11. Role-Based Design

Dr. Sebastian Götz

Software Technology Group

Department of Computer Science

Technische Universität Dresden

WS16/17 - Dec 7, 2016

1) Running Example
2) The Role-object Pattern
3) Object Schizophrenia
4) Delegation vs. Forwarding
5) Role types formally

Goals

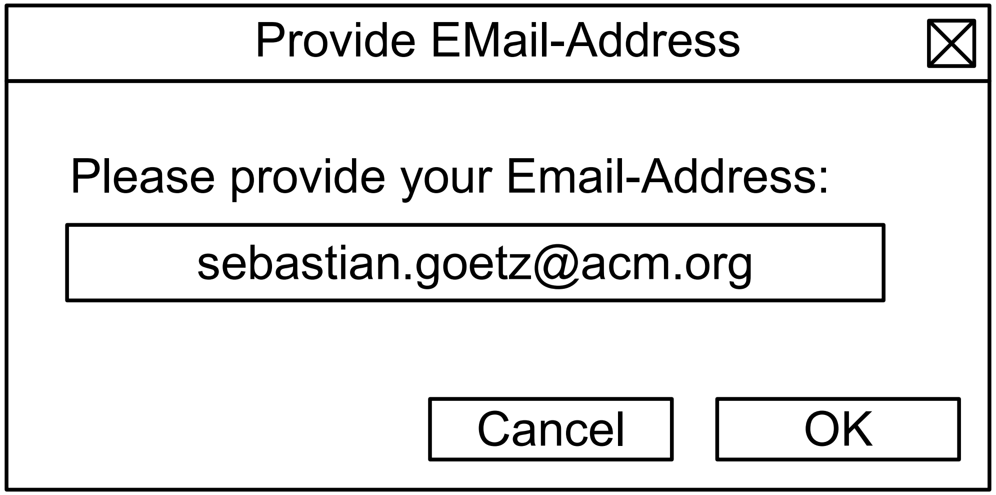
- Understand how roles can be implemented in current mainstream object-oriented languages (e.g., Java)
  - Role-Object Pattern
- Understand the problem of object-oriented compared to role-oriented programming
  - Object Schizophrenia
- Understand the problem of identity and state
  - Delegation versus Forwarding
- Know how role types can be formally distinguished from natural types (i.e., classes in OOP)

11.1 Running Example

A Dialog asking a User for its EMail-Address

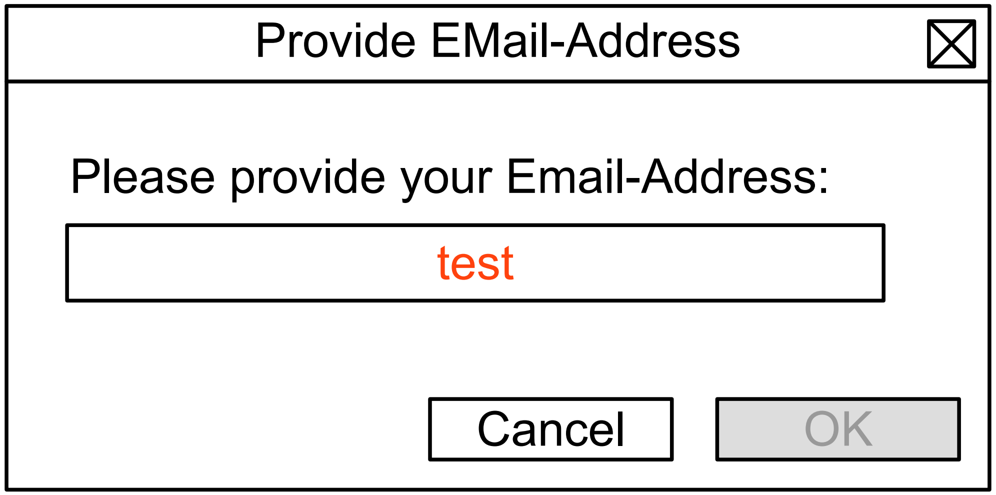
A Dialog Requesting an Email-Address

- User shall provide his Email-Address
- Application want's to ensure that the provided address is valid (Pattern: a@b.c)



A Dialog Requesting an Email-Address

- User shall provide his Email-Address
- Application want's to ensure that the provided address is valid (Pattern: a@b.c)
- Application want's to visualize invalid Email-Addresses



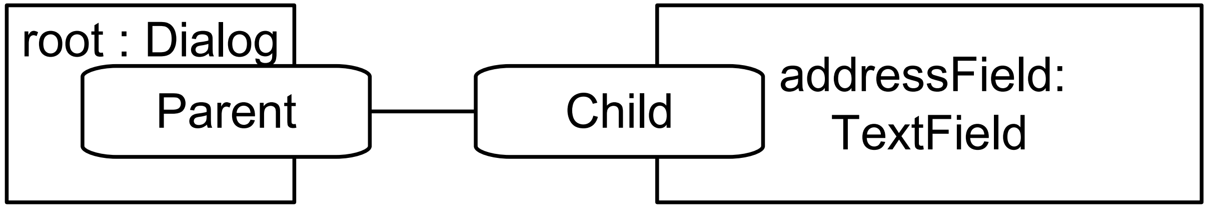
11.2 Role-object Pattern (ROP)

Delegation-based Realization of Roles in Object-oriented Languages

Slides based on:
Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf: **Role Object Pattern.**
In: Pattern languages of program design (PLoP) 4, pp. 15-32

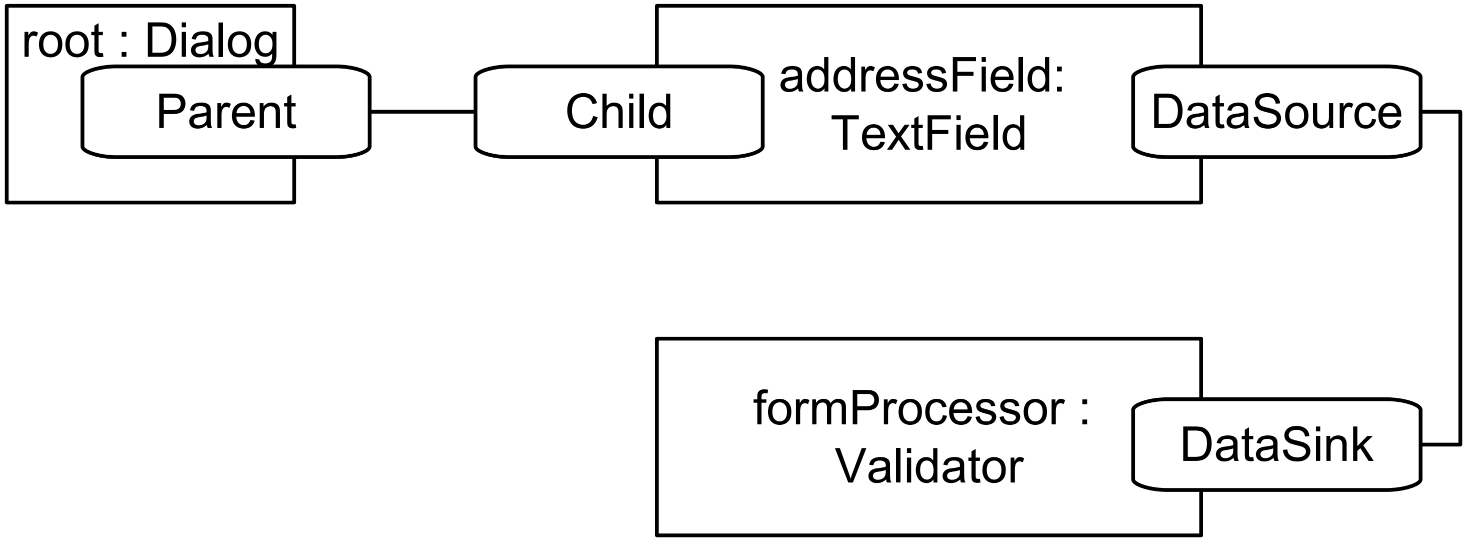
Purpose of Role-Object Pattern

- Transparently adapting objects to client context
- Management of role playership, where roles are represented as individual objects



