Implementing design patterns in Object Teams

Journal:	*Software: Practice and Experience*
Manuscript ID:	Draft
Wiley - Manuscript type:	
Date Submitted by the Author:	n/a
Complete List of Authors:	Monteiro, Miguel; CITI, FCT/UNL, Departamento de Informática Gomes, João; CITI, FCT/UNL, Departamento de Informática
Keywords:	aspect-oriented programming, modularity, design patterns, reusability, extensibility

Implementing design patterns in Object Teams

Miguel P. Monteiro and João Gomes
CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa 2829-516 Caparica PORTUGAL
mmonteiro@di.fct.unl.pt, clean_you@hotmail.com

Summary
This paper presents a study of the support for modularity and reusability of Object Teams, an aspect-oriented backwards compatible extension to Java. The study is based on implementations in Object Teams of two complete collections of the Gang-of-Four design patterns. An analysis of the implementations is provided, in terms of the advantages Object Teams brings relative to Java in terms of modularity and module composition. Object Teams provides direct language support for two of the creational patterns and in general yields pattern implementations that are more modular and more flexibly extensible than the Java counterparts. A detailed comparison with AspectJ is also provided, in terms of five modularity properties: locality, reusability, composition transparency, (un)pluggability and extensibility. AspectJ and Object Teams yield comparable results in terms of the former four properties, but Object Teams is clearly superior in terms of instance-level processing and flexible extensibility of the modules produced.

Keywords: aspect-oriented programming, modularity, design patterns, reusability, extensibility.

1. Introduction

Aspect-oriented Programming (AOP) is an emerging programming model primarily focused on the modularization of crosscutting concerns [11, 34]. AOP is undergoing maturation and many aspect-oriented programming languages (AOPLs) have been proposed [7]. Most such languages are backwards-compatible extensions to existing languages, among which Java features prominently.
Given the wide variety of proposals for AOPLs, it would be highly desirable that studies were made reporting on the various strengths and limitations of these new languages. However, studies comparing aspect-oriented languages are almost non-existent. Most existing reports on the use of AOPLs are by the creators of the given language. There is a dearth of studies by independent authors for most AOPLs. Though a few independent studies comparing representatives of the object-oriented programming (OOP) and AOP approaches were carried out, they are geared for comparisons *across* paradigms [19, 8, 22]. There have been very few examples reporting on comparisons *between* AOPLs [59, 55]. This paper contributes to fill this gap by presenting a comparative study of two aspect-oriented languages: AspectJ – currently the best-known AOPL [33, 37, 9] – and Object Teams/Java (OT/J) [29, 27, 24, 3, 2].
We can observe a significant variety in AOPLs as regards language features and supported composition mechanisms, even in languages that extend a common base language. For instance, both AspectJ and OT/J are backwards-compatible extensions to Java and are both labelled as aspect-oriented programming languages. Nonetheless, the mechanisms provided by the two languages are markedly different. Such variety reinforces the motivation for studying the relative advantages and disadvantages of the various AOPLs available, as well as reports on the actual results and applicability of the new language constructs they embody.
The approach taken in this study is to compare sets of functionally equivalent versions of a number of examples, coded in the two subject AOPLs and analyse the results in light of a set of modularity properties. The examples making the subject for comparison comprise two complete collections of implementations of all the Gang-of-Four (GoF) design patterns [18]. Design patterns were chosen as a case study on account of the variety and richness of situations they present even with relatively simple examples. Catalogues of design patterns serve as a distillation of issues pertaining to separation of concerns that often arise in real systems, as well as commonly practised solutions for those issues. Many of the issues are illustrations of the effects obtained from well-known mechanisms and features found in the OOP language space but that are not necessarily provided by Java.

The potential of design patterns for illustrating and/or assessing the relative advantages of an AOPL was explored in the past [1, 46, 23, 31, 50, 38], though few previous studies involve the full GoF collection of 23 patterns. To our knowledge, there two systematic studies involving all GoF patterns: those by Hannemann and Kiczales using AspectJ [23], and by Rajan using EOS [50]. Of these studies, only the former made the code examples publicly available, which were subsequently used as a basis for independent research [45, 19, 8]. For the present study, we developed in OT/J two complete Java repositories of implementations of the GoF patterns[1]. A comparison is made with Java and AspectJ based on functionally equivalent examples, with studies and comparisons involving other AOPLs left for future work.

Previous systematic studies based on GoF patterns are based on examples created by people with a stake on the language under study [23, 50], which makes them vulnerable to suspicions of bias. To avoid that shortcoming as much as possible, we avoided creating our own examples. Instead, the OT/J examples are re-implementations of two collections of Java examples, freely available on the Web. Both collections existed before the setup of the present study and by people that are independent of the authors of this paper and to our knowledge, independent of the languages under study as well. The newly developed examples in OT/J are functionally equivalent to the corresponding original Java examples.

The rest of this paper is structured as follows. Section 2 provides a short overview of the main concepts found in aspect-oriented languages. It is primarily focused on Object Teams, as AspectJ is presently is well known, having been presented, illustrated and documented in many papers, tutorials and books (e.g., [37, 9]). Section 3 provides an overview of the implementations in OT/J, highlighting cases that illustrate composition and modularization effects absent from plain Java. Section 4 compares the results obtained with OT/J with those obtained using the AspectJ collection by Hannemann and Kiczales, using the examples common to both collections. The comparison is based on a number of modularity properties of the implementations obtained.

2. Aspect-oriented programming languages

Ideally, each concern in a software system would be cleanly encapsulated in its own unit of modularity, yielding a one-to-one mapping between concerns and modules. However, in OOP systems it often happens that certain kinds of concern, such as persistence, exception handling, logging and distribution, are scattered across the various units of modularity, i.e., class modules. Traditional OOP mechanisms are unable to localise the source code related to such concerns within a single class. Consequently, the representation of such concerns takes the form of multiple, small code fragments, scattered throughout the class modules of the system, a phenomenon usually referred as *code scattering*. Kiczales *et al*. [34] refer to the concerns that give rise to code scattering as *crosscutting concerns* (CCCs). In addition, the various code fragments related to CCCs tend to be mixed with the code related to the primary functionality of the system's existing class modules, worsening the comprehensibility and ease of evolution of all concerns involved. This negative effect is dubbed *code tangling* [34]. The OOP implementations of a number of design patterns are examples of CCCs, as the various pattern roles span multiple classes and thus crosscut them [23]. *Aspect-oriented programming* (AOP) [11, 34, 15] provides constructs explicitly built to localise source code related to CCCs in their own units of modularity – which we call *aspect modules* or simply *aspects* – thus eliminating the code scattering and tangling symptoms.

2.1. Quantification, obliviousness and pointcut fragility

Regardless of differences between AOPLs, all share a few common characteristics and concepts, highlighted next. The primary concepts and commonalities were proposed in a widely cited proposal by Filman and Friedman [16] on how to distinguish the generality of AOPLs from languages that instantiate other models or paradigms. Filman and Friedman propose *quantification* and *obliviousness* as the two distinguishing characteristics of AOPLs. *Quantification* is the ability to specify assertions over execution events of a program, so that the intended behaviour of *aspect modules* is implicitly called upon reaching any of the specified events. By "implicitly", we mean that the remainder of the system does not need to have explicit mentions to the aspect, e.g., method calls. Thus, AOPLs support the *obliviousness* property, i.e., the possibility of existing programs to be subject to composition with aspect modules without the need for the programs to undergo invasive changes on their source code. Various degrees of obliviousness can be distinguished, namely *programmer obliviousness*, in

[1] Full sources are available for download as an Eclipse/OTDT project at http://ctp.di.fct.unl.pt/~mpm/AOLA/OTJGoF4SPE.zip

which a given software system is subject to aspect compositions without having been specifically prepared by the programmers, and *code obliviousness* – also known as non-invasiveness [61] – in which a system is free from dependencies at the source code level but whose designers may have taken the future composition of aspect modules into consideration – see Filman and Friedman [16] and Sullivan *et al.* [57] for more in-depth analyses. This paper uses the term *obliviousness* to mean code obliviousness.

It is worth noting that the more powerful the quantification capabilities of a given AOPL are, the greater is its scope to achieve full code obliviousness. However, powerful quantification capabilities also usually go hand in hand with *the fragile pointcut problem* [35]. In existing AspectJ-like AOPLs, quantification mechanisms – which often take the form of AspectJ-like pointcuts – tend to directly depend on implementation details of a base system, such as the name or the type of an object field. Depending on the style of programming, seemingly trivial changes in the code base, such as renaming a method or a field – even a field with private visibility – can have an impact of the effects of the quantification logic of the aspect module. The more expressive the quantification mechanisms of a given AOPL are, the more fragile its aspect modules tend to be. In consequence, users of AOPLs must take special care to attain styles of programming, particularly of pointcut expressions, that minimize the potential for breaking the logic of aspect modules when the base system evolves.

2.2. The Object Teams language

Unlike AspectJ, OT/J is based on an underlying theoretical model that existed prior to the design of the language: *role modeling* [52, 24], of which OT/J strives to provide direct language support. OT/J does not rely on an explicit notion of joinpoint and does not provide constructs for specifying or capturing joinpoints. Instead, the OT/J approach can be characterized by multiple dimensions of polymorphism and a flexible approach to manipulate objects. A form of quantification is still discernible though it features less prominently than in AspectJ.

Teams and roles as aspect modules

OT/J extends Java with a new kind of module, the *team*, which roughly corresponds to the aspect modules of AspectJ. However, team modules can be freely instantiated using new just like class modules and unlike AspectJ aspects.

