NET.OBJECTDAYS 2005

Aspect Oriented Programming with Views and Collaborations

The TOPPrax approach

Stephan Herrmann
Christine Hundt
Technische Universität Berlin

stephan@cs.tu-berlin.de
resix@cs.tu-berlin.de

www.ObjectTeams.org

Language & Method

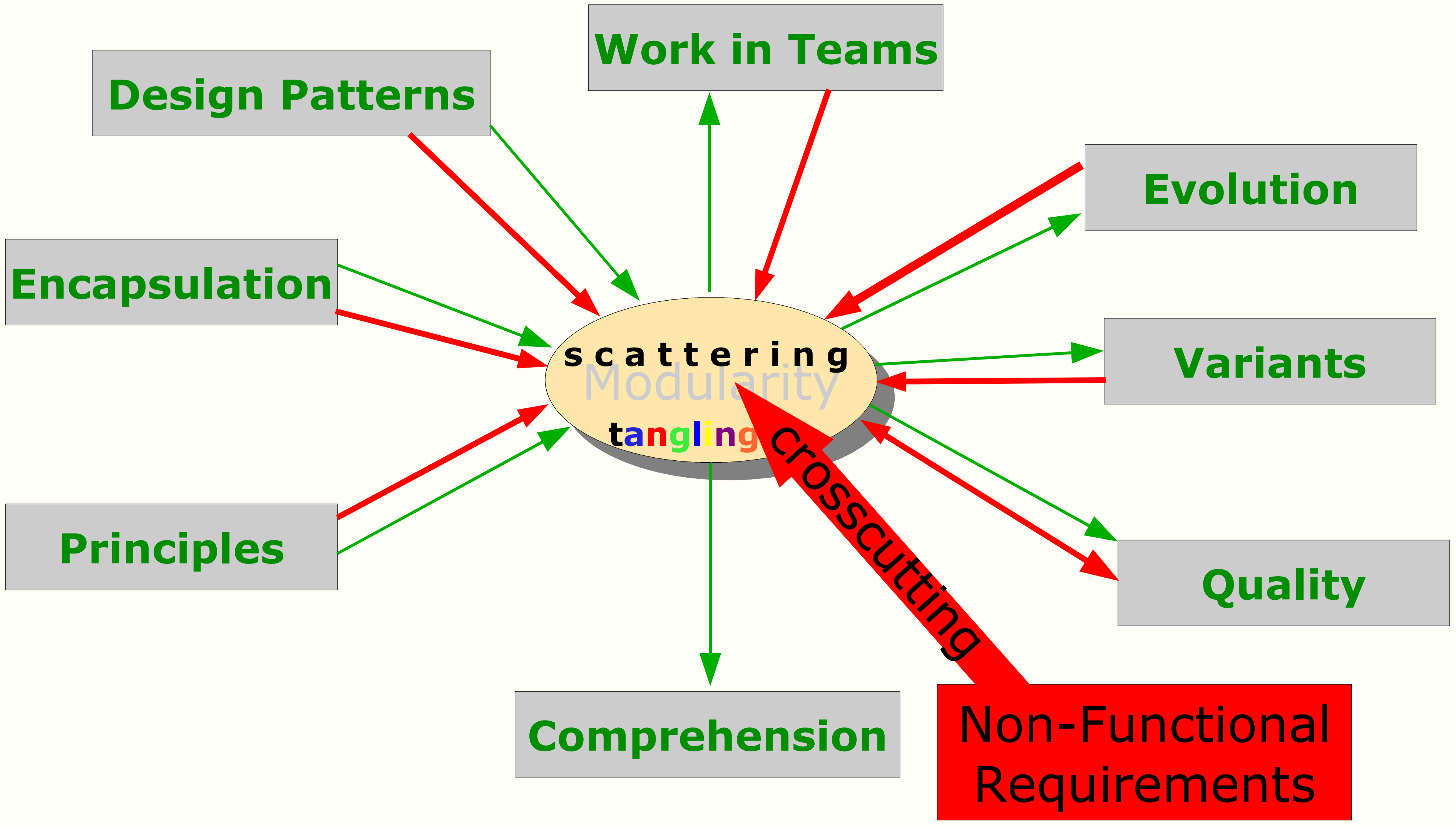
PART 1:

ObjectTeams/Java – The Language

PART 2:

Patterns of Good Design with OT/J

Motivation


Work in Teams
Design Patterns
Evolution
Encapsulation
scattering
Modularity
tangling
Variants
crosscutting
Principles
Quality
Comprehension
Non-Functional Requirements

Rescue?

What can a programming language help?

- **Define „module“**
  - Classes **don't scale**
  - Packages **are too weak**
  - Components **may be too heavy**

Language support for modules larger than classes?

- **Define module relationships**
  - Use
  - Adaptation
  - Encapsulation

Relationships for modules larger than classes?

Optimal Module Structure?

- **Objectively optimal?**
- **Subjectivity!**
  - is introduced by
    - Stakeholder, concern, variant, task, use case, diagram, ...
  - manifests as
    - Views, viewpoints, roles, aspects, ...

