Orthogonality in Language Design – Why and how to fake it.

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

Abstract

A major challenge in the design of programming languages is the definition of how elementary concepts interact such as to approximate the impression of orthogonality. We propose to use techniques of concern modeling for capturing such interactions with the goal of providing a language definition which aligns intuition and actual semantics. We report on the design of Object Teams, a programming model aiming at improving support for modularity by combining several well explored techniques.

1 Introduction

Most recent publications on the design of programming languages focus on the introduction of one or a few innovative concepts. The technical elaboration of such concepts is often performed by integrating the new concepts into an existing *host language*, these days usually Java. Such integration demonstrates, how the new concept fits into the "normal" object oriented programming model.

Hoare introduces a strict distinction between the activities of designing a language feature versus a complete programming language [9]. It is our belief, that lately too much emphasis has been put on the role of single features underestimating the importance of *interactions* between different concepts.

Is it still, that language designers believe in orthogonality between language concepts? Is adding a new concept to a language really adding a new dimension, just one more degree of freedom?

In this paper we propose a fresh look on orthogonality, which is based on the pessimistic assessment that concept interaction is the rule not the exception. Thus language designers should never assume any orthogonality. Through the hard work of language designers, however, users of that language (the programmers) should be assured that their programs will behave *as expected*. They should be given maximum freedom of combining different language concepts under the greatest possible *impression* of orthogonality.

We propose a process of three steps to achieve a usable programming language incorporating state-of-the-art concepts.

Un-tangling concepts. First, the concepts to be composed into the new language should be un-tangled. This step identifies the basic building blocks for language design. As an example, Ostermann and Mezini analyze the concepts underlying inheritance and aggregation, isolating five basic concepts which were tangled in the original concepts [15]. Further analysis should not be based on bundled concepts like inheritance, but on "more atomic" things like acquisition or overriding.

Concern Interaction Matrix. A second step should systematically explore all interactions between the given set of basic concepts. For this step we propose our technique of a Concern Interaction Matrix [7]. This centrally means to document for each pair of concepts how they interact with each other. It would be naïve to expect phases one and two to be strictly separable. More so, describing concept interactions helps to understand what should be considered a suitable level of concept decomposition. I.e., while constructing the Concern Interaction Matrix some supposedly atomic concepts may need to be decomposed further.

Usable concept bundles. The second step generates a very detailed picture of a programming language. All these details would be overwhelming for an ordinary programmer. After basic concepts and all their interactions have been explored and are well understood, programmers might need to be provided with more intuitive "bundles" of concepts. This looks like the initial input to the overall design process. We believe that in fact many existing concepts have proven useful in praxis *because* they bundle more atomic concepts in a usable way. This raises the level of abstraction but restricts the programmer, in that some unintended combinations of atomic concepts might not be expressible in the resulting language.

The underlying idea of this three-step process proposes two distinct views on a language: the designer's view must be very detailed, dealing with atomicity and concept interactions. The programmers' view should be less detailed, more based on abstraction and intuition. The programmers' view should give the impression of orthogonality, because this is the precondition to creative combination of language concepts in order to solve particular problems in each program's domain.

The final assessment of a programming language needs to take into account, whether intuition (including the impression of orthogonality) ever breaks or suggests results different from the actual semantics of the language. There might be cases, where the intuition cannot be maintained but the programming environment should inform the programmer what's going on, provide explanation and possible remedies.

Language design as concern modeling. According to our observation, language design is no less complex than design of large applications. In the latter field, concern modeling [17] is emerging as a systematic, yet pragmatic method for managing a complex network of requirements. While only a formal model of a programming language ensures complete and unambiguous definition of its semantics, we see two reasons for (also) using a less formal model: For a real-world programming language a formal model is very expensive to obtain. Most approaches to formally defining existing programming languages work with a reduced subset of the respective language. The effort of filling in remaining gaps grows surely (much) more than linear. This is not really an excuse on the long run, but before a language design has faithfully approached a fix-point more light-weight techniques are probably more efficient.

Secondly, a formal model may prove well-definedness for a programming language. It is, however, not clear whether it really gives an understanding whether intuitive and precise semantics are well aligned. The reason for this can be seen in the structure of formal models, which suffers from similar problems concerning separation of concerns as also programs do. Formal models, just like programs, are decomposed along one dominant dimension. After separating static and dynamic semantics this dimension typically is an enumeration of all syntactic elements of the language. This hardly supports looking at a language from multiple perspectives, especially if those perspectives are not independent.