A notable property of team modules is the ability to enclose a special kind of inner classes, the *roles*, that represent the internal concepts of a collaboration of objects (see the examples from Figure 1 and Figure 2). Teams provide the context within which the collaboration takes place, which can include state and behaviour pertaining to the context of the collaboration.

Virtual classes and family polymorphism

Roles are *virtual classes* [40], i.e., classes that are members of objects the same way as fields and methods. Virtual classes can be overridden and subject to dynamic dispatch similarly to method calls. With virtual classes, explicit references to class names, including expressions using the new keyword to instantiate the class, are subject to dynamic dispatch. This is in contrast to Java, in which an expression using new always refers to the exact same class. The type system of OT/J also supports *family polymorphism* [12], i.e., the ability to group a set of (virtual) classes into a larger class (the team) such that consistency between all member classes and their instances can be enforced by the type system. Instances of teams are used as type anchors to ensure that role instances from different teams do not get mixed.

Virtual classes combine with family polymorphism to yield highly extensible modules [13]. If a piece of code referring to a role class – possibly with new expressions – is inherited by a sub-team that also overrides the definition of that role class, the inherited piece of code will use the overridden role class *from the sub-team*. In consequence, the name of a class comprises a new kind of *variation point* not found in plain Java systems. These features enable sub-teams to adapt the role hierarchy inherited from its super-team, e.g., by inserting new intermediate role classes into the inherited hierarchy. Thus, OT/J supports a form of *higher-order hierarchies* [13], i.e., the possibility to build a class hierarchy incrementally, by treating hierarchies of teams as hierarchies of hierarchies (of roles). Also note that roles can themselves be teams, meaning that this approach can be extended to more than two levels. However, we felt no need to use such features in the small examples analysed in this paper.

Implicit role inheritance

As a result of OT/J's support for virtual classes and family polymorphism, team modules are extensible along more dimensions than is possible with plain Java classes. In addition to inheritance between teams, to which all possibilities of traditional class inheritance apply, role classes are also subject to inheritance in several ways. Inheritance can be used between roles within a team with the same flexibility as traditional classes within packages, e.g., by using the extends clause. For example, a role can extend another role from the same team, extend a differently named role from a super-team of the enclosing team, or inherit from a role of the same name from the enclosing team's super-team. The latter form is known as *implicit inheritance* and differs from the other forms of inheritance in that it is based on name equality rather than on an explicit extends clause. In all cases of extension, there is also the option of overriding members of roles and make late binding work as in mainstream OOP.

Figure 1 presents the small example of a team hierarchy, using the UFA notation, an extension to UML class diagrams proposed by Herrmann [25]. In the example, team AntiqueAuctionHouse acquires roles Bidder and Auctioneer through implicit inheritance (shown in grey in Figure 1) from super-team AuctionHouse. There is also the option to implicitly override them by defining new roles with the same names. Sub-team AntiqueAuctionHouse can also declare new roles: AntiqueSeller and AntiqueAuctionItem inherit from role Seller and AuctionItem respectively, through explicit inheritance through extends clauses. All advantages of multiple dimensions of inheritance and higher-order hierarchies apply to examples such as these.

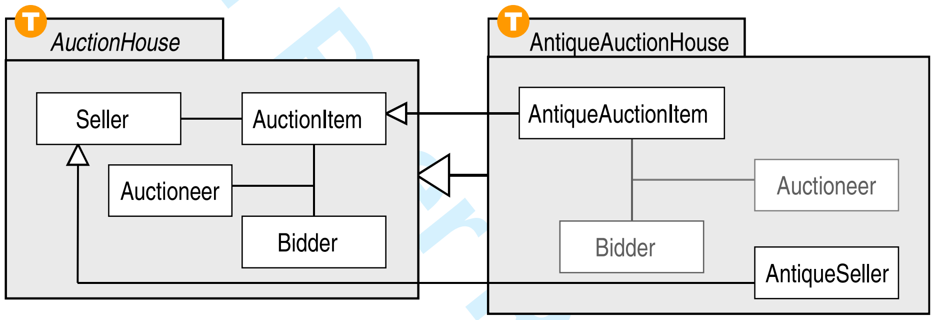


Figure 1. Example of extensible collaborations through team modules.

The role playing relation to bind between roles and bases

A role class has the option to declare a *role playing relation* to some class – usually a plain Java class, but can also be a team. This way, the language expresses a relation between the internal concept of an object collaboration and the modules specific to a given application. This is expressed through an explicit playedBy declaration placed in the header of the role and referring to a specific class, which becomes its *base* – see Listing 1 and Figure 2. Role playing relations are inherited by sub-roles.

```
public team class AuctionHouse {
  //...
  protected class AuctionItem playedBy ValuablePainting {
    //...
  }
}
```

Listing 1. Example of a binding between a role (AuctionItem) and a base class (ValuablePainting).

The playedBy declaration has no effect on its own, but it is the basis for two kinds of bindings between members of roles and members of bases, described below. A role also can be *unbound* (i.e., without a playedBy clause), which is often the case of roles within general, possibily abstract, teams, or when the role represents a concept within the collaboration that has no counterpart outside the team. Though each role can specialise only one base, multiple roles can specialise the same object simultaneously. A role playing relation can propagate through both explicit and implicit role inheritance and be refined, provided that does not violate the general typing rules. Due to technical reasons, role playing presently cannot be applied to proprietary classes such as those from Java standard APIs. For a systematic description of all supported variants of these bindings, see [24, 29].

The role playing relation bears some resemblances to inheritance in that also comprises a separate dimension of polymorphism. Their instances are substitutable under certain conditions, which are outlined next.

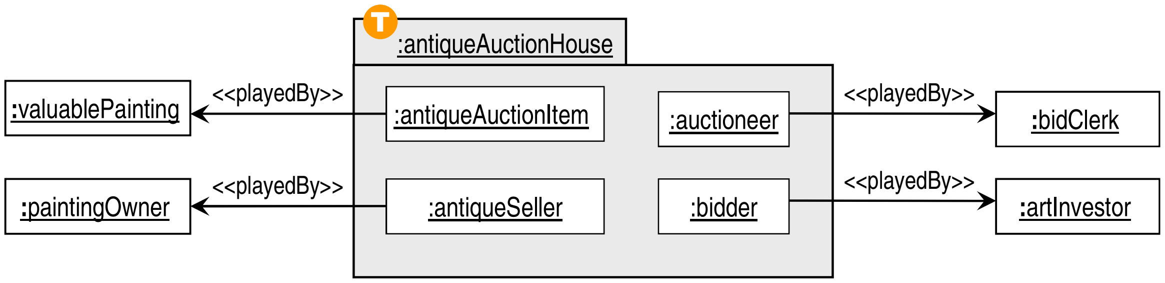


Figure 2. Concrete example of a team and respective roles bound to concrete objects.

Role playing differs from inheritance in two important points. First, role playing is more dynamic, working at the instance level rather than at the class level. Roles can be added and removed to specialise object instances at any moment during program execution. Second, role playing makes a distinction between *acquisition* of members and *overriding* of member definitions, treating them separately and on a case-by-case basis. In traditional inheritance, a subclass *acquires* all super-class members without exception. This is essential to ensure the full substitutability prescribed by type-subtype relationships. However, full substitutability is not required in role-playing relations. A role acquires members of its base class *selectively* and role instances and base instances are meant to be substitutable only in relation to code referring to those members. Each individual member acquisition requires an explicit declaration.

As regards overriding of super-class members, traditional inheritance is also constrained to cases of name/ signature equality. With few exceptions (e.g., Eiffel) OOP languages do not generally allow for a subclass to map an overriding method to a name or signature different from that of its super-class. By contrast, OT/J supports clauses within roles that specify mappings between role members and base members with different names/signatures, as well as specifying the necessary adaptations, e.g., order of parameters, type conversions, or some short glue-code expression.

OT/J provides two kinds of binding between roles and bases to support the aforementioned mappings: *callouts* and *callins*, where the "in" and "out" should be seen from the perspective of the role. A *callout binding* permits a role instance to acquire a member not available locally, i.e., in the base instance – the role object is said to "call out" to the base object. Callout bindings (henceforth just "callouts") are the means to integrate roles with specific base classes from a given system. Often, the roles declaring a playedBy relation and callouts are concrete sub-classes of an abstract super-role – probably declared in a different, abstract team. A role must declare all its members, but has the option of leaving some of them as an abstract declaration and acquire the corresponding concrete definitions from the base class. If all abstract declarations receive a concrete definition, the role can still be concrete and can be instantiated. Thus, we can discern similarities between callouts and the way with which an abstract class acquires members from a concrete sub-class (in e.g., the *Template Method* pattern [18]), so that all details in code that refer to the abstract class are resolved during compilation. An important difference between inheritance and role playing is that, by default, a role acquires nothing from its base: all acquisitions must be explicit, by means of callouts[2].

A callout is defined within a role class in a purely declarative style:

```
protected SomeRole playedBy SomeBase {
  //...
  type roleMethod (parameters) -> type baseMethod (parameters)
  //...
}
```

[2] The OT/J environment permits to configure the compiler to support *inferred callouts*, so that it will not issue an error when finding an undeclared reference to a base member. Instead, the compiler infers the corresponding declaration. However, in general that is not the recommended style.

The above example pertains to the straightforward case in which the method signatures and result types of role and base are compatible. OT/J also supports *parameter mappings* to deal with other, more complex cases.

Quantification in Object Teams: callin bindings

Callouts are both a means of communication from role to base and the means for roles to acquire the definitions of base members – methods *and* fields. OT/J also supports communication in the opposite direction, through *callin bindings* (henceforth just "callins"). Callins are also the means through which a role *overrides* base behaviour. Callins are denoted in a style very similar to that of callouts, only in the opposite direction. To differentiate from the callout binding style, callins are denoted by a reverse arrow <– because the role object instructs its base to (implicitly) "call into" the role.

