ECOOP 2009 Summer School

Object Teams: Programming with Contextual Roles

Stephan Herrmann
Independent

www.objectteams.org

© 2009 by Stephan Herrmann; made available under the EPL v1.0 | July 9, 2009

Roles are a seamless extension of O-O

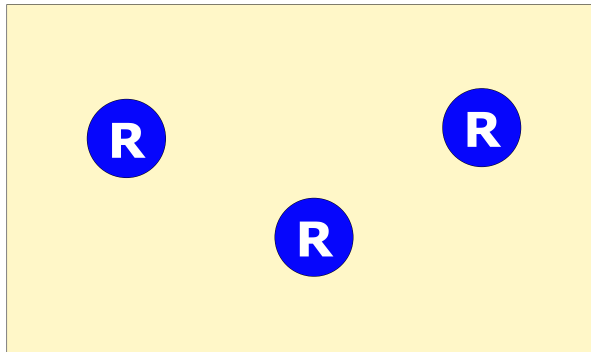
- classes & objects & roles ?
  - these are boring!
- what's happening between these things?
  - **association**
    - composition / containment (stricter semantics)
  - **inheritance**
    - delegation (more flexible)
    - nested inheritance (larger scale)
  - **interactions**
    - explicit message send
    - contextual dispatch

Relationships

- Object Teams introduces two relationships
  - object containment
    - instances nested within instances
    - supports interaction among siblings
  - playedBy relationship
    - inheritance-like delegation
    - supports interaction among parts of an object
- And then
  - virtual classes & family polymorphism
  - translation polymorphism
  - generics
  - recursive structures
  - ...

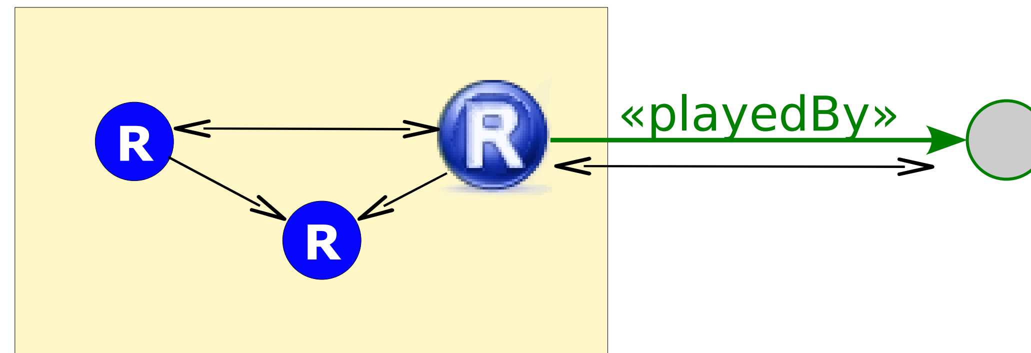
Coherence

- These concepts are connected by Roles



Coherence

- Concepts are connected by Roles



- The two faces of a Role
  - member of a context
    - interact with each other
  - view of an underlying base
    - interact with base / two parts of “self“

Powerful Pivot

- Roles
  - connect intuition to technology (metaphor)
  - emphasize objects over classes
  - introduce subjectivity into programming
  - are broadly explored in research
- Roles entail the concept of Contexts
- Designing with roles
  - adds one more dimension of separation of concerns

Powerful Pivot

- Roles
  - connect intuition to te
  - emphasize objects ov
  - introduce subjectivity
  - are broadly explored i
- Roles entail the concept o ntexts
- Designing with roles
  - adds one more dimension of separation of concerns

ICSE 2009:
„Most influential paper"
from ICSE 1999:

„N Degrees of Separation:
Multi-Dimensional Separation of Concerns"

Definition of Roles

- Natural types
  - relate to the essence of the entities
    - **rigid**
      - *being a Person doesn't change over time*
    - **absolute**
      - *being a Person depends on nothing outside*
- Role types
  - depend on an (accidental) relationship to some other entity
    - **dependent**
      - *being a Student depends on an enrollment relationship*
    - **lack semantic rigidity**
      - *Student-ness can change over time without loss of identity*

No more "things"
(classes, objects, roles - types)

Relationships!

OOD = has_a & is_a

Taxonomy of “is”

- is = instance-of
  - Eric Jul is_a Man
  - *set membership:* instance x type
- is = subtype
  - A Man is_a Person
  - *set inclusion:* type x type
- is = role-of
  - Eric Jul is_the President (of AITO)
  - *role attachment:* instance x instance
- is = generalized playedBy
  - A President is_a Person
  - *promise of role attachment:* type x type

more details!

- Graversen 2006: *The nature of roles*
  A taxonomic analysis of roles as a language construct.

Properties of Roles (1/4)

- 15 Criteria by Friedrich Steimann
    - and their mapping to Object Teams

- First approximization of ObjectTeams/Java
  - Java + Delegation
    - role containment : inner classes (instance containment)
    - playedBy : delegation (overriding, late binding of self)

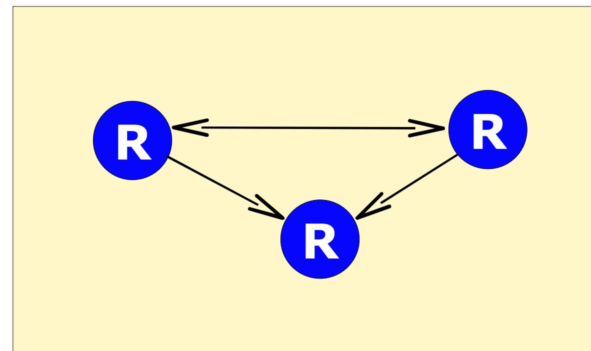
Properties of Roles (1/4)

- 15 Criteria by Friedrich Steimann
  - and their mapping to Object Teams

- **Roles depend on relationships**
  - roles depend on context (relationship, collaboration, ...)
- **A role comes with its own properties and behavior**
  - ✔ roles are types
- **The state of an object can be role-specific**
  - ✔ role objects have state
- **Features of an object can be role-specific**
  - ✔ roles can override base features
- **An object may acquire and abandon roles dynamically**
  - ✔ role playing is a dynamic relationship between objects

Reconsider O-O Basics: Association vs. Containment

Options & Choices



Restricting Access

- Encoding architectural constraints
  - access restricted to authorized clients
  - Java: private, _, protected, public
    - coarse grained
    - class based restrictions

    } not specific enough
- Making the type system instance-aware
  - dependent types: annotate types with instances
    - Role<@context>: type parameter is an instance
    - Role<@c1> ⋠: Role<@c2> (unless c1==c2)
  - protected roles cannot be exposed
    - CoffeeMachine<@Department.this> OK
    - CoffeeMachine<@yourDepartment> illegal

Stricter Alias Control

- Ownership could leak through polymorphism
  - every (dependent) type <: `Object`?
    - new top-level class: `Confined`
    - protected sub-classes of `Confined` cannot leak
- Ownership/confinement may be too strict
  - compromises
    - restricted inheritance: **reuse**, yet preserve “anonymity”
    - accessible as opaque, featureless roles
    - expose `readonly` interface, keep class inaccessible
- Not yet:
  - formalization, proofs
  - implementation for restricted inheritance, readonly

Richer Semantics

- Just one kind of associations is too weak
  - need different levels of protection
- Role containment
  - roles are dependent types
    - default: implicit owner can be omitted
  - ownership / confinement become more natural
    - intermediate variants possible
  - connects ownership to the role/context metaphor
- Make this the foundation for other concepts

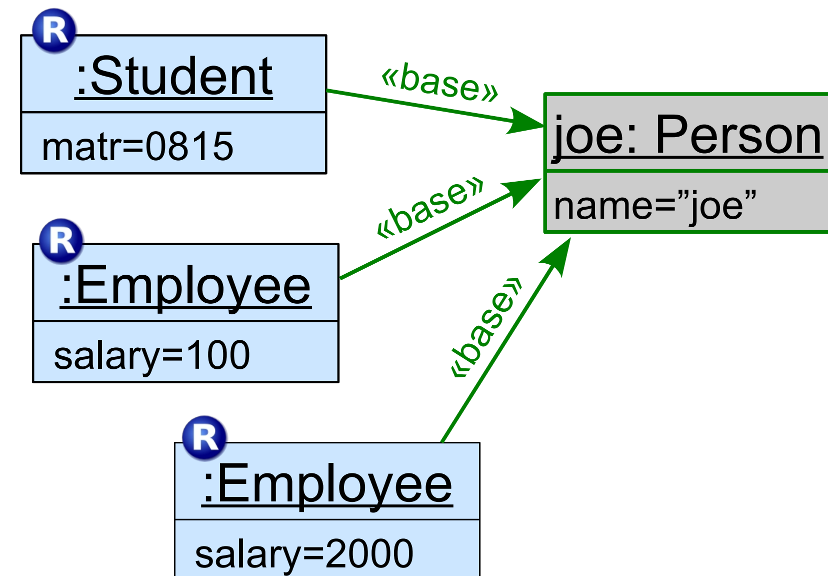
Comparing
Inheritance vs. Role Playing

Design Choices

Properties of Role-Playing

playedBy Relationship

- Advantages:
  - **Dynamism**:
    roles can come and go
    (same base object)
  - **Multiplicities:**
    one base can play several roles
    (different/same role types)
- Similarity to inheritance
  - `playedBy` declares **delegation**
  - [cf. “Treaty of Orlando“ 1988]

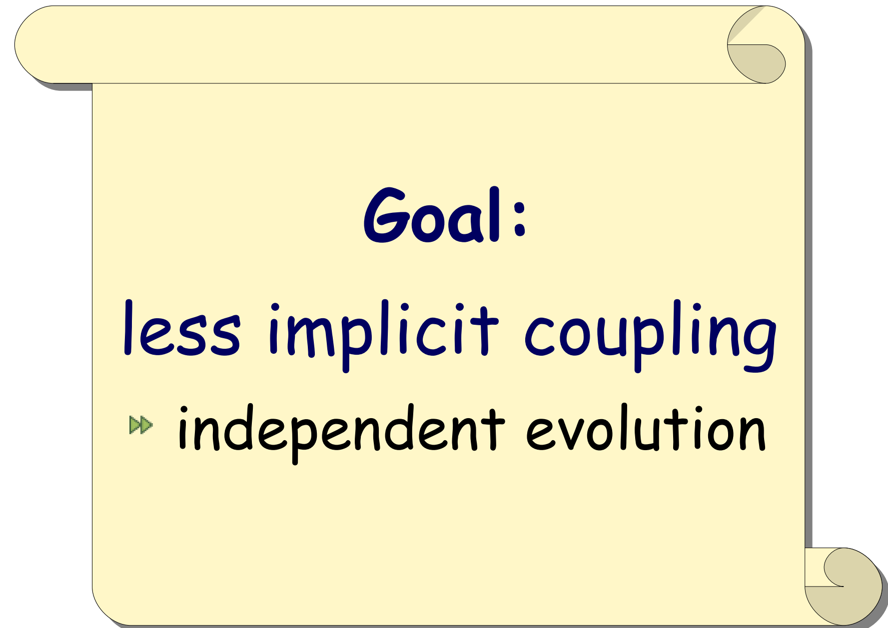


Detailed Comparison

Inheritance

- Import
  - dispatch sub → super
- Overriding
  - dispatch super → sub
- Substitutability
  - pass an instance of sub class where the super class is expected