The middle ground is to be found in a systematic multi-perspective approach.

Reporting from the design of Object Teams. Throughout this paper, we will report on the design of the programming model Object Teams[6] and its embedding into Java, ObjectTeams/Java (OT/J for short) [14]. The central goal of Object Teams is to raise the level of modularity for complex problems. Hence, this paper tries to un-tangle some issues of modularity. More precisely, we explore different aspects of concepts for modules and, subsequently, concepts for composition (Sect. 2). Section 3 starts with a terse illustration of Object Teams and unfolds how elementary concepts for modules and compositions map to the concepts of Object Teams. Section 4 reports on a few obstacles that where encountered when embedding the model into Java. These observations suggest the direction for a subsequent language design, which may not be hosted by Java.

In our conclusion we hope to convince the reader that we as a research community have enough language concepts, we need deeper understanding on their interactions, find smarter ways of combining them, pushing each concept to its full consequence.

2 Towards a concern model of modularity

The programming model Object Teams emerged from the field of AOP, where modularity is brought to problems which until then have been cross-cutting the dominant decomposition of a program, no matter how good the design. Object Teams transcend this pursuit for better modularity, incorporating ideas from adaptive programming, role objects, layered designs, to name only the most obvious sources of influence. In order to give some inside view, beyond advertising this model to programmers, we will elaborate on the two faces of the coin "modularity". First, we investigate what is a module. Only after un-tangling some motivations and solutions we will dive into the second face: composition of modules.

2.1 Decomposing module concepts

Much has been written about modularity, and all agree on the importance of modularity. This agreement may, however, be based on the unspoken diversity of interpreting the notion modularity. Throughout this section we will use the notion "module" in its broadest sense, including, e.g., concepts like classes, packages, components. Modularity is tightly coupled to a number of other important concepts like information hiding, encapsulation, locality, and decoupling. A more detailed analysis requires to investigate different *motivations* for wanting modules. Our views on module concepts will differ to a surprisingly high degree, depending on the respective motivation.

Organizational aspects. From an organizational point of view, modules are good for splitting development efforts. On a small scale this means that a module should be identifiable as one or a set of files. This is the pre-requisite for different developers working in parallel on different modules of the same system. Along this road, separate compilation should be regarded as an organizational issue, to. At a larger scale this calls for the possibility of independent deployment of components from different vendors.

In order to guarantee minimum independence between different developers, modules should define separate name spaces enabling developers to give elements in a module suitable names without worrying about name clashes with elements in other modules. From the organizational point of view all other properties of modules relate more to the issue of composition or integration. In that context we will look at interfaces and contracts.

Separate processing. Apart from development efforts in a narrow sense, what else do we want to do with a module? We have already mentioned separate compilation. Also separate type checking is of high importance. Pushing the limits of type checking we finally strive for "modular reasoning". Reasoning again will postulate different requirements to modules depending on the properties which we want to prove.

Difference modules. Some modules may not be intended for integration or deployment "as is". Frequent uses of modules are adaptation and refinement. This process of enhancement takes a module and provides a new module. Here the goal may be, to put all enhancements into a separate module, leaving the original module intact and bundling a consistent set of enhancements into a *difference module*. Some difference modules will be designed with respect to a specific base module, others should be composable with different base modules, which can be achieved by an explicit *expected interface*. Anticipating refinement, it is desirable to be able to construct incomplete modules, i.e., modules which can *only* be used together with a difference module. From this perspective, abstractness and an expected interface differ only in the intended style of composition.

Instances. If modules should provide autonomous entities also at runtime, a concept of *instances* comes into play. At an intuitive level, the conceptual value of instantiation is beyond discussion. A more precise motivation is much harder to grasp, *because* this concepts fits so nicely into our intuition. Of course, instances of primitive data types are indispensable. Instantiable array and record types add much expressiveness to a language. Why do we want more? From all object-oriented concepts, only dynamic binding cannot be simulated easily in a modular language like Modula-2. What is the reason for wanting dynamic binding? It is a matter of responsibility: the callee knows better about how to react to a message than the caller. This is the autonomy gained by instantiability in the object oriented context. It is a matter of composition techniques to provide different points in time, when to decide which behavior should be associated to which instance (compile time, creation time, or runtime).

