TECHNISCHE UNIVERSITÄT DRESDEN

DRESDEN concept

RoSI Seminar

Scala Roles Language
From Dust till Dawn

Lars Schütze
Christopher Werner

Courtesy to Max Leuthäuser

"The **advantage** of the **object-oriented** paradigm is that it neatly combines the strengths of the data-centered and the behavior-centered approaches.

It is great for modeling **information** and it is great for modeling **behavior**."

Working with Objects:

The OOram Software Engineering Method, 1996

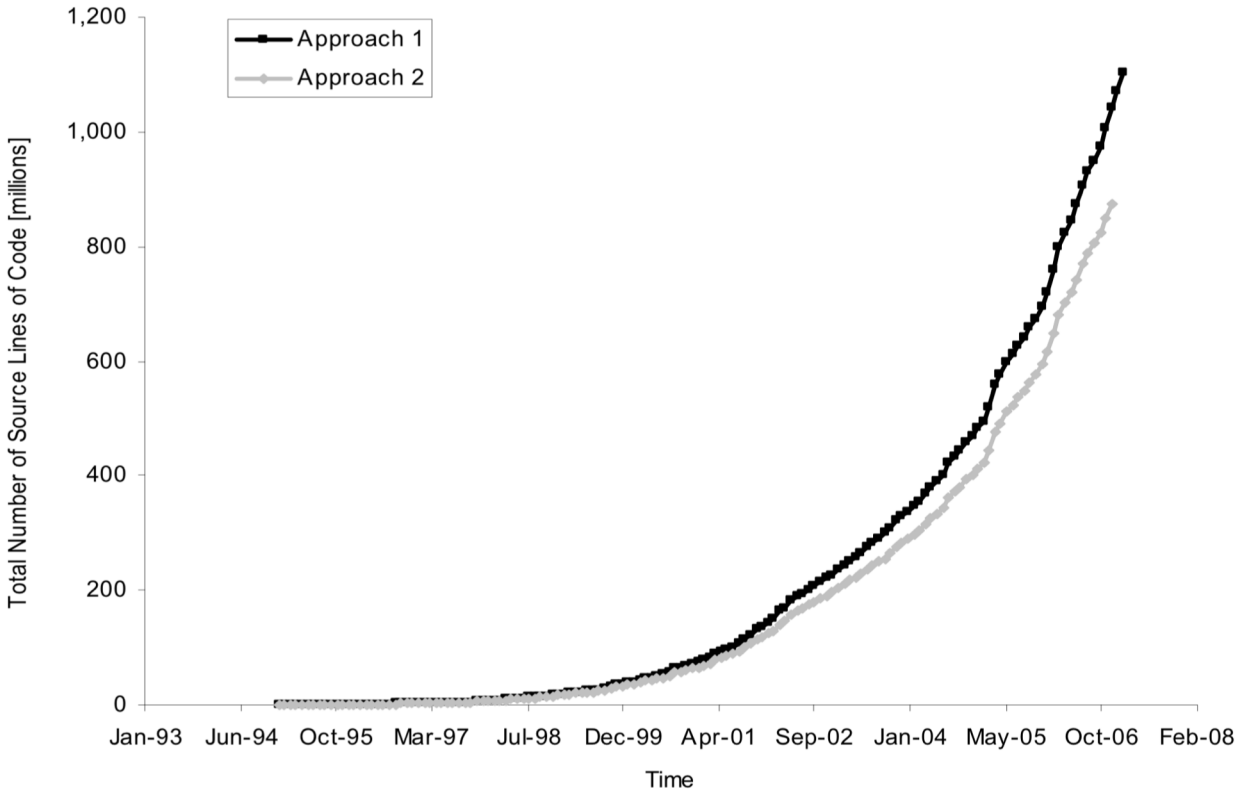
”The [...] system consists of a **very large number** of objects, and the object interaction processes [...] will be **very complex**.”

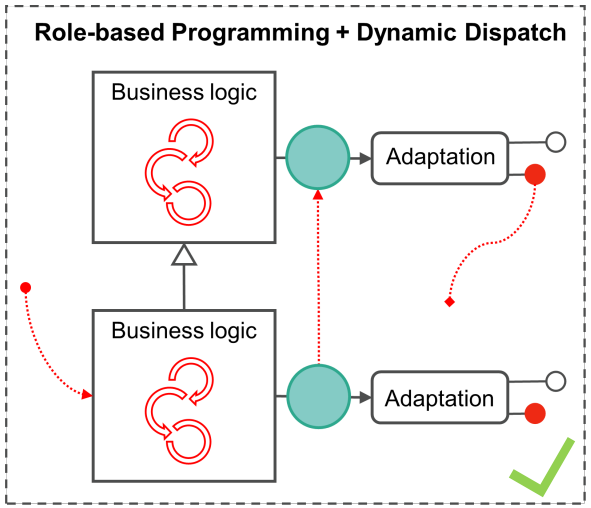
“The OOram method tells us to isolate an **area of concern** and to create a **role model** for it.”

Working with Objects:

The OOram Software Engineering Method, 1996

Historical Point of View Towards Today



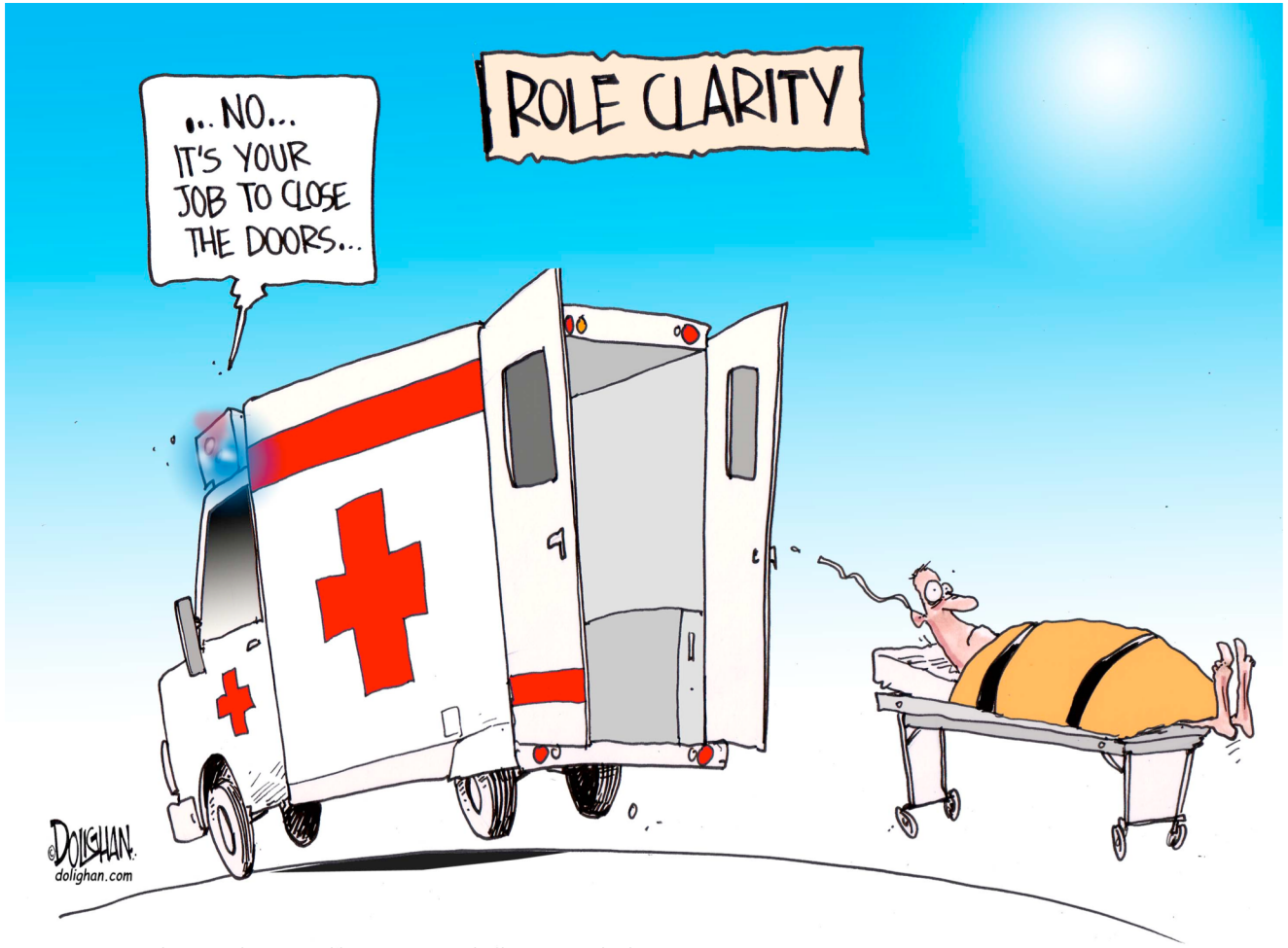


late 1970s

Increasing degree of separation of concerns over time

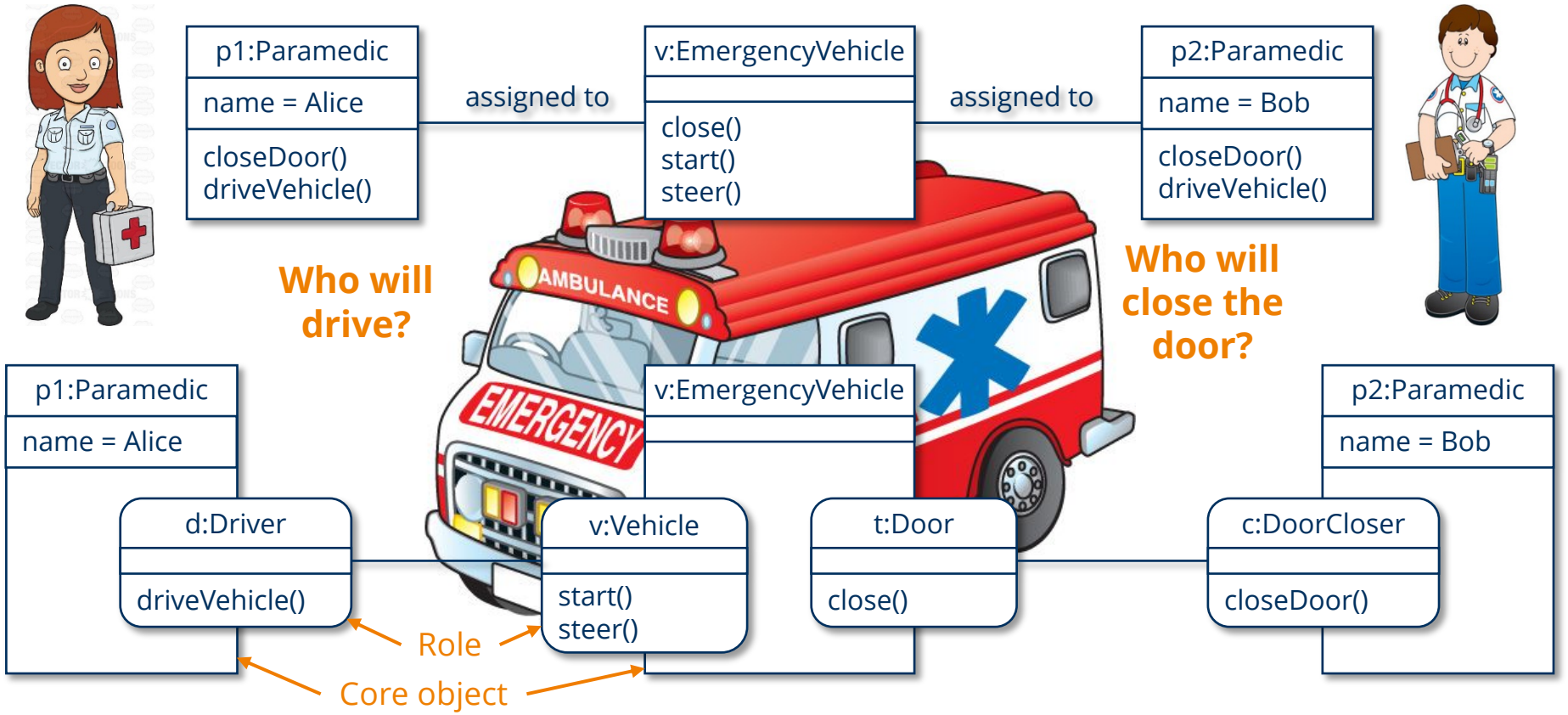
Deshpande, Amit, and Dirk Riehle. "The total growth of open source." *IFIP International Conference on Open Source Systems*. Springer, Boston, MA, 2008.

Role-based? What is a Role?



[http://mindset.com.au/blog/wp-content/uploads/2010/12/Role-Clarity-cartoon.jpg]

Role-based? What is a Role?