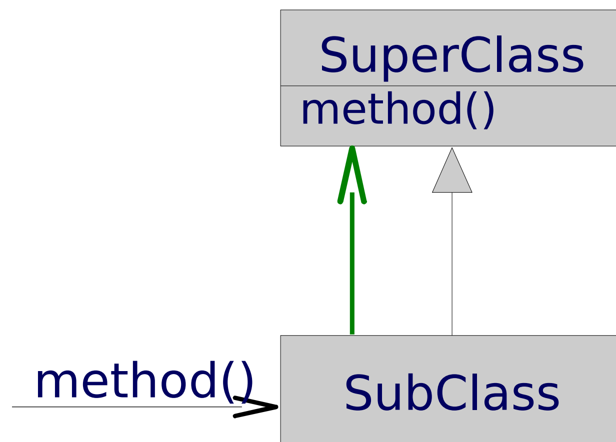
Role Playing



Inheritance vs. PlayedBy in OT/J

Inheritance

- Import
  - **dispatch sub → super**

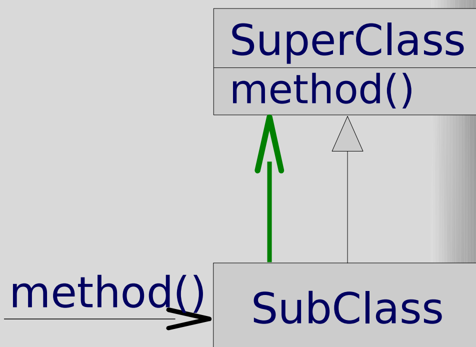


Role Playing

Inheritance vs. PlayedBy in OT/J

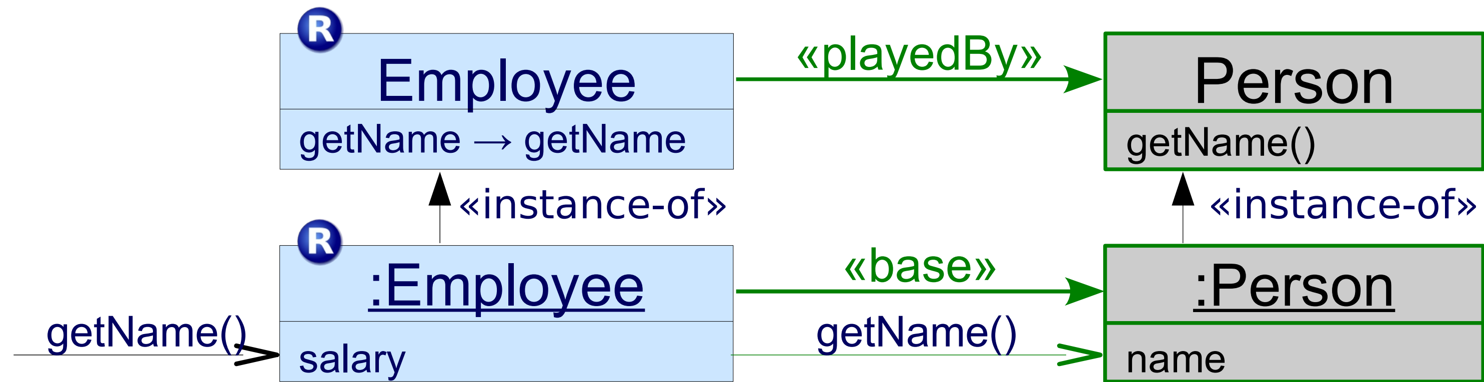
Inheritance

- Import
  - **dispatch sub**



Role Playing

- Callout binding
  - **dispatch role → base**



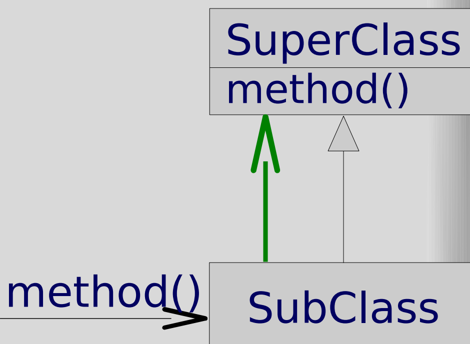

```
String getName() -> String getName();
```

- different names
- parameter mappings (implicit/explicit)
- callout to field
- decapsulation

Inheritance vs. PlayedBy in OT/J

Inheritance

- Import
  - **dispatch sub**



Role Playing

- Callout binding
  - **dispatch role → base**

getNa

No other access to «base»

- encapsulate semantics
- separate two worlds
- specific privilege

on
nce-of»
on

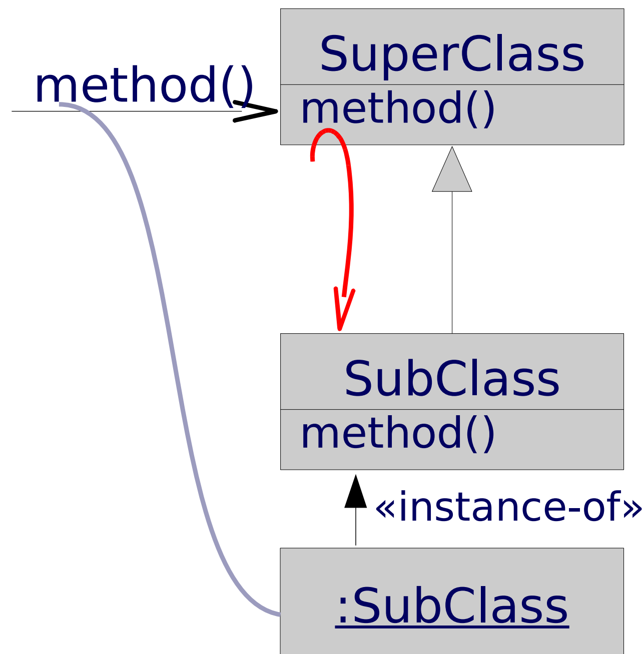
```
String getName() -> String getName();
```

- different names
- parameter mappings (implicit/explicit)
- callout to field
- decapsulation

Inheritance vs. PlayedBy in OT/J

Inheritance

- Import
  - **dispatch sub → super**
- Overriding
  - **dispatch super → sub**



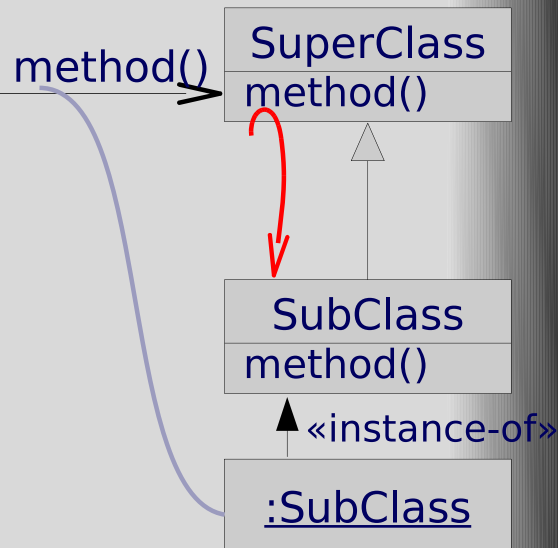
Role Playing

- Callout binding
  - **dispatch role → base**

Inheritance vs. PlayedBy in OT/J

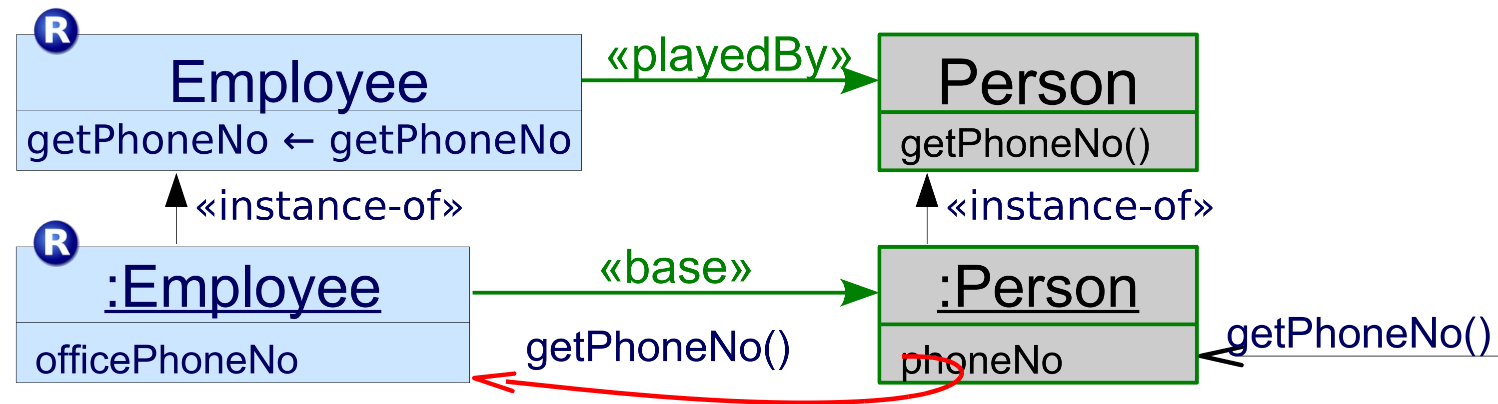
Inheritance

- Import
  - dispatch su
- Overriding
  - **dispatch su**



Role Playing

- Callout binding
  - **dispatch role → base**
- Callin binding
  - **dispatch role ← base**



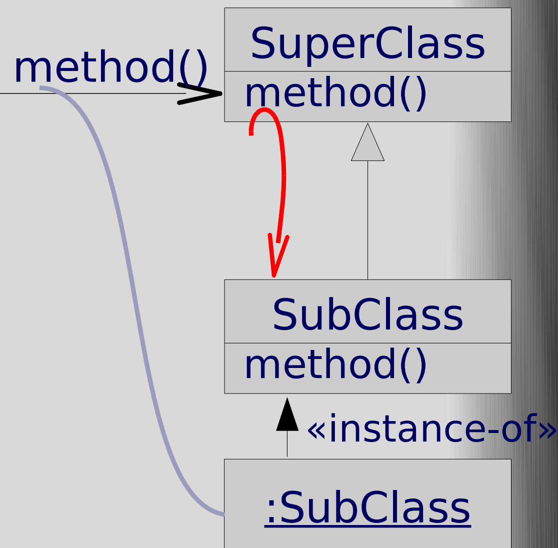

```
String getPhoneNo() <- replace String getPhoneNo();
```

- different names
- parameter mappings (implicit/explicit)
- before / replace / after
- base calls

