ECOOP 2009 Summer School

Object Teams:
Programming with Contextual Roles

Dr. Stephan Herrmann
Independent

www.objectteams.org

A Conservative Revolution

Object Teams ...

- shows respect for O-O principles
  - all new concepts must smoothly fit into O-O
- takes O-O to the extreme
  - fully elaborate the powers of
    - objects, inheritance, composition ...
- adds one new dimension
  - objectivity
    - an object is an object is an object
  - subjectivity
    - selective views with specific purposes

Adding Roles to O-O

Roles are a seamless extension of O-O

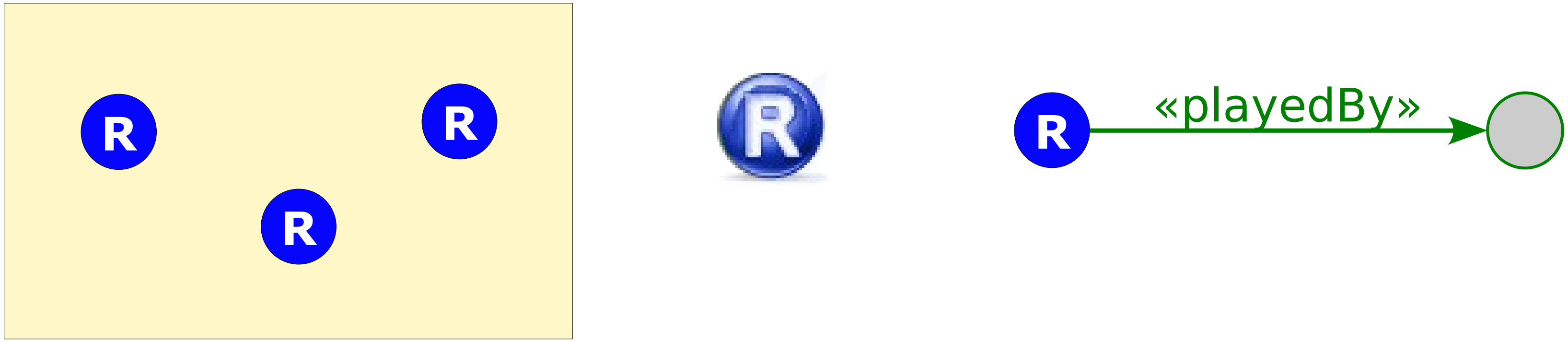
- classes & objects & roles ?
    - these are boring!
- what's happening between these things?
    - **association**
        - composition / containment (stricter semantics)
    - **inheritance**
        - delegation (more flexible)
        - nested inheritance (larger scale)
    - **interactions**
        - 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
- Application of inheritance to the above
  - inheritance of composed structures
    - virtual classes & family polymorphism
- Application of playedBy to inheritance structures
  - mapping between different structures
    - smart lifting & translation polymorphism

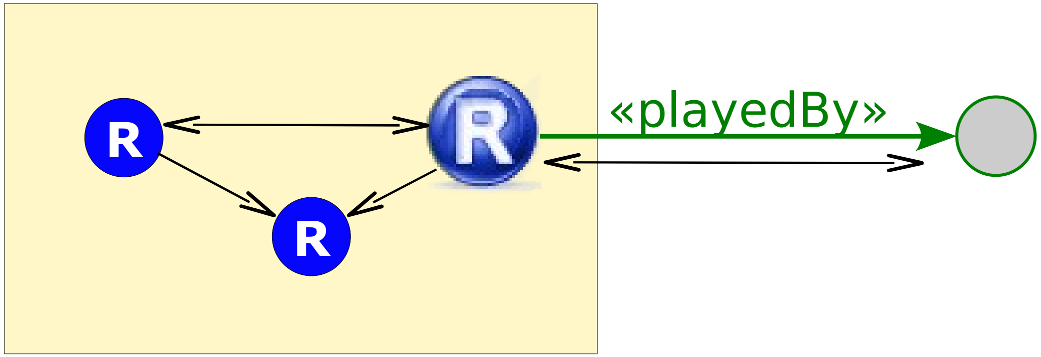
Coherence

- 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
  - emphasize objects over classes
  - introduce subjectivity int
  - are broadly explored
- The role metaphor
  - transcends its origin
- Roles entail
  - the concept of Contexts
- 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"**

Definitions of Roles

- Sowa [1984]
  - natural types
    - “relate to the essence of the entities“
  - role types
    - “depend on an accidental relationship to some other entity“
- Guarino [1992]
  - natural type
    - rigid, lacks foundation
      - *being a Person doesn't change over time,*
      - *does not depend on relationships*
  - role
    - founded, lacks semantic rigidity
      - *being a Student depends on an enrollment relationship*
      - *can change over time without loss of identity*

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

Properties of Roles (1/5)

- 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)

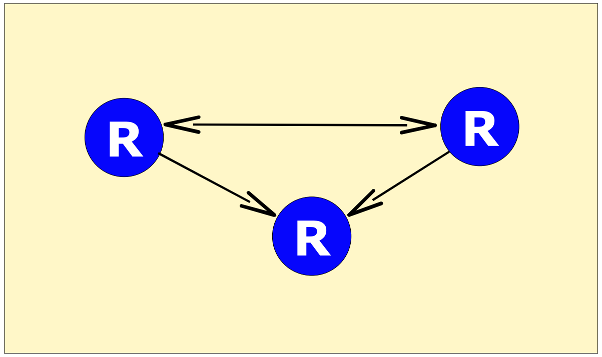
Properties of Roles (1/5)

- 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**
  - ✓ roles have state, contribute to state of compound object
- **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
  - Ja
- Mak
  - de   es

Fundamentally install
instance based access control

- Every role type is a dependent type

Hide complexity for the default case

- Within its team the type anchor can be omitted

```
CoffeeMachine<@Department.this>    OK
CoffeeMachine<@yourDepartment>     illegal
```

Stricter Alias Control

- Ownership could leak through polymorphism
  - every (dependent) type <: `Object`?
    - new top-level types: `Confined`, `IConfined`
    - protected sub-classes of `Confined` cannot leak
    - restricted inheritance: **reuse**, yet preserve “anonymity”
- Ownership may be too strict
  - compromises
    - accessible by empty interface IConfined:
      - opaque, featureless roles
    - grant `readonly` access
      - 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
  - cannot create large structures
  - cannot reason about structure

} Objects!

- Role containment
  - adds strict composition / ownership
  - adds intermediate variants
  - connects ownership to the role/context metaphor
- Make this the foundation for other concepts

Language Design Principle (1)

- Restrictions first
  - basic structures must dominate
    - e.g., **roles** are allways immutably attached to a base
- Flexibility first
  - concepts have to support many designs
    - e.g., dynamically attach/detach roles to a **base**
- Exceptions second
  - exceptions to restrictions
    - e.g., some roles may be re-attached
  - exceptions to flexibility
    - e.g., optimize unused flexibility

Language Design Principle (2)

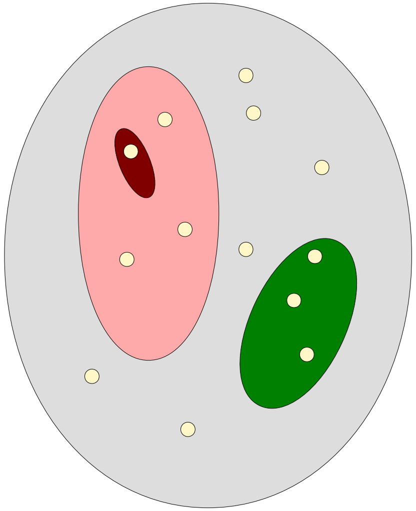
Respect your host

- ObjectTeams/Java behaves to the rules of Java
  - some rules hurt
  - yet, breaking customs hurts more
- Secondary concepts to consider:
  - modifiers: static, private, ...
  - constructors
  - overloading
  - threads
  - exceptions
  - generics

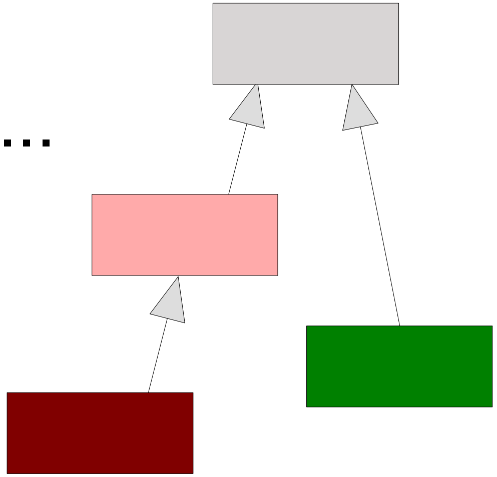
Reconsider O-O Basics: Generalization

Generalizing Inheritance

- Generalization = „is_a“
  - Classification with super sets / sub sets
  - Supports abstraction
    - use super set to **subsume** all sub sets
    - elements of sub set **share** properties of super
  - Supports specialization
    - use sub set to be specific
    - sub set defines more properties (exceptions?)

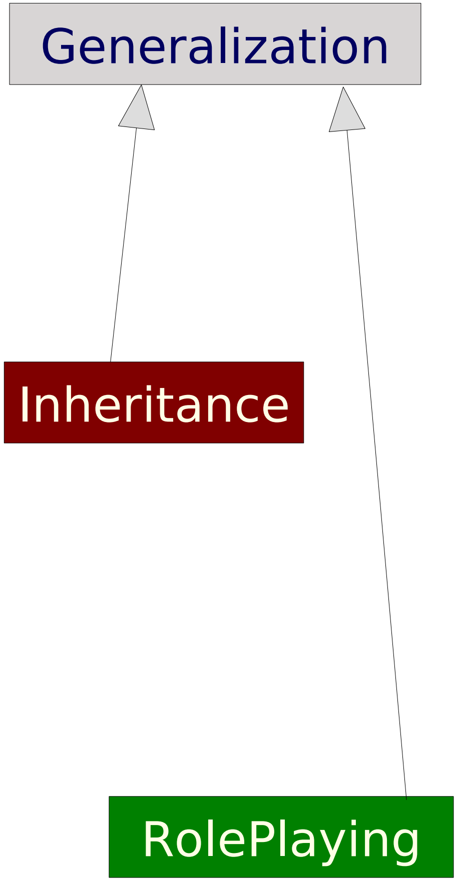


- Inheritance realizes generalization
  - Classification, abstraction, specialization ...
  - Inheritance is **rigid**
    - classification determined at birth once and for ever
- Non-rigid generalization?



Summary

- Generalization = „is_a“
  - generalizes over ...
- Inheritance
  - rigid
    - single type
    - determined at birth
  - focus on classes
- Role Playing
  - anti-rigid
    - multiple specialization
    - dynamic
  - focus on objects



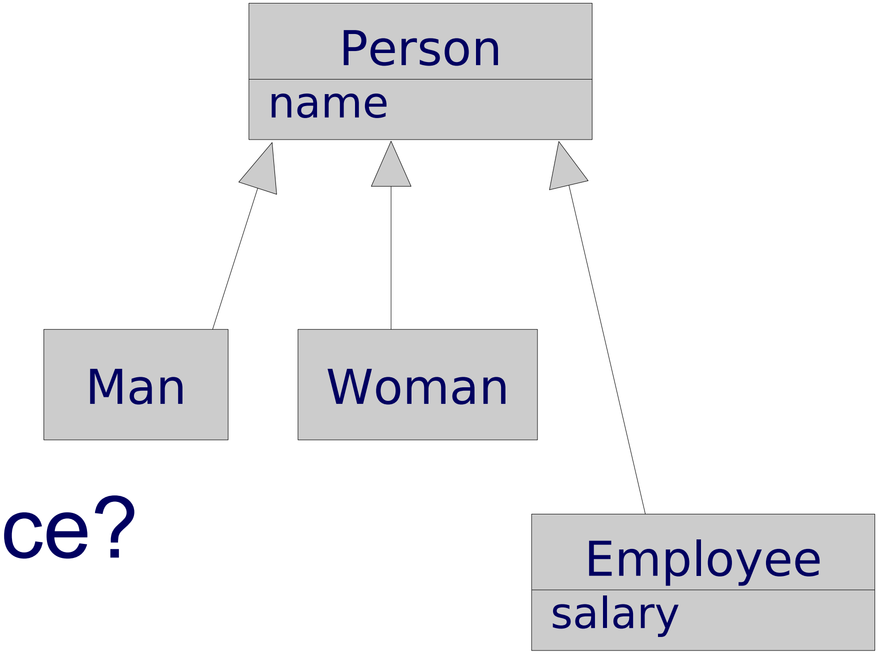
Comparing
Inheritance vs. Role Playing

Design Choices

Generalization ≠ Inheritance

A naive (textbook?) example

- A man/woman **is a** person, OK
- An employee **is a** person, OK?
  - Born as an employee?
  - Dying when loosing the job?
  - Several jobs, yet only one salary?
  - What gender do employees have?



Whats wrong with inheritance?

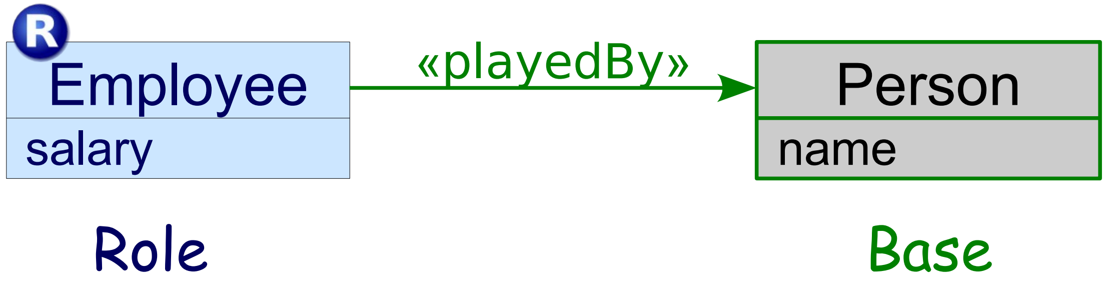
- Missing “become”, “quit” ☹
- Can't duplicate fields ☹
- Only one most-specific type/object ☹
- Employee & Person = 1 instance ☺

Missing support for

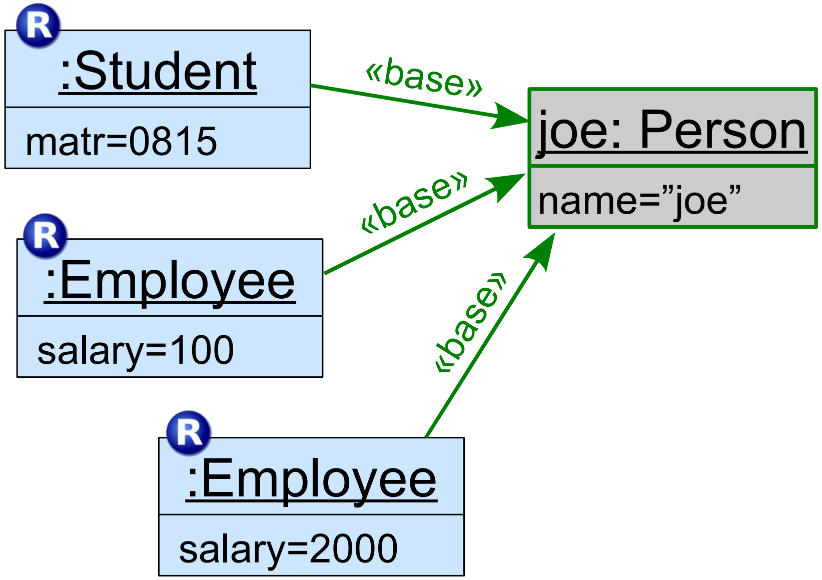
- changes over time
- flexible combinations & multiplicities

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]



Inheritance vs. PlayedBy in OT/J

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

Goal:

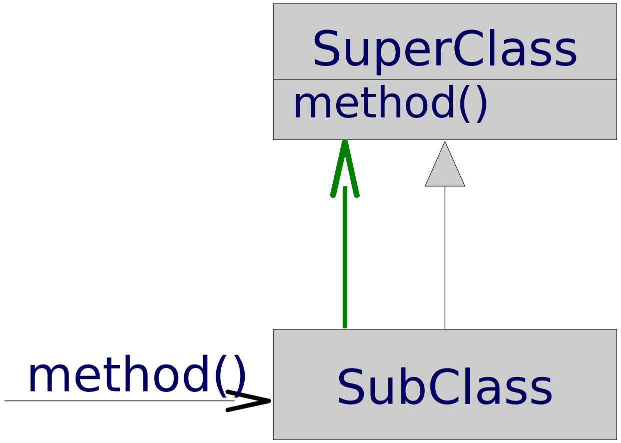
less implicit coupling

- independent evolution

Inheritance vs. PlayedBy in OT/J

Inheritance

- Import
  - **dispatch sub → super**

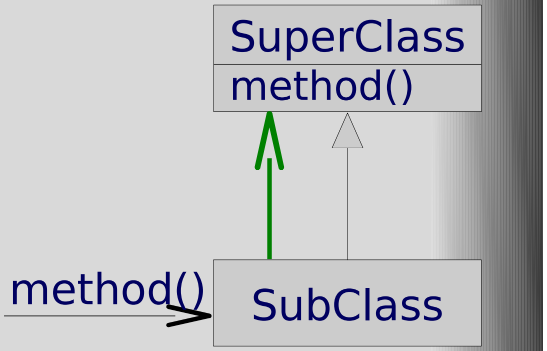


Role Playing

Inheritance vs. PlayedBy in OT/J

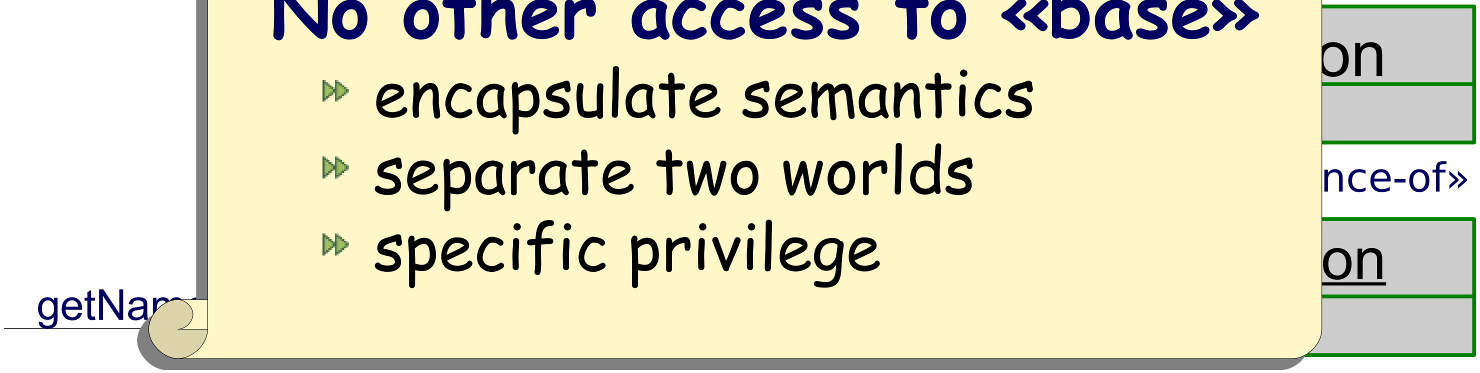
Inheritance

- Import
  - **dispatch sub**



Role Playing

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



No other access to «base»

