Roles and Collaborations in Scala

Michael Pradel

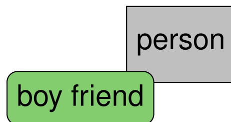
TU Dresden / EPFL Lausanne

Supervised by: Prof. Uwe Aßmann, Prof. Martin Odersky, Jakob Henriksson

June 26, 2008

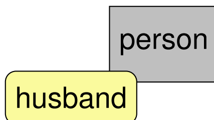
Motivation - Why Roles?

- Objects ...
  - evolve at runtime
  - are used differently depending on the context
  - interact in manifold ways
- Roles ...
  - dynamically add/remove members to/from objects
  - provide views
  - are grouped into collaborations



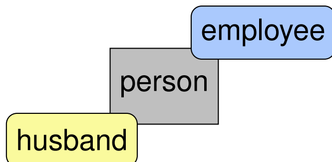
Motivation - Why Roles?

- Objects ...
  - evolve at runtime
  - are used differently depending on the context
  - interact in manifold ways
- Roles ...
  - dynamically add/remove members to/from objects
  - provide views
  - are grouped into collaborations



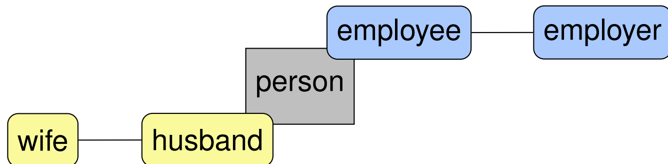
Motivation - Why Roles?

- Objects ...
  - evolve at runtime
  - are used differently depending on the context
  - interact in manifold ways
- Roles ...
  - dynamically add/remove members to/from objects
  - provide views
  - are grouped into collaborations



Motivation - Why Roles?

- Objects ...
  - evolve at runtime
  - are used differently depending on the context
  - interact in manifold ways
- Roles ...
  - dynamically add/remove members to/from objects
  - provide views
  - are grouped into collaborations



Motivation - Why Scala Roles?

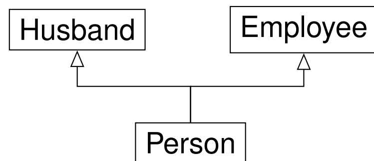
- Roles are known for a long time
- More or less accepted in modeling
- But not in programmer's toolbox
- Existing solutions to role-based programming
  - Inconvenient, bulky syntax, or
  - Heavyweight language extensions
- Idea: Let's do it in a Scala library
  - Easy, simple syntax
  - Lightweight - no change to language

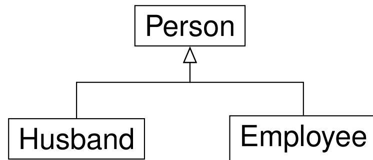
Goals

- 15 features of roles (Steimann), e.g.
  - Roles have state and behavior
  - Multiple roles per object
  - Dynamically adding and removing roles
- Conserve underlying language
- Type safety
- Collaborations as programming and reuse abstraction

Representing Roles

Roles as classes?





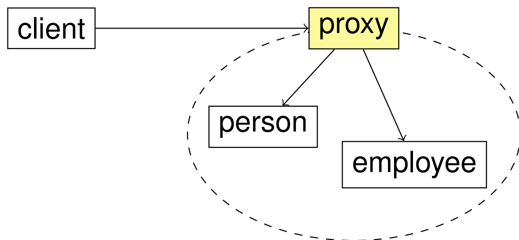
- All instances play the roles
- Roles depend on core types

Roles as traits? No dynamism.

→ Our approach: Roles as objects

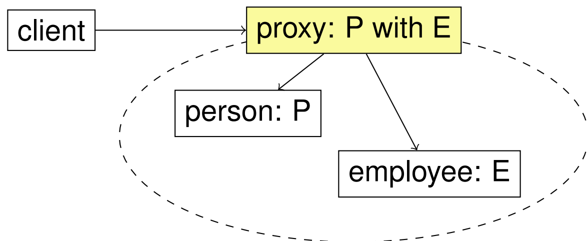
Compound Objects with Dynamic Proxies

- An object and its roles should appear as one object
  → Compound object
- Idea: Represent them with a dynamic proxy
- Created at runtime on demand
- Proxy delegates using reflection
- Type-safe access to role-playing objects



