

Demystifying Object Schizophrenia

Stephan Herrmann
Independent
stephan@cs.tu-berlin.de

ABSTRACT

A central dichotomy for specialization mechanisms is the divide between static class-based inheritance and dynamic instance-based delegation. Despite its greater flexibility delegation has not found its way into mainstream object-oriented languages. Searching for a reason of why this is so, I find one notion that seriously discredits any delegation-based approach: “Object Schizophrenia”. This paper tries to rationalize the discussion by demonstrating that not all forms of split identity are evil. This is done by unfolding how Object Teams supports split identities while carefully avoiding the known problems ascribed to object schizophrenia.

1. WHY DELEGATION?

The dispute between advocates of inheritance and delegation seems to be one of the fundamental characteristics of research on object-oriented languages[11]. The benefits of delegation are prominently brought forward, e.g., in the GoF book[3] and hardly ever questioned. Recently, the discussion has been revived by the interest in language support for roles [4, 1, 12, 8]. In this context the advantages of role instances having their own identity are clearly stated:

- **Dynamism:** Role instances can be attached to and removed from base instances at runtime thus accounting for properties that an object exhibits only in specific situations. For example consider a person who is not born as an employee but may be hired at some point during his/her life, and may also quit the job at some later point - without invalidating the person that is.
- **Multiplicities:** Multiple role instances (even of the same type) can be attached to each base instance thus allowing the base instance to have different state when seen from different contexts. If a person accepts multiple jobs, it must be possible to negotiate different salaries: one per employee role instance.¹

Whenever real world phenomena exhibit such dynamism or unbounded multiplicities, static inheritance fails to provide a “natural” modeling solution which makes role instances with delegation the superior concept in these situations.

Additionally, roles can be used in technical software designs in order to achieve modular structures that are easier to evolve. In both settings — modeling real world phenomena and technical designs — the strength of roles is rooted in their support for multiple, dynamic perspectives.

Still, neither delegation nor its use for roles has found its way into mainstream object-oriented languages. When searching for reasons for this reluctance, mentionings of the notion of “Object Schizophrenia” are found (mostly in informal communication), which is to describe a source for unnecessary complexity causing a well-known pattern of problems.

1.1 Emulating role instances and delegation

Another source for reluctance towards language support for roles and delegation is rooted in the belief that it suffices to emulate these concepts in the mainstream language of your choosing. Normally, I would like to take for granted that specialization mechanisms are sufficiently fundamental to programming that they deserve being supported by the programming language. However, with role instances and delegation some patterns of argumentation have an irrational air about them that deserves a closer analysis.

As for delegation the desired behavior can be emulated using some self-delegation programming patterns. Dynamic multiple roles, on the other hand, can be emulated, e.g., by introducing into a base object a collection typed field that dynamically stores all role-specific state.

So, self-delegation patterns require (a) that base classes are developed with explicit self arguments in all their methods in wise anticipation of any roles classes that might be added to the system lateron and (b) that programmers strictly follow the discipline as defined by the pattern.

Emulating dynamic multiple roles invariably involves the introduction of some surrogate key in order to retrieve specific

¹Even if static mechanisms for class composition would support repeatedly adding the same mixin to one class this composition would be statically fixed to provide, e.g., a fixed number of **salary** slots to the composed class. The implementation should, however, *not impose any upper bound* for such multiplicities.

role state from the mentioned collection. Again the programmer has to write more code (indirect lookup of role state) and must strictly follow a given discipline (like handling surrogate keys in a consistent way as to avoid accessing inexistent collection elements etc).

Without going into details of potential solutions, the net effect is as follows: compared to a “native” solution with explicit language support emulated solutions will always be more bloated and more fragile, because the programmer has to write additional boiler plate code and he must adhere to the pattern’s discipline. Both, adding stereotypical code and enforcing discipline are domains where compilers perform much better than programmers. If we reject language support for role instances with delegation we refuse any help that a better compiler can provide.

Still there’s nothing wrong with stating that emulation is possible and thus language support is not needed. Things start to get irrational if the same person would state that “Object Schizophrenia” causes problems and should thus be avoided *and* claim that language support isn’t needed because the issue can be handled using patterns. Such argumentation would come close to blaming the doctor after rejecting his medicine.