Protection. At a first look, working with modules seems to guarantee locality. Here object orientation introduces a new problem: aliasing breaks locality. An analysis of what messages will be sent to a given object may in the worst case involve analysis of the whole program, since little is known about number and location of references to an object. If an object shall be protected, e.g., due to security constraints, new, mostly instance based, borders have to established within a program. These borders may or may not be identified with modules. An even closer look reveals a set of different possible goals, concerning the protection of objects. Outside a give border an object may have to be completely inaccessible. It might also suffice to restrict outside visibility to read-only references. An different goal might be to just prevent access to internal information. In this case it may be perfectly legal to access an object by a narrower interface, if it can be assured, that this interface operates on an isolated slice of the object which is independent of the secret slice.

2.2 Decomposing composition techniques

The two seemingly elementary composition techniques in object orientation are references and inheritance. As already mentioned, inheritance is by itself a bundle of concepts. Slightly adapted from [15] the following elementary concepts can be identified:

acquisition: A class or instance may acquire functionality from a super-class or parent instance which is available for method calls without further qualification.

overriding: A class or instance may override methods of another class or instance to which it is attached. Overriding may occur shallow (once the control is passed to the parent, no overriding occurs any more) or deeply, i.e., the two entities are seen as one and every invocation of the original method is overridden by the new method.

subtyping: An instance of a more specific kind (subclass or delegation-based object composition) is substitutable to the more general kind.

The paper[15] also mentions transparent redirection (which we subsume as deep overriding) and polymorphism in a sense which is meaningless for classical inheritance, but relates to delegation based composition. We will come back to the latter.

Categories of associations. It has been discussed many times, many places, that modeling notations like the UML tend to provide more relationships than just references and inheritance. The detailed distinction of association, aggregation and composition is not found in traditional object oriented programming languages. The discussion of alias control already touched the issue, that stronger forms of referencing like composition are in fact desirable. Similar concepts in this context are containment and ownership. Only with some kind of containment, these techniques scale for large systems. Containment should allow nesting and layered designs.

Along those lines, the desire for modules larger than classes not necessarily calls for new modules but for new module relationships. Java packages are an example for modules larger than classes, which are just too weak, because there are no means to aggregate and refine packages. Components are one solution, but for several reasons such concept should be available within a programming language not by an external infrastructure, which might just be too heavy-weight.

Adaptation relations. Recently, aspect-oriented programming introduced a new style of composition which is mainly based on *method interception*. While some approaches try to unify traditional and aspect oriented message passing to event based programming, for the analysis at hand method call and method interception are more explicit. At a larger scale the new relationship is *adaptation*. By this, a difference module may from the outside change the behavior of an existing module. Such difference module may either be invisible, i.e., its only effect is a modified behavior of the adapted module, or the adapter may provide a public interface. In the latter case, the adapter mediates between an agnostic base and an informed part of the program.

In environments where modules cannot be composed using refinement and adaptation using explicit relationships, those tasks are usually performed by so-called *glue code*, techniques which are mostly outside public discussion. Effective glue code programming requires something like a reflective scripting language. Glue code is not measured by elegance, but the only goal is to get the job done, because structured techniques often just cannot solve the problem. Thus, glue code often intentionally breaks encapsulation, bypasses public interfaces etc. pp. Glue code, as it is, gives a hint at some severe nuisance. I don't want to be misunderstood here: this is not the fault of people writing glue code, they often have no alternative and just perform an awfully important job. Nor is it in all cases the fault of the people who wrote the components. The fact that these components don't fit together "as is" but have to be forged into a composition cannot be avoided for large scale development and large scale re-use. Any technology that helps to perform the task of composition in a somewhat more structured way greatly improves this situation. The point to note is: if programming languages in the first place provide means for adaptation relations this may eventually obsolete the "power-tools" by which glue code can force components into unintended usage contexts.

Granularity of relations. After talking about large scale module composition, on the other side of the spectrum we should ask: what is the level of detail by which interfaces and other relations are defined. Many maintenance tasks would be easier, if it were explicit which methods exactly are imported from a given module. Changes to methods not in this list are probably harmless. Also a module's export could be much more explicit than what can be expressed by Java's four levels of visibility. Independent visibility control for clients and sub-classes is definitively desirable and directed export to specific clients could also help to keep interfaces narrow. Even overriding of methods is a good candidate for explicit specification (cf. Eiffel's `redefine` clause). All these are just syntactic specifications. We just mention that contracts should also be smoothly integrated in a modern programming language as an enrichment of interfaces.

