A Precise Model for Contextual Roles: The Programming Language ObjectTeams/Java

Stephan Herrmann
Technische Universität Berlin

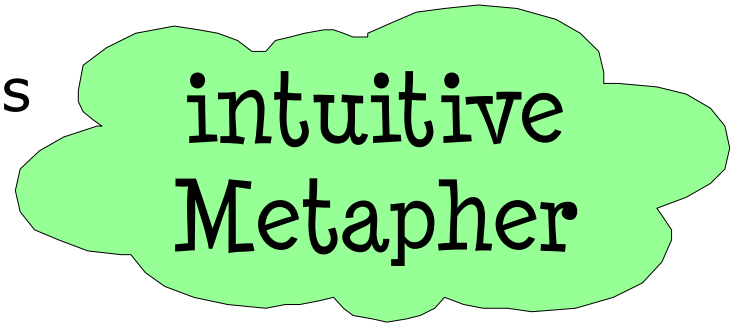
Dresden, 11.12.07

Thesen

Rollen haben sich für die Modellierung bewährt.

Rollenmodellierung ergänzt OO:

- dynamische Aspekte
  - Kollaborationen, Use-Cases
  - Analyse und Synthese
- relative Sichtweisen
  - Relationen, Subjektivität



Rollen sind auch für die Implementierung sinnvoll.

Rollenimplementierung weitgehend unbekannt

⇨ Mangelnde Durchgängigkeit für Rollenmodellierung

Also:

Eine Programmiersprache
mit direkter Unterstützung für Rollen
fördert das Rollenkonzept
bzgl. Modellierung *und* Implementierung

- Agenda:
  - Grundkonzepte für Rollen
  - Integration in objektorientierte Fundierung
  - Evaluation des präzisierten Rollenbegriffs

Object Teams

- Programmiermodell Object Teams
  - Umsetzung/Verfeinerung von
    - Aspectual Components [Mezini+ 1999]
    - Family Polymorphism [Ernst 2001]
    - Rollen (”Aspects”) [Richardson&Schwarz 1991]
- Programmiersprache ObjectTeams/Java (OT/J)
  - Werkzeugentwicklung seit Ende 2001
  - **O**bject **T**eams **D**evelopment **T**ooling seit 2003
  - §§ Sprachdefinition Version 1.0: Juli 2007
  - Einsatz:
    - Fallstudien (Projekt TOPPrax), Lehre, OTDT (OT/Equinox)

Grundkonzepte für Rollen

- Rollen brauchen Kontext
  - Team Klassen
    - definieren Kollaboration interagierender Rollen
    - OT/J: Schlüsselwort `team`

```
public team class FussballTeam {
    public class Stürmer playedBy Person { }
    public class Torwart playedBy Person { }
    public class Trainer playedBy Person { }
}
```

- Rollen werden gespielt
  - PlayedBy Relation
    - OT/J: Schlüsselwort `playedBy`
    - Rolle – Basis: Klassendeklaration bindet alle Instanzen

Objektschizophrenie?

Klassische Modellierung: `Stürmer extends Person`

- Ein Objekt
- Wenn der Stürmer nicht mehr stürmen kann,
  muß die Person gelöscht werden.

Rollenmodellierung: `Stürmer playedBy Person`

- Zwei verbundene Objekte
- Fragen:
  - Wie finde ich die Rollen, die eine Person spielt?
  - Wie finde ich die Person,
    von der die gegebene Rolle Stürmer gespielt wird?
  - Ist ein Stürmer eine Person?
    Ist eine Person ein Stürmer?

Substitutability

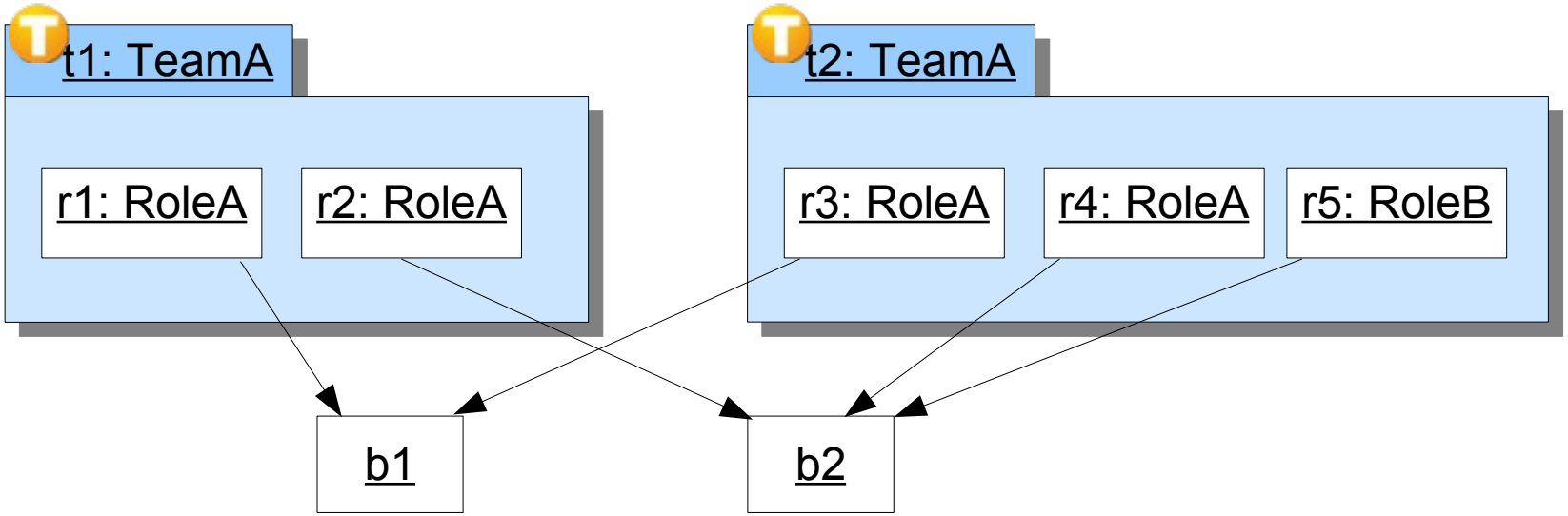
- Gilt Subtype-Polymorphie für Rollen?
- Sind einer Rolle und ihre Basis dasselbe Objekt?
  - Antwort: fast!
  - Substitutability ist möglich, erfordert eine Übersetzung
- **Translation Polymorphism**

```
Trainer trainer= new Trainer(new Person());
Person person= trainer;  // erlaubt!
trainer= person;         // erlaubt?
```

