Object Teams: Improving Modularity for Crosscutting Collaborations

Stephan Herrmann

Technical University Berlin
stephan@cs.tu-berlin.de

Abstract. In this paper, we investigate whether module concepts for capturing multi-object collaborations can be effectively used to implement crosscutting concerns in reusable, independently developed modules for a-posteriori integration into existing systems. A new kind of collaboration module, called Object Teams, is proposed which combines the best features of existing approaches, further enhances them with concepts for expressing crosscutting relations between independent collaborations, and facilitates *a-posteriori integration* of such collaborations into existing systems.

1 Introduction

Several proposals for modules that are larger than classes have emerged over the last decade. In addition to components as defined by standard component models such as the CORBA Component Model and Enterprise Java Beans, which are beyond the scope of this paper, several other module proposals have been made for supporting so-called *collaboration–based design* [4, 19, 16, 2, 17, 12, 3, 15].

Approaches of this kind build on the insight that the design of object-oriented applications can be organized in two dimensions: using structural abstractions (object types) and collaborations in which these abstractions are involved (see Fig. 1).

Generally speaking, an object is involved in several collaborations, a single collaboration spanning several objects, and each participating object playing a particular role within the collaboration. While the definition of an object is encapsulated within a class, standard object-oriented languages do not provide an appropriate module construct for capturing collaborations. Approaches to implementation support for collaboration–based design seek to fill this gap.

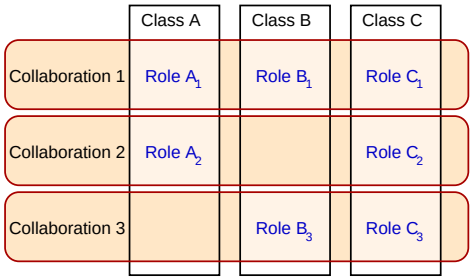

Fig. 1. Dimensions of classes and collaborations

With collaborations spanning several classes, it is evident that collaboration modules are *crosscutting* with respect to the basic modular structure of the system — the one dictated by the structure of data. This observation has motivated us to investigate the usefulness of techniques for *composing* modules, which were proposed in the context of

aspect–oriented programming (AOP) [10]. Two concerns are considered crosscutting, if given a modular structure for capturing one concern, the other cannot be encapsulated in a module, but rather cuts across several modules as introduced by the first concern. Our interest is in supporting independent development of reusable crosscutting concerns that are subsequently integrated into an existing system. We argue that, while providing good starting points, none of the existing module constructs fully meets our needs. Each of these proposals solves a different subset of problems on the road to reusable collaboration modules. Since combining different language extensions is not normally an option, we seek to develop an integrated programming language that, in a uniform programming model, provides a powerful toolkit for a wide range of issues relating to the definition and composition of collaboration modules.

Using an example scenario, we illustrate the features that are needed and propose a new kind of collaboration module, called *Object Teams*, which combines the best features of existing approaches and further enhances them with concepts for expressing crosscutting relations between independent collaborations and for *a-posteriori integration* of such collaborations into existing systems.

Object Teams can be related to more than a handful of recent approaches. Presenting Object Teams as an improvement over any particular approach would therefore mean randomly choosing a starting point for this presentation. To avoid expecting of the reader detailed knowledge of one particular approach, we have chosen a neutral presentation which introduces step by step the concepts that comprise the Object Teams model (Sect. 2). Sect. 2.3 gives a preliminary comparison with approaches for collaboration modules. Sect. 3 discusses related work from the field of AOP. Sect. 4 gives a summary, looks at the current status of development and indicates areas for future work.

2 The Object Teams Model

Throughout this paper, we use a minimal model of a flight booking system. A core model is assumed to exist containing the classes `Passenger`, `Flight`, and `Segment`. Segments are the constituent parts of a flight. A passenger can only book complete flights, though different flights may share common segments (cf. Fig. 2). Step by step, we introduce into the model a bonus programme by which passengers can collect credits when booking flights. We restrict discussion to the collection of credits, not covering how collected credits are later spent. It is the unanticipated, non–invasive and modular introduction of this bonus programme which we use to elaborate the concepts of Object Teams. An overall picture of the intended system is sketched in Fig. 3, which uses our UML extension UFA [8].

```
class Passenger {
   void book (Flight flight) { ... }
}
class Flight {
   Segment[] segments;
   /* ... methods omitted ... */
}
class Segment {
   void book (Passenger pass) { ... };
   int getMiles () { ... };
}
```

