Translation Polymorphism in Object Teams

Bericht-Nr. 2004/05

Stephan Herrmann
stephan@cs.tu-berlin.de

Christine Hundt
resix@cs.tu-berlin.de

Katharina Mehner
mehner@cs.tu-berlin.de

ISSN 14369915

Translation Polymorphism in Object Teams

Stephan Herrmann Christine Hundt* Katharina Mehner*
{*stephan,resix,mehner*}*@cs.tu-berlin.de*

Abstract

In this paper we present the mechanisms of lifting and lowering which have been incorporated into recent programing languages. In these languages, lifting and lowering are key features in the integration of static, class based inheritance and instance based composition with delegation. We present the embedding of these concepts into our model Object Teams. We elaborate the details of rules and constraints at the type level, which give to the approach the capability of non-invasively integrating modules with different inheritance structures. These rules setup a new kind of substitutability called translation polymorphism. We show how translation polymorphism integrates with subtype polymorphism. By generalizing over the presented techniques we conclude that lifting opens a second dimension for dispatch in object-oriented languages, be supplementing method dispatch with instance dispatch.

1 Introduction

The work presented in this paper is based on language research in the areas of collaboration based development, role modeling, aspect oriented programming, and declarative software composition.

While integrating findings from these diverse fields, we observe that different technologies seem to converge towards a new set of mechanisms that directly relate to the very fundamentals of object-orientation:

- Method dispatch will be supplemented by *instance dispatch*,
- Subtype polymorphism will be supplemented by *translation polymorphism*

It is the goal of this paper to present what we mean by instance dispatch and translation polymorphism.

With these concepts in place, some traditionally controversal topics will be settled in a constructive way. Given that *sharing* is a fundamental idea in object-orientation, we will show that we can smoothly combine static sharing — as realized by inheritance — with dynamic, instance based sharing — as used in delegation based languages.

The conceptual glue that allows us to integrate concepts from different fields of research is a special kind of module called *team*. The concept of teams combines the following desirable properties:

*This work has been supported by the German Federal Ministry for Education and Research under the grant 01ISC04A (Project TOPPrax).

- Teams realize views that group sets of interacting classes and objects.
- Teams define the context for role objects regulating the scope of roles.
- Teams serve as the unit for dynamic aspect activation.
- Teams may be used as declarative module connectors.

1.1 Focus of this paper

After an introduction to Object Teams this paper will focus on what we call dynamic instance dispatch, which is a mechanism for transparently navigating a graph of instances that delegate to one another. Sect. 1.2 will motivate this mechanism by a brief analysis of potential problems of approaches that use instance based software composition and delegation.

At this level, we will present two mechanisms called lifting and lowering and answer the following questions:

- When should instance dispatch be triggered?
- What context information is needed to perform the dispatch?
- When should roles (atomic instances involved in delegation) be created?

The body of this paper will elaborate precise rules for issues related to typing, answering these questions:

- How can the most suitable role type be found for on-demand role creation?
- How can type safety be ensured statically?
- How can mappings between inheritance hierarchies with different structures be supported?

Motivations for these questions will be given as we go along.

Finally the new concepts will be related to traditional methods dispatch. This results in an integrated view of method dispatch which includes solutions to some issues of polymorphism that have been reported regarding languages for aspect-oriented programming.

The concepts presented in this paper have been developed as part of the language ObjectTeams/Java, for which tool support (a compiler, a run-time environment, and an integrated development environment) is being finalized and consolidated at the time of this writing.

To-date a number of examples have been programmed in this language and first industrial application starts right now within the funded project TOPPrax.

Before presenting our contributions, we first re-investigate the controversy of static vs. dynamic sharing using inheritance or delegation, respectively. We identify four issues that should be settled for a smooth integration of both concepts, paving the road for a synergetic combination of language capabilities.

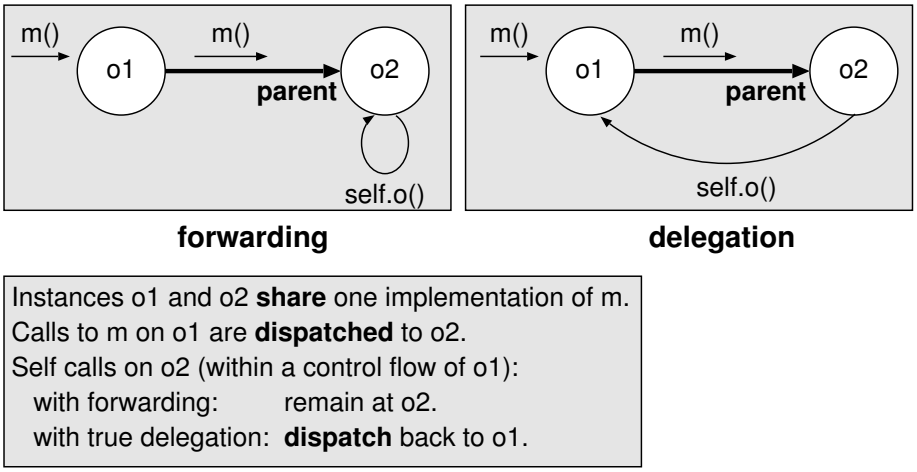

Figure 1: Forwarding/delegating to a parent instance

1.2 Sharing and dispatching revisited

In the early days of object-orientation, *sharing* and *dispatching* have been identified as fundamental concepts for structuring programs [27, 34]. Sharing is used for extracting common features to a single non-redundant location. Dispatching is then applied to define the inter-operation between shared and un-shared parts. The most illustrative example is the Template Method design pattern: a template method is implemented in a general class and calls other methods ("hook" methods) that have to be provided by more specific classes. All specific classes share the implementation of the template method. In order to invoke the suitable hook method dynamic dispatch has to be applied.

Two mechanisms have been identified that enable this kind of sharing: *inheritance* and *delegation*. While the mainstream has opted for the former, static approach, the interest in more dynamic approaches using some form of delegation has never ceased.

In languages that support delegation, *compound objects* can dynamically be assembled from atomic instances, which delegate to one-another. We see three reasons why instance based composition using delegation is desirable in object-oriented languages.

1. Assembling an object as a chain of delegating instances easily provides combinational flexibility comparable only to mixin-based approaches. Both approaches allow to freely combine the behavior of different classes.

2. Composition can be deferred until run-time with the advantages of further flexibility. This dimension of flexibility refers to the fact, that type combinations need not be fixed at compile time. Even after instance creation, new facets (delegating instances) can be added at will.

3. Instance based structures allow for multiplicities that cannot be achieved using purely static approaches. There is no limit on how many instances share a common parent instance to which they delegate certain messages.

The abundance of design patterns that apply "delegation"[1] demonstrates that static sharing is not sufficient in practical software development. To some extent these patterns complicate designs by adding infrastructural overhead and introducing new trade-offs. Thus, some pattern based solutions are far from optimal and call for more fundamental support for instance based sharing. A particular problem from which many of the considered design patterns suffer has been described as "object schizophrenia" [25, 26]. From the discussion about object schizophrenia, we draw the conclusion that the question whether two parts of a compound object share a common identity depends on the context. Thus object schizophrenia can only be remedied effectively, if a language has explicit support for views. We will come back to this issue after introducing our concept of teams in Sect. 2.1.

Given that instance based composition helps to solve practical problems, four issues have to be settled:

Issue (a). How does method *overriding* work for compound objects? Most authors agree that delegation should allow an object to override methods of its parent(s), which requires late binding of `self`. In this setting, messages must be sent to the more specific instance in order for its method overrides to be effective. More advanced approaches also provide means, by which overrides can adapt an instance globally, i.e., also for clients that access the shared instance (parent) directly (see, e.g., "constituent methods" in [8].)

Issue (b). How is the set of instances, which contribute to a compound object, *created*? It is usually not satisfactory if the creation and linking of atomic instances has to be done manually, since this significantly bloats the program.

Issue (c). How does *navigation* along the references between atomic instances work? One direction is usually supported by an explicit "parent" link, which also serves as the basis for delegation. However, navigating from a parent to its children is non-trivial. Firstly, the parent link usually has no reverse link, i.e., the parent doesn't explicitly know about its children. Secondly, there may be more than one child for a given parent.

Issue (d). How is *typing* affected by delegation? In order to use delegation in place of class based inheritance, a child instance may be considered to conform to the type of each of its parents.

In this paper we present our solution to the above issues. The Object Teams programming model combines class based inheritance with a variant of role objects that use forwarding and delegation for sharing. By integrating static and dynamic sharing together with a new module concept called teams, we combine and transcend the best of both worlds.