Inheritance vs. PlayedBy in OT/J

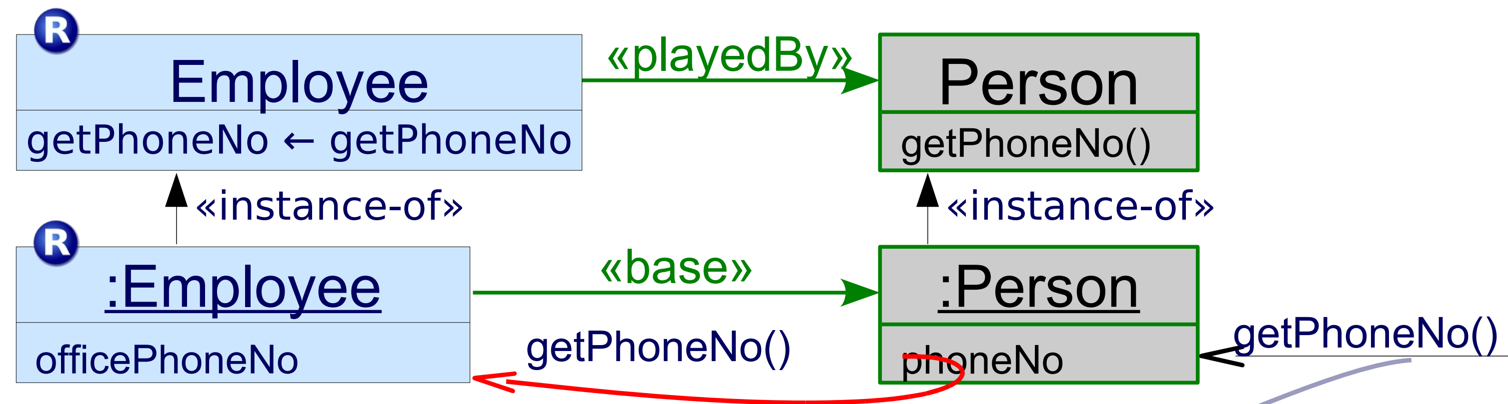
Inheritance

- Import
  - dispatch su
- Overriding
  - **dispatch su**



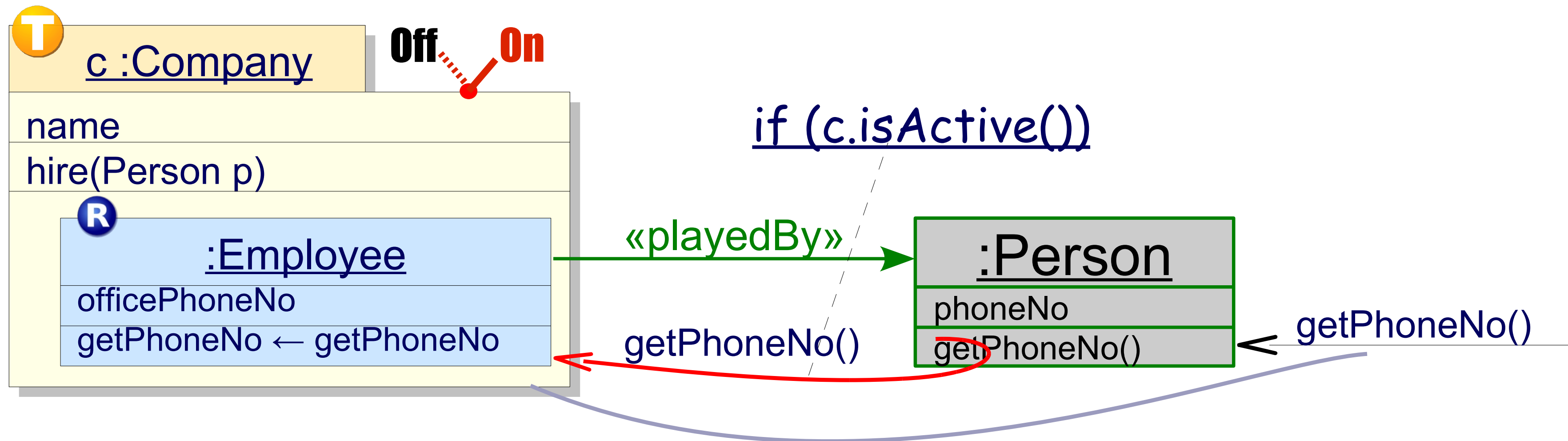
Role Playing

- Callout binding
  - **dispatch role → base**
- Callin binding
  - **dispatch role ← base**



which context
to select the appropriate
role instance?

Teams as Activation Context

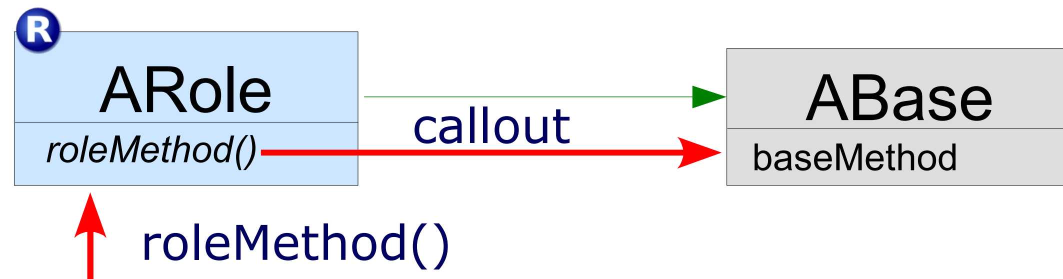


- **Roles depend on context**
- In OT/J contexts are reified as **Teams**
  - roles are inner classes of a **team class**
  - role instances are inner instances of a **team instance**
- Each team instance can be **(de)activated**
  - active team instances contribute to the **system state**
  - dispatch considers system state

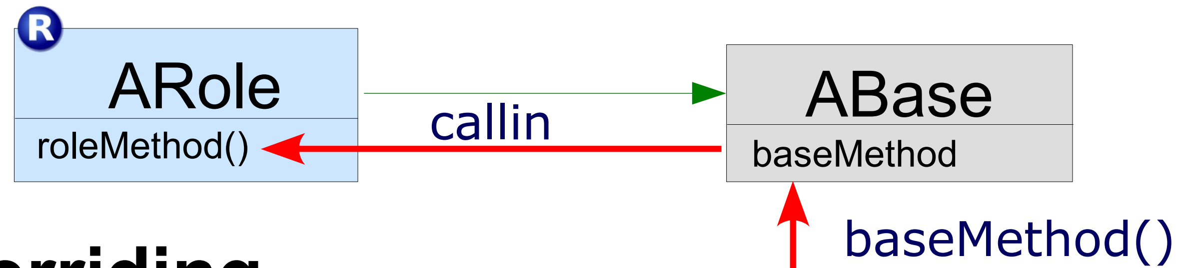
activation mechanisms:
- globally
- per thread
- implicitly
- per block

Buy 2 – Get 1 for Free

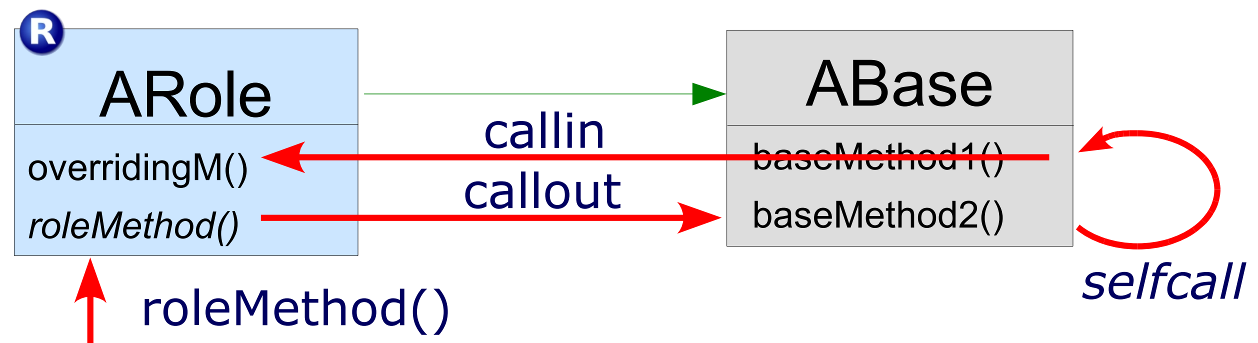
- 2 Mechanisms, 3 styles of dispatch
  - **Forwarding**



  - **Interception**



  - **Delegation w/ Overriding**
    = Forwarding
    + Interception



Inheritance vs. PlayedBy in OT/J (2/3)

Detailed Comparison

Inheritance	Role Playing
Import - **dispatch sub → super**	Callout binding - **dispatch role → base**
Overriding - **dispatch super → sub**	Callin binding - **dispatch role ← base**
Substitutability - **pass an instance of sub class where the super class is expected**	Translation polymorphism

Substitutability

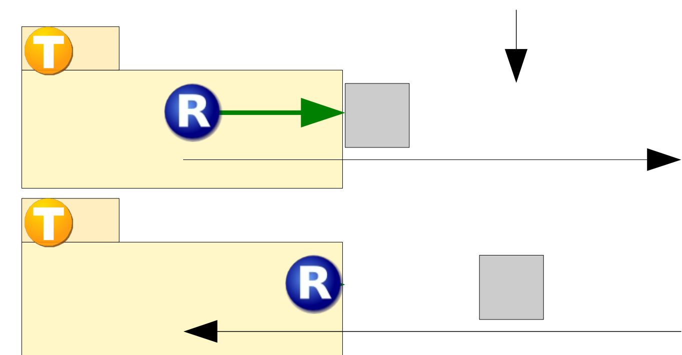
- Are the following assignments legal?

```
    Employee emp= ...
    Person person= ...
1.  person= emp;       // legal?
2.  emp= person;       // legal?
```

Normally not, but...

roles (usually) live within the team only

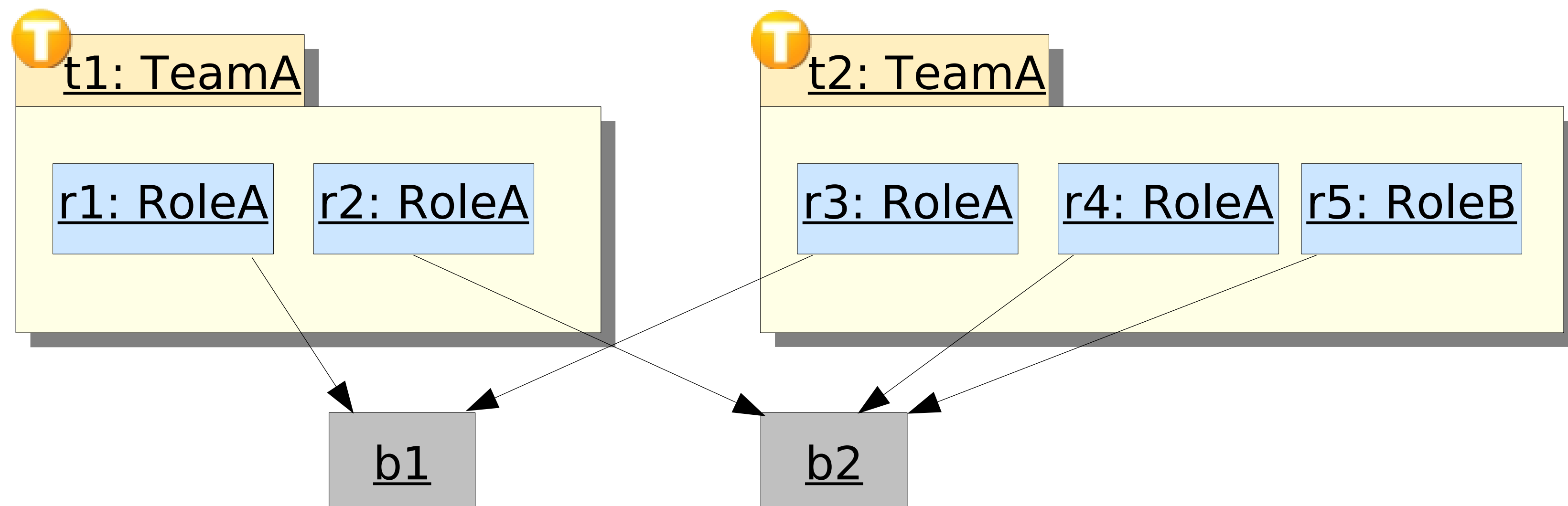
- When a role object **leaves** the team
  - it is **lowered** to its base
- When a base object **enters** a team
  - it can be **lifted** to a role