- encapsulate semantics
- separate two worlds
- specific privilege

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

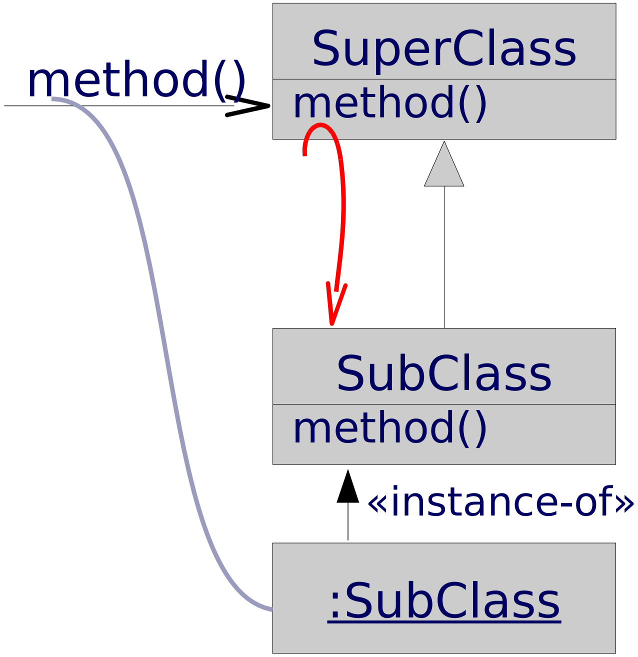
§§

- 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**



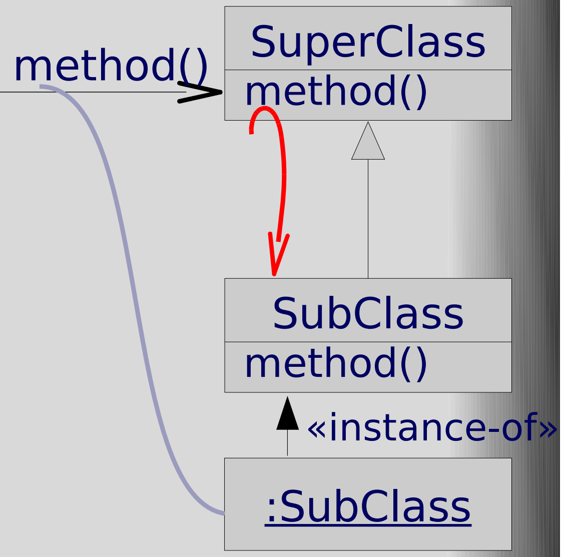
Role Playing

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

Inheritance vs. PlayedBy in OT/J

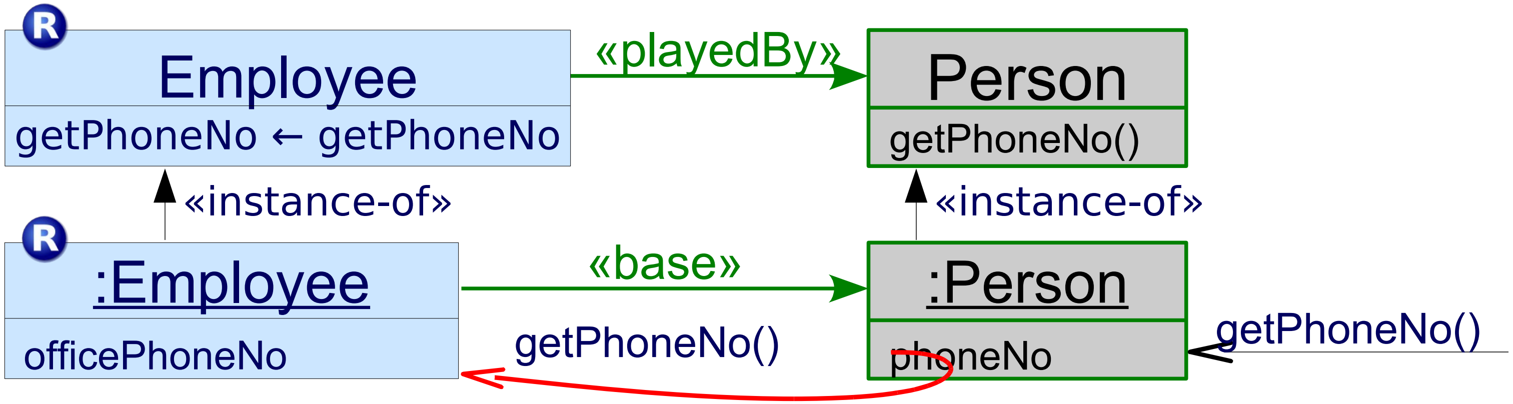
Inheritance

- 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

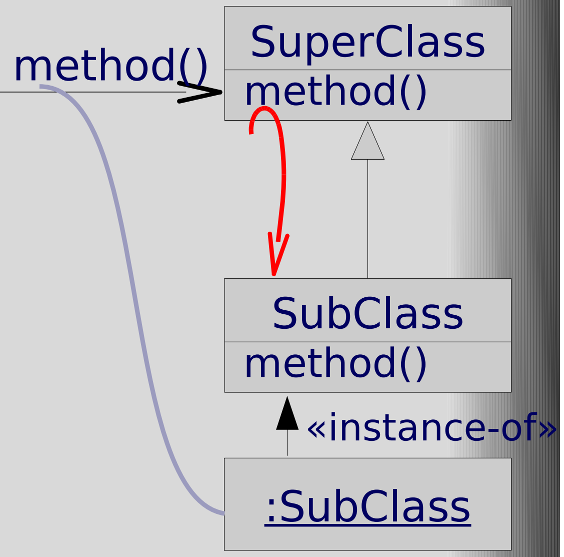
Inheritance vs. PlayedBy in OT/J

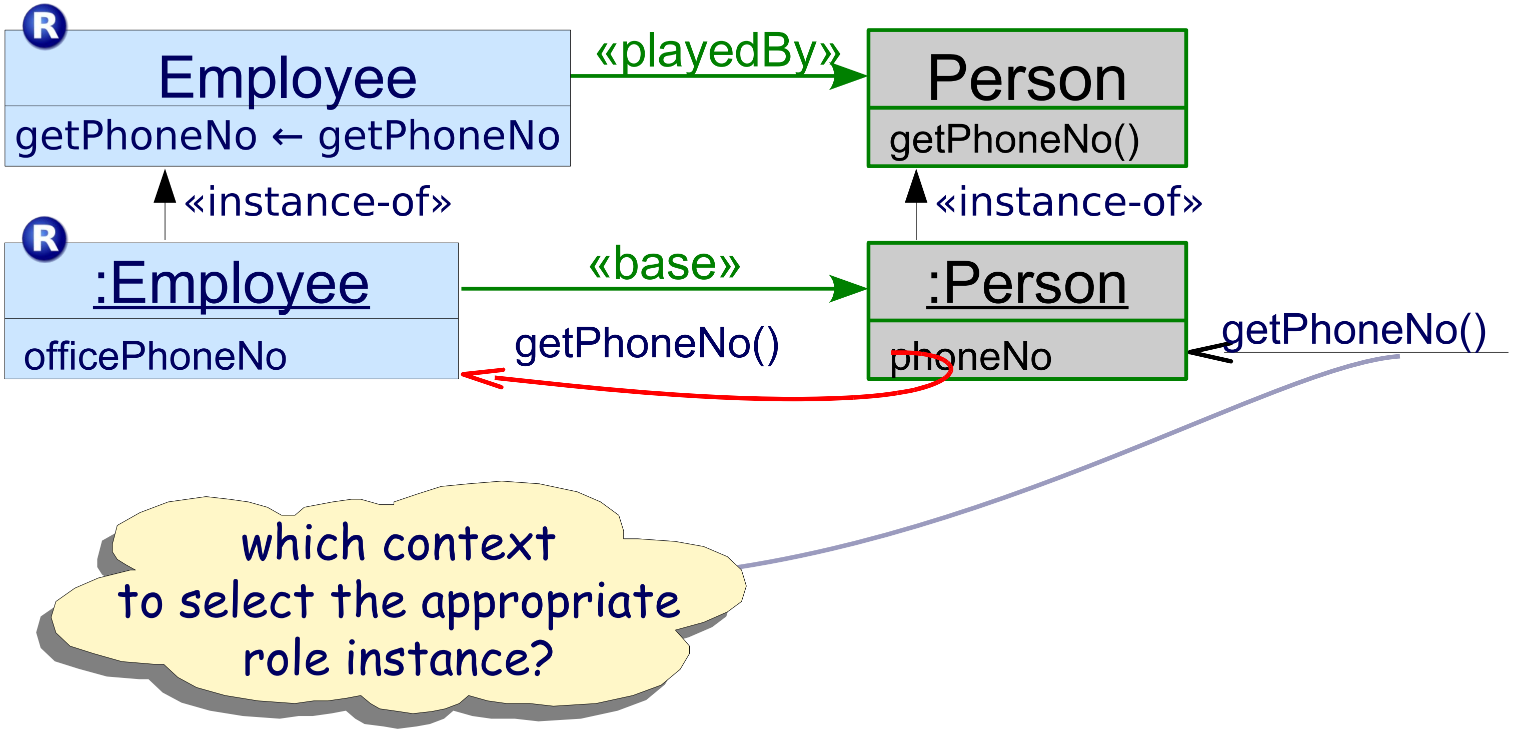
Inheritance

- 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?

Inheritance vs. PlayedBy in OT/J

Detailed Comparison

Inheritance

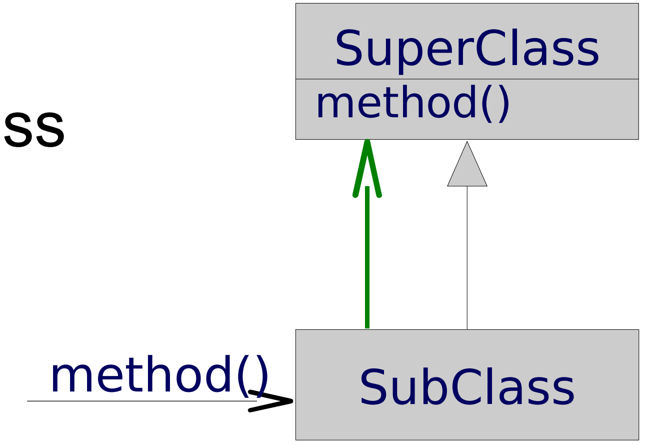
- Import
  - dispatch sub → super
- Overriding
  - dispatch super → sub
- Substitutability
  - pass an instance of sub class where the super class is expected

Role Playing

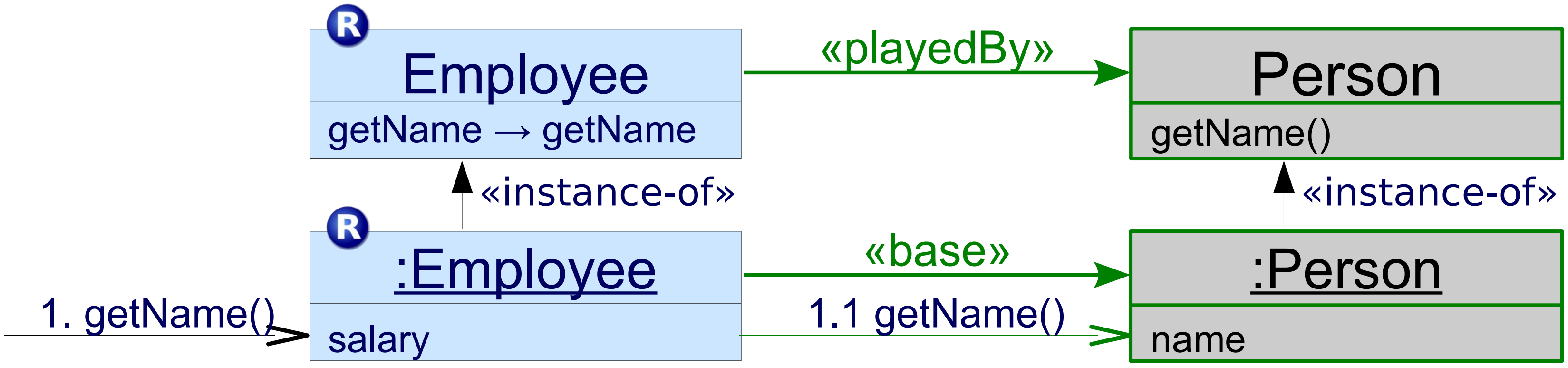
?

Inheritance (1): Import

- Sub-class imports from super-class
  - all members
    - except private
  - accessibility / scoping
    - extends the scope of the sub-class
  - renaming?
    - only in few languages
  - interpretation
    - forwarding sub → super (classes)

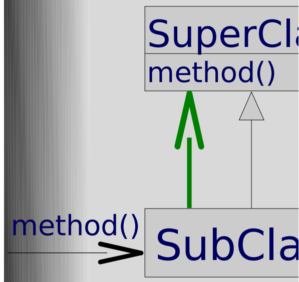


Role-Playing (1): Import

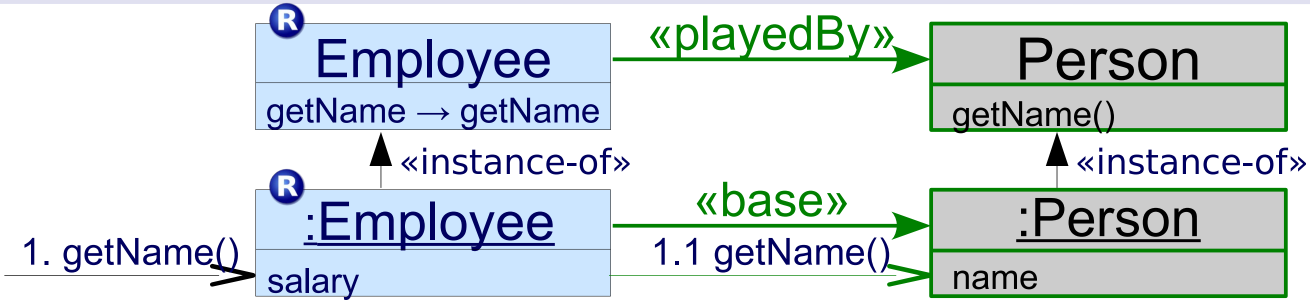


- Role-object imports from base-object
  - only by declared **callout binding**
    - inference as an option
  - accessibility / s
    - extends the sco the role-object
  - renaming
    - as part of callout binding (incl. parameter mapping)
  - interpretation
    - forwarding role → base (objects)

Supporting Evolution



Import in OT/J:



A **callout method binding**

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

- ... can use
- ... can adju
  - implicitl
  - explicitl

A callout to

```
String getN
```

No other access to «base»

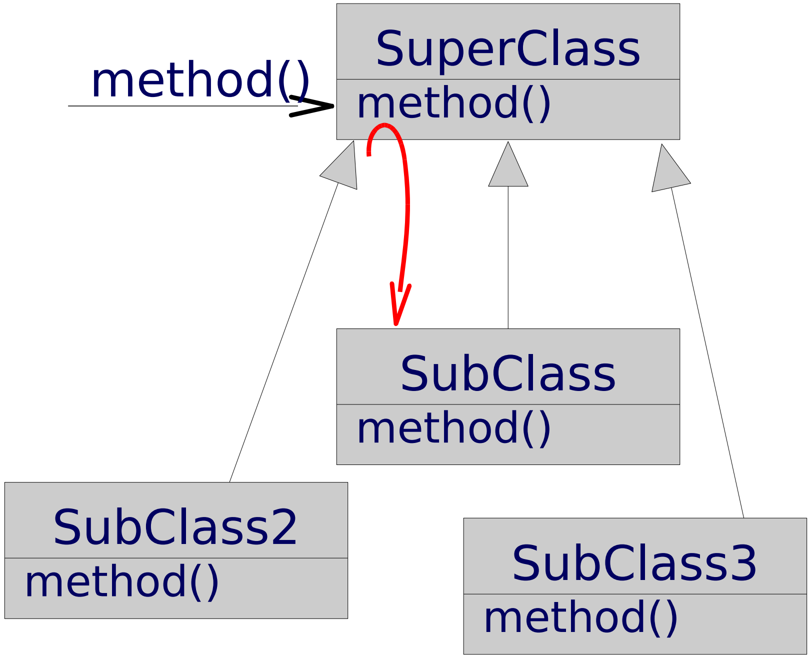
- encapsulate semantics
- separate two worlds
- specific privilege

Inferred callout

- for self calls
- for methods declared in a common interface

Inheritance (2): Overriding

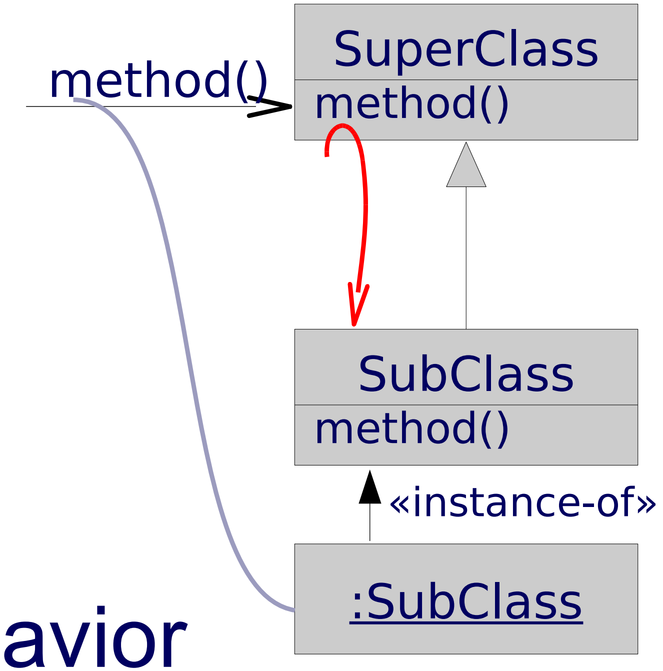
- Sub-class overrides super-class behavior
  - by name equality
    - except private, final
  - renaming?
    - only in few languages
  - interpretation
    - interception super → sub



- BUT
  - who selects among multiple sub-classes?