We will shortly introduce two mechanisms called lifting and lowering, which address issue (b) and issue (c) above. Sect. 4 will show details of overriding and dynamic method dispatch (issue (a)).

[1] Technically, we prefer the notion *forwarding* for the style of message passing used in standard design patterns. Unfortunately "delegation" has been introduced in [6], although true delegation would require the option of overriding using late binding of self (see Fig. 1).

Many papers on delegation focus on typing (issue (d)). In Object Teams this non-trivial discussion is essentially avoided by restricting substitutability: roles do not directly conform to the type of their base classes. With respect to role-base pairs, regular subtype polymorphism is replaced by a new kind of implicit type-safe conversion, called *translation polymorphism*. Translation polymorphism in turn is based on lifting and lowering. The remaining questions regarding type-safety will be discussed in Sect. 3.3.

Structure of this paper.
Sect. 2 will set the stage by showing how Object Teams extend the fundamental object oriented concepts for sharing and dispatching. Here the terminology of Object Teams will be introduced and illustrated by a running example. Sect. 3 will present the algorithm for "smart lifting" which is the sophisticated mechanism behind translation polymorphism. Sect. 4 shows how method dispatch integrates into this setting, presenting a solution to the problem of aspectual polymorphism. Sect. 5 interprets the concepts and mechanisms as two-dimensional dispatch, which is a rich, context aware concept for associating the most suitable implementation to a given method call. We will conclude with a discussion of related work (Sect. 6) and a look at future work (Sect. 7.1).

2 Object Teams

In order to set the stage for the translation polymorphism and its mechanism of smart lifting, we briefly present the Object Teams model. Sect. 2.2 will introduce a running example for this section. Sect. 2.3 associates the lifting and lowering translations with dataflows that cross the boundary of a team. Here you will find a definition of the term "translation polymorphism". Sect. 2.4 describes the problems which arise if independently developed modules are to be integrated using role-base relations. Finally, smart lifting is introduced as the answer to such structural mismatches.

2.1 Terminology and Concepts

As a brief introduction to Object Teams we will give definitions for its central notions and relate these to the concepts discussed in Sect. 1.2.

2.1.1 Kinds of classes

Object Teams discriminate different kinds of classes which relate to each other.

Definition 1 (Team)
A team is a container for a set of interacting roles. Each inner class of a team is automatically a role class.

Definition 2 (Role)
A role class is an inner class of a team. Each role instance is furthermore contained in a team instance. A role class may be bound *or* unbound.

Definition 3 (Bound Role)
A bound role class has a `playedBy` *declaration specifying a base class, such that each role instance will be linked to a base instance of that class. In Object Teams the role-base link is immutable once established.*

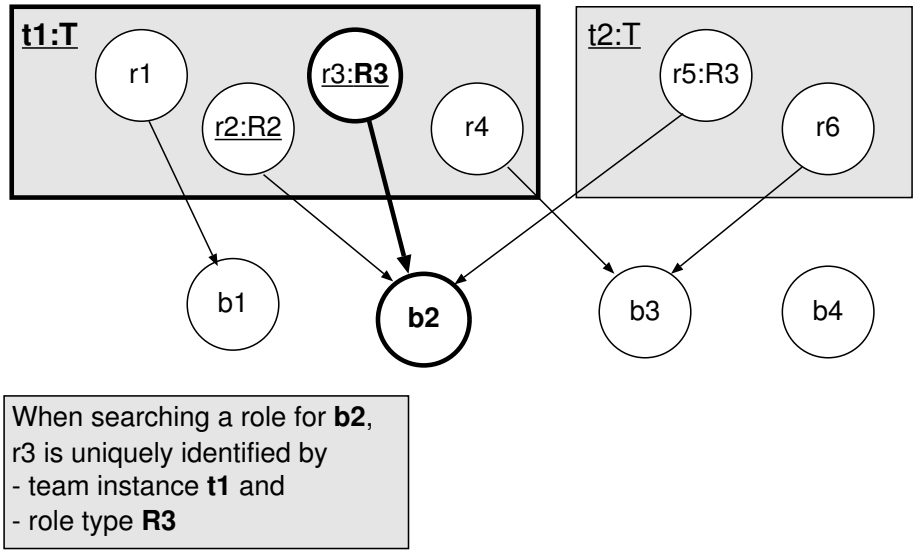

Figure 2: Role multiplicities and identification

Definition 4 (Base)
Any class can act as a base class, which simply means that the class appears in the `playedBy` *clause of a role. An instance can be the base of several role instances simultaneously.*

The role-base relation can be identified with instance based sharing: any number of role instances can share a common base instance. The invisible base link directly corresponds to the parent link in other languages.

2.1.2 Mechanisms for navigation

The fundamental mechanisms to be presented in this paper are lifting and lowering on which translation polymorphism is based.

Definition 5 (Lifting)
Navigating from a base instance to a role instance is called lifting. This may involve creation *of a role instance and has to consider the multiplicities of the role-base binding.*

Definition 6 (Lowering)
This operation is the inverse of lifting, i.e., navigating from a role instance to its base instance. It is trivial because each role instance has a link to its base instance. Yet, lowering is supported at the language level, since Object Teams for conceptual reasons don't give explicit access to the base link.

Given the multiplicities of the role-base binding, lifting is only well-defined if additional context information is given (see Fig. 2).

Definition 7 (Context of Lifting)
Lifting takes a base instance, *must specify the expected* role type *and must provide a* team instance.

Fig. 2 shows two different kinds of sharing: roles `r4` and `r6` share a common base `b3`, but within each team instance (`t1` or `t2`) there is only one role for `b3`. This way each team instance realizes a view on a potentially shared set of base instance. Sharing between different roles of the same team instance (`r2` and `r3` in the figure) is less common, but can be disambiguated by the types of those roles (`R2` vs. `R3`).

2.1.3 Method bindings

The language ensures that the context for lifting is always provided. In order to see how this is done, two integration mechanisms need to be shown, both of which depend on an existing role-base relation using a `playedBy` declaration.

Definition 8 (Callout)
A role class may declare by callout *method bindings that specific messages shall automatically be forwarded to the linked base instance.*

Definition 9 (Callin)
By a callin *method binding a role class may declare method interception in the vein of advice-weaving as it is used in aspect-oriented programming. A callin binding has to provide one of the combinators* `before`, `after` *or* `replace` *specifying how to compose the role method into one or more existing methods of its base class.*

Both kinds of method bindings define the inter-operation between a role and its base. In a program that uses callout binding only, method dispatch for role instances is mere forwarding. Overriding can be added by callin replace bindings. Combining callout and callin bindings we achieve true delegation. Note, that overriding must be declared explicitly. Thus, a role method which happens to have the same name as an existing base method will not override that method, unless a callin replace binding is explicitly given. Conversely, a role method need not have the same name as the base method which it overrides. These rules improve the independence between roles and bases.

2.1.4 Team activation

Teams realize the concept of dynamic aspects. For this purpose it is a most useful feature, to dynamically activate or deactivate individual team instances.

Definition 10 (Team activation)
In order for any callin binding to be effective a corresponding team instance has to be activated. *Activation may occur by invoking method* `activate()` *on the team instance or in a more controled fashion by a new block-construct*

```
within(teamInstance){ //statements }.
```

Assigning aspect activation to the team level has the advantage of better modularity and consistent activation of a group of interacting roles in only one step.

2.1.5 Role life-cycle

It has been said that lifting may need to *create* a role instance. In this we follow the goal that developers should not be required to manually handle role creation (although explicit role creation is still possible). On the other hand identity and state of role instances should persist across lowering and lifting. Both goals are reconciled by simply storing a map of base and role instances within each team instance. Lifting first tries to retrieve an existing role from this map. Only if this failed a new role is created and entered into the map. As a result, on-demand role creation is fully transparent to the programmer.

Under certain circumstances several base-role maps are needed per team. The exact rules will be given in Sect. 3.2.

To complete the role life-cycle, we would like to add an option of canceling existing roles. We will discuss problems of this task in Sect. 7.1.

2.2 Example — UML Editor

The first case study for Object Teams has demonstrated that teams are very well suited for implementing in the Model-View-Controller style [33]. As a running example in this section we elaborate some details of a repository based UML editor in the vein of PIROL [11].

Neglecting the distributed architecture of [11] we still identify two modules: the shared model ("repository") — in this case a meta model of object oriented structures — and a diagram editor.