Fig. 2. Features in the flight booking package

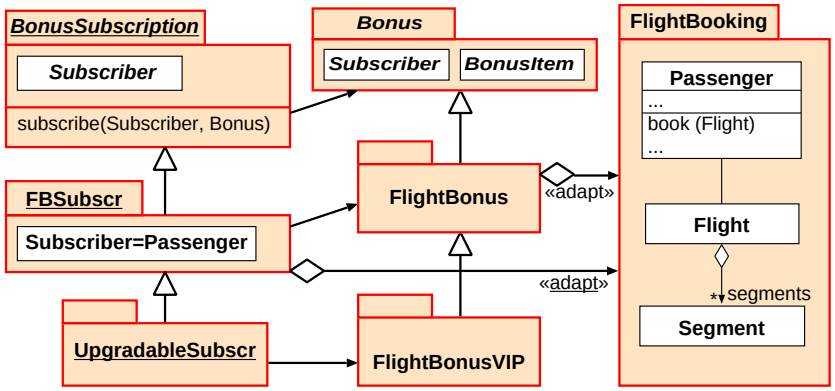

Fig. 3. Module structure overview of the intended system

2.1 Collaborations of confined objects

The key concept of Object Teams is a kind of module that combines properties of classes and packages. An *Object Team* is an instantiable aggregation of confined objects called *roles*. In our bonus example, a Team comprises objects of the classes `Subscriber` (persons participating in the bonus programme) and `BonusItem` (arbitrary items, whose "acquisition" is rewarded by a credit to the subscriber). A group of objects is reified into a compound object — called *Team instance* — which represents the Team and contains all participating roles.

Besides aggregating role objects, a Team instance may also have features of its own. In our example, we use the Team instance to accumulate credits earned due to a bonus item in the Team–level attribute `accumulator`. At the same level, the methods `put` and `take` encapsulate access to this attribute. Looking at Fig. 4, lines 2–10 define Team–level features. Lines 14 and 24 show access to the Team instance as the outer instance of contained roles. This is equivalent to Java's style of accessing an outer instance from an instance of an inner class.

```
1  abstract team class Bonus {
2      int accumulator = 0;
3      void put (int credits) {
4          accumulator += credits;
5      }
6      int take () {
7          int tmp = accumulator;
8          accumulator = 0;
9          return tmp;
10     }
11     class Subscriber {                    // role class
12         int collectedCredits = 0;
13         void collect() {
14             collectedCredits += Bonus.this.take();
15         }
16     }
17     abstract class BonusItem {     // role class
18         abstract int getRawCredit ();
19         int calculateCredit () {
20             // implement default policy:
21             return getRawCredit();
22         }
23         void earnCredit () {
24             Bonus.this.put(calculateCredit());
25         }
26     }
27 }
```

Fig. 4. Team definition for a bonus programme

Inheritance is introduced to our example by defining a hierarchy of Teams `Bonus`, `FlightBonus` and `FlightBonusVIP`. The definition from Fig. 4 is kept very general. It contains no details of flight booking. When moving from this abstract Team to the concrete Team `FlightBonus`, the method `calculateCredit` may, for example, be overridden to include some rounding operation according to the airline's policy:[1]

```
team class FlightBonus extends Bonus {
  // class Subscriber is reused without modification
  class BonusItem {                    // implicitly inherits Bonus.BonusItem
    int calculateCredit () {
      int credit = tsuper.calculateCredit();
      return ((credit + 999) / 1000) * 1000;
    }
  }
}
```

A Team specialization may refine any of the contained role classes. We do not add special syntax for this, establishing the role inheritance simply by name equality. Such implicit inheritance follows its own specific rules, which are not fully detailed in this paper. A super call along implicit inheritance is denoted as `tsuper()`, thus avoiding ambiguity with respect to regular, explicit inheritance.

Another Team, `FlightBonusVIP` (of which no details are given here), could also be defined to model an advanced bonus programme for specially qualified members. `FlightBonus` and `FlightBonusVIP` can be modeled as specializations of the more general `Bonus` Team with consistent refinement of all involved classes. Now polymorphism applies to instances of `Bonus`, `FlightBonus` and `FlightBonusVIP`, but not to their contained role objects. Consistent refinement of role types, which may refer to each other, may introduce covariance. By prohibiting substitution of roles between different Teams types we ensure that these covariant redefinitions do not introduce a hole in the type system.

Confined and externalized role objects Generally, a role instance is confined to its enclosing Team, i.e., role objects cannot escape their enclosing scope, which is already ensured by the type system of Object Teams. There is one exception to this rule: a variable outside the Team definition may use an instance–bound type declaration as in

```
final Bonus bonusInstance = new FlightBonus(...);
bonusInstance.BonusItem item = bonusInstance.getCurrentItem();
...
bonusInstance.deleteItem(item);
```

Here the variable `item` is declared to be of type `BonusItem` and belonging to the Team `bonusInstance`. This instance defines the Team and has its own understanding of the type `BonusItem`.