Callins comprise a form of quantification. Roles can specify that its methods be implicitly called whenever certain events from the associated base class take place. This effect is roughly equivalent to AspectJ pointcuts, though admittedly with a narrower range of applicability. The execution events supported by OT/J are the execution of methods and access to fields. As in AspectJ, it is possible to distinguish between accesses for reading (e.g., using it in an expression or as the right side of an assignment) and for writing (i.e., the left side of an assignment). Though these capabilities for crosscutting composition are significantly narrower when compared to AspectJ, they also avoid a number of shortcomings known to mar AspectJ-like languages, namely the fragile pointcut problem [35]. Crucially, OT/J quantification mechanisms do not include the use of wildcards, thus avoiding a number of problems associated to AspectJ-style pointcuts, namely the technical difficulties in providing refactoring support [58]. Automatic support for a number of refactorings has been available in the OT/J environment for years.

Callins are the (rough) equivalent to AspectJ advice. As in AspectJ, callins can execute before, after or instead of the captured event. Thus, role methods bound through callins must have one of three modifiers before, after, replace, corresponding to before, after, around advice respectively[3].

```
roleMethod <- after baseMethod;
```

A new modifier callin is required for each role method that overrides a base method, i.e., bound with the replace modifier. Similarly to proceed calls found in around advice from AspectJ aspects, the body of a callin method has the option to invoke the base method using a *base call*, using the base keyword:

```
callin type roleMethod(parameters) {
  //...
  … = base.roleMethod();
  //...
}
type roleMethod() <- replace type baseMethod(parameters);
```

The method name used in the base call is that of *role method* despite referring to the base method. This is so that the namespace of the role is kept self-contained, which is essential to ensure a complete independence of role code from base code.

For callins to take effect, the enclosing team instance must be *activated*, for which a few methods are available to all teams. OT/J also provides additional control for callin activation in the form of *guard predicates* comprising declarative clauses used to restrict the effect of callins to specific situations. Presently, OT/J supports the following levels of control: callin binding, role method, role and team module.

Finally, OT/J provides a declarative clause (precedence) to control the order in which callin bindings are triggered if multiple callins from the same team refer to the same base method (or field) and have the same callin modifier (before, after or replace).

Translation polymorphism between roles and bases

Role playing introduces still another form of polymorphism, *translation polymorphism* [28]. Instances of role classes and base classes comprise separate object identities but in some circumstances their instances can be interchanged in ways that are transparent to the type system checker. When the flow of control crosses the base-

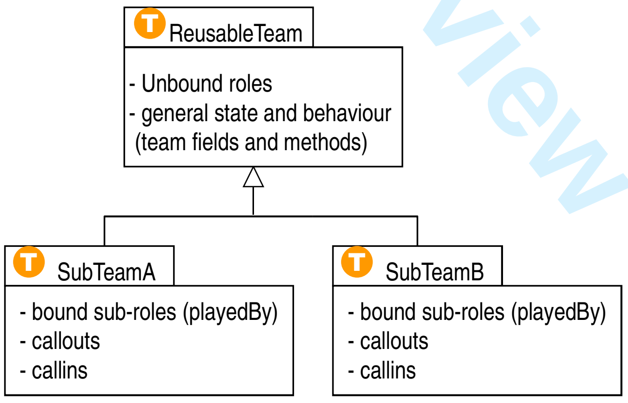
[3] This callin example uses the short notation with just the method names and without signatures. Both short and long notations with signatures are available for callouts and callins.

team boundary in the base-role direction – normally by way of callin – a base object is automatically replaced by the role instance that corresponds to the role's enclosing team. This replacement is called *lifting*. Likewise, when a role instance crosses the team-base boundary, it is automatically replaced by its corresponding base instance, a translation called *lowering*. Herrmann describes the type rules of these translations in detail and claims that this feature supports the integration of multiple, structurally mismatching hierarchies [28].
OT/J supports one other form of lifting – called *declared lifting* – in which the translation from the base type to the role type is made explicit the signatures of non-static team-level methods. Declared lifting is meant to support mappings of base objects to their role instances while avoiding exposing roles to details pertaining to the outside of the team boundaries. This feature proved very important for implementing the *Visitor* pattern (section 3.2). Next follows a simple example of declared lifting:

```
public team class SomeTeam {
  public void roleMethod(BaseClass as RoleClass parameter) { statements }
}
```

Explicit instantiation of roles
In the majority of cases, bound roles are created implicitly, whenever the program's control flow crosses the boundary of the team. Thus, the need to explicitly instantiate a bound role is not felt often. However, a few exceptions are bound to occur occasionally. To deal with such cases, OT/J provides *lifting constructors* for bound roles, which are default constructors that take exactly one argument of the type of the declared base class (after playedBy). These are generated by default in most circumstances. A few examples of the use of lifting constructor were included in our repository of implementations, which served to control explicitly the exact moment when the role associated to a base class is instantiated and enable its team to maintain an explicit reference to it.

Separation of reusable and specific parts: team inheritance
The approach to structuring team logic into (a) reusable and (b) case-specific parts follows the same line as AspectJ, and indeed class-based languages generally. Teams are usually partitioned into a reusable super-team and case-specific sub-teams (Figure 3). The primary difference between OT/J and AspectJ is that – unlike with AspectJ aspect modules – teams are not more restricted in the use of inheritance than is the case with plain Java. In AspectJ, an aspect cannot extend a concrete aspect, with the consequence that traditional forms of polymorphism with respect to aspects are blocked. OT/J imposes no such limitations: any concrete team or role remains open for further extensions through inheritance.


ReusableTeam
- Unbound roles
- general state and behaviour
(team fields and methods)
SubTeamA
- bound sub-roles (playedBy)
- callouts
- callins
SubTeamB
- bound sub-roles (playedBy)
- callouts
- callins
Figure 3. Partition between a reusable team and case-specific sub-teams.

Object Teams idioms
A collection of idioms is being developed that, like the GoF patterns, aims to teach techniques for achieving certain composition effects or solving certain design problems. The primary difference between true design patterns is that they are specific to the language features of OT/J, hence the reason for dubbing them "idioms". A number of OT/J implementations described in this paper use some of these idioms, for which we next provide a short overview:

- *Transparent Role* is the enabling of a role instance to be used outside its team without creating specific dependencies. The technique consists in making the role and base implement a common Java interface and make clients access the role only through the interface. This is by far the idiom most often used in our repository.
- *Object Registration* is about restricting the effect of a role to a subset of the base objects. When no explicit case is programmed, a role is created for every base instance that crosses the team boundary in a program's control flow. This idiom is based on the use of a guard predicate at the role level to restrict activation of the callin to a set of registered base instances. The team uses a team method to explicitly register all intended base instances. Other instances do not trigger the callins. Section 3.6 describes one use of the idiom in detail, in connection with the *Decorator* pattern.
- *Double Dispatch* is about emulating double dispatch and corresponds to the technique also employed to implement the *Visitor* pattern[4]. See the section 3.2 on the implementations for *Visitor* for more details.

Communication between roles and the outside of the team
In the general case, roles are meant to be part of the private implementation of the team (or a hierarchy of teams) and are not supposed to be accessed directly by clients. One issue felt in a number of pattern implementations was how to expose role functionality to the outside of the team instance. Aside from looking for a design in which the exposing of roles is not needed (recommended), one approach to deal with this issue is to use team-level methods to forward calls from outside the team to the role instance within the team. This approach can have the drawback of being rather verbose in some cases, as it entails creating a different team-level method for each role method involved. Note that in such cases, it is desirable that the team-level methods use declared lifting to map base instances to role instances. This way, the need for explicit team-level fields to support such mappings is avoided. A second approach, attractive for the cases in which lots of methods must be exposed, is to use the *Transparent Role* idiom.

3. The implementations

This section describes the implementations produced in OT/J, highlighting the cases in which compositional effects that cannot be obtained with plain Java were obtained. The presentation of the various implementations is organized into groups according to these compositional effects.

The subject material of the study
To ensure consistency across an entire collection of the patterns, we strived to redevelop complete Java repositories rather than picking ad hoc examples from many different sources. The two repositories on which the OT/J examples are based are the following:

- The collection by Hannemann and Kiczales (HK)[5], selected for to the availability of functionally equivalent implementations in both Java and AspectJ [23]. The basis for the OT/J implementations were the Java versions, though the availability of AspectJ versions was an important factor in deciding to use this repository, which provides material for a comparison between OT/J and AspectJ.
- The collection by Cooper[6] [10], because almost all implementations are based on graphical classes from the standard Java *swing* library. The license under which classes from Java standard APIs are publicly available precludes modifying proprietary bytecodes, which prohibits certain kinds of composition as usually supported by AOPLs – AspectJ and OT/J included. In turn, circumventing such problems gives rise to implementation hurdles of their own. Thus, assessing reusability and composability using examples rich in *swing* classes makes for rather stringent criteria, and had an impact on the results obtained. However, we believe this approach results in a slightly more interesting subject for analysis.