As a starting point we consider only two classes from the model: `Class` and `Relation`, the latter generalizing over associations and inheritance. A minimal set of methods suffices to demonstrate the mechanisms.

```
package model;
class Class
{
   void     addRelation  (Relation rel) { ... }
   Relation getRelToClass (Class other) { ... }
}
class Relation
{
   Class getClass1 () { ... }
   Class getClass2 () { ... }
}
```

Consider next an editor for UML class diagrams, where each `Class` is to be represented by a `ClassFig` and each `Relation` is shown by a `Connection`, i.e., a (possibly decorated) line between two class figures. Considering that each `ClassFig` and `Connection` adds layout information to the corresponding model element, it is natural to design the graphical entities as roles of corresponding model entities.

```
public team class ClassDiagram
{
  class ClassFig playedBy model.Class
  {
    abstract
      Connection getConnection(ClassFig other);
    void addConnection(Connection con)
      { ... }
    ...
  }
  class Connection playedBy model.Relation
  {
    abstract ClassFig getStartFig();
    abstract ClassFig getEndFig();
    ...
  }
}
```

While the graphical information is local to the `ClassDiagram` team, classes in this team also need access to the structural information defined by their base classes. This is sketched by the abstract methods `getConnection`, `getStartFig` and `getEndFig`. These methods are declared abstract, because their role classes provide no implementation, but implementation is to be bound by *callout* method bindings. Class `ClassFig` does however provide an implementation for `addConnection`, by which a new line is drawn in the diagram. One potential use of this method is for updating the view when the model has changed. What is missing at this level is just the trigger for invoking `addConnection`.

At the method level integration of both modules is defined by the following method bindings:

```
public team class ClassDiagram
{
   class ClassFig playedBy model.Class
   {
        ...
        getConnection -> getRelToClass
        addConnection <- after addRelation
   }
   class Connection playedBy model.Relation
   {
        ...
        getStartFig -> getClass1
        getEndFig   -> getClass2
   }
}
```

In the syntax of ObjectTeams/Java the right arrow (`->`) denotes a callout binding and the left arrow (`<-`) a callin binding, which must be further specified with a combinator, here: `after`. The callout bindings setup a forwarding dispatch as discussed above. The callin binding using `after` is a variant of overriding, which does not replace the given base method, but only adds additional behaviour at the end of its execution. Thus, callin bindings with `before` or `after` combinators can be interpreted as a purely additive mechanism for

inserting triggers into base methods.

For all four bindings presented so far, method signatures are "similar" enough for binding methods without signature adjustments. The language also supports declarative parameter mappings, which are not in the scope of this paper. By "similar" we mean that signatures can be matched given a few implicit translations which will be investigated next.

2.3 Dataflows

The idea behind teams as context for roles is illustrated by the concept of *views*: Within a team, base instances are seen as decorated by role instances; outside the team, roles are usually not visible. This kind of encapsulation is the key to improved modularity, where the worlds of base instances and role instances are self contained domains with very little coupling. This understanding of views gives the background for defining the points within a program where lifting and lowering take place.

Normally, data flows between a team and the outside are declared by callout and callin method bindings. Here is an intuitive rule for the translations to take place:

> Any role instance being passed from a team to the outside is implicitly lowered to yield the linked base instance.[2]
> When passing a base instance into a team that may have a role for that particular base, the base instance is implicitly lifted to its corresponding role instance.[3]

Of course method signatures have to correspond to the types involved in such data flows. As an example for lowering consider this binding (this time shown in the optional long form):

```
Connection getConnection(ClassFig other)
  -> Relation getRelToClass(Class other)
```

The argument `ClassFig other` is implicitly lowered to its base of type `Class` thus matching the signature of the base method. Lifting can be shown at the same binding. The return value being declared of type `Relation` is implicitly lifted to its role `Connection`, which again agrees with the expected type, here the return type of `getConnection`.

Using these translations, it is possible to implement both worlds — the diagram team and the model package — using their specific abstractions only. Method bodies of the team and its roles need not (and should not) mention any base classes. Nor are role classes mentioned in the base package.

When considering two directions of method bindings (callout and callin) and three positions (call target, argument, return value), the following translations can occur.

	call target	argument	return value
callout	lower	lower	lift
callin	lift	lift	lower

[2]This rule does not cover the case of externalized roles, i.e., roles which are explicitly passed to the outside. Externalized roles are not subject of this paper.

[3]Base instances for which no role type exists are passed unchanged. The same holds for basic type values, such as `int` and `char`.

Two more places of translation are supported by the language, which will be shown using the example of the UML editor.

First, there should be means to add a class to a diagram for which no figure exists yet. This operation will be called from outside the team where instances of `Class` are at hand. However, the body of that operation requires a `ClassFig` to be inserted into the drawing. In order to specify that a base instance should be lifted when being passed into a team-level method the following syntax is introduced:

```
public team class ClassDiagram
{
   void addClass (Class as ClassFig cls)
     { ... }
}
```

The new type declaration `Class as ClassFig` requires clients of this method to provide an instance of `Class`. Inside the method, however, the value is typed to `ClassFig`. The required lifting translation is inserted by the compiler. Again the syntax supports a clear separation of scopes dealing with bases only or roles only. Once some initial roles are available within a team, other roles can be retrieved by callout-bound methods like `getConnection`. So from an initial role the team may retrieve a *graph* of role instances as a view of a *graph* of base instance.

Secondly, implicit lowering can be used anywhere in a program. Because lowering is a trivial translation a role value can be attached to any base-typed identifier, meaning that the base instance is extracted prior to attachment. So the following code is type correct:

```
public team class ClassDiagram
{
   ClassFig rootClassFig;
   Class getRootClass ()
   {
     return rootClassFig;
   }
}
```

For programmers this feels just like subtype-polymorphism: assignments (or attachments of method arguments, or return statements) may drop static type information. Reverting this information requires some efforts (lifting or down-casts) which also have the possibility of failing at run-time, while up-casts and lowering will never fail. This analogy should not be surprising, considering that the role-base relation is a specific form of instance-based inheritance by delegation.

Completing the analogy to subtype polymorphism we can now define the notion "translation polymorphism".

Definition 11 (Translation Polymorphism)
Translation polymorphism defines the rules by which role values can be substituted for base types and vice versa. In addition to switching between different static type information, translation polymorphism also involves the execution of the translation operations lifting or lowering.

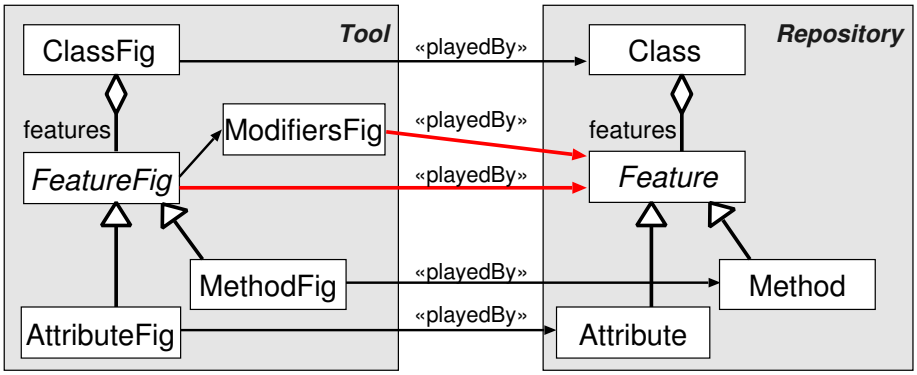

Figure 3: Mapping different hierarchies by `playedBy`

The primary design goal behind translation polymorphism was, to construct a language that makes navigating between bases and roles as convenient as changing views using regular subtype polymorphism.

Similar to subtype polymorphism, typing rules must be defined to ensure that lifting and lowering produce no unexpected runtime errors. Lifting is more challenging than all other conversions, since it may need to create a new role instance or select between different role instances of a given base instance. The next section will illustrate these issues with our running example. Thereby the concept of lifting will be refined to what we call "smart lifting". Sect. 3.2 presents the algorithm implementing smart lifting.

2.4 Mapping hierarchies

The preceding discussion has focussed on dispatching between role and base instances, neglecting the possibilities, that the classes involved may be part of arbitrary inheritance hierarchies.

However, our goal is to combine the best of the worlds of static and dynamic sharing — inheritance and delegation. As an illustration for the kind of questions that remain we augment our running example with some details of classes, namely attributes and methods (see Fig. 3).