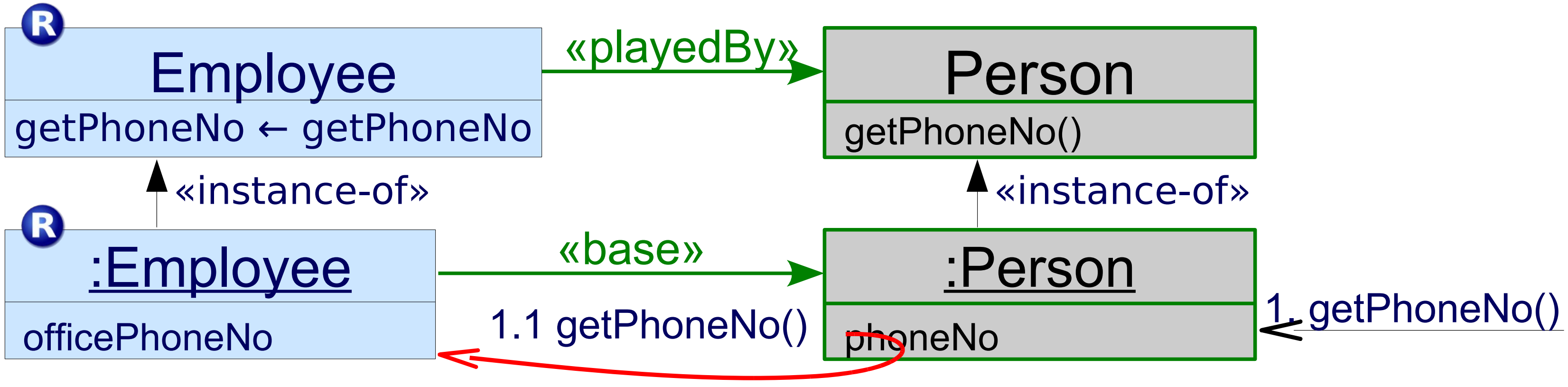
Inheritance (2): Overriding

- Sub-class overrides super-class behavior
  - by name equality
    - except private, final
  - renaming?
    - only in few languages
  - interpretation
    - interception super → sub



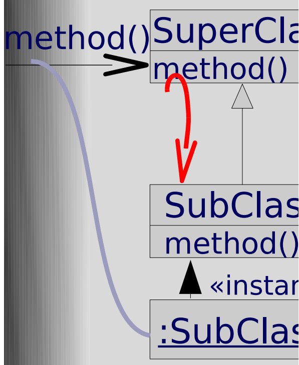
- Dynamic context selects behavior
  - the dynamic type of the current object

Role-Playing (2): Overriding

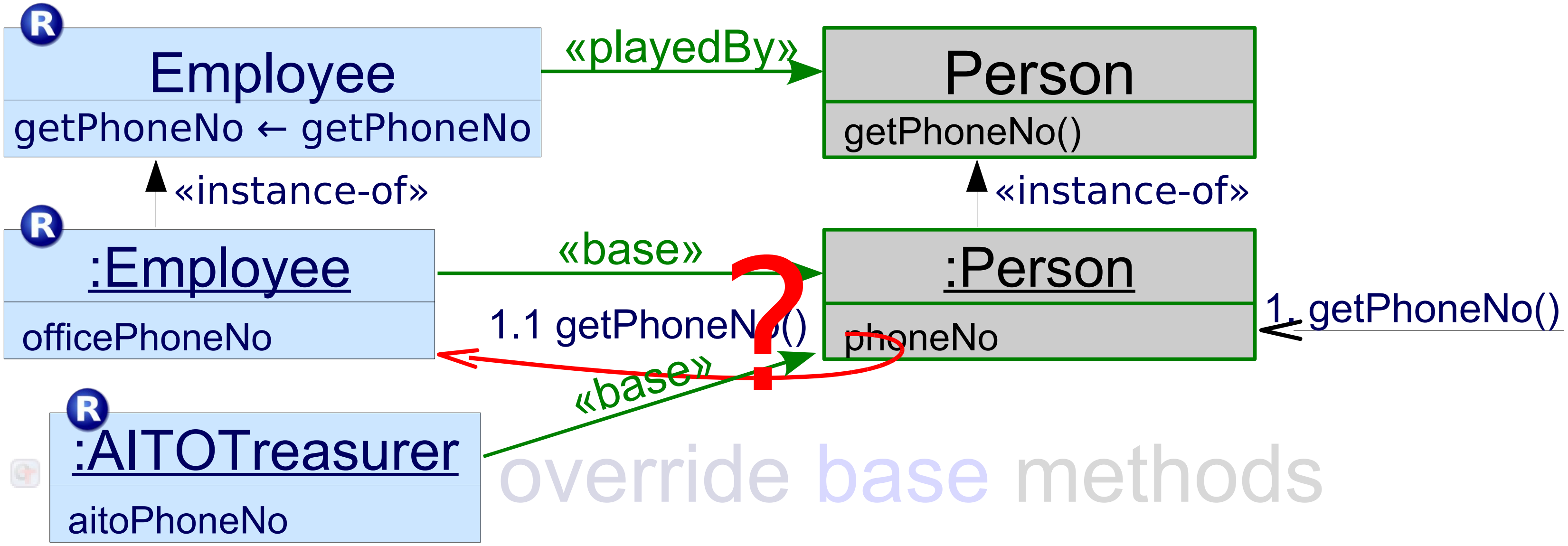


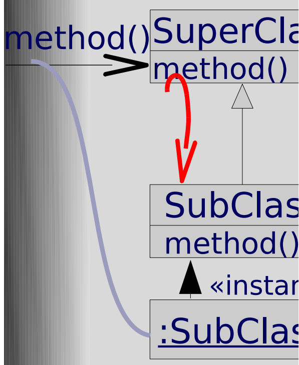
- Role-object override base methods
  - only by declare
    - <no exception>
  - renaming
    - as part of callin binding (incl. parameter mapping)
  - interpretation
    - interception role ← base (objects)

Supporting Evolution



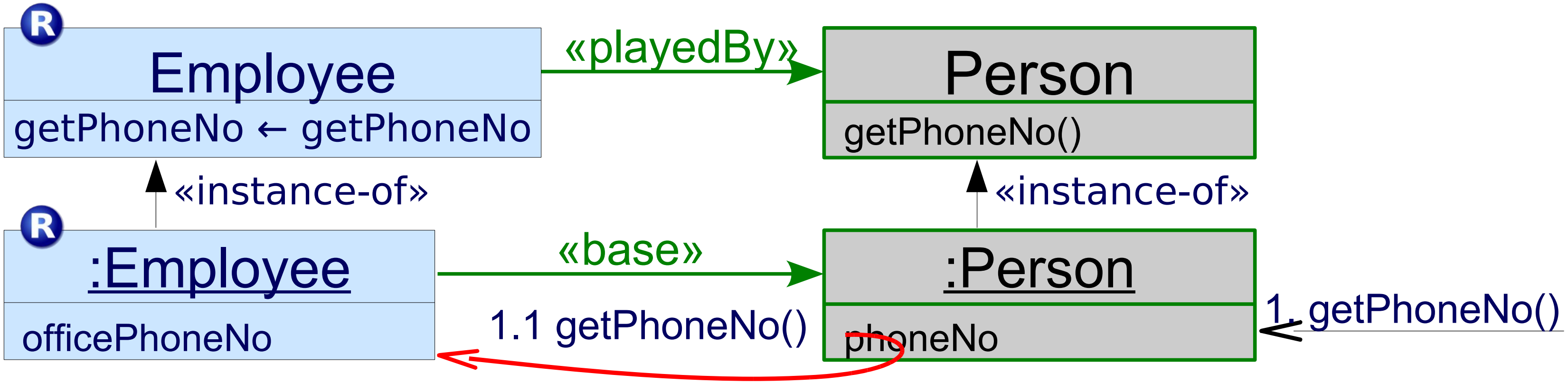
Role-Playing (2): Overriding



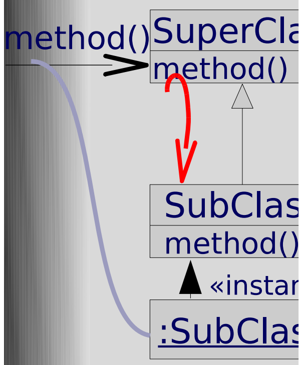


- override base methods
  - only by declared **callin binding**
    - <no exception>
  - renaming
    - as part of callin binding (incl. parameter mapping)
  - interpretation
    - interception role ← base (objects)
- **BUT**
  - who selects among multiple base objects??

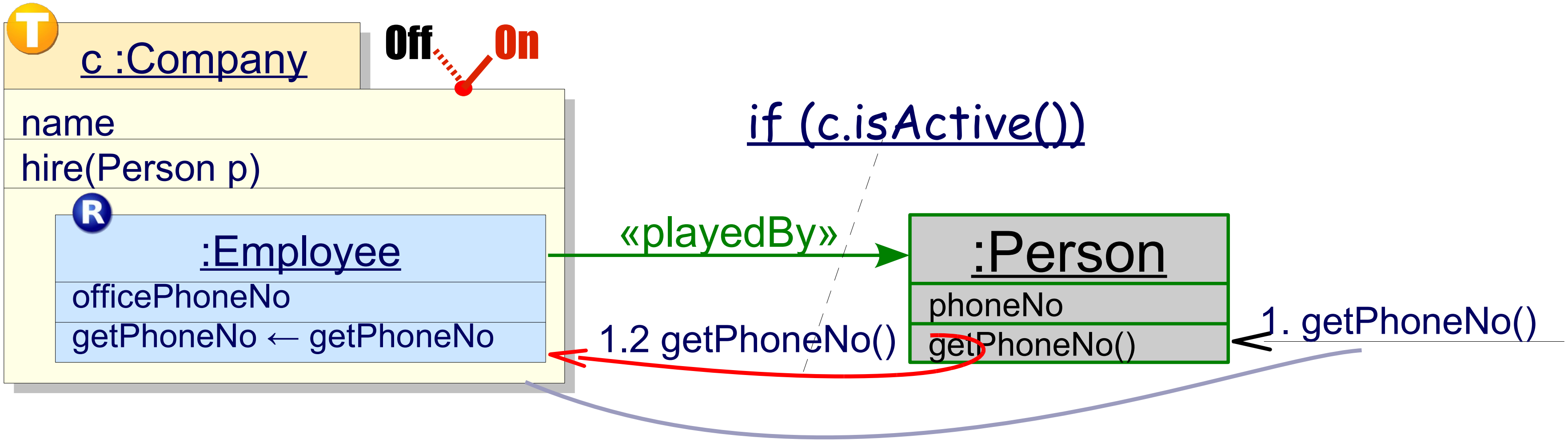
Role-Playing (2): Overriding



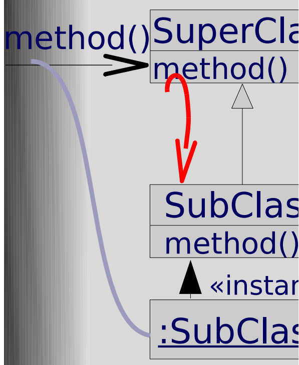
- Role-object override base methods
  - only by declared **callin binding**
    - <no exception>
  - renaming
    - as part of callin binding (incl. parameter mapping)
  - interpretation
    - interception role ← base (objects)
- Dynamic context selects behavior
  - role objects live **in a team object**



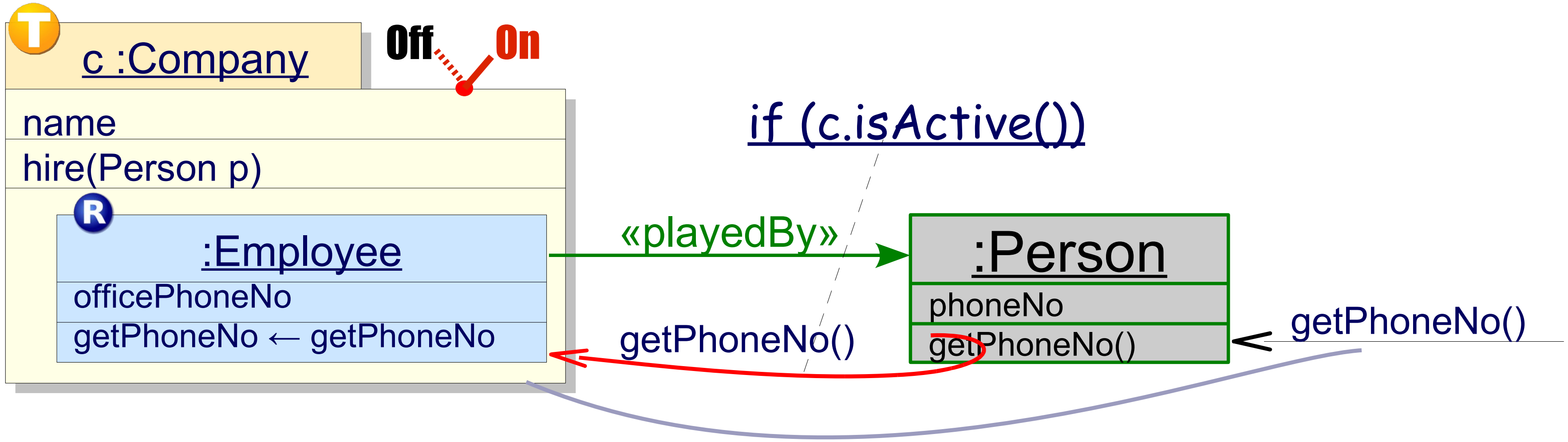
Teams as Context



- **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
  - several mechanisms: globally, per thread, implicitly, temporarily ...



Teams as Activation Context

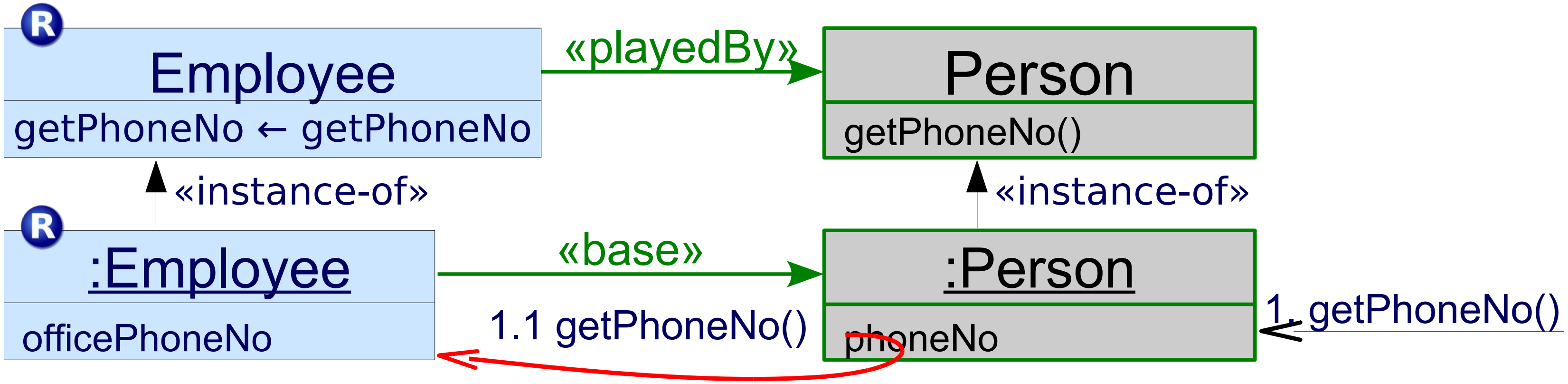


- **Roles depend on context**
- In OT/J contexts are reified as **Teams**
  - roles are inner classes of a **team class**
  - role instances are inner instances of a **team instance**
- Each team instance can be **(de)activated**
  - active team instances contribute to the **system state**
  - dispatch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

§§ activation mechanisms:

- globally
- per thread
- implicitly
- per block

Overriding in OT/J



- A **callin method binding** ...

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

... declares that calls to the base should be **intercepted** by its role

- ... can use different names on role / base sides
- ... can adjust signatures
  - implicitly / explicitly
- ... can have a guard predicate: `when (expr)`
  - Event / Condition / Action

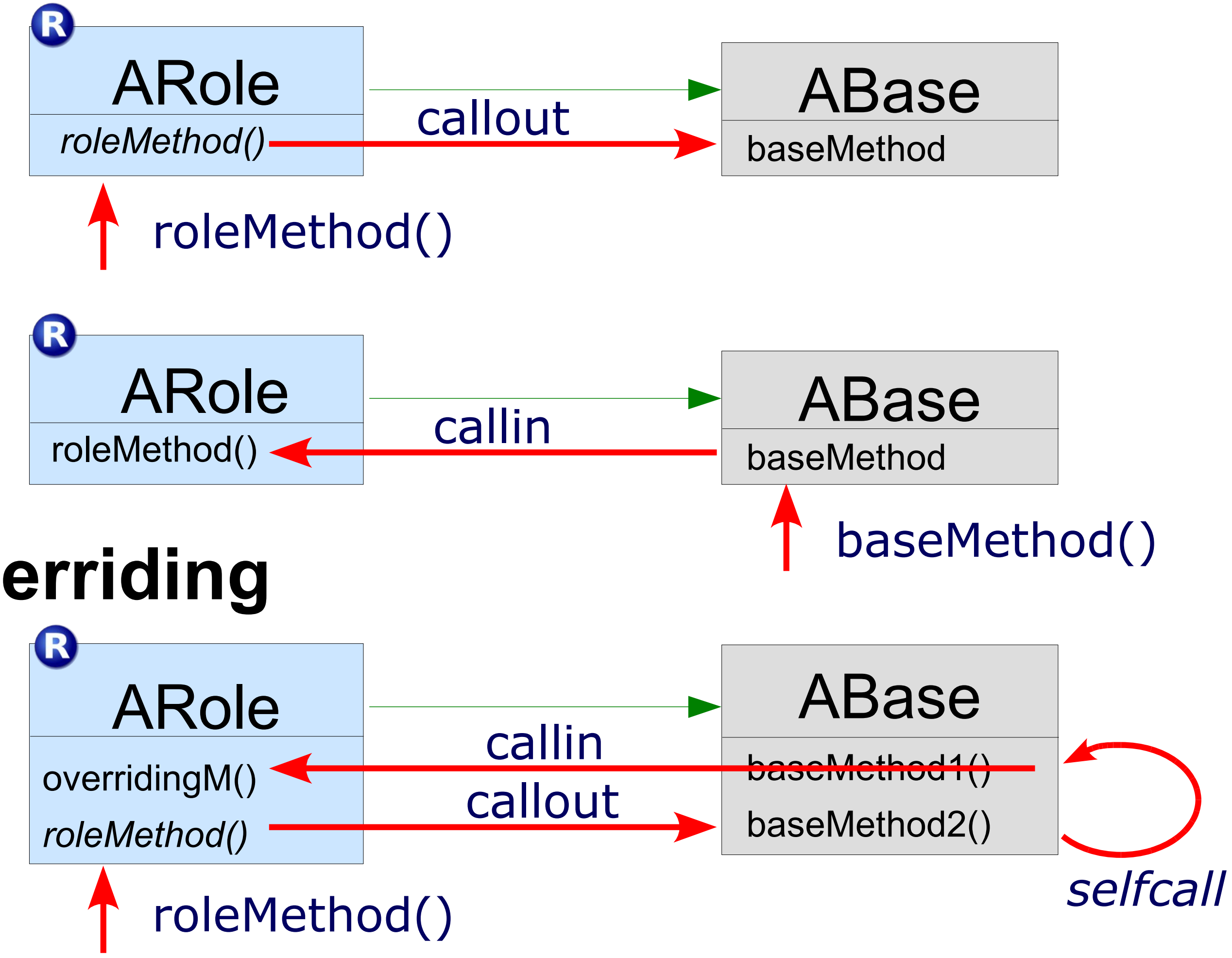
- Binding variants
  - **before**, **replace** or **after**

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

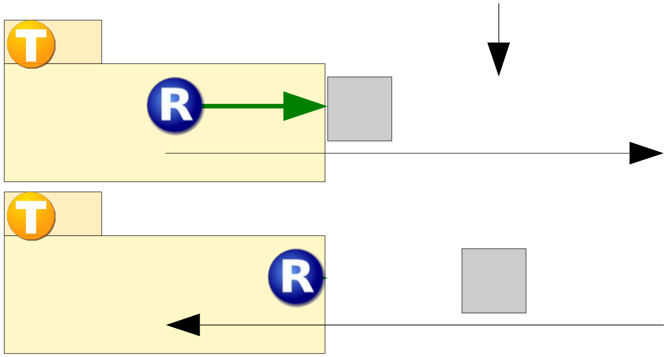
- Are the following assignments legal?