In this paper I elaborate how the Object Teams programming model *helps* to avoid problems that could potentially arise from dynamic multiple roles, without giving up on their flexibilities, thus arguing for the claim that language support for roles is the cure not the disease.

2. WHAT IS OBJECT SCHIZOPHRENIA?

The notion of “Object Schizophrenia” has been coined by Bill Harrison in the context of his work on Subject Oriented Programming (SOP). The introduction of this notion is not documented by any publication, but it first appeared in a set of web-pages [2] where certain deficiencies in standard design patterns were analyzed and contrasted with a superior solution using SOP. On these pages Object Schizophrenia was described like this:

“Object Schizophrenia results when the state and/or behavior of what is intended to appear as a single object are actually broken into several objects (each of which has its own object identity).”

When using the term Object Schizophrenia (OS in the sequel) one should keep in mind that this denotes a situation found in best-of-the-breed object-oriented designs – following the renowned GoF patterns. Moreover, Sekharaiah and Janika Ram – in their analysis of OS in the context of “is-role-of inheritance” [10] – made the valuable statement that OS is not a problem per se, but only certain symptoms resulting from OS potentially cause problems. Harrison et al [2] specify three kinds of problems:

1. **Broken delegation**, which describes that method calls are not always dispatched to the desired receiver if forwarding is used instead of delegation (with late binding of self).
2. **Broken assumptions**, which describes situations where incomplete encapsulation allows clients to invoke methods that are actually meant to be overridden.

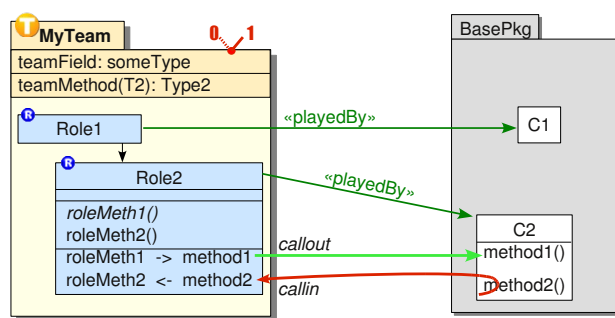


Figure 1: Fundamental Object Teams concepts

3. **Doppelgängers**, which describes a situation where the different identities of a split object cause duplicates in set-like data structures, because the conceptual unity of different parts of an object is not respected by object comparison.

In contrast to [2], which explains the three problems more by giving examples, my rephrased description lends itself to identifying three dimensions needing analysis: method call dispatch, encapsulation and object comparison.

3. COPING WITH OS

If OS is not a problem per se, how can we cope with it as to leverage the advantages of roles with delegation without being trapped by new problems? I will discuss this using the role-based approach Object Teams [8].

3.1 Object Teams in a nutshell

In Object Teams binding a role to a base is declared by a **playedBy** relation between this role class and the base class, as illustrated in Figure 1. Based on this declaration runtime role instances can be attached to a base instance. Behavioral details of this relation are declared using two kinds of method bindings: **callout** bindings defining that a given message should be forwarded from the role instance to its associated base instance. Conversely, a **callin** method binding has the effect that a role method may override a base method, which is realized by method call interception.

Furthermore, roles (classes/objects) are nested in a **team** (class/object). Teams define a border where data flows automatically translate role instances to base instances and vice versa: A base instance entering a team is **lifted** to an appropriate role instance (created on-demand and cached for later reuse). Data flows passing a role instance from a team to the outside apply **lowering** to actually pass the base instance to the sink of the data flow. These translations avoid a typical dilemma of role based languages: should roles be considered as sub-types of their base or perhaps even super-types? Either direction of sub-typing is desirable because one may want to provide role instance where a base instance is expected *or vice versa*. Object Teams avoids this dilemma by providing substitutability without sub-typing. By means of lowering a role can be substituted where a base is expected, and using lifting a base can be used as a substitute for an expected role. This creates a new kind of two-way substitutability which we call *translation polymorphism*[7]. Based