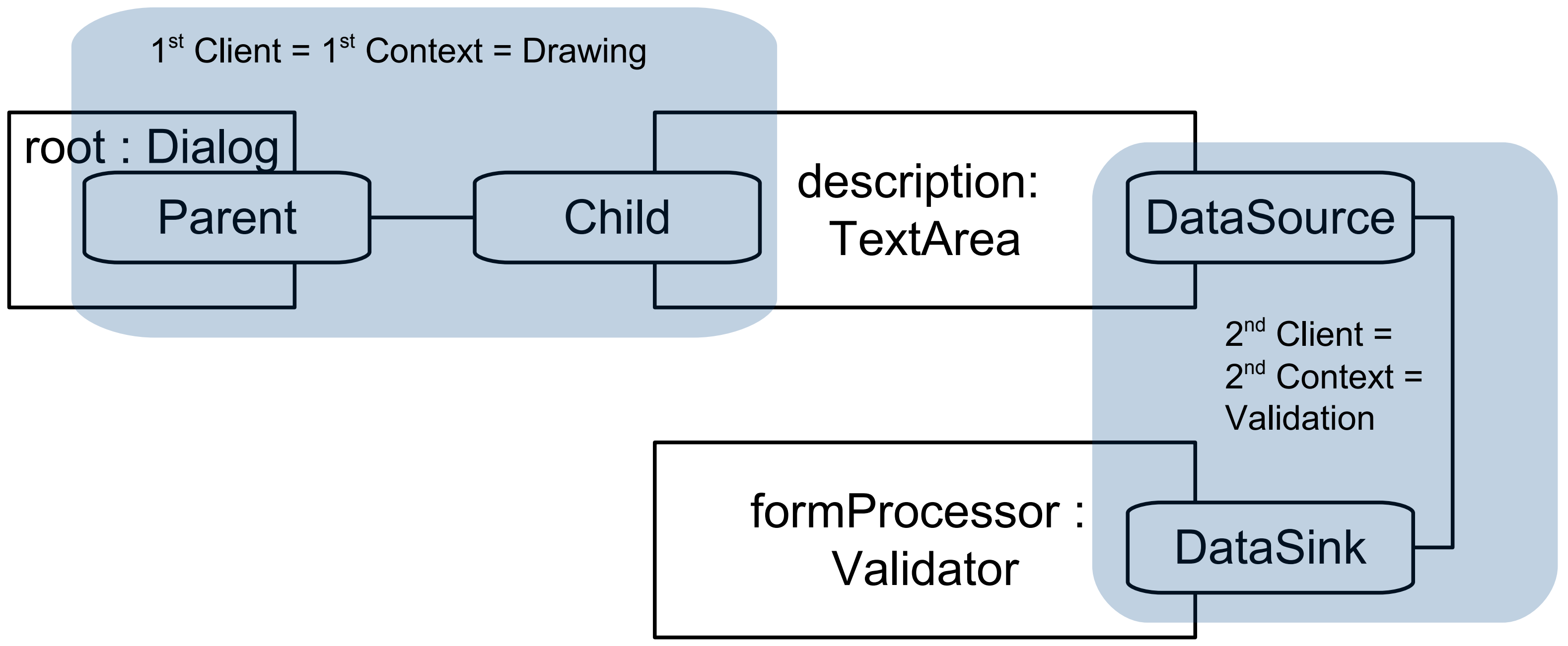
Purpose of Role-Object Pattern

- Transparently adapting objects to client context
- Management of role playership, where roles are represented as individual objects

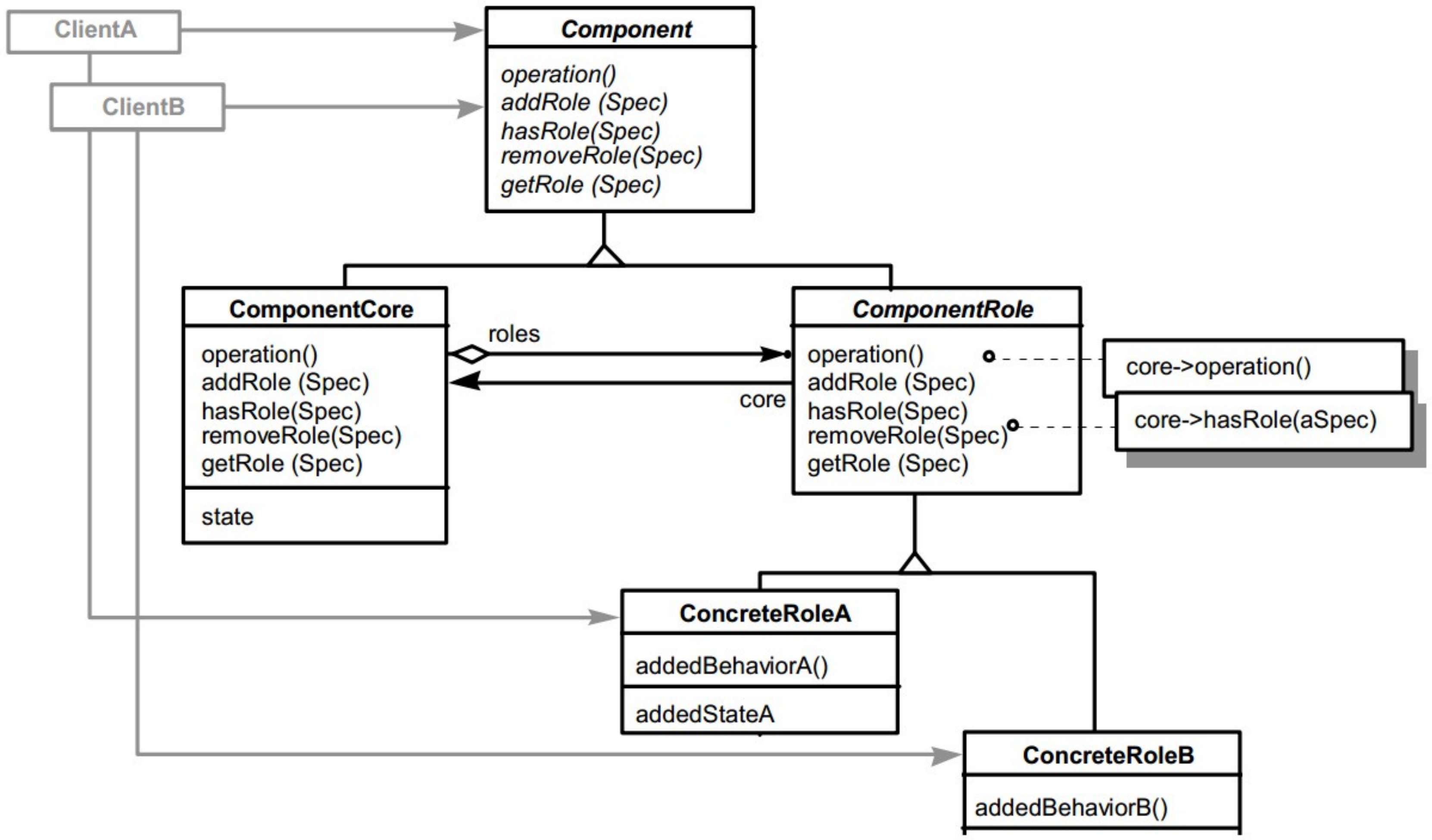


Purpose of Role-Object Pattern

- Transparently adapting objects to client context
- Management of role playership, where roles are represented as individual objects