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

Normally not, but...

roles (usually) live within the team only

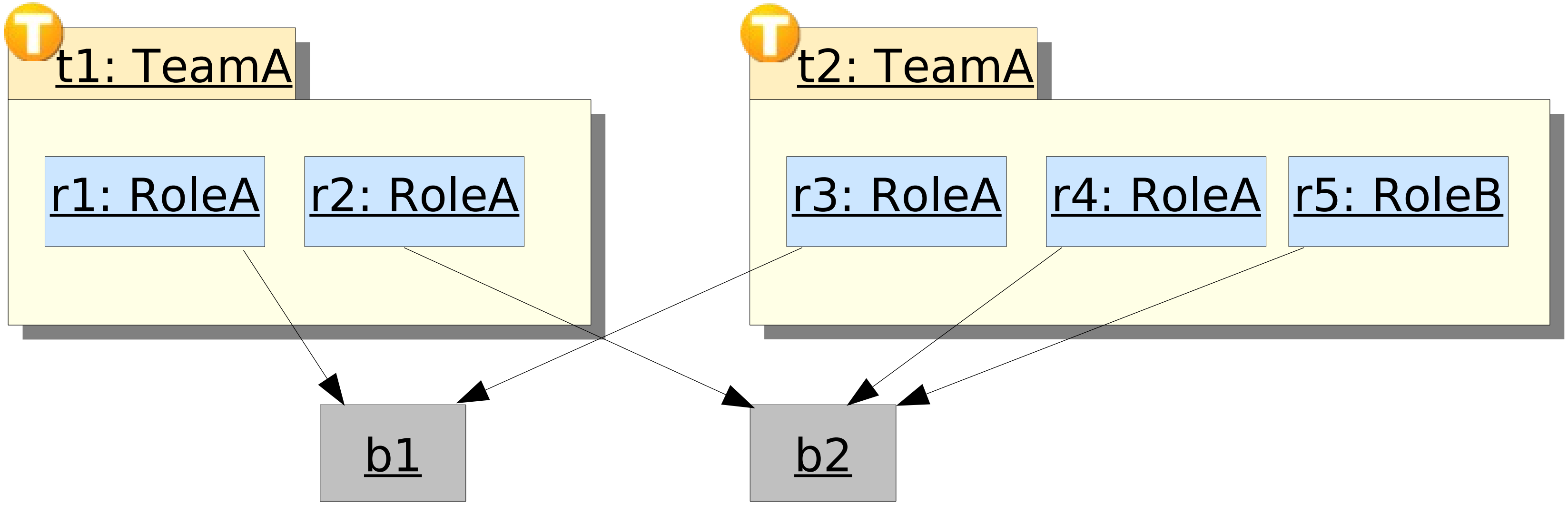
- When a role object **leaves** the team
  - it is **lowered** to its base
- When a base object **enters** a team
  - it can be **lifted** to a role



- 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, $\underbrace{\text{t2, RoleA)}}_{\textbf{RoleA<@t2>}}$ → r4 lift(b2, t2, RoleB) → r5

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?

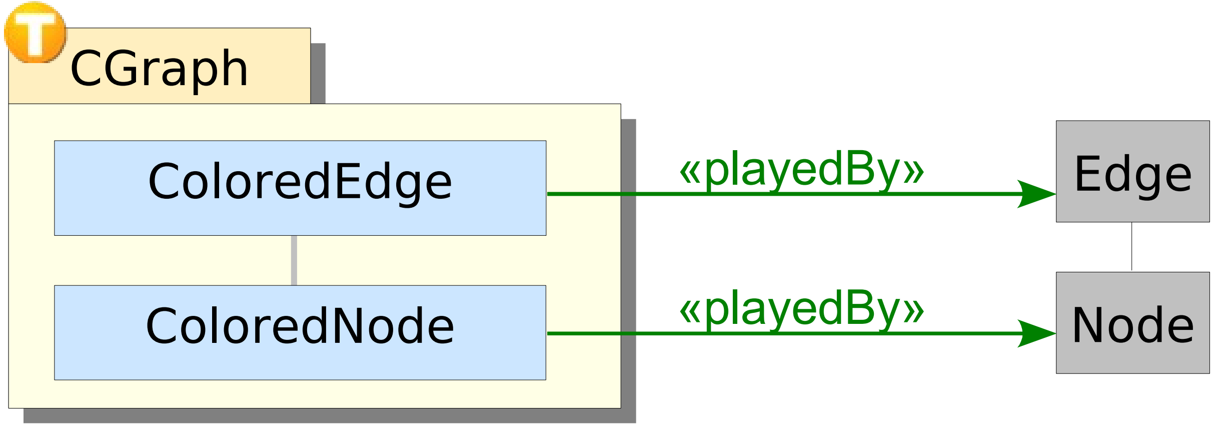
- All data flows entering the team
  - the callin call target ✔

```
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 ma pping
- Eat the cake ar
  - flexibility of multip nstances
  - no disadvantage of “*object schizophrenia*”
  - instances are “almost the same”

Pending: Optimizations
(compiler / runtime)

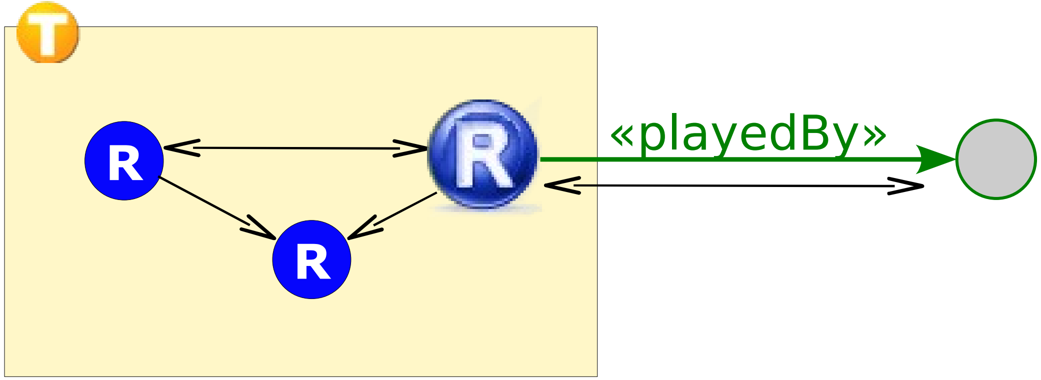
Inheritance vs. PlayedBy in OT/J

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

Properties of Roles (2/4)

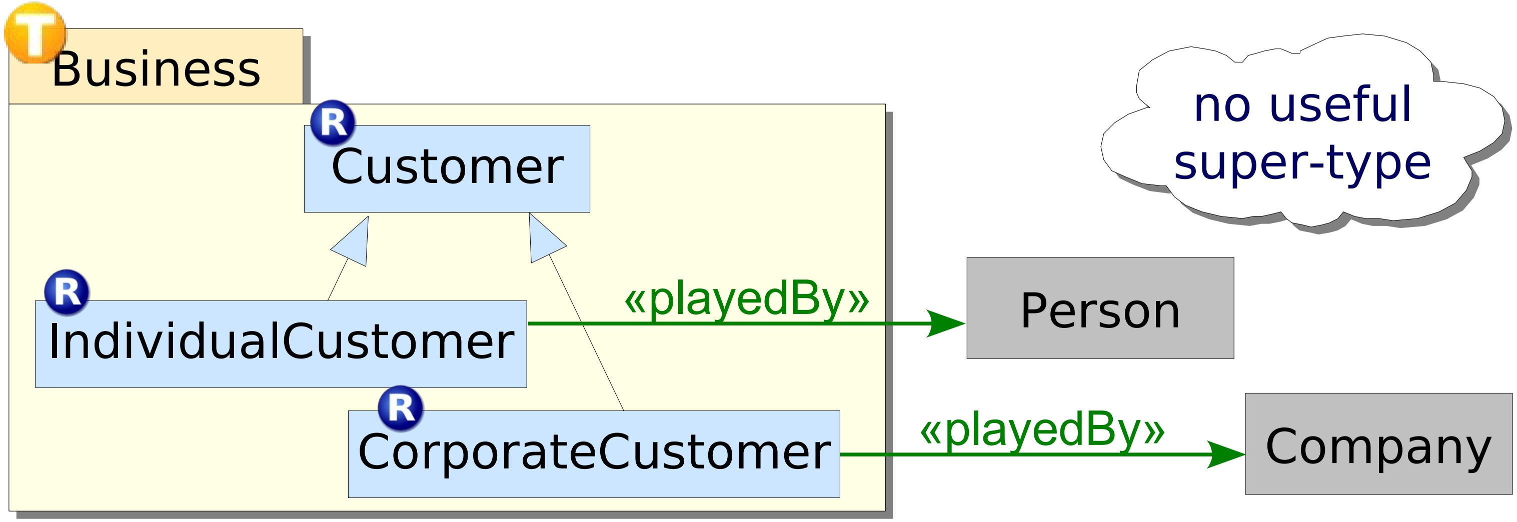
- 15 Criteria by Friedrich Steimann
  - and their mapping to Object Teams

- **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



And now for a Message from our Sponsor ...

Fact Sheet

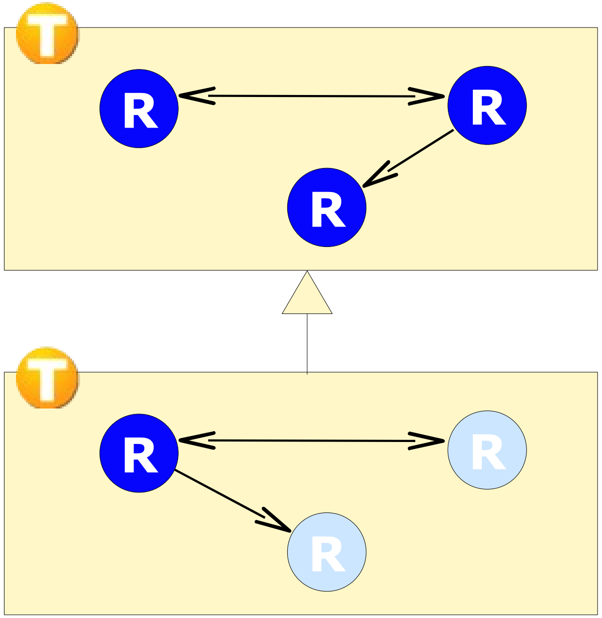
- ObjectTeams/Java (OT/J) since 2001
  - Java += roles, teams, bindings
  - OTJLD 1.0 *(current 1.2)* July 2007
- Object Teams Development Tooling since 2003
  - Java Compiler += OT/J constructs
  - JDT for OT/J (code assist, ui, launch ...)
- Other
  - OT/Equinox: Equinox += aspect bindings since 2006
  - Application
    - Case studies (project TOPPrax)
    - Class room
    - OTDT
    - UML2 tools (base on EMF/GEF/GMF) 2009

Incremental vs. Full Adoption

- Adaptation
  - Given an existing application
    - could be 3rd party
  - Any change task *can* be implemented as a team
    - new feature
    - changing an existing feature
    - even bug fixes (if you like)
- Initial design
  - Fully develop using Object Teams
  - Leverage additional dimension of separation

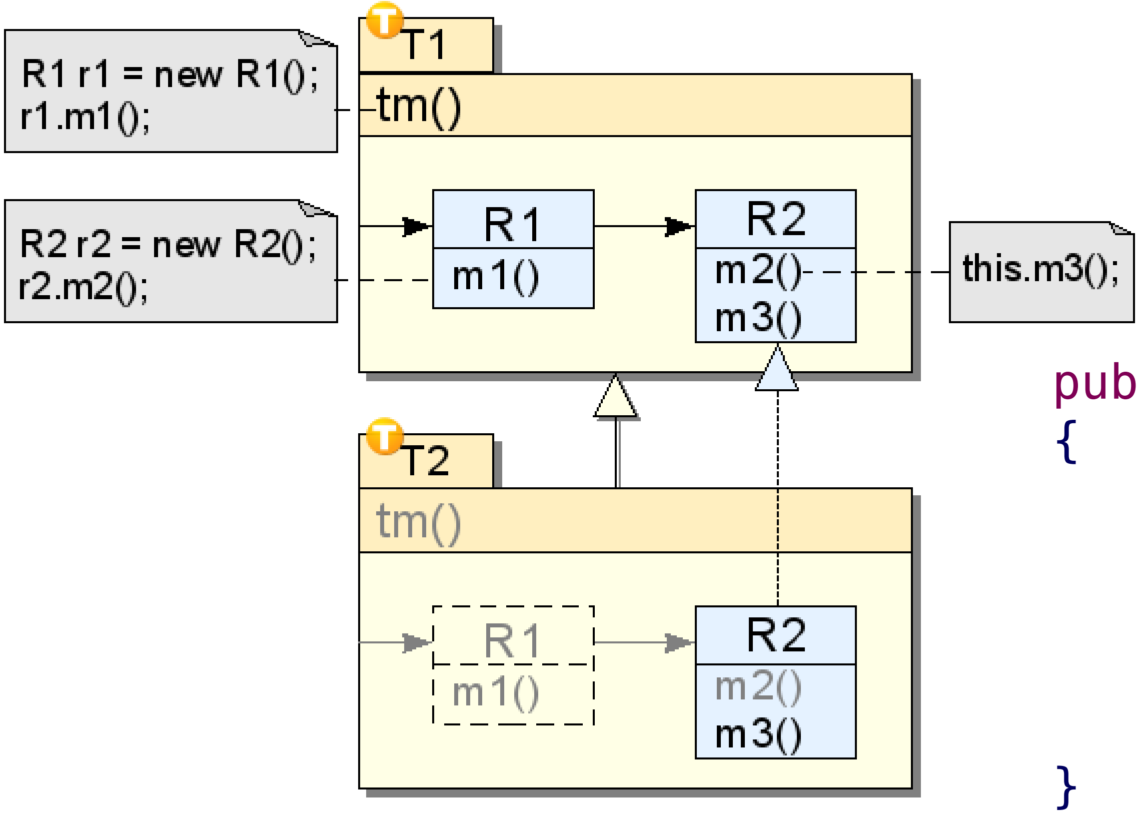
Applying Inheritance to Containment:

Team Inheritance



Team Inheritance

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



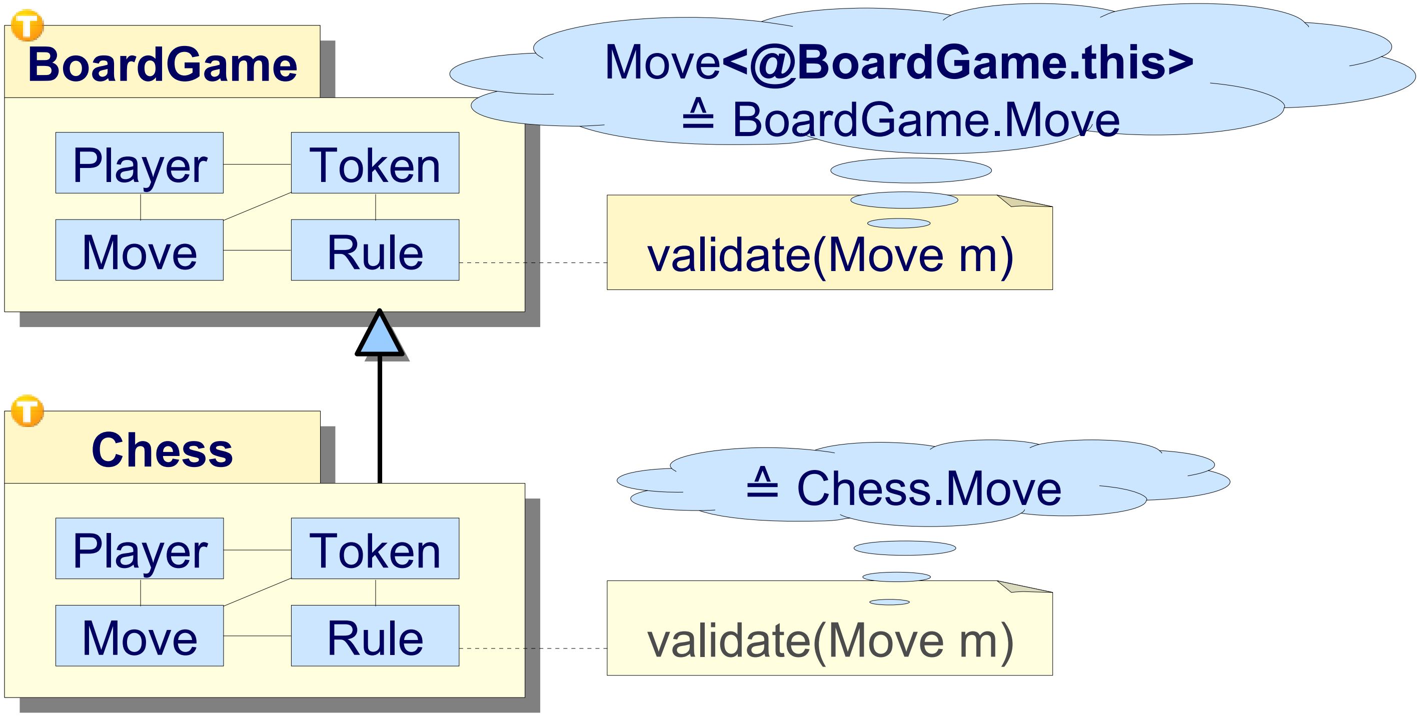
R1 r1 = new R1();
r1.m1();

R2 r2 = new R2();
r2.m2();

this.m3();

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

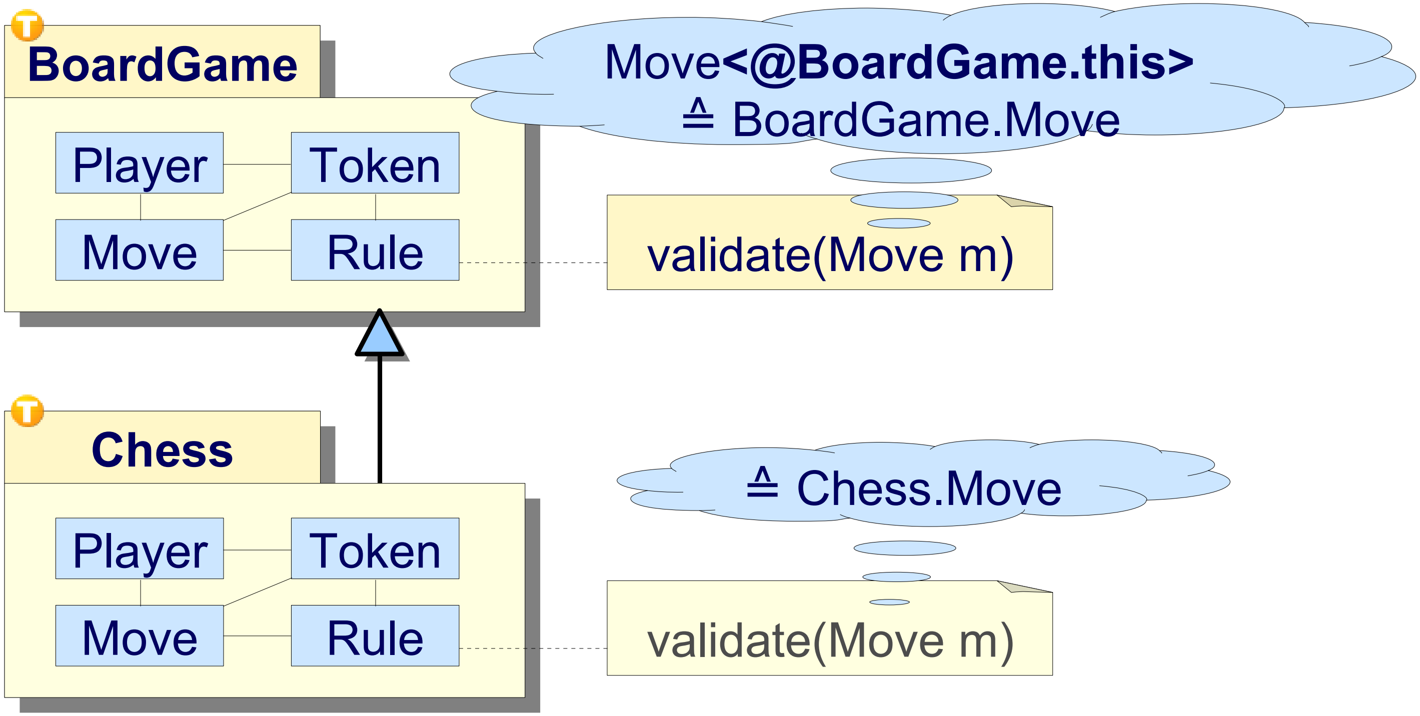
Consistent Polymorphism



- Virtual classes
  - Type safe covariance with dependent types
  - Family Polymorphism™
- Exception: role migration (to other team)

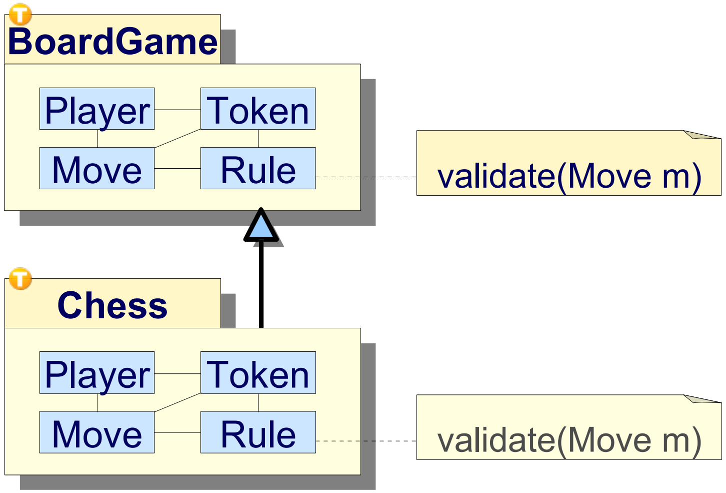
Team Inheritance

Consistent Polymorphism



- Consistently specialize a set of role classes
- No danger of mixing roles from different teams
- Scalable Template&Hook

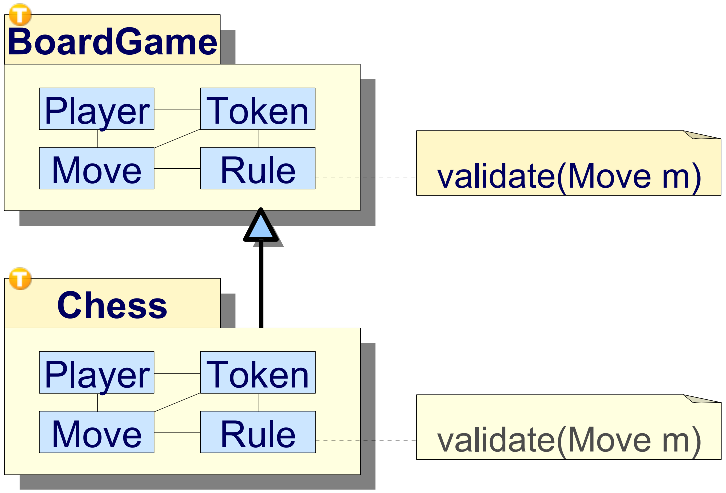
Consistent Polymorphism



- Q: which type “Move” is used?
- A: the one valid in the current context
  - `Chess.Rule → Chess.Move, TicTacToe.Rule → TicTacToe.Rule`
- Q: what if I don't have a context?
  - I hold a reference to a `Rule`, not knowing what game
- A: you have to know what game – instance!