- Substitutability by translation → **Translation Polymorphism**

Role Multiplicities

- Translation base ➞ role: **Lifting**
  - A base can have many roles,
  - but only one per context: Team



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

RoleA<@t2>

Role Life Cycle

- Roles are created …
  - on demand if lifting finds no existing instance
  - or, explicitly using **new**
- Role have state
  - state is persistent across invocations / liftings
- Garbage Collector “knows”
- Team maintains …
  - mapping base → role
  - provides reflective functions (seldomly needed):

```
hasRole(aBase)
getRole(aBase, aRoleClass)
unregisterRole(aRole)
…
```

Lifting - Where&When?

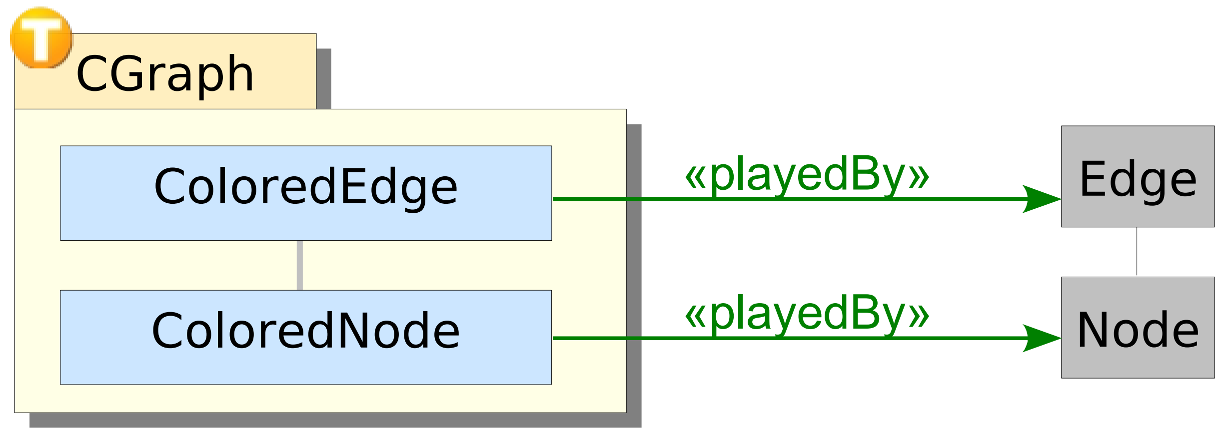
- All data flows entering the team
  - the callin call target ✔

```
team class CGraph {
    class ColoredEdge playedBy Edge {
        setStartNode(ColoredNode n) ← after setStartNode(Node n);
        ColoredNode getStartNode() → Node getStartNode();
    }
    setRootNode(Node as ColoredNode root)
```

a callin argument

a callout result

declared lifting (team method)



Translation Polymorphism

- Two-way substitutability
  - support data flows in both directions
  - no `ClassCastException`
    - if desired: `LiftingVetoException`
- Hidden at source level
  - no explicit conversions
    - if needed: `ILowerable.lower()`
  - no manual maintenance of instance mapping
- Eat the cake and have it
  - flexibility of multiple instances
  - no disadvantage of “*object schizophrenia*”
  - instances are “almost the same”

Translation Polymorphism

- Two-way substitutability
  - support data flows in both directions
  - no `ClassCastException`
    - if desired: `LiftingVetoException`
- Hidden at source level
  - no explicit conversions
    - if needed: `ILowerable.lower()`
  - no manual ma… …pping
- Eat the cake ar…
  - flexibility of multip… …nstances
  - no disadvantage of “*object schizophrenia*”
  - instances are “almost the same”

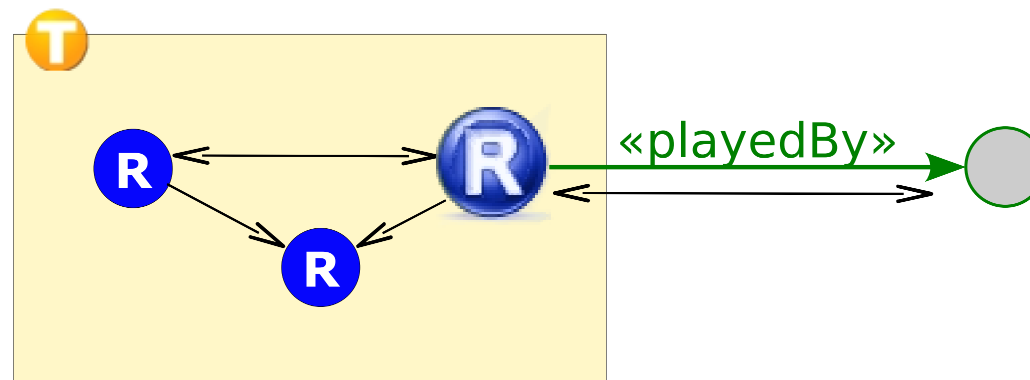
Pending: Optimizations
(compiler / runtime)

Detailed Comparison

Inheritance	Role Playing
Import / acquisition - **dispatch sub → super**	Callout binding - **dispatch role → base**
Overriding - **dispatch super → sub**	Callin binding - **dispatch role ← base**
Substitutability - **pass an instance of sub class where the super class is expected**	Translation polymorphism - lowering role → base - lifting role ← base - **two-way substitutability**

Roles & Teams

- Role playing: the powers of inheritance plus ...
  - dynamism
    - roles can come and go (same base object)
  - multiplicities
    - one base can play several roles (different/same role types)



- Teams
  - encapsulate a collaboration
    - set of interacting roles
  - team activation
    - controls the effect of all contained callin bindings

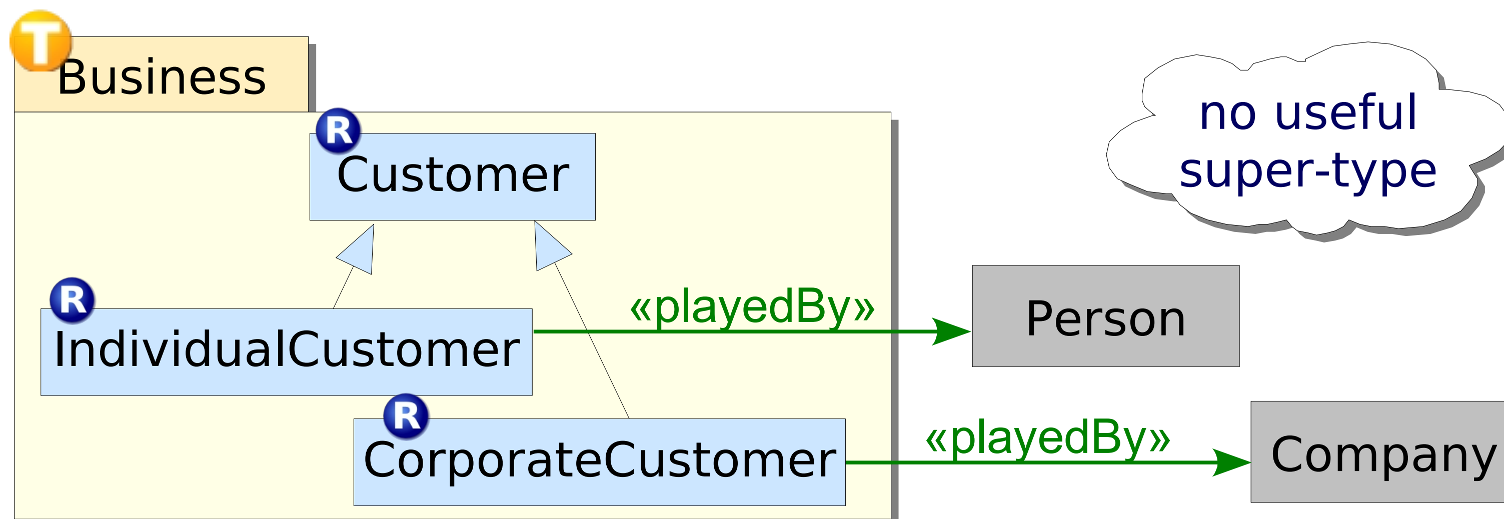
Properties of Roles (2/4)

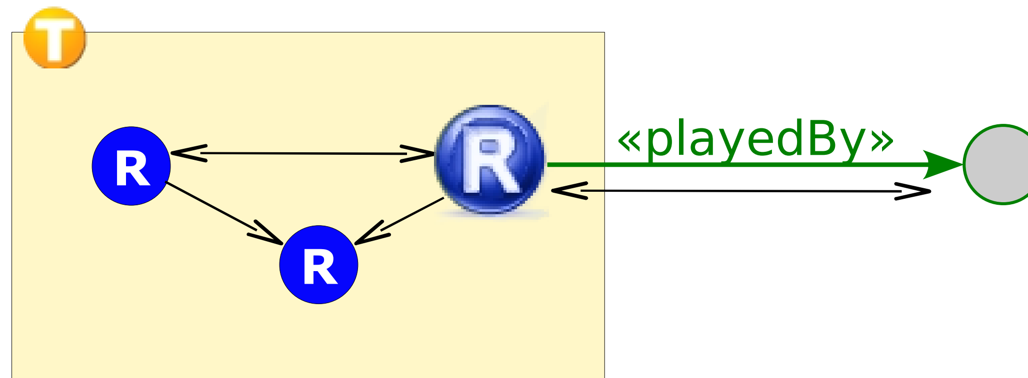
- 15 Criteria by Friedrich Steimann
  - and their mapping to Object Teams

- **An object may play different roles simultaneously**
  - ✔ roles are instances, base is agnostic of its roles
- **An object may play the same role several times, simult.**
  - ✔ differentiate by several containing team instances
- **An object and its roles have different identities**
  - ✔ roles are distinguishable instances
- **An object and its roles share identity**
  - ✔ translation polymorhpism hides difference,
use `roleEQ()` for relaxed comparison

Properties of Roles (3/4)

- 15 Criteria by Friedrich Steimann
  - and their mapping to Object Teams
- **Roles restrict access**
  - ✔ accessibility only via callout
- **Different roles may share structure and behavior**
  - ✔ inheritance among roles, or: delegation to base
- **Objects of unrelated types can play the same role**
  - ✔ role type as an a-posteriori super-type




R
R
R
«playedBy»

And now for a Message from our Sponsor ...