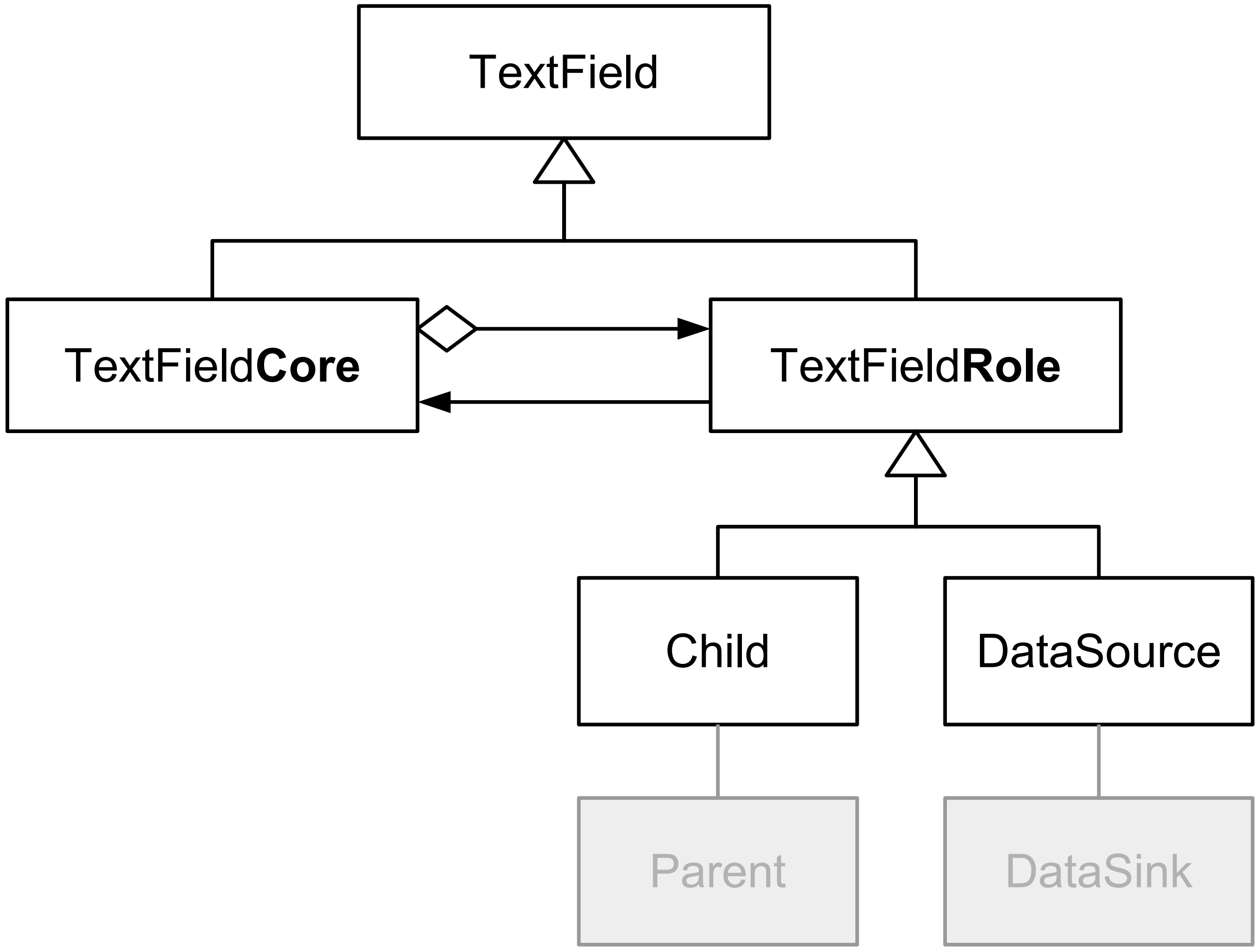


Structure of Role-Object Pattern

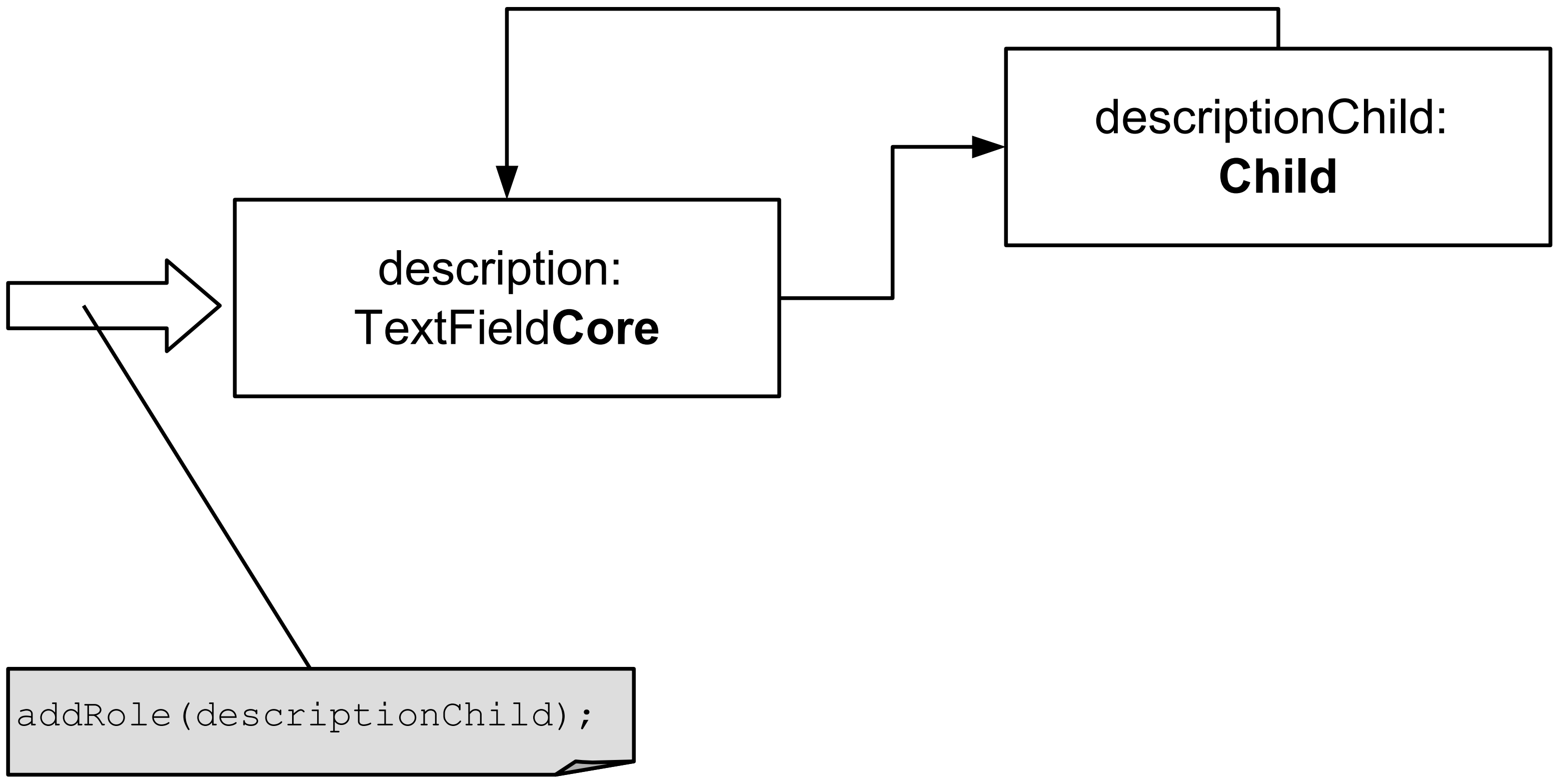


Dirk Bäumer, Dirk Riehle, Wolf Siberski, and Martina Wulf: **Role Object Pattern.**
In: Pattern languages of program design (PLoP) 4, pp. 15-32