At the repository side we introduce an abstract class `Feature` with two subclasses `Attribute` and `Method`. The same structure is given at the tool side by `FeatureFig`, `AttributeFig` and `MethodFig`. Each pair of corresponding classes is connected by a `playedBy` clause in the role class. Now, let's assume a function

```
FeatureFig[] getFeatures()
```

in the role class `ClassFig`. This method is to be bound by callout to a corresponding repository method

```
Feature[] getFeatures()
```

Obviously, each element in the resulting array should be lifted from `Feature` to `FeatureFig`. However, both classes are abstract and no roles of exactly this type can be created.

By intuition it is evident that each concrete object in the base array is either an `Attribute` or a `Method` and should thus be lifted to one of the concrete role

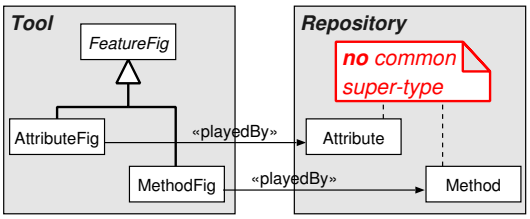

Figure 4: Variant: introducing a super-type

classes `AttributeFig` or `MethodFig`. This simple example demonstrates, that lifting must pay attention to the dynamic type of the base instance to be lifted. Only by looking at that type, a proper role type can be selected.

Definition 12 (Smart Lifting)
Smart lifting is a lifting translation that considers the dynamic type of a base instance in order to return a role instance of the most suitable role type.

Consider next a class `ModifiersFig`, which is used by the tool for displaying feature modifiers (public, synchronized ...) by a set of icons. This abstraction has no corresponding class in the repository. Since at the repository level, the information of modifiers is stored in `Feature` objects, `ModifiersFig` are modelled as a view of `Feature`s. Navigating from a `Feature` to its `ModifiersFig` means to navigate from one role to another regarding the same base object. This requires the lifting operation to distinguish roles based on the statically required role type. The same `Feature` object can be lifted to a sub-class of `FeatureFig` *or* to a `ModifiersFig` object. If static typing is strong enough this distinction is unambiguous and happens without further intervention by the programmer.

Finally, the given mapping still works if the repository model lacks a type `Feature`. A missing super-type may be introduced as a (possibly abstract) role which is not bound to a specific base class. Looking at Fig. 4 we see two incommensurable classes `Attribute` and `Method`. Only in their mapping to `AttributeFig` and `MethodFig` the generalization `FeatureFig` is introduced. Now the tool can polymorphically handle `FeatureFig`s of both types, while the repository remains unchanged. Thus, translation polymorphism introduces new options of using sub-type polymorphism in places where it has not been planned for.

As we have already pointed out in [13], the capabilities to map mismatching structures is a key feature for integrating independently developed components. Mechanisms in the vein of our lifting translation (sometimes called wrapper recycling) have been used for PIROL [11, 13], PCA [22], their implementation JADE [10], LAC [12] (a prototype for Object Teams) and Caesar [21].

Still, to-date neither the central role of lifting in the design of these languages, nor its precise rules have been worked out. It is the goal of the following section to define smart lifting such that type-safety is guaranteed. In doing so, we had the choice of defining rather conservative rules, that would provide unconditional type-safety but work only for simple situations of hierarchy mappings. At the other extreme, all checks could have been postponed until run-time in order to allow arbitrary structures to be mapped with greatest flexibility.

Our solution is a combination of strict static type safety and some more flexible options that combine compiler warnings and run-time checks as the safest-possible way to deal with really difficult program structures. Sect. 3.2 defines the constraints and the algorithm for standard situations. Sect. 3.3 will elaborate corner cases.

3 Smart Lifting Defined

The previous section has shown the central role that the lifting translation may play in a language design that combines inheritance and delegation. We have shown the embedding of lifting and lowering into a language the support modules for collaborating roles, in our case: teams. We have finally given some examples, where the role type to be used for lifting could only be determined at run-time using the dynamic type of a base instance. Thereby we refine lifting to "smart lifting", which allows to bridge mismatches between different inheritance hierarchies of base and role classes.

In this section we will present the rules for role-base bindings and lifting.

3.1 Concepts

As we have seen, inheritance and subtype polymorphism have significant impact on the lifting translation. We will now define the concepts and algorithm of smart lifting. Later we will come back to the different stages and analyze those corner cases that need special treatment.

Basic setting. Fundamental to lifting is the existence of role-base bindings at the class level. Binding a role class to a base class will be denoted as *playedBy(R)=B*. In that case `R` is a bound role (cf. Sect. 2.1.1, definition 3). Each lifting translation involves three static types (cf. Sect. 2.1.2, definition 7): A base class `B`, a team class `T` and a role class `R`, where `R` is an element of `T`. In addition to the static typing of lifting, at runtime two instances `b` and `t` are given whose dynamic types may be subtypes of the above, i.e., $\texttt{B}_{dyn}$ and $\texttt{T}_{dyn}$. The result of lifting is an instance `r` of dynamic type $\texttt{R}_{dyn}$ which must be shown to conform to the required type `R`.

Lift methods. The Object Teams compiler generates lift methods as methods of each team which has bound roles. One such method is generated for each bound role type `R`. Given that `R` is bound to `B`, this method must be capable to handle base objects of all sub-classes of `B`. For a given triplet (`T, B, R`) the lift method in `T` has the following signature

```
R liftTo$R(B b).
```

This leads us to our first constraint:

Rule 1 (bindings exists)
When lifting `B` to `R` with respect to team `T`, there must be a role `R1`$\in$`T`, `R1`$\sqsubseteq$`R`, and a base class `B1`$\sqsupseteq$`B`, such that playedBy(`R1`) = `B1`.

For illustration of this rule see Fig. 5. The rule states, that there must be a path from `B` to `R` which may only involve up-casts and lifting in order to translate an instance of class `B` to an instance of type `R`.

Lifting is always performed on behalf of a team instance. Thus, dynamic binding of the team is performed first. Throughout the lifting process, only

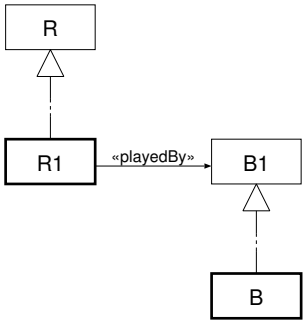

Figure 5: Lifting in the presence of inheritance

the dynamic type $\texttt{T}_{dyn}$ is of interest. This leaves as an obligation to show that the role `r` retrieved by `t` is conform to expectations based on the static type `T`, which is mainly an issue of the family polymorphism involved (see 3.4).

Role hierarchies. All role classes of a team are partitioned into disjoint role hierarchies spanned by a set of root roles, which are bound role classes $\texttt{R}_{root}$ that do not inherit from another bound role class. These hierarchies are completely independent with respect to lifting. Binding a base type to a hierarchy $\texttt{R1}_{root}$ does not affect any other hierarchy $\texttt{R2}_{root}$. The same holds on the instance level: lifting a base object `b` to hierarchy $\texttt{R1}_{root}$ does not affect any other hierarchy $\texttt{R2}_{root}$.

Role caches. Since the lifting/lowering translation shall preserve state, role objects must be cached. Each team instance contains one role cache for each of its role hierarchies. Each cache is a mapping of base objects (keys) to role objects (values).

The need for separate caches for separate role hierarchies can be illustrated by the example of class `ModifiersFig` above. In the given setting two caches will be used, one for `FeatureFig` and one for `ModifiersFig`. When a `Feature` is lifted to type `FeatureFig` the first cache is used, while lifting to `ModifiersFig` uses the second cache. It is thus ensured, that two different roles for the same base object don't interfere at the level of role caching.

3.2 Algorithm

Given the above concepts and definitions, for each triplet (`T,B,R`) a lift method in `T`

```
R1 liftTo$R1(B1 b)
```

is selected whose realization is defined below.

Step 1 (lift method selection)
According to Rule 1 at least one appropriate pair (`B1,R1`) exists. From these pairs the most general `R1` selects the lift method to use. Given `R1` a lift method is uniquely identified since a team contains exactly one lift method for each bound role.

If no lift method is found, this is due to a missing role-base binding, thus the requested lifting is not possible, resulting in a compile-time error. The following steps define the semantics of a lift method.

Step 2 (cache lookup)
The appropriate cache is determined by static lookup of the root role $\texttt{R}_{root}$ *of* `R`*. Using* `b` *as the key, a cached role* $\texttt{r}_{cache}$ *is retrieved from the cache of* $\texttt{R}_{root}$*. If* $\texttt{r}_{cache}$ *is* `null` *proceed with the next step. If* $\texttt{r}_{cache}$ *is not* `null` *try to cast it to* `R` *and return it. If casting fails, throw a* `WrongRoleException`*.*

The exception case will be discussed in Sect. 3.3.1.