Only by declaring `bonusInstance` *final*, i.e., immutable, can it be statically ensured that `item` is only passed back to the same Team from which it originates. This is

[1] The reader should not be confused by the still abstract method `getRawCredit`. Implementation for this method will be given — using a new mechanism — in Sect. 2.2.

the only way a role can be passed outside the Team. The application code using such an "externalized" role object must be aware of Teams and declare variables such that roles are only used together with members of the same Team, or the Team instance itself. Fig. 5 presents an example in which this technique is used to implement an action class that can be used by a graphical user interface to clear off the collected credits of a given subscriber object that is currently displayed at the user interface.

```
class ClearAction extends Action {
   final FlightBonus context;
   context.Subscriber subscriber;
   ClearAction(final FlightBonus c,
                  c.Subscriber s) {
      context = c;
      subscriber = s;
   }
   void actionPerformed () {
      subscriber.clearCredits();
   }
}
```

Fig. 5. Externalizing a role for use in the GUI

2.2 Role and base objects and their relation

After focusing on the consistency of objects within a Team, i.e., the graph formed by their links, and on Team refinement, we will now present how a Team is composed with an existing package of the application. This introduces a distinction of role and base objects. Concerning their relationship two obligations have to be met. Firstly, we must ensure that for each role a base object is always present. A `BonusItem` role object cannot exist without a matching `Segment` base object. Secondly, we take a new view of substitutability between these two categories of objects. Should it be possible to supply a `BonusItem` object where a `Segment` is required, or vice versa?

Dynamic composition To ensure flexible composition, we bind a Team to a base package using object–based inheritance (also known as delegation). The role–base relationship is declared using the keyword `playedBy` (cf. Fig. 6, line 9). We merge the classes `Segment` and `BonusItem` using both types of inheritance: we create a subclass of `BonusItem` — which owing to implicit inheritance does not require a new class name — that is also a *role* of `Segment` by virtue of a role-base link between instances of both classes. This link ensures dynamism because it can be established and removed at runtime. Flexible multiplicity is also enabled because any number of roles may refer to the same base object.

While object–based inheritance provides desirable flexibility, we have to be very careful not to break the type system. What we want is a clean interface between a role and its base object plus a fulfilment guarantee regarding this interface, i.e., we must declare which methods a role will acquire from its base object, and we need the guarantee that an instance providing this interface will actually be associated.

Declarative completeness and a-posteriori integration The straightforward approach to declaring an interface between a role and its base is to use an explicit typed link. Object Teams hide this link from client programs. Only the `playedBy` clause gives information on the relation between a role class and its base class (cf. Fig. 6, line 9). It is the responsibility of the runtime system to ensure that each role object always has a valid base object linked to it.

Both classes are further decoupled by making explicit the *expected interface* of a Team. We require all methods that are expected within a Team to be declared as abstract methods. When refining an abstract Team, we provide two options for realizing an abstract method: it may either be implemented in–place, i.e., by a regular method within the refined role class, or by declarative mapping to an existing method of a base class. The second option depends on a `playedBy` declaration, which restricts the type of the associated base object. Fig. 6 gives an example of such a binding.

```
1  abstract team class Bonus {
2      class BonusItem {
3          abstract int getRawCredit ();
4          /* ... other methods, which may  */
5          /*     use getRawCredit ...   */
6      }
7  }
8  team class FlightBonus extends Bonus {
9      class BonusItem playedBy Segment {
10         getRawCredit → getMiles;
11         /* ... other details omitted ... */
12     }
13 }
```

Fig. 6. A role binding

Lines 9–10 declare that calls to `getRawCredit` should be delegated to the base object of type `Segment` using the method `getMiles` of the latter class. Such bindings are the key to a-posteriori integration, because an abstract Team can now be partially implemented using its own ontology. By refining the Team, the missing details are filled in using the names of base classes, which provide the required functionality.

Mismatches, which can be bridged using these techniques, cover different inheritance structures (see [6] for a discussion in a similar setting), class names, method names, and even parameter lists.

Note that in this model each method acquired by object–based inheritance must be declared in a binding clause. This prevents accidental name clashes which would otherwise be likely to occur because two classes that were developed independently are combined. On the other hand, such explicit method acquisition is practical because of the succinct style of binding declarations. In order to distinguish this kind of method binding from another technique to be introduced in Sect. 2.4, we coin the notion *callout binding*, which is used to signify that a role object binds a method, which is not available locally, by "calling out" to the associated base object.