This list of techniques should give an impression of what might be possible in terms of fine grained, explicit and expressive definition of module composition. Todays programming languages restrict programmers in what can be expressed. Of course convenience is a striking argument for giving predefined bundles of composition techniques. Typical problems can be solved with little effort. Just once in a while the bundles have to be reconsidered in order to allow for the development of programming languages that better address todays problems in programming.

3 Modularity in Object Teams

The above discussion spans the field for a detailed presentation of the programming model Object Teams.

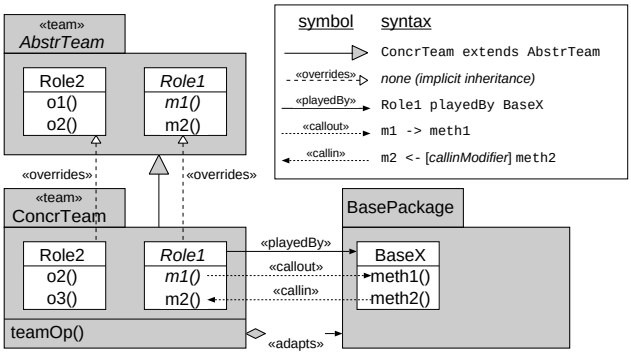

Figure 1: Typical structure using Object Teams

We will, however, start with a brief intuitive introduction. For brevity, we cannot present an example which demonstrates all features of Object Teams. We apologize for referring the reader to previous publications [6, 18].

3.1 An overview

Object Teams [6, 18] introduce two new kinds of modules and new relationships on four different levels. A typical combination of these constructs is illustrated in Fig. 1.

3.1.1 Modules

Team. A *Team* is an instantiable module for collaborating classes which combines properties of classes and packages.

Role. A *role* class is a class contained in a Team. Accordingly each role instance is contained in a Team instance.

3.1.2 Relationships

PlayedBy. A role class may be associated to a base class (any kind of class) by a *playedBy* relationship. This expresses that each role instance will be linked to a base instance of the given class. This relationship allows to exploit all the benefits of role objects (see e.g., [10]). In contrast to other techniques for role objects, we require a detailed specification of the role–base composition (see callout/callin below).

Translation polymorphism. There is no sub-type relation between associated role and base classes. Still in specific situations their instances are substitutable. Interestingly, this substitutability can be achieved for both directions. Substitution involves two kinds of translations which are called *lifting* and *lowering*. The latter is a simple navigation along a hidden parent link. The former retrieves an appropriate role instance for a given base object. Sequences of lifting and lowering preserve identity and state.

The lifting translation uses a Team instance as the context that disambiguates the 1:n base-role association. Thus, lifting is a ternary association: it links base instances to role instances with respect to the context of a Team instance.

Lifting and lowering apply at every data flow across the border of a Team which involves types that have a `playedBy` relation. For each point of translation the language implementation evaluates these three inputs: the dynamic type of the object to be passed, the Team instance whose border is to be crossed and the expected type at the receiving side. By these ingredients the translations are performed automatically, i.e., without explicit action by the programmer.

Callout. A declarative method binding, by which an abstract role method is bound to a base method. This binding is implemented by forwarding. Method bindings are part of the definition of a role class.

Callin. The inverse of *callout*, this binding declares that a given role method should be executed for every call of the associated base method. When specified with the modifier `replace` this means that a base method is overridden by a role method. Other options are to add the role method `before` or `after` the base method. Callin bindings implement a form of callbacks, rely on method interception, and can be seen as *advice* in the terminology of AspectJ.

Parameter mappings. Both kinds of method binding permit to declaratively re-arrange parameter lists.

Team inheritance. Inheritance between Teams works as normal for direct methods and attributes of the Team. It entails *implicit inheritance* for all contained role classes.

Implicit inheritance. A Team has access to all roles defined in one of its (indirect) super-Teams. If the Team defines a role by the same name as a role acquired from a super-Team, the new role implicitly inherits all features of its precursor and overrides the precursor. Thus, role classes are virtual classes (cf. [3]).

Lack of sub-typing. Implicit inheritance does not establish a sub-type relation between role classes from different Teams. This is the precondition to static type safety of implicit inheritance.

Adapts. This relationship is not explicit in the language, but all adaptations, which callin bindings perform on base classes, are effective only if a corresponding Team instance has been activated. Thus, adaptation is a dynamic relation between a Team instance and a set of base classes.

3.2 Mapping concepts to our analysis

We will no map the concepts of Object Teams to the elementary concepts of modules and composition from our analysis.

