Diploma Thesis

Roles and Collaborations in Scala

submitted by

Michael Pradel

born 10.03.1983 in Jena

Technische Universität Dresden

Fakultät Informatik
Institut für Software- und Multimediatechnik
Lehrstuhl Softwaretechnologie

Supervisors:
Prof. Dr. rer. nat. habil. Uwe Aßmann
Prof. Dr. sc. tech. Martin Odersky
MSc. Jakob Henriksson

Submitted June 13, 2008

Abstract

The interrelations of a set of software objects are usually manifold and complex. Common object-oriented programming languages provide constructs for structuring objects according to shared properties and behavior, but fail to provide abstraction mechanisms for the interactions of objects. Roles seem to be a promising approach to solve this problem as they focus on the behavior of an object in a certain context. Combining multiple roles yields collaborations, an interesting abstraction and reuse unit. However, existing approaches towards roles in programming languages require vast extensions of the underlying language or even propose new languages.

We propose a programming technique that enables role-based programming with commonly available language constructs. Thus, programmers can express roles and collaborations by simply using a library, and hence, without the need to change the language, its compiler, and its tools. We explain our proposal on a language-independent level. Moreover, we provide an implementation in form of a library for the Scala programming language. Finally, we apply our ideas to design patterns and analyze to which extent these can be expressed and reused with roles.

Zusammenfassung

Die Zusammenhänge zwischen Softwareobjekten sind vielfältig und komplex. In den meisten objektorientierten Programmiersprachen werden Objekte an Hand von gemeinsamen Eigenschaften und Verhalten klassifiziert. Konstrukte zum Strukturieren bezüglich ihrer Interaktionen fehlen jedoch. Ein vielversprechender Lösungsansatz sind Rollen, welche das Verhalten von Objekten in einem bestimmten Kontext beschreiben. Zusammenhängende Rollen können zu Kollaborationen abstrahiert werden. Diese sind insbesondere als wiederverwendbare Bausteine interessant. Allerdings verändern bisherige Ansätze zu rollenbasiertem Programmieren die zu Grunde liegende Sprache erheblich oder schlagen gar neue Sprachen vor.
Im Gegensatz dazu zeigen wir eine Programmiermethode, die rollenbasiertes Programmieren mit üblichen Sprachkonstrukten ermöglicht. Somit können Rollen und Kollaborationen als Bibliothek bereitgestellt werden, also ohne Sprache, Compiler und Werkzeuge anpassen zu müssen. Wir erläutern unseren Ansatz zunächst sprachunabhängig. Desweiteren wird eine Implementierung als Bibliothek für die Scala Programmiersprache präsentiert. Als praktische Anwendung stellen wir Entwurfsmustern dar und überprüfen, inwiefern sich diese mit Rollen ausdrücken und wiederverwenden lassen.

Acknowledgments

First of all, I want to thank Prof. Martin Odersky and Prof. Uwe Aßmann for giving me the unique and wonderful opportunity to work as a visiting student in the Programming Methods Laboratory at EPFL in Lausanne. Without their constant support and valuable advice this thesis would not have been possible. Moreover, I want to thank Jakob Henriksson, Ingo Maier, and Marco Zimmerling for giving me feedback on drafts of this work that has been of great benefit to me. Finally, I would like to thank my colleagues at LAMP, Iulian Dragos, Gilles Dubochet, Sebastian Hack, Philipp Haller, Lukas Rytz, and Geoffrey Washburn, for fruitful discussions on roles, research, and all the rest.

Contents

1 Introduction

In object-oriented software, real world entities and abstract ideas are represented by objects. Usually, there are a lot of them and representing their interrelations as a graph yields a complex network. One way to organize objects are classes, abstraction units that group objects according to common properties and behavior.

However, objects and classes fail to solve two important issues:

- Real world objects can fundamentally change their behavior depending on the situation in which they occur. In contrast, objects usually belong to the same class throughout their entire lifetime.
- Classes are inappropriate to model one specific concern of an object interaction, possibly involving multiple objects.

Let us illustrate these problems with an object representing a person. The person may be seen as a professor in some context (at work), and as a father in another context (at home). Professors and fathers, obviously, contain different state and provide different behavior. A common class-based approach to describe this person would be a large class including all state and behavior necessary to describe the different facets. However, this makes our person more complicated than needed. Instead, we would like to adapt the person on demand to the current environment, such that its state and behavior change whenever the person migrates into another context.

Furthermore, let us consider the objects our person is related to in both contexts. As a professor, it may interact with a university object and other person objects that are students; as a father, it may be related to a person that is a child. In order to structure our network of objects, we need an abstraction unit for the relations of multiple objects focusing on a single concern only.

The software modeling community has found a solution for the above problems: *role modeling* or *collaboration-based design*. The basic idea is that objects can play different *roles* during their lifetime. Each role provides a specific view on an object. For the above example, an object describing a person can play the roles *professor* and *father*. As one role is independent of the others, programmers can, depending on the context, concentrate on one role and ignore the rest. Figure 1.1 depicts the idea of role modeling.

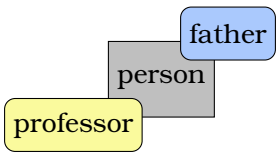


Figure 1.1: A person object playing two roles *professor* and *father*.

Usually, a role is defined through its relation to one or more other roles. For instance, a father is a father because he has a child. Related roles are

grouped into *collaborations*. Consequently, a collaboration describes the behavior of multiple related objects, and hence, can cover one specific concern of an application. Figure 1.2 extends Figure 1.1 with other roles. The roles *professor*, *student*, and *university* are related and form a collaboration. In contrast, the roles *father* and *child* belong to another concern, and thus, form another collaboration.

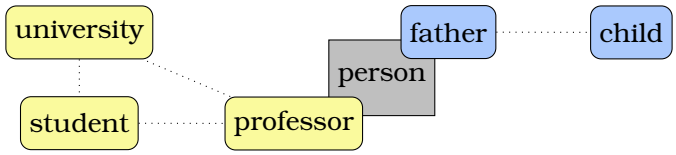


Figure 1.2: Extended version of Figure 1.1 showing related roles. As a result, the person is part of two independent collaborations that describe different concerns of the application.

The idea of roles has been extensively researched by the software modeling community. Introduced by Reenskaug [RWL96], role modeling has been applied to framework design [RG98, Rie00], design patterns [Rie96, Rie97a, Rie97b, KFGS03], framework documentation [Rie07], and hypermedia design [ADN+03]. A good overview on earlier work is given in [KRS98]. In [PHA07, Pra08], roles are applied to ontologies. Furthermore, the Unified Modeling Language (UML) [OMGO07] provides roles and collaborations as modeling constructs.

Although roles have found its way into modeling, they are not yet supported by major programming languages. There exist, however, a couple of interesting approaches, which either introduce roles as a new first-class language construct [KO96, BBvdT07, Her07] or express them with mechanisms available in the underlying programming language [VN96, BRSW00, SB02]. Our work builds upon them by providing a programming technique to express roles with constructs available in most object-oriented programming languages. These constructs offer at the same time a convenient syntax to the user. Hence, we take a lightweight approach in the sense that, instead of proposing a new programming language with role-specific constructs, one can realize our approach as a library.

Using roles and collaborations in programming has a number of benefits. First, programmers can express different *views* on an object, and therefore, adopt the notion of roles from modeling to source code. Second, collaborations can serve as a *programming abstraction*. This comes along with the following advantages:

- *Separation of concerns*. Collaborations can structure complex networks of objects. As a collaboration focuses on one specific aspect of a system, programmers can separate different concerns into independent collaborations.

- *Reuse*. Collaborations can capture reusable code fragments that involve multiple objects. By focusing on a single concern, reuse is more likely to happen than with a traditional set of classes, where different concerns of a system are often intermingled and not made explicit.

Contributions

The major contributions of this thesis are twofold. On the one hand, we contribute on a language-independent level by proposing:

- A programming technique to represent roles and collaborations with commonly available language constructs. The approach is applicable to other object-oriented programming languages as well.
- A novel reuse unit, dynamic collaboration, which captures the relations of objects in a context. In contrast to many other occurrences of the term *collaboration* in the literature, our approach allows for binding and removing roles to and from arbitrary objects at any point in the objects' lifetime.

On the other hand, we validate our approach in the Scala programming language with:

- A role library that allows to dynamically augment an object's type by adding state and behavior while preserving strong static type safety.
- A case study applying the role library to design patterns. We show to which extend pattern implementations can be captured with roles and present reusable collaborations for some of them.

Parts of this work are presented in [PO08].

Overview

This thesis is divided into seven chapters, the first one being this introduction and the last one a conclusion. The following provides a short overview of the remaining chapters:

Chapter 2: Background This chapter provides a short overview of basic concepts and ideas this thesis builds upon. Readers already familiar with the content, can safely skip it. At first, we give a short introduction to the Scala programming language, thereby focusing on features we use in this work. Furthermore, we introduce role modeling. Finally, we briefly describe dynamic proxies.

Chapter 3: Lightweight Roles and Collaborations We describe a programming technique to represent roles and collaborations with language constructs available in most object-oriented programming languages. Moreover, we explain important design choices and introduce the terminology used in the subsequent chapters. Since we present our approach in a language-independent way, this chapter serves well for implementing roles and collaborations in other languages than Scala.

Chapter 4: The Scala Roles Library This chapter presents an implementation of the approach described in Chapter 3. We explain interesting details of our implementation and illustrate how to use the library with examples. Moreover, we analyze its performance and discuss possible optimizations.

Chapter 5: Design Patterns as Collaborations We present the results of applying role-based programming to design patterns. We analyze 24 patterns and classify them into four groups: reusable pattern implementations, enhancements with roles, patterns that are obsolete in Scala, and patterns that are inappropriate for a role-based implementation in Scala.

Chapter 6: Related Work and Comparison This chapter briefly reviews existing approaches to role-based programming. We first evaluate Epsilon, ObjectTeams, aspect-oriented programming, mixin layers, and the Role Object pattern on its own, followed by a comparison with our approach.

Note on Scala Source Code

All Scala source code in this work has been compiled and tested with Scala 2.7.1.final.

2 Background

2.1 The Scala Programming Language

We use the Scala programming language to implement and test the ideas of this work. Our results are available as a Scala library. In this section, we provide a short overview of the language and explain specific features that we use, which may be new to readers not familiar with Scala. For more comprehensive literature, we refer to [OSV08, Ode08b, Ode08a]. A short introduction is available in [SH08].

Scala is a pure object-oriented programming language, that is, every value is an object. In particular, there are no primitive types as, for instance, in Java or C++. At the same time, Scala is a functional language that provides features such as higher order functions, pattern matching, function nesting, and currying.

As Scala is statically typed, type errors are found at compile time. Scala's type system supports generic classes, variance annotations, and upper and lower type bounds. Furthermore, the compiler provides a type inference mechanism, which unburdens programmers from giving redundant type information.

Scala is designed to be interoperable with common mainstream programming environments. It interoperates smoothly with the Java 2 Runtime Environment. Scala programs are compiled to Java bytecode and run on a Java Virtual Machine. This allows to use and extend existing Java libraries. There is also an alternative back-end for the .NET framework.

In the following, a selection of Scala features is explained that are vital to understand the concepts developed in this thesis.

Traits and mixin-based composition Objects can be abstracted with *classes* and *traits*, each of which describes state and behavior. Intuitively, a trait can be thought of as a Java interface, where some methods are provided with an implementation. While classes can have only one superclass, a class or a trait may extend multiple traits. The ambiguity problems associated with many implementations of multiple inheritance [SDNB03] are solved using a linearization order [Ode08b] and certain restrictions on traits. For example, traits may not have constructor parameters.

Type inference Scala has a local type inference mechanism allowing to omit certain type information. For instance, the following declaration contains redundant type information:

```
val s: String = "abc"
```

Since `"abc"` is clearly a String, one can simply write:

```
val s = "abc"
```

In the same manner, the return types of most functions and methods can be deduced from the type of their body. Also, missing type parameters can often be inferred.

Class hierarchy Scala is a pure object-oriented language, and therefore, has no primitive types. Instead, the class hierarchy is divided into *reference types* (`AnyRef`) and *value types* (`AnyVal`). Both inherit from Scala's root class `Any`.

Figure 2.1 depicts Scala's class hierarchy. There are exactly nine subclasses of `AnyVal`. The class `Unit` refers to a type that contains no usable value and corresponds to Java's `void`. Note that when Scala is used in a Java environment, `AnyRef` is equivalent to `Object`, the ancestor of all Java classes.

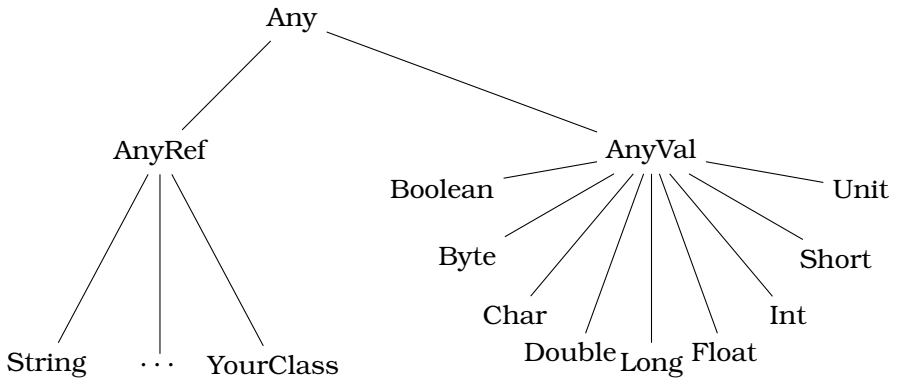


Figure 2.1: Overview of Scala's class hierarchy. User-defined classes are always subclasses of `AnyRef`.

Variables and values Scala distinguishes between *variables* (keyword `var`) and *values* (keyword `val`). While variables are mutable, values can only be assigned a value once on their initialization and are guaranteed not to change during their lifetime. For better code maintainability, using values is preferred over variables in general since their immutability prevents unwanted side effects.

Implicit conversions A very powerful language construct are *implicit conversions*. These are functions converting an object of one type into an object of another type. An implicit conversion is inserted by the compiler whenever a type error would otherwise occur. The following example wraps any object of type `A` into an object of type `B`:

```
class B(a: A) {
  def addedMethod = { /*..*/ }  // some method not present in A
}

implicit def AtoB(a: A) = new B(a)
```

As a result, one can call `addedMethod` on instances of `A` although it is implemented in `B`. This is possible because the compiler inserts a call to the conversion. Consequently, the `addedMethod` of `B` is actually called.

One application of implicit conversions is extending classes of a library with additional functionality without touching the library. Besides user-defined conversions, there are a couple of predefined conversions, for instance, enhancing Java's `String`s with additional functionality by converting it into a `RichString` when needed.

Infix operators Any method taking a single parameter can be used as *infix operator* in Scala. For instance, `x.add(y)` is equivalent to `x add y`. While infix operators are usually interpreted as left-associative method calls, a particular rule applies for method names ending with a colon ':'. These are treated as right-associative method calls. An example is the `::` operator (pronounced *cons*) that prepends a new element to an existing list. For instance,

```
"a" :: List("b", "c")
```

is interpreted as

```
List("b", "c").::("a")
```

Objects In some cases, exactly one instance of a class or trait is required in an application. This happens so often that the Singleton design pattern [GHJV95] was formulated. In Scala, this pattern is part of the language. Using the keyword `object`, we can define a type and its one and only instance. For example, one could write (assuming monotheism):

```
object God {
  def makeWorld = { /*..*/ }
}
```

Besides realizing the Singleton pattern as a language construct, objects are also a means for structuring an application. In fact, they can be used like packages, such that defining an object with inner classes and traits amounts to the same as defining the classes and traits in a package.

To avoid confusion, we use the term *Scala object* in this thesis when we mean a singleton object as described above.

Pattern matching and case classes Pattern matching is a conditional programming construct. It can be seen as a sophisticated version of *if-then-else* constructs or the switch statement from C and Java. In contrast to these, pattern matching can match on any kind of data, for instance, Integers or Strings. It is particularly useful in combination with *case classes*. These are classes which export their constructor parameters for matching. As a result, programmers can recursively decompose data structures built from case classes using pattern matching.

The following example defines case classes for expressions being either a number, or the sum, or the product of two expression. The function `eval` uses pattern matching to evaluate them. In the last line, we build the expression $3 * 2 + 4$ and print the result of its evaluation.

```
case class Expr
case class Sum(op1: Expr, op2: Expr) extends Expr
case class Prod(op1: Expr, op2: Expr) extends Expr
case class Num(n: Int) extends Expr
```

```
def eval(ex: Expr): Int = ex match {
  case Sum(op1, op2)   => eval(op1) + eval(op2)
  case Prod(op1, op2)  => eval(op1) * eval(op2)
  case Num(n)          => n
}

println(eval(Sum(Prod(Num(3), Num(2)), Num(4))))
```

Compound types *Compound types* denote that an object's type is a subtype of multiple other types. This can be specified by the keyword `with`. For instance, a function that requires an argument to have both the type of a trait `A` and of a trait `B` can be defined by:

```
def someFunction(arg: A with B)
```

Type parameters and type parameter bounds Similar to Java, parametric polymorphism is realized in Scala by providing classes and traits with *type parameters*. Moreover, functions and methods can be parametrized with types. One can limit a type parameter to be a subtype or supertype of some other type using upper and lower bounds.

In combination with type inference, a method may use the type of an argument as a type parameter without forcing the caller to explicitly specify it:

```
trait A
trait B extends A

def someFunction[T <: A](arg: T) = { /*..*/ }

someFunction(new B{})
```

The function `someFunction` takes any argument whose type is a subtype of `A` (specified with `<:`). The concrete type of the argument is bound to the type parameter `T`. Hence, in the last line, `T` is set to `B`.

Abstract member types *Abstract member types* are a concept very similar to type parameters. Whenever one wants to use a type in a trait or class without exactly knowing which type it will be in a concrete instance, it can be left abstract.

The following example defines a trait `A` with an abstract member type `T`, where `T` must be a subtype of `AnyRef`.

```
trait A {
  type T <: AnyRef
}
```

In order to instantiate `A`, we have to specify a concrete type for `T`, for instance:

```
val a = new A { type T = String }
```

Path-dependent types Nesting of traits and classes is a fundamental concept in Scala. In contrast to Java, inner types are not associated with their outer type but with an instance of the outer type. This principle is called *path-dependent types* as the inner type depends on a path. A path means a concrete reference to an object. The theoretical foundations of path-dependent types are given in [OCRZ03].

As an example why this is necessary, consider two nested traits:

```
trait A {
  type T
  trait B {
    def m(x: T) = { /*..*/ }
  }
}
```

The outer trait `A` contains an abstract member type `T`. Since the argument type of `B`'s method `m` depends on `T`, different instances of `A` can yield different versions of `B`. Consequently, `B` must be qualified with an instance of `A`. Assume we have two of them:

```
val a1 = new A{ type T = Int }
val a2 = new A{ type T = String }
```

This yields two different types `a1.B` and `a2.B` that both conform to the more general type `A#B`. The type `a1.B` has a method `m` taking an `Int` as argument. In contrast, the method `m` of the type `a2.B` takes a `String`.

equals, ==, and eq In object-oriented programming, one usually distinguishes between *object equality* and *structural equality* of objects. While the first refers to object identity, the second may implement a more meaningful semantics, for instance, comparing the structure of two objects.