Running Example: Email Checking

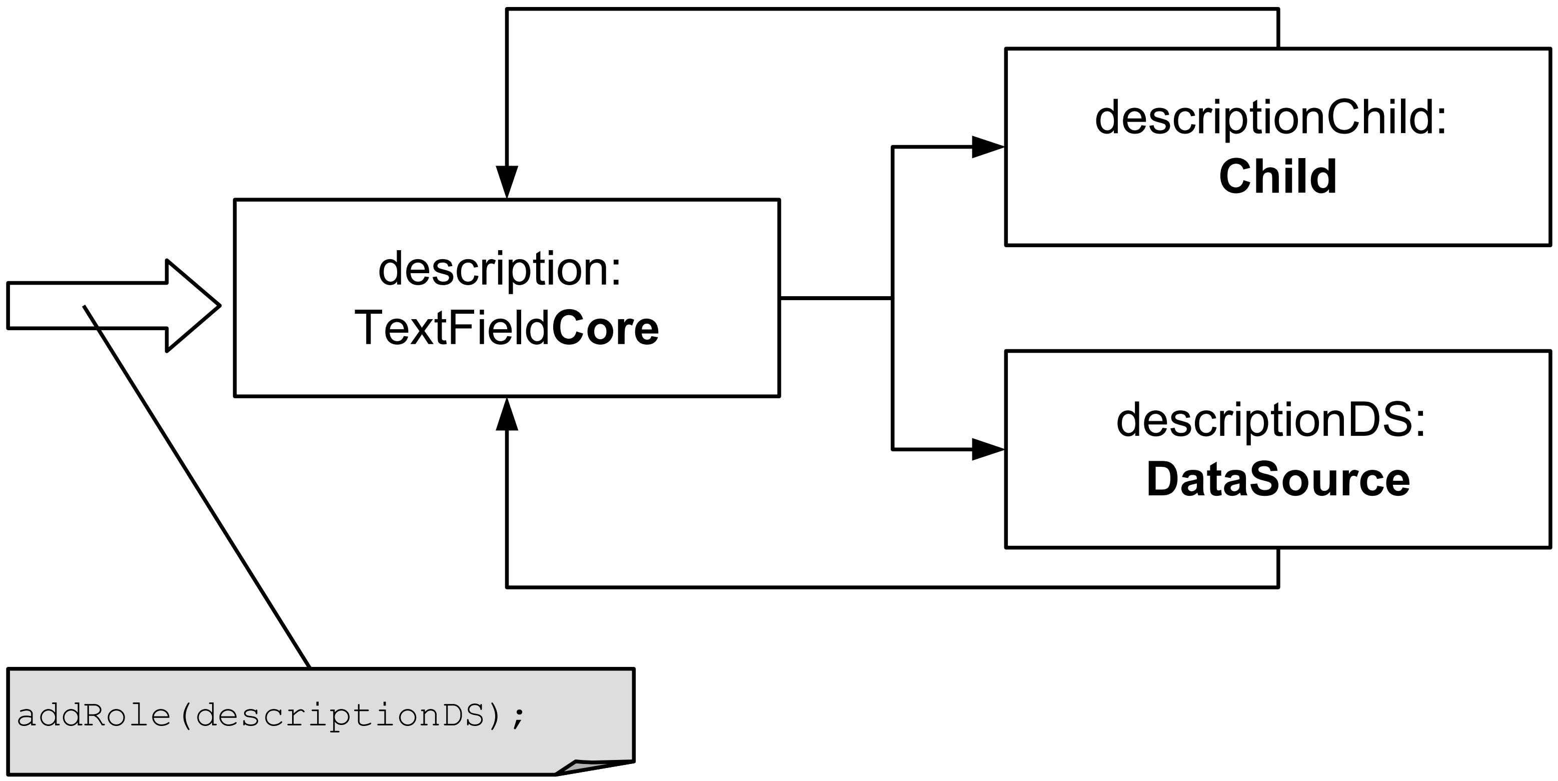


Running Example: Email Checking



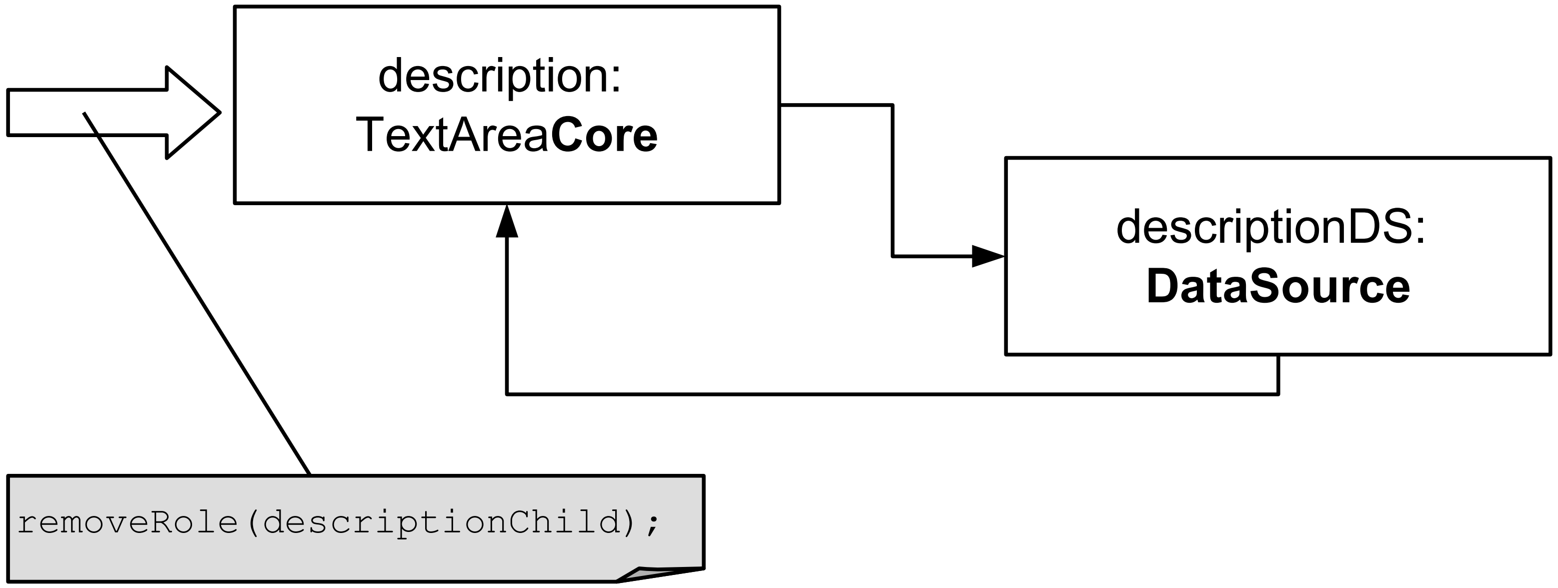
```
addRole(descriptionChild);
```

Running Example: Email Checking



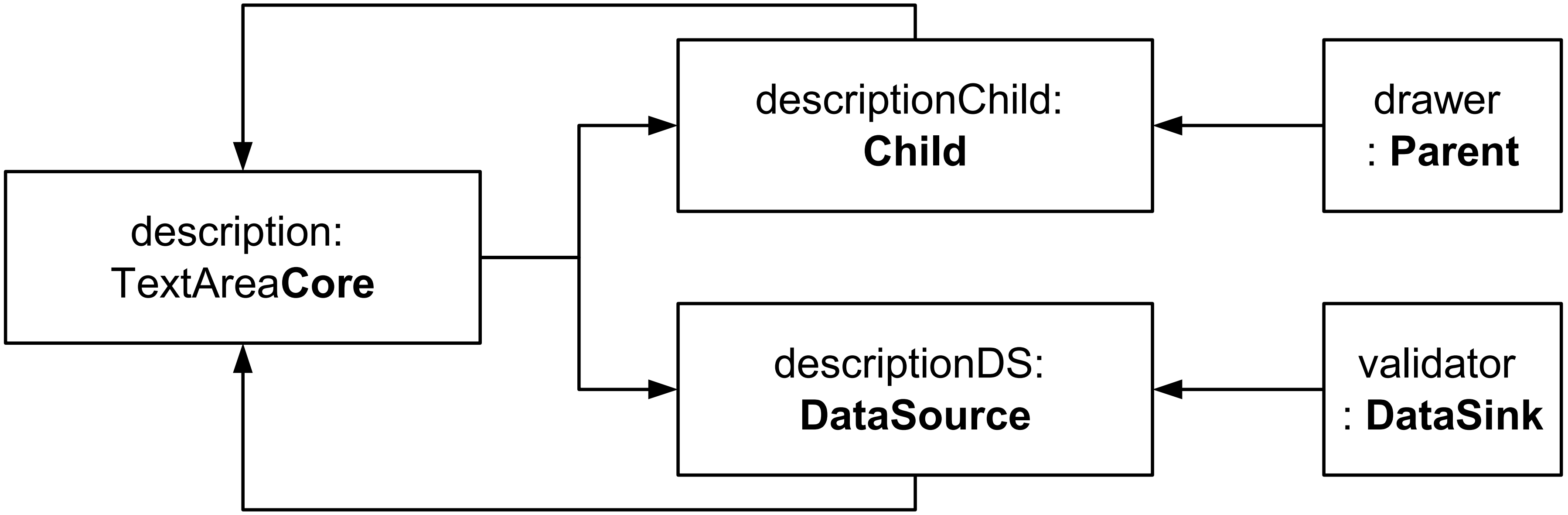
```
addRole(descriptionDS);
```

Running Example: Email Checking



```
removeRole(descriptionChild);
```

Running Example: Email Checking

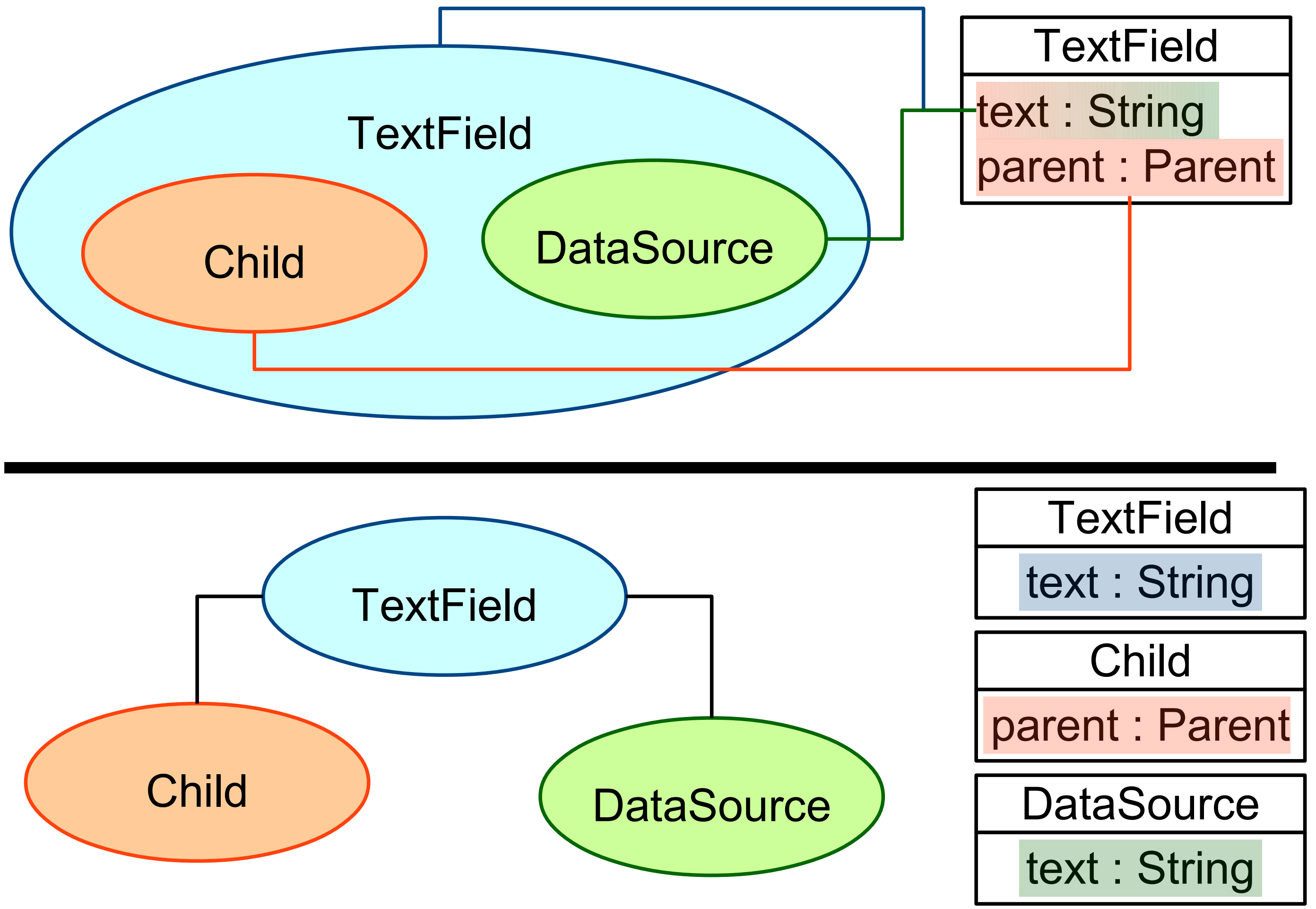


11.3 Object Schizophrenia

The Problem of Split Objects

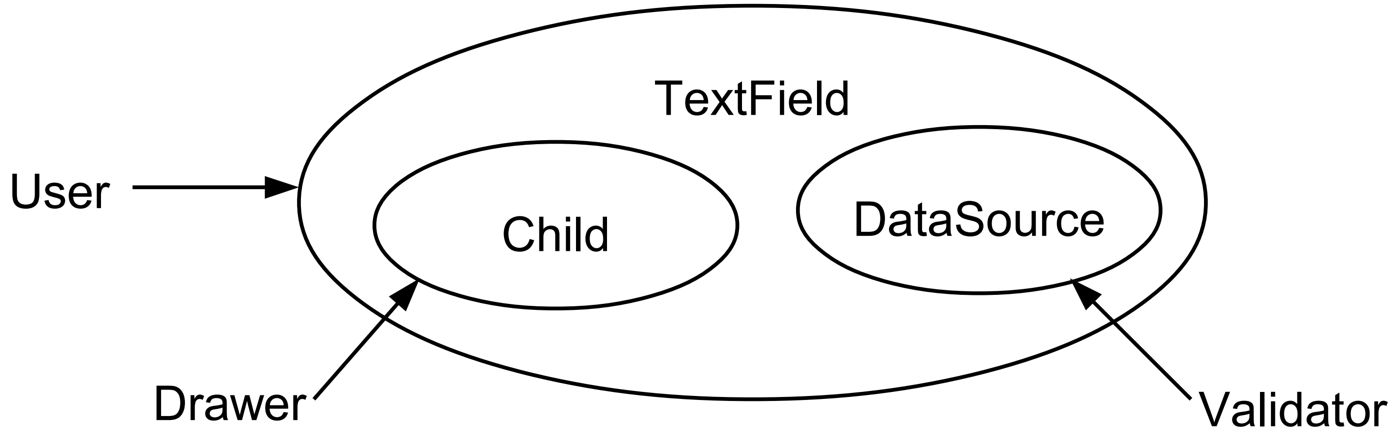
The problem of split objects

- Object schizophrenia covers the problems, which arise from splitting a conceptual object into multiple parts.