  - Jede Rolle hat genau ein Basisobjekt
  - Zuweisung ignoriert die Rolle, weist das Basisobjekt zu
  - **Lowering**

Substitutability (2)

- Translation base → role: **Lifting**
- Eine Basis kann viele Rollen haben,



- lift(b1, t1) → r1, lift(b1, t2) → r3
- lift(b2, t2, RoleA) → r4, lift(b2, t2, RoleB) → r5

Lifting und Lowering

- Wann?
  - Bei Überschreitung der Teamgrenze

```
public team class FussballTeam {
    protected class Stürmer playedBy Person { }
    Stürmer ms;
    public void anwerben (Person as Stürmer s) { ms= s; }
    public Person getMittelStürmer()           { return ms; }
}
```

- Fazit:
  - Rolle kann außerhalb des Teams unsichtbar sein
  - Basisobjekte können innerhalb des Teams unsichtbar sein

Lifting

- Welches Objekt?
  - Existierende Rolle wird wiederverwendet (Rollen haben eigenen Zustand).
  - Neue Rolle wird bei Bedarf erzeugt.
- Rollenerzeugung
  - implizit durch Lifting
  - explizit, verwendet spezielle Konstruktoren
- Lifting im Kontext von Vererbung
  - **smart lifting** findet stets die best-passende Rolle vgl. Entwurfsmuster Visitor, double-dispatch:
    - Methodendispatch
    - Rollendispatch

Lifting

- Welches Objekt?
  - Existierende Rolle wird wiederverwendet (Rollen haben eigenen Zustand).
  - Neue Rolle wird bei Bedarf erzeugt.
- Rollenerzeugung
  - explizit, verwendet spezielle Konstruktoren
- Lifting im Kontext von Vererbung
  - **smart lifting** findet stets die best-passende Rolle vgl. Entwurfsmuster Visitor, double-dispatch:
    - Methodendispatch
    - Rollendispatch

Lifting & Lowering = Translation Polymorphism

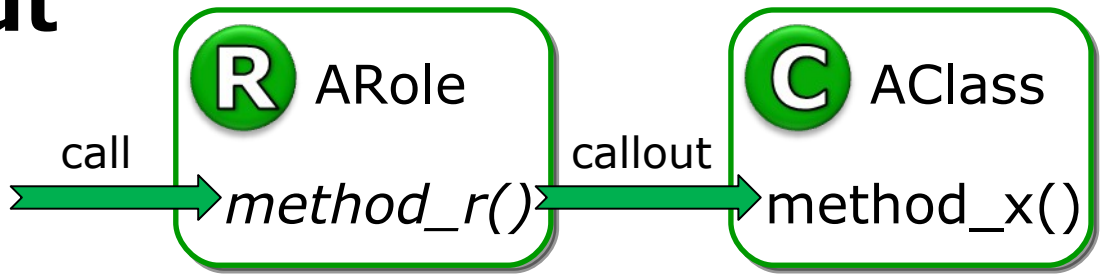
Rollen und Vererbung

Vererbung = Subtyping + Import + Overriding

- Subtyping ~ Translation Polymorphism
  - Rollen/Basen austauschbar
- Import:
  - Hat eine Rolle alle Eigenschaften ihrer Basis?
- Overriding:
  - Kann eine Rolle Eigenschaften ihrer Basis überschreiben?

Import

- Weiterleitung von Aufrufen: Rolle ➞ Basis
- "**Callout**"



  - Wiederverwendung
  - Restrukturierung
  - Brechen der Kapselung

```
public /* role */ class Dozent playedBy Professor {
    public abstract String getAnrede(); // expected method
    getAnrede -> getVorname; // connect to provided method
}
```

Callout Details (1)

Alternative Schreibweise

```
String getAnrede() -> String getVorname();
```

- Nötig bei overloading in der Basisklasse
- Erspart die Deklaration einer abstrakten Methode

Callout to Field

```
String getAnrede() -> get String vorname;
```

- Führt extern Getter-/Setter-Methode ein

Parameter Mapping

```
List<Bar> getBars(Integer i) -> Bar[] getBarArray(int i)
    with {     i.intValue()  -> i,
                      result  <- Arrays.asList(result) }
```

- Erlaubt Anpassungen zw. inkompatiblen Signaturen

Callout Details (2)

Lowering & Lifting

```
public /* role */ class Student playedBy Person {
    void defineFellow(Student student) -> void makeFriend(Person p);
    Student[] getFellows               -> Person[] getFriends();
}
```