In Scala, each reference object (that is, an instance of a subclass of `AnyRef`) provides a method `eq` for comparing object identities. In addition, there is a method `equals`, that may be overridden to define a meaningful notion of structural equality. The == operator maps to a call of `equals` (in contrast to Java, where == compares object identities).

As a result, comparing Strings with == refers to the actual values and not to object identity. For example, the following expression evaluates to true:

```
"xyz" == new String("xyz")
```

Dependent method types Finally, we make use of an experimental feature of Scala: *dependent method types*. The type of a method is dependent when its return type depends on the value of one of its parameters. Consider the following example:

```
def someMethod[T](arg: T): arg.type = { /*..*/ }
```

The type parameter `T` is inferred from the type of the argument `arg`. At the same time, the type of `arg` specifies the return type of the method. Hence, depending on which value we pass to `someMethod`, it has different return types.[1]

[1] At the time of this writing, dependent method types are only partially implemented in Scala and have to be enabled using the compiler option `-Xexperimental`.

2.2 Role Modeling

The term *role* appears not only in computer science but also in various other contexts. In this section, we focus on roles in the context of object-oriented modeling, that is, on *role modeling*. Large parts of role modeling research is concerned with the question how to define roles and their relation to objects and classes. The following presents the most important answers to this question.

Natural Types versus Role Types

The notion of a role seems to be an intuitive one. One can easily get an idea of what role modeling is about by considering the standard example of a person. A person may appear in different roles, for instance, as professor, parent, and sportsman. We also say that a person *plays* different roles. However, when it comes to more realistic examples in the field of object-oriented software, finding roles of objects becomes as ambiguous and debatable as many other modeling decisions.

A useful aid for modelers is to distinguish *natural types* from *role types*. This distinction is mentioned in [Sow84] which compares natural types "that relate to the essence of the entities" and role types "that depend on an accidental relationship to some other entity". Guarino [Gua92] elaborates on these ideas and provides definitions for both terms. These definitions are based on the notions of *founded* types and *semantically rigid* types.[2]

A type T is called founded on another type S if any instance of T has to be necessarily associated to an instance of S, where the relation between both instances must be different from a part-of relation. For instance, the type *Child* is founded on the type *Parent* as no child can exist without a parent. In contrast, the type *Bicycle* is non-founded since the existence of a bicycle relies only on other instances that represent parts of it, for example, instances of *Tire* and *Saddle*.

The second important notion is that of semantically rigid types. A type is semantically rigid if it contributes to the identity of their instances. That is, they cannot drop the type without loosing their identity. For example, the type *Cat* is semantically rigid whereas the type *Pup* is not.

Using both notions, we can finally define natural types and role types as follows:

- *Natural types* are non-founded and semantically rigid. Hence, they do not rely on other types and contribute to the identity of its instances.
- *Role types* are founded and semantically non-rigid. Thus, they rely on other types and are not part of the identity of their instances.

It must be clear that the distinction between role types and natural types is not absolute. What appears as a role type in one context may be a natural type in another. For example, a person may appear in the role of a professor, while a professor can be seen as a natural type having the roles lecturer, researcher, and supervisor. In practical settings, though, the above definitions are still valuable because modelers can concentrate on a specific problem whereas ambiguous cases as the one above occur only rarely.

[2] Guarino [Gua92] uses the term *concept* instead of *type*.

The OOram Software Engineering Method

Reenskaug brought the idea of roles to object-oriented software in his seminal work on *object-oriented role analysis and modeling* (OOram) [RWL96]. OOram assumes that the world is represented by an object model. Different areas of concern of such an object model can be treated independently since each corresponds to a certain pattern of objects. Similar patterns are abstracted into a role model, which describes the structure of a set of objects by assigning an appropriate role to each of them.

Simple role models can be composed by a synthesis operation to larger role models. This yields a derived model and finally leads to an overall collaboration model of all objects. The underlying idea is that objects may be part of multiple areas of concern, playing a different role in each of them. The phenomenon covered by a derived model is a combination of the phenomena covered by the base models it is composed of. Thus, role model synthesis introduces an inheritance-like mechanism on the level of modeling. In contrast to hierarchical decomposition mechanisms, it allows for decomposing large systems into arbitrary structures.

Even though the OOram method has never found its way into mainstream programming, it must be acknowledged for introducing a collaboration-based view on object-oriented software. The ideas brought to light by Reenskaug have been the cornerstone of many works, for instance [VN96, RG98, Rie00, BGE07], and also influence this thesis.

UML Collaborations

The Unified Modeling Language UML [OMGO07] provides *collaborations* to describe the structure of collaborating objects. The goal is to explain how a specific functionality is collectively accomplished by objects playing certain roles. Concentrating on one particular concern of a system, collaborations only incorporate relevant aspects and leave details such as the identity and class of objects aside.

Collaborations may be specialized, where the type of a role in a specialized collaboration must conform to the type of the role in the general collaboration. In contrast, no corresponding specialization for the classifiers that realize those roles is required. It is sufficient that classifiers conform to the constraints defined by the roles.

UML collaborations are not directly instantiable. Instead, they are realized by sets of collaborating instances that belong to classifiers playing the appropriate roles at runtime. The UML standard recommends to represent roles as interfaces, such that several roles can be played by one classifier and instances of otherwise unrelated classifiers can play the same role.

Role bindings are used to connect roles and classifiers by mapping features of a roles to features of classifiers. This mapping indicates which role a classifier plays in a collaboration. One classifier may also play multiple roles in the same collaboration.

The graphical notation proposed by the UML standard is shown in Figure 2.2. It describes two roles *Professor* and *Student* that form the *University* collaboration.

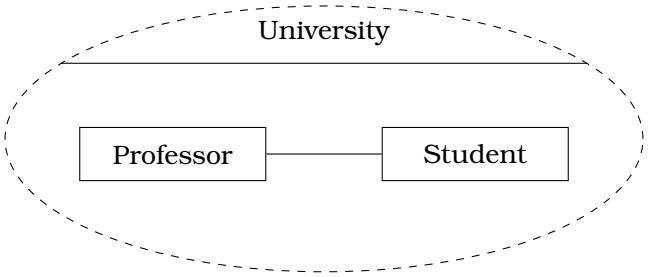


Figure 2.2: A UML collaboration describing a university with two roles *Professor* and *Student*.

Steimann's Role Definition

The concept of roles and role modeling has been discussed by numerous authors. Many contributions come with their own definition of the term *role*, thereby emphasizing different aspects of it. Steimann summarizes large parts of the literature and presents 15 features of roles that are commonly mentioned [Ste00]. In the following, we recite these features and relate them to the example of a person that may play the roles professor and father. We will refer to these features later on to evaluate our approach.

1. *A role comes with its own properties and behavior.*
   When the person is a professor, he has an academic title and may give lectures.

2. *Roles depend on relationships.*
   Being a professor makes only sense if the person is related to a university and students.

3. *An object may play different roles simultaneously.*
   When the person takes his children to work, he can be seen as a father and as a professor at the same time.

4. *An object may play the same role several times, simultaneously.*
   A busy person may work at two universities, thus, playing the professor role twice.

5. *An object may acquire and abandon roles dynamically.*
   Going home after work the person takes off the professor role and starts to play the father role.

6. *The sequence in which roles may be acquired and relinquished may be subject to restrictions.*
   Before becoming a professor, a person should have played the role of a PhD student.

7. *Objects of unrelated types may play the same role.*
   Not only persons can be fathers but, for instance, also animals. The type of animal and person are not necessarily related.

8. *Roles can play roles.*
   A person can play the role of a professor. In turn, a professor may play the roles lecturer and researcher.

9. *A role can be transferred from one object to another.*
   In case the person playing a professor goes into retirement, another person can take over his role and resume his lectures, etc.

10. *The state of an object may be role specific.*
    A person may have no time for students in his role as a professor but still be available for his children while being a father.

11. *Features of an object may be role specific.*
    Persons can have a method `greet` that returns a colloquial *Hi* in the father role and a more formal *Good morning* in the professor role.

12. *Roles restrict access.*
    When the person is a seen as professor, unrelated methods of the person are unvisible.

13. *Different roles may share structure and behavior.*
    The professor role may have the subroles full professor and assistant professor that inherit members from the professor role.

    The last two features are opposed to each other:

14. *An object and its roles share identity.*
    The person and the person playing the professor role have the same identity.

15. *An object and its roles have different identities.*
    The person and the person playing the professor role have different identities.

2.3 Dynamic Proxies

As Scala runs on the Java Virtual Machine, we can benefit from existing Java libraries and frameworks. In particular, we can use the powerful construct of dynamic proxies which is part of the Java standard library. The reflection API provides a class `Proxy`[3] that serves as the entry point for creating dynamic proxies. A dynamic proxy is an object that implements a list of interfaces. These interfaces are given when the proxy is created. The behavior of the object can be implemented using reflection.

```
1 public class Foo implements IFoo {
2   public void sayHello() { System.out.println("Hi all!"); }
3 }
```

Listing 2.1: A simple Java class with one method.

To explain the capabilities of dynamic proxies, we consider a simple example in Java. Let us assume a class `Foo` (Listing 2.1) and that we want to create a log message whenever one of its methods is called. The logging functionality should be implemented without changing the code of `Foo`.

Listing 2.2 shows how we can achieve this. We take an instance `foo` that is of interest for logging and wrap it into a proxy `fooProxy`. In line 6, we create the dynamic proxy with `newProxyInstance`. It requires three arguments:

[3] `java.lang.reflect.Proxy`

```
1 Foo foo = new Foo();
2 ClassLoader classLoader = foo.getClass().getClassLoader();
3 InvocationHandler handler = new LoggingInvocationHandler(foo);
4 Class[] interfaces = new Class[] { IFoo.class };
5
6 IFoo fooProxy = (IFoo) Proxy.newProxyInstance(classLoader, interfaces, handler);
7
8 fooProxy.sayHello();
```

Listing 2.2: The object `foo` is wrapped into a dynamic proxy `fooProxy` to log access to its members.

- The first argument is the class loader responsible for loading the proxy class. Before instantiating a proxy, a class with the required interfaces is created. As the type of the proxy is only known at runtime, the required byte code is generated dynamically and loaded by the given class loader.
- An array of interfaces constitutes the second argument. It specifies the proxy's type. In the example, we use the interface `IFoo` which contains the `sayHello` method.
- The third argument is an `InvocationHandler`.[4] Its task is to handle method calls to the proxy via reflection. Invocation handlers are a powerful means for adapting the behavior of methods.

```
1  public class LoggingInvocationHandler implements InvocationHandler {
2    private Object delegatee;
3    public LoggingInvocationHandler(Object delegatee) {
4      this.delegatee = delegatee;
5    }
6
7    public Object invoke(Object proxy, Method method, Object[] args)
8        throws Throwable {
9      System.out.println("Call to " + method.getName());
10     return method.invoke(delegatee, args);
11   }
12 }
```

Listing 2.3: An invocation handler that prepends each method call with a `println`.

The return type of `newProxyInstance` is `Object`. We therefore have to cast the proxy to `IFoo` before we can use it. This can be safely done as we configured the proxy to have exactly this type.

Our task of logging method calls is accomplished by the invocation handler in Listing 2.3. It takes the object that is wrapped by the proxy as a constructor argument (`delegatee`). The `invoke` method is called whenever a method of the proxy is called. Our implementation prints the name of the originally called method and forwards the call to the delegatee afterwards.

[4] `java.lang.reflect.InvocationHandler`

In general, the invocation handler does not have to call the original method, but can also reply itself or throw an exception.

Running our application from Listing 2.2 produces the following output:

```
Call to sayHello
Hi all!
```

The call to `sayHello` is dispatched to the invocation handler, which prepends a logging message before actually calling it.

3 Lightweight Roles and Collaborations

Before turning to our implementation in Scala, let us delve into more conceptual considerations. In this chapter, we explain language-independently how roles and collaborations can be represented in statically typed object-oriented programming languages using commonly available constructs and mechanisms.

We argue that the approach described in this chapter is not only valuable for Scala. In essence, it 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 via reflection (see Section 2.3 for details on dynamic proxies). Second, a notion of inner classes is needed to describe collaborations. Apart from these two constituents, our implementation uses other features of Scala, such as implicit conversions and dependent method types. Although being helpful in providing a convenient syntax for using roles, they are not absolutely necessary to implement our approach.

In Section 3.1, we argue for a set of requirements that programming support for roles and collaborations should fulfill. Afterwards, Section 3.2 analyzes important design decisions for representing roles in an object-oriented programming language. Finally, we define the terminology used in the remainder of this thesis in Section 3.3.

3.1 Goals

As discussed in [Ste00], a multitude of definitions and views of roles in object-oriented software exist within the research community. The list of features we recited in Section 2.2 subsumes the most important properties of roles and serves as a point of reference for our approach.

In addition to Steimann's criteria, we have further requirements. First, we want to underline the need for a *notion of collaborations*. A collaboration is an aggregation of roles that are related to each other in a certain context. Steimann acknowledges that roles depend on relationships and are only useful in their context (see the second criterion of Section 2.2). However, an explicit notion for a set of related roles is omitted. We believe that such a notion is indispensable to enable reuse of roles, though. A collaboration covers a self-contained set of roles describing a specific area of concern such that it may be used in different applications. This idea stems from the OOram method [RWL96], where the term *role model* is used instead of collaboration.

Second, we want to state rather pragmatic goals from three perspectives:

1. The language developer's point of view: What changes in the programming language are acceptable?

2. The collaboration developer's viewpoint: How do we want to express collaborations?
3. The perspective of collaboration users: How do we want to integrate collaborations into a piece of software?

Language Developer. There are a couple of promising implementations of roles in object-oriented languages that, however, have the drawback of changing their base language fundamentally [OK95, KO96, Her07]. Consequently, a customized version of the compiler is necessary and interoperability with existing pieces of software, development tools, etc. is limited or even impossible. Therefore, we aim at a solution that *conserves the underlying language* and builds only upon existing language constructs.

Collaboration Developer. A major purpose of collaborations is to provide a reuse unit that encapsulates the behavior of multiple objects. Programmers should be able to develop them on a very general level, without making any assumptions about concrete objects playing the roles. In other words, in order to maximize reusability, we require *self containment* of collaborations. A self-contained collaboration can be compiled and delivered independently and its roles can be bound to any set of objects.

Sometimes, roles may rely on certain properties or behavior of the objects they are bound to. For instance, roles of a person may want to access the person's name, and hence, require the underlying object to provide a field `name`. More generally, collaboration developers should be able to *restrict the type of possible core objects*,[1] for example, to ensure that roles can refer to a particular feature of it.

Collaboration User. Developers should be able to use existing collaborations with the *least possible overhead*. This should entail minimal new syntax that needs to be learned and little additional source code to write. Another requirement is *type safety*. As we assume a statically typed language, we want to type check code that uses roles as well, in order to capture certain programming errors already at compile time. In particular, user code should not require any unsafe downcasts.

3.2 Design Issues

In this section, we discuss typical design issues for implementing roles in an object-oriented programming language and our approaches to solve them.

One Type Hierarchy

As mentioned in Section 2.2, we can distinguish natural types, called classes in the remainder, and role types. While the first describe inherent properties of entities, the latter are based on relations to other entities and may be abandoned by an object while retaining its identity. This distinction may lead to the idea of keeping classes and role types separated by creating two independent type hierarchies [Ste00].

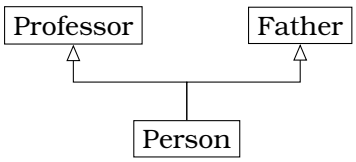
[1] We refer to the object a role is bound to as *core object*. An exact definition of our terminology is given in Section 3.3.

However, we prefer to represent role types as (special) classes for mainly two reasons. First, roles should be able to play roles. For instance, a person may have the role of a *professor* which in turn may play the roles *lecturer* and *researcher*. Role-playing roles, though, are a contradiction to separate type hierarchies; *professor* would have to appear both as a class and as a role type. Second, a pragmatic consideration, building on existing language features such as classes and traits prevents radical changes in the programming language and nevertheless turns out to be able to express roles adequately.

Relating Natural Types and Role Types

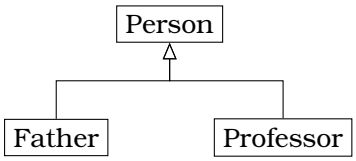
With one single type hierarchy, it seems an interesting idea to relate natural types and role types by subtyping. Basically, there are two options, both having advantages and disadvantages:

- *Role types as supertypes*
  For example, the fact that persons can play the roles father and professor would be represented by:



  This solution seems appealing, as a natural type inherits the features of a role type. Furthermore, several roles can be easily combined if multiple inheritance or mixin-based composition is available. However, representing roles as supertypes implies that all instances of the natural type are considered instances of the role type as well. This contradicts role dynamism, since roles may be played either temporarily or not at all by an instance of the natural type.

- *Role types as subtypes*
  The same example as above would be realized as follows:



  At first sight, this looks natural as roles appear to add state and behavior by specializing a natural type. Unfortunately, this solution prevents roles to be played by instances of otherwise unrelated natural types. A role type would have to inherit from all natural types it could possibly be bound to. This implies that role-playing instances are instances of all of these natural types. Assume, for example, to have a natural type *Animal*. In order to let animals play the father role, *Father* must not only extend *Person* but also *Animal*, such that all fathers would be animals and persons at the same time.

The main advantage of combining natural types and role types by subtyping is polymorphism. Role-playing instances that can equally be treated as having the natural type and the role type. However, as shown above, both possible subtype relationships seem incompatible with the concept of roles. We therefore choose to avoid subtyping and imitate polymorphism with dynamic proxies as described below.

Roles versus Traits

Roles and traits have similar objectives. Both extract parts of a class into a separate abstraction unit in order to enhance composability and reusability. Schärli et al. [SDNB03] state: "The purpose of traits is to decompose classes into reusable building blocks by providing first-class representations for the different aspects of the behavior of a class."

The main difference between traits and roles is the composition mechanism, in particular the binding time. Traits are mixed into a class during its instantiation. Once a set of traits is combined, the set of members of the resulting object is immutable. In contrast, roles can be attached to existing objects, such that we can compose behavior at arbitrary points during the lifetime of an object. Thus, roles provide even more flexibility concerning the composition of behavioral aspects than traits. To put it in a striking way: roles are dynamic traits.

Roles as Objects

A major question is whether the state and behavior covered by a role should be part of the role-playing instance or rather be represented by an additional instance that is in some way attached to the core instance. As objects should acquire and abandon roles dynamically, a one-object solution calls for adding (removing) state and behavior to (from) objects at runtime. Although this is possible in some research languages [DDDCG01, Ser99], it is not supported by today's mainstream programming languages.

Respecting the given language constraints and our requirement to leave the underlying programming language untouched, we opt for representing each role as a separate object. However, users should not be aware of dealing with multiple objects, but rather deal with a compound object including the core object and currently attached role objects (Figure 3.1).

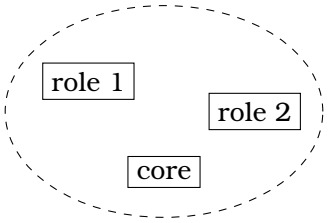


Figure 3.1: Conceptually, core object and role objects belong to a *compound object* (represented by an ellipse).