Each view suggests a good modular break down

Support different structures simultaneously!

Our Answers

- Against crosscutting:
  Aspect Oriented Programming
- Modules larger than classes:
  Collaborations („Teams“)
- Module relations for „Teams“
- Programming with views:
  Roles

Aspect Oriented Programming
with Views and Collaborations

OT/J Facts

- **Object Teams** **(Programming Model)**
  - Incorporates concepts from
    - Aspect Oriented Programming
    - Programming with Roles
    - Collaborations
- **ObjectTeams/Java** **(Programming Language)**
  - Fully compatible with Java (currently 1.4)
  - Compiler and runtime environment
- **OTDT** **(Development Environment)**
  - Eclipse extension
  - Extended convenience & new views

OT/J Status

- **The road we have come so far**
  - Work on tools started late 2001
  - First class-room use summer 2003
  - Continuous testing
    - Two test-suites:
      ```
      compiler:  > 1100 cases (programs), 98% PASS
      OTDT:      > 1600 cases, 95% PASS
      ```
- **Project TOPPrax:**

  Universities ↔ Fraunhofer ↔ Companies
  - Consolidation
  - Method
  - Evaluation

The TOPPrax Approach

gefördert vom:
Bundesministerium
für Bildung
und Forschung

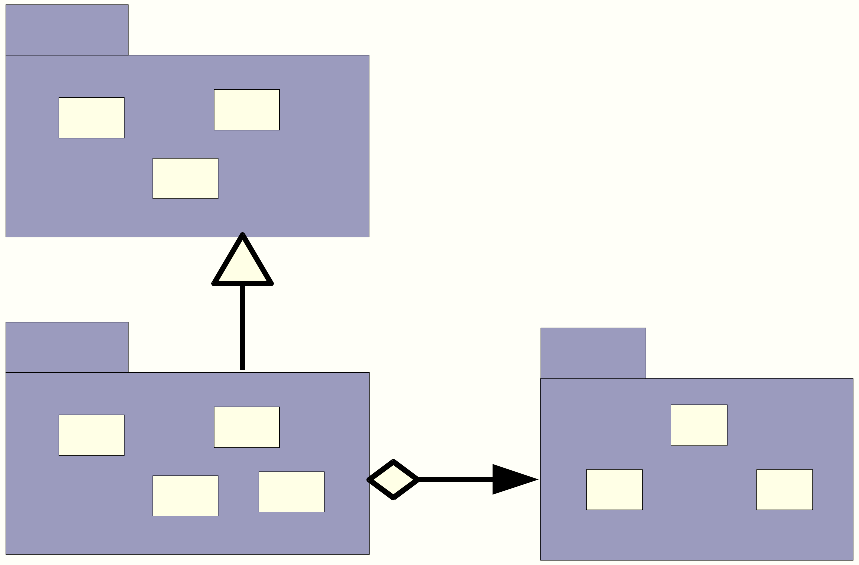
TOPPRAX
www.topprax.de

Core Concepts

- **2 new kinds of modules:**
  - **Team** = Group of **Roles**

- **New relationships:**
  - Team «adapts» base
  - Team inheritance

  Both relationships will be refined



- **Integration:**

Classes	• Role-base	
Methods	• Forwarding	• Overriding/Interception
Dynamism	• Activation/Deactivation	• Instantiation

Aspectoriented Programming with Views and Collaborations

The TOPPrax approach

Teaser Example

Mini C R M

- **Existing application**
  - „Database“ application with simple GUI
  - Shipped in a jar-file
- **Existing module for input validation**
  - A-posteriori integration of
    - Validation (field types: `String, (phone) number, city-codes`)
    - Error-Dialog
  - Select extension at launch time

Demo Time ...

Mini C R M

- **Existing application**
  - „Database‟ application with simple GUI
  - Shipped in a jar-file
- **Existing module for input validation**
  - A-posteriori integration of
    - Validation (field types: `String, (phone) number, city-codes`)
    - Error-Dialog
  - Select extension at launch time