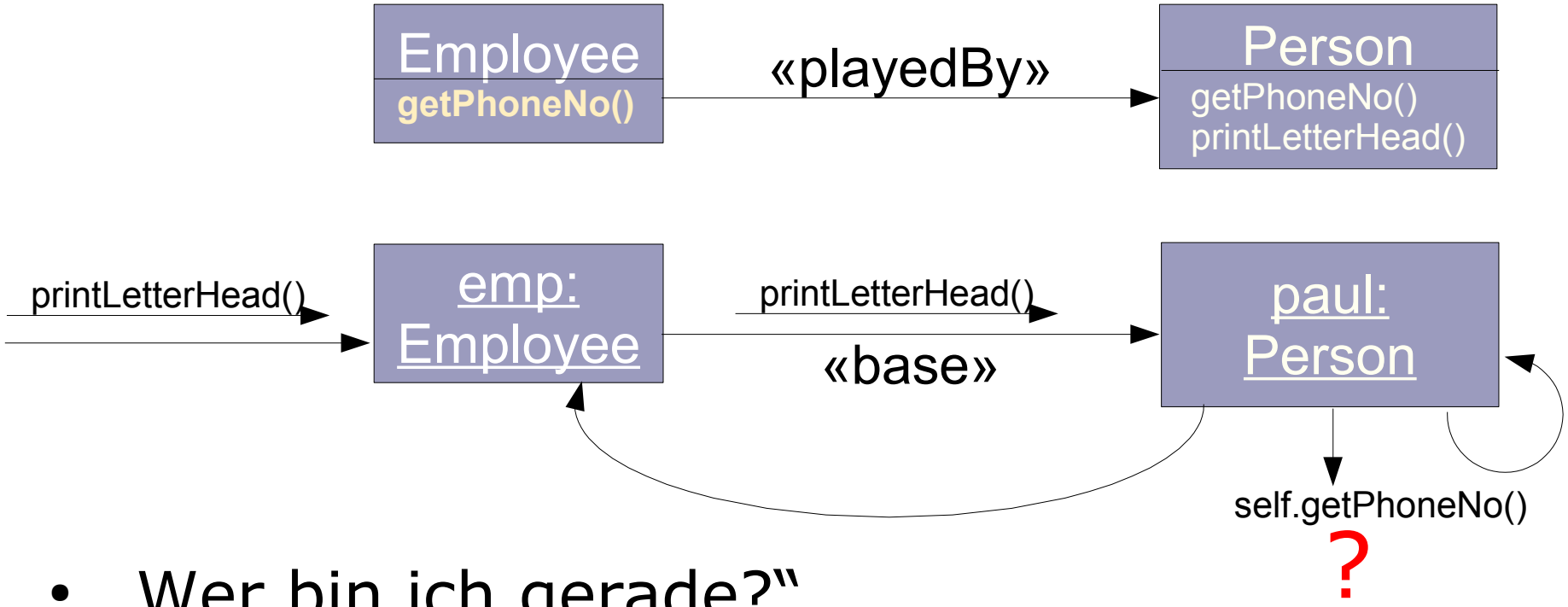
- Trennung von Innen- und Außenwelt

Decapsulation

- Referenzierte Basismethoden/-felder müssen nicht sichtbar sein
  - Rolle/Basis gilt als eine Einheit
- Warning, @SuppressWarnings(”decapsulation”), sealing ...

Callout = Import = Forwarding

Grenzen des Forwarding



- „Wer bin ich gerade?“
  - **Forwarding** = complete transfer of control
    Result: private phone number ☹
  - **Delegation** = Methodlookup at *base* without changing of *self*
    Result: office phone number ☺

Deklariertes Overriding

```
public class Person {
    public String getPhoneNumber() { ... }
    public void printLetterHead()  { ...getPhoneNumber()... }
}
public /*role*/ class Employee playedBy Person {
    callin String getPhoneNumber() { ... }
    void printLetterHead()  -> void printLetterHead();
}
```

- getrennte Namensräume für Rolle/Basis
  - Callout kann umbenennen
  - gleicher Name bedeutet noch *kein* Overriding

Deklariertes Overriding

```
public class Person {
    public String getPhoneNumber() { ... }
    public void printLetterHead()  { ...getPhoneNumber()... }
}
public /*role*/ class Employee playedBy Person {
    callin String getPhoneNumber() { ... }
    String getPhoneNumber() <- replace String getPhoneNumber();
    void printLetterHead()  -> void printLetterHead();
}
```

- getrennte Namensräume für Rolle/Basis
  - Callout kann umbenennen
  - gleicher Name bedeutet noch *kein* Overriding
  - Overriding **deklariert** durch **Callin-Bindung**

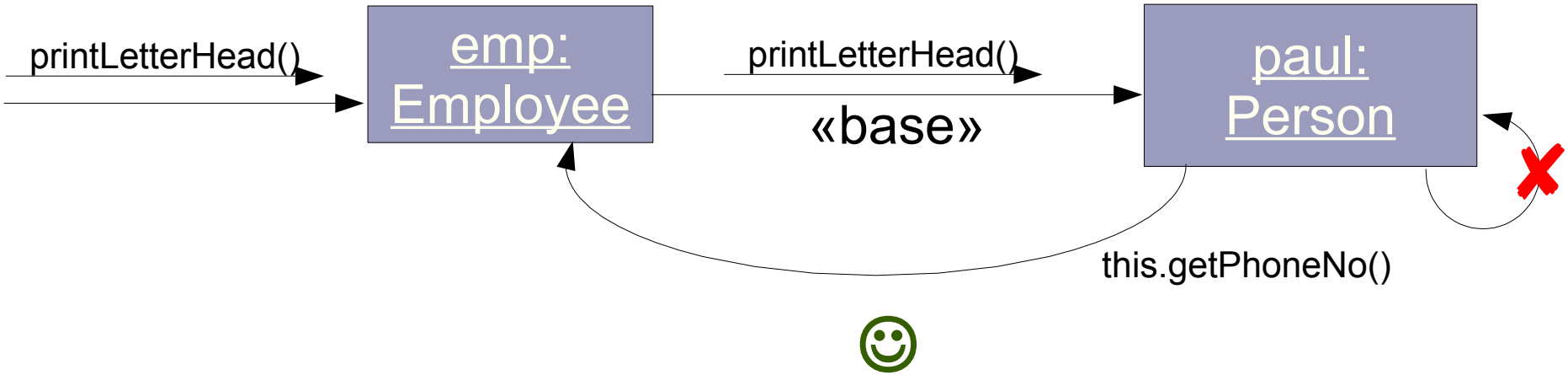
Implementierung

```
public /*role*/ class Employee playedBy Person {
    callin String getPhoneNumber() {
        String priv= “private: “ + base.getPhoneNumber();
        String offi= “office:  “ + this.phoneNumber;
        return priv + ”\n” + offi;
    }
    String getPhoneNumber() <- replace String getPhoneNumber();
    void printLetterHead()  -> void printLetterHead();
}
```