The problem of split objects

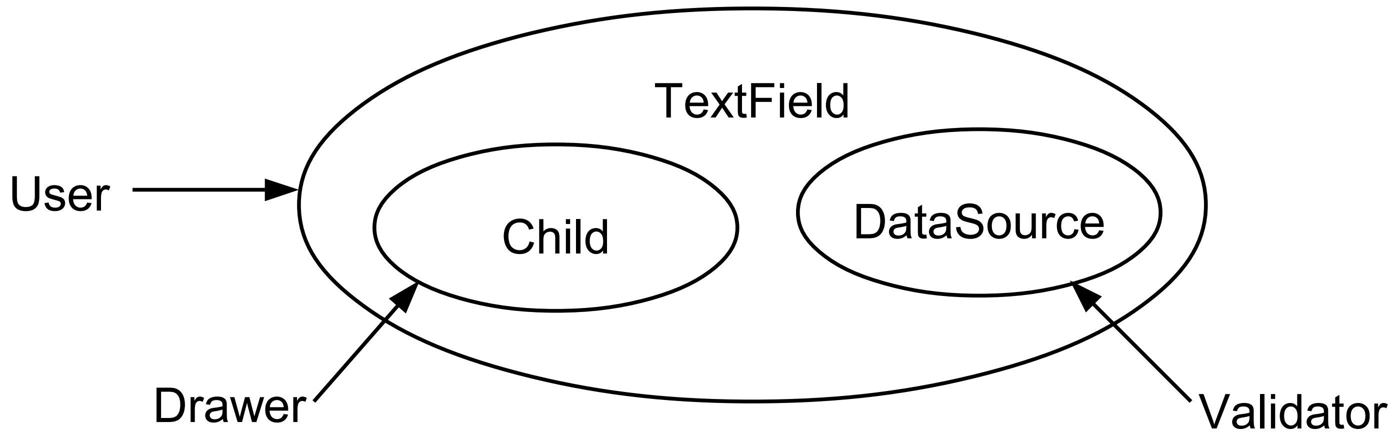
- Object schizophrenia covers the problems, which arise from splitting a conceptual object into multiple parts.



- Question for **identity** depends on which object is asked. “Who are you?”
  - User: “I'm a TextField.”
  - Drawer: “I'm the child of this parent.”
  - Validator: “I'm a data source for you.”

The problem of split objects

- Object schizophrenia covers the problems, which arise from splitting a conceptual object into multiple parts.



- Who manages the **state** of the compound object?
  - The text of the field is required for both roles
  - The size of the field is specific to the drawing task