[4] This idiom is more recent than the others and is not found in the original Object Teams site. It was recently added to the *eclipsepedia* and is linked through the new site at the eclipse portal.
[5] The original project for eclipse with the AJDT plugin for AspectJ is available at: http://hannemann.pbworks.com/f/gof1.11.zip
[6] The refactored version used as a basis for this work is available as an eclipse project at http://ctp.di.fct.unl.pt/~mpm/AOLA/JavaGoFJCooper.zip

Terminology
Each design pattern prescribes *roles* for the concrete objects and/or modules (e.g., classes and Java interfaces) that participate in the pattern. Thus, the term *role* is often used in the context of both design patterns and OT/J. To avoid confusion about uses of the term, this section uses *role* to refer to the role classes as supported by OT/J and *participant* to denote a role in a pattern specified in that pattern's documentation [18]).
Throughout the descriptions, we use the term *scenario* to refer to the idea or metaphor used to set up the classes of which a given pattern example is comprised. For instance, Cooper's scenario for *Visitor* is based on the idea of computing the vacation days to which employees are entitled. A special kind of employee, the boss, has vacation days following special rules that must be computed differently from other employees. The scenario for *Visitor* by Hannemann and Kiczales is very different, being about traversing a simple binary tree whose leaf nodes hold an integer value. Scenarios for any given pattern can be extremely varied, ultimately depending mostly on the imagination of its authors. The collections of Java implementations by Cooper and Hannemann and Kiczales comprise one scenario for each of the GoF patterns.
We use the term *example* to refer to a specific implementation in a given language of a scenario for a pattern. Naturally, each specific example is also an instance of a specific pattern. The scenario for a pattern gives rise to at least one example for each different language. The collection by Hannemann and Kiczales comprises two examples for each pattern, as each scenario is implemented in Java and AspectJ. Even with a single language, a given scenario can give rise to more than one example if it is possible to implement that scenario in multiple, different ways in the given language.
The distinction between scenario and example is useful because we found scope for different ways to implement some scenarios in OT/J. For instance, two different approaches to implementing *Iterator* were identified, which gave rise to four examples for *Iterator* – two examples for each scenario. In some cases, the additional examples were developed only for one of the scenarios, as the other scenario did not prove suitable for the approach.

Preparing the material for the study
The Java collection by Cooper was subject to a number of refactorings prior to the study. The motivation for the refactorings were mostly to tidy up the structure and programming style, to ensure that all Java code is made to conform to modern notions of good OOP style [17]. It is important to make sure the OOP style is good, as it is known that good OOP style is a necessary prerequisite for good AOP style [60]. The changes most often performed were to ensure that each class has a single set of responsibilities. For instance, each example includes a class with the main method for a given example, but in the Cooper collection those classes often doubled as a GUI for the example. They extended the class JFrame from the standard *swing* API and in some cases also implemented Java interfaces from *swing* such as ActionListener to provide the actions that execute when GUI widgets are clicked. The adaptations entailed separating the various responsibilities into separate classes – many simple classes such as button actions were thus turned into separate classes. After the refactorings, each class with a main method serves only as the driver for the example. In most cases, at least one second class was produced, which represents the GUI for the example.

How the new material was developed
Development of the OT/J implementations was carried in two phases. The majority of the implementations was created in the first phase, yielding at least one OT/J implementation of each scenario from the HK study and most of the scenarios from the Cooper collection.
The goals for the first phase included the assessment of support by OT/J for reusability [21]. Thus, the approach taken was to produce an OT/J implementation for each HK scenario and next try to reuse it in the Cooper scenario for the same pattern – possibly with some adaptation to generalize it. Only the modules that are used in examples of both scenarios are classified as reusable. It was also established that modules comprising just abstract declarations with no concrete definitions whatsoever are not taken into account in assessments of reusability. This was necessary because, due to the nature of some of the features of OT/J, it is very easy create modules with abstract declarations that can be used in many different examples but add no concrete members to the actual implementations. The results obtained in that first phase are presented in our SAC/OOPS paper [21]. It should be pointed out that the designs adopted during the first phase were strongly influenced by the AspectJ versions of those scenarios. This is arguably a drawback, as an implementation in OT/J mimicking the AspectJ approach is not guaranteed to exploit the capabilities of OT/J to the full. All reusable modules used in the analysis from section 4 originate from the first phase.

The second phase comprised a thorough revision of all examples, taking advantage of the experience gained in the first phase. The revision was performed with an eye on alternative designs that would take advantage of the features of OT/J, namely to obtain simpler or more intuitive designs. The two main outcomes of the second phase were (1) to complete the collection of examples derived from the Cooper collection and (2) the creation of alternative implementations for several patterns – *Composite*, *Iterator*, *Prototype* and *Visitor*. The second examples are arguably more intuitive than the first versions, for the reasons presented in the following sections. However, they also deviate more from the design approaches taken by Hannemann and Kiczales with AspectJ because they exploit language features not found in AspectJ (e.g., making the team be one of pattern participants and declared lifting). All examples from the second phase comprise approaches that are arguably simpler and/or more adequate but also discard the reusable modules created in the first phase. *Visitor* is a notable case: a version was developed in the first phase that includes an abstract team that mimics the approach taken in the HK study and used in both scenarios. However, in the second phase new implementations for *Visitor* were developed that look simpler and less contrived and no longer use that reusable abstract team.

3.1. Extensibility of instantiation code: *Factory Method* and *Abstract Factory*

Factory Method

One important difference between Java and OT/J is that the name of a class in instantiation code using new has fixed semantics in Java but is overridable in OT/J, due to virtual classes. Thus, constructors of role classes are polymorphic the same way as methods. Emulating polymorphic constructors is precisely the purpose of *Factory Method*, by leveraging traditional method polymorphism [18]. For this reason it was reasonable to expect OT/J to represent a significant advancement over Java on implementations of *Factory Method*. In fact, we expected that implementations of *Factory Method* should "disappear" altogether. Factory methods would be replaced with straightforward calls to constructors. Surprisingly, this is not the case with the examples of *Factory Method* presented here.

Many examples of teams from the present repository do indeed contain many cases of expressions using new that are extensible, which illustrates constructor polymorphism can be obtained without explicit code for the purpose. Nevertheless, the actual examples for *Factory Method* implemented in OT/J still include explicit factory methods, because the original Java examples use the *parameterized variant* of *Factory Method*, in which an argument is passed to the method to specify the actual concrete class to instantiate. The code for the selection of the actual class to instantiate on the basis of the argument value must be placed somewhere and that place is still a method – indeed a factory method. For this reason, the OT/J implementations of the two *Factory Method* scenarios are not devoid of actual factory methods. Having said that, we believe that in cases in which explicit decision logic does not depend on a specific variable, code pertaining to factory methods can indeed be replaced by a direct use of language constructs.

Take the simple scenario by Cooper as an example (Figure 4). An operation receives a text field containing a name comprising at least two words. The operation extracts the first and last names from the field in two different ways, depending whether it has a comma. If yes, the format "<last>, <first>" is assumed. If it hasn't, it assumes the first word in the field is the first name. An instance of class Namer (participant *product*) presents the extracted first and last names. Two sub-classes of Namer are available (FirstFirst and LastFirst), i.e., two instances of participant *concrete product* that implement the two actual ways to present the name. The *Factory Method* pattern also defines the roles *creator* and *concrete creator*, which are abstract and concrete representations respectively, of the object providing the factory method. Both correspond to class NamerFactory, whose factory method creates the suitable instance of Namer depending on the format of the text field.

Figure 5 shows the OT/J implementation of that scenario. In the OT/J implementations of *Factory Method*, *products* are roles and the creator participant is the team, which defines the factory method as a top-level team method returning an instance of required concrete product according to its argument's value. Role instances are sent to the outside of the team as instances of a Java interface. The product participant was changed from a class (Namer) to a Java interface (INamer) so that the *Transparent Role* idiom could be used. As an alternative, we can keep the role instance hidden within the team, which is the option taken for the HK scenario.

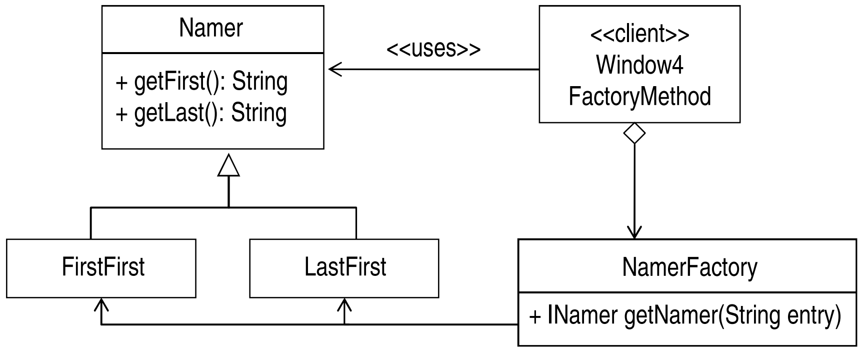
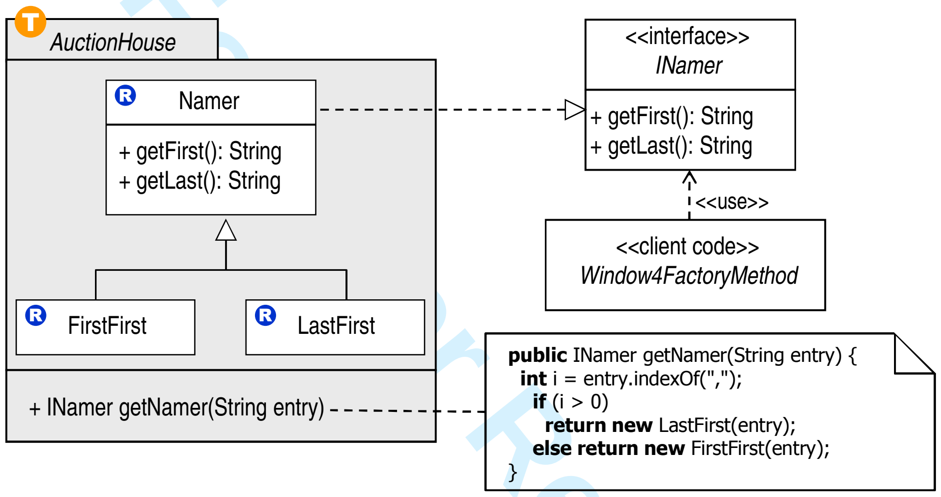


Figure 4. *Factory Method* in Java (Cooper scenario).



```
public INamer getNamer(String entry) {
  int i = entry.indexOf(",");
   if (i > 0)
    return new LastFirst(entry);
   else return new FirstFirst(entry);
}
```

Figure 5. *Factory Method* with OT/J (Cooper scenario).