If all goes well, cache lookup will retrieve a role object created by a previous lifting translation or by explicit role creation. While implicit role creation due to lifting is by far the normal case with Object Teams, the latter case requires a further constraint:

Rule 2 (explicit role creation)
Explicitly creating an instance of a bound role class, i.e., creation other than via lifting translation, must ensure proper setup of a corresponding base object and registration in the corresponding role cache.

In ObjectTeams/Java this rule is ensured by a constraint on role constructors and by generating one additional statement into these constructors, which inserts the role-base pair into the corresponding role cache.

Rule 3 (role constructors)
Each client constructor of a bound role class must as its first call contain a base constructor invocation, which has the syntax `base(`*args*`)`*. Alternatively, self-calls are allowed, provided they eventually lead to a base constructor call.*

The given base constructor syntax has the effect of creating a base object passing the arguments *args* to its constructor. A link to the base object is internally stored in the role object. Inserting the role-base pair into the role cache happens immediately after base object creation.

These rules ensure that explicitly created roles (a) have a valid base link and (b) are available in the corresponding cache.

In case cache-lookup finds no role object, a fresh role has to be created. Selection of the actual role class is based on some static analysis, which is performed by the compiler for each team class.

Step 3 (role hierarchy analysis)
The hierarchy of sub-roles of `R` *in* `T` *is statically analyzed in order to create a unique mapping from base classes to role classes with respect to the considered team class. This analysis inspects all sub-roles of* `R` *performing (1) elimination of irrelevant roles and (2) base class folding.*

Step 4 (elimination of irrelevant roles)
When traversing down the role hierarchy, a super-role `R1` *of role* `R2` *is discarded iff* `R1` *is not bound to a base class or bound to the same base class as* `R2`*.*

If a role class has no `playedBy` clause, it inherits this binding from its superclass, if such a binding is present there. The above elimination achieves that more specific bindings override less specific ones. This helps to prevent the problem of mismatching roles (see 3.3.1) for common situations.

Step 5 (base class folding)
From all roles that passed Step 4 the bound base classes are inspected. If one base class is bound by more than one role in the set, all roles bound to this class are discarded from the set of lift candidates and the compiler issues a warning "Potential ambiguity in role binding".

The compiler warning signals that the program may or may not fail at runtime. Ambiguities will be discussed in detail in Sect. 3.3. A program that compiles without such warnings is guaranteed to contain only safe liftings.

Step 6 (conditional role creation)
Given the base-to-role mapping from the previous steps, and given the dynamic type of the base argument B_{dyn}*, from all base classes contained in the mapping the most specific super class of* B_{dyn} *is chosen. By the given base-to-role mapping this base class* B_{dyn} *determines the role class to be used. The lift method finally creates a role object of the determined role class.*

Step 7 (role setup)
Roles are created using a generated constructor that does nothing but establish the (invisible) role-base link. If a method `initializeRole()` *is present in the role class, it is invoked. This method may access the enclosing team and the linked base object. If in this process any exception is thrown, lifting aborts throwing a* `LiftingFailedException`*. If no exception occurs, the initialized role object is entered into the role cache along with its base object as the key.*

The impact of `LiftingFailedException`s will be discussed in Sect. 3.3.3 below. In order to ensure consistency regarding the role-base link binding monotony is required as defined by the following rule.

Rule 4 (binding monotony)
A role class which inherits a `playedBy B1` *declaration from its super-class, may redefine this binding only to bind to a more specific base class, i.e., a sub-class of* `B1`*.*

Binding monotony ensures that no super-type of the concrete role `R1` expects a base object of a more specific type than the declared base class of `R1`. Lifting does not strictly require this rule, since a violation would simply mean that a defined role will never be instantiated. The rule is introduced for the sake of method bindings that might otherwise break.

Note, that binding monotony explicitly admits the case, that the role-base link can be redefined covariantly, which is exploited by the above algorithm. Considering that this link is immutable, covariance introduces no problems here.

Example of smart lifting.
The effect of the above algorithm regarding different role and base hierarchies is demonstrated by the example in Fig. 6.

Given that statically type `B3` is to be lifted to role `R1`, given furthermore, that the dynamic type of the base object is `B5`, the lift method `liftTo$R2` will be called and return an instance of `R4`.

Here are the details: Step 1 using Rule 1 on `B3` and `R1` finds the pairs `(R2,B2)` and `(R3,B3)`. The more general role `R2` selects the lift method to

```
R2 liftTo$R2(B2 b)
```

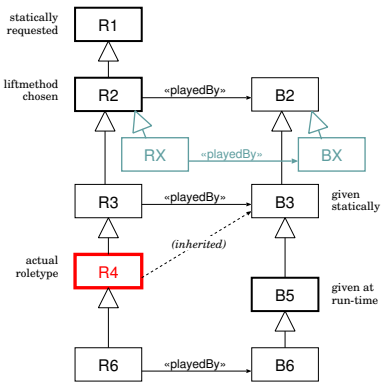

Figure 6: Mapping two hierarchies

Cache lookup (Step 2) uses the cache for `R1`. Let's assume, no instance is recorded yet by the key of `b`. Thus, hierarchy analysis (Step 3) is required for all sub-roles of `R1`. During this analysis, Step 4 discards the role `R3`, because its base-to-role binding is overridden by the direct sub-role `R4`. This leaves the pairs `(R2,B2)`, `(RX,BX)`, `(R4,B3)`, and `(R6,B6)` as candidates. Step 5 does not change the selection in the given setting, since no base class appears more than once. Inspecting the dynamic type in Step 6 finds the classes `BX` and `B6` not to be a super-type of `B5`. Instead, search moves "up" from `B5` until it finds `B3` (which is the most specific from the remaining set `(B2,B3)`) and selects the bound role class `R4`. An instance of `R4` is created, linked to `b` and entered into the cache associated to `R1`, the root role of this hierarchy (Step 7).

`R4` is the most specific type that can result from lifting `b` of dynamic type `B5` to anything below `R1`. Lifting the same object to any other sub-role of `R1` will always yield an instance of type `R4`. This ensures that in a program without statically detected ambiguities (see next section) lifting a given base object to any member of a given role-hierarchy will always yield a role of the same type. Thus, a cache hit within the cache for the common super-role will always find an instance that is conform to the role type required by the lift method.

For teaching the language the following more intuitive rule can be used to describe how the class structure is traversed to find the most suitable role class:

1. Move upwards in the base hierarchy, until you find a base class to which a role from the desired hierarchy is bound.
2. Switch over to the corresponding role class containing the given playedBy binding.
3. Move down in the role hierarchy, as long as no more specific playedBy binding is encountered.

Of course, this rule of thumb assumes bindings without any form of ambiguities. These will be discussed next.

3.3 Corner Cases

Three kinds of problems could occur with lifting: problems related to ambiguities, abstract roles, and exceptions thrown by the hook method `initializeRole`.

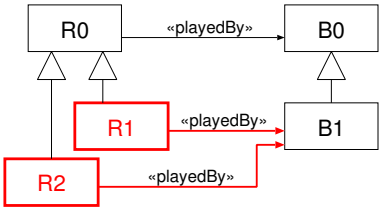

Figure 7: Ambiguous role binding

3.3.1 Levels of Ambiguities

Step 5 identified a situation of *potential ambiguity* which results in a compile-time warning. Note, that this warning is independent of any lifting requests in the program. As an example consider Fig. 7: the pairs `(R1,B1)` and `(R2,B1)` are discarded in the analysis for role `R0` (Step 5). Note, that ambiguity only exists regarding the role type `R0`. A request to lift `B1` to `R1` (or `R2`) is still unambiguous. Removing `R0` altogether eliminates the potential ambiguity.

If the program statically requests lifting of a base class that was discarded in Step 5 because several roles are bound to the same base the program is illegal, signaled by a compile-time error. This situation corresponds to an attempt to lift base `B1` to type `R0` in the figure. It could have been produced by the following method signature:

```
void meth(B1 as R0 o);
```

We call this level of ambiguity *definite ambiguity*.

If a lifting request is encountered at run-time for the dynamic type $\texttt{B}_{dyn}$ that has been discarded in Step 5, this is an *actual ambiguity* which results in a `LiftingFailedException` to be thrown. Regarding Fig. 7 this situation can be produced by a legal method

```
void meth(B0 as R0 o);
```

if this method is invoked with a value of dynamic type `B1`.