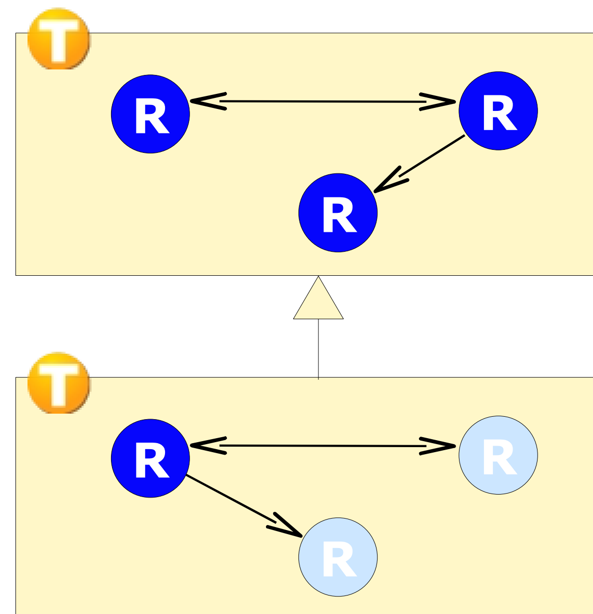
The Object Teams Project

- ObjectTeams/Java (OT/J) — since 2001
  - Java += roles, teams, bindings
  - OTJLD 1.0 *(current 1.2)* — July 2007
- Object Teams Development Tooling — since 2003
  - Java Compiler += OT/J constructs
  - JDT for OT/J (code assist, ui, launch ...)

- Other
  - OT/Equinox: Equinox += aspect bindings — since 2006
  - Application
    - Case studies (project TOPPrax)
    - Class room
    - OTDT
    - UML2 tools (base on EMF/GEF/GMF) — 2009

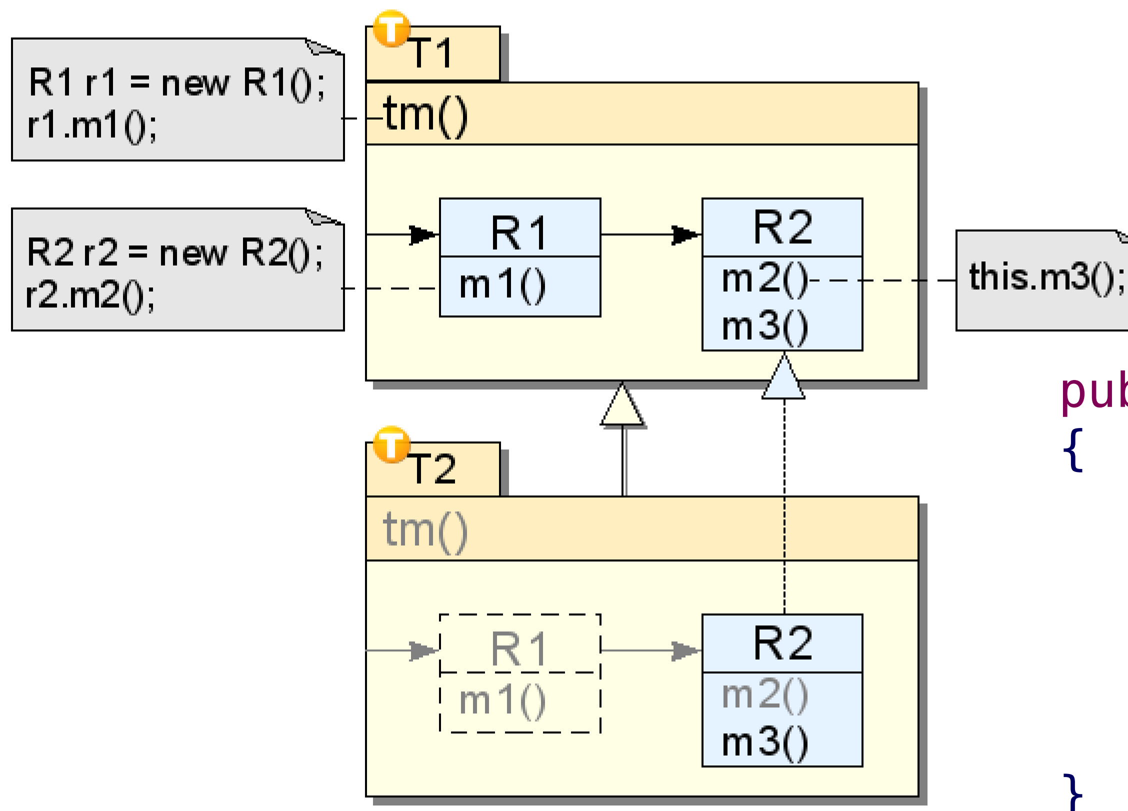
Applying Inheritance to Containment:

Team Inheritance



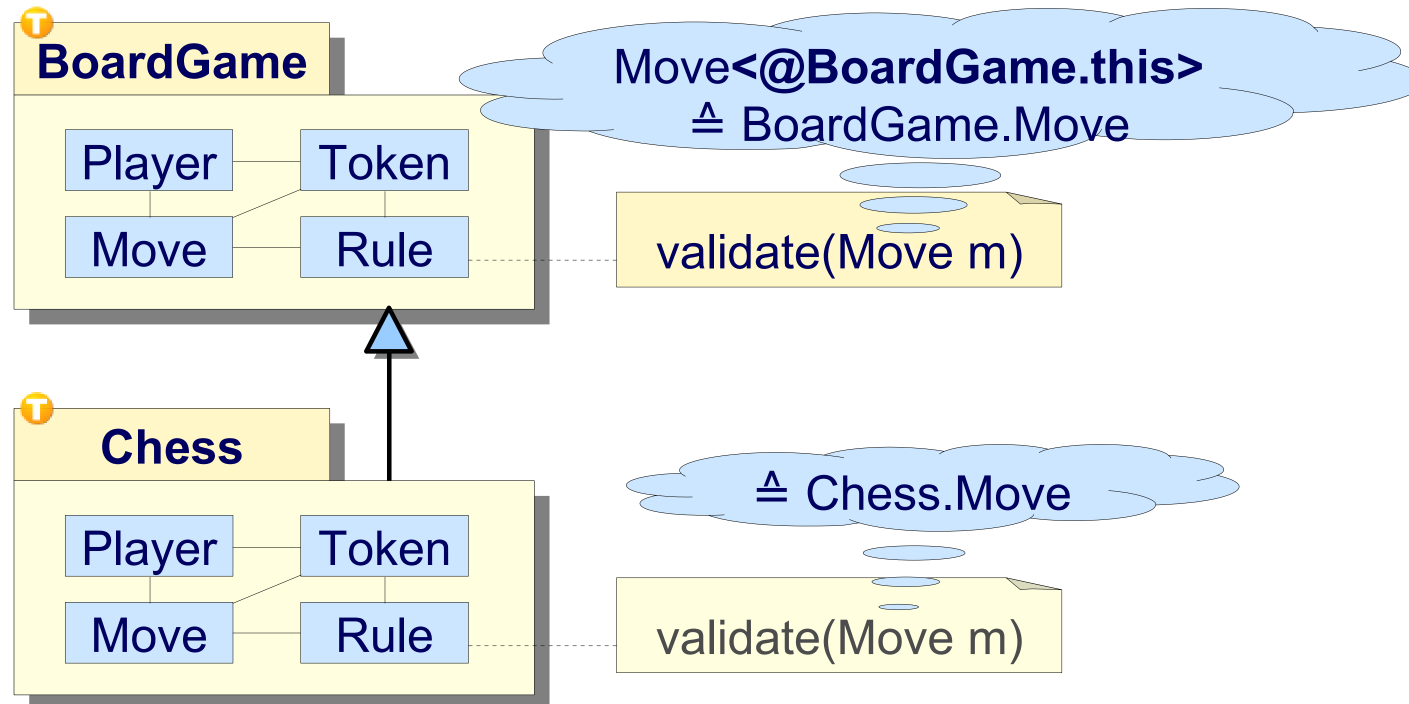
Team Inheritance

- Inheritance = Import, Override, Substitutability
- Attributes, Methods, Role Classes
- **Propagating Specialization**




```
public team class T2 extends T1
{
    protected class R2 {
        void m3() {
            doMyStuff();
        }
    }
}
```

Consistent Polymorphism



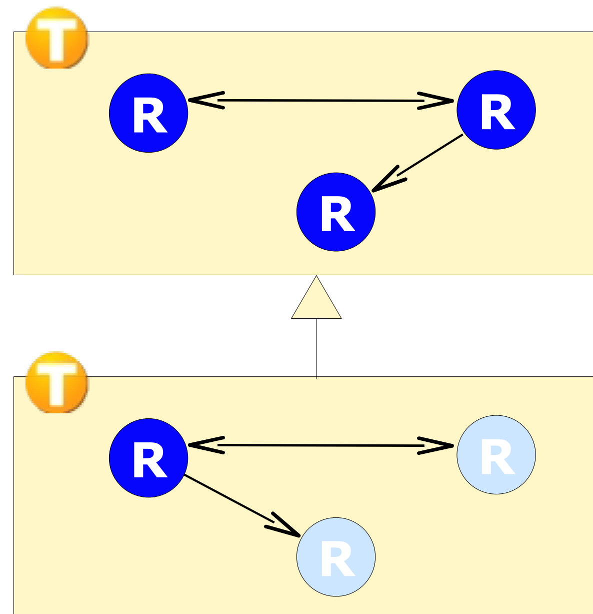
- Virtual classes
  - Inherent covariance
  - Family Polymorphism™
    - type safety through dependent types
- Exception: role migration (to other team)

Class level Template&Hook

```
abstract team class BoardGame
{
    abstract class Player {...}
    Player a;
    void init() {
        a = this.new Player();
    }
}
team class Chess extends BoardGame
{
   class Player {...}
}
```

```
abstract class C
{
    abstract void hook();

    void template() {
        this.hook();
    }
}
class D extends C
{
   void hook() {...}
}
```

Team BoardGame is a template:	incomplete implementation
Role Player is a hook:	opening filled in team Chess