Dynamic Proxy

Readers might ask how to represent multiple objects by one object in a type-safe fashion when roles can be added to and removed from a core object at runtime. The answer is a dynamic proxy, as for example provided by the Java API (see Section 2.3). A dynamic proxy is an object whose type is given dynamically on its instantiation and whose behavior can be controlled via reflection.

The fundamental idea of our approach is to create such a proxy each time a core object is required to play a certain role. We configure the proxy to have the type of the role object and that of the core object. In this way, role-playing objects can be type-safely accessed without downcasts (Figure 3.2). Internally, all calls are intercepted and forwarded either to the core object or to one of the role objects. This mechanism allows for sophisticated forwarding policies. For instance, is is possible to specify that a method call should only be forwarded to the role object when a condition holds. A reasonable and simple default is to reflectively forward to role objects whenever they provide the required method and to the core object otherwise. Following this strategy, it is possible that roles override the behavior of their core object.

In Figure 3.2, we added type information to illustrate why accessing the proxy is type-safe for clients. Assuming that the core has the type `Player` and the roles are typed as `R1` and `R2`, we can set the proxy type to be compatible with all three of them. Hence, the proxy object provides all members of the core object and the role objects. This property can be statically checked by the compiler.

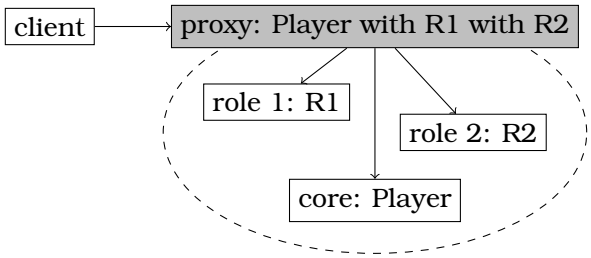


Figure 3.2: The proxy intercepts calls to the compound object and forwards them via an invocation handler.

Representing Collaborations

We propose to represent collaborations as outer classes whose inner classes are treated as role types. An instance of it, a *collaboration instance* or *context*, embodies a concrete context of collaborating objects. Consider for instance a collaboration `University` with two role types `Student` and `Professor`. The concrete collaboration of two role instances `aStudent` and `aProfessor` would then be encapsulated by an instance `aUniversity` of type `University`. The surrounding collaboration instance mainly serves two purposes:

- Internally storing collaboration specific data, for instance, the state of a collaboration or metadata such as the binding between core objects and roles.

- Access to a concrete role from the outside using the collaboration instance as a qualifier.

The second allows to identify roles even if one core object plays the same role multiple times in different contexts. Accessing a role-playing object can thus be seen as a function:

$$core \times role \times context \rightarrow core\,as\,role$$

In different programming languages, inner types are handled differently. In Java, for example, the inner type depends only on the outer type, such that accessing the inner type via one instance or another amounts to the same. In contrast, Scala has path-dependent types, that is, the type of inner types is different for each instance of the outer type (see also Section 2.1). With path-dependent types, certain errors in programs using roles can be found by the compiler which could otherwise only be detected at runtime. We will give a concrete example in Chapter 4.

Dynamic Role Binding and Role Transfer

A major feature of our approach is the possibility to attach and remove roles to and from objects dynamically. If roles would be bound to classes instead, they could be easily represented by mixins (see Section *Roles versus Traits* above).

An interesting question is how strong the cohesion between a role and the object that plays that role should be. Two different models seem reasonable. On the one hand, roles are transient in the sense that we can bind them to one object at one moment and to another later on. On the other hand, roles may be sticky in some applications and should only rarely or not at all change their core object.

We propose to leave the choice between both approaches to programmers by providing *transient roles* and *sticky roles*. Both kinds of roles are stored in a collaboration. Transient roles can be accessed using a method `as` or its inverse `playedBy` that establishes a temporary binding between a given core object and a role instance. The following expression returns a role-playing object:

```
collaboration.someRole.playedBy(someObject)
```

To change the role player, the method is called again, yielding another object playing that role:

```
collaboration.someRole.playedBy(anotherObject)
```

In contrast, sticky roles are bound to their players when creating the surrounding collaboration, for example, by passing them as constructor arguments:

```
collaboration = new Collaboration(someObject)
```

As the binding is already established, we can access role-playing objects without referring to core:

```
collaboration.someRole
```

If a sticky role should be bound to another core object later on, we propose a method `bind`:

```
collaboration.someRole.bind(anotherObject)
```

When a role is transferred from one core object to another, its state should be retained. *Stateful roles* are easy to realize if roles are present as objects. An implementation must ensure to always take the same role object for a certain role, independent of the core object playing it.

Multiplicities per Collaboration

A collaboration defines a number of role types. Sometimes, programmers require exactly one instance of each role type, whereas in other situations, arbitrarily many instances should be supported. This is mostly a modeling question and cannot be answered in general. Suppose to model the relationship between students and supervisors at a university. One valid solution is a collaboration instance containing exactly two role instances, `theStudent` and `theSupervisor`. Hence, we need a new instance of the collaboration for each individual relationship. Another option is to model all supervisions of a supervisor with one collaboration instance, such that there are arbitrarily many instances of the role type `Student` but only one instance of `Supervisor`. A third option consists of arbitrarily many supervisors and students, possibly representing all supervision relationships of a department in one collaboration instance.

To give programmers the flexibility to decide upon multiplicities as needed, we propose two ways to deal with role instances in a collaboration. In both cases, one defines the required role types. For roles with a fixed number of instances, they are directly instantiated in the collaboration and made available to the user.

In contrast, roles with an arbitrary number of instances are visible to users in form of *role mappers*. A role mapper creates a new role object whenever a core object that has not yet played the role should be bound. To allow for stateful roles, role objects and their binding to a core object have to be stored, such that the same role object can be reused when a certain core object requires the role another time.

To access role-playing objects, we propose that role mappers overload the method `playedBy`; a new role instance is created whenever a yet unknown core object should be bound. Thus, the following calls will create three different role instances:

```
collaboration.someRole.playedBy(someObject)
collaboration.someRole.playedBy(anotherObject)
collaboration.someRole.playedBy(stillAnotherObject)
```

With this semantics of `playedBy`, we cannot use it to transfer roles from one object to another. The reason is that we cannot unambiguously refer to a specific role instance because there are arbitrarily many of them. Instead, we can transfer roles by giving not only the new player but also the old one. Hence, we propose that role mappers provide a method `transfer`:

```
collaboration.someRole.transfer(oldPlayer, newPlayer)
```

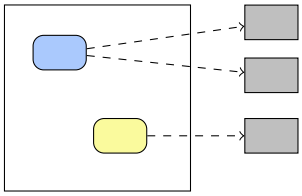
Summary: Kinds of roles

Let us summarize the discussion on different kinds of roles and role mappers. We propose three concepts behind the general term *role*. The following sums up their basic features and depicts them graphically. In the figures, a dashed line denotes a temporary binding of a role to a core object, whereas

a solid line stands for a permanent binding. Multiple dashed lines from one role instance indicate that it has been transferred from one player to another.

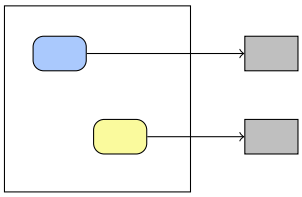
1. *Transient roles*

   - Fixed number of role instances per collaboration instance.
   - Access to role-playing objects by temporarily attaching roles with `playedBy` or `as`.
   - Transfer of roles by calling `playedBy` or `as` again.



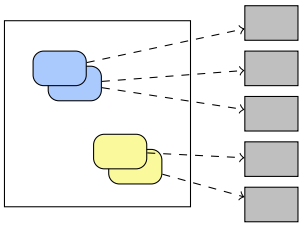
2. *Sticky roles*

   - Fixed number of role instances per collaboration instance.
   - Permanent binding of roles when creating collaboration instances and rebinding with `bind`.
   - Access of role-playing objects with `collaboration.role`.



3. *Role mappers*

   - Arbitrary number of role instances per collaboration instance.
   - Binding and accessing with `playedBy` or `as`.
   - Transfer of roles with `transfer`.



Object Identity

Another issue is to clarify the notion of object identity for role-playing objects. As a result of representing roles by individual objects, programmers find multiple objects where, conceptually, only one is expected. In our approach, the identity of role-playing objects seems to be split up into a core object and one or several role objects. Problems arising from this situation have been summarized as *object schizophrenia* [Har97]. The main problem to resolve is the unclear notion of object identity that can, for instance, lead to unexpected results when comparing objects.

The question of what the identity of a role-playing should be amounts to whether the following two expressions should evaluate to true or false:

```
object as role == object
object == object as role
```

Two arguments seem valid:

- *Roles change identity.* On the one hand, one can argue that adding a role to an object changes its identity. Hence, a person and the same person as a father can, for example, be seen as different objects in a hash set. In this case, the two expressions given above should both evaluate to false.
- *Roles do not change identity.* On the other hand, roles are, at least according to their ontological definition (see Section 2.2), independent of the identity of their players. In fact, the identity-related properties of person and father, such as its name and date of birth, seem to be equal. Hence, one can also argue that the above expressions should both evaluate to true.

Consequently, we propose to allow for both notions of object identity, such that programmers can choose depending on their needs. If and how this can be realized in a programming language is discussed in Chapter 4.

Forwarding versus Delegation (Self Problem)

A subtle issue when implementing roles is to distinguish between *forwarding* and *delegation*. At first, let us define both terms. Suppose we have two objects `a` and `b`, where `a` has a reference to `b`. When `a`'s method `m` is called, it calls `b`'s method `m` and returns its result. Imagine now that in `b`'s `m`, another method `this.n` is called. In case of forwarding, the callee will be `b`, whereas in case of delegation, the dispatch starts again at `a`. This is illustrated in Figure 3.3.

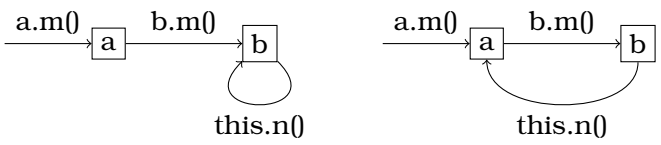


Figure 3.3: Forwarding (left) versus delegation (right).

To achieve delegation, the value of `b`'s `this` must be overwritten with `a`. More generally, the value of `this` must be set to the original receiver of

a method call whenever a call is delegated. As `this` is also called `self` in some languages, the issue of realizing delegation is also called the *self problem* [Lie86].

Whether one can achieve delegation depends on the programming language at hand. As we show in Section 4.1, delegation can be provided in Scala. In the following, we will discuss the more general case where the language only supports forwarding.

In this case, method dispatch is managed by the proxy and the role objects. The proxy delegates calls to role-playing objects based on some policy. The default is to delegate calls to the role object if possible and otherwise call the core object. Hence, role objects may override methods from their core objects. When a role method is invoked, it can decide on which object it calls further methods. The `this` variable points to the role object itself, however, it can also use `core` that refers to the core object. This case is depicted in Figure 3.4, left-hand side.

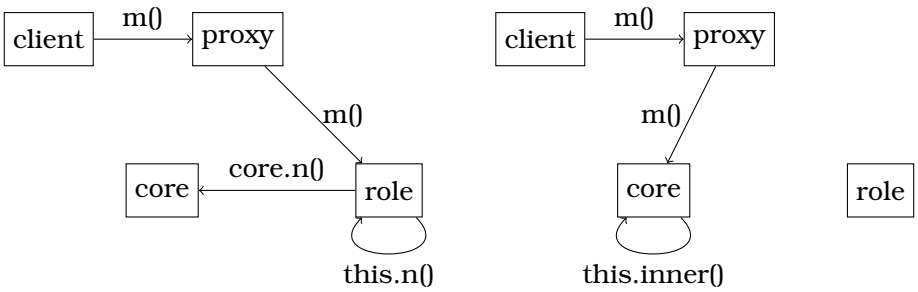


Figure 3.4: Role objects can choose between calling on `this` or on `core` (left), whereas core objects are oblivious of attached roles (right).

Unfortunately, we loose this choice when calling the core object as shown in Figure 3.4, right-hand side. As core objects are oblivious of having an attached role, they can only call their own methods but not those of the role object. To solve the problem, the implementation language must provide some mechanism to set the value of the `this` variable when calling a method. If it does, we can set `this` to the proxy and dispatch all method calls following a policy of our choice.

3.3 Terminology

After explaining our design choices, we want to fix the terminology used in this work. Let us begin with the most fundamental concept, namely a role.

> **Definition: Role**
> A *role* describes the state and the behavior of an object in a certain context.

Roles do not occur alone but always depend on other roles.

> **Definition: Collaboration instance (or context)**
> A set of related roles that collaborate in a certain context is called *collaboration instance* or *context*.

Types are used in programming to describe a set of similar instances. In case of roles, this idea leads to role types.

Definition: Role type
A *role type* describes a set of similar roles, that is, having similar state and behavior. Roles are instances of role types.

Similarly to roles, collaboration instances can be abstracted to collaborations.

Definition: Collaboration
A set of related role types is called a *collaboration*.

The above definitions are valid in general, that is, independent from a specific implementation of roles in a programming language. The following three definitions are related to our choice of implementing roles as objects.

Definition: Role object
An object representing a role at runtime is called *role object*.

Role objects cannot be accessed directly but must be bound to a core object.

Definition: Core object
Any object to which at least one role is bound is a *core object*.

Note that role object and core object are relative notions. In principle, a role can be bound to another role, which implies that one object must be seen as role object and core object at the same time.
Since we want to hide the fact to deal with multiple objects, we introduce the notion of compound objects.

Definition: Compound object
A compound object is a conceptual structure containing one core object and one or more role objects.

4 The Scala Roles Library

The value of conceptual thoughts multiplies with a concrete realization. This chapter presents the implementation of the ideas of Chapter 3 in form of a Scala library for role-based programming. We explain interesting implementation issues and our solution to them. Afterwards, the usage of our library is illustrated by examples. Finally, we evaluate our work using Steimann's criteria [Ste00] and analyze the performance of the library.

4.1 Implementation Issues

The two most important parts of the library are the following. First, we have a base trait for collaborations. It contains different kinds of roles and members to access and use them. Second, there is an invocation handler that reflectively manages calls to the dynamic proxy that represents role-playing objects. It fixes the protocol of interaction between clients, cores, and roles.

During the implementation we encountered a number of issues, both of technical and conceptual nature. On the technical side, problems such as setting the type of role-playing objects for arbitrary combinations of core and role objects have to be resolved. At the same time, there are fundamental questions, such as distinguishing forwarding and delegation. In the following, we highlight the most interesting problems we came across and discuss our solution to them.

Interface Extraction

We represent role-playing objects through dynamic proxies, which are available in the Java API (see Section 2.3). Their hallmark feature is that the type of a dynamic proxy can be set dynamically by giving a set of interfaces. Internally, a new class implementing these interfaces is created and instantiated. To represent a compound object with a dynamic proxy, we require a set of interfaces containing all members of the objects that should be combined.

In Scala, sets of objects can be abstracted by classes and traits. Traits can be mixed into other traits and classes without having a fixed position in the class hierarchy. As a result, they cannot simply be translated into usual classes when compiling Scala source code into Java bytecode. Instead, the Scala compiler creates a corresponding interface for each trait. It contains exactly the members of the trait, such that it can be used wherever the type of the trait is needed. For instance, for the trait

```
trait T {
  def increment(i: Int) = i+1
  def decrement(i: Int) = i-1
}
```

the following interface is created:

```
public interface T extends scala.ScalaObject {
  public int decrement(int);
  public int increment(int);
}
```

This translation scheme allows us to reflectively extract an interface for each trait. Hence, we can represent the type of multiple objects by combining their corresponding interfaces.

The situation is more complicated for instances of classes or Scala objects, because they are not accompanied by an interface. We therefore restrict our library to work with traits. As an outlook, we experimented with dynamically creating interfaces for these types using the ASM framework for Java bytecode manipulation.[1] Extending our library to support instances of arbitrary types seems a promising direction for future work.

Handling Invocations

Each instance of `java.lang.Proxy` is parametrized with an invocation handler. Its task is to reflectively deal with calls to the proxy. We provide an invocation handler that either calls the core or the role object.

The power of our approach stems from the reflective handling of method calls to role-playing objects. It allows us to imitate polymorphic method dispatch such that, seemingly, the role object extends the core object. Our method dispatch policy is to call the role object whenever it provides the required method and fall back on the core object otherwise. Hence, a role can override members of the core. In prototype-based programming languages (as opposed to class-based ones), such as Self [US87], objects can be combined in a similar way.

One might ask what happens if neither the role object nor the core object provides the required method. This cannot happen, though, as we configure the proxy to have exactly the type that results when combining the interfaces of the core and the role object. As a result, a call a non-existing method on a role-playing object will already be identified during compilation.

Another remark concerns field accesses. Since Java interfaces may only contain methods and constants (that is, final static fields), accessing non-static fields of role-playing objects seems to be an issue. However, the Scala compiler generates getter and setter methods for all fields of an object. For instance, the trait

```
trait T {
  var state = 3
}
```

is translated into

```
public interface T extends scala.ScalaObject {
    public void state_$eq(int);  // setter
    public int state();          // getter
}
```

Each field access in Scala is implicitly translated into a method call to the corresponding setter or getter. Thus, we do not have to consider field accesses particularly, because they are already treated by Scala's method dispatch mechanism.

[1] `http://asm.objectweb.org`

Solution to the Self Problem

As explained in Section 3.2, the self problem is an important challenge for an implementation of role-based programming. We can solve it by, once again, taking advantage of Scala's translation scheme of traits. Besides the interface that is created for each trait, there is an abstract class providing the trait's implementation as static methods. For example, the trait

```
trait T {
  def increment(i: Int) = i+1
  def decrement(i: Int) = i-1
}
```

yields not only the interface given above but is also accompanied by the following class:

```
public abstract class T$class {
  public static int increment(T $this, int i) { return i+1; }
  public static int decrement(T $this, int i) { return i-1; }
}
```

Calling a method of `T` is translated by the Scala compiler to a call to the corresponding method in `T$class`. Note that each static method has an additional parameter `$this` of type `T`. This parameter provides a reference to the object on which the method is invoked and is used instead of `this` inside the static methods.

We use this translation scheme to realize delegation (as opposed to forwarding) when dispatching methods in our invocation handler. We call the static method and pass the proxy as the value for `$this`. Thus, the self variable always refers to the proxy, that is, the original receiver of a method call. Figure 4.1 depicts the delegation protocol in prototype-based languages (left-hand side) and the protocol using our approach (right-hand side). Since the dispatch policy of the proxy is to call the role if possible and the core otherwise, the resulting behavior is exactly the same as that of the core delegating to the role.

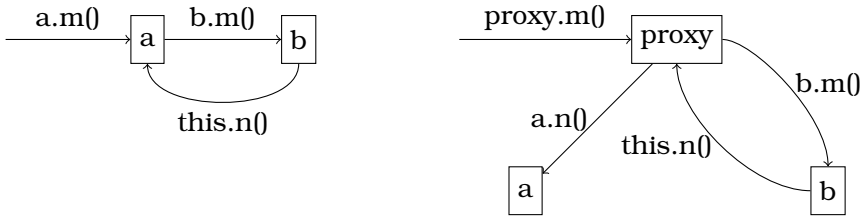


Figure 4.1: Delegation in prototype-based languages (left) and delegation with a proxy and traits (right) yield exactly the same behavior.