- callin-Methode implementieren:
  - (fast) ganz normale Methode
  - **base**-call entspricht super-call bei Vererbung

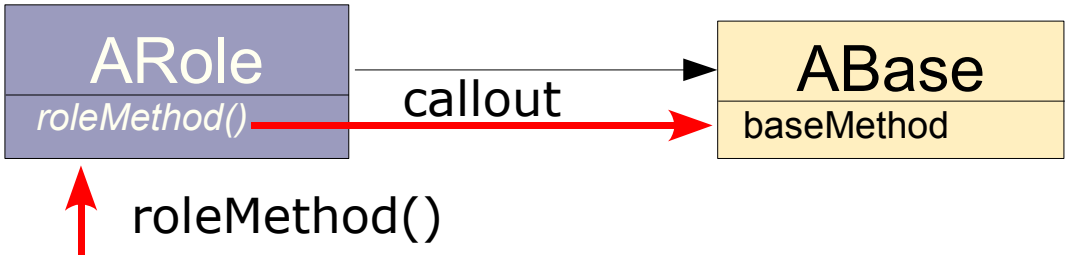
Delegation, indeed

```
public class Person {
    public String getPhoneNumber() { ... }
    public void printLetterHead()  { ...getPhoneNumber()... }
}
public /*role*/ class Employee playedBy Person {
    callin String getPhoneNumber() { ... }
    String getPhoneNumber() <- replace String getPhoneNumber();
    void printLetterHead()  -> void printLetterHead();
}
```

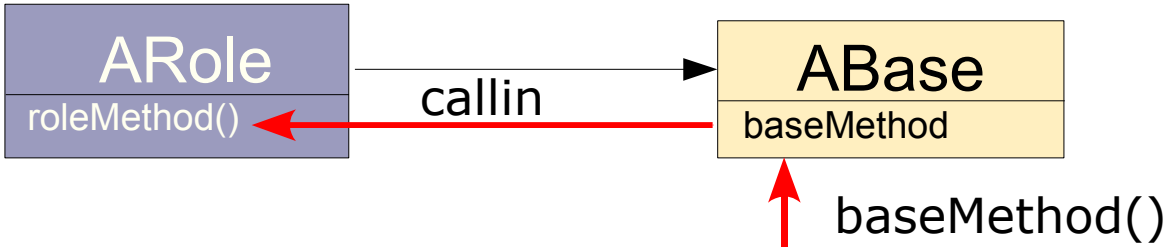


Aus 2 mach 3

- 2 Mechanisms, 3 styles of dispatch
  - **Forwarding**



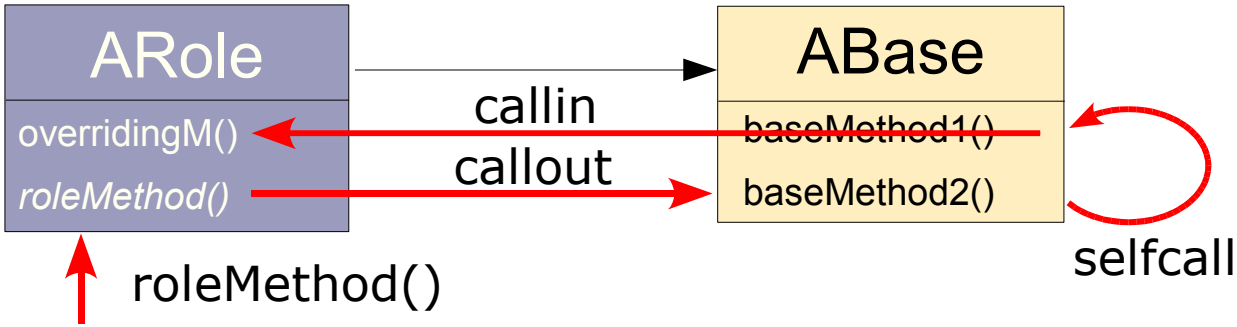
  - **Interception**



  - **Delegation w/ Overriding**

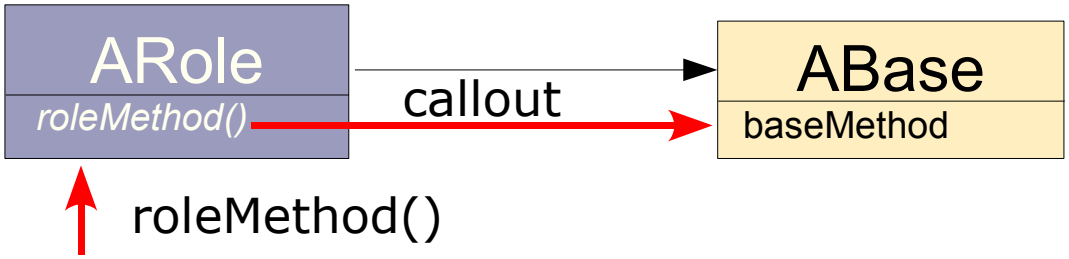
    = Forwarding

    + Interception



Aus 2 mach 3

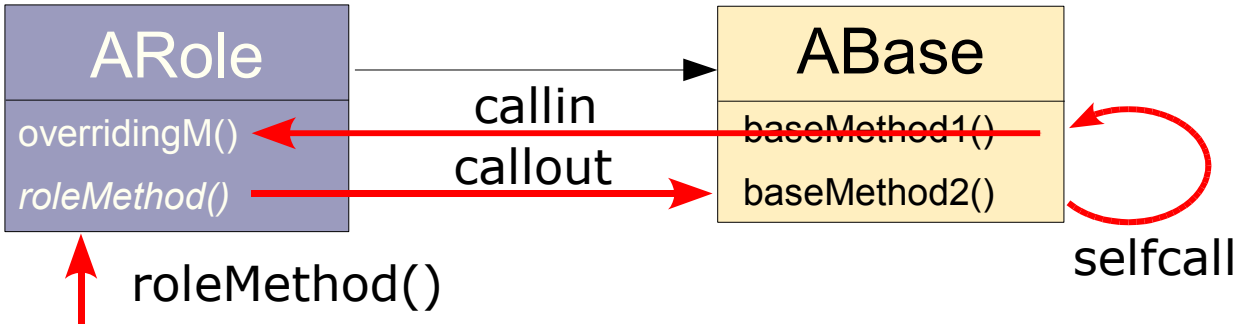
- 2 Mechanisms, 3 styles of dispatch
  - **Forwarding**