T
R
R
R
T
R
R
R

Applying Translation Polymorphism to Inheritance Structures

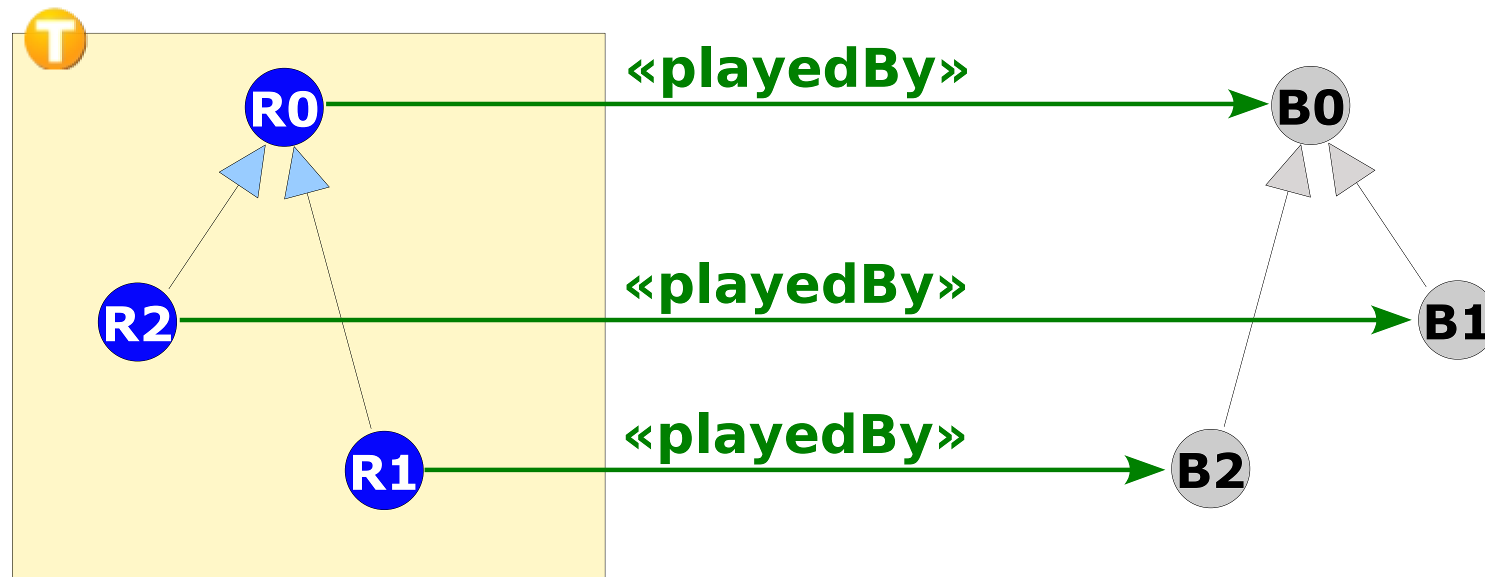
“Smart Lifting”

Most Specific Type

- Attempt #1
  - Connect roots of inheritance trees
  - Let lifting always choose the most specific type
- It works
  - always use R1 for any base ⩽: B0
  - cannot handle multiple subtypes of R0

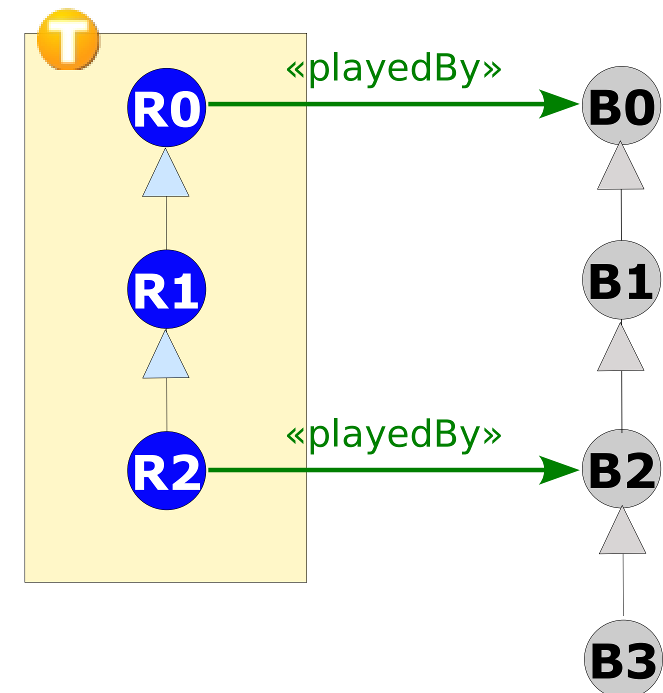
Lifting with Constraints

- Individual **playedBy** declarations
  - constrains lifting to bases of more specific types
  - covariant redefinition of «base»
- Mapping of inheritance structures
  - 1:1



Lifting with Constraints

- Individual **playedBy** declarations
  - constrains lifting to bases of more specific types
  - covariant redefinition of «base»
- Mapping of inheritance structures
  - 1:1
  - ignore sub-base B3
  - insert R1 (R0 never instantiated)
  - skip B1 (subsumed by R0)



Double Dispatch

- Adding instance dispatch to method dispatch

```
team class PrettyPrinter {

    abstract class Expr playedBy Expr {
        abstract void print(); }

    class BinOp playedBy BinOp {
        Expr getLeft() -> Expr getLeft();
        void print() {
            getLeft().print(); /* ... */ }}

    public void visit(Expr as Expr node) { node.print(); }
}
```

Smart Lifting selects:
- Team: *Function*
- Role: *Node Type*

Applying Generics to Role Playing

Generic Callin Bindings

Problem:

- replace callin binding requires 2-way compatibility

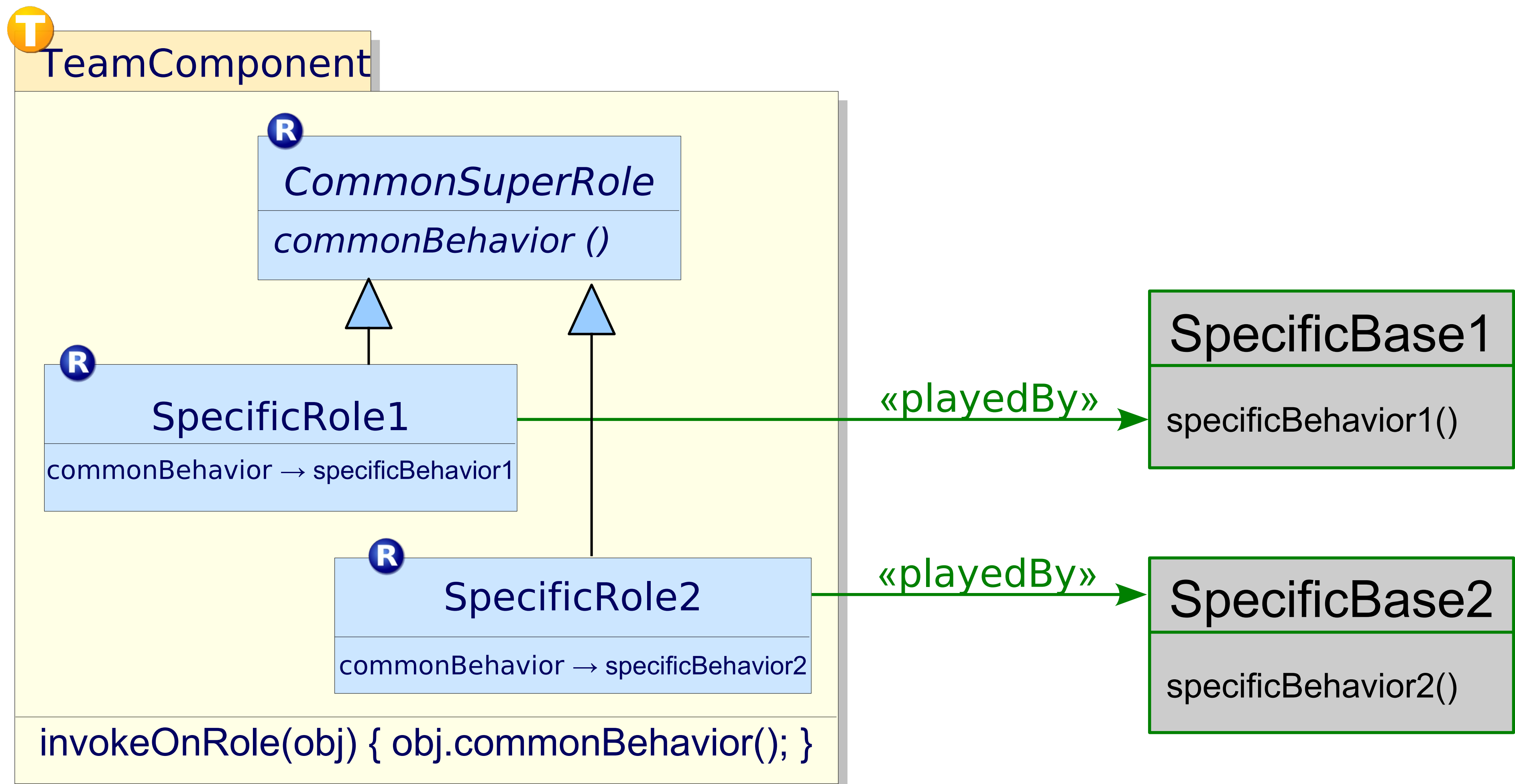
```
callin T1 roleMethod() {
      T1 oldResult = base.roleMethod();
      return new T1();
}
T1 roleMethod() <- replace T1 baseMethod();
```

- Java 5 introduces covariant returns
  - binding to `T2 baseMethod()` fails → ClassCastException
- OT/J enforces the use of generics where needed
  - explicitly capture covariant methods
  - use type bound

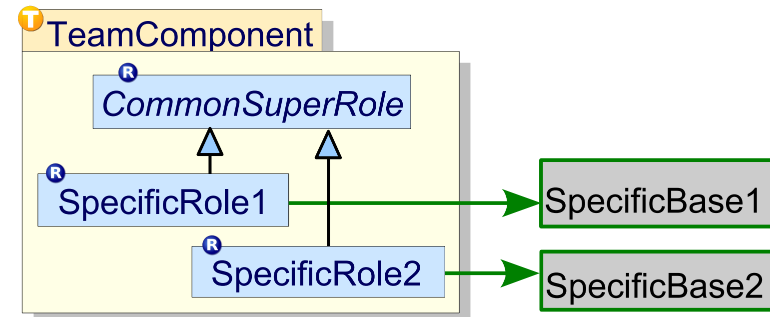
```
callin <E extends T1> E roleMethod() {
      return base.roleMethod(); // OK, but: new T1() NOK
}
<E extends T1> E roleMethod() <- replace T1+ baseMethod();
```

Base Class Generalization

- Recall this structure



Base Class Generalization



- But,
  - how can this method be typed?

```
void invokeOnRole(? as CommonSuperRole anyObj) {
      anyObj.commonBehavior();
}
```

  - want to allow `SpecificBase1` and `SpecificBase2`
- answer:
  - new kind of type bound:

```
<B base CommonSuperRole>
void invokeOnRole(B as CommonSuperRole anyObj) {
      anyObj.commonBehavior();
}
```

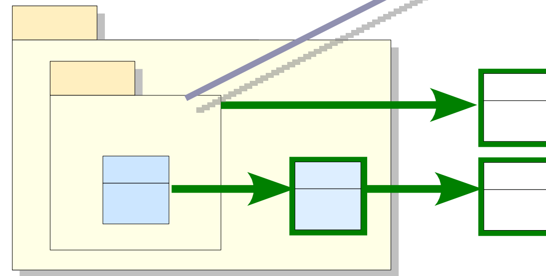
  - B is the union of all classes that can be lifted to `CommonSuperRole`