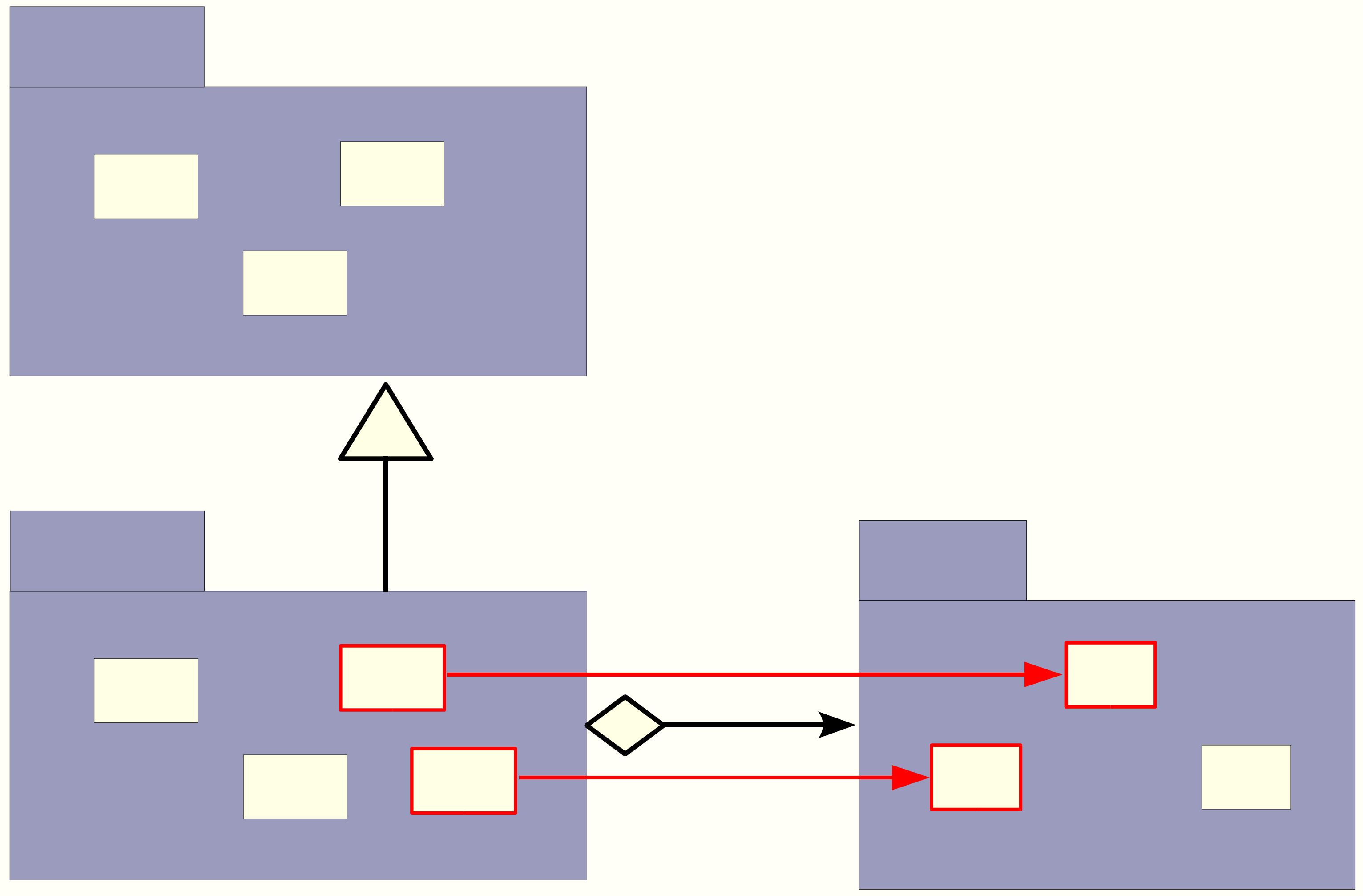
☺ Adapt existing applications.
☺ No need for pre-planning.
☺ Extension is a module, too.