3.2.1 Modules

Organization and separate processing. Role classes, which are similar to inner classes, can optionally be stored either in-line in the file of the enclosing Team, or as separate files in a directory dedicated to all roles of on Team class. Name spaces between a Team and the adapted base are strictly separated. Only the declarative bindings tie a link between both name spaces.

The concrete language OT/J does not require access to source code of base classes because the *adapts* relation is realized using byte code transformation at load time. Emphasis in separate processing lies on a strict separation between a Team and the adapted base. We are currently elaborating the type system for OT/J in a formal calculus which will show locality of matters of type safety. Adding contracts to Object Teams is an area of future work which will set the foundation for modular reasoning.

Difference modules. Teams are difference modules in two ways: by using Team inheritance, a Team may enhance its super-Team. Using the adapts relation, a Team modifies an existing base. Team inheritance is a static property, while adaptation is a dynamic, instance based relation. Adaptation is decomposed into smaller difference modules: role classes, methods bindings, parameter mappings. For both kinds of relations, inheritance and adaptation, a Team is a module which groups a set of smaller difference modules. It should be emphasized that this may be the major contribution of Object Teams: to allow for modular differential designs.

Note that Object Teams require no new technique for expression an expected interface: abstractness of role classes and methods suffices to indicate that some functionality is still missing. Implementation for abstract methods may be provided either by a sub-Team *or* by forwarding to a base method (callout binding). This provides a balance between explicitness and flexibility.

Instances. Both new modules, Teams and role, make extensive use of instantiation. Role instances are contained in a Team instance. In fact, the type of a role object is associated to the surrounding Team instance. So in some some situations a role type is denoted explicitly as `someTeamInstance.RoleClass`. This technique is borrowed from family polymorphism [4].

Adaptation and the lifting translation are under the responsibility of a Team instance. This abstraction is powerful enough to even address the problem of aspect ordering, which is difficult to express in other aspect oriented languages.

Protection. Instance based typing (see the previous item) was originally introduced just for type safety in the context of covariant overriding of role classes. The same concept can now be pushed to its full consequence which allows different levels of protection.

The most flexible level allows *externalized roles*, i.e., role objects that are referenced outside their Team. Here instance based typing only prevents role objects from different Teams to be mixed. Apart from this restriction, an externalized role object can be accessed just like a regular object.

We are currently elaborating a concept of *opaque roles*, which is to say, that a role object cannot be accessed by its role interface, but only by widening to a non-role type. This requires, that the two slices of the role object don't interfere, i.e., execution of non-role methods may never access any role attributes or role methods (which could happen by overriding non-role methods within the role class). By prohibiting casts to the original role type it is thus possible to use role objects outside the Team without allowing access to any specific role properties. By opaque roles, we hope to provide a useful balance between strict, provable protection combined with the flexibility to point into the state of a Team.

A similar concept of object-slices is already present in the model by the role-base relation: roles may typically encapsulate specific enhancements, which may be kept private to the Team, while different contexts (Teams or non-Team contexts) may share the underlying base objects.

Finally, *confined objects* — designed along the lines of confined types [19] and Universes [13] — are guaranteed to be accessible only within the enclosing Team instance. The type system will disallow confined objects to ever escape the context of their Team. More specifically, confined objects cannot be widened to non-role types. Previous works have pointed out, that traditional object oriented models lack a natural border for restricting the scope of references. A Team instance provides exactly this natural border of confinement.

We believe the three options of externalized, opaque and confined role objects to suffice for real world tasks of protection. For that reason we do not offer a further alternative: read-only references. If really needed such situations can be crafted by non-role classes implementing the read-only access. Roles will then inherit those read-only slices and add the modification slice to each object.

3.3 Composition.

Here we investigate which of the elementary kinds of composition can be found in Object Teams.

Acquisition. While it is not surprising that Team inheritance makes available all roles from the super-Team, the role-base link deserves a closer look. Object Teams require an explicit *callout* binding for every base method that shall be accessible from a role. Due to the declarative style of binding we consider this an acceptable burden. By making acquisition of base features explicit, name spaces are clearly separated and dependencies are minimized.

Overriding. Along the lines of the previous item, overriding is implicit along Team inheritance, including the possibility to override a role *class*. By default, role methods do not override base methods. Callout bindings are defined as mere forwarding without late binding of self. A role method may still override a base method by an explicit *callin* binding. Again the conservative default and a more flexible option help to decouple role and base without loosing any expressive power.