Abstract Factory

The purpose of *Abstract Factory* is to provide an interface (*abstract factory*) for creating families of related objects (*products*) and ensure that instances of a given family are created consistently, avoiding undesirable mixing between families [18]. That is exactly the purpose of family polymorphism [12]: to provide guarantees of consistency among object families. Family polymorphism supports the intended effect comprehensively, with direct support from the type checker. In sum, OT/J provides direct support for *Abstract Factory*. This lends credence to the claim that OT/J directly supports *Factory Method* as well, since *Abstract Factory* is really an expanded version of *Factory Method*.

In both scenarios for *Abstract Factory*, the abstract factory and concrete factory participants are teams, with the actual products being represented by roles. Figure 6 and Figure 7 show the implementations of the HK scenario for *Abstract Factory* in Java and OT/J respectively. Abstract team ComponentFactoryTeam defines two roles as subclasses of javax.swing.JLabel and javax.swing.JButton, which form the roots of hierarchies of products that can be flexibly extended in that same team or in sub-teams. Team-level methods createLabel and createButton are factory methods and comprise an illustration of direct language support for *Factory Method*. The key fact to note in the OT/J implementation is that instantiation statements using new are used polymorphically. For example, team-level method createButton (code shown in the note from Figure 7) is inherited by sub-teams RegularFactoryTeam and FramedFactoryTeam without overriding but the actual RButton class instantiated is that defined in the sub-team. Note also that the return type of createButton is JButton rather than RButton to avoid exposing the role to the outside of the team, i.e., this is an instance of the *Transparent Role* idiom. Abstract method getName is less important, merely a team-level method that can be used by some code along the team hierarchy.

In this design, the extends clause that make RLabel and RButton inherit from the swing classes are placed in the roles from the super-team. A possible alternative design could have placed the extends clauses in the roles from the sub-teams. This would be the right approach if the swing classes comprised just one of possible implementations for RLabel and RButton. That is not used in this example to avoid duplicating the extends clauses (and constructor definitions) in two sub-teams.

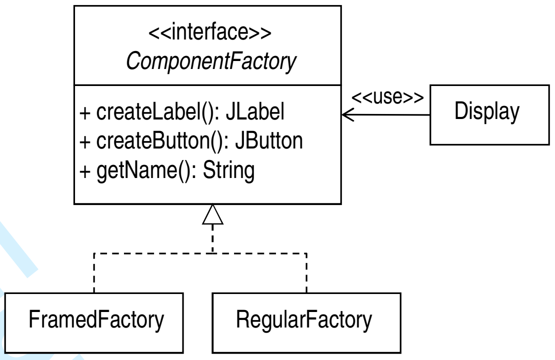
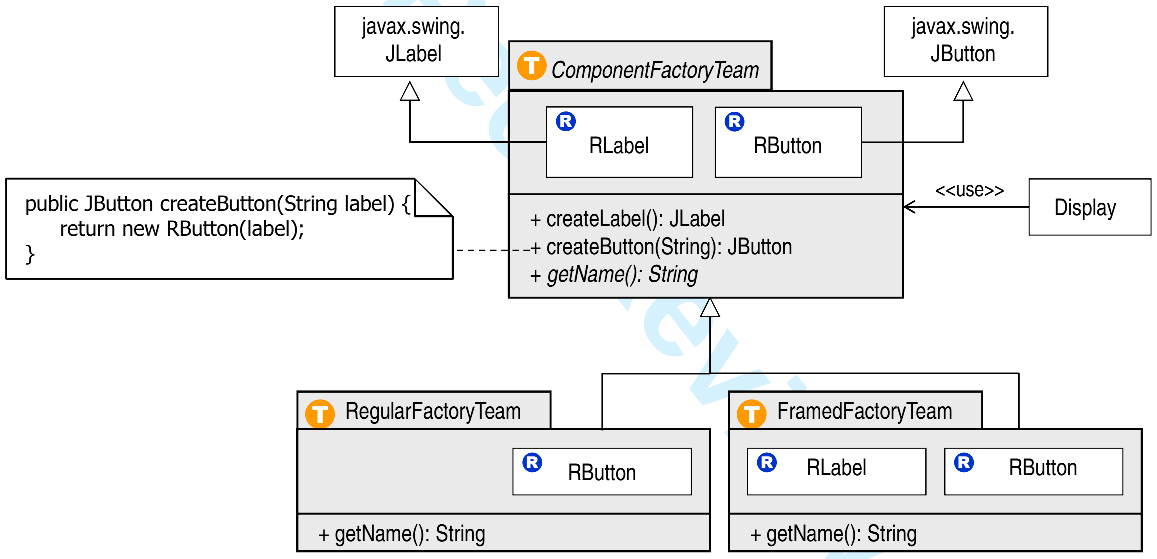


Figure 6. *Abstract Factory* in Java (HK scenario).



```
public JButton createButton(String label) {
    return new RButton(label);
}
```

Figure 7. *Abstract Factory* in OT/Java (HK scenario).

3.2. Support for double dispatch: *Visitor*

Subclassing from traditional inheritance is the standard solution for the problem of adding new operations to a class without performing invasive changes on the original class. Inheritance fulfils the requirements of the *open-closed principle* [41], which states that modules should be closed for modifications while open for extensions. However, when the system to be extended already comprises *an entire hierarchy*, subclassing is unable to provide a solution – inheritance is about exploiting the dimension of subclassing and in this case the dimension is already "taken". In the absence of a straightforward solution, adding new operations to entire hierarchies entails modifying each class that belongs to the hierarchy, which risks giving rise to an excess of operations in each class. The problem can be viewed as caused by a limitation in traditional inheritance, which supports only *single dispatch*, i.e. method dispatch based on the runtime type of the object that received the method call. In single dispatch, the types that can potentially be selected are the sub-types of the static type (i.e., the type of the object reference for the target of the call) found in the inheritance hierarchy. *Multiple dispatch* provides one solution to this problem. Multiple dispatch is the ability of a language to select the block of code to execute on the basis of the types of *all* parameters from the signature of the called method [32]. More specifically, *Visitor* relates to *double dispatch*, i.e., dispatching on the basis of two independent types. Double dispatch is less general than multiple dispatch but seems to be the most common case. *Visitor* is a technique to emulate double dispatch in traditional OOP languages [18].

Visitor is a technique to add operations to instances of a pre-existing class hierarchy, through the encapsulation of the additional operations on separate *visitor* objects. Each class from the hierarchy (*concrete element*) must define an *accept* operation that can receive a visitor and use its service, through a call to a *visit* operation that all visitors must declare. The *visit* operation in turn receives the instance of the class hierarchy and runs the suitable behaviour. Often, different classes from the hierarchy map to different behaviour, which entails defining different visit methods for each of those *concrete element* classes. This usually achieved through *signature overloading*, i.e., defining several different signatures for the operation, one for each different class. Therein lies the problem with *Visitor*: adding a new class to the hierarchy entails adding a new signature definition for that class in every visitor class.

Two different OT/J implementations were created for *Visitor*. One was created in the first phase and closely relates to the design used in AspectJ by Hannemann and Kiczales, which is about using inter-type declarations (a.k.a. *introductions* and the *open class mechanism*) to compose the additional accept and visit methods to concrete participants. That OT/J implementation of *Visitor* uses role playing for the same purpose. An abstract team is (re)used in both scenarios.

A different approach was developed in the second phase, described next. It eschews the reusable team but also does without explicit accept and visit methods. It is based on a role hierarchy within a team to represent a second dimension of polymorphism and dynamic dispatch, which is exploited to support the non-invasive addition of visitors (i.e., additional operations) to an entire hierarchy of base classes. By building a suitable hierarchy of visitor roles within a team, dispatch to the intended visitor can be obtained without the need for explicit *accept* and *visit* operations. Figure 8 shows one example – from the HK scenario – of the team/class structure required for the technique.

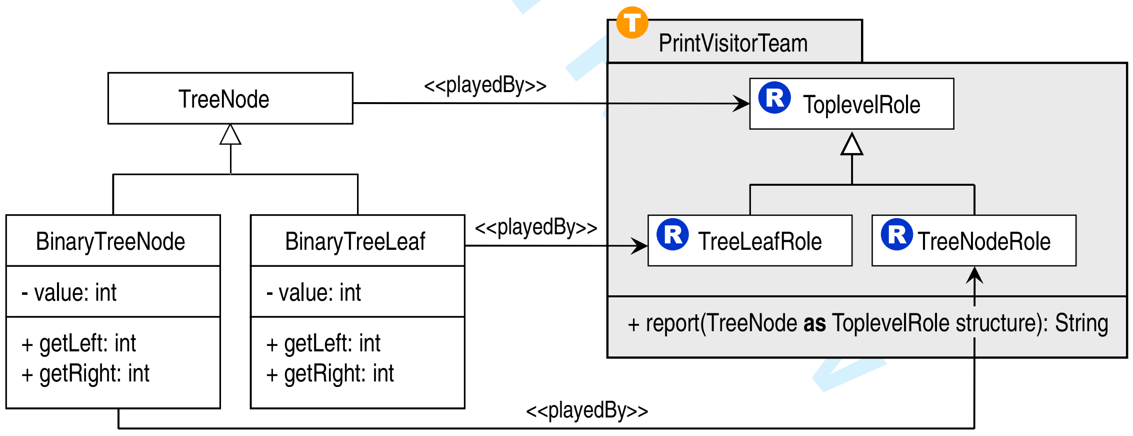


Figure 8. OT/J implementation of Visitor supporting double dispatch

The problem is to add new operations to a structure of instances of these two classes, which is done by the roles. In Java, these correspond to two visitor classes defining two visit methods each. The operations are (1) to calculate a sum of the integer fields from all instances of the structure to be traversed; and (2) to obtain a string representation of the entire structure, built from the string representation of each object from the structure.