Three levels of type safety can be identified. A Team can be type–checked statically in complete isolation. The refining Team that defines bindings to base classes can be type–checked using the interfaces of both packages to be integrated — more precisely: the expected interface of the abstract Team and the provided interface of the base package. The third level adds the dynamic dimension: it must be ensured that each role object is, at all times, bound to a proper base object of the type declared after the `playedBy` keyword. We elaborate on this in the following paragraphs.

Substitutability by implicit translation Objects of types `Passenger`, `Flight` and `Segment` along with other classes not mentioned here may also participate in other Teams, like the planning and booking of travel packages. This should illustrate that Teams operate on their own very specific, selective view of the world, with different focuses on both the type and instance level. Research on programming views has taught us to think of different views by means of mapping functions. We introduce a new kind

of substitutability by implicit translation. Our translations are defined according to [12] and are called *lifting* and *lowering*.

Lowering is already used when forwarding a `getMiles` method call (invoked as `getRawCredit`) from a `BonusItem` to its `Segment` base object. Lowering simply means stripping a role off its base. The method is then executed without knowledge of the role. Note that this contrasts with normal object–based inheritance, where it is crucial to maintain the original self–reference within delegated calls. We will address the issue of overriding along role–base links in Sect. 2.4.

Lifting is required for example, whenever a `Segment` instance is to appear in a `FlightBonus` Team. For this purpose, the `Segment` has to be wrapped or decorated with a `BonusItem` role. It is important that the same `Segment` is always represented by the same role instance within a given Team. The translation using a cache of already known role objects is under the implicit control of each Team instance.

Both translations are performed transparently by the runtime system. Subtype polymorphism and translation polymorphism can thus be exploited by the programmer in a similar way. It is the runtime system that takes care of the difference. Substitutability of role objects for an expected base object is not surprising. This is what polymorphism in languages with object–based inheritance is all about. The reverse can only be achieved by the lifting translation. We say a base object is *fit* for playing a given role if a corresponding `playedBy` relation is defined for the given role and base classes.

It is the lifting translation that the Object Teams runtime uses to ensure that roles exist only with a valid base object. Once created, a role object may not alter the link to its base object. Lowering, on the other hand, ensures that no role object escapes its enclosing Team instance, except for explicitly externalized roles.

To summarize the role–base relationship: implicit lifting and lowering are the only mechanisms that have access to the role–base link. Client code may never explicitly use or modify this link. These mechanisms in concert ensure that each role always has a consistent base object.

2.3 Comparing proposals for collaboration modules

The features presented so far roughly correspond to the capabilities of some approaches for composition and refinement of collaborations. A comprehensive discussion can be found in [15]. A comparison in terms of the major issues discussed so far is given in the table below. Object Teams fulfil all listed criteria.

Object confinement is borrowed from the concept of family polymorphism [3], but Object Teams provide the option of more flexibility. While both approaches confine inner objects to the scope of their outer instance, different Teams may *partially share* objects by first lowering to a common part and then lifting to a different role. A member of a family is “born” into its context for its entire lifetime. By contrast, Team membership is a dynamic issue. Each role wraps an intrinsic object which is independent of any Team.

Many previous proposals employing role objects suffer from the explicit schizophrenia between a role and its base object. Object identity is quite a dubious concept in such a setting. For Object Teams, this is a non–issue because comparing a role and a base object is meaningless: both exist in disjoint worlds.

	mixin layers	dynamic view connectors	pluggable composite adapters	family polymorphism	delegation layers
reference	[17]	[6]	[13]	[3]	[15]
collaboration refinement	×	×	×	×	×
declarative method binding	–	×	–	–	–
confined objects	–	–	–	×	×
object–based inheritance	–	×	×	×	×
dynamic composition	–	×	×	×	×
multiple instance binding	–	×	×	–	×
lifting/lowering	–	×	×	–	×

In contrast to some notable similarities as shown in the above table, the features presented in the following sections are not supported by any of the cited approaches.

2.4 Callin binding: weaving into existing code

Introducing implicit lowering for base calls made from a role object implies that we apply a forwarding mechanism, not true delegation with late binding of self. This precludes overriding base methods within the role. While it is desirable to override a base method within a role class, we consider overriding based on name equality inappropriate for the role–base relationship, because the independent development of role and base modules might easily produce methods with the same names with no intention of overriding. Instead, we maintain separation of name spaces and support explicit overriding as a special case of advice weaving in the style of aspect–oriented programming.