Role-based? What is a Role?

p1:Paramedic
name = Alice
closeDoor()
driveVehicle()

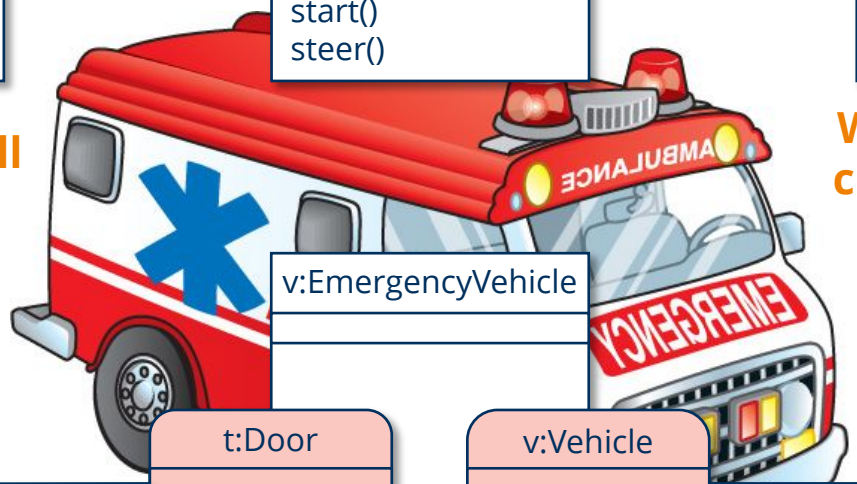
assigned to

v:EmergencyVehicle
close()
start()
steer()

assigned to

p2:Paramedic
name = Bob
closeDoor()
driveVehicle()

Who will drive?



Who will close the door?

p1:Paramedic
name = Alice

c:DoorCloser
closeDoor()

v:EmergencyVehicle

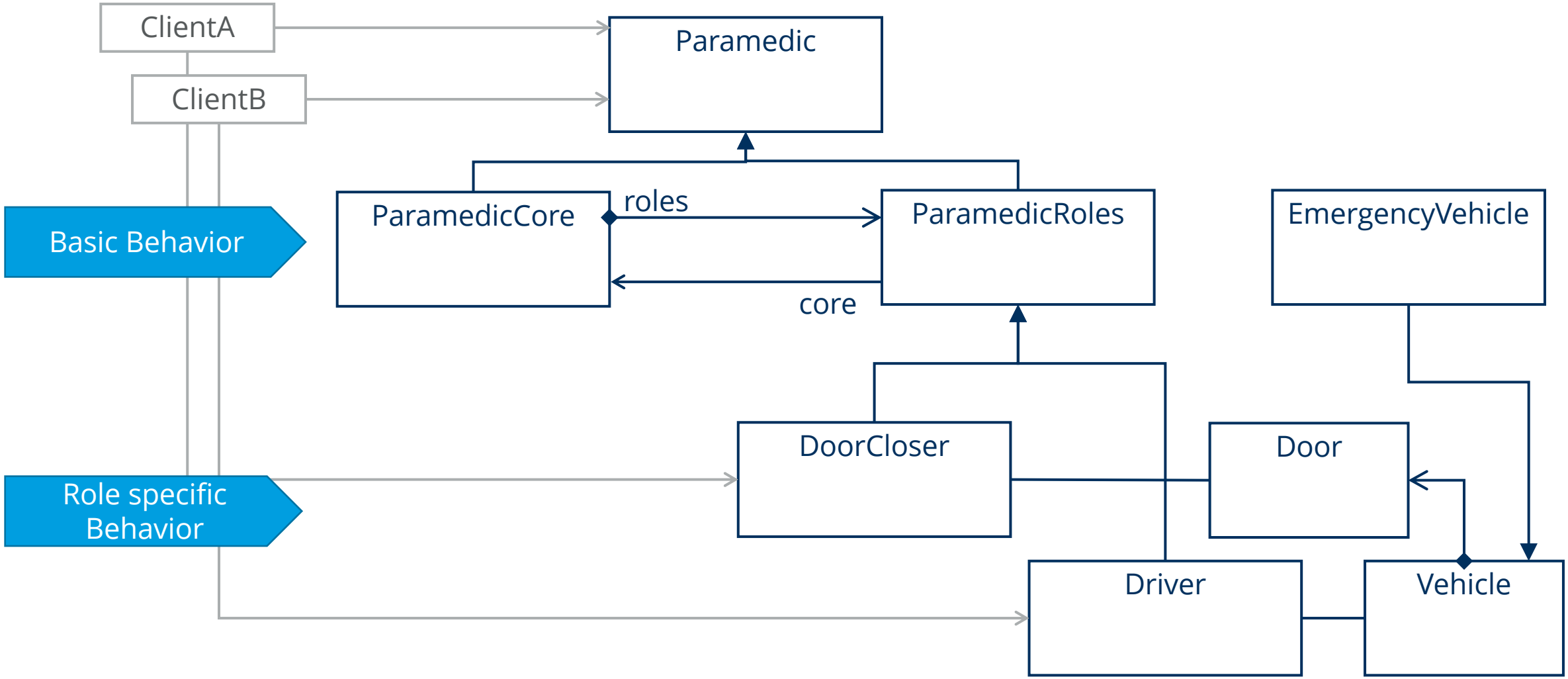
t:Door
close()

v:Vehicle
start()
steer()

p2:Paramedic
name = Bob

d:Driver
driveVehicle()

Role Object Pattern



Dirk Bäumer et. al. (1997) “The Role Object Pattern”, PLoP ‘97 Conference


Roles
Context
OOP

Two Kinds of Context

Context

- Environment inferable from sensor data
- Example: cold and snowy day in London

- Has no identity
- Has no intrinsic behavior
- Indefinite lifetime

Compartment

- Constructed entity containing other entities
- Example: first class train car

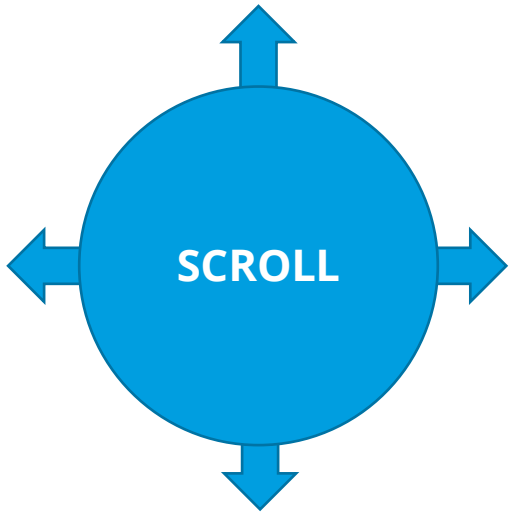
- Instances carry identity
- Has behavior and state
- Defined lifetime