```
final BoardGame myGame = …
Rule<@myGame> rule = myGame.getSomeRule();
Move<@myGame> move = myGame.getRandomMove();
rule.validate(move);
```

Consistent Polymorphism



- A: you have to know what game – instance!

```
    final BoardGame myGame = …
    Rule<@myGame> rule = myGame.getSomeRule();
    Move<@myGame> move = myGame.getRandomMove();
    rule.validate(move);
```

- Q: haven't I lost polymorphism, now?
- A: no, myGame is still polymorphic
  - type Move is dynamically bound relative to myGame.
  - and everything is rock solidly type-safe

Class level Template&Hook

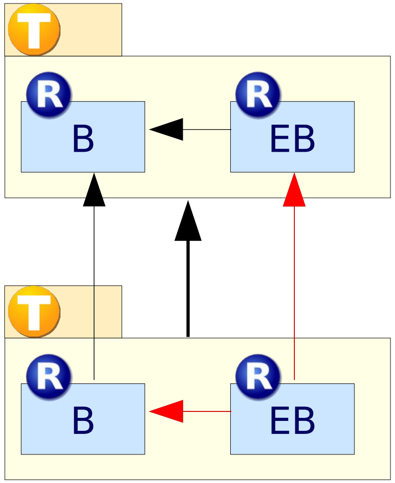
```
abstract team class BoardGame
{
    abstract class Player {...}
    Player a;
    void init() {
        a = this.new Player();
    }
}
team class Chess extends BoardGame
{
   class Player {...}
}
```

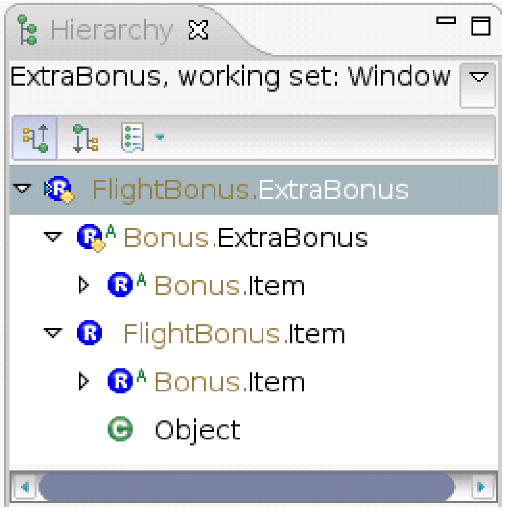
```
abstract class C
{
    abstract void hook();

    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

Team Inheritance

- Roles are virtual classes
  - can be overridden in sub-teams
  - overiding role implicitly inherits from overridden role
  - → mild form of multiple inheritance





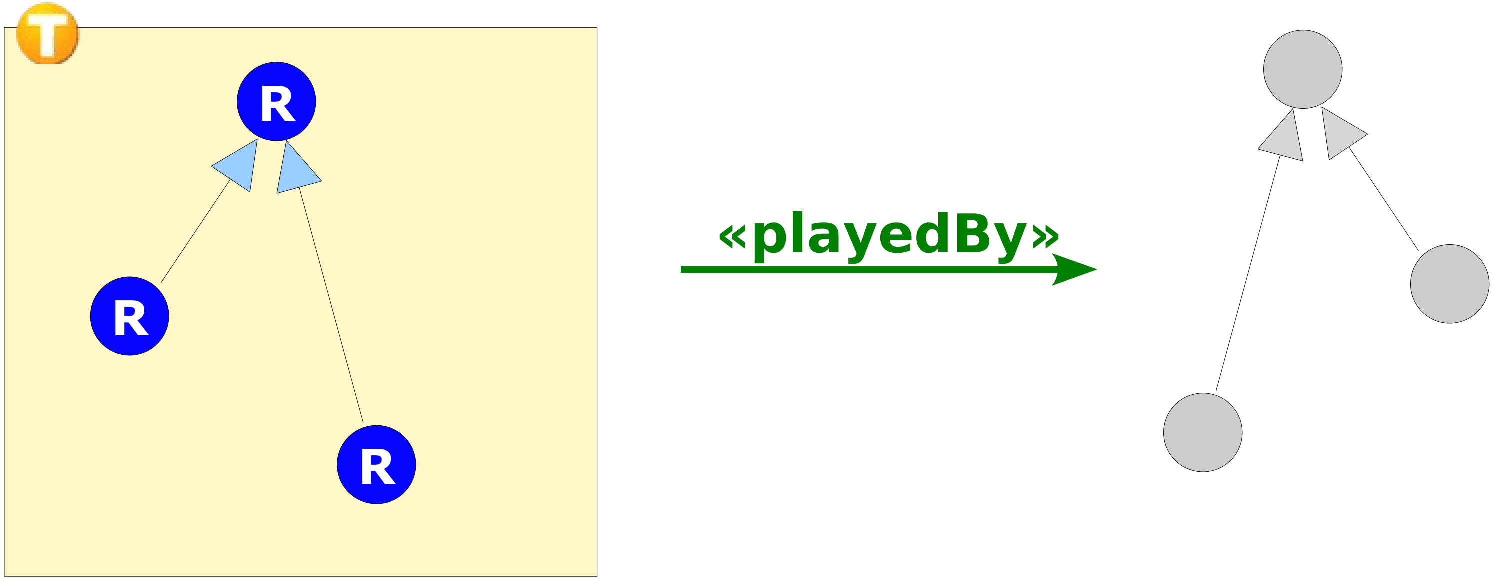
ExtraBonus, working set: Window

```
FlightBonus.ExtraBonus
  Bonus.ExtraBonus
    Bonus.Item
  FlightBonus.Item
    Bonus.Item
    Object
```

  - → two kinds of super-call:
    - `super();` (constructor) – `super.m();` (method)
    - `tsuper();` (constructor) – `tsuper.m();` (method)

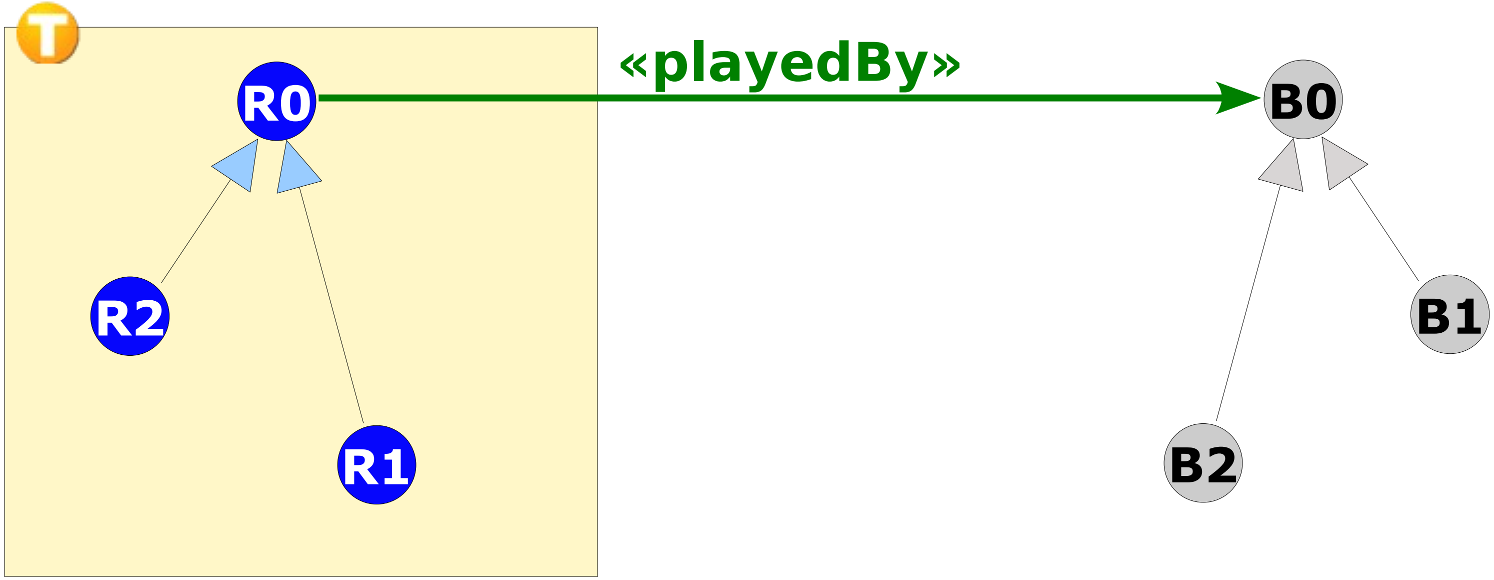
Applying Translation Polymorphism to Inheritance Structures

“Smart Lifting”



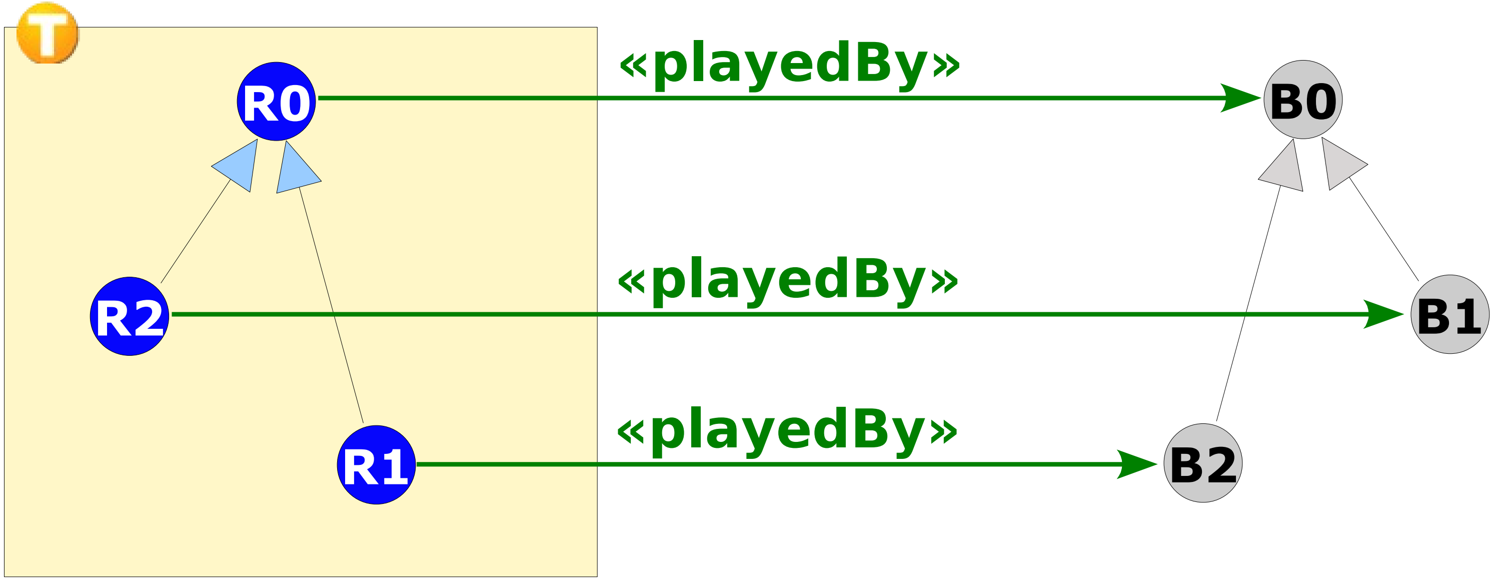
Most Specific Type

- 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



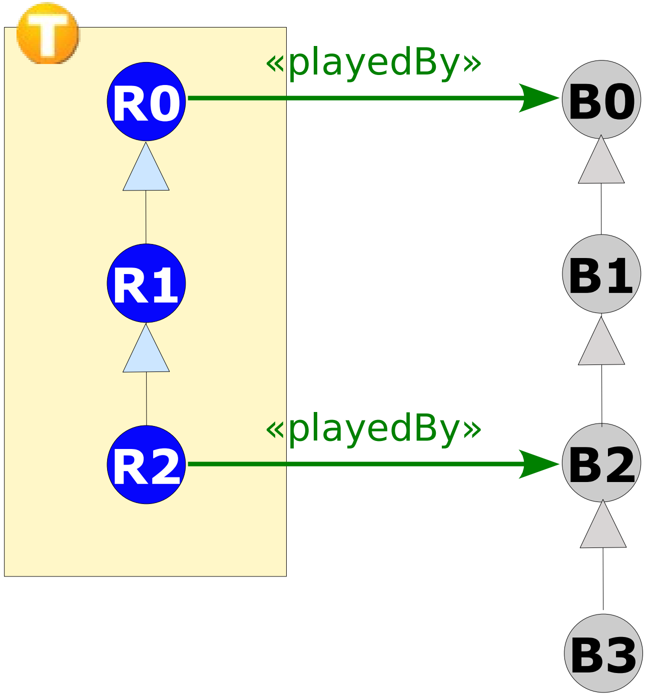
Lifting with Constraints

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



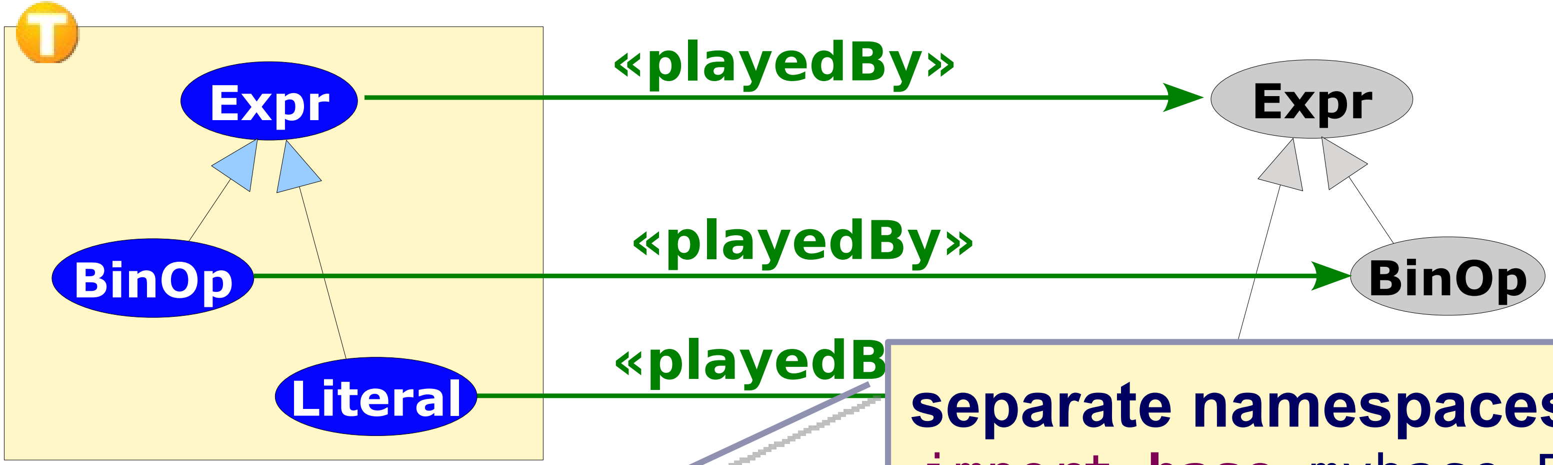
Lifting with Constraints

- 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 (never instantiated)
  - skip B1 (subsumed by R0)