  - The color of the text crosscuts both roles (drawing + validation)
- When should a role delegate to the player and when should a player delegate to its roles?

The ROP and Object Schizophrenia

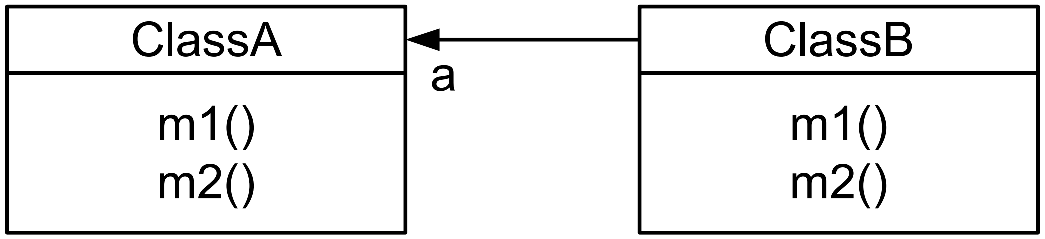
- Clients always have to “ask” the core object
- The core object delegates the call to the respective role
- The core object represents the identity
- But, all of this has to be implemented manually!
  - Role management code
    - AddRole, RemoveRole, Operation
  - Code for reflection
    - HasRole, GetRole
- Roles need to be implemented aware of their core
  - Delegation to core object for every method call, as it could be overridden by another role object, which is currently being played.