Object Identity

As we argued in Section 3.2, we aim at supporting two ways for defining object identity. On the one hand, an object and the same object playing a role can share one identity. On the other hand, roles may change an object's identity. We allow for both in our library such that users can decide which semantics is more appropriate for their needs.

Generally, there are four interesting cases when comparing objects and role-playing objects:

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

Let us, at first, consider shared identity. This is the default in our library because it corresponds to the ontological definition that role types do not contribute to the identity of an object. With shared identity, all four expressions should evaluate to `true`.

To achieve (seeming) object equality between objects and role-playing objects, we modify identity-related methods of dynamic proxies. One can 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 core objects to inherit from a type `RolePlayer` which contains an adapted `equals` method. If the argument of the adapted `equals` is a proxy, it would compare with its core object and otherwise fall back on the default implementation of `equals`. However, this makes adding roles to arbitrary objects impossible. Finding a satisfactory solution to this issue remains as future research.

Realizing the second point of view, that is, roles that change the identity of their core, is easier. In this case, expressions (1), (2), and (4) should result in `false`, which is automatically obtained if we do not modify identity-related methods. In contrast, expression (3) should evaluate to `true`. Therefore, we have to make sure that each time a certain role is attached to an object, the same proxy is used. To achieve this, we store the binding of core objects to proxies in a hash map for each role and reuse an existing proxy if possible.

Collaboration developers can choose the semantics for object identity by setting the flag `sharedIdentities`. The default value is `true`.

The `as` Operator

There are different ways to access role-playing objects. Transient roles, that is, those that are only temporarily attached to core objects, and role mappers, hiding an arbitrary number of role instances, can be accessed with the `as` operator. For instance, `obj as collab.myRole` results in `obj` playing the role `myRole` which belongs to the collaboration `collab`.

`as` and `playedBy` are complementary in the sense that `object as role` and `role playedBy object` are equivalent. However, we believe `as` to be more intuitive, and hence, focus on its implementation in the following.

In Scala, all method calls can be written as infix operators. Hence, `obj as role` is equivalent to `obj.as(role)`. However, we want to bind roles to arbitrary objects, that is, we cannot assume `obj` to provide an `as` method. In contrast, we can provide the complementary method `playedBy` in role objects. We found two ways to allow for an `as` method in Scala.

One solution uses the fact that, in Scala, methods ending with a colon ':' are considered to be right-associative when used as an infix operator. Thus, `obj -: role` is equivalent to `role.-:(obj)`. Using this, we provide a method `-:` in role objects that is intended to be used in infix position and read as `as`. The drawback of this approach is that users have to remember the cryptic method name `-:`.

Another approach makes use of implicit conversions to turn a method call `obj as role` into `role.playedBy(obj)`. An implicit conversion is a special method inserted by the compiler whenever an expression would not otherwise be type-correct.

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

Listing 4.1: An implicit conversion that adds the method `as` to arbitrary objects.

The implicit conversion in lines 1 to 2 of Listing 4.1 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 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 5).

However, the second approach is not universally applicable. Suppose we have a role for some type `T`, that is, `Role[T]`, and want to attach it to an instance of some subclass `S` of `T`. The type of the role `r` in line 5 will be inferred to `Role[S]`, which is too restrictive for our role of type `Role[T]`. Unfortunately, we cannot solve this problem without loosing some type safety, and hence, `as` can only be used when the core object has exactly the type that the role expects.

Our library includes both, a universally applicable method `-:` and a method `as` with the above mentioned restriction. We propose to use the latter if possible and fall back on `-:` otherwise.

Instances of Collaborations

We represent collaborations as outer traits whose inner traits can be role types. Our library contains a base trait `Collaboration` and several subtraits to provide the different flavors of roles, transient roles, sticky roles, and role mappers as described in Section 3.2. To use a collaboration in a concrete context, it must be instantiated. A collaboration instance gives access to its roles.

As Scala supports path-dependent types, we can identify certain errors concerning multiple instances of one collaboration at compile-time. Consider, for example, a collaboration `Conversation` (Listing 4.2) describing persons talking to each other and the application in Listing 4.3 using it. The last line would cause a type error, because the type `Participant` depends on the surrounding instance of `Conversation`. Hence, instances of

the same role type belonging to different collaboration instances are distinguished statically.

```
1 trait Conversation extends TransientCollaboration {
2   object participant extends RoleMapper[Person, Participant] {
3     def createRole = new Participant{}
4   }
5
6   trait Participant extends Role[Person] {
7     def talkTo(other: Participant) = { /*..*/ }
8   }
9 }
```

Listing 4.2: A simple collaboration describing the conversation of persons.

```
1 val paul = new Person{}; val marry = new Person{}
2 val jim  = new Person{}; val jane  = new Person{}
3
4 val conv1 = new Conversation{}
5 val conv2 = new Conversation{}
6
7 (paul as conv1.participant).talkTo(marry as conv1.participant)
8 // (jim as conv2.participant).talkTo(marry as conv1.participant)
```

Listing 4.3: The last line would yield a type error because participants from `conv1` and `conv2` have different types.

Advising Members

Our role library allows not only to enhance objects with new members, but also to adapt existing members to new requirements. Similar to aspect-oriented programming [KLM+97], programmers can specify functions to be executed before and after existing methods, as well as before and after setting fields. In aspect terminology, we allow for dynamically advising members of objects.

An important question is how to select members that should be advised with new functionality. Implementing a library, we cannot add new language constructs for this task. Instead, we use a string-based solution that is easy to understand and use. Three kinds of strings are allowed:

- *Field selectors*, for instance `"f"`, select a specific field of a core object by its name. The attached behavior is executed whenever the field is set to a new value.
- *Method selectors*, for instance `"m()"` or `"m(Int,Boolean)"`, select a specific method of the core object by its name and the parameter types. The attached behavior is executed whenever the method is called.
- *Wildcard method selectors*, for instance `"m(*)"`, select all methods of a core object with a given name. The attached behavior is executed whenever a method with the given name is called.

As a simple example, let us count how many times certain methods get called. The following applications introduces a counter for all methods `m` with arbitrary parameter types and all methods `n` without parameters:

```
val counter = new Counter("m(*)", "n()")

(myObj -: counter.counted).m(5)
```

After adding the `counted` role to an object, calls to it are counted. The corresponding collaboration is shown in Listing 4.4. It takes a parameter list of arbitrary length (indicated by the `*` behind `String` in line 1) and contains one role type `Counted`. In line 5, we relate the parameter list with the method `increment` by calling `addAfterCalls`. It specifies that after each access to one of the members described in `toCount`, a method call to `increment` should immediately follow. Likewise, there is a method `addBeforeCalls` available in each role to prepend functionality to existing members.

```
1  class Counter(toCount: String*) extends TransientCollaboration {
2    val counted = new Counted{}
3
4    trait Counted extends Role[AnyRef] {
5      addAfterCalls(increment, toCount:_*)
6
7      var counter = 0
8      def increment() = {
9        counter = counter + 1
10       println("counter: " + counter)
11     }
12   }
13 }
```

Listing 4.4: The `counted` role appends a call to `increment` to all methods specified in the repeated parameter `toCount`.

String-based member selectors like the above `"m(*)"` have the drawback that the compiler cannot verify whether a string may actually match a member of the core type. Also, programmers are not notified in case a member changes such that an existing member selector becomes invalid.

Internally, we can easily realize method advices since each call to a role-playing object is treated reflectively by the invocation handler. Before and after executing a method, the invocation handler checks whether additional functionality has been registered for this method. Field accesses are treated similarly since they are translated to getter and setter methods by the Scala compiler.

Roles Playing Roles

An important feature of role-based programming is that roles themselves can play roles. One idea to enable this feature is to reuse the existing proxy of a role-playing object by adding a new role-object to the compound object. However, the type of a proxy is specified during its creation, and cannot be changed afterwards. Hence, we always have to create a new proxy when adding a role.

A better approach is to treat a role-playing object as the core object for new roles. For example, consider a person that plays the role of a professor, which in turn plays the roles lecturer and researcher. Using our library, this scenario can be expressed as follows:

```
(person as university.professor) as professorship.researcher
(person as university.professor) as professorship.lecturer
```

The first line returns the person as a researcher, while the second makes him appear as a lecturer. If we define researcher and lecturer as roles of professors, we cannot add them to persons directly. For instance, the following would result in a type error at compile time because the person is not necessarily a professor:

```
(person as professorship.researcher)
```

Figure 4.2 depicts how role-playing roles are internally represented by nested proxies. Method calls to the researching professor are delegated by the upper proxy to the role object `researcher` or to the lower proxy. This, in turn, delegates either to `person` or to `professor`. A drawback of this solution is efficiency, since method calls may go through reflection multiple times.

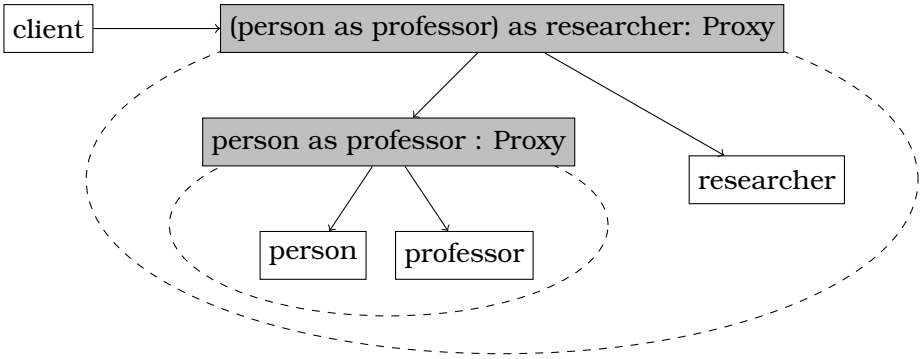


Figure 4.2: Role-playing roles are represented by nested proxies, such that one proxy appears as the core object of the other.

4.2 Usage and Examples

This section gives three examples of how to use our library and explains the different possibilities of collaboration developers and users. We present three mechanisms – transient roles, sticky roles, and role mappers – describing each time a university with ordinary students, PhD students, and professors. For the conceptual differences of these types, readers are referred to Section 3.2.

Transient Roles

Transient collaborations extend the trait `TransientCollaboration`. It contains a trait `Role` to be extended by inner traits that represent a role. `Role` takes a type parameter that specifies the upper bound for core objects. The

number of instances of transient roles inside a collaboration instance is constant. A collaboration developer has to instantiate the required number of roles to make them accessible for users.

From a user's perspective, roles are contained in an instance of a collaboration. After creating one, transient roles are accessed with the `as` operator that binds a named role instance to a core object. If the same role instance is used again with the same or another core object, the state of the role is retained. This allows for seamlessly transfering a role from one object to another.

```
1  trait ThesisSupervision extends TransientCollaboration {
2    val supervisor = new Supervisor{}
3    val student = new Student{}
4
5    trait Supervisor extends Role[Person] {
6      def advise = student.motivation += 5
7      def grade = student.core.name + " is " +
8        (if (student.wisdom > 80) "excellent" else "satisfactory")
9    }
10
11   trait Student extends Role[Person] {
12     var motivation = 50
13     var wisdom = 0
14     def work = wisdom += motivation/10
15   }
16 }
```

Listing 4.5: A collaboration with two transient roles, `supervisor` and `student`.

Let us look at a concrete example. Listing 4.5 shows a collaboration with two role types, `Student` and `Supervisor`, each of which are instantiated exactly once (lines 2 and 3). As the role instances are fixed, that is, it is not possible to create new instances dynamically, the implementation of the role types can refer to them. For example, the supervisor accesses the `wisdom` field of the student for the purpose of grading (line 8). Roles can access their own core object as well as the core object of other roles inside the collaboration (line 7).

A small application using the above collaboration is given in Listing 4.6. There are three persons: a master student, a PhD student, and a professor. They are related through two supervision collaborations describing a master project and a PhD thesis. One should note that `Paul` is playing different roles, appearing as master supervisor in line 9 and as PhD student in line 11.

Role Mappers

Role mappers are a convenient mechanism to provide an arbitrary number of instances of a role type by creating them on demand. At the same time, the roles it creates are transient, and hence, a collaboration with role mappers extends `TransientCollaboration`. Instead of instantiating role types directly, programmers create an associated role mapper by extending `RoleMapper`. This trait is parametrized with a core type and a role type. Ad-

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

Listing 4.6: Usage of the collaboration from Listing 4.5. Roles are bound temporarily using the `as` operator.

ditionally, role mappers have to override a method `createRole` that specifies how to create a new role instance.

For users, role mappers are very similar to transient roles. The main difference is that `as` does not transfer roles from one core object to another but rather creates a new role instance each time it sees a new core object. To transfer roles, one has to use the method `transfer`.

```
1  trait ThesisSupervision extends TransientCollaboration {
2    object supervisor extends RoleMapper[Person, Supervisor] {
3      override def createRole = new Supervisor{}
4    }
5
6    object student extends RoleMapper[Person, Student] {
7      override def createRole = new Student{}
8    }
9
10   trait Supervisor extends Role[Person] {
11     def advise(student: Student#Proxy) = student.motivation += 5
12     def grade(student: Student#Proxy) = student.name + " is " +
13       (if (student.wisdom > 80) "excellent" else "satisfactory")
14   }
15
16   trait Student extends Role[Person] {
17     var motivation = 50
18     var wisdom = 0
19     def work = wisdom += motivation/10
20   }
21 }
```

Listing 4.7: A collaboration with two role mappers `supervisor` and `student` creating new role instances on demand.

Listing 4.7 is similar to the previous example. However, the number of students and supervisors per collaboration is not limited anymore. Instead, lines 2 to 8 create role mappers for the roles `Supervisor` and `Student`. Also, the role type implementation cannot directly refer to other role instances

anymore. In fact, users must pass them, for instance, to the method `grade` in line 12. The type of `grade`'s argument, `Student#Proxy`, describes an object playing the role of a student.

```
1  val fritz = new Person{ val name = "Fritz" } // a master student
2  val klaus = new Person{ val name = "Klaus" } // a master student
3  val paul  = new Person{ val name = "Paul"  } // a PhD student
4
5  val master = new ThesisSupervision{}
6
7  (fritz as master.student).work
8  (klaus as master.student).work
9  (paul as master.supervisor).advise(fritz as master.student)
10 (paul as master.supervisor).advise(klaus as master.student)
11 (paul as master.supervisor).grade(klaus as master.student)
```

Listing 4.8: The supervision of multiple students is encapsulated in one collaboration instance `master`.

An application is given in Listing 4.8. Although multiple students are supervised, only one instance of the collaboration is required. The rather wordy syntax can be shortened by assigning role-playing objects to a variable. For example, we could write

```
val supervisingPaul = paul as master.supervisor
```

and use this as a shortcut subsequently.

Sticky Roles

In contrast to transient roles, sticky roles are not intended to change the core object frequently. Instead, the objects that participate in a sticky collaboration are given in its constructor. One can create such a collaboration by extending `StickyCollaboration`, which contains a trait `Role`. Sticky roles offer a method `bind` which is typically called in the collaboration's constructor and attaches them to a core object.

Since the core object is fixed, one can use sticky roles without the `as` operator. Instead, the role-playing object is directly accessed, for example with `collab.role`. Such collaborations are similar to first-class relationships [BW05, BGE07], which encapsulate a number of participants and make the relation between them more explicit.

Listing 4.9 shows our running example as a sticky collaboration. It takes two constructor arguments to specify the persons that participate in the collaboration. In lines 4 and 5, we instantiate the `Supervisor` and `Student` roles and bind them to the core objects in lines 7 and 8. Users can access the role-playing objects through the methods `supervisor` and `student` that return the corresponding proxy.

The example application in Listing 4.10 differs from the preceding examples in two ways. First, we pass the participating persons to the collaborations when instantiating them (lines 5 and 6). Second, role-playing objects can be accessed with a more concise syntax; the core object needs not to be specified explicitly.

```
1  class ThesisSupervision[SupervisorP <: Person, StudP <: Person]
2      (supervisorP: SupervisorP, studP: StudP)
3      extends StickyCollaboration {
4    val supervisorRole = new Supervisor{}
5    val studentRole = new Student{}
6
7    supervisorRole.bind(supervisorP)
8    studentRole.bind(studP)
9
10   def supervisor = supervisorRole.proxy
11   def student = studentRole.proxy
12
13   trait Supervisor extends Role[SupervisorP] {
14     def advise = studentRole.motivation += 5
15     def grade = studentRole.core.name + " is " +
16       (if (studentRole.wisdom > 80) "excellent" else "satisfactory")
17   }
18
19   trait Student extends Role[StudP] {
20     var motivation = 50
21     var wisdom = 0
22     def work = wisdom += motivation/10
23   }
24 }
```

Listing 4.9: A sticky collaboration takes the core objects as constructor arguments and binds them to roles immediately.

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

Listing 4.10: Usage of a sticky collaboration. Roles can be accessed without giving the core object each time.

4.3 Evaluation

In the following, we evaluate our library using Steimann's criteria (see Section 2.2). Furthermore, we present a short performance analysis. Finally, we discuss limitations of our approach and give directions for future work.

Steimann Criteria

Our approach meets 13 out of 15 Steimann criteria. Criteria 12 (*Roles restrict access*) would, in principle, be possible with our approach. For instance, `person as student` could return a proxy of type `Student` that internally can delegate to its core object of type `Person`. However, we opted to always use the compound type, for example, `Person with Student`, such that programmers can access all members through one object. Furthermore, criteria 14 (*An object and its roles share identity*) is only partly solved.

1. *A role comes with its own properties and behavior.* Yes, role types can define fields and methods.

2. *Roles depend on relationships.* Yes, role types belong to a collaboration and cannot be used without.

3. *An object may play different roles simultaneously.* Yes, multiple roles can be bound to one object at the same time.

4. *An object may play the same role several times, simultaneously.* Yes, a core object can play roles of the same role type in different collaboration instances. Those roles can be unambiguously accessed due to path-dependent types.

5. *An object may acquire and abandon roles dynamically.* Yes, roles can be attached to objects and removed from them dynamically, for example, using the `as` operator.

6. *The sequence in which roles may be acquired and relinquished may be subject to restrictions.* Yes, a collaboration can keep track of role bindings and throw an exception on illegal sequences. However, we cannot statically ensure such properties.

7. *Objects of unrelated types may play the same role.* Yes, roles can specify an upper bound for their core objects. Choosing `AnyRef` allows for arbitrary objects.

8. *Roles can play roles.* Yes, see *Roles Playing Roles* in Section 4.1.

9. *A role can be transferred from one object to another.* Yes, we provide stateful roles that can be transferred, for instance, using the `as` operator.

10. *The state of an object may be role specific.* Yes, roles contain state and can override the state of their core objects.

11. *Features of an object may be role specific.* Yes, roles contain methods and can override the methods of their core objects.

12. *Roles restrict access.* No, our approach does not allow to diminish the set of members of an object by attaching a role (except by removing previously attached roles).