Callin = Interception = Overriding

  - **Delegation w/ Overriding**

    = Forwarding
    + Interception



extends vs. playedBy

- **Subtyping**
  - Ersetzbarkeit
    - up cast: sicher
    - down cast: unsicher
      - ev. ClassCastException
- **Import**
  - Implizit
  - Vollständig bis auf `private`
- **Overriding**
  - Implizit, per Namen
  - `@Override`
  - dynamisches Binden
  - Template&Hook

- **Translation Polymorphism**
  - Ersetzbarkeit
    - **lowering**: sicher
    - **lifting**: sicher
      - ev. Rollenerzeugung
- **Import**
  - Explizit durch **callout**
  - Selektiv inkl. `private`
- **Overriding**
  - Explizit durch **callin**
  - dynamische Binden
    - ev. Rollenerzeugung
  - Template&Hook
  - Method Call Interception

extends vs. playedBy

- **Subtyping**
  - Ersetzbarkeit
  - up-cast sicher
  - @Override
  - dynamisches Binden
  - Template&Hook

- **Translation Polymorphism**
  - Ersetzbarkeit
  - lowering sicher
  - dynamische Binden
    - ev. Rollenerzeugung
  - Template&Hook
  - Method Call Interception

PlayedBy =

Vererbung

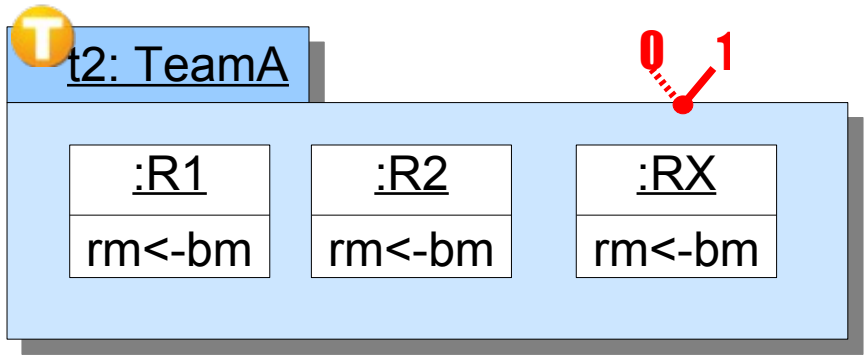
+ entkoppelte Namensräume

+ Multiplizitäten

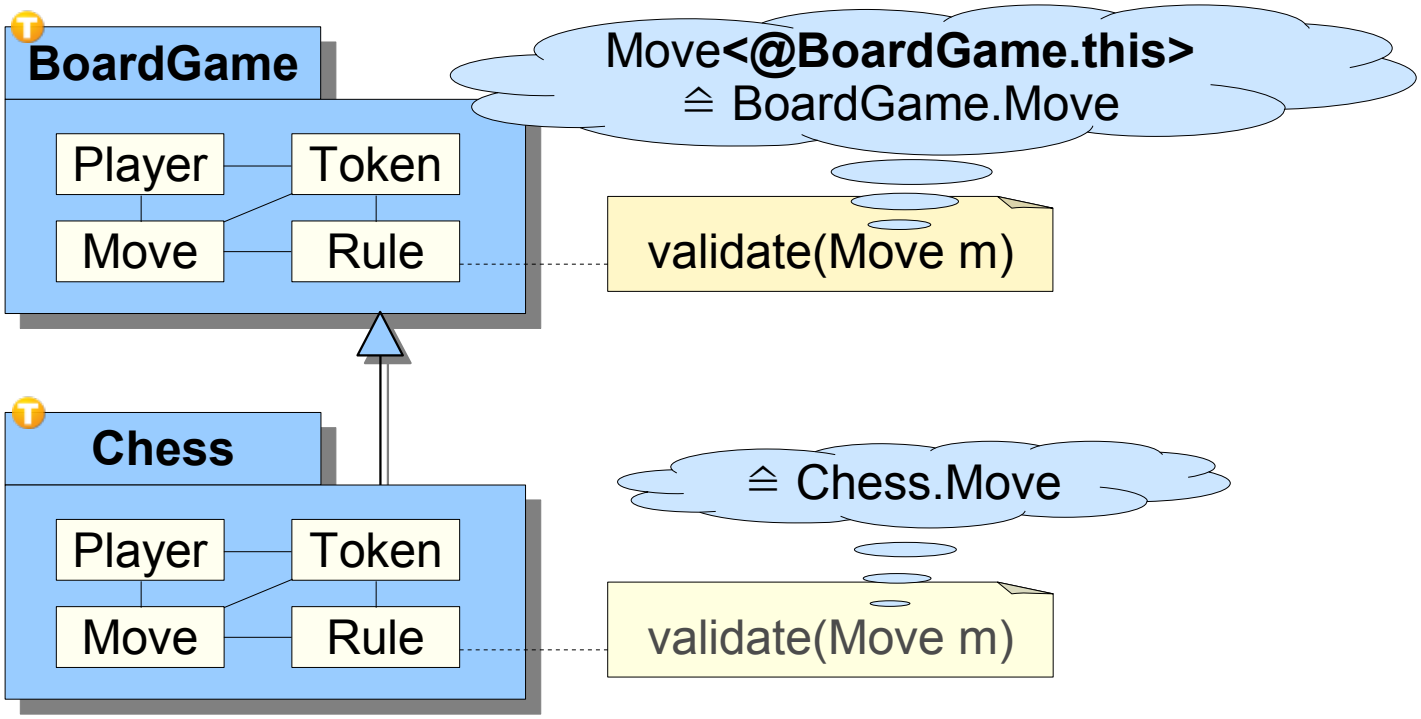
+ Dynamik

Was tun Teams?