Composed Structures

Applying Object Teams Concepts to each other

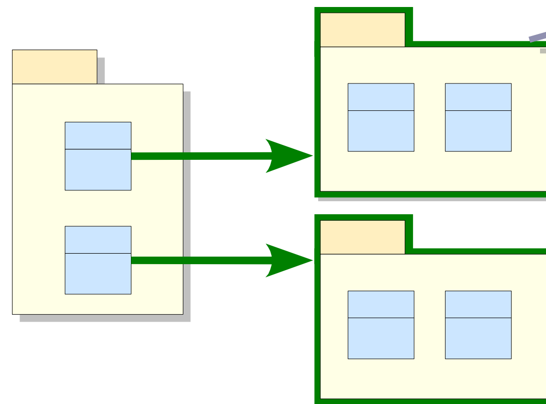
Nesting – Stacking – Layering

- Nesting



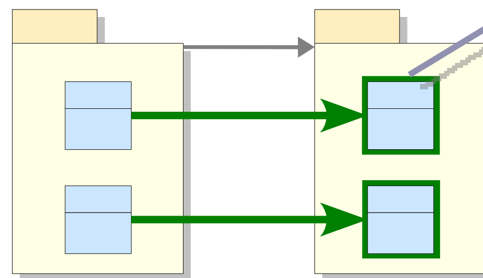
Team plays the role **Role**

- Stacking



Team plays the role **Base**

- Layering



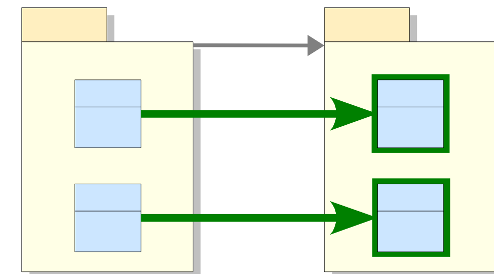
Role plays the role **Base**

Layering – Detail

```
public team class ColoredGraph
{


   protected class CNode
            playedBy Node
   { ... }

   protected class CEdge
            playedBy Edge
   {
      abstract CNode getStart();
      getStart -> getStartNode;
   }
}
```

Missing anchor (team instance) for role type Graph.Node outside its team context (OTJLD 1.2.2(b))

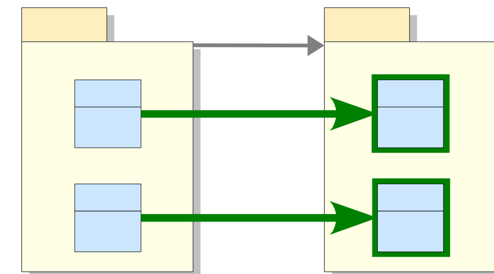
```
public team class Graph
{


      public class Node {
            ...
      }

      public class Edge {
            Node getStartNode()..
      }

}
```



Layering – Detail

```
public team class ColoredGraph
{
   final Graph graph = ...;

   protected class CNode
         playedBy Node<@graph>
   { ... }

   protected class CEdge
         playedBy Edge<@graph>
   {
      abstract CNode getStart();
      getStart -> getStartNode;
   }
}
```

```
public team class Graph
{


    public class Node {
          ...
    }

    public class Edge {
          Node getStartNode()..
    }

}
```



Properties of Roles (4/4)

- 15 Criteria by Friedrich Steimann
  - and their mapping to Object Teams

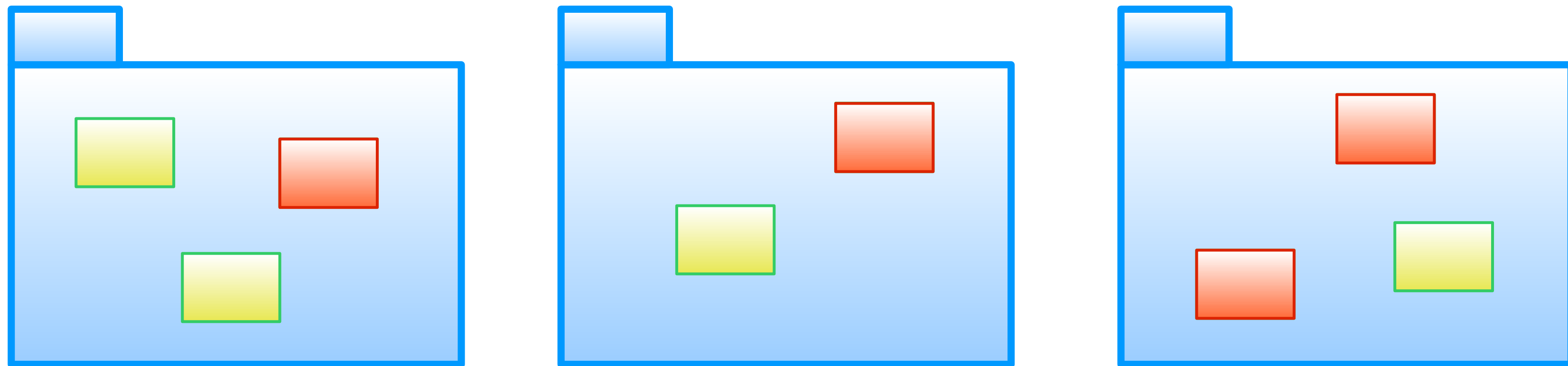
- **Roles can play roles**
  - ✔ use team layering
- **The sequence in which roles may be acquired and relinquished can be subject to restrictions**
  - ✔ role-of-role, guard predicates, role constructor throwing
- **A role can be transferred from one object to another**
  - ✔ use `IBaseMigratable`

```
class President implements IBaseMigratable
                playedBy Person { /* body */ }

void transferPresidency(Person as President   currentP,
                        Person                newP) {
    currentP.migrateToBase(newP);
}
```

Architecture
with
Object Teams

Classes – Packages - Components

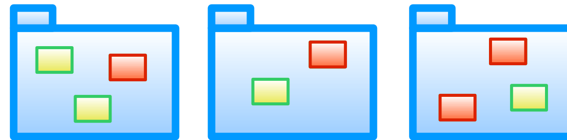
- Traditional decomposition



- New feature requested
  - identify affected classes
  - no way to define new feature as a module ☹

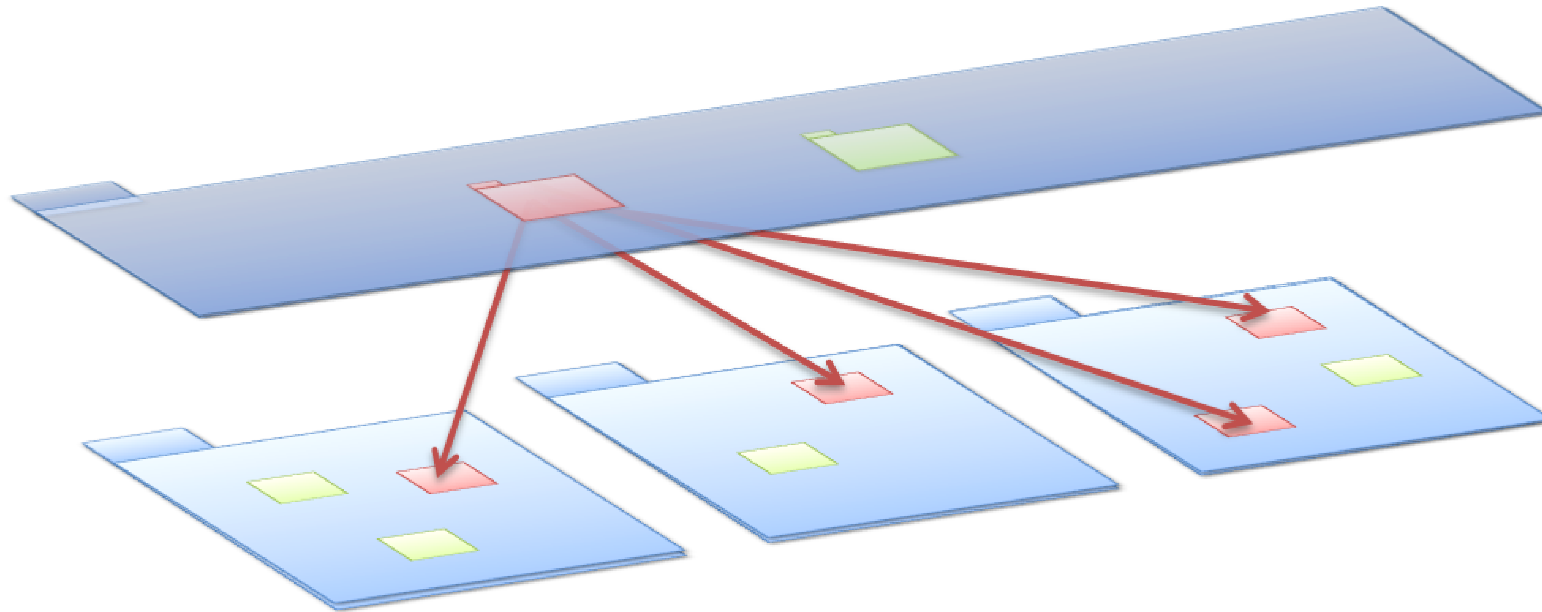
Be Inventive!

- We need to zoom out!

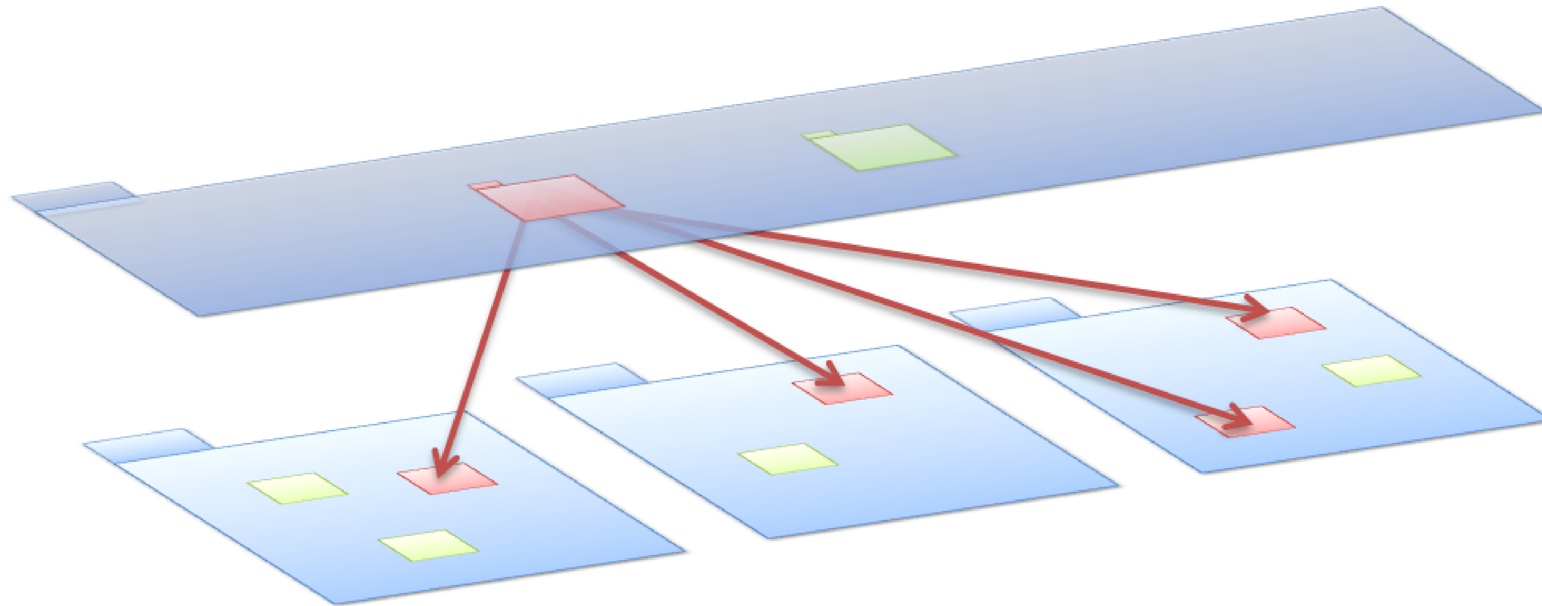


  - See a new solution?
  - No.
  - Try again!