Each type driving the double dispatch originates from one of the two hierarchies shown in Figure 8 – the base class hierarchy and the role hierarchy. The (very simple) scenario comprises an (empty) class TreeNode and subclasses BinaryTreeNode and BinaryTreeLeaf. A combination of translation polymorphism (lifting) and sub-class polymorphism perform the double dispatch. This second OT/J implementation uses a different team for each additional visiting operation. Listing 2 shows the code for the HK example in OT/J. The team defines a role for each different class from the hierarchy plus a top-level team method *report* (Listing 2, lines 24-26) receiving the base object, which must be called from client code. Double dispatch works in two phases: (1) the team method translates (lifts) the base object to its corresponding role instance through declared lifting. Next, (2) traditional late binding across the role hierarchy selects the exact block of code. Note that the signature of the team method uses only the top-level types from both hierarchies. This technique corresponds to an idiom (*Double Dispatch*) that was recently added to the OT/J official site at Eclipse [3].

```
public team class PrintVisitorTeam {
  protected class TreeNodeRole playedBy TreeNode {
    public String getRepresentation() {
      return ""; //this method is not supposed to be called, ever
    }
  }
  protected class TreeLeafRole extends TreeNodeRole playedBy BinaryTreeLeaf {
    public abstract String getRepresentation();
    String getRepresentation() -> int getValue() with {
      result <- Integer.toString(result)
    };
  }
  protected class Composite extends TreeNodeRole playedBy BinaryTreeNode {
    public abstract TreeNodeRole getLeft();
    getLeft -> getLeft;
    public abstract TreeNodeRole getRight();
    getRight -> getRight;

    public String getRepresentation() {
     return "{"+getLeft().getRepresentation()+","+getRight().getRepresentation()+"}";
    }
  }

  public String report(TreeNode as TreeNodeRole structure) {
    return “...” + structure.getRepresentation();
  }
}
```

Listing 2. Implementation in Object Teams of one concrete example of Visitor.

3.3. Connecting distinct hierarchies: *Bridge*

The GoF book presents *Bridge* as a way to “decouple an abstraction from its implementation so that the two can vary independently” [18]. *Bridge* deals with the problem that arises when an abstraction requires more than a single class to represent it and one wants to use inheritance to bind the abstraction to different implementations for different specific cases. In cases the abstraction comprises an entire hierarchy, we are unable to use inheritance to incrementally extend the hierarchy – the logic related to the implementation ends up in the same hierarchy, tangled with the representative elements of the abstraction. The solution proposed by *Bridge* is to place abstraction and implementation in separate hierarchies. The connection between the two hierarchies is through a field in the class at the top of the abstraction hierarchy that points to the top class of the implementation hierarchy. When calls from the abstraction are made through that field, traditional polymorphism ensures that the right block of code in the implementation hierarchy is selected. An example of the class structure proposed by *Bridge* is shown in Figure 9.

The two main participants defined by *Bridge* are *abstraction* and *implementor*, which represent the top of the two hierarchies to be connected. In the HK scenario (Figure 9), these correspond to classes ScreenImplementation and Screen respectively. In traditional OOP, the connection is usually implemented through aggregation, i.e., by placing in *abstraction* a field that refers to *implementor*.

When discussing implementations of *Bridge*, the GoF book considers the case in which the *implementor* is selected during runtime, on the basis of some requirement that is assessed dynamically. The GoF book points out that it is possible to delegate the decision on which *implementor* to select to another object altogether, using *Abstract Factory* to introduce factory objects whose sole purpose is to encapsulate *implementor* specifics. Note that such decisions are naturally supported by OT/J’s extra dimensions of polymorphism. The traditional OOP approach to *Bridge* for achieving this object-level composition (including Java) is to pass the *implementor* object to the constructor of *abstraction*. This way, each *abstraction* object holds a reference to an *implementor* object. The primary difference between the Java and OT/J implementations is that role playing is used instead of aggregation to connect the elements of *abstraction* to the *implementor* elements – an example is shown in Figure 10, based of the OOP scenario from Figure 9. The *abstraction* participant is turned into a role that is played by base *implementor*. It is not necessary to maintain a field referring to *implementor* in each *abstraction* class as all members of *implementor* required by *abstraction* are acquired through callouts. However, the team itself holds a field referring to a specific *implementor* object (Listing 3), to ensure that an *abstraction* instance acquires the implementation from a specific *implementor* object.

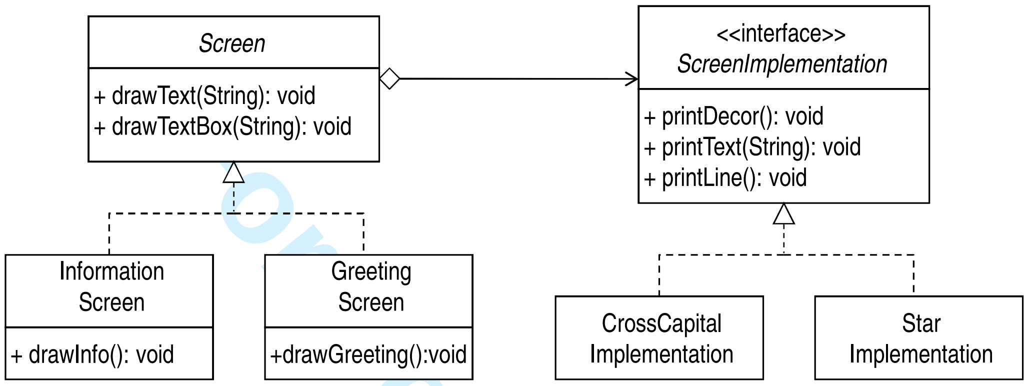


Figure 9. Traditional OOP implementation of *Bridge* (HK scenario).

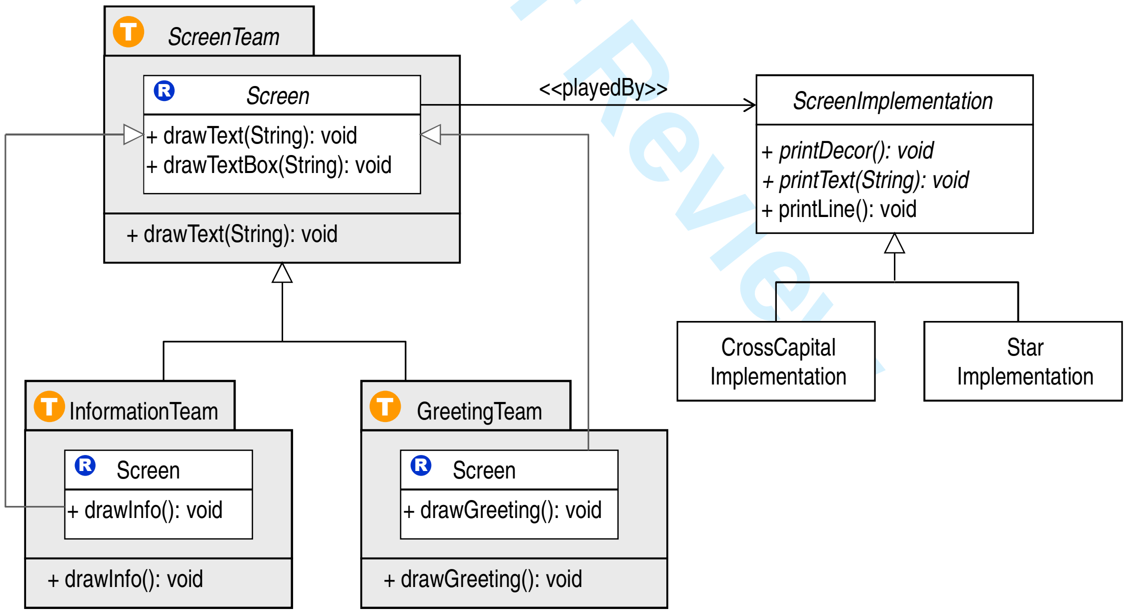


Figure 10. OT/J implementation of *Bridge* (HK scenario).

The OT/J implementation uses the lifting constructor implicitly synthesized for bound roles to pass the *implementor* object to the constructor of *abstraction*. Thus, this approach is an instance of the problem of how to support communication between roles and the outside of its enclosing team without violating team encapsulation. The solution used in this case is to use team-level methods to forward calls from outside the team to the role instance within the team. When the role is created, it is bound to the base object passed to the lifting constructor.