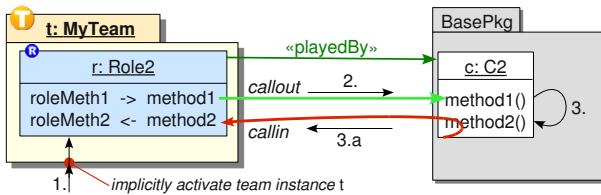


Figure 2: Callout, callin and implicit activation yield delegation with overriding

on the classes in Fig. 1 translation polymorphism renders the signatures in these method bindings compatible:

```
void roleMeth1(Role1 r) -> void method1(C1 c);
void roleMeth2(Role1 r) <- replace void method2(C1 c);
```

Here, the callout binding applies lowering of its argument, whereas the callin binding applies lifting.

Additionally, a team instance can be **activated** and deactivated to the effect that all callin bindings of its contained roles are enabled or disabled. In Fig. 1 this is symbolized by the red switch on top of `MyTeam`. Additional means for filtering method call interception are provided by way of so-called guard predicates.

Based on this core understanding of Object Teams I will now discuss how the problems of OS are approached.

3.2 Broken Delegation

When using only `playedBy` declarations and callout bindings, an Object Teams design would indeed exhibit plain forwarding and thus broken delegation. However, by the use of callin bindings a role can effectively override methods of its base thus turning forwarding into true delegation. To faithfully reconstruct the semantics of delegation the mode of *implicit team activation* should be used.² In this mode a team is activated whenever the control flow enters the team and deactivated when it leaves the team. This ensures that role methods are only invoked in a state where the team is active. Since callout bindings are treated as a special case of role methods, also role-to-base forwarding only ever happens when the enclosing team is active. Thus callin bindings are enabled during forwarding, yielding the desired semantics of delegation with overriding (see Fig. 2). Conversely, when sending a message directly to a base object, the team is not active, thus any callin bindings are without effect.

Sekharaiah and Janaki Ram [10] describe a situation where even a correct implementation of delegation would result in wrong message interpretations. The description of the example doesn't actually motivate why a certain message interpretation is expected and no indications are given how any programming language could "know" what style of dispatch to use when. Yet, Object Teams actually provides the necessary tools for achieving any desired outcome in the situation of chains of role-of-role as discussed in [10]: Since callout and callin binding are separately defined for individual methods, and since furthermore team activation can be

²Since version 1.3 of the OTJLD[5] this mode must be explicitly requested, see OTJLD §5.3(d).

configured individually even for specific team instances at runtime, the developer can freely combine plain forwarding and delegation. As a third option method call interception is supported (where a role intercepts external calls directed at its base), which will be discussed next.

3.3 Broken Assumptions

This kind of problems is related to the matter of method dispatch but here the focus is on invariants and encapsulation. Classes are intended as a means for encapsulation in the sense that only methods of the class can be used to modify the state of its objects and all these methods are obliged to maintain the class' invariants. In the case of delegation two classes compete regarding the responsibility for an object. For role objects, the conflict can be illustrated by describing base and role as two overlapping objects: The base object encapsulates its internal state, but the state of the role object comprises the role's fields *plus* the fields of the associated base.³ Role invariants should be able to reason about this composed state.

In Object Teams broken assumptions could result if a role attempts to override an invariant of its base class: If a method is invoked directly on an instance of this base class, and if the role has no chance to override this call, executing the base method only may result in the role's invariant being broken. However, this situation only arises if the corresponding team is not active, since in the presence of an active team the role can effectively intercept such method call using a callin binding.

Object Teams does not generally decide which class (role or base) is stronger in the competition of enforcing its particular invariants. This also reflects the understanding that considerations regarding delegation and invariants could potentially conflict with each other – and domain knowledge will be needed for finding a particular non-conflicting solution. In particular it is a matter of the composed program to decide whether or not role behavior should override base behavior. By globally activating a team instance (perhaps at application launch time) callin bindings are enabled by default to the effect that role behavior overrides base behavior, thus role invariants are maintained. Still, by fine-tuning the team's activation even during runtime it is possible to flip the switch towards the base class, meaning that the base class' sole responsibility for its objects is re-established. Conversely, if maintaining the role's invariants is crucial, the enclosing team could override its `deactivate()` methods to ensure that its callin bindings are always enabled.