13. *Different roles may share structure and behavior.* Yes, inheritance of role types inside a collaboration is possible. Furthermore, collaborations can extend others, allowing refinements similar to mixin layers [SB98, SB02].

14. *An object and its roles share identity.* Partly. However, we could not solve this problem completely (see *Object Identity* in Section 4.1).

15. *An object and its roles have different identities.* Yes, one can configure collaborations to have this semantics.

Performance Analysis

Our approach makes use of a dynamic proxy to represent a compound object to the user. The flexibility of this solution comes from dynamically building class files and loading them with a class loader. Furthermore, all calls to the compound object are reflectively delegated by the proxy. Unfortunately, this flexibility does not come for free but involves performance losses. In the following, we describe the results of a first performance analysis and examine possible optimizations.

We perform a micro-benchmark to give an estimate of the performance losses that come with common use cases. Our analysis uses two techniques. On the one hand, we measure the execution time by calling `java.lang.System.nanoTime()` before and after the code fragment of interest. On the other hand, we use the Java Memory Profiler[2] to analyze in which methods the most time is spent during the execution of our test cases.

We constructed a use case involving nine objects that play roles in six collaborations. During one execution of the example, 21 method calls to role-playing objects are performed. For comparison, the same example has been implemented in Scala in two other styles: without roles and using the Role Object pattern. In the first case, roles are replaced by subclasses, whereupon, of course, all dynamic properties of the role concept are lost.

The Role Object pattern approach is, on average, between two and three times slower than the version without roles. Using our library causes a slowdown of factor five to six. Profiling has shown that most of the performance loss is due to generating the dynamic proxy. One possible optimization is to cache the proxy created for a pair of a core object and a role object inside a collaboration. However, caching only pays off when one object is used multiple times playing the same role.

Overall, optimizing the performance of the library was not our main focus. Also, the above results are only micro-benchmarks. The actual performance in a real world application may depend on factors other than those analyzed by us.

Limits of the Library Approach

One of our main contributions is to enable role-based programming with a library, whereas most other proposals build new languages or extend exist-

[2] `http://www.khelekore.org/jmp/`

ing ones. This lightweight approach is flexible and satisfies many requirements of programming with roles. However, there exist also some restrictions which give rise to the following list of open problems:

- *String-based member selectors.* We specify which members should be prepended or appended with new functionality using strings. It would be interesting to find a way of specifying members that allows to verify whether the specification actually matches something.
- *Hiding members.* Adding a role widens the set of members of an object. It would also be interesting to diminish it, though, in order to adapt it to a given context.
- *Arbitrary Scala types.* Currently, our implementation works only for instances of traits. As future work, it will be interesting to extend support to instances of arbitrary Scala types, such as classes, objects, and structural types.
- *Equality problem.* The problem of comparing an arbitrary object with a role-playing object with shared identity semantics also remains open.

5 Design Patterns as Collaborations

Since its publication in 1995, the Gang-of-Four book [GHJV95] has had a large impact on practitioners and researchers. Today, design patterns are a well-accepted means for communication and an aid in software design and implementation. However, the claim of its title to provide "Elements of Reusable Object-Oriented Software" seems only partially fulfilled. While the conceptual ideas captured by a pattern are reusable on a design level, there often is no reuse when it comes to implementation. Inspired by role model descriptions of design patterns [Rie96, Rie97b], we investigate in this section to which extent patterns can be expressed as collaborations using our Scala library. The underlying idea is that occurrences of patterns can be represented by attaching roles to objects that interact with each other. Consequently, the pattern-related source code is encapsulated in a collaboration.

Our approach turns out to be useful in two ways:

1. Some patterns can be made *reusable on the implementation level*. As a result, programmers can bind roles from an existing pattern collaboration to objects, instead of extending classes with pattern code. Hence, the original classes need not to be changed, and the pattern implementation can be reused.

2. Although some patterns turn out to be inappropriate for a reusable implementation, capturing them as a collaboration allows for *separating the concern* introduced by the pattern, and hence, improve the structure of the resulting software.

3. Some patterns can be *replaced* by a role-based approach.

As a proof of concept, we provide a design pattern library with reusable implementations of Composite and Observer. Using them brings the following advantages over implementing the pattern directly in the corresponding classes:

- *Less code to write.* The patterns contain source code fragments that are application-independent. We extract them into roles to make them reusable.

- *Dynamic binding and detaching.* Long-living objects may only need pattern-specific behavior in some situations. Using roles, behavior can be added and removed as required.

- *Enhance objects without modifying their source code.* As roles are bound at runtime to existing objects, one can enhance them with pattern-related functionality without accessing or modifying their source code.

We study the 23 design patterns described in [GHJV95] and the Role Object pattern [BRSW00], and analyze if and how they can be represented with roles. As a result, we classify the patterns into the following categories:

- *Reusable Collaboration* (Section 5.1). The pattern contains reusable source code fragments that can be represented as roles.
  - Composite
  - Observer
- *Enhancements with Roles* (Section 5.2). Occurrences of the pattern can be enhanced using roles or completely replaced by a role-based approach.
  - Decorator
  - Mediator
  - Proxy
  - Role Object
  - Template Method
- *Obsolete in Scala* (Section 5.3). The pattern is obsolete in Scala, because there is a simpler solution.
  - Adapter
  - Command
  - Interpreter
  - Singleton
  - Strategy
  - Visitor
- *Invariant* (Section 5.4). The pattern neither contains reusable code nor can satisfactorily be described as collaborating objects.
  - Abstract Factory
  - Bridge
  - Builder
  - Chain of Responsibility
  - Facade
  - Factory Method
  - Flyweight
  - Iterator
  - Memento
  - Prototype
  - State

A related approach investigating how patterns can be transformed into components in Eiffel is given in [Arn04].

We assume readers to be familiar with the Gang-of-Four design patterns. As a short reminder, we give the pattern's intent as given in its original description, before discussing the details.

5.1 Reusable Pattern Collaborations

This section presents reusable implementations of the Composite and the Observer design pattern as collaborations.

Composite

> **Intent** Compose objects into tree structures to represent part-whole hierarchies. Composites let clients treat individual objects and compositions of objects uniformly.

```
1  trait 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 trait BorderFigure extends Figure {
11   var borderColor = "black"
12
13   def addChild(f: Figure) = { /* ... */ }
14   // implementations of other
15   // abstract methods
16 }
17
18 trait TextFigure extends Figure {
19   var text = ""
20
21   // implementations of abstract methods
22 }
```

Listing 5.1: A figure hierarchy implementing the Composite pattern (pattern-related parts are highlighted).

As a motivating example, let us consider a simple figure hierarchy consisting of a base type `Figure` and two subtypes `BorderFigure` and `TextFigure`. They contain source code to describe properties such as colors and the text of the `TextFigure`. Moreover, figures should be nested. We can express this by using the Composite pattern as shown in Listing 5.1.

The Composite-related source code is simply added to the appropriate traits. This approach has two major drawbacks:

- *No separation of concerns.* The concern of nesting is intermingled with inherent properties of figures like their color.
- *No reuse.* Although most parts of the implementation of the Composite pattern can be generalized, they are not reused.

Instead of the above solution, we can represent the Composite pattern as a reusable collaboration. Expressed in terms of roles, the pattern consists

```
1  trait Figure {
2    var bgColor = "white"
3  }
4  trait BorderFigure extends Figure {
5    var borderColor = "black"
6  }
7  trait TextFigure extends Figure {
8    var text = ""
9  }
10
11 val f1 = new Figure{}; val f2 = new Figure{}
12 val f3 = new BorderFigure{}; val f4 = new TextFigure{}
13
14 val c = new Composite[Figure]{}
15
16 (f1 -: c.parent).addChild(f2 -: c.child)
17 (f1 -: c.parent).addChild(f3 -: c.child)
18 (f3 -: c.parent).addChild(f4 -: c.child)
19
20 (f1 -: c.parent).getChild(0)  // f2 -: c.child
21 (f4 -: c.child).getParent     // f3 -: c.parent
```

Listing 5.2: With the Composite collaboration, figures can be treated as members of a composite without containing the implementation of the pattern.

of a parent role and a child role. Listing 5.2 shows the example using the Composite collaboration from our library. Now, the figure classes (lines 1 to 9) do not contain Composite-related source code anymore. Instead, we instantiate the Composite collaboration and parametrize it with the desired type of core objects (line 14). Afterwards, the figures can be used as parent or child of a Composite by attaching the corresponding role using the `as` operator (lines 16 to 21).

If figures are often used in a Composite, we can shorten the source code with implicit conversions transforming figures into figures playing the parent or child role:

```
implicit def figure2parent(f: Figure) = f -: c.parent
```

Consequently, Composite-related members can be accessed without explicitly attaching the roles using `as`.

Another solution would be to generalize the pattern-related code into supertypes that are extended by the figure traits. While this may be a valuable solution in many cases, it does not provide the dynamic properties of roles. In contrast, we can attach Composite behavior to arbitrary objects at runtime in a role-based approach.

Applicability As a refinement to the situations for using the Composite pattern described in [GHJV95], the Composite collaboration can in particular be applied when:

- The concern of nesting objects should be separated from other concerns.

- You want to reuse an existing implementation of the pattern instead of writing it yourself.
- Only some instances of a class should be part of a Composite.
- It is only during a part of their lifetime that objects should belong to a Composite.

Our approach is not appropriate when core objects, for instance `Figure`'s, have to access their children for some purpose. The reason is that core objects are oblivious of playing a role, that is, their source code cannot refer to role members.

```
1  trait Composite[Core <: AnyRef] extends TransientCollaboration {
2    trait Parent extends Role[Core] {
3      private val children = new LinkedList[Child]()
4      def addChild(newChild: Child#Proxy) = { /*..*/ }
5      def getChild(i: Int): Child#Proxy = { /*..*/ }
6      def removeChild(oldChild: Child#Proxy) = { /*..*/ }
7    }
8
9    trait Child extends Role[Core] {
10     private[Composite] var parentR: Parent = null
11     def getParent: Parent#Proxy = { /*..*/ }
12   }
13
14   object parent extends RoleMapper[Core, Parent] {
15     def createRole = new Parent{}
16   }
17
18   object child extends RoleMapper[Core, Child] {
19     def createRole = new Child{}
20   }
21 }
```

Listing 5.3: Implementation of the Composite collaboration defining two role types `Parent` and `Child` and corresponding role mappers.

Pattern Implementation The pattern's implementation as a collaboration is given in Listing 5.3. The outer trait `Composite` represents the collaboration. It takes a type parameter `Core` that indicates the type of core objects. In the above example, we set `Core` to `Figure`. In lines 2 to 12, we define the role types `Parent` and `Child` including the fields and methods that are common to most implementations of the Composite pattern. Clients access the role using role mappers (lines 14 to 20). As explained in Section 3.2, role mappers manage the bindings between core objects and their associated roles by creating new role instances on demand. The types `Parent#Proxy` and `Child#Proxy` are type aliases for `Parent with Core` and `Child with Core`.

Note that for practical purposes, it is desirable to have multiple Composite collaborations implementing different flavors of the pattern. For instance, we could leave open the choice of having a parent link or vary the way children are stored and accessed.

Observer

> **Intent** Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Let us consider the example of a library software with a trait `Book`. It has a field `state` that can have the values `"available"` and `"borrowed"`. Furthermore, there is a trait `HelpDesk`, whose instances should be informed whenever the state of a book changes. Hence, books should play the role of a subject, such that help desks, playing the role of an observer, can react whenever their status changes.

```
1  trait Subject {
2    val observers = new HashSet[Observer]()
3
4    def addObserver(o: Observer) = observers += o
5    def removeObserver(o: Observer) = observers -= o
6    def notifyObservers = observers.foreach(_.update(this))
7  }
8
9  trait Observer {
10   def update(changedSubject: Subject)
11 }
12
13 trait Book extends Subject {
14   var status = "available"
15
16   def borrow = {
17     status = "borrowed"
18     notifyObservers
19   }
20   def returnIt = {
21     status = "available"
22     notifyObservers
23   }
24   def turnPage = { /*..*/ }
25 }
26
27 trait HelpDesk extends Observer {
28   override def update(changedSubject: Subject) = {
29     val b = changedSubject.asInstanceOf[Book]
30     b.status match {
31       case "available" => // ..
32       case "borrowed"  => // ..
33     }
34   }
35 }
```

Listing 5.4: Example of the Observer pattern without roles. Pattern-related parts are highlighted.

Listing 5.4 illustrates how one could implement this without roles. It shows two traits `Subject` and `Observer` (lines 1 to 11). Observers can be added to and removed from subjects. Whenever the observed state of a sub-

ject changes, it calls `notifyObservers`, which invokes the `update` method of all observers, passing itself as argument.

The traits `Book` and `HelpDesk` extend `Subject` and `Observer` (lines 13 to 35). To notify help desks about changes in observed books, `Book` calls `notifyObservers` in lines 18 and 22. Note that it is not called in line 24 since turning a page does not change a books status.

```
1  trait Book {
2    var status = "available"
3    def borrow = { status = "borrowed" }
4    def returnIt = { status = "available" }
5    def turnPage = { /* .. */ }
6  }
7
8  trait HelpDesk extends Observer[Book] {
9    def update(book: Book) = {
10     book.status match {
11       case "available" => // ..
12       case "borrowed"  => // ..
13     }
14   }
15 }
```

Listing 5.5: Traits of our library application. `Book` does not support observers.

The above solution may be satisfactory in many situations but has a major drawback: adding observers requires putative subjects to support it. The issue becomes obvious if we suppose that `Book` is implemented as in Listing 5.5. Using our role library, we can add behavior to arbitrary objects dynamically without changing their source code. To treat books as subjects although the `Book` trait does not support it, we attach the subject role to them.

```
1  val book = new Book{}
2  val helpDesk = new HelpDesk{}
3
4  val o = new ObserverCollab[Book]("status")
5
6  val observableBook = book as o.subject
7  observableBook.addObserver(helpDesk)
8
9  observableBook.borrow   // invokes call to HelpDesk.update
10 observableBook.returnIt // invokes call to HelpDesk.update
```

Listing 5.6: As the `Book` trait contains no subject-related members, we add this behavior at runtime using the `subject` role from the `Observer` collaboration.

Listing 5.6 shows an application making a non-observable book observable using a role. At first, we have to instantiate the Observer collaboration (line 4). It is parametrized with the core type of subjects, `Book`. In addition, we have to specify what should trigger a notification of observers of `Book`.

We do this using a member selector `"status"` (see Section 4.1 for details on member selectors). Hence, each time the variable `status` is set, observer will be informed.

Having configured and instantiated the observer collaboration, we can add the subject role to books using the `as` operator in line 6. The result is an `observableBook` that provides all methods of a subject. Thus, we can add a help desk as an observer in line 7. Consequently, calling the methods `borrow` and `returnIt` of the observable book invokes `HelpDesk.update`.

The observing trait, `HelpDesk`, must extend `Observer` giving the type of objects it observes (Listing 5.5 lines 9 to 14). Furthermore, it overrides the method `update`. As the behavior of an observer when receiving change notifications differs for each occurrence of the pattern, `update` is not defined in our collaboration. Instead we simply define the `Observer` trait to be extended in concrete applications.

Applicability The Observer collaboration can be applied when the Observer pattern does. It is particularly useful when:

- You want to reuse source code for observable objects.
- You want to observe objects, but they do not provide the proper members for being a subject and you cannot or want not change their source code.

```
1  class ObserverCollab[SubjectPlayer <: AnyRef](stateChangers: String*)
2      extends TransientCollaboration {
3
4    trait Subject extends Role[SubjectPlayer] {
5      private[this] val observers = new HashSet[Observer[SubjectPlayer]]()
6
7      def addObserver(o: Observer[SubjectPlayer]) = observers += o
8      def removeObserver(o: Observer[SubjectPlayer]) = observers -= o
9      def notifyObservers() = observers.foreach(_.update(core))
10
11     addAfterCalls(notifyObservers, stateChangers:_*)
12   }
13
14   object subject extends RoleMapper[SubjectPlayer, Subject] {
15     def createRole = new Subject{}
16   }
17 }
18
19 trait Observer[SubjectPlayer] {
20   def update(s: SubjectPlayer)
21 }
```

Listing 5.7: Implementation of the Observer collaboration. It defines the `Subject` role and a corresponding role mapper.

Pattern Implementation Listing 5.7 shows the implementation of the Observer collaboration. We define a role type for subjects containing the methods for adding, removing, and notifying observers (line 4 to 12). In con-

trast, observers are not realized as roles, but every object extending the trait `Observer` (line 19 to 21) can act as one.

Another noteworthy detail is the constructor argument `stateChangers` (line 1). It contains the member selectors that trigger the notification of observers. We invoke `addAfterCalls` in line 11 to register a call to `notifyObservers`. Consequently, `notifyObservers` is always executed after invoking one of the members described by `stateChangers`. The invocation handler will look up such calls and invoke the methods accordingly.

There is an interesting similarity to aspect-oriented programming. Our approach to make the Observer pattern reusable is akin to the aspect-oriented implementation proposed in [HK02]. Roughly, the member selector `stateChangers` corresponds to a pointcut; the code of `notifyObservers` is equivalent to an after advice.

Another solution for a reusable implementation of the Observer pattern in Scala is presented in [OZ05b]. It enhances the solution based on simple subclassing from Listing 5.4 by restricting the type of observers to a concrete subclass of `Observer`, in our example `HelpDesk`. In Listing 5.4, in contrast, instances of all subclasses can be added via `addObserver`. This is realized using explicit self-types and abstract type members. In contrast to our solution, it statically composes the subject behavior into a class, whereas we add the behavior of a subject dynamically to objects.

Case Study: SCells We applied the Observer collaboration to SCells, a minimalistic spreadsheet application written in Scala [OSV08]. It uses the observer mechanism for multiple purposes, among others, to update cells that depend on other cells. We want to implement this update mechanism using the Observer collaboration.

As the concern of updating cells is spread over multiple classes, we provide the collaboration as a Scala object so that it can be imported where required:

```
object CellObserver extends ObserverCollab[Model#Cell]("v") {}
```

Its type parameter, `Model#Cell`, gives the type we want to observe. The constructor argument, `"v"`, specifies that we are only interested in changes to the field `v` (which means *value* and holds the current value of a cell in the spreadsheet). To add and remove observers, as well as for setting `v`, we expand cells to subjects, for example:

```
(cell -: subject).v = newValue
```

Otherwise, cells do not contain subject-related members, such that the concern of observing cells is only visible when needed.

The original version of SCells presented in [OSV08] leaves cyclic dependencies between cells as an open issue. In particular, formulas that refer to each other in a cycle lead to a stack overflow during their evaluation. We solve the problem by checking for cycles in the Observer collaboration. It provides a field `cycleCheck` for enabling that feature. If enabled, the collaboration accumulates the updated core objects until the method `cycleReset` is called. When a core object occurs twice, the updating is interrupted. Implementing the cycle checking was straightforward, because the Observer code is located in one place which makes it easy to extend.

5.2 Enhancing Patterns with Roles

The patterns we describe above contain common code fragments that reoccur in most of their use cases. Unfortunately, this is not the case for all design patterns, such that the dream of a library with reusable pattern collaborations is only partially realizable. However, it turns out that a role-based approach is still beneficial for some patterns. In the following, we present a set of design patterns that we propose to implement or enhance with roles and collaborations.