Lifting constructors can only be called from within the enclosing team, so the constructor of the team is used to create the *abstraction* role instance, passing the *implementor* object to the lifting constructor. A blemish in this solution is the need to add team-level methods just to serve as bridges between clients outside of the team and the given role. Listing 3 illustrates this OT/J approach applied to the HK scenario. Team ScreenTeam represents the top of the *abstraction* hierarchy, which can flexibly extended by sub-teams such as ScreenGreetingTeam and its enclosed roles. Listing 4 shows the driver for the HK example in OT/J – from the point of view of the client, it looks just like a traditional OOP approach and supports the setting up of the various combinations of *abstraction*s and *implementation*s.

```
public abstract team class ScreenTeam {
  protected Screen _screen;
  public abstract class Screen
      playedBy ScreenImplementation {
    public abstract void printLine();
    printLine -> printLine;

    public abstract void printDecor();
    printDecor -> printDecor;

    public abstract void
      printText(String text);
    printText -> printText;
    public void drawText(String text) {
      //...
    }
    //...
  }

  public void drawText(String text) {
    _screen.drawText(text);
  }
}
```

```
public team class GreetingScreenTeam
    extends ScreenTeam {

  public class Screen {
    public void drawGreeting() {
      drawTextBox("Greetings!");
    }
  }

  public
  GreetingScreenTeam(ScreenImplementation si)
    _screen = new Screen(si);
  }

  public void drawGreeting() {
    _screen.drawGreeting();
  }
}
```

Listing 3. Team modules implementing an example of Bridge.

```
public static void main(String[] args) {
  System.out.println("Creating implementations...");
  ScreenImplementation i1 = new StarImplementation();
  ScreenImplementation i2 = new CrossCapitalImplementation();

  System.out.println("Creating abstraction (screens) / implementation combinations...");
  GreetingScreenTeam gs1 = new GreetingScreenTeam(i1);
  GreetingScreenTeam gs2 = new GreetingScreenTeam(i2);
  InformationScreenTeam is1 = new InformationScreenTeam(i1);
  InformationScreenTeam is2 = new InformationScreenTeam(i2);

  System.out.println("Starting test:\n");
  gs1.drawText("\nScreen 1 (Refined Abstraction 1, Implementation 1):");
  gs1.drawGreeting();
  gs2.drawText("\nScreen 2 (Refined Abstraction 1, Implementation 2):");
  gs2.drawGreeting();
  is1.drawText("\nScreen 3 (Refined Abstraction 2, Implementation 1):");
  is1.drawInfo();
  is2.drawText("\nScreen 4 (Refined Abstraction 2, Implementation 2):");
  is2.drawInfo();
}
```

Listing 4. Example of an use of the OT/J Bridge (HK scenario).

Though *Bridge* and *Visitor* relate to the common issue of integrating two distinct hierarchies, the solutions proposed for each pattern are different. In the *Visitor* scenarios, the connection between hierarchies is performed in an entirely dynamic way, by supporting a two-phase dispatch system that is based on both the base class and role hierarchies. In *Bridge*, the team holds a reference to the corresponding role object, which was created upon instantiation of the team. Since the team already holds that reference, declared lifting is not used, in contrast with the solution for *Visitor*.

3.4. Partial support for the pattern: *Memento*

The intent of *Memento* is "without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later" [18]. The pattern defines three roles: *originator*, which is the object whose state is to be saved, *memento* – the object that keeps the saved state and prevents it from being accessed by unauthorized objects – and *caretaker*, the object that controls when and how the state is saved, keeps the memento and uses it to restore the state to the originator if needed.

The requirement that it should be done "without violating encapsulation" is often difficult to meet thoroughly, due to limitations on the language's encapsulation features. Many programming languages do not provide a fine-grained control of encapsulation that exactly corresponds to what is required for *Memento*. For C++, Gamma *et al*. suggest that the *friend* construct be used. For Java, one option – used by Cooper – is to provide package-level visibility to the state of the originator, with the drawback that all classes placed in the same package as the originator also have access to the memento state.

OT/J does not provide language support for creating and managing snapshots of object states, but some of the features that adds to Java help to enforce the encapsulation required by *Memento*. *Confined* and *opaque roles* are two features that provide stronger guarantees that a role's state remains encapsulated within the team [26]. The differences between opaque and confined roles are noticeable with respect to the use of role instances outside the team. While opaque roles allow for certain uses of role instances outside the team, with limited access, confined roles provide the strictest form of encapsulation, guaranteeing that no features to that object can be accessed outside of the team, even those generally accessible through java.lang.Object. Obtaining a reference to the object through a special interface from the OT/J API and passing it back to the team is all that is allowed by the type system. Thus, using a confined role for representing the *memento* participant goes further than Java in ensuring encapsulation of the memento.

To illustrate use of a confined role in an implementation of *Memento*, Listing 5 shows a simple implementation of the HK scenario for *Memento*. The team acts as *caretaker* and role Originator is bound to Counter, a simple class whose state is obtained through method getCurrentValue. The key detail is the unbound role *memento* that extends Team.Confined. To illustrate the impact of this technique on clients, Listing 6 shows the driver for this example. The lines in strikethrough illustrate attempts to use feature defined in java.lang.Object. Contrary to what would happen with plain Java, they give rise to compiler errors. However, note that the above feature does not prevent defining public methods in role *memento* to expose its state. Though Memento no longer has the members traditionally acquired from java.lang.Object, care must still be taken not expose Memento's state anew.

```
public team class MementoTeam {
  protected Confined savedMemento;

  protected class Originator playedBy Counter {
    /* creates a Memento with the current state of Originator */
    protected Memento createMemento() { return new Memento(this); }

    /* returns an object with the current state of the Originator */
    protected abstract Object getState();
    getState -> getCurrentValue;

    public void setState(Memento m) -> void setCurrentValue(int value) with {
      (Integer) m.getState() -> value
    }
  }

  public class Memento extends Confined {
    private Object state;
    protected Memento(Originator o) { this.state = o.getState(); }
    protected Object getState() {  return state; }
  }

  public Memento createMementoFor(Counter as Originator o) { return o.createMemento(); }
  public void setMemento(Counter as Originator o, Memento m) { o.setState(m); }
}
```

Listing 5. Illustration of the use of Confined roles to support Memento in OT/J (HK scenario).

```
  public static void main(String[] args) {
    final MementoTeam mementoTeam = new MementoTeam();

~~    mementoTeam.Memento o1 = mementoTeam.new Memento();~~
~~    mementoTeam.Memento o2 = mementoTeam.new Memento();~~
~~    if(o1.equals(o2)) System.out.println("equal"); else System.out.println("not equal");~~

    Counter counter = new Counter();
    Memento<@mementoTeam> memento = null;
    for (Integer i=1; i<=5; i++) {
      counter.increment();
      counter.show();
      if (i == 2) memento = mementoTeam.createMementoFor(counter);
    }
~~    System.out.println("\nTrying to reinstate state (" + memento.getState() + ")...");~~
~~    System.out.println("\nTrying to reinstate state (" + memento + ")...");~~
    System.out.println("\nTrying to reinstate state (2)...");
    mementoTeam.setMemento(counter, memento);
    counter.show();
  }
```

Listing 6. Driver for the HK Memento to illustrate the impact of Confined roles on clients.

Finally, it is worth noting that a different approach could be used to implement *Memento*. Instead of caring about the encapsulation of the *memento* object, the focus could be on how the *memento* accesses the state of the *originator*. An alternative approach could be created that exploits the privileged access of roles to the members of their bases, which are likely to achieve results similar to the *friend* feature of C++.

3.5. Packaging multiple entitities more cohesively: *Builder*, *Composite*, *Flyweight*, *Interpreter*, *Iterator*, *Mediator*, *State*

The patterns from this group share a common trait in that a group of different classes share a common context or set of features within the context of the pattern. Instances of those classes form a collaboration of objects that are represented by participants in the pattern. The participants serve to abstract from the specifics of concrete classes. All patterns include the notion of a global context shared by the collaborators. In OT/J, it seems natural to use roles to represent the collaborating objects and to assign the responsibility for holding and managing the context of the collaboration to the enclosing team. The underlying design principle is as follows. Teams impose a boundary around their roles. Thus, introducing teams to a traditional OOP design can be seen as an opportunity to *improve* that design by reducing exposure of some of the participants to clients. In each particular case, it must be checked whether a team can be made to represent in a natural way some abstraction in the application. This thinking is absent in the GoF patterns, which have no concept of boundaries similar to a team boundary, with the possible exception of *Façade*, whose purpose is to make such a boundary explicit. The OT/J implementations for all the patterns covered in this section follow this approach. For the same reason, virtual classes and family polymorphism come much to the fore in all the examples from this group.
Some special cases from this group warrant a few clarifications. *Iterator* was implemented using two different approaches, one in each of the development phases mentioned at the start of this section. The approach developed in the second development phase has the common characteristics covered in this section and for this reason is described here. See section 3.6 for the other approach to *Iterator*. *Builder* has some of the characteristics just highlighted, in that a common context can also be identified. It differs from the other patterns in that it does not define a collaboration between several participants. However, one of the participants it defines (*director*) corresponds to a context for others and therefore lends itself to be played by the team. Since the OT/J implementation of *Builder* shares characteristics with the patterns from this group, *Builder* is included in this group.
A short overview of each pattern is provided next, highlighting collaborator participants and the use of global context:

- *Composite* is about setting up a unified way to interact with a complex object (*composite*) and other objects (*leafs*) comprises its internal structure. The pattern is concerned with hiding the complexity that arises when one must treat individual objects and composites differently. To abstract from such differences, the pattern prescribes that all objects expose a common interface (*component*). The common context is supported by *composite* participant.