Another result of potential ambiguity may be that a role object is retrieved from the cache that is not conform to the expected type (see Step 2). This may happen in a program containing lifting to both classes `R1` and `R2` as in

```
void meth1(B1 as R1 o);
void meth2(B1 as R2 o);
```

If both methods are called in sequence passing the same instance of `B1`, the second invocation will find a *mismatching role* (of type `R1`) in the cache, causing a `WrongRoleException` to be thrown.

3.3.2 Abstract roles

Of course, lifting would also fail if it tried to instantiate an abstract role. For that purpose the notion *relevant role* is coined to denote those roles for which abstractness matters. A simple rule now ensures that lifting will not fail due to an abstract role:

Rule 5 (abstract role)
If any relevant role *is abstract, the enclosing team must be marked abstract, too.*

Surely, all roles identified as irrelevant in step 4 may be abstract without further consequences: they will not be instantiated by lifting.

It may occur frequently, that an abstract base class `B` is bound to an abstract role class `R`, which should normally cause no problem. Proving irrelevance of such role-base pair would however require to show, that *every* instantiable sub-type of `B` has a binding to a non-abstract role. Even if this condition is met by a program at a given point in time, merely adding a new sub-class of `B`, which has no specific role binding, would break safety of the lifting operation. Therefore, all that can be done under the open-world assumption of Java, is to report as a warning, if an abstract but possibly relevant role-base pair is defined by a non-abstract team. It is the responsibility of the developer, to ensure that an instance of the abstract role is not needed. Otherwise, at run-time a `LiftingFailedException` will signal the problem.

Again it is a matter of style to remain within the statically proven safety of lifting or to accept compiler warnings for the risk of a run-time exception.

3.3.3 Exceptions during Lifting

Lifting also fails if the hook method `initializeRole` throws a `LiftingFailedException`. If lifting was invoked due to a callin binding, invocation of the role method is silently aborted and execution continues as if the team was inactive. In all other cases, lifting was triggered by a team or one of its roles. These call sites have a chance to either catch the `LiftingFailedException` or else propagate the exception.

3.4 Lifting and family polymorphism

So far we have pretended, we knew the exact type of the team performing the lifting translation. In order to support polymorphism regarding team instances, we need to show, that a sub-type of the static team type will always yield a compatible role. Due to implicit inheritance of roles (which is not explained here) a sub-team provides at least the same set of lift-methods. Thus static typing is ensured.

Typing of roles in Object Teams applies family polymorphism[5] to ensure consistent typing when overriding role classes (implicit inheritance). This includes the fact that any role type will in fact be anchored by the *instance* of the team containing the role (denoted as `teamInstance.RoleType`). This requires to observe a few constraints for *externalized roles*, i.e., references to a role that are stored outside the team. However, for translation polymorphism anchoring of role types is close to trivial, since lifting only happens *within* a team, thus all role types are anchored by the `this` reference of the enclosing team instance. By this type anchor the dynamic type of the team is already taken into account, which means that the lift method being used always matches the typing requirements of the context that caused the translation.

3.5 Concluding safety considerations

We have given a definition of an algorithm for smart lifting. The problems that might occur while executing this algorithm have been identified. We reconciled flexibility with safety by making explicit different levels of safety. In a program

which compiles with no errors and no warning no runtime errors can occur. Lifting can only be aborted due to a `LiftingFailedException` that is explicitly thrown by an `initializeRole` method.

Programs compiled with a "potential ambiguity" warning can at runtime produce *actual ambiguity* or find a mismatching role in the cache. Both cases are reported by an exception. Similar considerations hold for abstract relevant roles.

Code that may be aware of these lifting problems may catch the exception, while lifting due to a callin binding silently aborts the callin call, since callers in this case have no knowledge about lifting.

Class `org.objectteams.Team`, which is the super-class of all teams has a hook method `liftingExceptionDetected`, which may be overridden in order to react to lifting problems. This method may manually select a suitable role, in which case execution continues normally.

4 METHOD DISPATCH REVISITED

The previous section presented the dynamic selection of role types during the process of lifting. As shown, details of this polymorphic access of role classes are regulated by the *smart lifting* mechanism. This selection of an appropriate role type commonly precedes the sending of a method to the role. This section analyzes the dynamic dispatch for role methods in Object Teams by examining how roles fit into the general setting of object-oriented methods.

Some concepts of Object Teams are capable for serving for aspect-oriented programming. With focus on the aspect-oriented facilities of Object Teams roles are comparable to aspects in AspectJ[35]. From this point of view role methods are designated to play the part of advice. Finally, callin method bindings realize aspect weaving by identifying the targeted joinpoints.
For higher flexibility and reusability it is desirable for role methods to behave like normal methods in object-oriented languages. A matter of particular interest are the polymorphic qualities of such methods. Considering the problems with aspectual polymorphism as pointed out in [4], this is not naturally given. The following discusses some of the posed problems with respect to polymorphic capabilities in AspectJ and contrasts them to our solution in Object Teams.

4.1 Extending aspects

Extending aspects is carried out by deriving and augmenting roles. In AspectJ it is not possible to further extend an already concrete aspect[4]. In contrast, Object Teams allow to extend even concrete roles at multiple levels. This facility results from the fact, that Object Teams does not statically weave advice into the adapted methods. Instead of weaving concrete code only dispatching functionality is inserted into the designated methods, which allows dynamic navigation to the sub-aspect determined by *smart lifting*.

[4]We assume that this results from the restriction of static weaving. While statically weaving, it is not possible to decide which of several possible concrete aspects should be woven.

4.2 Late binding of advice - redefined role methods

In Object Teams every role method may serve as an AspectJ-like *advice*. To this end it has to be bound to a base method in the corresponding base class by an `after` respectively `before` callin binding. Because such role methods are in fact normal Java methods they offer the polymorphic late binding facility as well. So a binding in a super-role may cause an invocation of a redefined method in a sub-role.

The rules for role methods that are to be bound with a `replace` callin binding are slightly different. Such a binding can only be applied to role methods with the `callin` modifier. Inside a `callin` method it is possible to perform an invocation of the overridden (base-) method[5]. This is done by a so called *base-call* which can be seen as the equivalent of a *super*-call for delegation based inheritance. `callin` methods must not be called explicitly, because of the otherwise undefined semantics of the base-call.

Base-calls bring in another form of dynamic dispatch. Here the full set of binding information is dynamically evaluated. A callin method may be bound to several base methods. The selection of the base method to invoke on a given base-call can only be decided at run-time. Statically neither the target nor the method name of this call is known.
Given a role is bound to a base and one of its methods is bound by a replace callin with a base-call, the following applies for subclassing: If a sub-role is bound to a sub-base, the base-call is *re*calling the (maybe redefined) sub-base-method.
After all it is even possible, that a base-call is entirely unbound at certain points in the role hierarchy. This is not a problem, because unbound callin methods are never called.

Consider the example of a tracking aspect for a method `incrXY` for `FigureElement`s, as discussed in [4]. The aspect also applies to the subclass `Point`, where it should behave in a slightly different way. For this purpose the appropriated advice shall be redefined. In AspectJ this encounters a problem, because of the impossibility of overriding an advice (as criticized in [4]) In AspectJ advices are nameless and thus bad candidates for redefinition.

The following listing shows, that Object Team has the ability to face such scenario: callin bound role methods can be overridden in sub-role classes which are bound to more special base classes.

[5] In AspectJ this is achieved by a *proceed* call.

'Advice' overriding for dynamic dispatch:

```
public team class TrackerTeam {
  class FigureTracker playedBy FigureElement {
    public void trackIncr() {
      System.out.println("TrackerTeam: "
        + "Moving a figure element");
    }
    abstract int getX();
    abstract int getY();
    getX -> getX;
    getY -> getY;

    trackIncr <- after incrXY;
  }
  class PointTracker extends FigureTracker
                     playedBy Point
  {
    public void trackIncr() {
      System.out.println("TrackerTeam: "
        + "Moving the point at ("
        +getX() + ", " + getY() + ")");
    }
  }
}
```

The role `PointTracker` inherits from `FigureTracker` and is bound to the more special base class `Point`. In the sub-role the method `trackIncr` is redefined as needed. Calling `incrXY()` on a `FigureElement` will now cause an invocation of `FigureTracker.trackIncr`, while calling the same method on a `Point` object (or any subclass thereof), will lead to an execution of the overridden `trackIncr` in `PointTracker`, because *smart lifting* lifts each `Point` to a `PointTracker` role.

