Object Teams: Programming with Contextual Roles

Dr. Stephan Herrmann
Independent

www.objectteams.org
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
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“
Roles
- connect intuition to technology
- emphasize objects over classes
- introduce subjectivity into programming
- are broadly explored in research

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 approximation of ObjectTeams/Java
- Java + Delegation
  - role containment : inner classes (instance containment)
  - playedBy : delegation (overriding, late binding of self)
15 Criteria by Friedrich Steimann

- 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
  - Java: private, protected, public

**Making the type system instance-aware**
- dependent types: annotate types with instances
  - Role<\(@\rangle\rangle$: type parameter is an instance
  - Role<\(@\rangle\rangle\rangle\rangle\): unless $c_1 \neq c_2$

Fundamentally install instance based access control
- Every role type is a dependent type
  - CoffeeMachine<@Department\.this> OK
  - CoffeeMachine<@yourDepartment> illegal

Hide complexity for the default case
- Within its team the type anchor can be omitted
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

- 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 always 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
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

«playedBy»
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?

What's 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
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`

```
SuperClass
  method()

SubClass
  method()
```

**Role Playing**
Inheritance vs. PlayedBy in OT/J

### Inheritance
- Import
  - `dispatch sub`

```
SuperClass
  method() -> SubClass
  method()
```

### Role Playing
- Callout binding
  - `dispatch role -> base`

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

### No other access to «base»
- encapsulate semantics
- separate two worlds
- specific privilege

```
  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 super

- Overriding
  - dispatch super

Role Playing

- Callout binding
  - dispatch role → base

- Callin binding
  - dispatch role ← base

**Difference:**

- Different names
- Parameter mappings (implicit/explicit)
- Before / replace / after
- Base calls

```
String getPhoneNo() <- replace String getPhoneNo();
```
Inheritance vs. PlayedBy in OT/J

Inheritance

- Import
  - dispatch suffice
- Overriding
  - dispatch suffice

Role Playing

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

which context to select the appropriate role instance?
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**
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 / scoping
  - extends the scope of the role-object
- renaming
  - as part of callout binding (incl. parameter mapping)
- interpretation
  - forwarding role → base (objects)
Import in OT/J:

A callout method binding

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

... can use different names on role / base sides
... can adjust signatures

**Inferred callout**

1. `getName()`

1.1 `getName()`

No other access to «base»

- encapsulate semantics
- separate two worlds
- specific privilege

Inferred callout

- for self calls
- for methods declared in a common interface
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 \(\rightarrow\) sub

Dynamic context selects behavior

- the dynamic type of the current object
Role-Playing (2): Overriding

Role-object override base methods only by declaration of a new method with no exception and renaming as part of calling binding (incl. parameter mapping).

Supporting Evolution

Interception role ← base (objects)
Role-Playing (2): Overriding

- override base methods only by declared callin binding
  - <no exception>
  - renaming
  - as part of callin binding (incl. parameter mapping)
  - interpretation

**BUT** interception role ← base (objects)

- 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
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...
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() \(<-\) \textit{replace} String getPhoneNo();

... declares that calls to the base should be \textbf{intercepted} by its role

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

\textbf{Binding variants}

- \texttt{before}, \texttt{replace} or \texttt{after}
2 Mechanisms, 3 styles of dispatch

- **Forwarding**

- **Interception**

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

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Role Playing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Import</strong></td>
<td><strong>Callout binding</strong></td>
</tr>
<tr>
<td>- dispatch sub → super</td>
<td>- dispatch role → base</td>
</tr>
<tr>
<td><strong>Overriding</strong></td>
<td><strong>Callin binding</strong></td>
</tr>
<tr>
<td>- dispatch super → sub</td>
<td>- dispatch role ← base</td>
</tr>
<tr>
<td><strong>Substitutability</strong></td>
<td><strong>Translation polymorphism</strong></td>
</tr>
<tr>
<td>- pass an instance of sub class where the super class is expected</td>
<td></td>
</tr>
</tbody>
</table>
Substitutability

Are the following assignments legal?

<table>
<thead>
<tr>
<th>Employee emp= ...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Person person= ...</td>
</tr>
</tbody>
</table>

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

\[
\begin{align*}
\text{lift}(b1, t1) & \rightarrow r1 \\
\text{lift}(b2, t2, \text{RoleA}) & \rightarrow r4 \\
\text{lift}(b2, t2, \text{RoleB}) & \rightarrow r5
\end{align*}
\]
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
to the callin call target

```java
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 mapping

- **Eat the cake and have it**
  - flexibility of multiple instances
  - no disadvantage of “object schizophrenia”
  - instances are “almost the same”

**Pending: Optimizations** (compiler / runtime)
## Detailed Comparison

### Inheritance vs. Role Playing

<table>
<thead>
<tr>
<th>Inheritance</th>
<th>Role Playing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import / acquisition</td>
<td>Callout binding</td>
</tr>
<tr>
<td>dispatch sub → super</td>
<td>dispatch role → base</td>
</tr>
<tr>
<td>Overriding</td>
<td>Callin binding</td>
</tr>
<tr>
<td>dispatch super → sub</td>
<td>dispatch role ← base</td>
</tr>
<tr>
<td>Substitutability</td>
<td>Translation polymorphism</td>
</tr>
<tr>
<td>pass an instance of sub class</td>
<td>lowering role → base</td>
</tr>
<tr>
<td>where the super class is</td>
<td>lifting role ← base</td>
</tr>
<tr>
<td>expected</td>
<td>two-way substitutability</td>
</tr>
</tbody>
</table>
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
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 polymorphism hides difference, use \texttt{roleEQ()} for relaxed comparison
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)**
  - Java += roles, teams, bindings
  - OTJLD 1.0 *(current 1.2)*

- **Object Teams Development Tooling**
  - Java Compiler += OT/J constructs
  - JDT for OT/J (code assist, ui, launch ...)