Double Dispatch

- Adding instance dispatch to method dispatch



separate namespaces:
`import base mybase.Expr;`

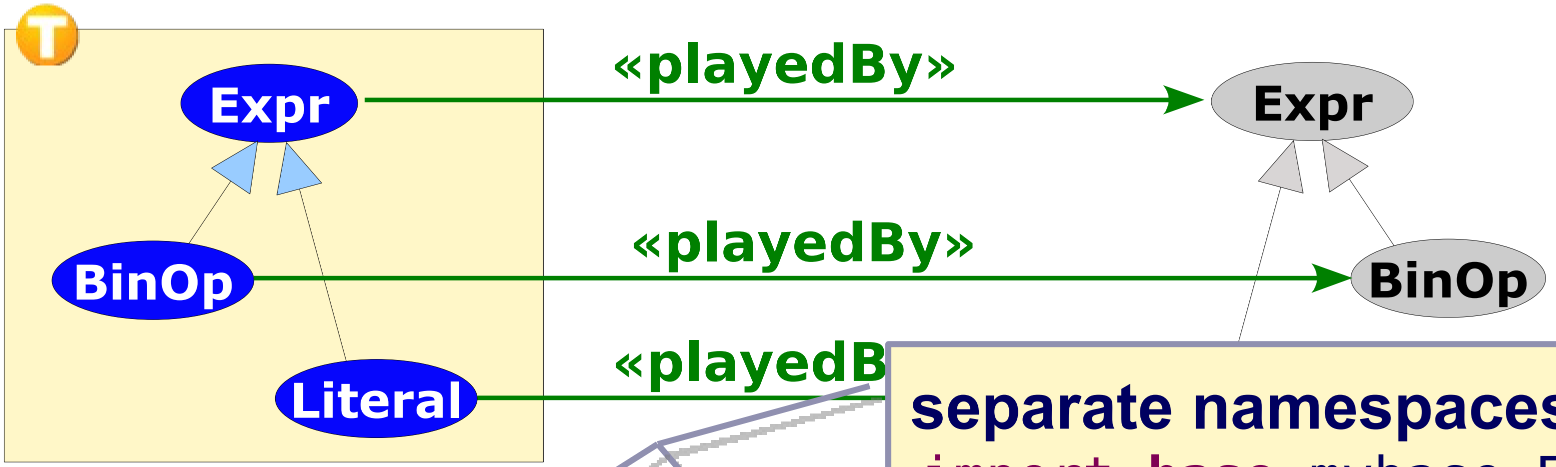
```
team class PrettyPrinter extends ExprVisitor {
   abstract class Expr playedBy Expr
      abstract accept(); }
   // other role classes
   void visit(Expr as Expr node) {
      node.accept(); }
}
void print(Expr node) {
   ExprVisitor visitor = new PrettyPrinter();
   visitor.visit(node); }
```

dispatch on visit:
- **selects the Function**

dispatch on node:
- **selects the Node-Type**

Double Dispatch

- Adding instance dispatch to method dispatch



separate namespaces:
`import base mybase.Expr;`

```
team class PrettyPrinter extends ExprVisitor {
   abstract class Expr playedBy Expr {
      abstract void accept(); }
   class BinOp playedBy BinOp {
      Expr getLeft() -> Expr getLeft();
      void accept() {
         getLeft().accept(); /* ... */ }
   void visit(Expr as Expr node) { node.accept(); }
}
void print(Expr node) {
   new PrettyPrinter().visit(node); }
```

Smart Lifting selects:

Team:	*Function*
Role:	*Node Type*

Applying Generics to Role Playing

Generic Callin Bindings

Problem:

- replace callin binding requires 2-way compatibility

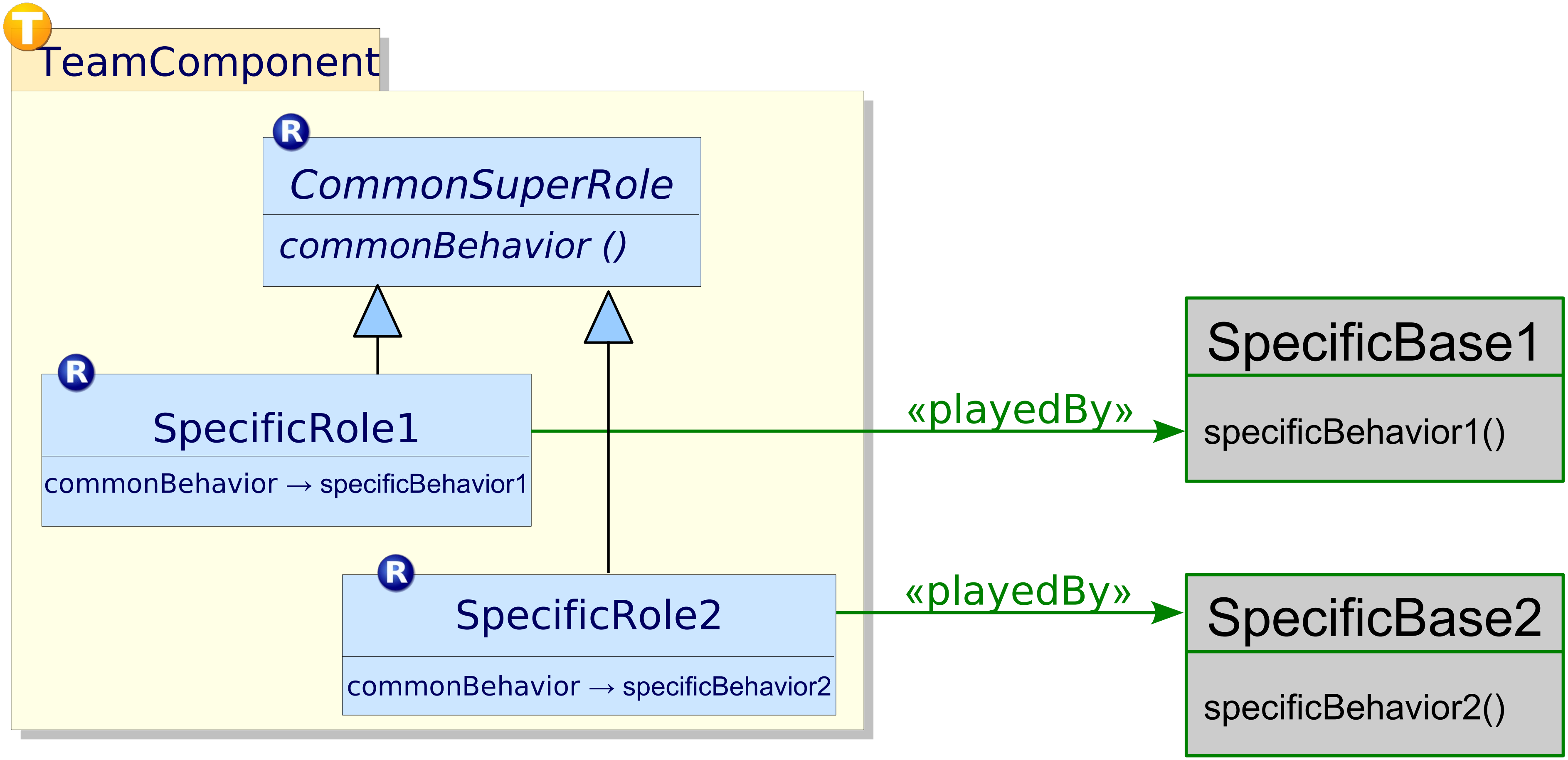
```
callin T1 roleMethod() {
      T1 oldResult = base.roleMethod();
      return new T1();
}
T1 roleMethod() <- replace T1 baseMethod();
```

- 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

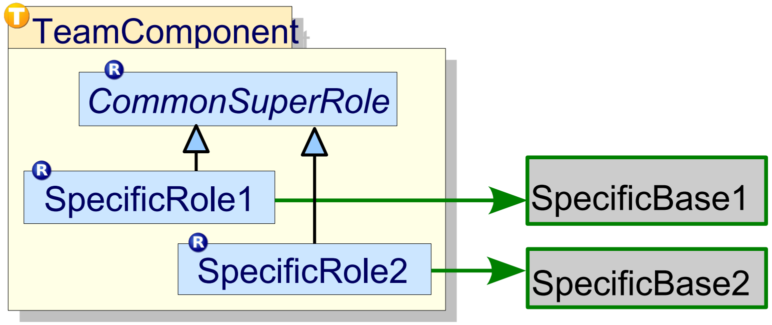
```
callin <E extends T1> E roleMethod() {
      return base.roleMethod(); // OK, new T1() NOK
}
<E extends T1> E roleMethod() <- replace T1+ baseMethod();
```

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();
}
```

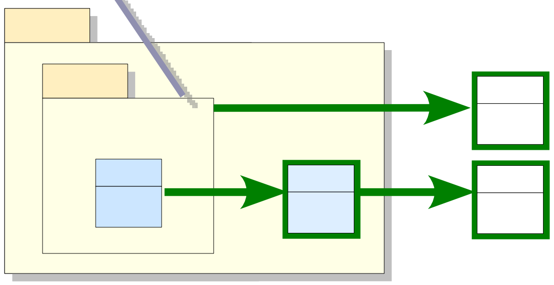
  - B is the union of all classes that can be lifted to `CommonSuperRole`

Composed Structures

Applying Object Teams Concepts to each other

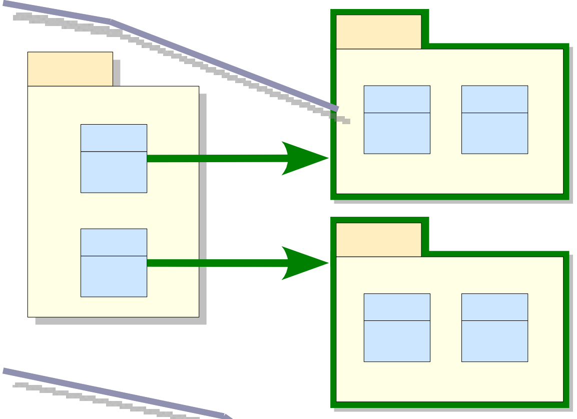
Nesting – Stacking – Layering

Team plays the role **Role**



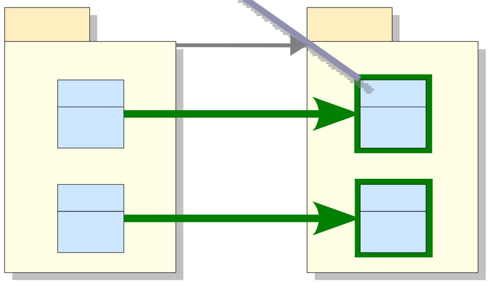
- Nesting
  - Team can contain teams as its roles
  - Nesting applies to instances, too

Team plays the role **Base**



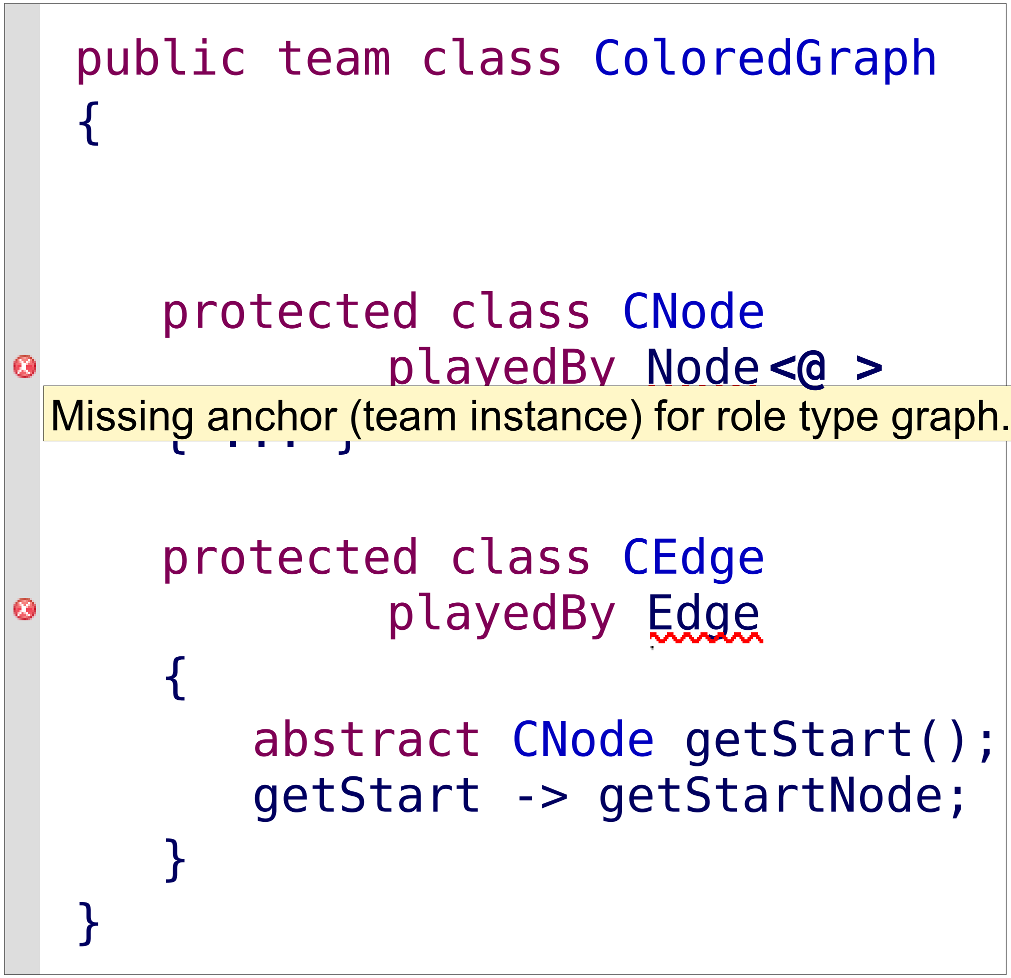
- Stacking
  - Role can adapt another team
  - Multiple roles coordinate multiple teams

Role plays the role **Base**



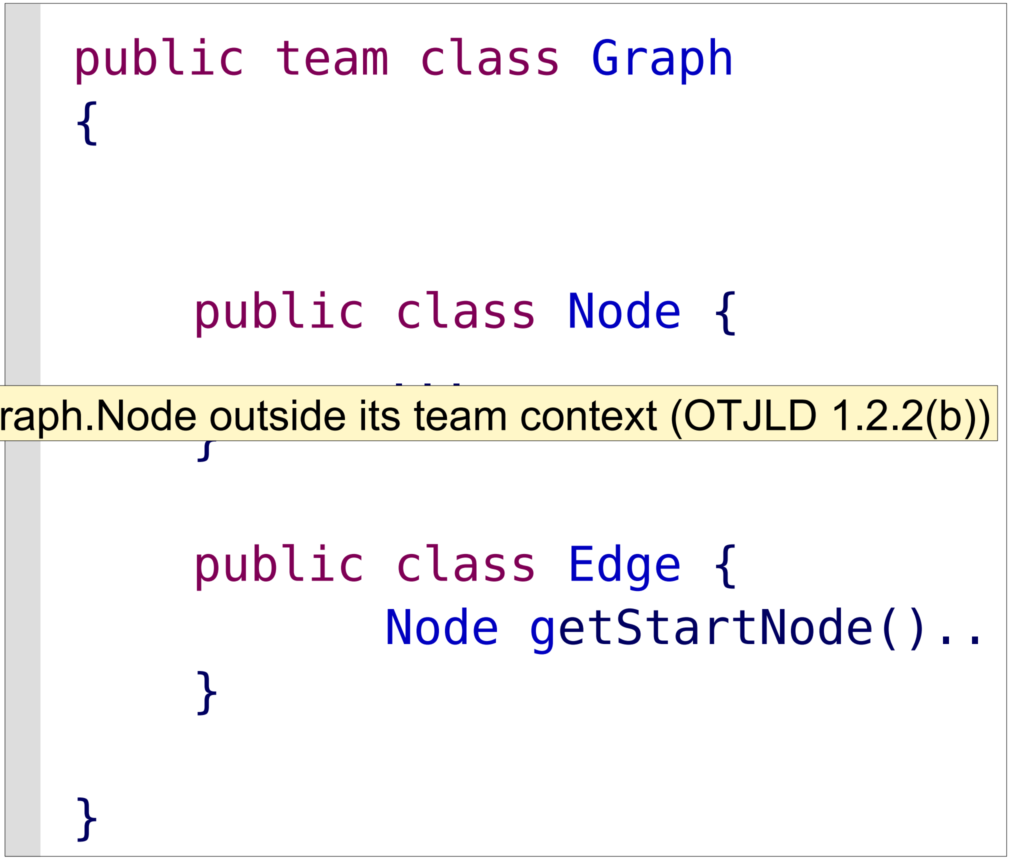
- Layering
  - Roles adapt roles of another team
  - Define a view of an existing team

Layering – Detail

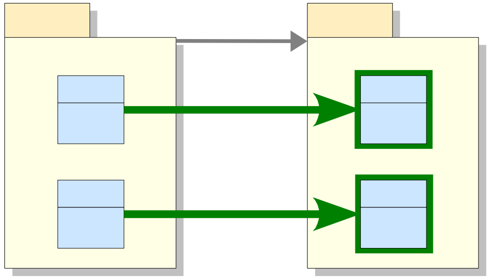


```
public team class ColoredGraph
{


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

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

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



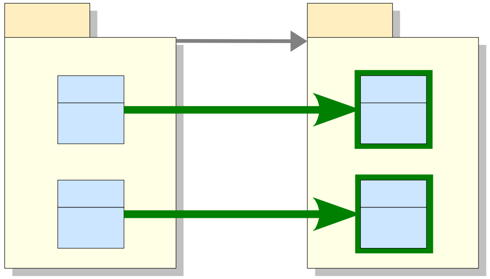
```
public team class Graph
{


        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)

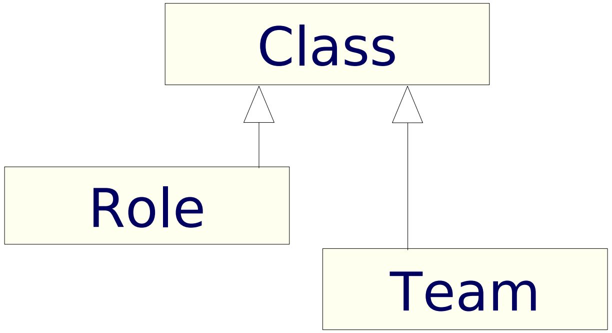
- 15 Criteria by Friedrich Steimann
  - and their mapping to Object Teams

- **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);
}
```

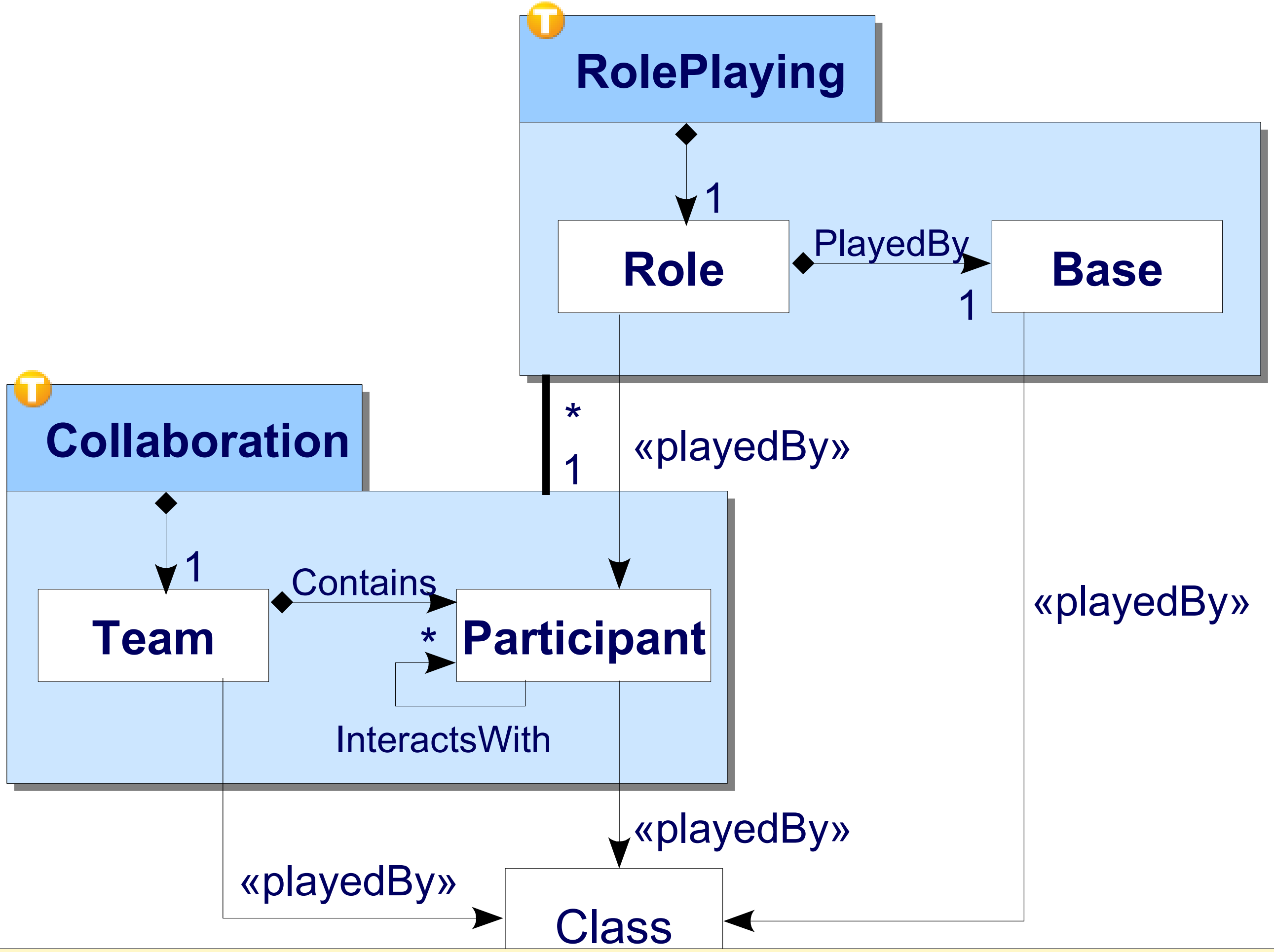
A Meta Model for Object Teams

Meta Model for OT/J?