4.3 Inheritance of method bindings

In Object Teams method bindings are inherited to sub-roles. If a sub-role redefines a method, the binding will now apply to the new version of the method. On the other hand method bindings can be omitted and deferred to sub-roles. This allows the flexible definition of an *advice* without specifying the point to which it shall be applied.
In AspectJ this is supported by the possibility of abstract pointcut designators which can be used in an advice definition and made concrete in a derived aspect. Furthermore it is possible to override an already concrete pointcut designator. Unfortunately this holds the danger of breaking inherited advice code that makes assumptions on the return type of the pointcut designator, which are not fulfilled by the redefined version, as reported in [4]. The problem is the incomplete signature definition of pointcut designators: the return type is not contained in its declaration.

4.4 Base-side aspect inheritance

The previous examinations focused on the role-side of aspect inheritance. The remaining of this section will take a look at the polymorphic characteristics regarding the hierarchy of base classes.

4.4.1 Advicing inherited methods

A role may be bound to a base class in order to adapt one of its methods. A question arises if the designated method is not defined in the bound base class itself but inherited from a superclass. In this case there is no code located in the subclass where to weave in the advice. Like an unbound base-call such an operation initially has no target. To allow the adaption of inherited methods (which does not affect the super-classes) we choose the option of generating a redefinition of the relevant method with nothing but a call to the original version. Now weaving into the redefined version is straight forward.

4.5 Orthogonality

We have demonstrated how aspect-oriented programming with role methods and callin bindings avoids problems that have been reported with respect to AspectJ. Role methods and method bindings behave sound in the context of inheritance, overriding and method dispatch. These are hints that translation polymorphism is sufficiently orthogonal to subtype polymorphism. In the next section we will elaborate on these issues as two dimensions of dispatch and polymorphism.

5 Two Dimensional Dispatch

In the introduction we discussed dynamic dispatch as the mechanism for combining shared and un-shared parts of an implementation. The Template Method pattern served as an example where the shared template method must implicitly know about the invocation context — the original call target — in order to dispatch calls of a hook method to that original call target. This invocation context is not visible in the source code of the template method, but at run-time the dynamic type of `self` carries this information.

Instance dispatch.
We will now interpret lifting in analogy to method dispatch. Consider a method call on some base instance which is potentially affected by some callin binding in the system. Before any method is actually invoked, lifting may perform dynamic *instance dispatch*. This means the current context is investigated in order to find the most suitable call target.

Now what makes the context for instance dispatch? Here lies the major difference between both kinds of dispatch. Method dispatch only needs one type for lookup: the dynamic type of the call target. Lifting requires three pieces of information: a base instance, a team instance and a required role type.

- The *base instance* is given at the call site.
- Candidates for the *required role type* are all role classes that have a callin binding for the particular base method being called.

- *Team instances* are retrieved from the global state of the program as the set of all currently active teams.

Of course, only matching pairs of role types and team instances are considered, i.e., the role type must be a role of the particular team. Given that more than one team may be active affecting the same base method by callin, teams are internally kept in a stack, where the most recently activated team has highest priority for method interception.

Contextual lookup.
While method dispatch relies on local context (the dynamic type of the call target) only, instance dispatch depends on the *global state of the application* — the stack of active team instances. Thus it is possible to implement different kinds of program *modes* in a modular way.

From a technical point of view, method calls now consist of a sequence of lookups regarding callin bindings, active teams, roles within those teams before finally using normal method dispatch to find the most suitable role method. Conceptually, any program state could also be seen as a *stack of layers*, each layer being defined by an active team instance containing a set of role instances, which decorate underlying base instances. Some of these role instances may exist only *virtually*, which means they have not been created yet, but will transparently be created by lifting prior to the first usage. Any virtually existent role instance may contain overrides for a given base method, where these overrides are specified by callin method bindings. When a base method is called, all these overrides are considered from the top to the bottom of the stack of layers (teams).

Also some dispatching is involved, when invoking a method that is bound by *callout*. However, the lookups involved in callout bindings are fairly trivial, because only lowering and normal method dispatch are combined here.

Overcoming Object Schizophrenia.
Considering instance dispatch as just another technique that *transparently combines* shared and un-shared parts (here: instances), it becomes easier to think of sets of atomic instances as one *compound object*, since explicit navigation between atoms is not needed any more. Just like subtype polymorphism allows to view an instance under different guises (static types) translation polymorphism *pretends* that a role and its base are just different guises of the same entity. Within a team, it is usually sufficient to see role instances as full representatives for their base instance. Any relevant base property should be accessed through the role and be bound by callout. In this perspective a role instance and its base are considered *the same instance*.

Control over role instances is given by creating and activating team instances. Each team instances defines a self-contained view of the application. For more intricate cases, fine grained control over team membership is given through API functions of class `org.objectteams.Team`. While these mechanisms fall beyond the scope of this paper, it is important to notice that role instances can be created and attached to base instances at different levels of granularity, which pertains the full flexibility of the dynamic approach, using instance composition and delegation.

With these two perspectives of transparency *and* control we balanced the forces that in other approaches lead to all the problems of "object schizophrenia".

Practical value and generality.
We have presented a solution that adds flexibility to class based approaches and makes instance based composition and delegation more convenient to use. Ongoing practical experiments are encouraging as they demonstrate the feasibility of the approach. It appears that designing a system with Object Teams not only improves modularity but first of all leads to a better understanding of conceptual structures and their interdependencies. Of course, such claims need further evaluation in controlled practical case studies.

We are confident, that we have come to a solution that besides being useful in practise is also general enough to cover a large set of structures, problems, requirements etc. The reason for believing so is the symmetry of method dispatch and instance dispatch, of static inheritance and delegation. The following table contrasts both dimensions to each other, demonstrating how well they complement each other.

Method dispatch	**Instance dispatch**
Sharing of *method* implementations by static inheritance	**Sharing** of *instances* by role-base links and delegation
Dynamic type of call target as *local* **context**	Stack of active team instances as *global* **context**
Substitutability by *subtype* polymorphism	**Substitutability** by *translation* polymorphism

6 Related Work

In the direct context of Object Teams, the following approaches apply some sort of lifting or "wrapper recycling": PIROL [11, 13], PCA [22], their implementation JADE [10], LAC [12] (a prototype for Object Teams) and Caesar [21]. Aspectual Collaborations [18] — although they evolved from the same source: Aspectual Components [17] — appear to statically weave code instead of our instance based composition.

Some programming languages do support delegation and role objects including Self [32], Lava [16] and Chameleon [8]. In these models, a caller mostly needs to be aware of specific roles in order to use role behavior. So called "constituent methods" in the Chameleon model are similar to our callin-bound methods and provide for some kind of dynamic AOP. Several dynamic AOP systems [24, 14, 28] maintain aspects as runtime instances that can be attached, even chained and detached. With all systems mentioned, the lookup of attached aspects only retrieves instances that have been attached explicitly. This requires explicit knowledge about all instances that should be affected. The lookup only concerns the target object of a method call. The lifting operation of Object Teams may create roles on demand and furthermore applies to callin parameters and callout results, thus mapping graphs of base objects to graphs of role objects, much in the spirit of AP[19], AP&PC[20] and related models.

Recently, Truyen and Jørgensen developed LasagneJ [15] realizing their general model Lasagne [31]. In this language, wrappers can be defined similar to our roles. Wrappers can furthermore be grouped to extensions, denoted as

packages. While extensions share the idea of grouping collaborating roles, they cannot be instantiated and are thus limited when compared to teams. LasagneJ features so-called "constructive downcasts", which have similar semantics as lifting in Object Teams. LasagneJ also applies on-demand wrapping while in the control flow of a wrapper. We interpret this as a variant of contextual dispatch: LasagneJ passes an invocation context along each control flow which determines the extensions whose wrappers should be effective.

In LasagneJ dynamic selection of a suitable wrapper class is explicitly avoided, by allowing only one wrapper class for each base class (including sub-classes) per extension. In [15] the authors mention this restriction as an area of further work. We have shown, to which extent smart lifting can do away with these restrictions. Also the static nature of extensions (i.e., lack of instantiability) restricts the (need for) wrapper selection. Finally, method bindings are not explicit as in Object Teams, but established by name equality. Interestingly, LasagneJ allows to select between two flavors of overriding — `consult` vs. `delegate` — by adding a modifier to the overriding method.