- Team verwalten Abbildung Rolle↔Basis
  - Lifting/Lowering findet an der Teamgrenze statt
- Team kapseln ihre Rollen
  - protected roles, confined roles, opaque roles
- Teams steuern callin-Bindungen
  - callins müssen durch Aktivierung des Teams eingeschaltet werden
  - Programm-Modi situatives Verhalten
  - diverse Mechanismen

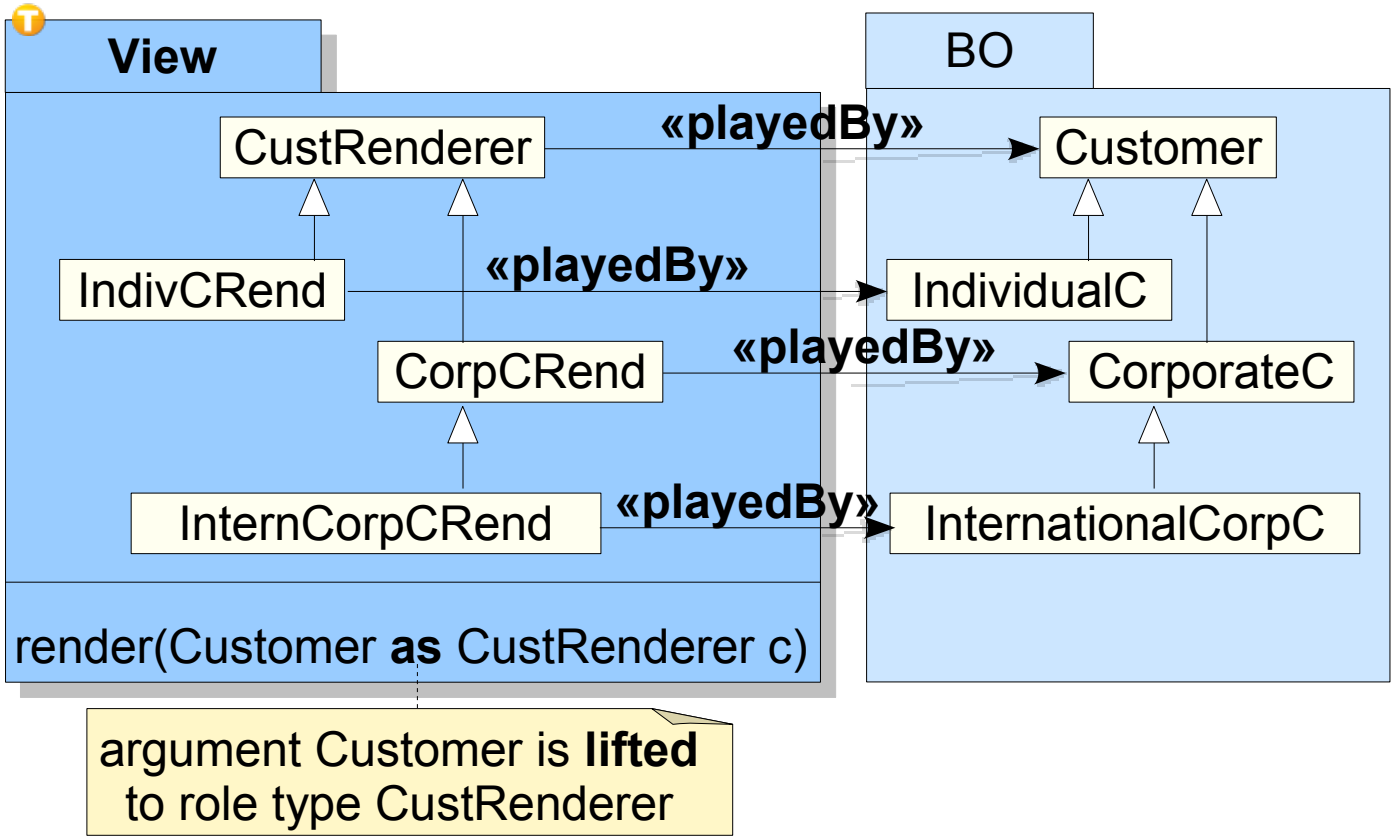


Teamvererbung



- Typsichere Kovarianz
  - dank Family Polymorphism [Ernst 2001]
  - virtual classes, dependent types
  - **new** ist sogar für abstrakte Klassen verwendbar

Adaptierung einer Hierarchie



- smart lifting
  - Hierarchie von Rollen adaptiert Hierarchie von Basen

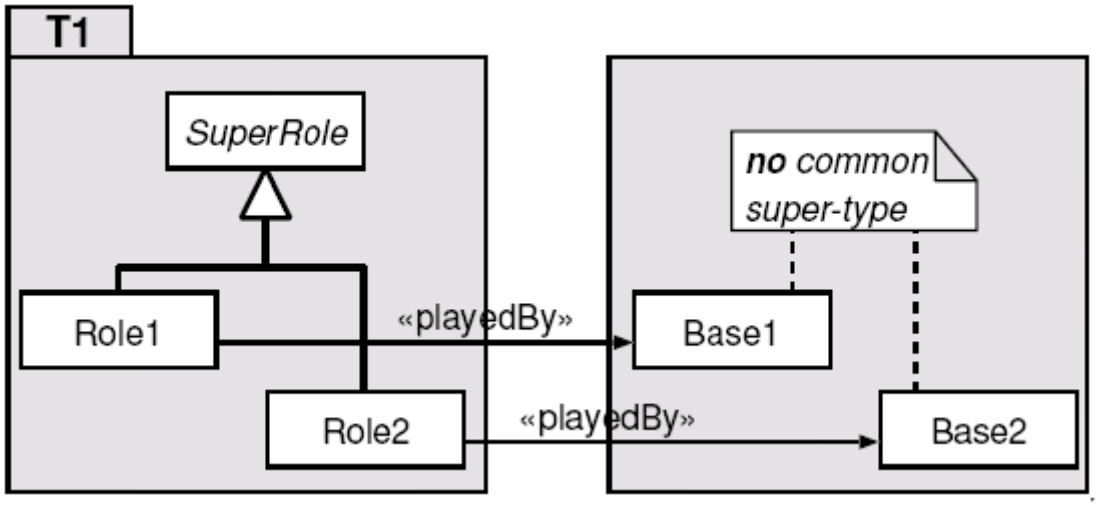
Einordnung – Steimanns Kriterien

- Multiplizitäten (4 Kriterien)
- Features: Zustand und Verhalten (5 Kriterien)
- Dynamik (3 Kriterien)
- Sonstige (3 Kriterien)