Subtyping. Object Teams introduce implicit inheritance, which does not define a sub-type relation between roles. Yet a *Team* is a sub-type of its super-Team. The concept of translation polymorphism also relates to sub-typing. A role class and its associated base class together define an imaginary tuple type. This compound type is conform to both element types. Each facet of this type can be translated to the other facet.

Polymorphism. The kind of polymorphism investigated in [15] is approximated by an abstract Team without role bindings. Such Team can be bound to (almost) arbitrary sets of base classes. In this respect Caesar [12], which is otherwise quite similar to Object Teams, goes one step further by introducing collaboration interfaces, by which even bindings can be re-used against a polymorphic hierarchie of collaborations.

Modules larger than classes. Clearly, Teams fulfill the request for modules larger than classes. They come with useful semantics including instantiation, refinement and containment of role objects.

Method interception. Callin bindings unify the concepts of selective overriding and method interception. The modifiers `before`, `after`, `replace` relate this inheritance in CLOS and advice weaving in AspectJ.

Adaptation. We have already pointed out that a Team is a module for a complex set of adaptations. Just like in prototype based languages, adaptations are first-class. Unlike those languages, a Team may hide details of adapting many classes, methods, parameter lists. Teams furthermore introduce the concept of runtime activation and deactivation which is also more modular than in the few other approaches which feature runtime aspect activation (e.g., [8]).

Granularity. Composing role and base classes can be controlled with much greater detail than inheritance and import in standard object oriented languages. Everything that is not mentioned in a method binding is not visible. It is even possible, to import (via callout binding) a base method, or to intercept a base method (via callin binding), with only a subset of its parameters.

At the bottom line, Object Teams are prepared to compose modules with mismatching interfaces. By reasonable defaults and the option of very fine grained control adaptation can be defined in a succinct and well-structured way. A Team groups a set of adaptations to a consistent unit. The detailed control over module composition is achieved by only these programmer-visible concepts: Teams and roles, implicit inheritance and playedBy relation, callout and callin method bindings.

3.4 Interaction of basic concepts

The analysis in the previous sections has identified some elementary concepts of modules and module composition in Object Teams. The same kind of analysis can be performed regarding the host language into which the new features are integrated. This level of granularity facilitates precise statements about the interactions amoung the given concepts.

For this purpose, a Concern Interaction Matrix [7] is to be constructed. Within this matrix, each pair of concepts shall be analyzed to see whether their combination behaves well without further effort, or — what tends to be the normal case — define the constraints under which these concepts can be combined. Thus the detailed language semantics are completed following

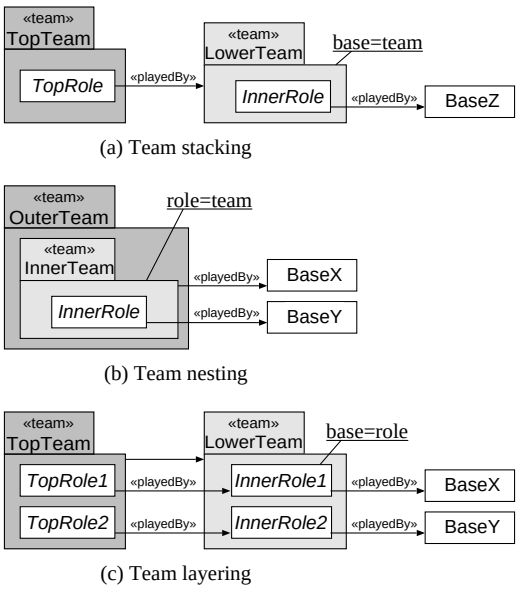

Figure 2: Composing Teams, roles and bases

the "principle of least surprise" [11]: the programmer should not be surprised by unexpected behavior when applying language concepts in an un-foreseen manner.

Due to the prevalence of concept interactions, it is also of high value to document which concepts do *not* interact, and why this is so.

We will give a few examples on what needs to be done to avoid ill-behaved programs.

3.4.1 Large scale structures

Teams are an answer to the quest for modules larger than classes. In order to scale for large systems it is desirable to create decomposition hierarchies using Teams. The different kinds of structures that can be built follows from an analysis of which properties can be combined in one class.