11.4 Delegation vs. Forwarding

The meaning of **this**

Delegation vs. Forwarding

- What does `this` or `self` actually mean?



```
class A {
  m1() { this.m2(); }
  m2() { print("A"); }
}
```

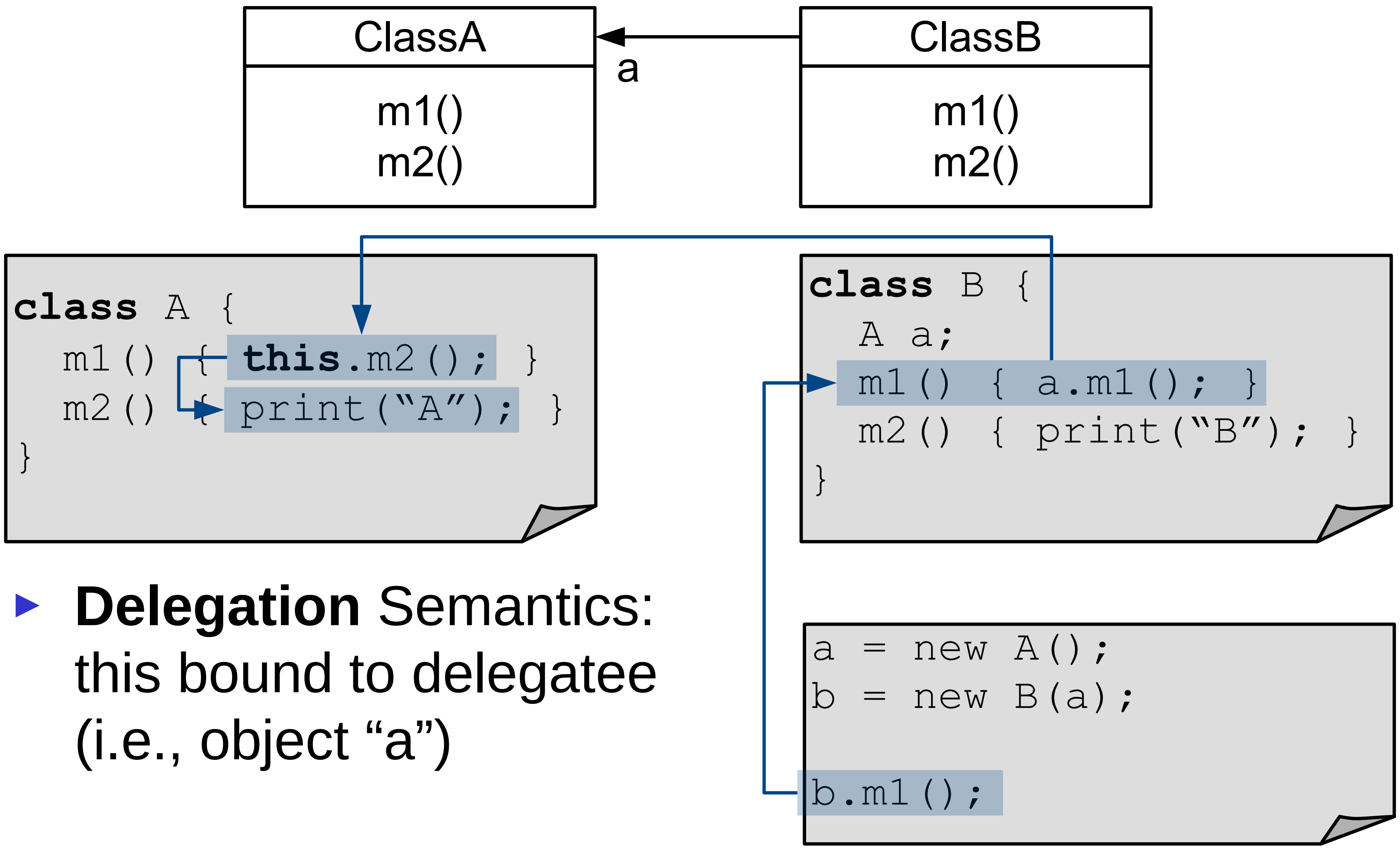
```
class B {
  A a;
  m1() { a.m1(); }
  m2() { print("B"); }
}
```

```
a = new A();
b = new B(a);

b.m1();
```

Delegation

- What does **this** or **self** actually mean?



```
class A {
   m1() { this.m2(); }
   m2() { print("A"); }
}
```

```
class B {
   A a;
   m1() { a.m1(); }
   m2() { print("B"); }
}
```

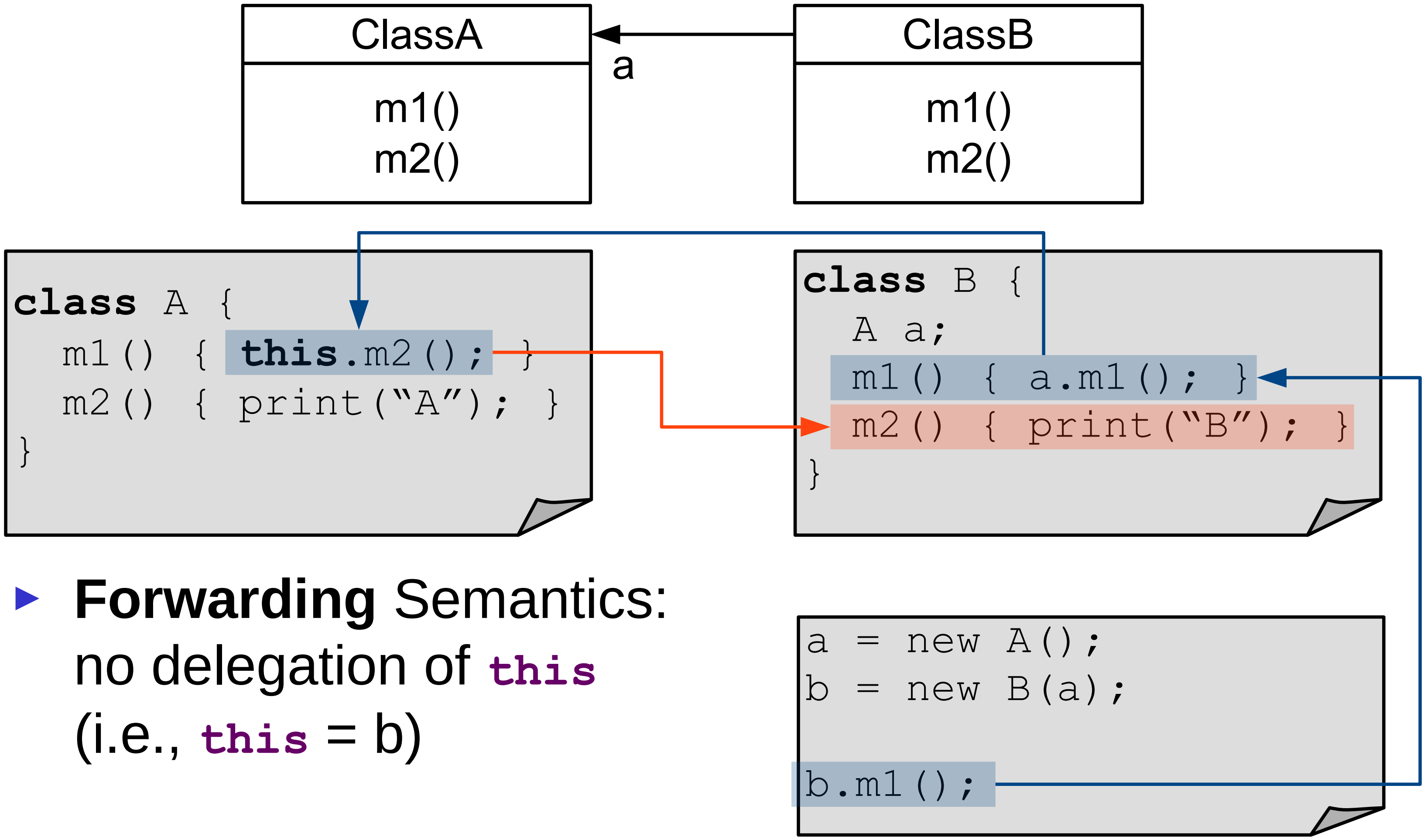
```
a = new A();
b = new B(a);

b.m1();
```

- **Delegation** Semantics: this bound to delegatee (i.e., object "a")