Multiplizitäten

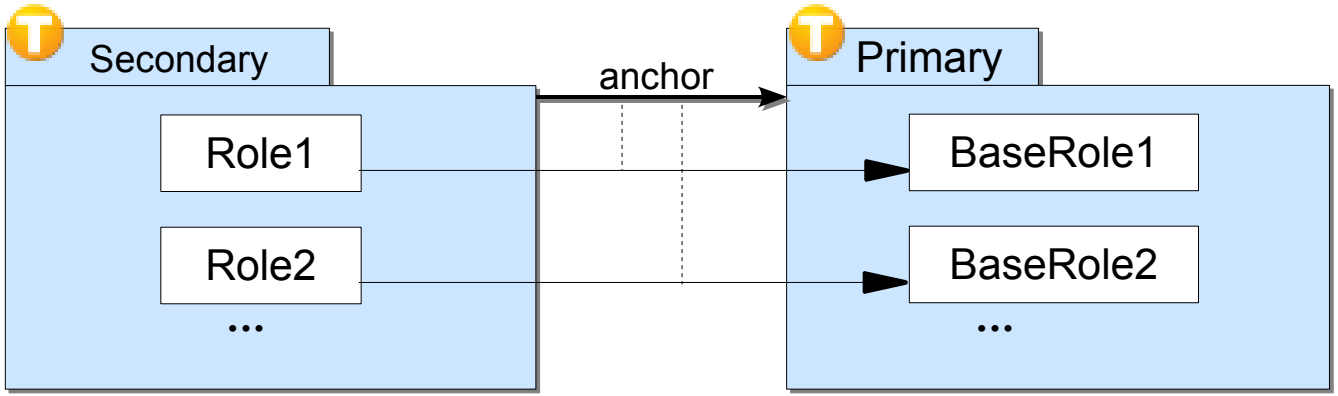
in verschiedenen Teams

- Ein Objekt kann mehrere Rollen gleichzeitig spielen.
- Ein Objekt kann mehrfach gleichzeitig die gleiche Rolle spielen.
- Objekt unabhängiger Typen können die gleiche Rolle spielen.
- Rollen können Rollen spielen.



Multiplizitäten

- Ein Objekt kann mehrere Rollen gleichzeitig spielen.
- Ein Objekt kann mehrfach gleichzeitig die gleiche Rolle spielen.
- Objekt unabhängiger Typen können die gleiche Rolle spielen.
- Rollen können Rollen spielen.



Features: Zustand und Verhalten

- Eine Rolle hat eigene Eigenschaften und Verhaltensweisen.
- Der Zustand eines Objektes kann rollenspezifisch sein.
- Funktionen eines Objektes können rollenspezifisch sein.
- Rollen beschränken Zugriff.
- Verschiedene Rollen können Strukturen und Verhalten teilen.

Dynamik

aTeam.unregisterRole(aRole);

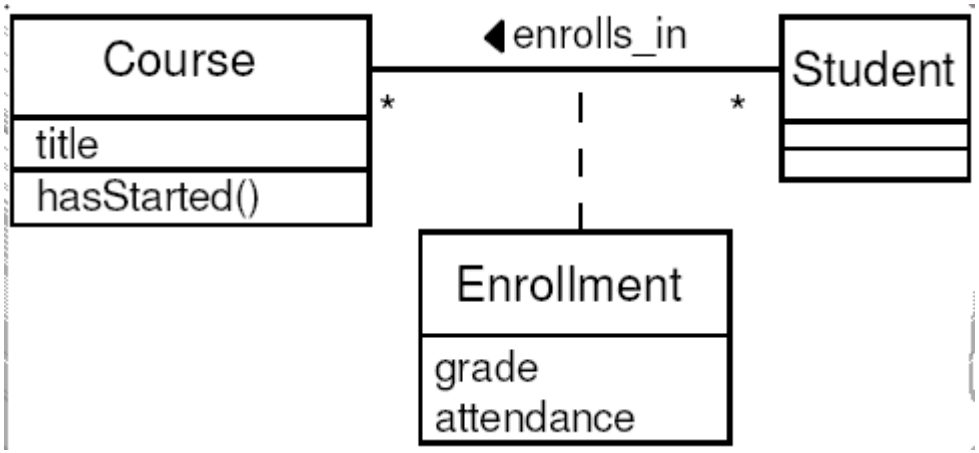
- Ein Objekt kann Rollen dynamisch annehmen und ablegen.
- Die Reihenfolge, in der Objekt Rollen annehmen und ablegen kann Beschränkungen unterworfen sein.
- Eine Rolle kann von einem Objekt zu einem anderen transferiert werden.