Aspectoriented Programming with Views and Collaborations

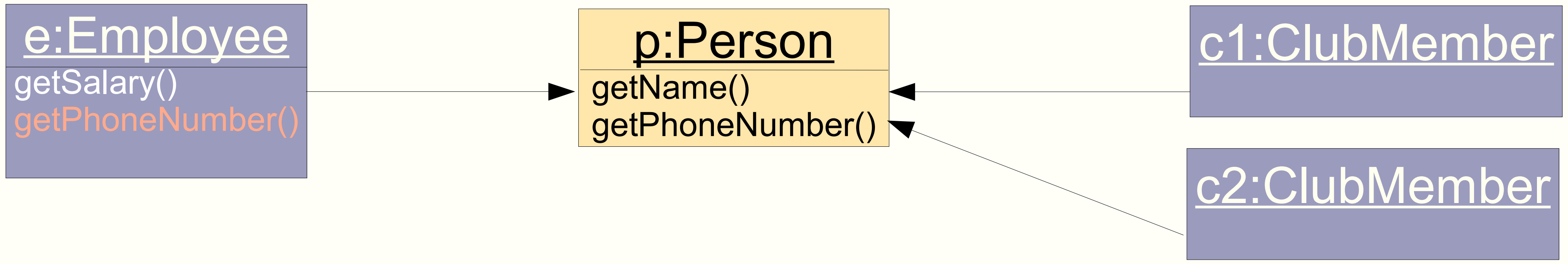
The TOPPrax approach

Roles, Bases & Teams

Overview



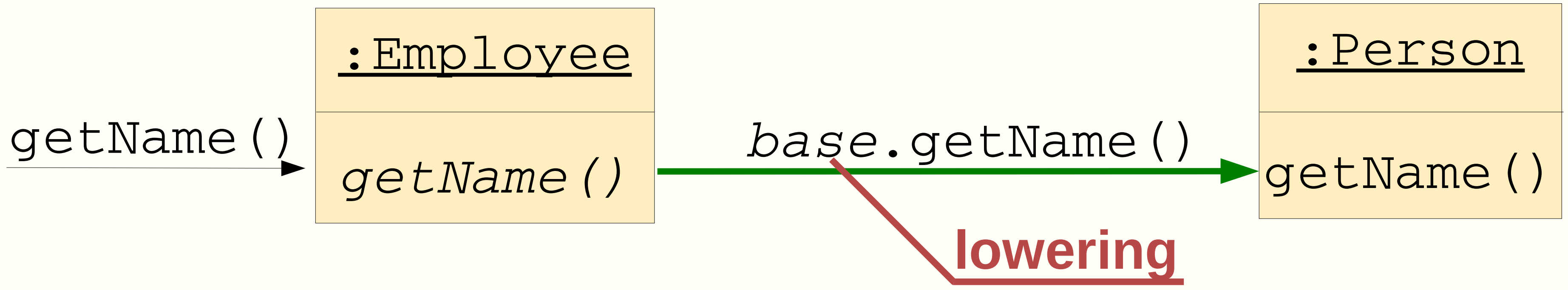
Roles and Bases



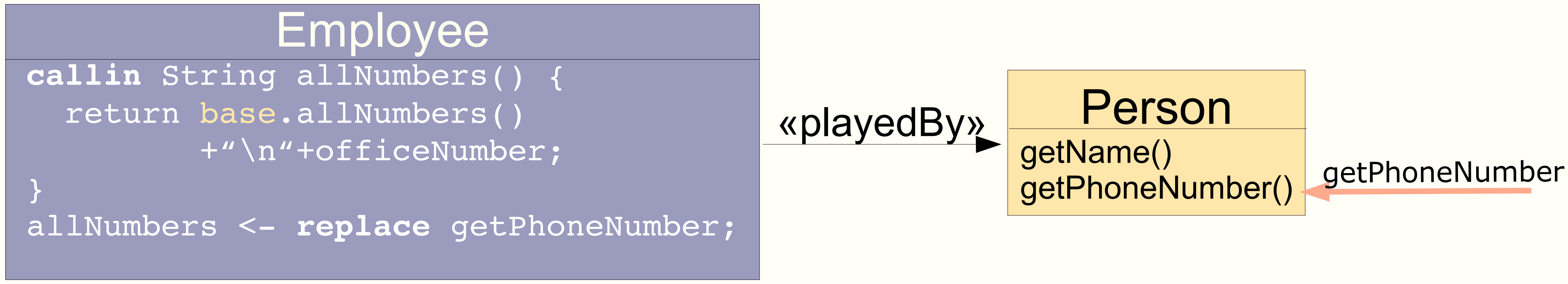
- **Roles**
  - provide a view to the base
  - add additional behavior
  - use part of the base functionality
  - multiple roles played by a base
  - instance level: multiple role objects
  - method dependencies

Method Binding (1)

- **Callout Binding**
  - Forwarding (instance based inheritance)
  - declarative: `getName -> getName`
  - adaptable: name, signatur



Method Binding (2)

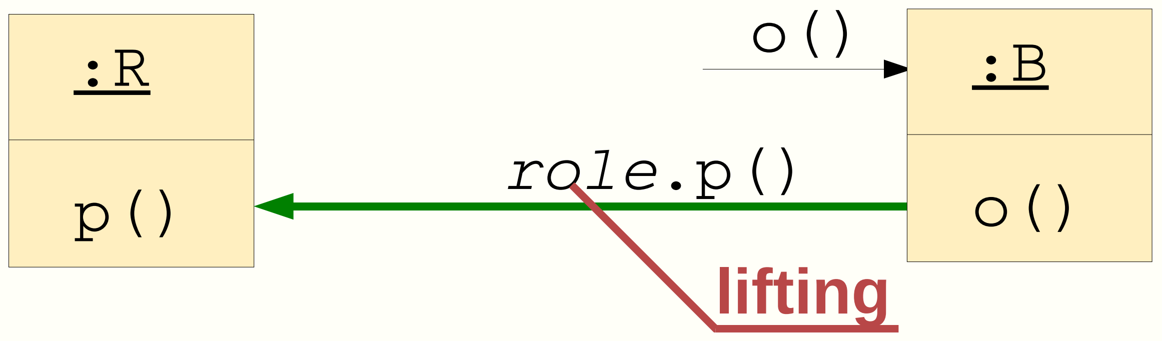


```
callin String allNumbers() {
  return base.allNumbers()
         +"\n"+officeNumber;
}
allNumbers <- replace getPhoneNumber;
```