In order to emphasize the difference from the callout binding style introduced above, the new style of binding is called *callin* and denoted by a reverse arrow ←, because here the bound role instructs a base class to "call into" the role. Both binding styles, callout and callin, are to be seen from the perspective of a role object that somehow interacts with an external base object.

To enable a role class to override base methods, we apply the style of the aspect–oriented language AspectJ. We regard this kind of overriding as the insertion of new code into existing classes. In the tradition of CLOS and AspectJ, we allow new code to be inserted either *before* or *after* the original method. Replacing the original version is also possible (in AspectJ: "around"). In our example, an `after` callin binding is used to modify the method `book` of class `Segment` as shown in Fig. 7, line 3. As the effect of this binding, each invocation of `book` from `Segment` entails a call to `earnCredit` on an instance obtained from the current call target by lifting to class `BonusItem` of Team `FlightBonus`.

Only the replace style, which is the default if no keyword is given, has explicit control over executing the original version. Thinking of the role relation as an inheritance relation, invoking the original version is equivalent to a *super* call. We thus employ a similar syntax, using *base* instead of super. All in all, Object Teams support three different styles of inheritance: regular inheritance using `extends`, implicit role inheritance (by name), and the role–base relationship. Unlike multiple inheritance, these

three mechanisms avoid ambiguities by precedence and explicit declaration of overriding. Each style has its own keyword for invoking a parent version: `super`, `tsuper` and `base`.

Because base calls are limited to methods bound by a replace callin, these methods have to be marked "callin" before allowing the use of a base call. Fig. 7 shows a callin method (lines 6–9) and its binding (line 10). This is a variant of the method `collect` from Fig. 4, which could instead be bound as `after` callin with the same effect.

```
1  team class FlightBonus extends Bonus {
2    class BonusItem playedBy Segment {
3      earnCredit ← after book
4    }
5    class Subscriber playedBy Passenger {
6      callin void collect () {
7        base.collect(); // invoke original version
8        collectedCredits += Bonus.this.take();
9      }
10     collect ← book; // overriding (replace)
11   }
12 }
```

Fig. 7. Binding Bonus to the flight booking system

While callout bindings do not affect an application unless the Team is explicitly invoked, callin bindings offer the chance to insert triggers into the application, where Team–specific behavior should be added. In our example, we were lucky that only two such triggers suffice for weaving the Team into the base: each class `Passenger` and `Segment` has a single method responsible for any booking action. If such a unique hook cannot be found, quantification in the style of AOP can be used to denote as a join point a set of methods into which a given advice is to be woven. It is beyond the scope of this paper to discuss the expressiveness of declarations of join points, but preliminary comparisons suggest that in Object Teams a much leaner sublanguage for join–point definition suffices as compared to AspectJ, without sacrificing any of our goals.

2.5 Adjusting signatures

The last level of adaptation covered by Object Teams is parameter lists. If a-posteriori integration is to be fully supported, it must be possible to bind methods that have different signatures. To this end our model allows the full signature with parameter names to be included in the binding specification. Each method binding may then be followed by a list of parameter mappings as in

```
void abstrMeth(T1 a1, T2 a2) → void baseMeth1(T3 a3)
                    with { a2 → a3 };
        void callinMeth(T4 a4) ← void baseMeth2(T5 a5, T6 a6)
                    with { a4 ← a6 };
```

Note that both kinds of binding may ignore provided parameters (both times appearing on the origin side of the arrow). In the case of a callin binding, the ignored parameters are not discarded but only hidden by the binding. To illustrate this, consider an invocation of `baseMeth2(a5, a6)`. This invocation will cause the overriding callin method `callinMeth(T4 a4)` to be executed with `a6` mapped to `a4`. Within the body of `callinMeth`, a call to the original version `base.callinMeth(a4)` will automatically be translated back to `baseMeth2(a5, a4)`, retrieving the hidden value `a5` from the binding context. Similar considerations hold for the return values of bound

methods. Parameter and result mappings may even contain simple calculations as in

```
int getCredit() → int getMiles() with { result ← ((result + 999) / 1000) * 1000; }
```

Here, the keyword `result` represents the function result and in order to illustrate the direction of data flow, result values are mapped with an arrow which is reverse to the method binding. In all these cases, the origin side of an arrow may be an expression, while the target side must be a parameter name or `result`.

2.6 Context selection

When binding collaborations with callout, behavior is defined in a conventionally modular style. Introducing callin weaving we *blend* behavior from two different views. The last concept to be introduced is concerned with selecting which Team definition should take effect at which point during program execution.

As a default, we declare a Team to be ineffective with respect to its callin bindings unless it is activated beforehand. Activation may take place explicitly or implicitly.