Compound Objects with Dynamic Proxies

- An object and its roles should appear as one object
  $\longrightarrow$ Compound object
- Idea: Represent them with a dynamic proxy
- Created at runtime on demand
- Proxy delegates using reflection
- Type-safe access to role-playing objects



The `as` operator

- One simple operator for accessing roles:
  `object as role`
- Returns `object` and `role` hidden behind a proxy
- Problem: Roles can be bound to arbitrary objects, i.e. not having a method `as`
- Solution: Implicit conversion
  `object.as(role)` $\longrightarrow$ `role.playedBy(object)`

Representing Collaborations

- Nesting of traits (or classes)
- Outer trait is collaboration, inner traits are roles

```
class Employment(hourlyWage: Int) extends TransientCollaboration {
  val employee = new Employee{}
  val employer = new Employer{}

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

  trait Employer extends Role[Person] {
    def payOff = {
      employee.money += employee.hoursWorked * hourlyWage
      employee.hoursWorked = 0
} } }
```

.. and how to use it

```
val jack = new Person{}
val bill = new Person{}
val mary = new Person{}

val company = new Employment(15)
val pub = new Employment(7)

(bill as company.employee).work
(jack as company.employer).payOff

(mary as pub.employee).work
(bill as pub.employer).payOff
```

Roles and Role Mappers

- Sometimes useful: Arbitrary many instances of a role
- Role mappers ...
  - create new role instances on demand
  - manage binding between cores and roles
- Same syntax: `object as role`
- Example: Multiple employees

```
bill as company.employee
paul as company.employee
bill as company.employee
```

→ Two role instances

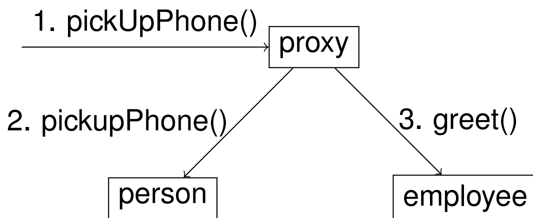
Sticky Roles

- Alternative to `as`: Sticky roles
- Similar to first-class relationships
- Participants of collaboration given in constructor
- Example:

```
val company = new Employment(jack, bill)
company.employee.work
company.employer.payOff
```

Forwarding vs. Delegation (Self Problem)

- Delegation: `this` always refers to the original receiver of a method call
- Usual behavior in object-based languages
- Example: Employee overrides `greet` method



Delegation with Proxies and Traits

How Scala translates traits:

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

→

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

- Idea: Set `$this` to the proxy
- Method dispatch is done reflectively
  1. Delegate to role object, if possible
  2. Delegate to core object, otherwise

Case Study: Design Patterns

- Patterns assign roles to participating objects
- Applying the Scala Roles library to 24 patterns (23 Gang of Four + Role Object)

Results:

- Reusable collaborations: Composite, Observer
- Enhancements with roles: Decorator, Mediator, Role Object, Template Method
- Obsolete in Scala: Adapter, Command, Interpreter, Singleton, Strategy, Visitor
- Invariant: remaining 11

A Reusable Pattern Collaboration: Observer

- Observer contains two roles: Subject and Observer
- Most code of the subject can be easily reused:

```
private val observers = new HashSet[Observer]()
def addObserver(o: Observer) = observers += o
def removeObserver(o: Observer) = observers -= o
def notifyObservers = observers.foreach(_.update(this))
```

- Idea: Dynamically add subject role to objects
- Arbitrary objects become observable without changing their class

Example

```
trait Book {
  private var status = "available"
  def borrow = { status = "borrowed" }
  def returnIt(late: Boolean) = { status = "available" }
  def turnPage = { }
}

val b = new Book{}; val l = new Library{}
val o = new ObserverCollab[Book]("status")
// or "borrow()", "returnIt(Boolean)", "returnIt(*)", etc.

val observableBook = b as o.subject
observableBook.addObserver(l)

observableBook.borrow   // invokes l.update(observableBook)
```

Conclusions

- Roles are a useful programming abstraction
- Programming technique to express roles and collaborations
- Compound objects with dynamic proxies
- Access to role-playing objects is type-safe
- It's all just a library: No change of compiler, tools, etc.

See also:

Michael Pradel, Martin Odersky
Scala Roles - A Lightweight Approach towards Reusable Collaborations
ICSOFT 2008

Thanks! Questions?