Forwarding

- What does **this** or **self** actually mean?



```
class A {
  m1() { this.m2(); }
  m2() { print("A"); }
}
```

```
class B {
  A a;
  m1() { a.m1(); }
  m2() { print("B"); }
}
```

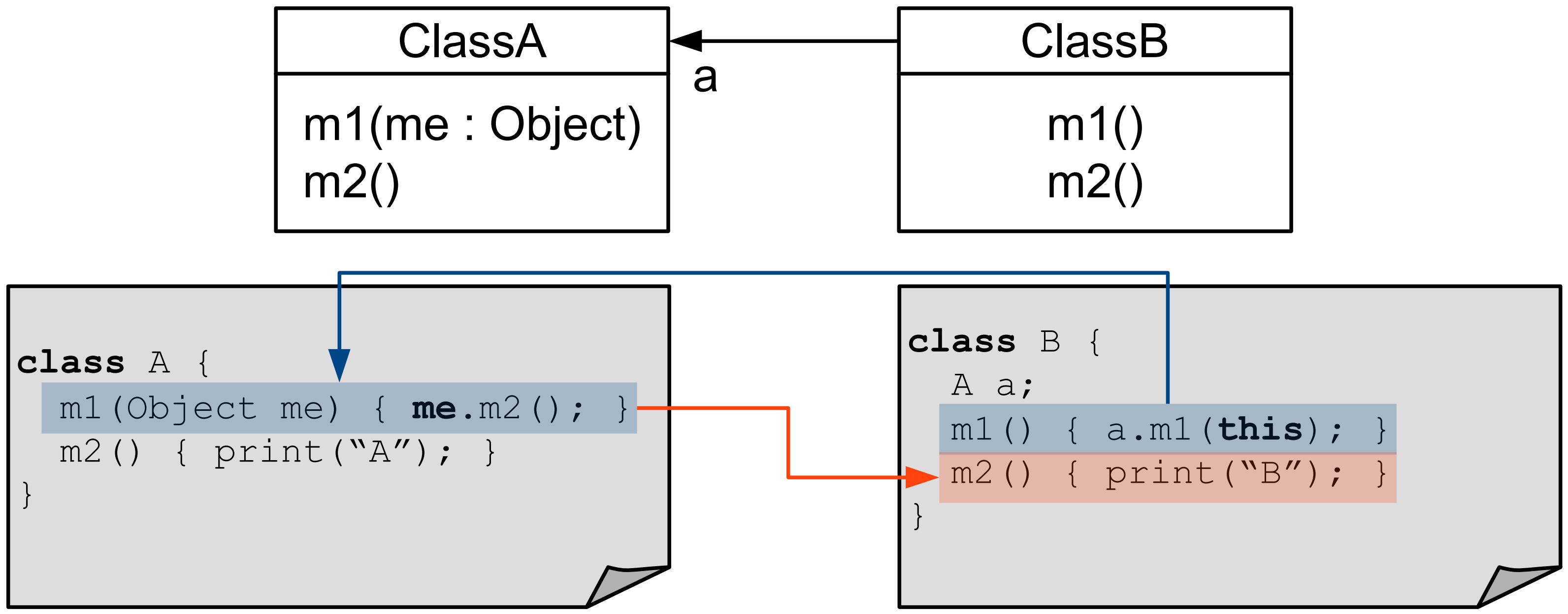
```
a = new A();
b = new B(a);

b.m1();
```

- **Forwarding** Semantics: no delegation of **this** (i.e., **this** = b)

Forwarding in Java

- Java does not directly support forwarding
- Workaround required
  - Passing `this` to the receiver
- Keep this in mind when implementing operations in the Role-Object Pattern!



```
class A {
  m1(Object me)  {  me.m2(); }
  m2()  { print("A"); }
}
```

```
class B {
  A a;
  m1()  {  a.m1(this); }
  m2()  {  print("B"); }
}
```

11.5 Roles Types Formally

How role types differ from other types.

Rigidity

- Role types differ from natural types in terms of rigidity
  - Natural types are rigid
  - Role types are non-rigid
- Instances of a rigid type, cannot stop being of this type without ceasing to exist
- Instances of a non-rigid type can!
  - You can stop being an employee without dying
    - Employee is a role type
  - You cannot stop being a human
    - Human is a natural type
- Instances of rigid types provide identity
- Instances of non-rigid types derive identity from players
- The non-rigidity property and the need for identity motivate the distinction of players and their roles

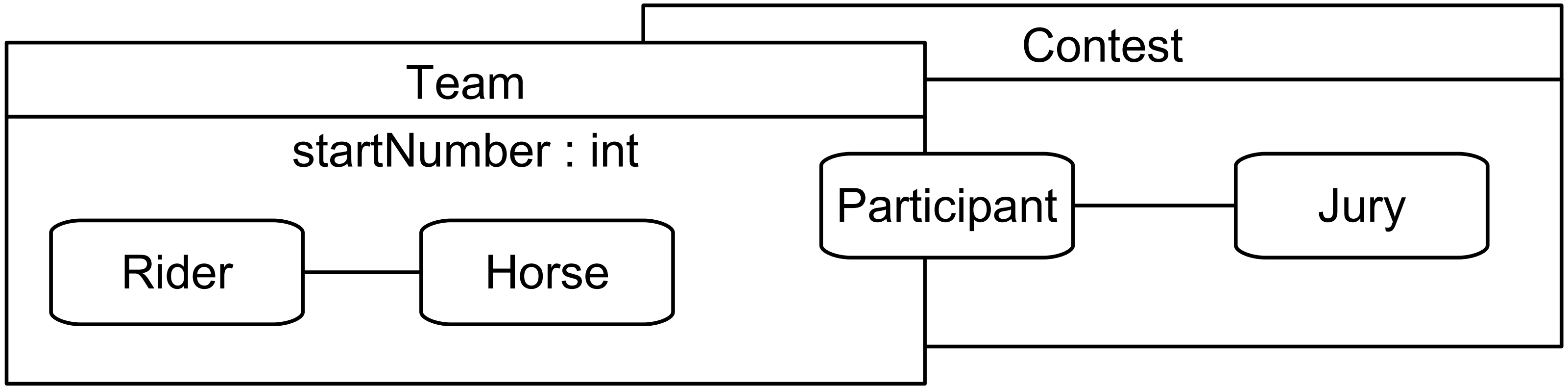
Foundedness

- Role types differ from natural types in terms of foundedness
  - Natural types are non-founded
  - Role types are founded
- Instances of a founded type, cannot exist on their own; they always need to be connected to another instance
  - Being a listener only works if there is a speaker
    - Listener is a founded type
  - Being a tree does not have such a constraint
    - Tree is a non-founded type
- Instances of founded types always require a counter-type against which they are defined
- The foundedness property of role types motivates the need for at least two role types forming a role model

Current Research on Role Types

	Non-Founded	Founded
Rigid	Natural types	Compartment Types
Non-Rigid	Phase types	Role types

- Phase types don't have an own identity (non-rigid), but do not depend on other types. They describe phases of an object.
  - For example, Child and Adult are phase types of Person
- Compartment types describe objectified collaborations



What have we learned?

- The Role-object Pattern
  - Realization of roles in object-oriented languages
  - Using delegation and forwarding
- Object Schizophrenia
  - Problem of identity
  - Problem of state management
- Formal properties of role types (and others)
  - Rigidity
  - Foundedness