- Combinations?
  - Role & Team: nested Team
  - Role & Base: layered Team, Role-of-Role
  - Team & Base: stacked Teams
- Model evolution?
  - group of classes ↣ collaboration of roles

Meta Model for OT/J !



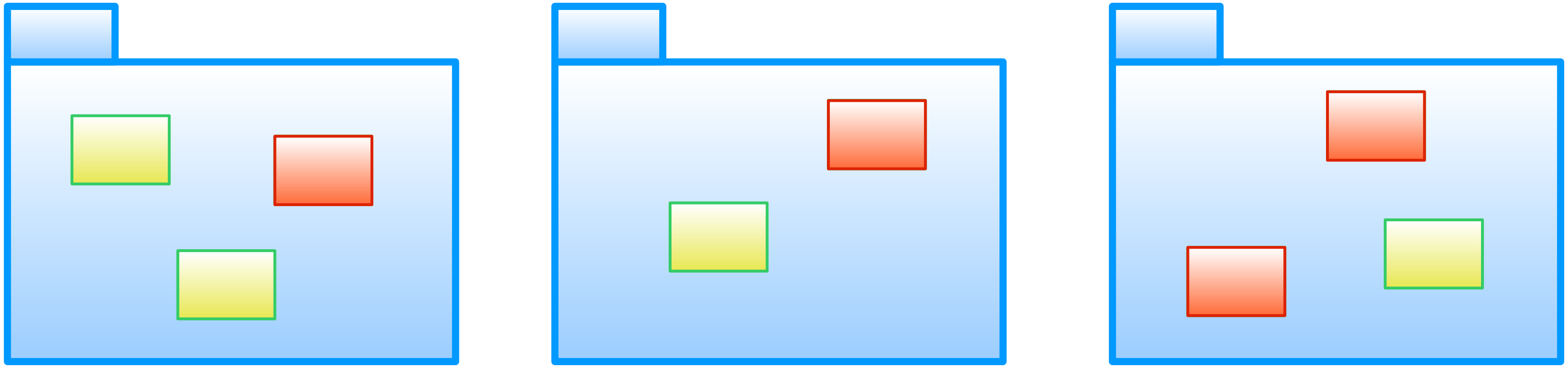
In order to explain Roles
you first have to explain Roles.

Big Picture

Architecture
with
Object Teams

Classes – Packages - Components

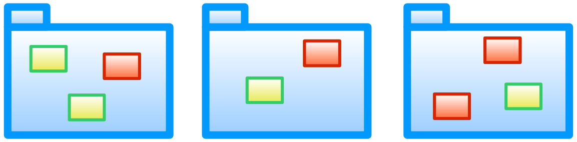
- Traditional decomposition



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

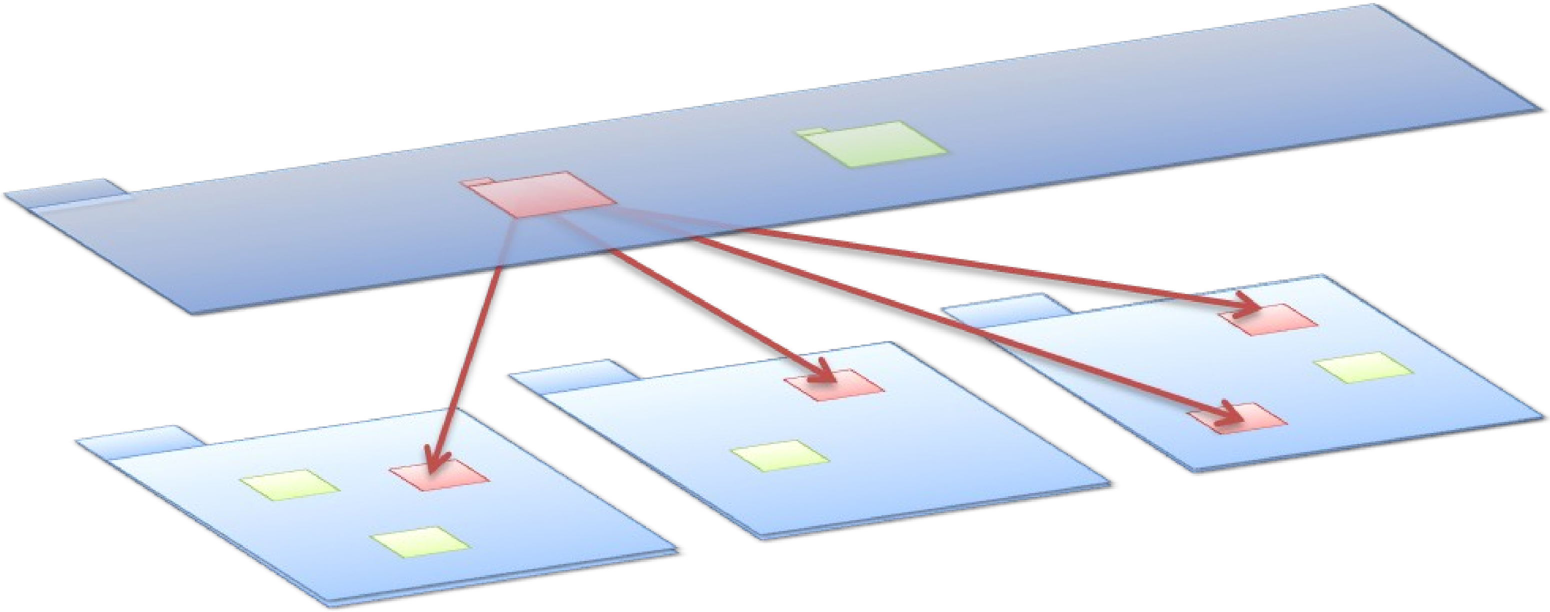
Be Inventive!

- We need to zoom out!



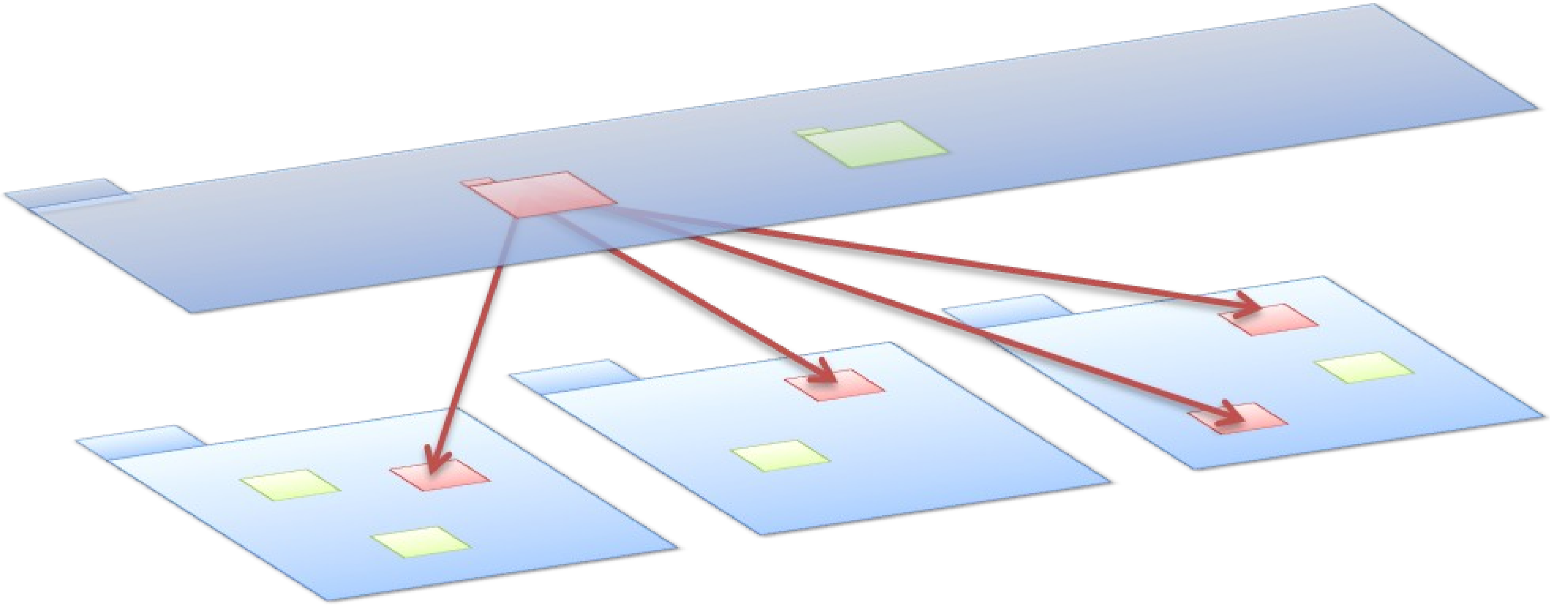
  - See a new solution?
  - No.
  - Try again!

Entering The Next Level



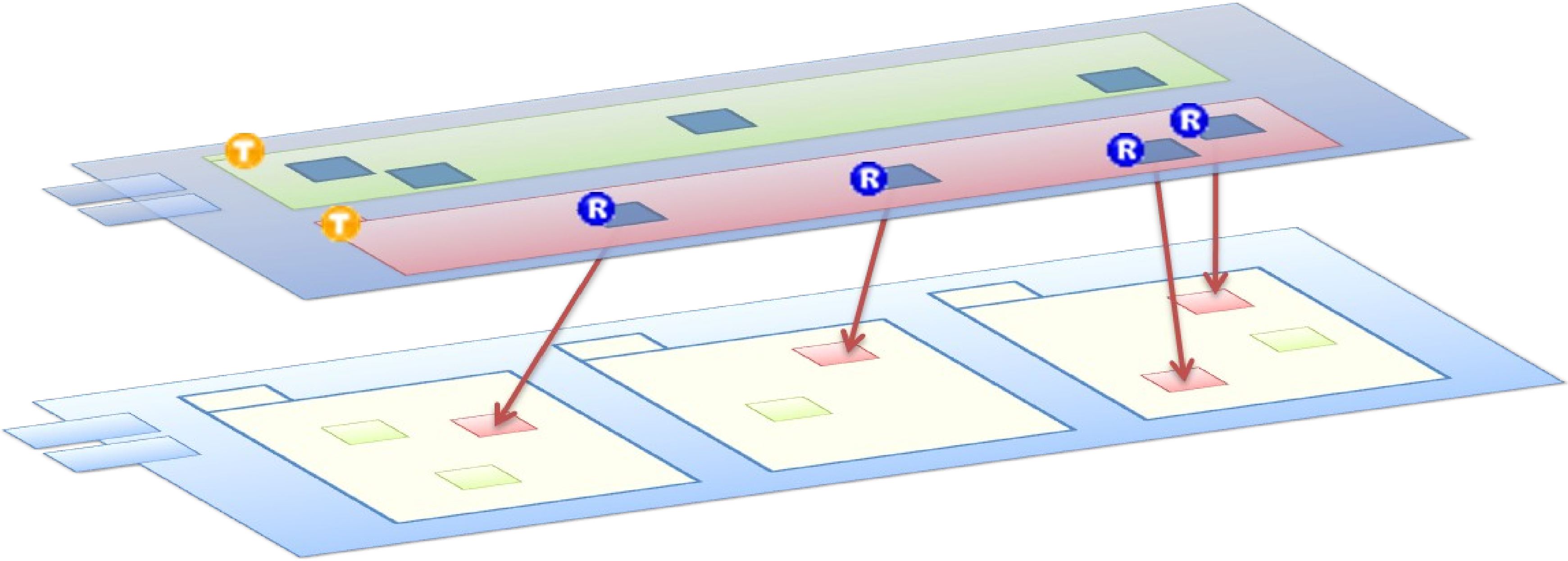
- Can you see it now?
- Now?
- Now!

Layered Design



- In truely layered designs
  - each layer may have its very own structure
  - layers are connected to each other by a mapping
  - 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

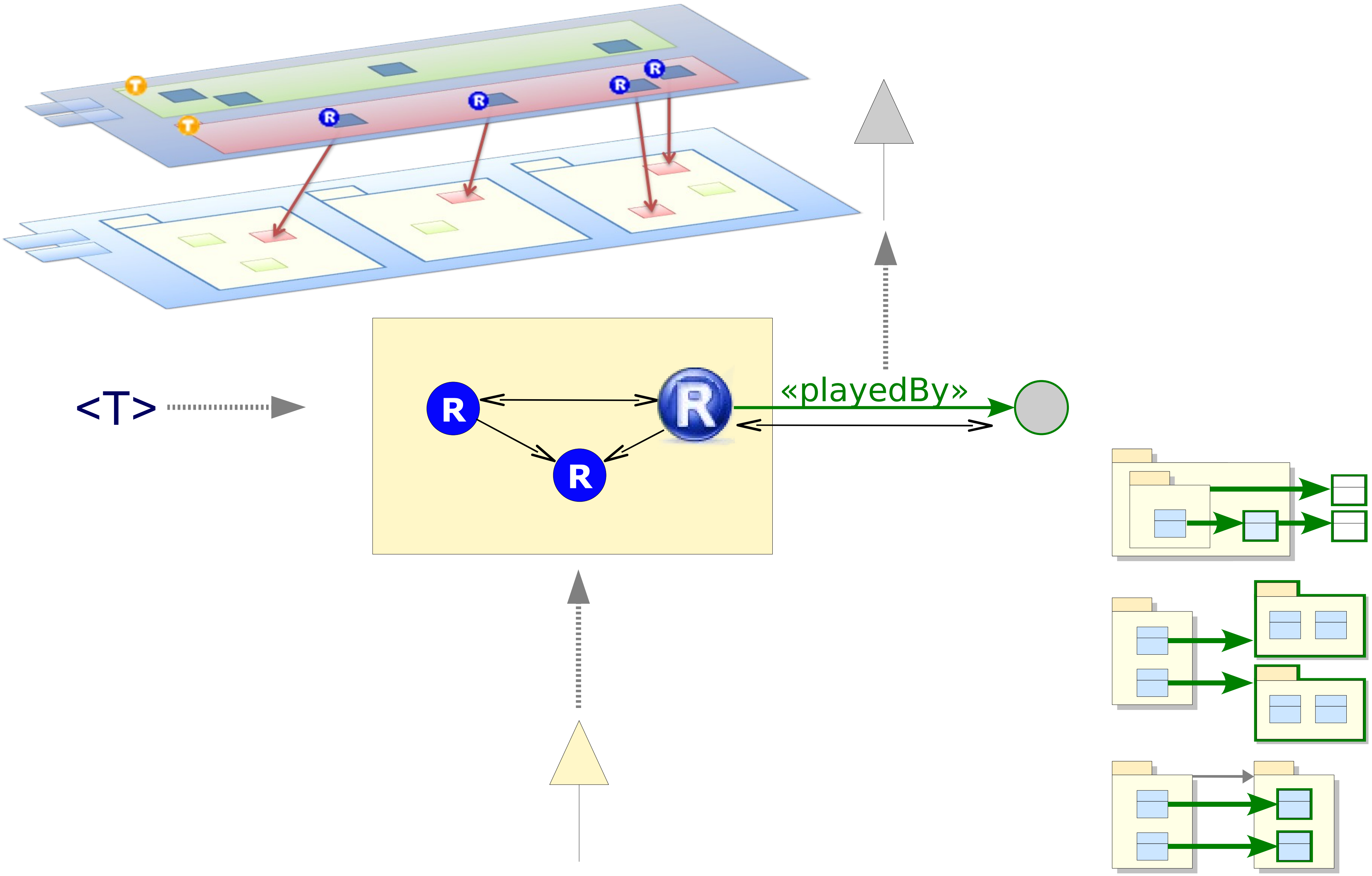
- Role defines view on base class
- Team encapsulates a set of roles

Summary

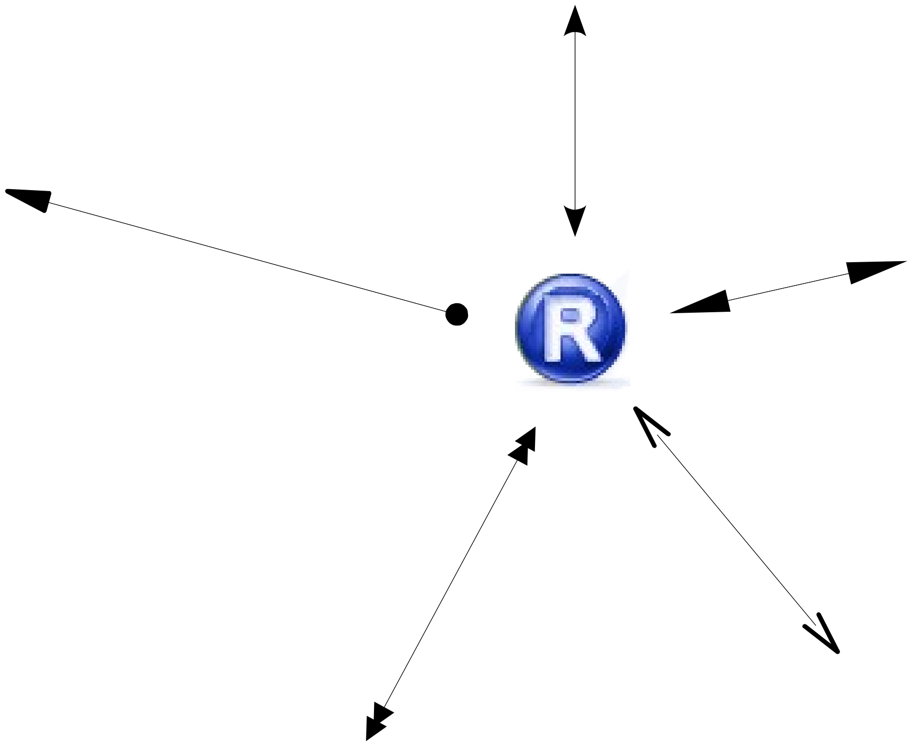
Conclusion

Epilogue

Summary



Summary



Objectivity ↔ Subjectivity

- Objectivity
  - Objects are exhaustively defined in one place
  - Definition must consider all special cases
- Subjectivity
  - Consider only relevant properties

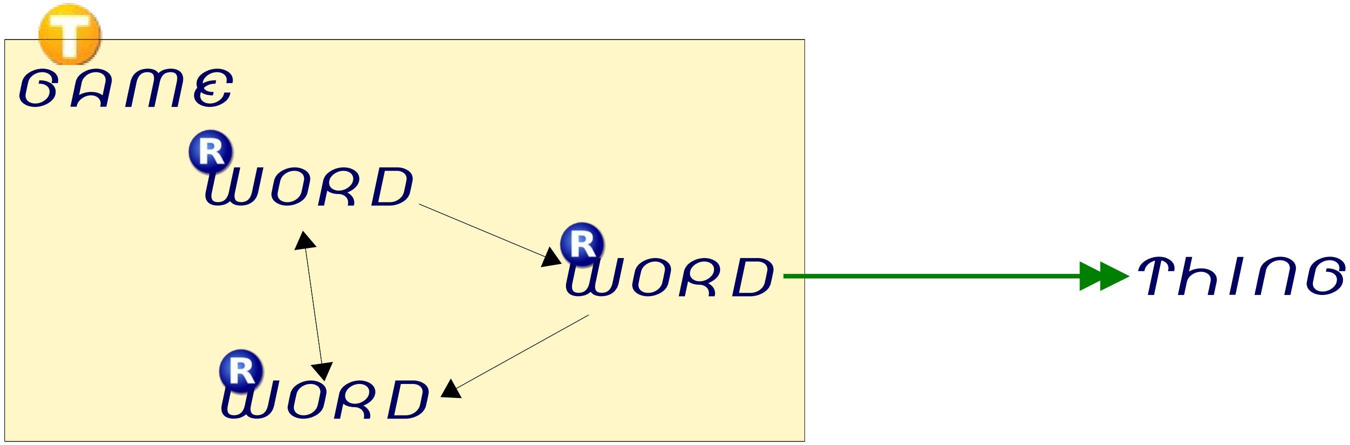
» Roles Rule!