A Team may be activated *explicitly* by invoking a special method `activate()` on the Team instance. This turns on all callin bindings of this Team until a corresponding `deactivate()` call is made. A shorthand for these two calls exists as a new block construct:

```
in (myBonusInstance) do {
   somePassenger.book(someFlight);
}
```

The effect is that, during the execution of this block, all callin bindings of `myBonusInstance` are effective. While it is obvious that the advice of `Passenger.book` is triggered, nested calls (at arbitrary nesting level) to `Segment.book` will also trigger their callin bindings. Now the desired behavior is complete: first, method `collect` is invoked that overrides `book` from `Passenger` (cf. Fig. 7). The method `collect` first passes control to its base method, i.e., `book`. Within this invocation, an arbitrary number of segments may be booked and each segment invokes `earnCredit` on its `BonusItem` role (by an `after` callin binding). Method `earnCredit` (Fig. 4) deposits the corresponding credits within an attribute `accumulator` of the enclosing Team instance. When the method `book` of class `Passenger` finishes, the callin method `collect` will gain control again and transfer the credits from the Team instance to the subscriber object. Note that this collaboration is partially controlled by objects of types `Passenger` and `Segment`, although neither class is modified for the bonus programme.

Two other situations require *implicit* activation of a Team in order to maintain a consistent system state. When calling a Team–level method, the Team is activated because, within this method, role objects may be manipulated, which requires the context to be active. By way of an example, consider a method

```
class FlightBonus {
   void setPassenger(Passenger as Subscriber p) { /* ... */ }
}
```

Note how lifting is declared in this case: the parameter is required to be of type `Passenger`, while within the method body the same object is to be seen as a `Subscriber`.

In a similar vein, an externalized role instance (cf. Sect. 2.1) may trigger context activation. Invoking a method of a role while the context is not active must ensure its activation as well. To illustrate this feature, let us continue the scenario of our action class (Fig. 5). If a button is included in the graphical user interface, which has a `ClearAction` associated, we have the following situation: while the interface is external to the Team and makes abundant use of and/or specializes classes from a library like Swing, it also refers — via the `ClearAction` instance — to an externalized role of the `FlightBonus` Team. Since nothing can be said about when the button will be pressed by the user, each time `performAction` calls to the externalized role, the enclosing Team instance `context` must be activated in order to set up an appropriate context for the execution of role methods.

As a result, callout calls can only occur while the corresponding Team is active. Thus, if a method bound by callout is a template method whose hooks are overridden using callin, the expected behavior where hooks call back to the Team is automatically ensured. Experiments with LAC [7] have shown that a certain class of problems known as the "jumping aspects" problem [1] can be avoided if re-entrance of a Team is explicitly disabled by temporarily deactivating the Team for a callout. Fine–tuned support for activation and lifting is provided by the built-in class `Team`, which is the implicit super–class of all Teams.

Static weaving A Team which should always be active, i.e., permanently modify the application, may be declared static. Staticness means first of all that no Team instances can be created. All features of such a Team are static features. Secondly, and more importantly, all callin code from a static Team is unconditionally woven into the application. The Team in Fig. 8 adds a `BonusPassenger` role to *each* `Passenger` object and statically overrides `book` (line 10). The callin method `entryHook` checks whether a context has been registered with this passenger. In the positive case, the original version of `book` is called in the appropriate activation context (line 6), otherwise it is called unmodified (line 8).

This Team finally makes obsolete the explicit activation on the previous page. Now we have a completely operational and modular implementation of our bonus programme without any modification to the base package and/or main program. Fig. 3 illustrates how subscription can also be differentiated into an inheritance hierarchy.

2.7 Some remarks on method

Initial experience with the programming model of Object Teams suggests that activation of dynamic Teams can typically be realized by one static Team. This way, even concern activation is a first–class concern that can easily be implemented within the language. This approach is more flexible than a keyword–based approach as in AspectJ [9] (cf. the keywords `of eachJVM`, `of eachobject`, etc.). At the same time, it provides better support for reuse than mere design patterns, which cannot be implemented in a reusable way. The `BonusSubscription` Team can even be generalized for attaching arbitrary contexts to arbitrary base objects. By simple and succinct bindings of such a general Team definition, context activation can be woven into arbitrary join–points by binding `entryHook`.

```
1  static team class BonusSubscription {
2      class BonusPassenger playedBy Passenger {
3          Team myContext = null;
4          callin void entryHook () {
5              if (myContext != null)
6                  in (myContext) do { base.entryHook(); }     // invoke in context
7              else
8                  base.entryHook();                           // invoke unmodified
9          }
10         entryHook ← book;
11     }
12     static void setActive (Passenger as BonusPassenger p, Team c)
13         { p.myContext = c; }
14 }
```

Fig. 8. A static entry trigger mechanism.