The SCROLL Approach

The SCROLL Approach

Embedded
Host language Scala
No additional tooling

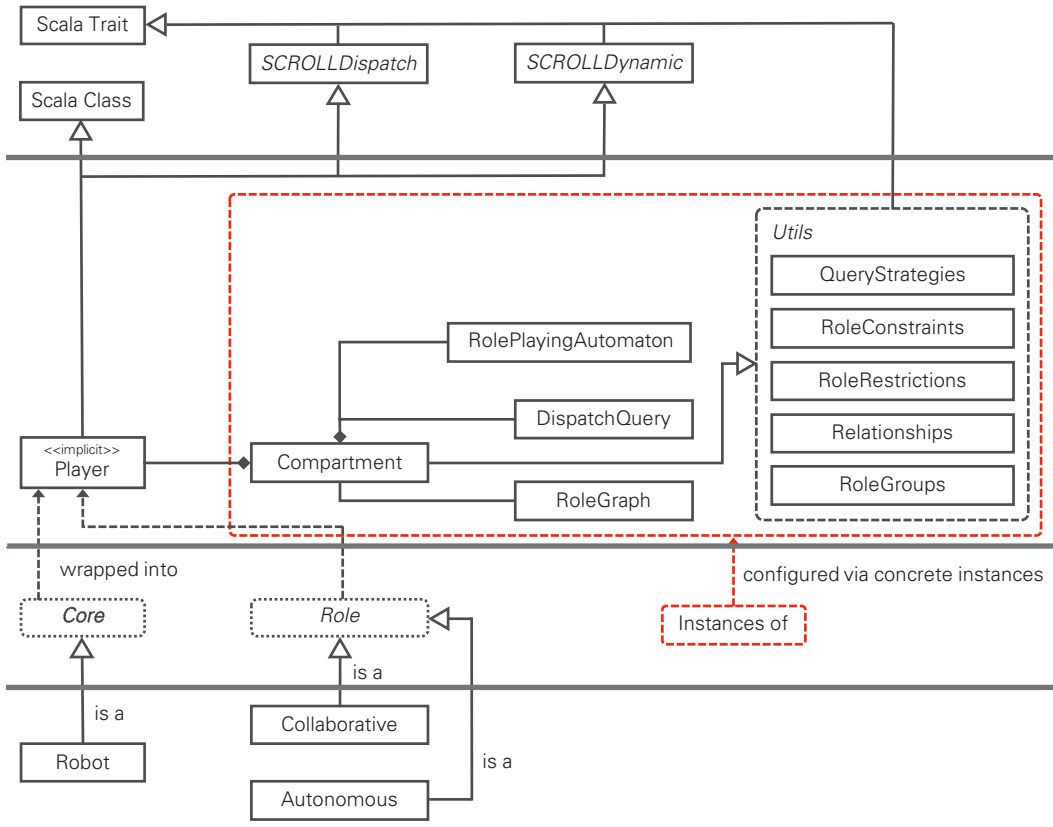
Meta-Object Protocol
Role-related semantics
Dispatch in a layered approach



Method-Call Interception
Compiler-supported rewrites

Domain Specific Language
Easy to use and adapt

SCROLL Layers



Specification Layer

Contains the specification for the dispatch

Metaobject Protocol Layer

Contains the implementation of metaclasses and utility traits

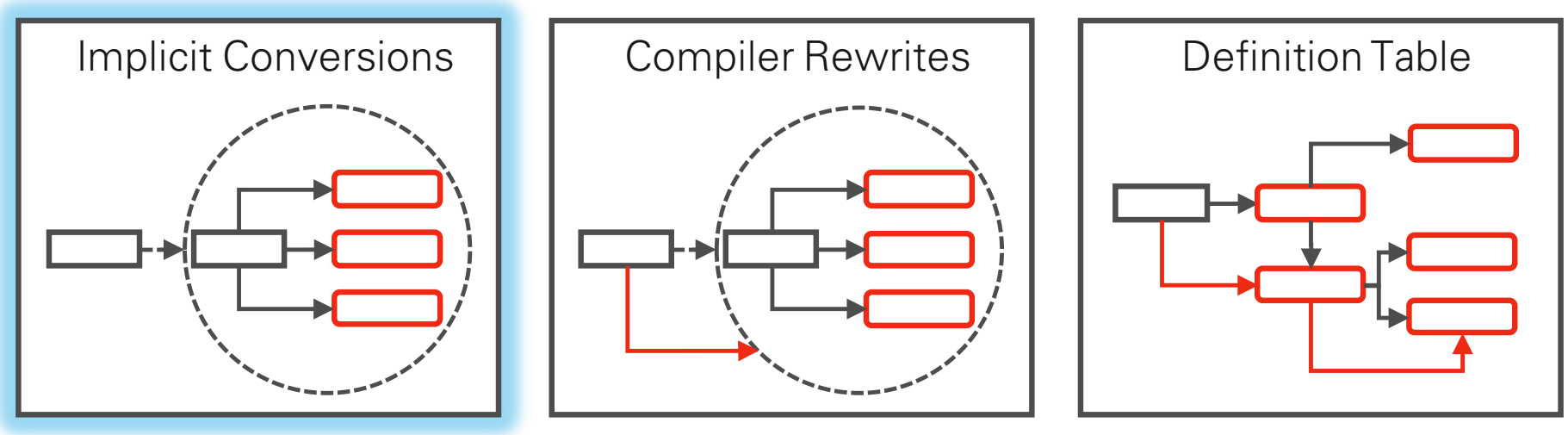
Configuration Layer

Compartment and all utility traits configurable via concrete instances or subclassing

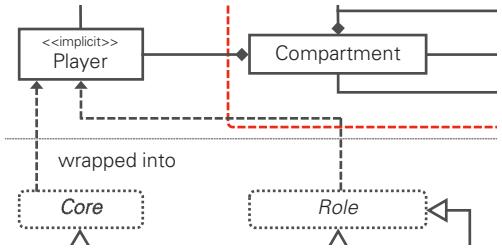
Usage Layer

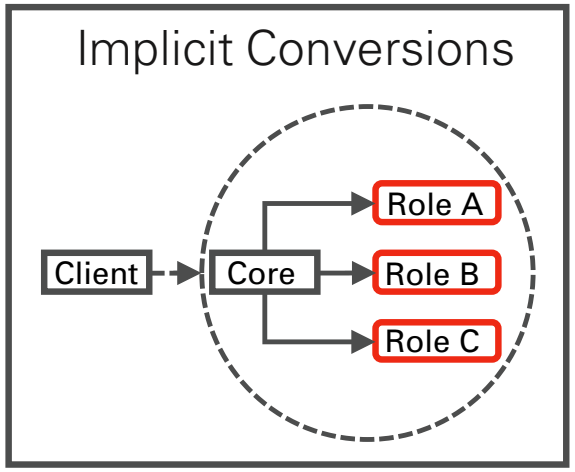
Instantiation and use of role-playing objects

Implementation Pattern



Implicit Conversions





Problem:
Splitting a logical object into physical objects normally leads to object schizophrenia.