Three combinations are meaningful for which the semantics shall be discussed (cf. Fig. 2).

base=team. A base class which is adapted by a role class may in turn be a Team class. By this technique a special form of hierarchical structure can be constructed which is illustrated in Fig. 2(a). This combination is possible without further precaution, because from a client perspective the intermediate Team `LowerTeam` behaves like a regular object. It serves as a facade to its contained roles, which do not directly participate in the upper level role-base binding.

role=team. A strong form of nesting can be achieved by implementing roles of a larger Team as contained Teams (Fig. 2(b)). At a first glance this looks unproblematic, too. Only when measuring this against Team activation we find a possibility of inconsistency: If only a contained Team is activated, while the outer Team is inactive, a callin binding might direct the control flow into the inner Team. This Team as a role of the outer Team is not activated, i.e., its callin bindings do not override corresponding base methods. This *might* produce unexpected behavior, but it is not strictly wrong to execute role methods while the surrounding Team is inactive. We have seen situations where this actually makes sense. So we only give the *advice* that nested Teams *should* redefine the `activate()` method to consistently propagate activation to inner and outer Teams.

base=role. Using a role object as a base is only possible under the following constraint: the upper level Team must have an immutable (final) reference to the Team who's role should be used as base. Then it is possible to declare:

```
class TopRole1 playedBy thatTeam.InnerRole1
{ ... }
```

Of course special considerations regarding activation are again needed. This style of composition is useful for realizing layered designs (Fig. 2(c)).

3.4.2 More examples for concept interactions

Also interactions like the following can be found systematically by investigating each pair of concepts.

Exceptions. The rules on method binding must define constraints regarding declared exceptions of role and base methods. Another interaction is less obvious: the language introduces a new block statement

```
within(teamInst) {
  stmts;
}
```

which expresses that for the given block of statements the Team `teamInst` should be activated. The block structure shall guarantee that no-one forgets to deactivate a Team after use. In a language with exceptions this *must* be translated into:

```
teamInst.activate();
try {
    stmts;
} finally {
    teamInst.deactivate();
}
```

Garbage collection. The `playedBy` relation establishes an invisible link from role to base. The lifting translation requires to store existing base–role mappings in a cache. The references introduced by both

mechanisms must not interfere with garbage collection. It is correct for a role–base link to prevent deletion of a base object, the lifting cache must allow to jointly delete a role–base pair. This calls for a realization of this cache as a `WeakHashMap`.

Visibility. Acquiring methods by callout bindings has the intention of creating a relation that is robust and yet fairly tight. Java defines visibility only concerning client usage and inheritance. In order to allow unanticipated enhancements our role–base binding establishes special priviledges overriding even the `private` modifier. For organizational reasons this access may need to be disallowed which can be done by sealing a (base) package.

3.5 Challenging orthogonality

We have reported only about a small selection of concept interactions. In some cases orthogonality can indeed be established by skillful translation. In other cases a few constraints suffice to simply disallow programs for which no reasonable semantics could be defined.

The full Concern Interaction Matrix for Object Teams is work in progress at the time of this writing. The difficulty lies in identifying suitable elementary concpts. If we would ask how, e.g., inheritance interacts with instantiation, the question were not concrete enough to be answered in a complete way. By contrast, the Concern Interaction Matrix of elementary concepts will systematically eliminate any ambiguities from a language definition. We can already see how the work towards this Matrix helps to elaborate and consolidate Object Teams.

4 Java as a host language

When using Java as the host language for Object Teams, we had to adapt to some peculiarities of Java. In four areas Java hindered a clean design.

Java's notion of *static* members does not fit the way we would like to define static Teams: a static Team should be effective without instantiation and activation. Thus, it would be most suitable to use the Team class as a (singleton or meta) object. Unfortunately, this is not what static members in Java allow, since dynamic binding does not apply to them.

Overloading unnecessarily complicates declarative method bindings, since names do not suffice to refer to methods. The robustness under evolution achieved by Object Teams is impaired by overloading, since addition of a method may render an existing binding ambiguous.

Java's *access modifiers* are unfortunate bundles of export scopes. It is neither satisfactory to redefine their semantics for Object Teams nor to add new export constructs from scratch.

Constructors are not a good technique when it comes to different initialization strategies. Constructor overloading is a very poor workaround. This affects Object Teams in the context of role instance creation which happens implicitly during lifting. Also explicit role creation is possible. Named initialization methods would integrate much smoother with both styles of method bindings, too.

Upcoming generics in Java yield a new challenge in language design: virtual classes are known to have similar expressiveness as generics. For many applications a Team with a fully abstract role is much more appropriate than a generic Team. Still for non-Team classes genericity will be a valuable addition. It remains to be shown how these two worlds combine most elegantly.