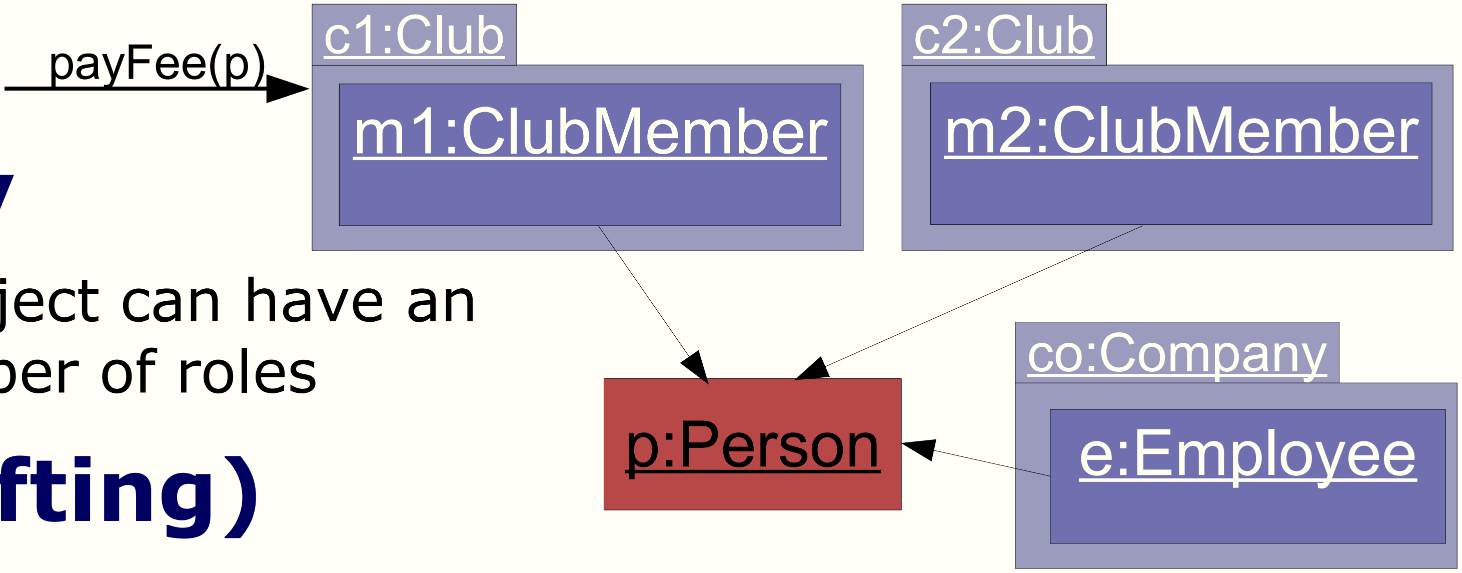
- **replace callin:**
  - replace the originial base method (overriding)
  - only for callin methods
- **base call:**
  - semantics: call of the original method, recommended
  - syntax: base.rm()
    - role method signature -> independent of binding

Method Binding (2)

- **Callin Binding**
  - **replace** (overriding); **before, after** (additive)
  - *advice weaving*
  - declarative: `p <- after o`
  - adaptable: name, signatur



Role Lookup

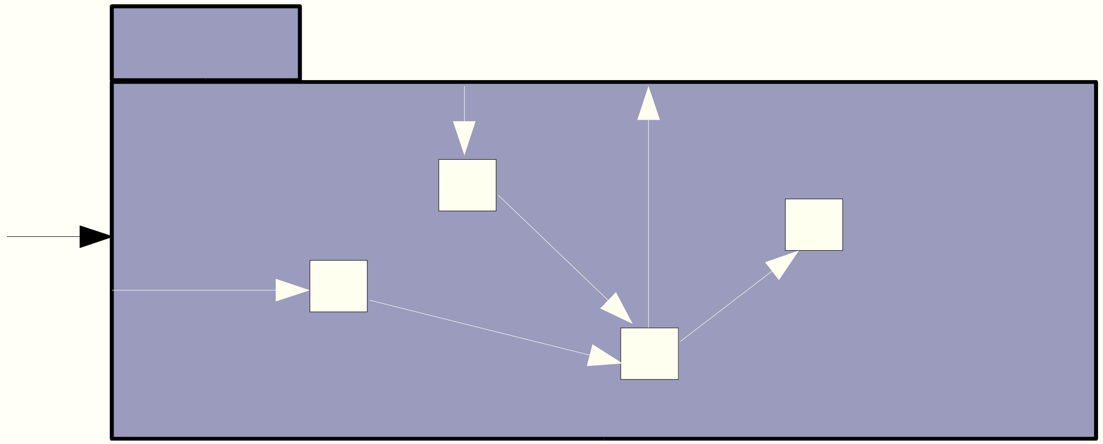


- **Multiplicity**
  - Every base object can have an arbitrary number of roles
- **Lookup (Lifting)**
  - How to find the proper role?
  - Automatism at runtime

```
team class Club {
   class ClubMember playedBy Person { ... }
   void payFee(Person as ClubMember cm) {...}
}
...
c1.payFee(p);
```