Goal:
Transparently being able to access the different physical objects that constitute a logical object from outside.

Solution:
Automatic boxing with Implicits.

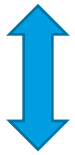
Boxing with Implicits

Definition

```
1  implicit class Player[T](val wrapped : T) {
2     def unary_+ : Player[T] = this
3     def play( role : Any ) : Player[T] = ...
4     override def equals( o : Any ) = ...
5  }
```

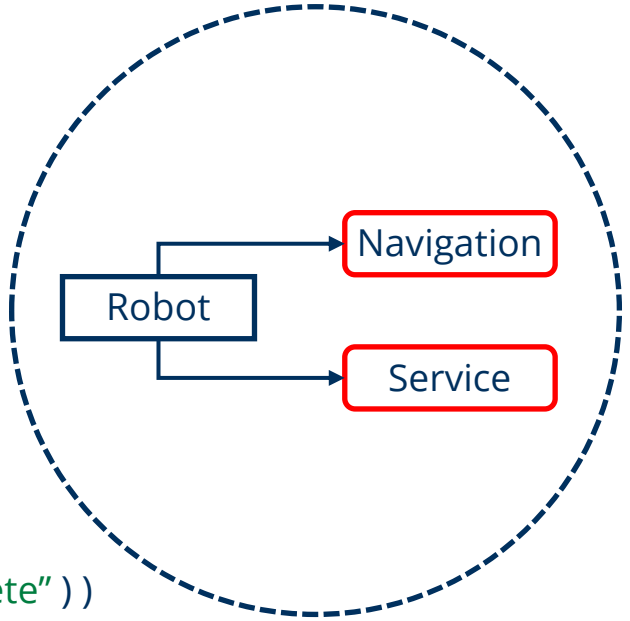
Usage

```
1  new Compartment {
2     val robot = Robot( "Pete" )
3        play Service() play Navigation()
4     ...
5  }
```

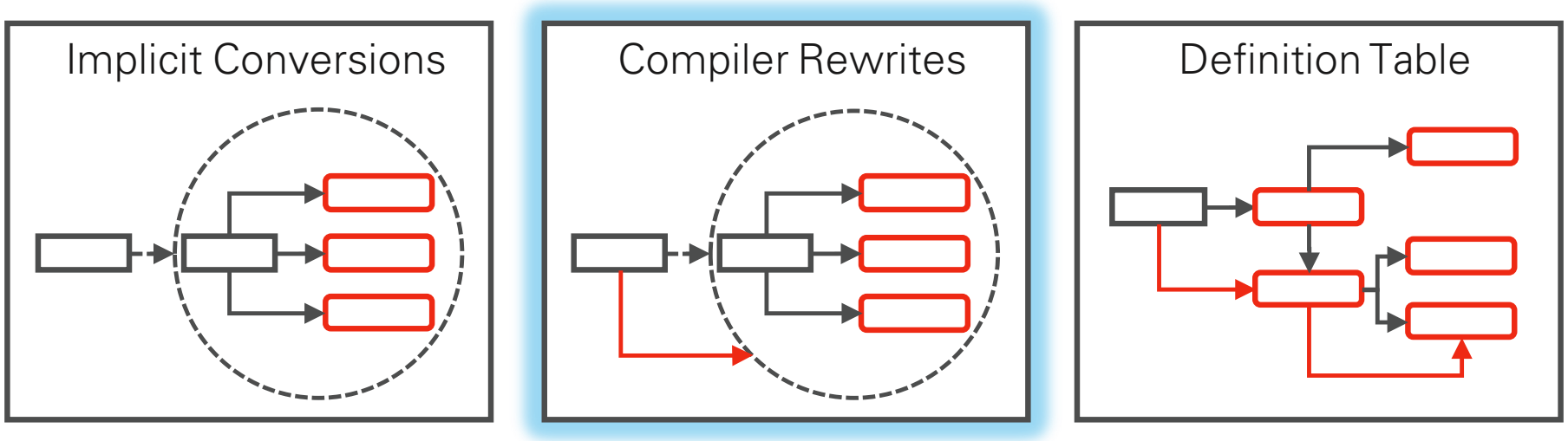


```
1  new Compartment {
2     val robot : Player[ Robot ] =
3        new Player[ Robot ]( new Robot( "Pete" ) )
4        .play( new Service() )
5        .play( new Navigation() )
6     ... }
```