Many other features of Java are fairly neutral to our additions. The technique of translation into byte code nicely supports adaptation of base classes without seeing their source code. Surprisingly, even the byte code is so tightly coupled to the Java type system, that overriding role classes could not easily be reflected, but plenty of casts are needed in the byte code generated by the OT/J compiler.

5 The road ahead

Following Hoare [9], language design should always focus on "most difficult aspects of a programmer's task". To some extent programming is not even the most difficult task of a programmer. Communication — both with peers as well as with customers — tends to be notably more difficult. Of course, programming languages should also be seen as a means for communication. From that it should always be kept in mind that readability of programs is much more important than that they are easy to write. Evaluation case studies for Object Teams are planned, which will among other goals assess whether Object Teams provide a suitable basis for communication.

That said, we see greater problems in overall management of complexity (under evolution) than in particular things that could not be expressed by todays state-of-the-art language features. A severe problem remains in the fact, that usually only a small selection of such features is at hand, since language features cannot (easily) be composed by application programmers. We intentionally avoid the discussion about meta programming in this place.

Object Teams give an example of the expressiveness of a programming language that can be achieved mainly by carefully combining existing concepts and

features. It should be emphasized that combination happens an the level of techniques as well as on the level of motivations. Evolution, organization and security are very different aspects of programming. Still all of these benefit from encapsulating roles in a Team. Yet, each motivation contributes a slightly different view. As a result a Team is defined by a directory, a set of interfaces and provides a boundary around role instances. These three aspects of a Team are neither identical nor orthogonal. The concept "Team" is a careful superposition of different views on the concepts of modules and their composition.

Some highlights, which Object Teams bring together, are modules larger than classes, modular aspect-oriented programming including dynamic activation, role objects, integration oriented software development, security, type safety and explicit interfaces including expected interfaces. None of these are actually new any more. From all the techniques involved, perhaps the lifting translation has been first implemented in precursors of Object Teams [5].

The programming model of Object Teams has been incarnated in these host languages: Lua, Ruby, Java, C++. Each instance has given different stimulus to the design. Still it is difficult to tell, what would be the ideal core language to build Object Team upon. Judging by intuition, Eiffel might be a very good candidate, but the complexity of the matter disallows any judgment without a careful analysis of concepts and their interactions.

References

[1] M. Aksit, M. Mezini, and R. Unland, editors. *Objects, Components, Architectures, Services, and Applications for a Networked World, International Conference NetObjectDays, NODe 2002, Erfurt, Germany, October 7-10, 2002, Revised Papers*, volume 2591 of *Lecture Notes in Computer Science*. Springer, 2003.

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

[3] Erik Ernst. *gbeta – a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Århus, Denmark, 1999.

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

[5] 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.

[6] Stephan Herrmann. Object teams: Improving modularity for crosscutting collaborations. In Aksit et al. [1].

[7] Stephan Herrmann. *Views and Concerns and Interrelationships - Lessons Learned from Developing the Multi–View Software Engineering Environment PIROL*. PhD thesis, Technical University Berlin, 2002.

[8] Robert Hirschfeld. AspectS - aspect-oriented programming with squeak. In Aksit et al. [1].

[9] C.A.R. Hoare. Hints on programming language design. Computer Science Department Report CS-403, Stanford University, 1973.

[10] B. Kristensen and K. Østerbye. Roles: Conceptual abstraction theory and practical language issues. *Theory and Practice of Object Systems*, 2(3):143–160, 1996.

[11] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall International, New York, 1992.

[12] M. Mezini and K. Ostermann. Conquering aspects with caesar. In AOSD'03 [2].

[13] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[14] Object Teams home page. http://www.ObjectTeams.org.

[15] K. Ostermann and M. Mezini. Object-oriented composition untangled. In *Proc. of OOPSLA 2001*, volume 36 of *Sigplan Notices*, pages 283–299. ACM, 2001.

[16] D. Parnas and P. Clements. A rational design process: How and why to fake it. *Transactions on Software Engineering*, SE-12(2):251–257, 1986.

[17] S. Sutton, Jr. and I. Rouvellou. Modeling of software concerns in Cosmos. In *in:*, pages 127–133, Enschede, Netherlands, 2002. ACM Press.

[18] M. Veit and S. Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In AOSD'03 [2].

[19] J. Vitek and B. Bokowski. Confined types in Java. *Software– Practice and Experience*, 31(6):507–532, 2001.