A predecessor of the current model, Aspectual Components [11], made a strict distinction between the definition of a collaboration and a *connector* that binds it to the application. Technically, this distinction is not needed, and in fact we have observed situations in which partial Team definitions with partial bindings are helpful. Thus, we allow the implementation of a Team to be mixed with binding details. It is, however, a good style of programming with Object Teams to use one Team only for implementing the collaboration and to define another module as a specialization of the Team, which only acts as a connector, i.e., binds classes and methods between roles and base.

2.8 Summary of features

Object Teams provide high expressiveness for a wide range of design styles. Ideas and findings from many different publications and language prototypes have influenced the genesis of Object Teams. When combining positive properties from several programming languages, care must be taken not to bloat the language with different concepts that could easily outweigh the intended improvements. We consider Object Teams a comparably lean model. In order to support this claim, we summarize the features of Object Teams in two sections. First, we list those features that are visible to developers using the language. Second, we focus on features that are implemented by the compiler and runtime system.

From the concepts presented so far, only the following need to be learned by developers:

- Role objects are dependent objects that reside within an enclosing team instance and also need a base instance.
- Callout bindings declare forwarding from a role instance to its base object.
- Callin bindings declare overriding by which a team definition may alter the behavior of base classes.
- Team activation turns on and off all its callin bindings.

Other features can be deduced more or less directly by integrating these concepts with standard object–oriented principles. Inheritance can be applied to team definitions, super calls work in all dimensions of inheritance. Implementation for abstract methods can be provided by either class–based inheritance or callout binding.
The following is taken care of by the compiler and runtime system:

- Base classes and team definitions can by statically type–checked in a modular way.
- Bindings are only allowed in a type–safe manner. This includes type–safe adjustments of method signatures.
- Lifting/lowering translations are implicitly inserted when data flows enter/leave a team.
- Teams are implicitly activated whenever a control flow enters a Team.

3 Related work

Object Teams aim at implementing collaboration based designs [16, 2, 19, 17]. They have their roots in Adaptive Plug&Play Components [12], PCA [13], and Aspectual Components [11]. They have been influenced by Hyper/J [18] with respect to the separation of concern definition and concern composition as well as its capabilities for "on–demand" restructuring. Influence from AspectJ [9] concerns the technique of weaving into existing code. Virtual classes from gbeta and especially family polymorphism [3] have shaped the type system of Object Teams. Delegation layers [15] have added a new look on combining collaboration based design, family polymorphism, and delegation.

From this it should be clear, that Object Teams share properties with each of the mentioned approaches. A comparison against other approaches to collaboration modules has been presented in 2.3. In this final discussion we will focus on differences to other approaches from the field of aspect–oriented software development.

The sublanguage for specifying joint–points in AspectJ [9] is more sophisticated (giving rise to criticism for language bloat) than ours. On the other hand Object Teams provide better decoupling, modularization and flexibility. A Team without role binding is completely unaware of any base package; yet, it is able to make use of base behavior using abstract methods that are later on bound by callout. Capability of callout binding is missing from AspectJ. AspectJ does not provide modules comparable to Teams. Also, an AspectJ program has no control over instantiating and activating aspects at run–time. Finally, AspectJ requires the source code of all classes to which aspects are woven. Object Teams are implemented using byte–code weaving at load–time.

All concerns in Hyper/J [18] are equally independent, with no base–aspect distinction. Concerns are merged at compile time. In contrast, Teams are first–class entities, which persist at run–time with the mentioned capability of (multiple) instantiation and activation. Integration of Hyper/J concerns is specified by declarative composition rules. An Object Team that is used as a connector uses callout and callin binding as well as regular method implementation. Customized combination of the different versions that are merged into one method can be achieved using super, tsuper and base calls.

Within the direct ancestry of proposals, Object Teams make the following advances: Aspectual Components [11] misleadingly subsume callin and callout methods as "expected" methods, but distinguish collaborations and connectors. Object Teams make

explicit that the callin/callout distinction is imperative since very different rules hold for both kinds of methods and bindings. In contrast, a collaboration/connector distinction is just a matter of style, which may be blurred for some applications without loss of safety or structure. Aspectual Components were not clear on dynamic issues regarding instantiation and activation of collaborations and they lack the notions of lifting/lowering.