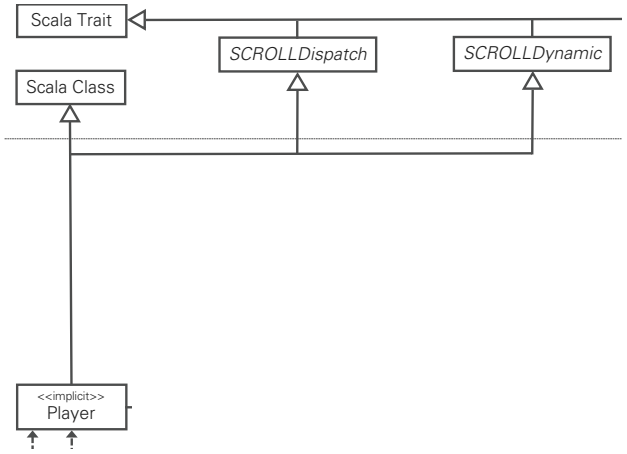
Structure

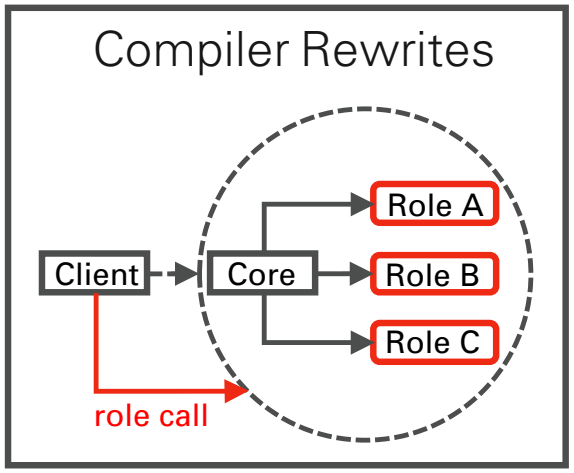


Implementation Pattern



Compiler Rewrites



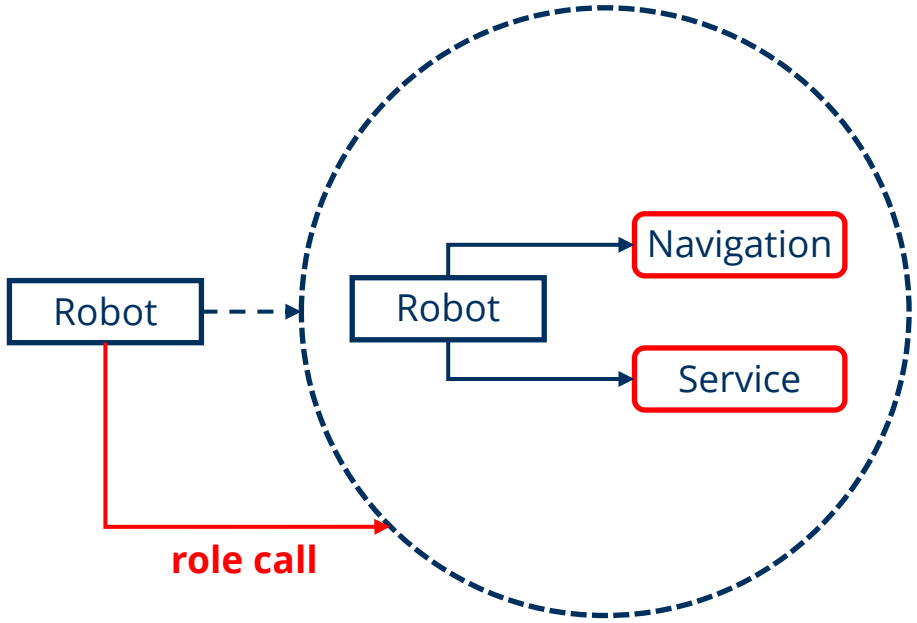


Problem:
Accessing structure or behavior not defined in an objects type is not possible with traditional OOP.

Goal:
All structure and behavior aggregated in the compound object becomes available.

Solution:
Compiler-supported rewrites.

Compiler Rewrites



Rewrite Rules

```
foo.method("param")      ⇝ foo.applyDynamic("method")("param")
foo.method(x = "param")  ⇝ foo.applyDynamicNamed("method")(("x", "param"))
foo.method(x = 1, 2)     ⇝ foo.applyDynamicNamed("method")(("x", 1), ("", 2))
foo.field                ⇝ foo.selectDynamic("field")
foo.varia = 10           ⇝ foo.updateDynamic("varia")(10)
foo.arr(10) = 13         ⇝ foo.selectDynamic("arr").update(10, 13)
foo.arr(10)              ⇝ foo.applyDynamic("arr")(10)
```

Dynamic SCROLL Dispatch

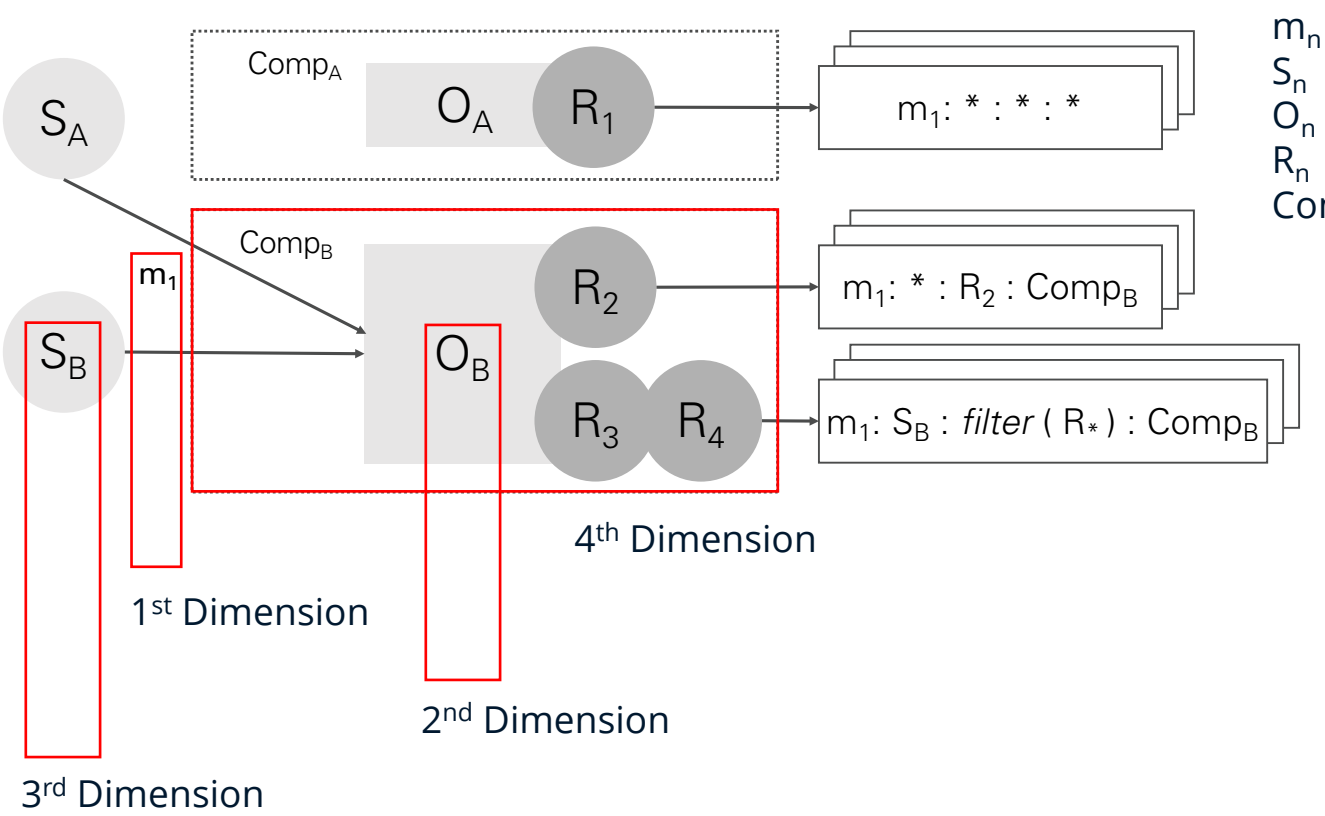
```
foo.method("param")        ⇝ foo.applyDynamic("method")("param")
```

```
1  override def applyDynamic[ E ]( name : String ) ( args : Any* )
2    ( implicit dispatchQuery : DispatchQuery = DispatchQuery.empty ) :
3        Either[ SCROLLError, E ] = {
4   val core = coreFor( wrapped ).last
5   dispatchQuery.filter( plays.roles( core ) ).collectFirst {
6   case r if ReflectiveHelper.findMethod( r, name, args ).isDefined =>
7        ( r, ReflectiveHelper.findMethod( r, name, args ).get )
8   } match {
9     case Some( ( r, fm ) ) => dispatch( r, fm, args: _* )
10    case _ => Left( RoleNotFound( core.toString, name, args ) )
11  }
12 }
```