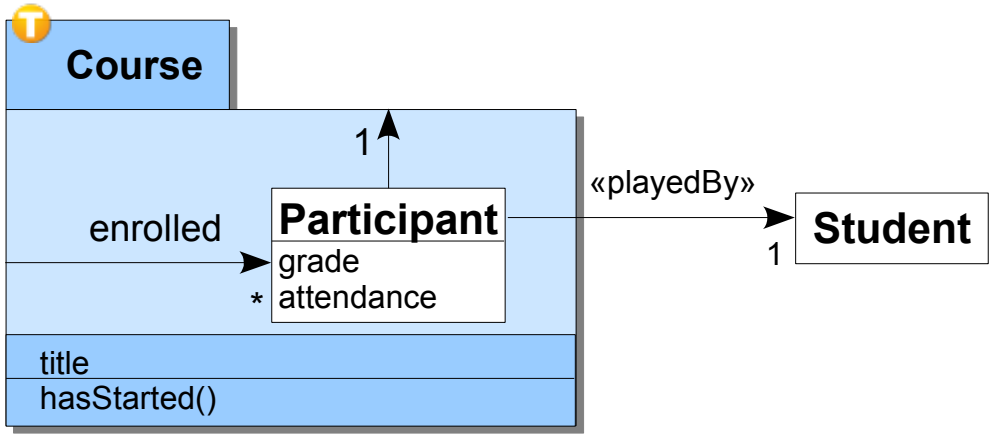
Guardprädikate

derzeit **nicht möglich**:
- base-link ist `final`
- `non-null` würde ausreichen

Sonstige

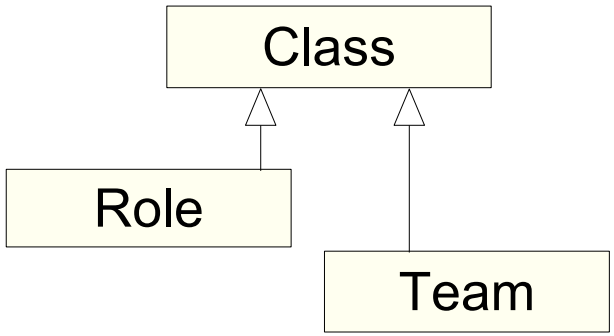
- Rollen basieren auf Beziehungen.
- Ein Objekt und seine Rolle teilen Identität.
- Ein Objekt und seine Rolle haben verschiedene Identitäten.





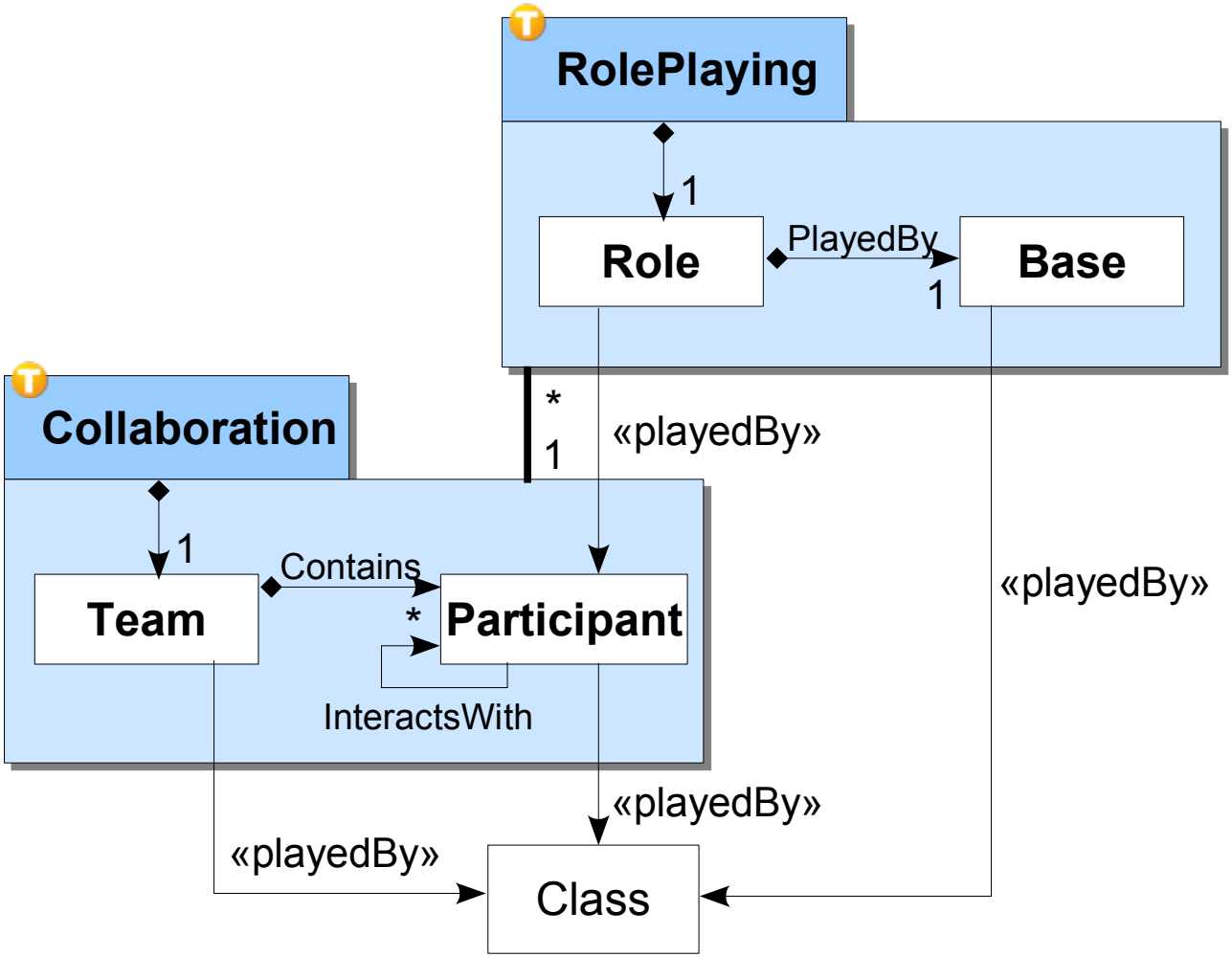
ObjectTeams/Java = 93% steimannsch

Metamodell für OT/J



- Kombinationen?
  - Role & Team: geschachtelte Teams
  - Role & Base: Role-of-Role
  - Team & Base: ”stacked teams”
- Model-Evolution?
  - Gruppe v. Klassen ↣ Kollaboration v. Rollen

OT/J Metamodell



Um Rollen zu erklären,
muß man vorher Rollen erklären.