Given these conceptual tools for tuning the effectiveness of callin bindings I argue that any remaining conflicts, where the desired dispatch cannot be achieved with the given means, most likely hint at inconsistent requirements that are impossible to get right in the program. Moreover, by making different perspectives explicit in the program roles should significantly facilitate tracing conflicts in the program back to their conflicting requirements.

³A role in Object Teams can declare access to its base's fields by callout-to-field bindings.

3.4 Doppelgänger

The doppelgänger problem occurs when the comparison of a base and its role yields **false** where actually **true** is expected. The problem is not normally relevant in Object Teams, because a team defines a self-contained world. Normally, roles are only referenced within the context of their enclosing team. Conversely, the team shouldn't directly refer to any base objects for which it also has a role. The compiler can be configured to enforce the use of so-called base imports, which ensure the strict separation of scopes which turns every explicit mentioning of a bound base class within a team into a compile time error.⁴ When using the team boundary as the border between roles and bases, the translations of “lifting” and “lowering” are key for any data flows between the two worlds. The language furthermore helps maintaining this separation by automatically inserting these translations where needed. This solution is illustrated by the following example (in the vein of [2]), introducing a global set of objects requiring notification when the system shuts down:

```
public class SomeBase {
    public static Set<SomeBase> toNotify; // can't contain roles
    ...
}
public team class SomeTeam {
    protected class SomeRole playedBy SomeBase { ... }
    void register(SomeRole r) {
        SomeBase.toNotify.add(r); // will automatically lower r
    }
}
```

Here doppelgänger in the set `toNotify` are avoided by lowering translations that are inserted by the compiler to the effect that only base instances are ever added to the set.

Still two situations exist that could cause doppelgänger even in Object Teams: polymorphic references (where the compiler cannot distinguish roles from bases) and public roles, that are *intended* for reference outside the team.

To illustrate the first issue consider a changed declaration in the above example:

```
public class SomeBase {
    public static Set<Object> toNotify; // shouldn't contain roles
    ...
}
```

Such designs arise in real world situations where it might be difficult to find a suitable type `SomeBase`. Especially infrastructure code typically uses type `Object` for collection elements. Now it can indeed happen that a role *and* its base individually register, causing doppelgänger in the set. In Object Teams it is still possible to statically distinguish base objects typed to `Object` from roles by declaring the roles as subclasses of the predefined class `Team.Confined` which does *not* extend `Object`. This role class is intended for strict encapsulation by avoiding widening to type `Object` (see [6]). As a side-effect subclasses of `Confined` require lowering in order to be assigned to a variable of type `Object`, which resolves the doppelgänger problem even for `Set<Object>`.

⁴This enforcement cannot prevent references to base instances via an unbound super type of the base, see below.

Furthermore, if lowering cannot be used to *avoid* comparing roles to bases, and if roles are *intended* to be referenced outside their team, the comparison can be made using different operators/methods. Various methods like `roleEQ` (forcing lowering before comparison) and `deepRoleEQ` (forcing multiple levels of lowering before comparison) can trivially be added to the Object Teams API to support relaxed forms of comparison that consider roles as identical to their bases. Although method `roleEQ` has been proposed in previous publications, its implementation was never published, simply because the need hasn't yet occurred in real life Object Teams code.

Aside from polymorphic collections there are other situations that could potentially suffer from OS in ways related to doppelgänger. As pointed out in [9] using an object as a monitor for synchronization depends on object identity⁵. Here OS can indeed produce undesired results, which can again be avoided by lowering. Regarding methods like `wait` a simple callout binding ensures that the base object is used as the monitor. Regarding the `synchronized` keyword support for the following idiom is planned:

```
synchronized(base) { /* code */ }
```

Although this is not implemented at the time of this writing, implementation is trivial and the semantics will simply extend the advantages of lowering to apply to synchronization, too.

4. CONCLUSION

Language mechanisms based on delegation obviously provide greater flexibility to the developer. Specifically language support for roles benefits from the added dynamism and unbounded multiplicities of instance-based⁶ role-binding. At a naive look such approaches seem to suffer from Object Schizophrenia (OS). At a closer look OS can actually be observed independently of the language used. In fact the standard object-oriented design patterns are full of examples for OS, because the flexibility these patterns are designed for cannot easily be achieved without splitting conceptual entities into several implementation objects.