Mediator

> **Intent** Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

This pattern does not contain large fragments of reusable code, and hence, cannot be part of a general pattern library. However, mediators and collaborations have similar goals. A mediator encapsulates the interactions between several objects. Collaborations do the same by assigning roles to the participants.

We argue that the mediator pattern can be replaced by a role-based approach. Let us consider an example derived from the one given in the Gang-of-Four book. There is a dialog to select fonts from a list and enable or disable properties of a font, such as *bold*. We consider a minimalistic version containing three widgets: a box of type `ListBox` listing all available fonts which the user can choose from by clicking on one of them, an editable field of type `EntryField` that allows to give a font name directly, and a button of type `RadioButton` to choose between bold and normal fonts.

Listing 5.8 shows an implementation using the Mediator pattern. The widgets interact, in that whenever the user chooses a font from the list, its name appears in the field. Other interactions, such as updating the field when the bold property is chosen, are omitted here. The Mediator pattern proposes to realize this with a `DialogDirector` that has a reference to each widget and that is also referenced from all widgets. Whenever a widget changes, it calls the `changed` method defined in line 6 that informs the director. A concrete subclass of it, like `FontDialogDirector` (line 27 to 39), reacts on the changes in a method `widgetChanged`.

The source code that is related to the interaction between the widgets is highlighted in Listing 5.8. While most of the implementation is located in `FontDialogDirector`, references to the director and calls to the `changed` method are scattered over the classes. We can avoid scattering and the bidirectional coupling between the widgets and the director by expressing the interaction as a collaboration. The strong coupling is obviated since core types require no reference to the collaboration or any role. That is, we replace active widgets that initiate a reaction on changes by passive ones that are observed using roles.

Listing 5.9 shows the `FontDialogCollaboration` which replaces all highlighted code from Listing 5.8, including the scattered references to the director. Consequently, the interactions between our widgets are completely described in one place. The collaboration contains one role that is bound to the list box. It adds functionality for adapting the text field each time the

```
1  abstract class DialogDirector {
2    def widgetChanged(w: Widget)
3  }
4
5  abstract class Widget(val d: DialogDirector) {
6    def changed() = d.widgetChanged(this)
7    def handleMouse(e: MouseEvent)
8  }
9
10 class ListBox(override val d: DialogDirector) extends Widget(d){
11   var items = List[String]()
12   var selected = -1
13   def getSelection: String = items(selected)
14   override def handleMouse(e: MouseEvent) = { /* .. */ changed() }
15 }
16
17 class EntryField(override val d: DialogDirector) extends Widget(d) {
18   var text: String = ""
19   def handleMouse(e: MouseEvent) = { /* .. */ }
20 }
21
22 class RadioButton(override val d: DialogDirector) extends Widget(d) {
23   var text: String = "bold"
24   def handleMouse(e: MouseEvent) = { /* .. */ }
25 }
26
27 class FontDialogDirector extends DialogDirector {
28   val box = new ListBox(this)
29   val field = new EntryField(this)
30   val button = new RadioButton(this)
31
32   def widgetChanged(w: Widget) = {
33     if (w == box) {
34       field.text = box.items(box.selected)
35     } else if (w == button) {
36       // ...
37     }
38   }
39 }
```

Listing 5.8: A mediator that encapsulates the interaction of three widgets. The highlighted parts concern the interaction.

`selected` variable is set (line 10). We could similarly adapt the behavior of other widgets by attaching roles to them.

To use the widgets with their roles, they must be referenced via an instance of `FontDialogDirector`. For example, when adding the list box to a window, one has to use `collab.box`, where `collab` is an instance of `FontDialogDirector`.

```
1  class FontDialogCollab(boxCore: ListBox, field: EntryField,
2                         button: RadioButton)
3      extends StickyCollaboration {
4    val boxRole = new BoxRole{}
5    boxRole.bind(boxCore)
6    def box = boxRole.proxy
7
8    trait BoxRole extends Role[ListBox] {
9      addAfterCalls(() =>
10       { field.text = box.items(box.selected) }, "selected")
11   }
12 }
```

Listing 5.9: A collaboration encapsulating the highlighted code from Listing 5.8.

Decorator

Intent Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

The intent of the Decorator pattern is very similar to that of roles. Both techniques aim at enhancing the capabilities of an object dynamically by adding new members. We argue that the concept of roles subsumes decorators. To illustrate this, let us consider the example from [GHJV95].

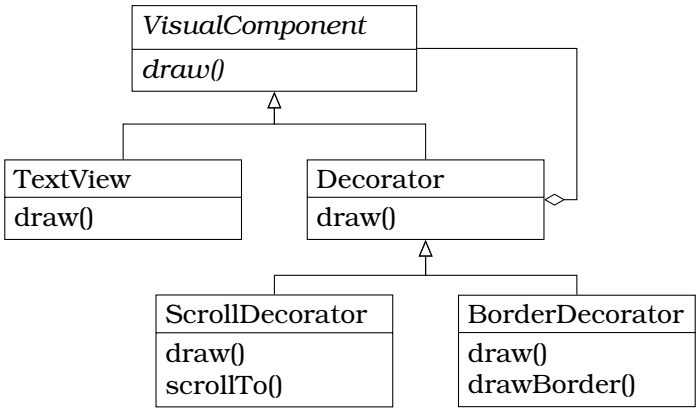


Figure 5.1: The example for the Decorator pattern given in [GHJV95].

Figure 5.1 shows an abstract class `VisualComponent` that can be decorated using a `ScrollDecorator` and a `BorderDecorator`. Visual components have a `draw` method which decorators override. In addition, they add new members, for instance, the method `drawBorder`.

Instead of adding decorators to objects, we propose to attach roles. Transforming the above example into a role-based style, we express the border decorator as a role of text views (Listing 5.10). Similarly, we can define a role that corresponds to the `ScrollDecorator`.

Assuming we want to build an application that enhances a text view with a scroll bar and a border, we simply add the corresponding roles to the text

```
1  trait BorderCollab extends TransientCollaboration {
2    val bordered = new Bordered{}
3    trait Bordered extends Role[TextView] {
4      def draw = {
5        core.draw
6        drawBorder
7      }
8      def drawBorder = { /*..*/ }
9    }
10 }
```

Listing 5.10: A role for `TextView` objects that adds a border.

view (Listing 5.11). Consequently, the text view plays two roles and calling `draw` in line 5 invokes the added behavior as well as the draw method of the text view.

```
1 val text = new TextView{}
2 val bc = new BorderCollab{}
3 val sc = new ScrollCollab{}
4 val scrollableBorderedText = (text -: sc.scrollable) -: bc.bordered
5 scrollableBorderedText.draw
```

Listing 5.11: Enhancing a text view with two roles to add a scroll bar and a border.

Roles provide a number of benefits over decorators. First, the fact that behavior is distributed over multiple objects is well hidden from the user. Second, one can transfer a role from one object to another while preserving its state. Third, decorators are hard to use for objects of different classes in a large hierarchy. When the decorator extends the top-level class, decorated objects do not contain features from subclasses. The other option is to provide a decorator for each subclass whose instances should be decorated, which results in redundant decorator classes. In contrast, the `as` operator considers the dynamic type of a core object, such that one decorator is sufficient for a complete class hierarchy.

The close relation of roles and decorators has also been discussed by others. In [BRSW00], the authors describe the Role Object pattern as a combination of Decorator and the Product Trader pattern [BR97]. The authors of [KO96] examine decorators as an implementation mechanism for roles, however, emphasizing the drawbacks of such a solution.

Template Method

> **Intent** Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The pattern is mainly a structural description of the interplay between abstract and concrete methods as it can be found in almost every object-oriented software. Since there are no code fragments occurring again and

again in the pattern's use cases, we cannot generalize it into a reusable collaboration.

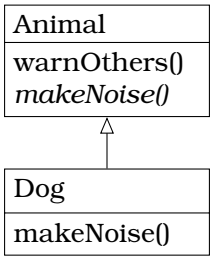


Figure 5.2: Two simple classes realizing the template pattern. The method `warnOthers` relies on the implementation of `makeNoise` in a subclass.

However, it is an interesting idea to change the behavior of concrete methods for certain objects using roles. Suppose we have a class structure as depicted in Figure 5.2. Animals warn other animals by running around and making noise. The noise they make is different for each species, and thus, deferred to subclasses, such as `Dog`. In some cases, programmers might want to adapt the behavior of an instance of such a subclass, for example, to express that a particular dog is too shy for making lots of noise. We can do this by attaching a role that overrides the `makeNoise` method:

```
myDog as c.shyDog
```

One should note that this only works because we support true delegation (see Section 4.1). Figure 5.3 shows the invocation protocol of a dog playing the role `shyDog`. A client calls `warnOthers` on the proxy which delegates to `myDog`. Its implementation invokes `this.makeNoise`, which is called on the proxy since `this` refers to it. Hence, the proxy can delegate `makeNoise` to `shyDog`. If, in contrast, we would use forwarding, the call to `this.makeNoise` would be on `myDog`, ignoring that the role overrides it.

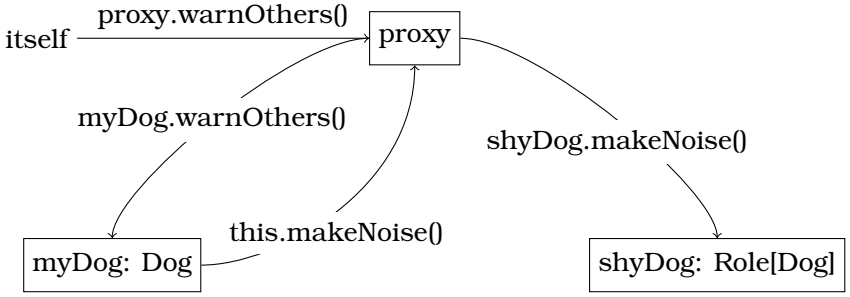


Figure 5.3: A role can override methods of a subclass that are called in a superclass because `this` always refers to the proxy.

Proxy

Intent Provide a surrogate or placeholder for another object to control access to it.

The Proxy pattern describes how one object can hide and represent another, for example, because the latter is remote, should only be accessed with certain permissions, or is lazily created. Instead of instantiating a normal object, one creates a proxy, possibly passing information on how to create or access the hidden object, or even the object itself.

Our role approach allows to modify or extend the behavior of members of a core object. Hence, we can achieve similar goals as the Proxy pattern, however, in a more dynamic way. In particular, the decision whether to use the real object or a proxy has not to be taken when the object is created. Instead, the proxy behavior, encapsulated as a role, can be attached and removed as required.

Let us sketch an example for a remote proxy that hides an object of type `Data`, which has a method `getValue`. Listing 5.12 shows a role for `Data`. It modifies the implementation of `getValue` by accessing a remote version of its core object. The details of the remote access are omitted as they depend on the distribution mechanism, which is not relevant here.

```
1 trait RemoteCollab extends TransientCollaboration {
2   val remote = new Remote{}
3   trait Remote extends Role[Data] {
4     def getValue = /* retrieve remote version of core object and call getValue */
5   }
6 }
```

Listing 5.12: The `remote` role accesses its core object remotely and returns the result of `getValue`.

With such a role, programmers can choose dynamically whether an instance of `Data` should be accessed directly or remotely. For example, one can attach the `remote` role when being online and otherwise use the `Data` object containing cached state directly (Listing 5.13).

```
1 val c = new RemoteCollab{}
2 import c._
3
4 val value = if (offline) myData.getValue
5             else (myData as remote).getValue
```

Listing 5.13: Depending on whether we are online, we attach the `remote` role or use `myData` directly.

Dynamic proxies can also be used to make arbitrary objects remotely accessible. This approach has already been studied, for instance in [BCH02], and is out of the scope of his thesis.

Role Object

Intent Adapt an object to different client's needs through transparently attached role objects, each one representing a role the object has to play in that client's context. The object manages its role set dynamically. By representing roles as individual objects,

different contexts are kept separate and system configuration is simplified.

The Role Object pattern [BRSW00] proposes an implementation technique for representing roles in mainstream programming languages such as C++ or Java. Hence, it is applicable in real world applications without requiring particular tools [BGK+97].

We argue that our approach can replace the pattern, as it also goes without language extensions, while providing further benefits, such as a solution to the self problem and type-safe access to role-playing objects. In the following, we will compare an example based on the Role Object pattern with our approach. For a detailed description of the pattern and a more general comparison with our work, see Section 6.5.

The motivating example in [BRSW00] is a bank application, where a customer can occur as a borrower to one department and as an investor to another. There is a `Customer` class with two subclasses `CustomerCore` and `CustomerRole`. The latter is further refined by `Borrower` and `Investor`.

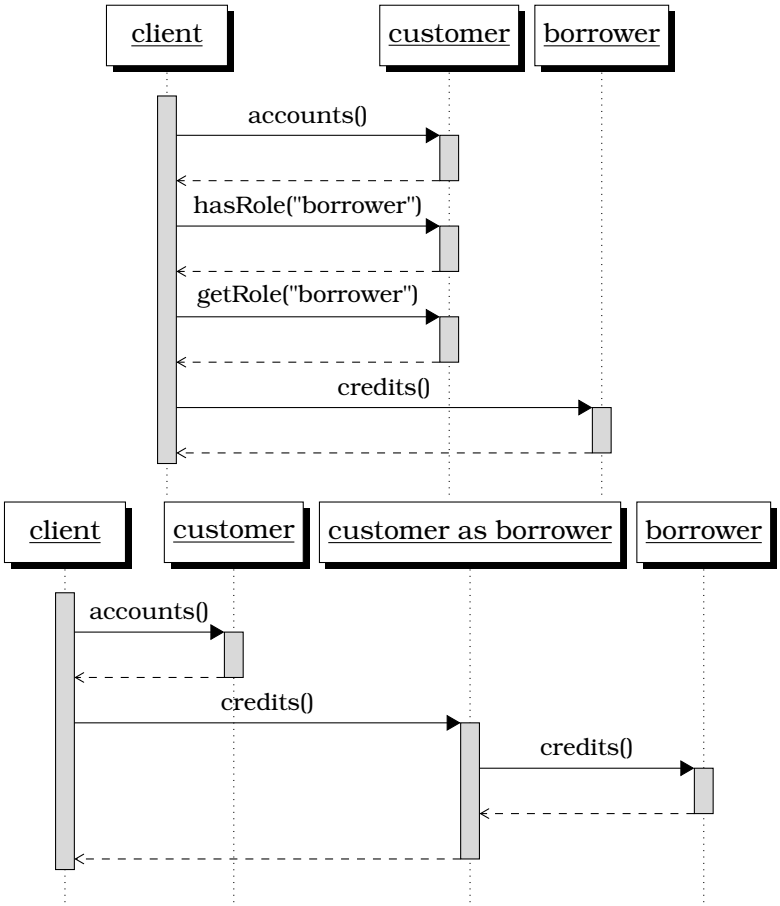


Figure 5.4: Comparison of the protocol of the Role Object pattern (above) and that of Scala Roles (below).

Figure 5.4 (upper part) shows a typical sequence of method calls using the Role Object pattern. Core functionality, like the `accounts` method, is

accessed by calling the core object, `customer`. Before invoking role members, clients must check whether the object provides the required role using `hasRole` and get the role object with `getRole`. Afterwards, one can call role methods, like `credits` on it.

The lower part of the figure depicts the equivalent sequence diagram using our approach. Role behavior can be directly invoked on `customer as borrower`, without the need to check if the role is available. Also, clients have no direct access to the role object, `borrower`. Instead, the role-playing object delegates the call of `credits` to it.

5.3 Obsolete in Scala

We classify 6 out of 24 patterns as obsolete in Scala. This is either because they are part of the language, like Singleton, or because the goals of the pattern can be achieved in a much simpler way, as for Visitor. From a practical viewpoint this means that there is no motivation to solve the problem addressed by the pattern with a role library in Scala. Instead, programmers should consider the solutions presented in the following.

Visitor

> **Intent** Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

The visitor pattern is one of the most widely used design patterns and also one of the most complex ones. It describes two class hierarchies, one for data structures and one for operations to be performed on elements of the data structure. The main benefit of the pattern is that new operations can be added without changing the data structure hierarchy using a double-dispatch mechanism. There are numerous proposals for enhancing the original pattern description [PJ98, Gro03, OZ05a].

```
1  abstract class CompanyElement
2  class Company extends CompanyElement {
3    val departments = new HashSet[Department]()
4  }
5  abstract class SubUnit extends CompanyElement
6  class Department(var manager: Manager) extends SubUnit {
7    val subunits = new HashSet[SubUnit]()
8  }
9  class Employee(var salary: Int) extends SubUnit {}
10 class Manager(var man_salary: Int) extends Employee(man_salary)
```

Listing 5.14: A recursive class hierarchy describing the organization of a company.

In Scala, the goals of the Visitor pattern can be achieved in a much simpler way using pattern matching, a construct originating in functional programming. To illustrate the idea, let us consider an example similar to the one presented in [LJ03]: Listing 5.14 contains classes describing the organizational structure of a company. A company consists of multiple departments,

each having a manager and a number of subunits. A subunit is either another department or an employee. Both managers and ordinary employees are persons and receive a salary.

```
1 def augmentSalaries(el: CompanyElement): Unit = el match {
2   case c: Company     => c.departments.foreach(augmentSalaries)
3   case d: Department  => augmentSalaries(d.manager)
4                          d.subunits.foreach(augmentSalaries)
5   case m: Manager     => m.salary += 10
6   case e: Employee    => e.salary += 20
7   case _              =>
8 }
```

Listing 5.15: A simple replacement for visitors: functions descending a data structure recursively with pattern matching, for instance, to augment the salary of all employees in a company.

We can apply an operation on instances of such a data structure using pattern matching in a recursive function. Pattern matching allows for dividing the control flow according to the dynamic type of an object. We can use it to recursively descend our data structure and apply an operation on the way. For example, Listing 5.15 shows a function that augments the salary of all employees and managers of a company.

The described approach fulfills the two main goals of the Visitor pattern. First, the data structure hierarchy and operations performed on it are separated. Second, we can easily add new operations without adapting (or recompiling) the data structure classes. Another benefit of the Visitor pattern is that related operations can be put in one visitor, while others go into another. In Scala, this can equally well be achieved by grouping functions into Scala objects.

Interpreter

> **Intent** Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

This pattern proposes to model a language with a hierarchical data structure of classes, each representing an expression (either a terminal or a non-terminal expression). The operation of interpreting the language is defined by a method `interpret` in each class, that, in case of non-terminal expressions, calls the `interpret` methods of its descendants.

In fact, the pattern solves a similar problem as the Visitor pattern. This time, the authors propose a different solution, though, assuming that adding new ways of interpreting the language, that is, adding new operations, only rarely occurs.

We argue that Scala again provides an easier solution based on recursive functions and pattern matching. Its basic idea is the same as explained above for the Visitor pattern. A function recursively iterates over sentences of the language, identifying different types of expressions with pattern matching. A simple example illustrating this approach can be found in [Oa06].

Strategy

> **Intent** Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

This pattern provides an alternative to subclassing for varying the implementation of one or more methods. The basic idea is to encapsulate each variant of a method into an object. Variability is achieved by exchanging one such object for another.