The language C#[1] allows to provide custom conversions, which could help in implementing smart lifting. Lifting would be considered an explicit conversion, lowering an implicit conversion. The application of lifting would then require a cast operator while lowering would look like a polymorphic assignment. This analogy confirms us in our goal to generalize over sub-type polymorphism and translation polymorphism. It is not clear, how the Team context would be available to custom conversions, since these are static methods that have access only to their explicit parameter. Of course, C# lacks the concepts of Teams and roles, and thus translation polymorphism can only be mimicked partially, requiring a lot of hand-coding.

The programming language Sather allows to introduce super-types a-posteriori [29]. However, it lacks support to do so at run-time. In an outline, how Subject Oriented Programming could become more dynamic, Harrison et al. [9] use the notion of dynamic super-types. They combine concepts of viewpoints and dynamic re-typing in order to allow per-instance subject activation. Unfortunately, the ideas from this work have never seen a realization. It's spirit is, however, closely related to Object Teams. The Hyper/J[30, 23] language was the first to introduce on-demand re-modularization. By extracting elements from existing packages and assigning them to concerns and dimensions Hyper/J allows to integrate modules with mismatching internal structures. Hyper/J does this in a purely static way by composing new classes at compile time. Object Teams maintain separate objects at run-time which by the lifting/lowering operations are related to each other.

Lifting also implements environmental polymorphism[7], as it depends on the set of currently active Teams. We are currently investigating how Team activation and lifting can be used to implement `cflow`-like aspect binding while adhering to the requirements of environmental advice[4].

Harrison et al [9] also compare the issues at hand to schema evolution in object-oriented databases. One difficulty with dynamic re-typing is the question of reversibility. In [13] we have presented a mechanism for run-time specialization of objects, called *upgrading*. Upgrading uses sub-type polymorphism to ensure type safety. Once a link to the specialized object exists, the object may never drop its acquired type. Object Teams facilitate reversing the adaptation by two different means. First, the adaptation is not intrusive on the object,

since roles are added as distinct entities. Secondly, a Team can be used as a border for strictly encapsulating all its roles. By not letting a link to a role escape from the enclosing Team instance, it can be assured that a role can safely be destroyed, once the Team ceases being.

7 Conclusion

We have presented the concepts and mechanisms of translation polymorphism as realized in the ObjectTeams/Java language. By this concept we combine findings of research regarding inheritance, delegation, collaborations, aspects and views.

First, team activation provides a means to abstract from the activation of elementary objects. This requires role objects to be created on-demand. In order to preserve state of role objects, a cache of roles must be used per team or role hierarchy.

Second, by applying translation polymorphism to call targets, parameters and results we achieve a mapping between graphs of objects, not just isolated objects.

Third, the fact, that translation polymorphism is mostly transparent in a program helps to decouple aspects from affected base classes. Both worlds are implemented completely in their own ontology, only declarative bindings define the relation of both worlds.

Fourth, if inheritance on the sides of bases and roles should be supported, smart lifting is inevitable. Smart lifting helps to attach an aspect with its own structure to a base module with a different structure. Smart lifting ensures that always the most suitable role will be created or retrieved.

In a program that translates without warnings, lifting is a perfectly sound and safe operation. If more flexibility is needed, a compiler warning and two kinds of exceptions alert the developer and ensure that the program will behave deterministically.

Fifth, regular role methods supersede advice in aspect languages. By imposing only minimal restrictions on methods to be bound by callin, Object Teams avoid problems relating to aspect polymorphism as reported with respect to AspectJ.

Thus, translation polymorphism subsumes a number of proposed concepts by a single new mechanism: dynamic instance dispatch.

7.1 Future work

In order to fully support a flexible life-cycle of roles, we would like to find means for canceling existing roles. Currently, we can only dispose a team as a whole. If no externalized roles are used, all existing roles of a team to be disposed can safely be disposed, too. Disposing or canceling single roles is difficult, because this would invalidate all links to the given role, and to-date we see no mechanism that would easily allow dynamic role replacement.

The Object Teams model and its realization as ObjectTeams/Java are fairly stable by now. The next level of research, which has already started, is the evaluation of the language in real industrial software development.

References

[1] C# language specification, 2. edition. Standard ECMA-334, 2002.

[2] M. Aksit, editor. *Software Architecture and Component Technology: State of the Art in Research and Practice*. Kluwer Academic Publishers, 2001.

[3] *Proc. of 2nd International Conference on Aspect Oriented Software Development*, Boston, USA, 2003. ACM Press.

[4] E. Ernst and D. Lorenz. Aspects and polymorphism in AspectJ. In AOSD'03 [3], pages 150–157.

[5] Erik Ernst. Family polymorphism. In *Proc. of ECOOP'01*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[7] J. Gil and D. Lorenz. Environmental acquistion — a new inheritance like abstraction mechanism. In *Proc. of OOPSLA'96*, pages 214–231. ACM, 1996.

[8] Kasper B. Graversen and Johannes Beyer. Chameleon, August 2002. Masters thesis. IT-University of Copenhagen.

[9] W. Harrison, H. Kilov, H. Ossher, and I. Simmonds. From dynamic supertypes to subjects: a natural way to specify and develop systems. Systems Journal 35(2), IBM Systems Journal, 1996.

[10] Michael Haupt. JADE: Entwurf und Implementierung eines Sprachkonstruktes zur dynamischen Komposition wiederverwendbarer Softwaremodule als Erweiterung der Programmiersprache Java. Diploma thesis, Universität-Gesamthochschule Siegen, `www.st.informatik.tu-darmstadt.de/projects/JADE/`, December 2000.

[11] S. Herrmann and M. Mezini. PIROL: A case study for multidimensional separation of concerns in software engineering environments. In *Proc. of OOPSLA 2000*. ACM, 2000.

[12] S. Herrmann and M. Mezini. Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23nd ICSE*, 2001.

[13] S. Herrmann and M. Mezini. Connectors for bridging mismatches between the components of a software engineering environment. *IEE Proceedings – Software*, 2001.

[14] Robert Hirschfeld. AspectS - aspect-oriented programming with squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

[15] B. Jørgensen and E Truyen. Evolution of collective object behavior in presence of simultaneous client-specific view. In *Proceedings of the 9th international Conference on Object-Oriented Information OOIS'03*, volume 2817 of *LNCS*, pages 18–32. Springer Verlag, 2003.

[16] Günter Kniesel. *Dynamic Object-Based Inheritance with Subtyping*. PhD thesis, Universität Bonn, 2000.

[17] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, April 1999.

[18] K. Lieberherr, D. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, sept 2003.

[19] Karl Lieberherr. *Adaptive Programming: the Demeter Method*. PWS Publishing Company, 1996.

[20] M. Mezini and K. Lieberherr. Adaptive Plug-and-Play Components for evolutionary software development. In *Proc. OOPSLA'98*, volume 33 of *SIGPLAN Notices*, pages 97–116. ACM, 1998.

[21] M. Mezini and K. Ostermann. Conquering aspects with caesar. In AOSD'03 [3].

[22] M. Mezini, L. Seiter, and K. Lieberherr. *Software Architecture and Component Technology: State of the Art in Research and Practice*, chapter Component Integration with Pluggable Composite Adapters. In Aksit [2], 2001.

[23] H. Ossher and P. Tarr. *Software Architecture and Component Technology: State of the Art in Research and Practice*, chapter Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Aksit [2], 2001.

[24] R. Pawlak, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible solution for aspect-oriented programming in java. In *Proc. Reflection 2001*, number 2192 in LNCS, pages 1–24. Springer Verlag, 2001.

[25] IBM Research. Subject-oriented programming and design patterns. `http://www.research.ibm.com/sop/sopcpats.htm` (last accessed March 2004), 1997.

[26] K. C. Sekharaiah and D. J. Ram. Object schizophrenia problem in modeling is-role-of inheritance. In *Inheritance Workshop at ECOOP 2002*, 2002.

[27] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The Treaty of Orlando. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989.

[28] D. Suvee, W Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In AOSD'03 [3], pages 21–29.

[29] C. Szyperski, S. Omohundro, and S. Murer. Engineering a programming language — the type and class system of Sather. In *Proc. International Conference on Programming Languages and System Architectures*, volume 782 of *LNCS*. Springer-Verlag, March 1994.

[30] P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*. IBM Corporation, 2000.

[31] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. Nørregaard Jørgensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the 23rd ICSE*, 2001.

[32] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Proc. of OOPSLA'87*, 1987.

[33] M. Veit and S. Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In AOSD'03 [3].

[34] Peter Wegner. Dimensions of object-based language design. In *Proc. of OOPSLA'87*, 1987.

[35] Xerox Corporation. *AspectJ Programming Guide*. available from `http://eclipse.org/aspectj`.