- Find core if is wrapped by Player object
- Get roles played by core
  - Filter using Dispatch Query rules
  - Collect first resulting role
  - **If** it implements the method

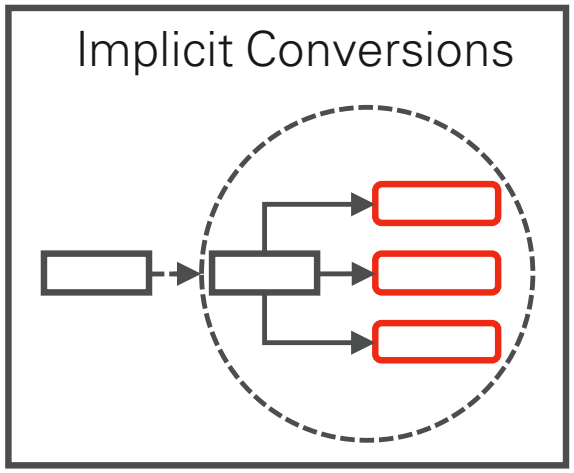
- If so: dispatch to it
- Else: Error!

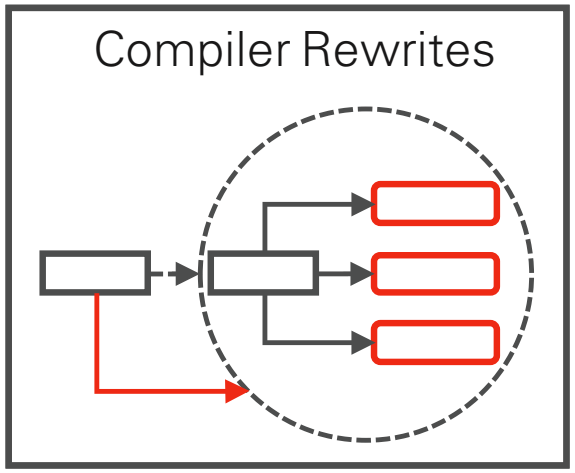
4-Dimensional Dispatch for Roles at Runtime

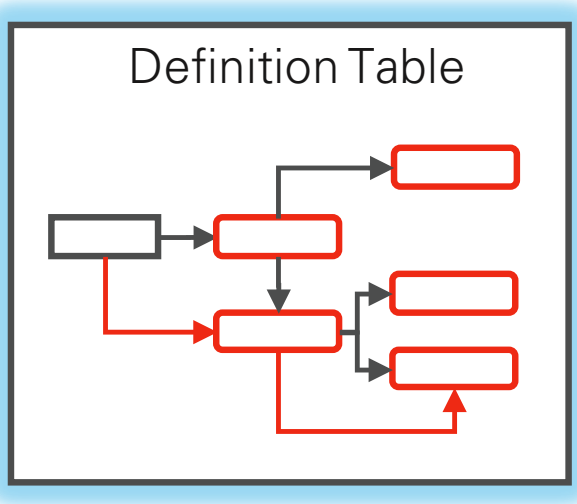


m_n	… Method name
S_n	… Sender Type
O_n	… Player Type
R_n	… Role Type
$Comp_n$	… Compartment Type

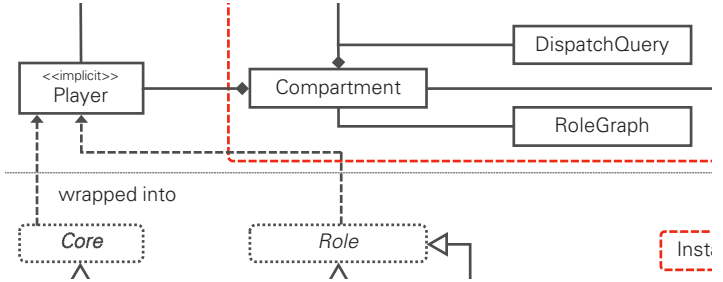
Implementation Pattern

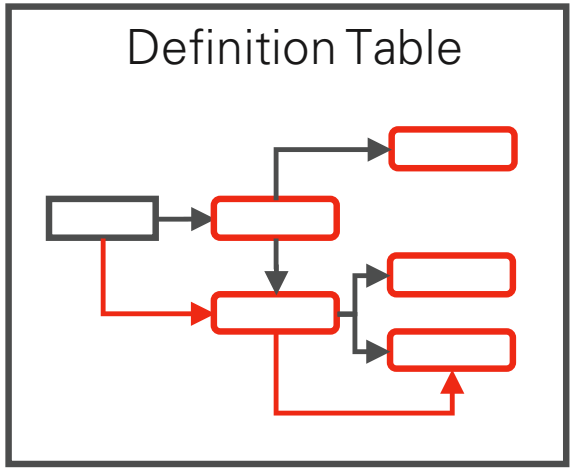






Definition Table





Problem 1:
The core object of a role can also be a role.

Problem 2:
Different roles can have the same signature leading to ambiguity.

Goal:
Store and traverse role playing relations between objects with an adaptable dispatching semantics.

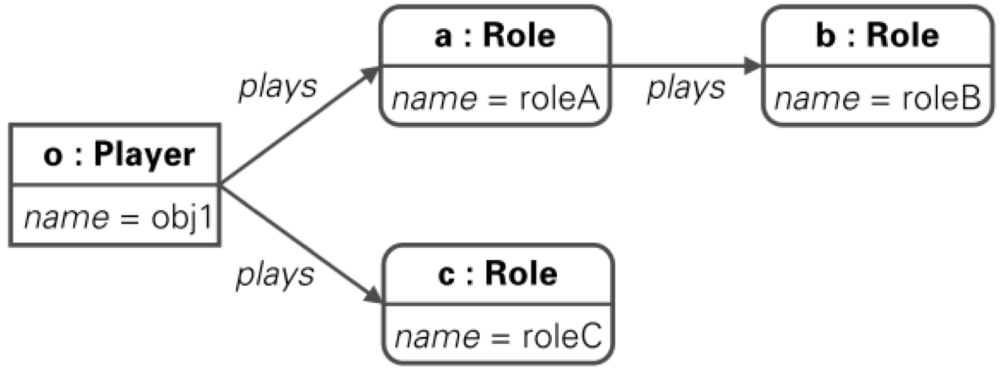
Solution:
Graph-Traversal and programmable Dispatch Filters.