In a language supporting higher order functions like Scala, we can solve the problem much easier. Since functions are values, we can directly exchange them without wrapping them into an object. In contrast, creating strategy classes can make sense when the algorithm consists of multiple methods that should vary together.

Command

> **Intent** Encapsulate a request as an object, thereby letting you parametrize clients with different requests, queue or log requests, and support undoable operations.

The Command pattern, basically, describes two things:

- How to represent commands as objects using a Command class.
- How to combine commands with their invokers and receivers, that is, objects calling the command and objects affected by executing it.

Similarly to the Strategy pattern, Command is about encapsulating functions into objects in order to pass them as arguments. If a command provides only one method `execute`, we can do the same trick as above and directly pass a function. Consequently, an abstract command class is not required anymore and the implementation becomes simpler. This solution cannot be applied when commands contain other methods, though, for example, to undo them. In this case, one usually requires a surrounding object to store state, like information about the receiver before executing the command.

Adapter

> **Intent** Convert an interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Adapters can be realized in two ways. Both propose a class `Adapter` that inherits from the target class. The first variant, called *object adapter*, wraps the object that should be adapted, the *adaptee*. An object adapter overrides methods of the target class by forwarding calls to the adaptee. In the second variant, *class adapter*, the adapter class also inherits from the adaptee class (in addition to the target class). It overrides methods from the target class by simply calling the corresponding method in the adaptee class.

In Scala, adapters can be replaced by implicit conversions. As explained in Section 2.1, an implicit conversion is a method transforming an object of

one type into an object of another type, for instance, through wrapping. The conversion is automatically inserted by the compiler whenever a type error would otherwise occur. As a result, programmers do not have to adapt objects explicitly anymore, but simply provide a conversion method and make it available in the required scope.

```
1  class Target { def needToProvide }
2
3  class Adaptee { def wantToCall = { /* .. */ } }
4
5  implicit def adaptee2Target(adaptee: Adaptee) = new Target {
6      def needToProvide = adaptee.wantToCall
7    }
8
9  def main(args: Array[String]) = {
10   val a = new Adaptee
11   a.needToProvide  // calls Adaptee.wantToCall
12 }
```

Listing 5.16: The `Adaptee` class is adapted to the `Target` class using an implicit conversion.

A simply example illustrating the idea is giving in Listing 5.16. The class `Adaptee` has to be adapted to fit the interface of the class `Target`. This is achieved with an implicit conversion `adaptee2Target` wrapping an `Adaptee` object into an instance of a subclass of `Target` (line 5). It implements the method `needToProvide` by forwarding to the corresponding method `wantToCall` of the adaptee. As a result, we can use instances of `Adaptee` as if they were `Target`s (line 11).

Singleton

> **Intent** Ensure that a class has only one instance, and provide a global point of access to it.

In Scala, the Singleton pattern is part of the language in form of Scala objects (see Section 2.1), and hence, there is no need to use the pattern anymore.

5.4 Invariant Patterns

A number of patterns turn out to be invariant towards our approach. Most of them describe how to assemble methods and classes to achieve a specific task, but neither contain reusable pieces of source code nor can beneficially be described with roles.

State

> **Intent** Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

The intent of the state pattern seems to be quite related to the idea of roles. Hence, it is an an obvious idea to replace the pattern with a role-based approach. An object having a certain state would correspond to an object playing a certain role.

According to the pattern description, an object is responsible for changing its own state, for instance, during specific method calls. In terms of roles, this implies that an object can access and manipulate its own roles. This is not the case using the approach presented in this work, though. In contrast, roles are attached from the outside to objects. As a (unfortunate) result, this means that we cannot replace the State pattern. A language allowing for dynamic reclassification, hence, making the State pattern obsolete is presented in [DDDCG01].

Other Patterns

The Iterator pattern is part of the collection library of most modern languages, and hence, not relevant for our discussion. Similarly, the Prototype pattern is also included in most languages, for example, in form of the `cloneable` annotation and the `clone` method in Scala. For some other patterns, such as Chain of Responsibility, we found a way to express them with roles, however, making its application more complex than the original pattern version.

This finishes our analysis of design patterns. Our results are diverse, such that a general statement claiming patterns to be expressible with roles seems oversimplifying. However, at least for some patterns using roles yields benefits. Our work can be continued by analyzing more patterns that go beyond the Gang-of-Four book. Also, it seems interesting to compare to other role-based implementations, for example, in ObjectTeams/Java.

6 Related Work and Comparison

There exist a number of interesting proposals for implementing roles and collaboration-based designs. We briefly review them in this chapter and compare them with our approach using nine criteria. In contrast to Steimann's criteria that capture important features of roles in modeling, we focus on questions that emerge when implementing roles in a programming language (see Section 4.3 for an evaluatuion of our approach using Steimann's criteria). Our criteria are the following:

1. *Language compatibility*: Is the approach compatible with its underlying programming language or does it involve an extension of it?
2. *Dynamism*: Can roles be attached to and removed from objects at runtime?
3. *Type safety*: Is accessing role members type checked at compile time? Are any casts necessary?
4. *Views*: Are role members always visible or do roles provide views on an object that are only available when needed?
5. *Collaborations*: Is there a notion of collaborations to group related roles?
6. *Distinguishing roles of same type*: If there are multiple roles of the same role type but from different collaborations, can they be (statically) distinguished?
7. *Independent development*: Can roles be developed independently from their core classes?
8. *Self problem*: Is the self problem solved? (see Section 3.2)
9. *Identity problem*: Is the identity problem solved? (see Section 3.2)

At first, we analyze three approaches involving special-purpose programming languages: Epsilon, ObjectTeams, and aspects-oriented programming. Afterwards, two language-independent solutions are considered: mixin layers and the Role Object pattern.

6.1 Epsilon and EpsilonJ

Epsilon [TUI07] is a role-based model for describing collaboration fields where sets of roles interact to achieve collaboration. EpsilonJ is a Java-based implementation of Epsilon. The basic elements of the Epsilon model are *collaboration fields* and *roles*. A collaboration field describes the environment in which roles collaborate. It provides a unit of concern and reuse.

Roles can be seen as objects, having members and exchanging messages inside an environment. However, roles are not accessed directly, but via their surrounding collaboration field.

Roles and objects are composed by binding a role instance to an object. Consequently, the object acquires the members of the role. Binding roles and removing them happens dynamically at runtime, such that an object can play different roles during its lifetime. Also, one object may play multiple roles from different collaboration fields.

The dynamic binding of roles to objects distinguishes the Epsilon approach from earlier works such as the OOram method and mixin layers. Both approaches compose roles and classes statically on the level of types. In contrast, Epsilon proposes roles as first-class citizens at runtime.

We will shortly explain the most important language features using examples from [TUI07]. A context containing roles can be defined as follows (newly introduced keywords with respect to Java are highlighted):

```
context Company {
  role Employer {
    int salary = 100;
    void pay() { Employee.getPaid(salary); }
  }
  role Employee {
    int save;
    void getPaid(int salary) { save += salary; }
  }
}
```

To bind roles to objects and access them, an instance of the surrounding context is required (`Person` is a normal Java class left out here):

```
Context c = new Company();
Person paul = new Person();
Person jim = new Person();
c.Employer.bind(paul);
c.Employee.bind(jim);
(c.Employer)paul.pay();
```

As indicated in the last line, one must cast a role-playing object to access it. According to the authors, this is required to distinguish multiple role instances that may be bound to one object. Note that the cast may fail, since it is only dynamically checked if an object actually plays a certain role.

Roles may require all objects that want to play them to provide a certain interface. For instance, employee could require a method to deposit money at a bank:

```
role Employee requires { void deposit(int); } {
  int save;
  void getPaid(int salary) { save += salary; }
}
```

In case an object provides the required method but uses a different name, the `replacing` keyword can map one method to another. Similarly, it can be used to let a role override methods of an object if the methods do not have the same name.

Comparison

1. *Language compatibility*: No. EpsilonJ is an extension of Java.

2. *Dynamism*: Yes. Role can be attached with `bind` and removed with `unbind` at runtime.

3. *Type safety*: No. To access role members, programmers must cast the core object to the required role type. A method `boundObject` is defined for each role instance to manually check the binding between a role and a core. It returns the current core object, or null if there is none.

4. *Views*: Yes. Role members are only visible when the core object has been casted to the corresponding role type.

5. *Collaborations*: Yes. Our notion of collaborations corresponds to EpsilonJ's collaboration fields.

6. *Distinguishing roles of same type*: Partially. Multiple roles played by one core object are distinguished by casts using the surrounding collaboration field of the role as its qualifier.

7. *Independent development*: Yes. Collaboration fields and roles can be defined without referring to concrete core classes.

8. *Self problem*: Unfortunately, the authors do not consider this question.

9. *Identity problem*: Same as with 8.

6.2 ObjectTeams

A recent and very inspiring realization of the role concept in a programming language is the ObjectTeams project[1] [Her07]. It mainly consists of the Java-based programming language ObjectTeams/Java [HHM07] and an extension of the Eclipse platform[2] which embeds the language into the IDE. The terminology used in ObjectTeams differs slightly from ours: collaborations are called *teams*, the type of a role is defined in a *role class*, and an object that plays a role is a *base object* defined by a *base class*.

ObjectTeams/Java enhances Java with two basic language constructs, namely *roles* and *teams*. Teams are particular classes, *team classes*, whose inner classes are considered role classes. Roles are accessed via instances of team classes. Each team instance represents the context for a collaboration of objects. Teams must be instantiated and activated. *Team activation* can be done explicitly or implicitly by calling a role method on an object.

Role classes and base classes can be combined through a `playedBy` declaration. `R playedBy B` specifies that each role instance of `R` is associated to a base instance of `B`. Furthermore, role classes and base classes can be related with *callout* and *callin* bindings. The terms *in* and *out* are both chosen from the perspective of the role. A callout binding declares that a method call to a role object should be forwarded to a method of the associated base object. In contrast, callin bindings specify that calls to a base object are intercepted and forwarded to a role object. Callin bindings are similar to weaving of additional source code into existing base methods. The modifiers `before`, `replace`, and `after` declare how the original method and the callin method of the role object should be composed.

[1] `http://www.objectteams.org/`
[2] `http://www.eclipse.org`

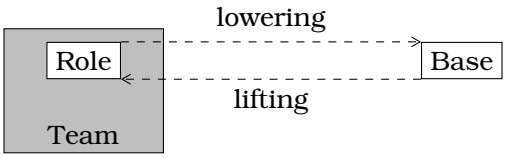


Figure 6.1: Role instances and base instances are transformed into each other through lifting and lowering operations.

Since roles are objects in ObjectTeams/Java, the problem that roles do not directly conform to the type of their base classes needs to be solved. Therefore, the usual subtype polymorphism is accompanied by *translation polymorphism*, an implicit type-safe conversion between role instances and their base instances. Navigating from a base instance to a role instance is called *lifting*, the inverse is referred to as *lowering* (see Figure 6.1).

Lifting can be thought of as a function:

$$base\,instance \times role\,type \times team\,instance \rightarrow role\,instance$$

That is, given a base instance, the expected role type, and the surrounding context, lifting provides the appropriate role instance. The lifting translation considers the dynamic type of a base instance and returns a role instance of the most specific role type. The ObjectTeams/Java compiler generates a lift method for each role type that is related to a base class via `playedBy`. This is sufficient since all role bindings must be explicitly given by the programmer using `playedBy`. Lowering can be implemented in a straight forward way, because each role instance has a *parent* reference to its base instance. For more details on the implementation of lifting see [HHM04].

The ObjectTeams/Java approach can be used in multiple ways, one of them being team components that encapsulate the contained roles completely. Hence, roles are only indirectly accessed via methods of their team instance. Listing 6.1 provides an example of such a team, describing a university with students and professors. A team is a particular class labeled with the keyword `team` (line 1). Its inner classes `Student` and `Professor` are role types (lines 6 to 12). The keyword `playedBy` indicates the required type of base classes for a role. Both roles are exclusively accessed via methods of the university team, such as `enroll` (line 14). The `as` in its parameter type, `Person as Professor`, is an explicit lifting from an instance of `Person` to a role instance of type `Student`. As a result, the parameter `stud` can be accessed as `Student` inside the method. In line 25, we see an example for the opposite operation; the method performs an implicit lowering when returning an instance of `Student`, because the actual return type is specified as `Person`.

Comparison

1. *Language compatibility*: No. ObjectTeams/Java is an extension of the Java programming language and requires its own compiler.

2. *Dynamism*: Yes. A core object can be lifted to different roles and lowered afterwards.

```
1  public team class University {
2    private HashMap<Integer, Student> studentIds
3                = new HashMap<Integer, Student>();
4    private int maxId = 0;
5
6    protected class Student playedBy Person {
7      protected Professor supervisor;
8    }
9
10   protected class Professor playedBy Person {
11     protected Set<Student> students = new HashSet<Student>();
12   }
13
14   public int enroll(Person as Student stud) {
15     studentIds.put(++maxId, stud);
16     return maxId;
17   }
18
19   public void supervise(Person as Professor prof,
20                         Person as Student stud) {
21     stud.supervisor = prof;
22     prof.students.add(stud);
23   }
24
25   public Person getStudent(int id) {
26     return studentIds.get(id);
27   }
28 }
```

Listing 6.1: A team with two roles `Student` and `Professor` that are exclusively accessed via methods of their surrounding team instance.

3. *Type safety*: Yes. Type-safe access to role members is guaranteed due to translation polymorphism. For most use cases, type casts are not necessary.

4. *Views*: Yes. Role members are made available via lifting, and thus, are not always visible.

5. *Collaborations*: Yes. Collaborations are called teams.

6. *Distinguishing roles of same type*: Yes. Role instances from different collaborations have different types.

7. *Independent development*: Yes. Roles can be developed independently. However, programmers must establish a binding with base classes using `playedBy`. For instance, this can be done by creating a concrete subclass of an abstract team class.

8. *Self problem*: Yes. The problem is solved by overwriting the *self* variable of roles with the originally receiver of a method.

9. *Identity problem*: Partially. The problem is not solved, that is, comparing an object and the same object playing a role will result in `false`. However, the authors argue that direct comparisons are not required

in common use cases due to the automatic translation by lifting and lowering.

6.3 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [KLM+97] is a programming paradigm for specifying *cross-cutting concerns* to enhance modularization. Most aspect languages share the concepts of *advices*, additional behavior added to existing methods, *join points*, points in the control-flow of a program where aspects and the main program meet, and *pointcuts*, specifications of sets of join points. An *aspect* is the combination of pointcuts and advices that cover one concern of an application.

Obviously, AOP and role-based programming have very similar objectives. Both search for complements to the dominant decomposition mechanism of classes and try to modularize cross-cutting concerns. Hanenberg and Unland [HU02] conceptually analyze the relation between both approaches, concluding that aspects offer most features of roles, whereas the inverse does not hold.

In [Ken99], the authors investigate implementations of role-based designs with AOP, distinguishing two kinds of aspects. On the one hand, *introduce weavings* permit to statically add new members to classes, and hence, add new functionality. On the other hand, *advise weavings* change existing methods by prepending or appending behavior, or in aspect lingo, advise the method. The latter may not only be applied to classes but also at runtime to instances, in this manner, adding functionality to objects dynamically. Based on these mechanisms, two options for implementing roles are proposed: a hybrid approach using introduce and advise weavings to represent roles, and a solution combining classes and role types via glue aspects.

The hybrid approach consists of two parts. First, introduce weavings statically enhance the interface of classes with role-specific methods. Second, advise weavings dynamically provide role-playing objects with an implementation of the role methods whenever an object is required to play a role. The main advantage of the hybrid approach is that role type source code is separated from core class source code. One drawback, though, is missing static type safety for objects that currently do not play a certain role. Their interface always contains all role methods it could have. When such a method is called, one can either do nothing or, as proposed in [Ken99], throw a runtime exception.

Another proposal in [Ken99] are glue aspects. Here, role types are implemented as classes that are kept separate from their putative core classes. At runtime, the state and behavior of a role is captured by an object. To enhance a core class with role members, introduce weavings statically add methods that forward to methods in the role object. Furthermore, a general role aspect extends core classes with methods for dynamically binding roles, similar to the mechanism of the Role Object pattern (see Section 6.5). In contrast to the hybrid approach, role types need no knowledge of their putative core classes, and consequently, can be developed independently. On the downside, the approach requires three levels of components (core classes, role classes and glue aspects) and suffers from the same type safety problem as the hybrid approach due to static aspect weaving.

Hannemann and Kiczales propose to implement design patterns with aspects [HK02]. Similar to our approach, patterns are assumed to assign

roles to their participants. The authors find pattern implementations that are reusable and provide better code locality, since pattern-related code is encapsulated into an aspect.

Comparison

We forgo a comparison of roles and aspects in general, and instead, compare our approach with the aspect-oriented implementations of role-based designs proposed in [Ken99].

1. *Language compatibility*: No. AOP requires an extension of the underlying programming language such as AspectJ for Java.[3]
2. *Dynamism*: Yes. Roles can be added and removed at runtime using dynamic advise weavings.
3. *Type safety*: Partially. Role members are statically added using introduce weavings, such that accessing a member that does neither belong to the core nor to a role leads to a compile error. However, objects provide role members even if they do not play the corresponding role.
4. *Views*: No. Role members are always visible for all instances of a core class, since they are statically added via introduce weavings.
5. *Collaborations*: No. Each role is represented as an aspect and related roles are not grouped.
6. *Distinguishing roles of same type*: There is no notion of collaborations.
7. *Independent development*: Partially. Using the glue aspect approach, roles can be developed independently. In contrast, the hybrid approach proposes aspects that are specific for a certain core class.
8. *Self problem*: Partially. Using the hybrid approach, there is only one object, and hence, no ambiguity concerning the `self` variable. In contrast, glue aspects combine core objects and role objects with forwarding; hence, true delegation is not supported.
9. *Identity problem*: Partially. As there is only one object in the hybrid approach, objects and role-playing objects share identity. On the contrary, we believe that using glue aspects will lead to different identities. Unfortunately, the authors do not provide further insight concerning that question.

6.4 Mixin Layers

Mixin layers [SB98, SB02] are an implementation technique for collaboration-based designs using mixin classes [BC90]. The basic idea of a mixin class is that it can be defined without giving its superclass. Instead, one mixin class can be combined with different superclasses, thus, allowing to put it at arbitrary places in the inheritance hierarchy. As a result, many different classes can be composed from a set of mixin classes.

[3] `http://aspectj.org`

In [VN96], the authors propose *role components* to implement collaboration-based designs. A role component is a class template representing a role. It is parametrized with the core type of the role itself as well as with the type of all collaborating objects that it is related to. These type parameters lead to a high complexity for realistic examples. Mixin layers build upon this work by scaling the approach to multiple classes, and hence, solving the scalability problems of role components.

```
1 template <class CollabSuper>
2 class CollabThis : public CollabSuper {
3 public:
4   class FirstRole  : public CollabSuper::FirstRole  { ... };
5   class SecondRole : public CollabSuper::SecondRole { ... };
6   class ThirdRole  : public CollabSuper::ThirdRole  { ... };
7   // ... more roles
8 }
```

Listing 6.2: The general form of mixin layers. The superclass of `CollabThis` is given as a template parameter `CollabSuper`. Each of its inner classes inherits from an inner class with the same name.