I have shown how the Object Teams approach copes with OS as to avoid the problems observed in the context of OS: broken delegation, broken assumption and doppelgänger. Object Teams solves the problems by the following key concepts:

1. Separate mechanisms are provided for forwarding (callout) and method call interception (callin), from which a wide range of dispatch patterns can be constructed, to always create the desired behavior.
2. Team activation (and guard predicates) provide means for controlling the effect of callin bindings at runtime.

⁵Since OT/J is developed as an extension of Java, replacing object monitors with other synchronization mechanisms is not an option for this specific language. Other incarnations of the Object Teams model, which are not based on Java, may not have this issue.

⁶As opposed to static class composition

3. Roles are encapsulated in a team, which generally avoids comparing roles to bases.
4. Roles are translated to bases (lowering) and bases are translated to roles (lifting) to enable data flows between the inside of a team and the outside without mixing objects from both worlds. The compiler automatically inserts these translation when a data flow crosses the boundary of a team.
5. Various methods for instance comparison have been sketched which could be used to hide the different identities when comparing instances.

The above solutions to the problems of OS demonstrate that delegation should not be regarded in isolation. Firstly, roles in Object Teams depend on a context reified as an enclosing team. It is the team boundary that helps to avoid confusion that might occur when roles would be compared to bases. Furthermore, the compiler heavily relies on a smooth integration of roles and teams with generics. Only with the additional type information from generic types the compiler can effectively deduce where lifting or lowering is needed.

Even if Object Teams programs are not free of OS, they surely needn't suffer from it. Object Teams provides the full flexibility of an instance-based approach without the problems of OS. This means that regarding OS, Object Teams is superior to mainstream object-oriented languages which require the use of design patterns for achieving desired flexibility. Object Teams has this flexibility built into the language. Only so the language can also provide the means to cope with those problems that could follow from OS. At this point it seems desirable to avoid the term Object Schizophrenia in favor of something like "split objects" to avoid the connotation of an illness, which it no longer need be – *if* using appropriate conceptual tools.

5. REFERENCES

- [1] M. Baldoni, G. Boella, and L. van der Torre. Roles as a coordination construct: introducing powerjava. In *Proceedings of MTCoord05*, 2005.
- [2] Bill Harrison et al. Subject-oriented programming vs. design patterns. <http://www.research.ibm.com/sop> - archived as of May 1997.
- [3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley, 1997.
- [4] Kasper Bilsted Graversen. *The nature of roles—A taxonomic analysis of roles as a language construct*. PhD thesis, IT University of Copenhagen, Denmark, 2006.
- [5] S. Herrmann, C. Hundt, and M. Mosconi. ObjectTeams/Java Language Definition current version (OTJLD). <http://www.ObjectTeams.org/def/>, 2002–2010.
- [6] Stephan Herrmann. Confinement and representation encapsulation in object teams. Technical Report 2004/06, Technical University Berlin, 2004.
- [7] Stephan Herrmann. Translation polymorphism in Object Teams. Technical Report 2004/05, Technical University Berlin, 2004.
- [8] Stephan Herrmann. A precise model for contextual roles: The programming language ObjectTeams/Java. *Journal on Applied Ontology*, 2(2):181–207, 2007.
- [9] H. Kegel and F. Steimann. Systematically refactoring inheritance to delegation in java. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 431–440, New York, NY, USA, 2008. ACM.
- [10] K. C. Sekharaiah and D. Janaki Ram. Object schizophrenia problem in modeling is-role-of inheritance. In *Proc. of Inheritance Workshop at ECOOP*, Málaga, 2002.
- [11] L. A. Stein, H. Lieberman, and D. Ungar. A shared view of sharing: The Treaty of Orlando. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading (MA), USA, 1989.
- [12] T. Tamai, N. Ubayashi, and R. Ichiyama. An adaptive object model with dynamic role binding. In *Proc. International Conference on Software Engineering (ICSE2005)*, pages 166–175, May 2005.