Role-Play Graph

Role-play graph is a 4-tuple (V, E, Lab, Lsum) with

- V : set of objects (core + roles)
- E : set of edges representing relationship
- Lsum : set of type names for all V
- Lab : V → Lsum : assigns each object its type

Example



- V := { o, a, b, c }
- E := { (o, a), (a, b), (o, c) }
- Lsum := { Player, Role }
- Lab := { o → Player, a → Role, b → Role, c → Role }

Graph Traversals and Parameterizable Filters

Basics:

$f\colon D \longrightarrow R$		Functions signature for function f with domain D and range R
$f\,(a_1, a_2, \ldots)$		Function application with arguments a_1, a_2, etc.
$\circ$		Path composition
$\hat{P}(A)$		Power multiset of A, infinite set of all subsets of multisets of A

Traversals:

$\mathcal{E}_{out}$:	$\hat{P}(V) \longrightarrow \hat{P}(E)$	Yield all outgoing edges of a multiset of vertices
$\mathcal{E}_{in}$:	$\hat{P}(V) \longrightarrow \hat{P}(E)$	Yield all incoming edges of a multiset of vertices
$\mathcal{V}_{out}$:	$\hat{P}(E) \longrightarrow \hat{P}(V)$	Traverse the outgoing (i.e., sink) vertices of the edges
$\mathcal{V}_{in}$:	$\hat{P}(E) \longrightarrow \hat{P}(V)$	Traverse the incoming (i.e., source) vertices of the edges
ϵ:	$\hat{P}(V \cup E) \times R \longrightarrow \hat{P}(S)$	Get the element property values for key $r \in R$

Filters:

$\mathcal{E}^{\sigma}_{lab\pm}$:	$\hat{P}(E) \times \Sigma \longrightarrow \hat{P}(E)$	Allow (+) or filter (-) edges with the label $\sigma \in \Sigma$
$\epsilon_{p\pm}$:	$\hat{P}(V \cup E) \times R \times S \longrightarrow \hat{P}(V \cup E)$	Allow (+) or filter (-) elements with the property $s \in S$ for key $r \in R$

Marko A. Rodriguez and Peter Neubauer. The Graph Traversal Pattern. *arXiv* preprint arXiv:1004.1001, 2010

Graph Traversals and Parameterizable Filters

Θ

Set of selection functions

$\alpha : N \rightarrow \{\top, \bot\}$:

```
1 f: Any => Boolean
```

Selection function $\alpha \in \Theta$ assigning boolean values to a given node *N*

$\triangleright : \hat{\mathcal{P}}(N) \times \Theta \rightarrow \hat{\mathcal{P}}(N)$:

```
1 class From(val sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
2   override def apply(edges: Seq[Any]): Seq[Any] =
3     edges.slice(edges.indexWhere(sel), edges.size)
4 }
```

Filter selecting the source node, w.r.t. the selection function $\alpha \in \Theta$

$\triangleleft : \hat{\mathcal{P}}(N) \times \Theta \rightarrow \hat{\mathcal{P}}(N)$:

```
1 class To(val sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
2   override def apply(edges: Seq[Any]): Seq[Any] =
    ↳   edges.lastIndexWhere(sel) match {
3     case -1 => edges
4     case _ => edges.slice(0, edges.lastIndexWhere(sel) + 1)
5   }
6 }
```

Filter selecting the sink nodes, w.r.t. the selection function $\alpha \in \Theta$

Graph Traversals and Parameterizable Filters

$\trianglerighteq: \hat{\mathcal{P}}(N) \times \Theta \to \hat{\mathcal{P}}(N)$:

```scala
class Through(sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
  override def apply(edges: Seq[Any]): Seq[Any] = edges.filter(sel)
}
```

Filter to specify which nodes to keep , w.r.t. the selection function $\alpha \in$

$\not\trianglerighteq: \hat{\mathcal{P}}(N) \times \Theta \to \hat{\mathcal{P}}(N)$:

```scala
class Bypassing(sel: Any => Boolean) extends (Seq[Any] => Seq[Any]) {
  override def apply(edges: Seq[Any]): Seq[Any] = edges.filterNot(sel)
}
```

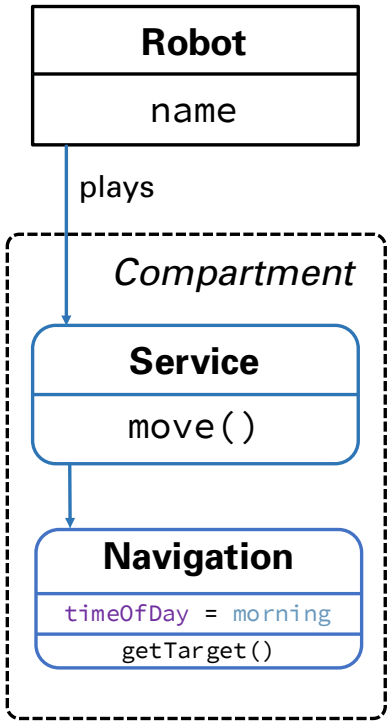
Filter to specify which nodes to remove, w.r.t. the selection function $\alpha \in$

$\Omega : \hat{\mathcal{P}}(N) \to \hat{\mathcal{P}}(N)$:

```scala
class DispatchQuery(from: From, to: To, through: Through, bypassing:
  ↪ Bypassing) {
  def filter(anys: Seq[Any]): Seq[Any] =
    ↪ from.andThen(to).andThen(through).andThen(bypassing)(anys.distinct)
}
```

Composed dispatch query (i.e., a dispatch description)
$\Omega\,(\hat{P}(N)) = (\not\trianglerighteq \circ \trianglerighteq \circ \triangleleft \circ \triangleright)(\hat{P}(N))$

Graph Traversals and Parameterizable Filters




```scala
implicit val dispatchDescription =

  // Filter selecting the starting node:
  From( _.isInstanceOf[Robot] ).

  // Filter selecting the end node:
  To( anything ).

  // Filter to specify which nodes to keep:
  Through( anything ).

  // Filter to specify which nodes to skip:
  Bypassing( _ match {

    // E.g., always skipping instances of Service:
    case Service() => true
    // but keeping instances of Navigation:
    case Navigation(timeOfDay)
         if timeOfDay == morning => false

  })
```