Adaptive Plug&Play Components [12], DVC [6] and PCA [13] all could be regarded as some kind of Object Teams without callin binding and Team activation. From these, only DVC feature declarative bindings. PCA — notably in their implementation by JADE [5] — support multiple "customizes" clauses, which correspond to implicit inheritance in Object Teams. The explicit variant using "customizes" yields a proliferation of class names but allows to compose several collaborations in one step.

Finally, LAC [7] is the first published implementation of the Aspectual Components model. LAC can be seen as the direct predecessor to Object Teams, but many details of orthogonality between standard object–oriented concepts and Teams have only been settled after the cited workshop paper.

4 Status of Development and Future Work

This paper investigated whether module concepts for capturing collaborations can be effectively used to implement crosscutting concerns in reusable, independently developed modules that are a-posteriori integrated into existing systems. We proposed a new kind of collaboration module, called Object Teams, which combines the best features of existing approaches, further enhances them with concepts for expressing crosscutting relations between independent collaborations, and facilitates *a-posteriori integration* of such collaborations into existing systems.

A prototype implementation of a predecessor model of Object Teams, called LAC [7], already exists. During the transition from the predecessor model as implemented in LAC to Object Teams, the terminology and many concepts have been consolidated. Object Teams improve the orthogonality of standard object–oriented techniques with the few specific enhancements. This results in greater flexibility in terms of combining the various mechanisms, and a smaller number of new features and keywords to be learned by programmers. A compiler for Object Teams as an extension to Java is currently under development and already shows the feasibility of most techniques. It is due to be usable for real application before fall 2002 (cf. our web site [14]). Several case studies for assessing the usefulness of Object Teams are already in progress. In this paper we have only looked at the application to one specific domain. Application to other typical problems has yielded encouraging results, but further research, including the development of various supporting tools, is needed to make a final assessment regarding the real–world usefulness of Object Teams.

Acknowledgements

The work presented here would not have been possible without the many contributions by Mira Mezini, including ideas put forward in [11] and many intensive discussions on these issues. The following local colleagues and students contributed to Object Teams by discussions and implementations: Jan Wloka, Christof Binder, Christine

Hundt, Matthias Veit, Florian Hacker, Carsten Pfeiffer, Marco Mosconi and Timmo Gierke. Thank you all!

References

1. J. Brichau, W. De Meuter, and K. De Volder. Jumping aspects. position paper at the workshop "Aspects and Dimensions of Concerns", ECOOP 2000, June 2000.
2. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML – The Catalysis Approach*. Addison-Wesly, 1998.
3. E. Ernst. Family polymorphism. In *Proc. of ECOOP'01*, number 2072 in LNCS, pages 303–326. Springer Verlag, 2001.
4. W. Harrison and H. Ossher. Subject-oriented programming: a critique of pure objects. In *Proc. of OOPSLA'93*, pages 411–428. ACM, 1993.
5. M. 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.
6. 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.
7. S. Herrmann and M. Mezini. Combining composition styles in the evolvable language LAC. In *Proc. of ASoC workshop at the 23nd ICSE*, 2001.
8. Stephan Herrmann. Composable designs with UFA. In *Workshop on Aspect-Oriented Modeling with UML* at 1^{st} *Intl. Conference on Aspect Oriented Software Development*, 2002.
9. G. Kiczales, E. Hisdale, J. Hugunin, M. Kersten, and J. Palm. An overview of AspectJ. In *Proc. of 15th ECOOP*, number 2072 in LNCS, pages 327–353. Springer–Verlag, 2001.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect Oriented Programming. In *Proceedings of ECOOP '97*, number 1241 in LNCS, pages 220–243, 1997.
11. K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. In *Technical Report*, Northeastern University, Apr. 1999.
12. 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.
13. 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 M. Aksit (ed.) *Software Architecture and Component Technology: State of the Art in Research and Practice*. Kluwer Academic Publishers, 2001.
14. Object Teams home page. `http://www.ObjectTeams.org`.
15. K. Ostermann. Dynamically composable collaborations with delegation layers. In *Proc. of ECOOP 2002*, LNCS. Springer Verlag, 2002.
16. T. Reenskaug. *Working with Objects – The OORAM Software Engineering Method*. Prentice Hall, 1996.
17. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. In *Proc. of ECOOP'98*, number 1445 in LNCS, pages 550–570. Springer Verlag, 1998.
18. P. Tarr and H. Ossher. *Hyper/J User and Installation Manual*, `http://www.research.ibm.com/hyperspace`. IBM Corporation, 2000.
19. M. VanHilst and D. Notkin. Using role components to implement collaboration-based design. In *Proc. of OOPSLA'96*, volume 28(10) of *ACM SIGPLAN Notices*, 1996.