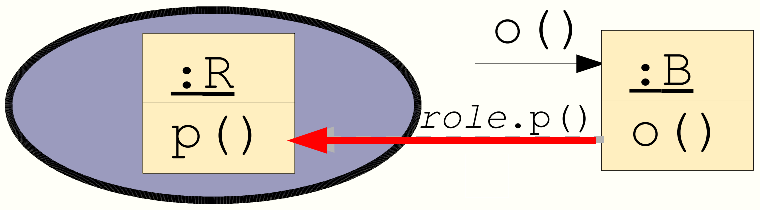
Teams

- **Modules larger than classes**
  - Contains roles (Container)
  - Encapsulation (Façade)
  - Interaction (Collaboration)
  - Group identity (Mediator)



Team Activation

- **When do callins have an effect?**
  - for every objects of the base class
  - for every active instance of the team
- **Semantics:**
  - switch on **all** callin bindings of a team
  - for individual Team instances
  - program *mode*
- **Methods:**
  - `Team.activate()` and `Team.deactivate()`



Guard predicates

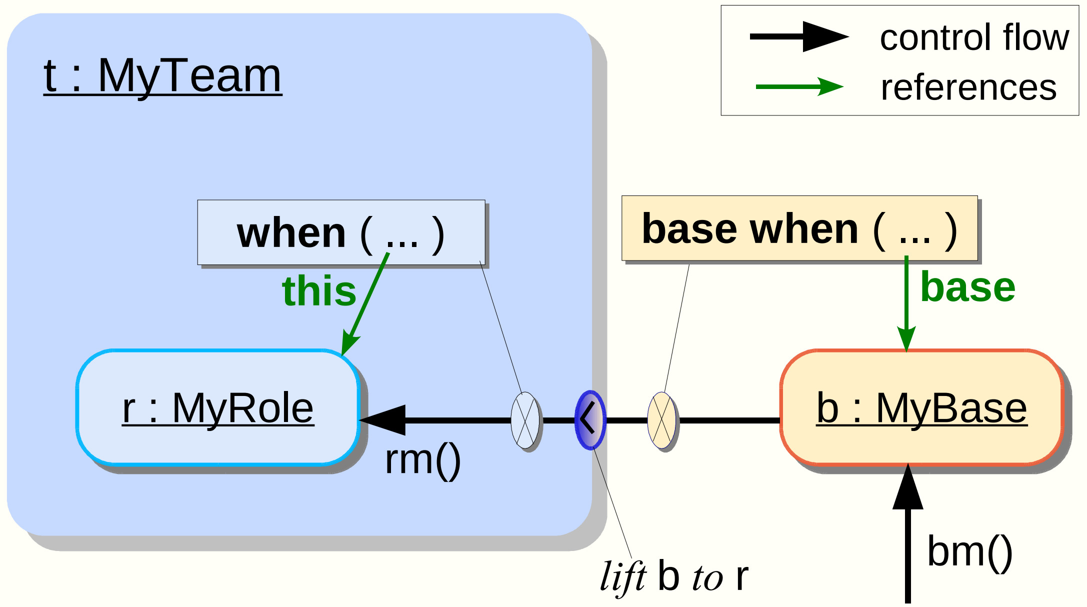
- **Example:**

```
team class Company {
  class Employee playedBy Person when (!onLeave) {
    boolean onLeave;
    callin String allNumbers() {...}
    allNumbers <- replace getPhoneNumber;
  }
}
```

- **Granularity of *guard predicates*:**

Location	**Affected role methods**
role method binding	call of the role method via callin from the corresponding base method
role method	every call of the role method via callin
role	all in this role
team	any in every role of the team

Guard predicates



Control of activation

- *role side*
- **`when`** `(<boolean expression>)`
- Access to role features via `this` and callout-bound base features

Control of instantiation

- *base side* (pre-role instantiation)
- **`base when`** `(<boolean expression>)`
- Access to base features via **`base.`**`<base-feature>`

Summary

- **Roles played by Bases**
- **Methodbinding: Callout, Callin**
- **Navigation: Lifting, Lowering**
- **Teams**
- **Team activation, Guard predicates**

And now...

Example: StopWatch

Aspectoriented Programming with Views and Collaborations

The TOPPrax approach

Team Inheritance

What is a Team?