- **Object Teams makes Roles Real**

  - Express how your perspective relates to „the world"
- Subjectivity in Software Engineering
  - Perspectives during RE
  - Views / diagrams during design
  - In programming: Roles!

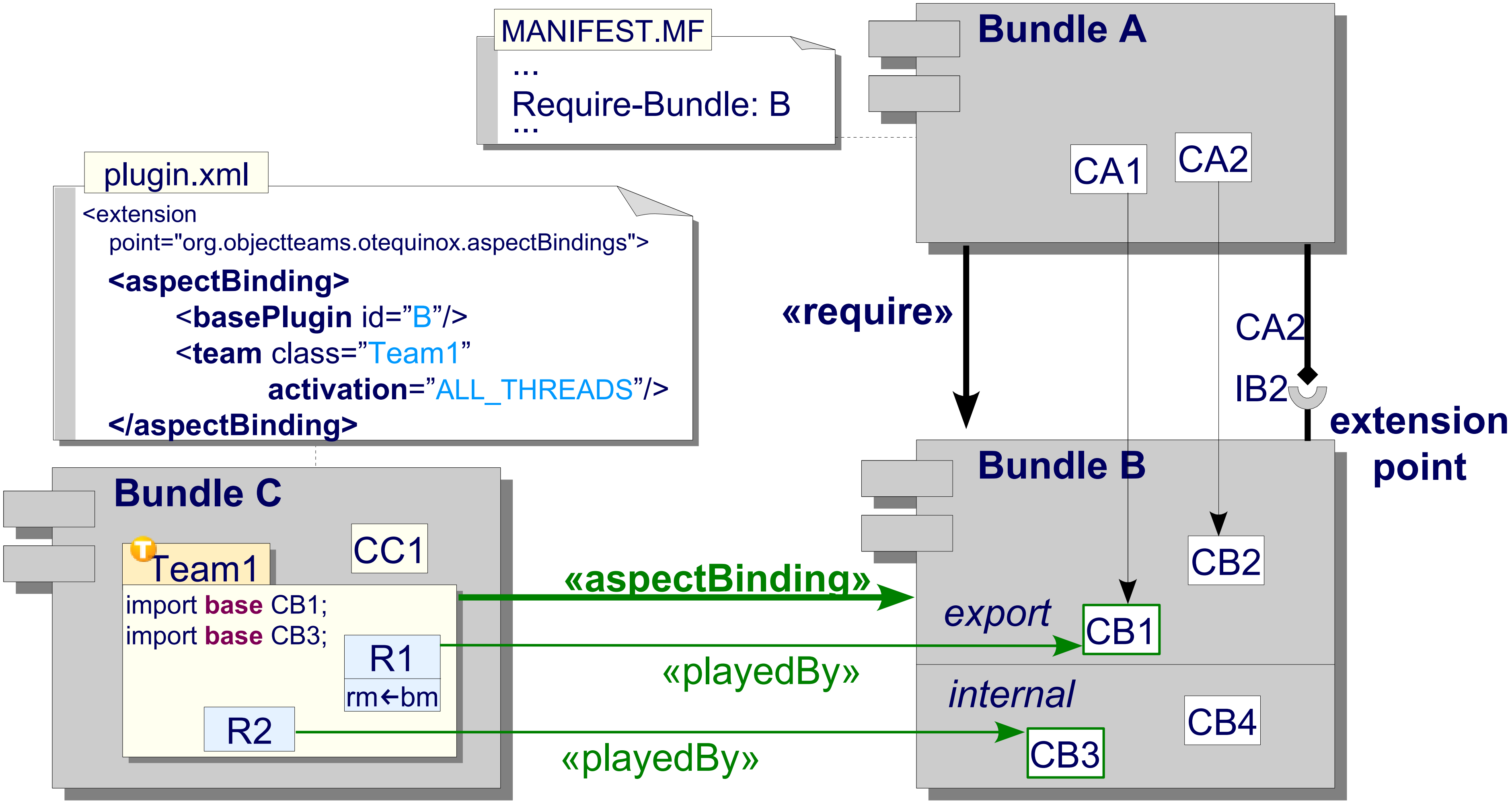
Epilogue

- 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 **game** of words



Mechanisms in more Detail

Components: OT/Equinox



MANIFEST.MF

```
...
Require-Bundle: B
...
```

plugin.xml

```
<extension
   point="org.objectteams.otequinox.aspectBindings">
   <aspectBinding>
         <basePlugin id=”B”/>
         <team class=”Team1”
               activation=”ALL_THREADS”/>
   </aspectBinding>
```

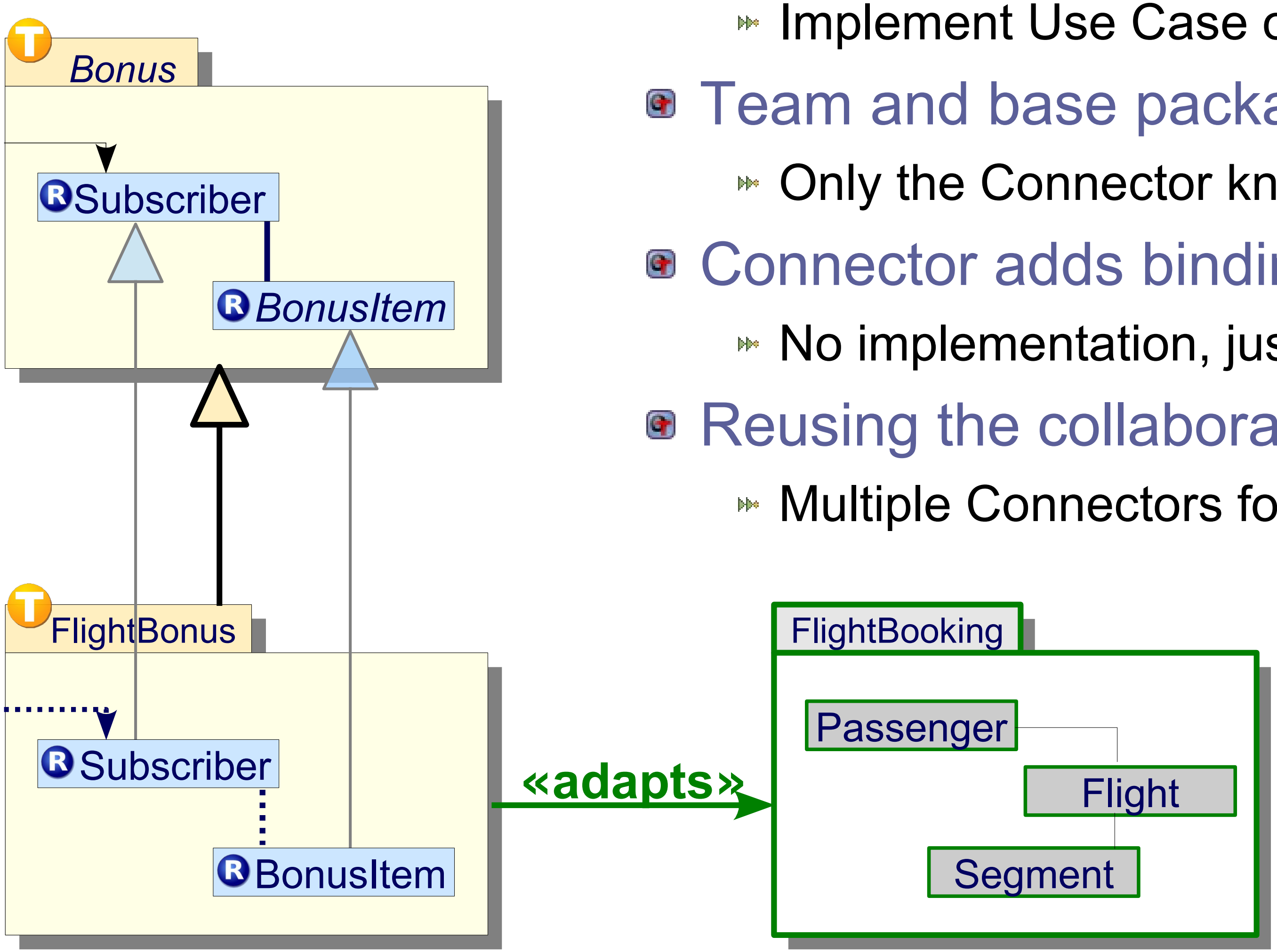
Team1

```
import base CB1;
import base CB3;
```

Patterns

Connector Pattern

- Abstract team provides implementation
  - Implement Use Case only in terms of roles
- Team and base package are independent
  - Only the Connector knows both
- Connector adds bindings to base package
  - No implementation, just integration
- Reusing the collaboration
  - Multiple Connectors for multiple base packages



Virtual Restructuring

Define this view

```
team class PrintFlight {
   class FlightRole playedBy Flight {
      int getSegmentCount()            → get List<Segment> segments
         with { result ← segments.size() }
      SegmentRole segmentAt(int i)     → get List<Segment> segments
         with { result ← segments.elementAt(i) }
   }
}
```

Parameter Lifting

- base class
- new interface as a view

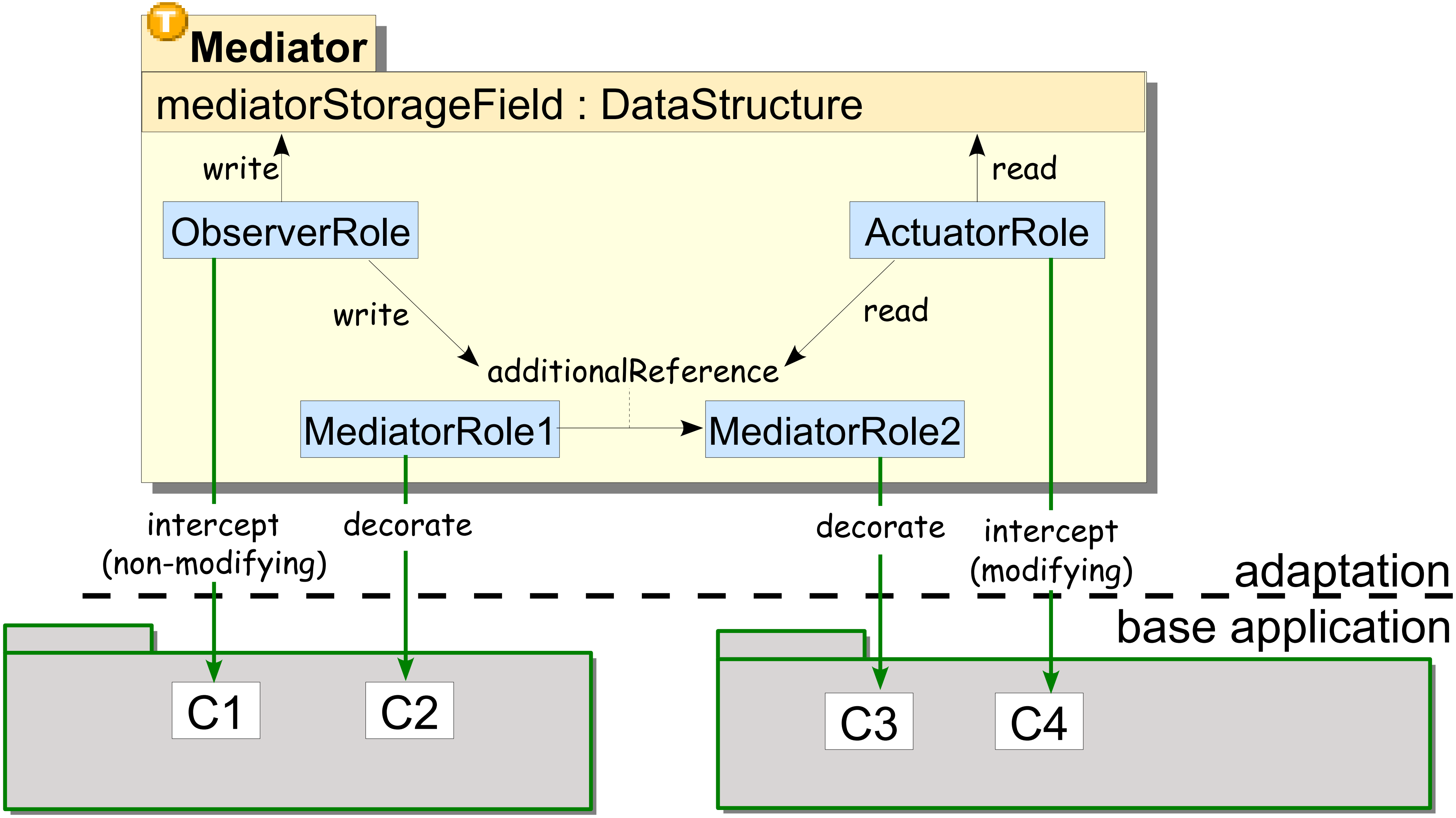
Given this data class

```
class Flight {
   private List<Segment> Segments;
}
```

Traditionally

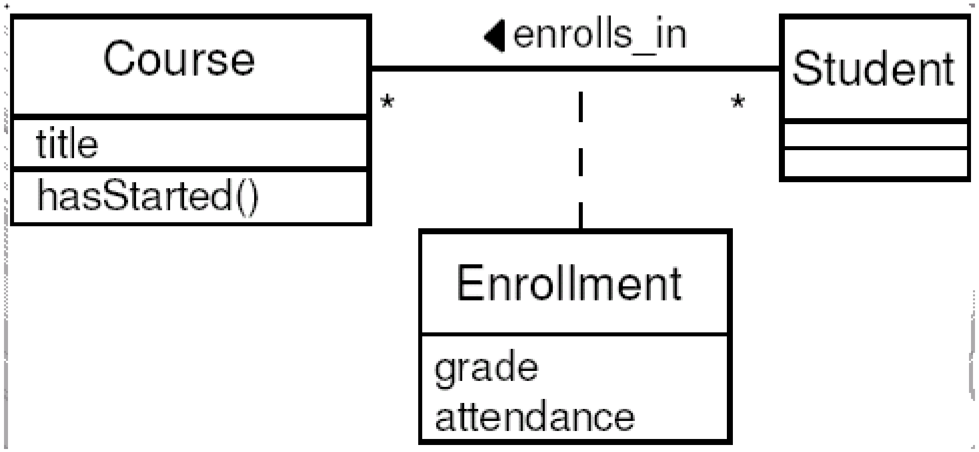
- apply refactoring: Encapsulate Collection

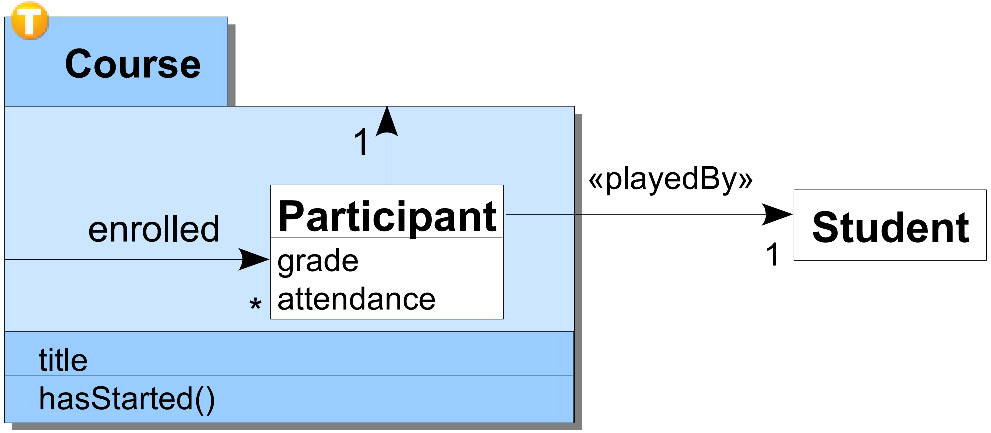
Observer-Mediator-Actuator

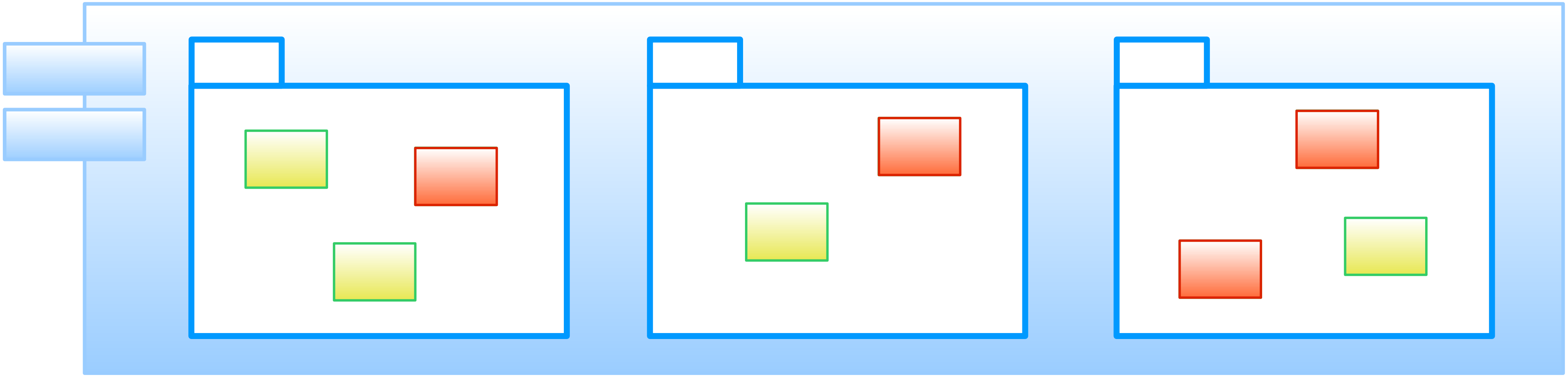


Relationships

- Implementing a stateful relationship







Topics

- Role object life cycle
  - lifting, instance management, multiplicities
- Team inheritance
  - specializing whole frameworks w/ propagation
- Patterns
  - Connector: separating implementation ↔ binding
  - Base class generalization: post-hoc super type
  - Virtual restructuring: changing structure not code
  - ...
- Architectures
  - Observer-Mediator-Actuator, Stacking, Nesting, Layering
- Component technology
  - OT/Equinox: architecture level aspect bindings