In a mixin layer, a collaboration is represented by a mixin that encapsulates other mixins, its roles. [SB98] proposes a concrete implementation technique for C++, using only standard language constructs such as inheritance, inner classes, and templates. Listing 6.2 shows the general form of such mixin layers. Each collaboration extends another collaboration that is given as a template parameter, and hence, left abstract. The inner classes, that is the roles of the collaboration, each extend an inner class of the supercollaboration with the same name. The approach is based upon the assumption that all role implementations use standardized names.

To instantiate a combination of mixin layers, that is, compose different mixins into concrete classes, we parametrize a sequence of layers:

```
typedef Collab3<Collab2<Collab1<CoreClasses>>> MyConcreteClasses;
```

The last layer, `CoreClasses`, is the most general in the inheritance hierarchy. It must be a class with inner classes that do not depend on other template parameters. For implementations of role-based designs, these are usually the core classes.

Graphically, mixin layers can be depicted as in Figure 6.2. Obviously, they can be used to implement layered designs, in which each layer covers one aspect of an application. One major benefit is that, in principle, each layer can be exchanged by another, allowing different variants of a piece of software. However, there can be subtle semantic dependencies between mixin layers that are not reflected in their interfaces. For instance, a correct instantiation can depend on the existence or the ordering of layers.

Comparison

Mixin layers are particularly interesting because they allow the implementation of role-based designs with commonly available programming language constructs. However, the assumption that all layers use standardized names for role implementations is very restrictive. In particular, reuse of layers in different applications becomes practically impossible.

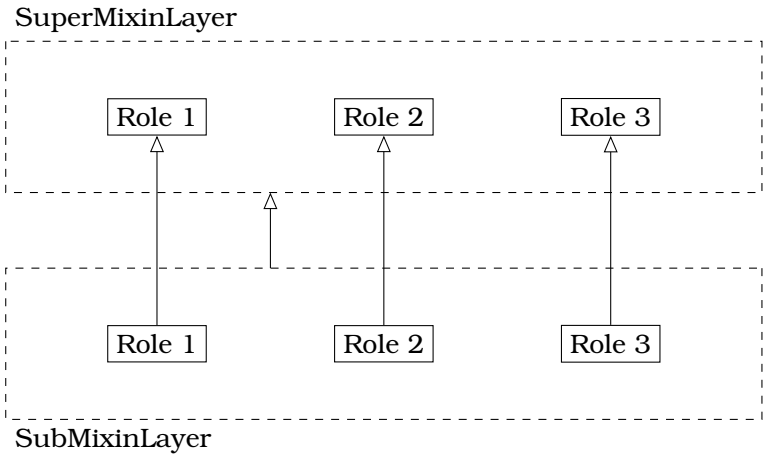


Figure 6.2: Mixin layers graphically: One outer class extends another; their inner classes extend each other as well.

Let us analyze mixin layers using our criteria:

1. *Language compatibility*: Yes. Mixin layers build upon commonly available language features. Implementation proposals for C++, Java, CLOS, and Smalltalk are given in [SB98].
2. *Dynamism*: No. Mixins are composed into concrete types. Once objects are instantiated, their interface cannot change anymore.
3. *Type safety*: Yes. Instances of composed mixins can be accessed type-safely.
4. *Views*: No. As mixin layers are composed before instantiating objects, their set of members is immutable.
5. *Collaborations*: Yes. Mixin layers describe collaborations.
6. *Distinguishing roles of same type*: No. Collaborations are composed statically and are not instantiated. Thus, an object cannot play the same role multiple times.
7. *Independent development*: No. In principle roles can be developed independently from their core classes. In practice, however, we consider this to be hardly possible as each role must have the same name as its core class.
8. *Self problem*: As there is only one object combining core members and different roles, the problem does not exist.
9. *Identity problem*: Same as with 8.

6.5 Role Object Pattern

The Role Object pattern [BRSW00] is another implementation technique for roles that goes without any language extensions. The pattern 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. A client deals with a common superclass of core classes and role classes. Clients add roles to and remove them from a core object by calling appropriate methods on it and passing a role descriptor as argument. Figure 6.3 shows a class diagram of the pattern.

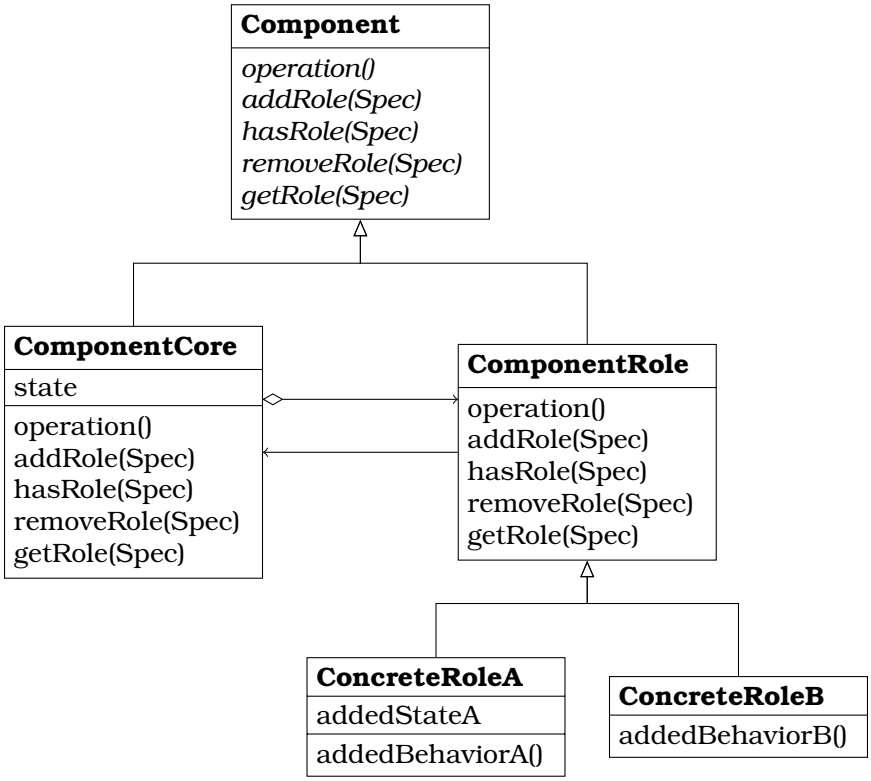


Figure 6.3: A class diagram of the Role Object pattern.

However, the approach has 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 which makes code more complex. Second, the role object pattern suffers from the problem of object schizophrenia. This means the fact that multiple software objects are used to describe one physical object, leading to an unclear notion of object identity. As a result, clients cannot rely on comparing objects, for instance, when adding them to a hash map.

Comparison

1. *Language compatibility*: Yes. The pattern only relies on language constructs available in most object-oriented languages.

2. *Dynamism*: Yes. Clients can add and remove roles using `addRole` and `removeRole`.

3. *Type safety*: No. Before accessing a role, clients should check if it is available with `hasRole`, and then, need to down-cast it to the concrete role type.
4. *Views*: Yes. Concrete roles can provide members that are not visible in the core.
5. *Collaborations*: No. The pattern does not consider relations between roles.
6. *Distinguishing roles of same type*: As there is no notion of collaborations, the question is not sensible.
7. *Independent development*: No. The core and the concrete roles must extend a common superclass `Component`.
8. *Self problem*: No. Role objects forward method calls they cannot answer to the core without overwriting the `self` variable.
9. *Identity problem*: No. The core and its roles have different object identities.

6.6 Summary

This section summarizes the comparison of related work with our proposal. At first, we examine our approach in the light of our nine criteria used above:

1. *Language compatibility*: Yes. Our approach builds upon commonly available language features and mechanisms, such as inner classes, type parameters, and dynamic proxies.
2. *Dynamism*: Yes. Roles can be added to and removed from objects dynamically, for instance, using the `as` operator.
3. *Type safety*: Yes. Role-playing objects have the type of both the core object and the attached roles.
4. *Views*: Yes. Role members are only visible when the corresponding role is attached to the core.
5. *Collaborations*: Yes. Collaborations are represented as outer classes that must be instantiated by clients.
6. *Distinguishing roles of same type*: Yes. Due to path-dependent types, roles with the same role type that belong to different collaboration instances have different types. Hence, they can be distinguished at compile time.
7. *Independent development*: Yes. Collaborations can be developed independently from their core types, hence, allowing reuse in different contexts.
8. *Self problem*: Yes. The problem can be solved due to Scala's way to translate traits (see Section 4.1).
9. *Identity problem*: Partially. Core objects and role-playing objects seem to share identity, however, not in all cases (see Section 4.1).

Finally, we can summarize the comparison in a table:

	Language compatibility	Dynamism	Type safety	Views	Collaborations	Distinguishing roles of same type	Independent development	Self problem	Identity problem
Scala Roles	+	+	+	+	+	+	+	+	0
Epsilon	-	+	-	+	+	0	+		
ObjectTeams	-	+	+	+	+	+	+	+	0
AOP	-	+	0	-	-		0	0	0
Mixin Layers	+	-	+	-	+	-	-	+	+
Role Object pattern	+	+	-	+	-		-	-	-

To the best of our knowledge, ObjectTeams/Java is the most sophisticated system for role-based programming. There is a whole body of research on it available, accompanied by experimental results.[4] The major drawback is that ObjectTeams/Java is a special-purpose language, and hence, incompatible with existing tools and libraries. In contrast, our approach is embedded as a library in the general purpose language Scala.

Epsilon contains interesting ideas for realizing roles and seems to provide a syntax that is similar to ours. Unfortunately, there is no ready-to-use tool available to verify its usefulness.

The relation between aspects and roles seems an interesting topic for further research. The implementation techniques presented above are a first step but lack a number of important features.

The two language-independent approaches, mixin layers and the Role Object pattern, both have its benefits. However, the naming assumption of the first seems to limit its usefulness in practice. The most important advantage of the Role Object pattern is, without doubt, its universal applicability. Important issues like the identity problem and the self problem are not solved, though.

[4] http://www.objectteams.org/publications/index.html

7 Conclusions

The main question of this thesis is whether programming with roles is possible in a lightweight manner, that is, without changing the underlying programming language. The value of roles and collaborations to express the interrelations of objects in a certain context has been proven in modeling. However, roles have not yet found their way into the toolbox of programmers. We argue that this is due to the lack of support by mainstream programming languages and the fact that most proposals towards role-based programming involve tremendous language changes or even completely new languages. Such radical changes offend the average programmer and make it hard to establish role-based programming in everyday's work.

In contrast, we propose a library approach for enabling programming with roles and collaborations. Based on a language-independent analysis of design issues, we show how to realize our proposal in Scala. We verify our approach by numerous examples, in particular, its application to design patterns. Hence, the answer to our question is *yes*; programmers can benefit from roles in a language that was not designed for that special purpose.

We believe our results to be useful in multiple ways. First of all, our approach may serve as a foundation for similar works in other languages. Furthermore, our implementation, Scala Roles, is applicable for implementing or refactoring applications in a role-based style. Finally, our work may serve as a practical example for teaching roles to future software engineers.

A number of problems are left open for future research. First, the problem of shared object identity for objects and role-playing objects could not be solved completely. Second, it remains to clarify how to provide access to collaboration instances for scattered concerns in large applications. Third, it seems worthwhile to investigate how existing references to objects could be updated when an object gets enhanced with a role. This work may also lead to interesting new topics. For example, the relation between collaborations and first-class relationships should be further investigated. It could also be worthwhile to generalize our approach to enable delegation in Scala.

Bibliography

[ADN+03] Heidrun Allert, Peter Dolog, Wolfgang Nejdl, Wolf Siberski, and Friedrich Steimann.
Role-oriented models for hypermedia construction - conceptual modeling for the semantic web.
Technical report, 2003.

[Arn04] Karine Arnout.
From Patterns to Components.
PhD thesis, ETH Zürich, 2004.

[BBvdT07] Matteo Baldoni, Guido Boella, and Leendert W. N. van der Torre.
Interaction between objects in powerJava.
Journal of Object Technology, 6(2), 2007.

[BC90] Gilad Bracha and William Cook.
Mixin-based inheritance.
In Norman Meyrowitz, editor, *Joint Conference on Object-Oriented Programming: Systems, Languages, and Applications and the European Conference on Object-Oriented Programming*, pages 303–311. ACM, 1990.

[BCH02] Gregory Biegel, Vinny Cahill, and Mads Haahr.
A dynamic proxy based architecture to support distributed Java objects in a mobile environment.
In *On the Move to Meaningful Internet Systems: Confederated International Conferences DOA, CoopIS and ODBASE '02*, pages 809–826. Springer, 2002.

[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.

[BGK+97] Dirk Bäumer, Guido Gryczan, Rolf Knoll, Carola Lilienthal, Dirk Riehle, and Heinz Züllighoven.
Framework development for large systems.
Communications of the ACM, 40(10):52–59, 1997.

[BR97] Dirk Bäumer and Dirk Riehle.
Product Trader.
In *Pattern Languages of Program Design 3*, pages 29–46. Addison-Wesley, 1997.

[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.

[DDDCG01] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini.
Fickle: Dynamic object re-classification.
In *European Conference on Object-Oriented Programming (ECOOP '01)*, pages 130–149. Springer, 2001.

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

[Gro03] Christian Grothoff.
Walkabout revisited: The runabout.
In *European Conference on Object-Oriented Programming (ECOOP '03)*, pages 103–125. Springer, 2003.

[Gua92] Nicola Guarino.
Concepts, attributes and arbitrary relations: Some linguistic and ontological criteria for structuring knowledge bases.
Data & Knowledge Engineering, 8(3):249–261, 1992.

[Har97] William Harrison.
Homepage on subject-oriented programming and design patterns.
http://www.research.ibm.com/sop/sopcpats.htm, 1997.

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

[HHM04] Stephan Herrmann, Christine Hundt, and Katharina Mehner.
Translation polymorphism in Object Teams.
Technical Report 2004/05, TU Berlin, 2004.

[HHM07] Stephan Herrmann, Christine Hundt, and Marco Mosconi.
ObjectTeams/Java language definition - version 1.0.
Technical Report 2007/03, Technical University Berlin, 2007.

[HK02] Jan Hannemann and Gregor Kiczales.
Design pattern implementation in Java and AspectJ.
SIGPLAN Notices, 37(11):161–173, 2002.

[HU02] Stefan Hanenberg and Rainer Unland.
Roles and aspects: Similarities, differences, and synergetic potential.
In *International Conference on Object-Oriented Information Systems (OOIS '02)*, pages 87–94. Springer, 2002.

[Ken99] Elizabeth A. Kendall.
Role model designs and implementations with aspect-oriented programming.
SIGPLAN Notices, 34(10):353–369, 1999.

[KFGS03] Dae-Kyoo Kim, Robert France, Sudipto Ghosh, and Eunjee Song.

A role-based metamodeling approach to specifying design patterns.
In *International Conference on Computer Software and Applications (COMPSAC '03)*. IEEE Computer Society, 2003.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin.
Aspect-oriented programming.
In *European Conference on Object-Oriented Programming (ECOOP '97)*, pages 220–242, 1997.

[KO96] Bent Bruun Kristensen and Kasper Osterbye.
Roles: Conceptual abstraction theory and practical language issues.
Theory and Practice of Object Systems, 2(3):143–160, 1996.

[KRS98] Gerti Kappel, Werner Retschitzegger, and Wieland Schwinger.
A comparison of role mechanisms in object-oriented modeling.
In *Modellierung '98, GI-Workshop*. CEUR, 1998.

[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.

[LJ03] Ralf Lämmel and Simon Peyton Jones.
Scrap your boilerplate: A practical design pattern for generic programming.
SIGPLAN Notices, 38(3):26–37, 2003.

[Oa06] Martin Odersky and al.
An overview of the Scala programming language.
Technical Report 2006-001, EPFL Lausanne, Switzerland, 2006.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger.
A nominal theory of objects with dependent types.
In *European Conference on Object-Oriented Programming (ECOOP'03)*, 2003.

[Ode08a] Martin Odersky.
Scala by example.
http://www.scala-lang.org/docu/files/ScalaByExample.pdf, April 2008.

[Ode08b] Martin Odersky.
Scala Language Specification, May 2008.
Version 2.7.

[OK95] Kasper Osterbye and Bent Bruun Kristensen.
Roles.
Technical Report R-95-2006, Department of Mathematics and Computer Science, Aalborg University, March 1995.

[OMGO07] Object Management Group OMG.

OMG Unified Modeling Language (OMG UML), Superstructure, v2.1.2, November 2007.

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

[OZ05a] Martin Odersky and Matthias Zenger.
Independently extensible solutions to the expression problem.
In *International Workshop on Foundations of Object-Oriented Languages (FOOL 12)*, 2005.

[OZ05b] Martin Odersky and Matthias Zenger.
Scalable component abstractions.
In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*, pages 41–57. ACM, 2005.

[PHA07] Michael Pradel, Jakob Henriksson, and Uwe Assmann.
A good role model for ontologies: Collaborations.
In *International Workshop on Semantic-Based Software Development*, 2007.

[PJ98] Jens Palsberg and C. Barry Jay.
The essence of the visitor pattern.
In *International Computer Software and Applications Conference (COMPSAC)*, pages 9–15, 1998.

[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.

[Pra08] Michael Pradel.
Ontology composition using a role modeling approach.
In *Informatiktage*, Lecture Notes in Informatics (LNI). GI, 2008.

[RG98] Dirk Riehle and Thomas Gross.
Role model based framework design and integration.
In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '98)*, pages 117–133. ACM, 1998.

[Rie96] D. Riehle.
Describing and composing patterns using role diagrams.
In *International Conference on Object-Oriented Technology (WOON '96)*, 1996.

[Rie97a] Dirk Riehle.
Composite design patterns.
In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97)*, pages 218–228. ACM, 1997.

[Rie97b] Dirk Riehle.
A role-based design pattern catalog of atomic and composite patterns structured by pattern purpose.
Technical Report 97-1-1, Ubilab, 1997.

[Rie00] Dirk Riehle.
Framework Design: A Role Modeling Approach.
PhD thesis, ETH Zürich, 2000.

[Rie07] Dirk Riehle.
Junit 3.8 documented using collaborations.
Technical report, 2007.

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

[SB98] Yannis Smaragdakis and Don Batory.
Implementing layered designs with mixin layers.
In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '98)*, pages 550–570. Springer, 1998.

[SB02] Yannis Smaragdakis and Don Batory.
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, 2002.

[SDNB03] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew Black.
Traits: Composable units of behavior.
In *European Conference on Object-Oriented Programming (ECOOP '03)*, pages 248–274. Springer, 2003.

[Ser99] Manuel Serrano.
Wide classes.
In *European Conference on Object-Oriented Programming (ECOOP '99)*, pages 391–415. Springer, 1999.

[SH08] Michel Schinz and Philipp Haller.
A Scala tutorial for Java programmers.
http://www.scala-lang.org/docu/files/ScalaTutorial.pdf, April 2008.

[Sow84] John F. Sowa.
Conceptual structures: information processing in mind and machine.
Addison-Wesley, 1984.

[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.

[VN96] Michael VanHilst and David Notkin.
Using role components to implement collaboration-based designs.

In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '96)*, pages 359–369. ACM, 1996.

Confirmation

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, June 13, 2008