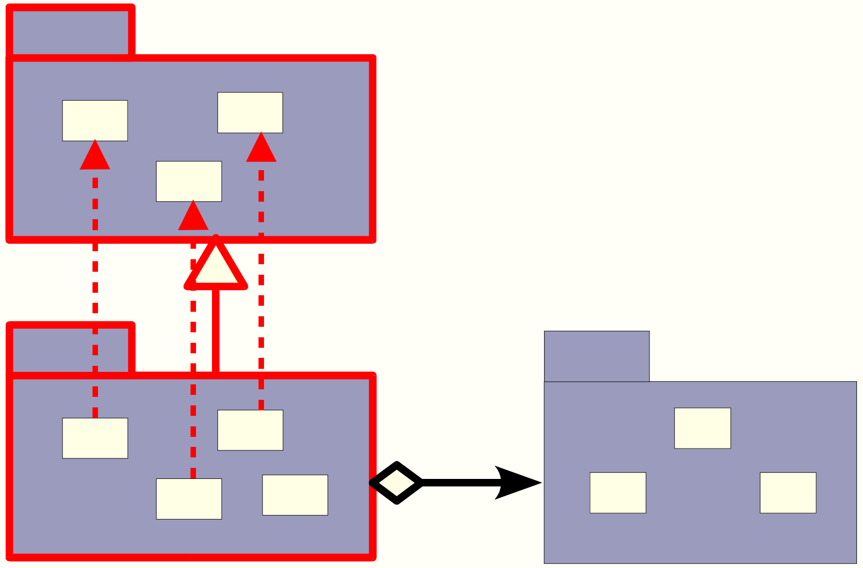
- **Team = Container for Roles**

  - Is it a class?
  - Yes: class (team) with **inner classes** (roles).
  - Is it a package?
  - Yes: roles may be stored in a **team directory** ...
  - Is it a component?
  - Team **encapsulates** its roles, flexible: black, white, gray box.

☺ better scalable than classes
☺ stronger semantics than packages
☺ strong, flexible encapsulation

Team-Classes

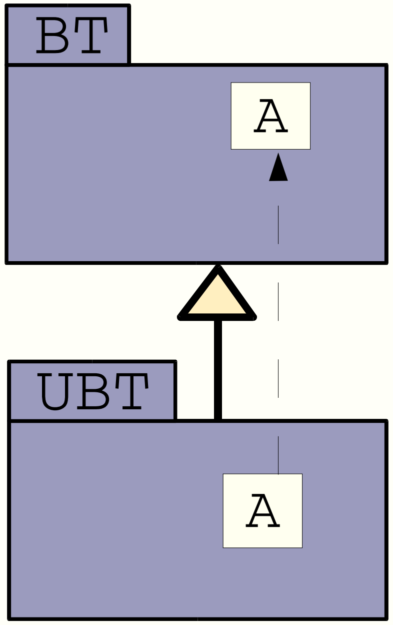
- **Team Inheritance**
  - „import": Use features **& role classes** from the super-team
  - „overriding": Adapt mismatching methods **& role classes**
    - Java cannot override inner classes!
    - Overriding of roles classes in Object Teams



⏩ virtual classes
⏩ „implicit inheritance"

Implicit Inheritance

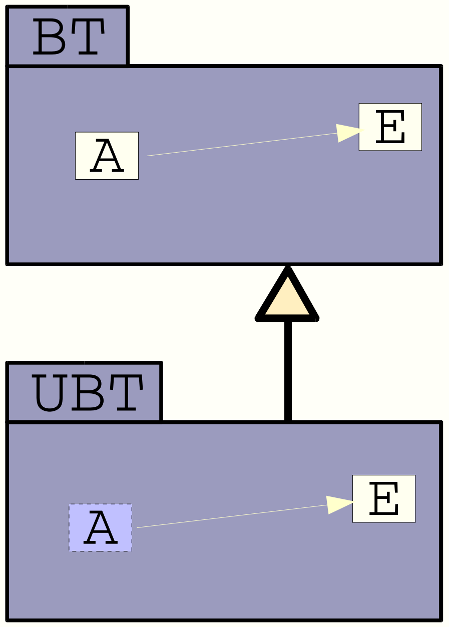
```
team class BusinessTrip {
   protected class Application {
      Calendar start, today;
      boolean isValid() { ... }
   }
}
team class UniBT extends BusinessTrip
{
   protected class Application {

   }
}
```



- UBT.A **implicitly inherits** from BT.A
- Relation is defined by **name equality** „A".
- Implicit inheritance supports
  - **import**
  - **overriding**

Multi-Class Refinement

```
team class BusinessTrip {
   protected class Event { ... }
   protected class Application {
      Event getEvent() { ... }
   }
}
team class UniBT extends BusinessTrip {
   protected class Event {
      boolean havePaper;
   }
   boolean checkGrant (Application appl) {
      Event e = appl.getEvent();
      return e.havePaper;
   }
}
```



Multi-Class Refinement

```
team class BusinessTrip {
   protected class Event { ... }
   protected class Application {
      Event getEvent() { ... }
   }
}
team class UniBT extends BusinessTrip {
   protected class Event {
      boolean havePaper;
   }
   boolean checkGrant (Application appl) {
      Event e = appl.getEvent();
      return e.havePaper;
   }
}
```

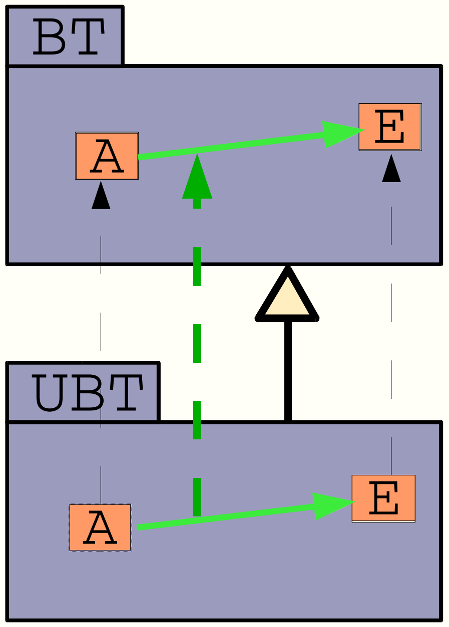
Type `Event` is bound dynamically

Overrides class `BusinesTrip.Event`

has type „`UniBT.Event`"