- **Other**
  - OT/Equinox: Equinox += aspect bindings

- **Application**
  - Case studies (project TOPPrax)
  - Class room
  - OTDT
  - UML2 tools (base on EMF/GEF/GMF)
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

```java
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)
Consistent Polymorphism

- Consistently specialize a set of role classes
- No danger of mixing roles from different teams
- Scalable Template&Hook
Q: which type “Move” is used?
A: the one valid in the current context

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!

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

Consistent Polymorphism

A: you have to know what game — instance!

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

```java
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 Inheritance

**Roles are virtual classes**
- can be overridden in sub-teams
- overriding role implicitly inherits from overridden role
  \[ \rightarrow \text{mild form of multiple inheritance} \]

\[ \rightarrow \text{two kinds of super-call:} \]
- \text{super}(); (constructor) – \text{super.m}(); (method)
- \text{tsuper}(); (constructor) – \text{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 \( \leq: B0 \)
- cannot handle multiple subtypes of **R0**

![Diagram](attachment:diagram.png)
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

```java
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);
    }
}
```

separate namespaces:
```java
import base mybase.Expr;
```

dispatch on visit:
* selects the Function

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

Adding instance dispatch to method dispatch

```
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);
  }

  separate namespaces:
  import base mybase.Expr;

  Smart Lifting selects:
  ‣ Team: Function
  ‣ Role: Node Type
```
Applying Generics to Role Playing
Problem:

- replace callin binding requires 2-way compatibility

```java
callin T1 roleMethod() {
    T1 oldResult = base.roleMethod();
    return new T1();
}
```

```java
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

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

```java
<E extends T1> E roleMethod() <- replace T1+ baseMethod();
```
Recall this structure

TeamComponent

SpecificRole1

commonBehavior \rightarrow \text{specificBehavior1}

SpecificRole2

commonBehavior \rightarrow \text{specificBehavior2}

CommonSuperRole

\text{commonBehavior} ()

SpecificBase1

\text{specificBehavior1} ()

SpecificBase2

\text{specificBehavior2} ()
**But,**

how can this method be typed?

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

want to allow `SpecificBase1` and `SpecificBase2`

**answer:**

**new kind of type bound:**

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

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

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;
    }

}

}

public team class Graph
{

public class Node {

}

public class Edge {
    Node getStartNode();

}


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

Layering – Detail

```java
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();
    }
}
```
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

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

void transferPresidency(Person as President currentP, Person newP) {
    currentP.migrateToBase(newP);
}
```
A Meta Model for Object Teams
Combinations?
- Role & Team: nested Team
- Role & Base: layered Team, Role-of-Role
- Team & Base: stacked Teams

Model evolution?
- group of classes ↔ collaboration of roles
In order to explain Roles you first have to explain Roles.
Architecture with Object Teams
Traditional decomposition

New feature requested
- identify affected classes
- no way to define new feature as a module 😞
We need to zoom out!

See a new solution?

No.

Try again!
Can you see it now?

Now?

Now!
In truly 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 something 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

Bundle A

- CA1
- CA2

Bundle B

- CB1
- CB2
- CB3
- CB4

Bundle C

- CC1
- Team1
- R1
- R2

MANIFEST.MF

... 
Require-Bundle: B
...

`<extension point="org.objectteams.otequinox.aspectBindings">`

`<aspectBinding>`

`<basePlugin id="B"/>`

`<team class="Team1" activation="ALL_THREADS"/>`

`</aspectBinding>`

`<extension point="org.objectteams.otequinox.aspectBindings">`

`<aspectBinding>`

`<basePlugin id="B"/>`

`<team class="Team1" activation="ALL_THREADS"/>`

`</aspectBinding>`

R1 «require» Bundle C

R2 «aspectBinding» Bundle C

R2 «playedBy» Bundle A

R2 «playedBy» Bundle B

IB2 extension point
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
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)
        }
    }
}

base class unchanged
new interface as a view

Given this data class

class Flight {
    private List<Segment> Segments;
}

Traditionally

apply refactoring: Encapsulate Collection
Observer-Mediator-Actuator

mediatorStorageField : DataStructure

ObserverRole
写入

ActuatorRole
读取

MediatorRole1

MediatorRole2

C1  C2

C3  C4

adaptation
base application

拦截
（非修改型）

装饰

拦截
（修改型）
Implementing a stateful relationship

```
Course
  title
  hasStarted()

Student

Enrollment
  grade
  attendance

Course
  enrolled
  Participant
    grade
    attendance
  title
  hasStarted()

Student
```

«playedBy»
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