Entering The Next Level

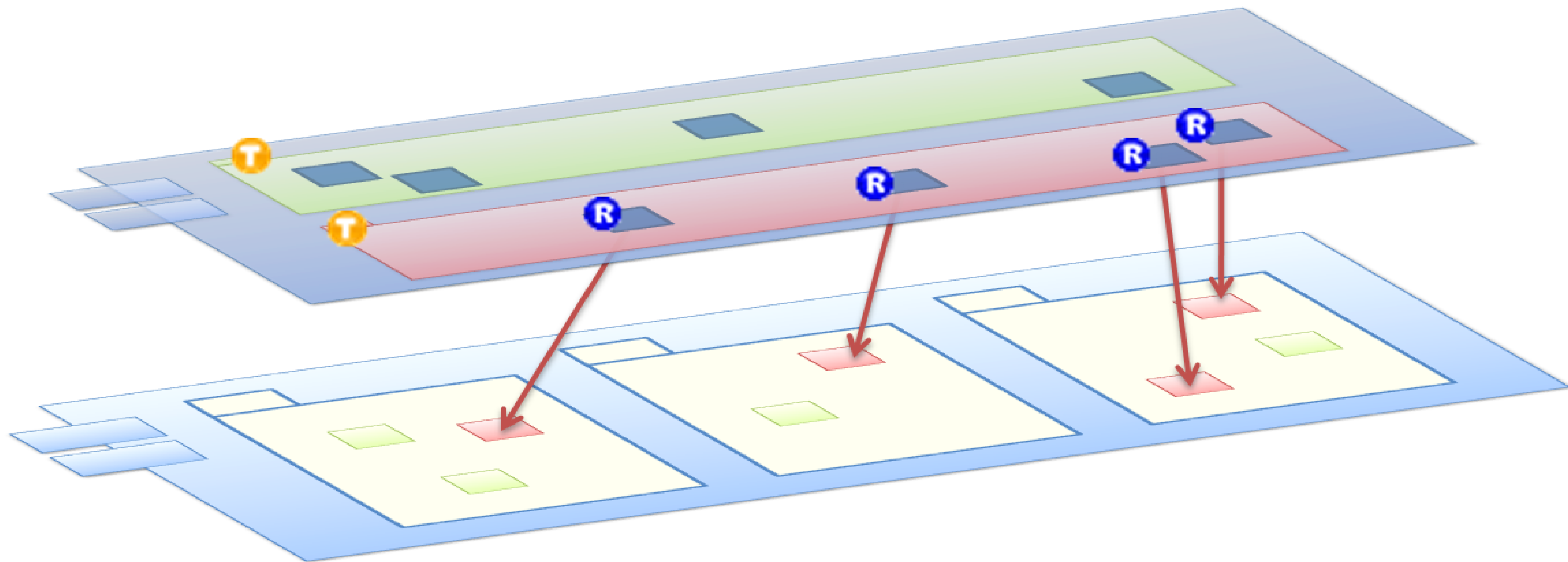


Layered Design



- In truely layered designs
  - each layer may have its very own structure
  - layers are connected to each other
  - mapping
    - can be 1:n
    - exposes / hides elements from other layer

Layers with Object Teams



Mapping

- playedBy — to connect classes / objects
- callin / callout — to connect methods / fields

Modules

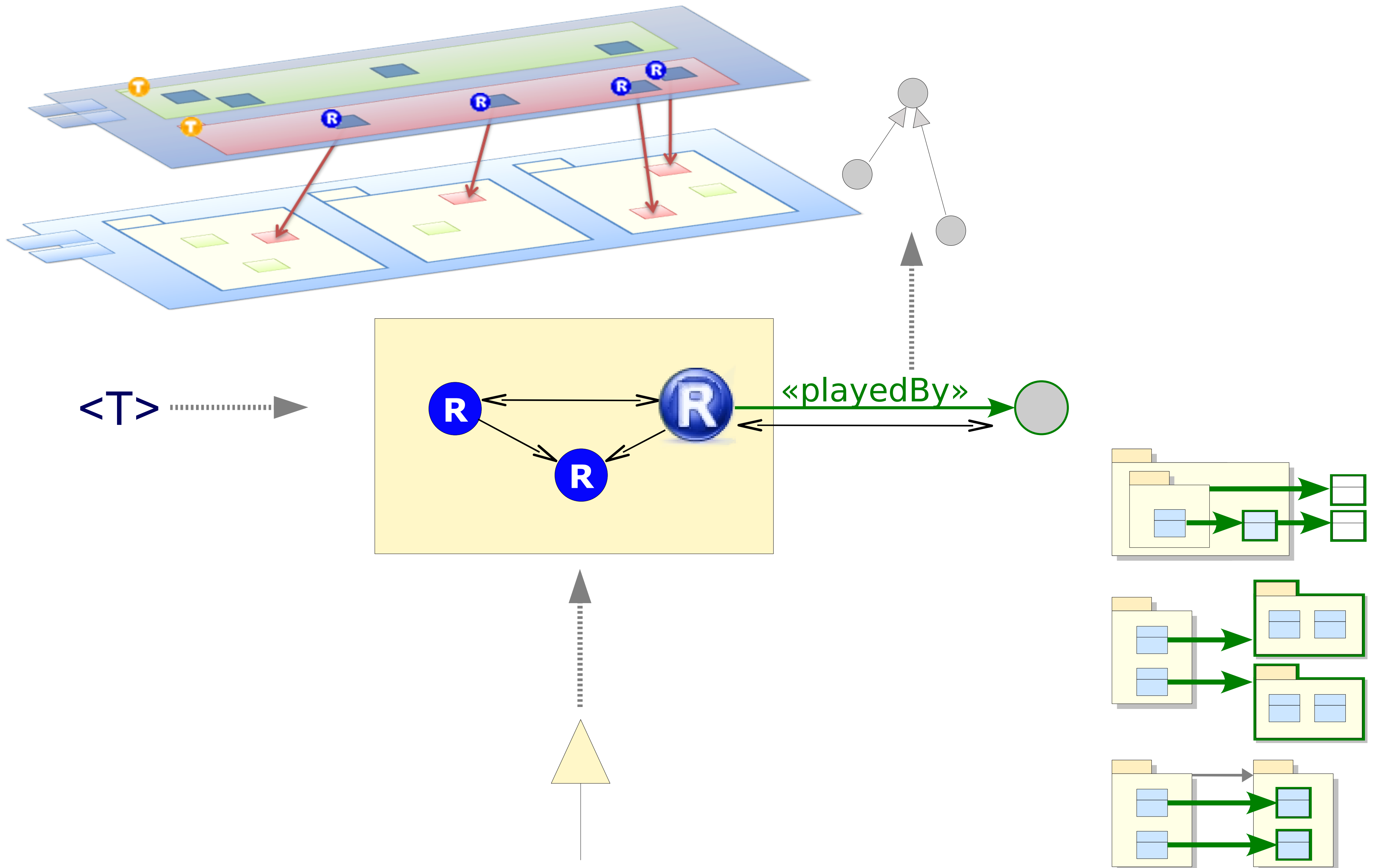
- Role defines view on base class
- Team encapsulates a set of roles

Summary

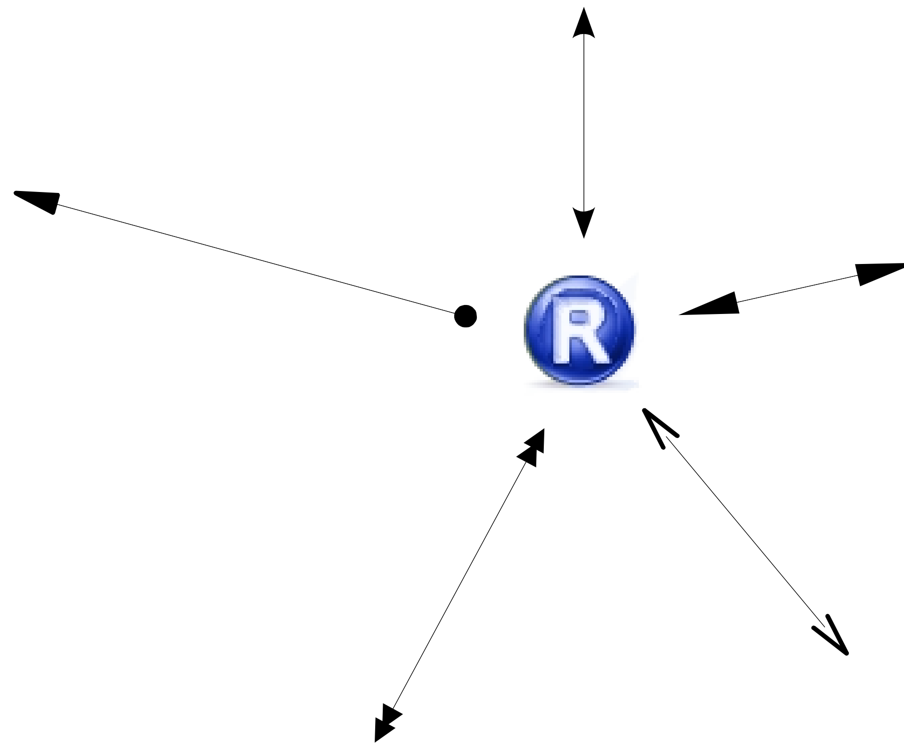
Conclusion

Epilogue

Summary



Summary



Conclusion

- Objectivity
  - Objects are exhaustively defined in one place
  - Definition must consider all special cases
- Subjectivity
  - Consider only relevant properties
  - Different perspectives
    - consider different properties as relevant
    - may express exceptions
  - Express how your perspective relates to „the world“
- Subjectivity in Software Engineering
  - Perspectives during RE
  - Views / diagrams during design
  - In programming: Roles!

Epilogue

- In the end also “Role” is just a **word**
- We may try to define this word
  - as referring to some**thing** out there
- Or we may find it useful
  - when used together **with other words** like “Context”
  - in order to create a new **language game**