Multi-Class Refinement

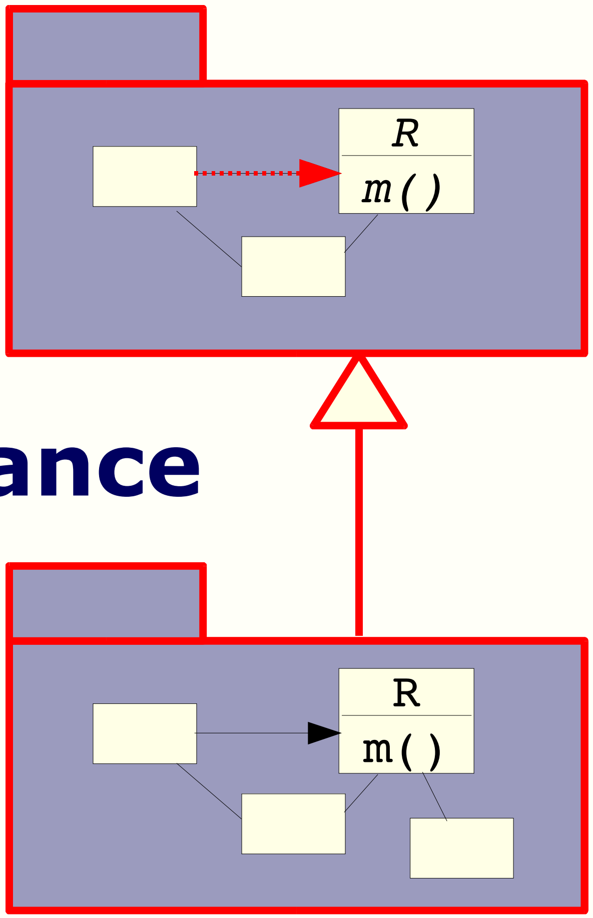
```
team class BusinessTrip {
   protected class Event { ... }
   protected class Application {
      Event getEvent() { ... }
   }
}
team class UniBT extends BusinessTrip {
   protected class Event {
      boolean havePaper;
   }
   boolean checkGrant (Application appl) {
      Event e = appl.getEvent();
      return e.havePaper;
   }
}
```



☺ Implicit overriding of associations

Frameworks

- **Team ≃ Framework**
  - Partial implementation (compound)
  - Hotspots
    - (abstract) methods
    - (abstract) role classes
- **F. instantiation ≃ T. inheritance**
  - Adaptations at hotspots
    - define/override methods
    - define/override role classes
- **Role instantiation?**
  - Factories?



Abstract Roles

```
abstract team class BT
{
   abstract class A {}
   A appl = new A();
}
team class UBT extends BT
{
   class A {...}
   ...
}
```

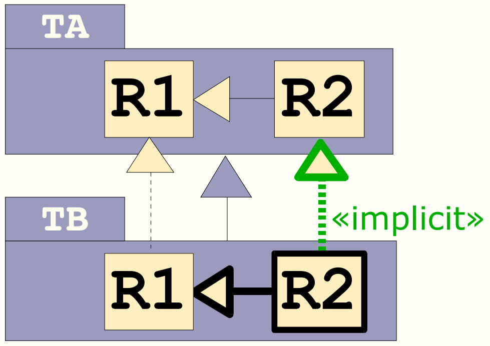
```
abstract class C
{
   abstract void hook();
   void template(){
        hook();
   }
}
class C2 extends C {
   void hook() {...}
}
```

Template & Hook for Classes

- Team `BT` is template ⇒ incomplete implementation
- Rolle `A` is hook ⇒ opening filled in `UBT`

Implicit Inheritance

- **Overriding role <u>implicitly</u> inherits**
  - Inheritance relation by name equality
  - Import features of super-role
  - Override features of super-role
- **Difference to regular inheritance**
  - Even constructors are inherited
  - ⏩ *making covariance safe:* no sub-type relation
- **Both kinds can be combined**



Summary Part 1

- **Concepts presented**

– roles played by bases	playedBy
– method bindings	callout/callin
– navigation role ↔ base	lowering/lifting
– role creation	
• implicitly	lifting
• explicitly	even abstract roles
– teams	
• class & package	
• activation	explicit + guards
• inheritance	role overriding + implicit inheritance

Outline Part 2

- **Patterns of good design with OT/J**

  Patterns found in existing applications:
  - Connector
  - Notification
  - Virtual Association
  - Virtual Restructuring
  - Variant

  Scalable Designs:
  - Nesting, stacking and layering of Teams.