes how to split one conceptual object into a core object and multiple role objects, such that roles can be added and removed dynamically. The pattern does not discuss grouping of roles into collaborations. We develop the ideas of the pattern by allowing for role member mixin with delegation semantics and statically type-checked access of them.

ObjectTeams/Java is an extension of Java which provides roles and *teams* as first-class language constructs [Her07]. A team is a set of related roles and corresponds roughly to our collaborations. The self problem is solved with particular semantics for roles that override methods, where potentially overridden calls are intercepted and redirected to the role. A similar approach is Epsilon [TUI07] and its Java-based implementation EpsilonJ. The authors propose *collaboration fields* that contain roles. Similar to our approach and ObjectTeams/Java, roles are defined as nested types. In the Epsilon model, roles exist as first-class citizens at runtime and are bound to objects with a `bind` primitive. Our work differs from ObjectTeams/Java and EpsilonJ by building upon an existing language instead of designing a special-purpose language.

Another extension of Java to support roles is powerJava [BBvdT07a]. In powerJava, the execution of methods depends on the view of the caller, which assigns a role to the callee. Access to different roles is accomplished with *role castings*; they roughly resemble the `as` operator. In contrast to our work, powerJava's method dispatch for role-playing objects is pure forwarding. The authors also discuss the implementation of relationships with roles [BBvdT07b]. It is proposed to implement binary relationships by defining the role of one participant as inner class of the other (and vice versa) and referring to it with attributes. This extends the *relationship as attribute* pattern [Nob99] by allowing to add state and behavior to objects entering a relationship. One drawback is the strong coupling between classes whose instances may participate in a relationship.

7. Conclusions

We propose an implementation technique to express relationships and roles, and implement it as a Scala library. It allows for adding behavior to objects that participate in a relationship. Mixing new members into existing objects has delegation semantics, a feature usually known from object-based languages.

Although our approach is implemented as a library using existing programming language constructs, it provides a number of benefits from languages with first-class relationships. Making the relationships between objects explicit enhances the structure of software and makes it easier to understand and maintain. The core classes of an application become smaller, since interactions with other classes are extracted into collaborations. Hence, the core objects become less complex and more likely to reuse. Moreover, our approach allows for transferring notions from modeling, such as association classes, to the level of implementation.

As future work we want to apply our library to implement a larger application. Since our approach embeds seamlessly into an existing language, it seems to be a good candidate for such experiments. An open issue to investigate further is how objects can be safely removed from relationships at runtime. Finally, inheritance of relationships and its semantics given in [BW05] should be studied in the context of our approach.

Acknowledgments

Thanks to Antonio Cunei, Martin Odersky, Geoffrey Washburn, and the anonymous reviewers for their comments and suggestions.

References

[BBvdT07a] Matteo Baldoni, Guido Boella, and Leendert van der Torre. Interaction between objects in powerJava. *Journal of Object Technology*, 6(2), 2007.

[BBvdT07b] Matteo Baldoni, Guido Boella, and Leendert van der Torre. Relationships meet their roles in object oriented programming. In *International Symposium on Fundamentals of Software Engineering (FSEN)*, pages 440–448. Springer, 2007.

[BGE07] Stephanie Balzer, Thomas R. Gross, and Patrick Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *European Conference on Object-Oriented Programming (ECOOP '07)*, pages 323–346. Springer, 2007.

[BRSW00] Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. Role Object. In *Pattern Languages of Program Design 4*, pages 15–32. Addison-Wesley, 2000.

[BW05] Gavin M. Bierman and Alisdair Wren. First-class relationships in an object-oriented language. In *European Conference on Object-Oriented Programming (ECOOP '05)*, pages 262–286. Springer, 2005.

[EMO06] Burak Emir, Sebastian Maneth, and Martin Odersky. Scalable programming abstractions for XML services. In *Dependable Systems: Software, Computing, Networks, Research Results of the DICS Program*, pages 103–126. Springer, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Her07] Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Applied Ontology*, 2(2):181–207, 2007.

[HO06] Philipp Haller and Martin Odersky. Event-based programming without inversion of control. In *Joint Modular Languages Conference*. Springer, 2006.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '86)*, pages 214–223. ACM, 1986.

[NG95] James Noble and John Grundy. Explicit relationships in object-oriented development. In *Technology of Object-Oriented Languages and Systems Conference (TOOLS 18)*, 1995.

[Nob99] James Noble. Basic relationship patterns. In *Pattern Languages of Program Design 4*. Addison-Wesley, 1999.

[Ode08] Martin Odersky. *Scala Language Specification*, May 2008. Version 2.7.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, A comprehensive step-by-step guide*. Artima, 2008.

[PN06] David J. Pearce and James Noble. Relationship aspects. In *International Conference on Aspect-Oriented Software Development (AOSD '06)*, pages 75–86. ACM, 2006.

[PO08] Michael Pradel and Martin Odersky. Scala Roles - A lightweight approach towards reusable collaborations. In *International Conference on Software and Data Technologies (ICSOFT '08)*, 2008.

[Rum87] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '87)*, pages 466–481, 1987.

[RWL96] Trygve Reenskaug, P. Wold, and O. A. Lehne. *Working with Objects, The OOram Software Engineering Method*. Manning, 1996.

[Ste00] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, 2000.

[TUI07] Tetsuo Tamai, Naoyasu Ubayashi, and Ryoichi Ichiyamah. Objects as actors assuming roles in the environment. In *Software Engineering for Multi-Agent Systems*, pages 185–203. Springer, 2007.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *SIGPLAN Notices*, 22(12):227–242, 1987.

[Øs07] Kasper Østerbye. Design of a class library for association relationships. In *Symposium on Library-Centric Software Design (LCSD '07)*, 2007.