- *Flyweight* is about the sharing of object references to support large numbers of fine-grained objects in an efficient way. The fine-grained objects are represented in an abstract way by the *flyweight* participant, which also declares a common interface through which the objects can receive and act on the shared state. Actual flyweight classes are represented by a *concrete flyweight* sub-type of *flyweight*. There is also an *unshared concrete flyweight* sub-type of *flyweight* to represent sub-classes of *flyweight* that do not need to be shared. The common context is supported by the *flyweight factory* participant, which is also responsible for creating the *flyweight*s and managing the way state is shared among them.
- *Interpreter* defines a representation for the grammar of a simple language – often an abstract syntax tree – along with an interpreter uses the representation to interpret sentences in the language. The pattern defines an *abstract expression* participant that represents the various distinct nodes comprising the abstract syntax trees, which is in turn divided into sub-types *terminal* and *non-terminal*. The common context is represented by a *context* participant that contains global information within the interpreter.
- *Iterator* defines an *aggregate* object (participant *aggregate*) whose elements are to be accessed sequentially without exposing its underlying representation. The *iterator* participant defines an interface for accessing and traversing the elements of the *aggregate*. The *aggregate* provides the context in which the iterator carries out its task.
- *Mediator* defines an object that encapsulates how a set of objects interact and promotes loose coupling between those objects by preventing them from referring to each other explicitly. A *colleague* participant represents all those objects in an abstract way. A *mediator* participant defines an interface for communicating with *colleague*s and a *concrete mediator* represents the actual mediating class, which knows and maintains the *colleague*s and implements their cooperative behaviour by coordinating them. The common context is represented and maintained by the *concrete mediator*.
- *State* is about allowing an object to alter its behaviour when its internal state changes. The object will appear to change its class. *State* defines a *state* participant representing in an abstract way the internal states through which the object can go. The *concrete state* participants are sub-types of *state* that represent the actual internal states on which the object's behaviour depends on. The common context is represented by the *context* participant, which also defines the interface of interest to clients and maintains an instance of the concrete state participant that defines the current state.
- *Builder* is about separating the construction of a complex object (*product*) from its representation so that the same construction process can create different representations. The object responsible for the construction process is the *concrete builder*, whose interface is defined by its super-type – participant *builder*. The *director* participant constructs the *product* in stages, using the *builder* interface. In some cases it also holds the concrete builder. In all OT/J examples of *Builder*, a team plays the role of *director*, which serves to provide the context for the *builder* and the building process.

Despite the common characteristics of these patterns, as well as the common approach in implementing them, there is still room for representing the concrete participants in different ways – two approaches are possible:

1. Turn the original Java classes into roles within a team, in which case they usually are unbound roles
2. Leave the participants as plain Java classes and create roles that representing the participant within the pattern, which are bound to the classes through role playing.

The repository of implementations of these patterns includes instances of both approaches. Choice between the above two approaches depends on the nature of the pattern, and the specific case at hand. If a group of classes is used only within the collaboration specified by the pattern, it makes sense to encapsulate them completely, by turning the classes into (protected) roles. In such cases, the roles are usually unbound roles, in which case the team does not use callins and does not require activation. This approach has the benefit of reducing the complexity the system from the point of view of clients of the team: several pattern-specific classes are replaced by a single module. In cases in which the pattern participants are used elsewhere in a system, the second approach is more appropriate. Moving code specific to the pattern participant to the roles simplifies the original classes, i.e., it removes the *Double Personality* code smell [45, 43]. Composition between the classes and roles is usually carried out by means of the playedBy relation. The team is likely to use callins and therefore require activation.

The first approach is illustrated using *Interpretor*. Figure 11 and Figure 12 show the implementations of the HK scenario in Java and OT/J respectively. A number of classes are replaced by two teams representing the interpreter. In this case, the abstract syntax tree was partitioned into two layers but this is just one of several design alternatives, as the team can be managed and extended very flexibly [13]. A single team could easily be used instead.
The second approach is illustrated using *Mediator*. Figure 13 and Figure 14 show the Java and OT/J implementations of scenario by Cooper. An instance of the team MediatorTeam (Figure 14) holds state to refer to the participant objects, by means of references to the role instances. The base instances associated to the role instances are indirectly manipulated through those references. Note that all references to concrete participants are made through role Colleague declared in super-team MediatorProtocol.

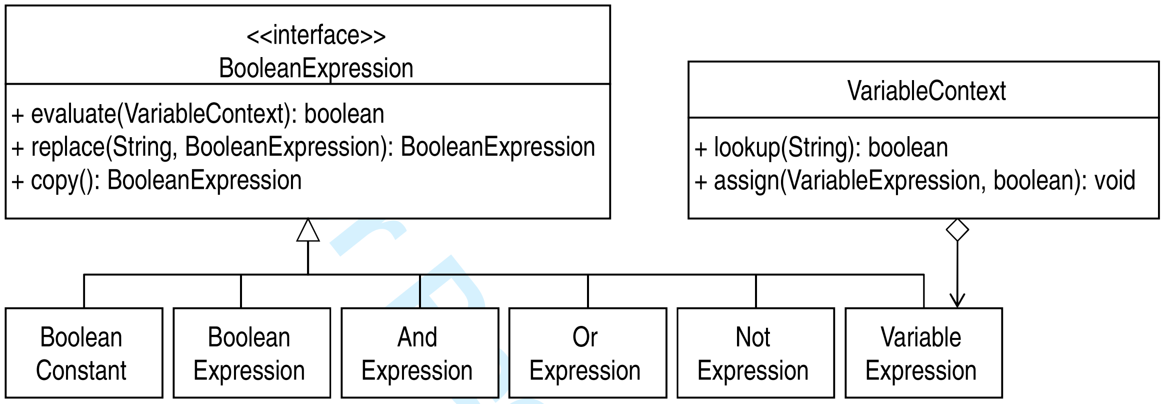

<<interface>>
BooleanExpression
+ evaluate(VariableContext): boolean
+ replace(String, BooleanExpression): BooleanExpression
+ copy(): BooleanExpression
VariableContext
+ lookup(String): boolean
+ assign(VariableExpression, boolean): void
Boolean Constant
Boolean Expression
And Expression
Or Expression
Not Expression
Variable Expression

Figure 11. Java implementation of *Interpretor* (HK scenario).

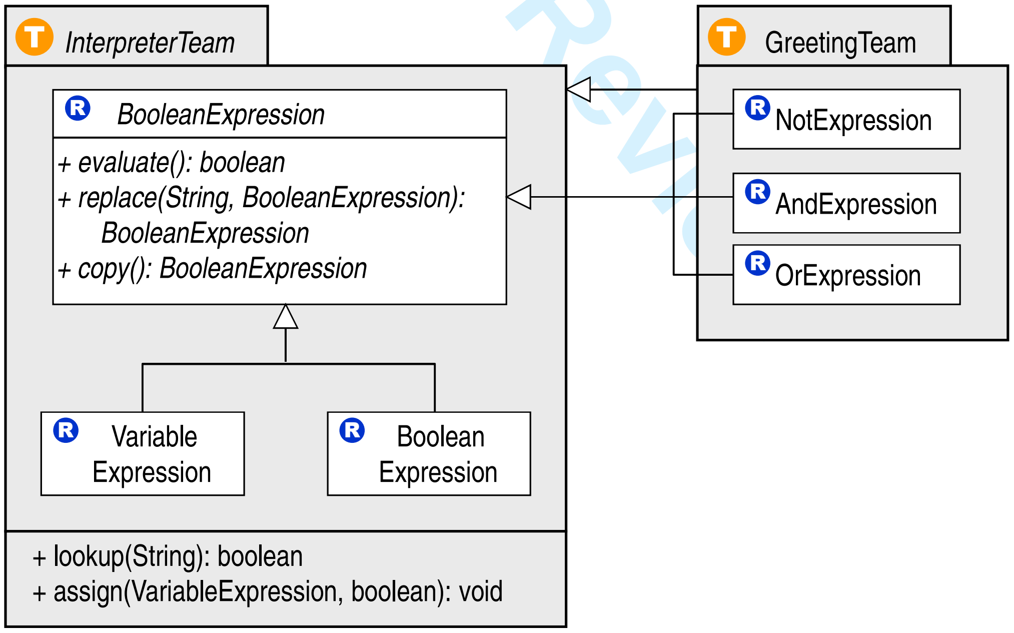
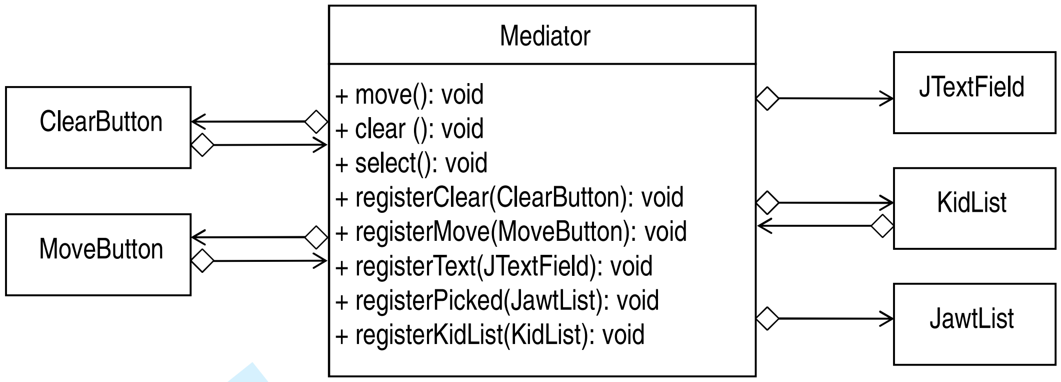

InterpreterTeam
BooleanExpression
+ evaluate(): boolean
+ replace(String, BooleanExpression): BooleanExpression
+ copy(): BooleanExpression
Variable Expression
Boolean Expression
+ lookup(String): boolean
+ assign(VariableExpression, boolean): void
GreetingTeam
NotExpression
AndExpression
OrExpression

Figure 12. OT/J implementation of *Interpretor* (HK scenario).


Mediator
+ move(): void
+ clear (): void
+ select(): void
+ registerClear(ClearButton): void
+ registerMove(MoveButton): void
+ registerText(JTextField): void
+ registerPicked(JawtList): void
+ registerKidList(KidList): void
ClearButton
MoveButton
JTextField
KidList
JawtList
Figure 13. Java implementation of *Mediator* (Cooper scenario).

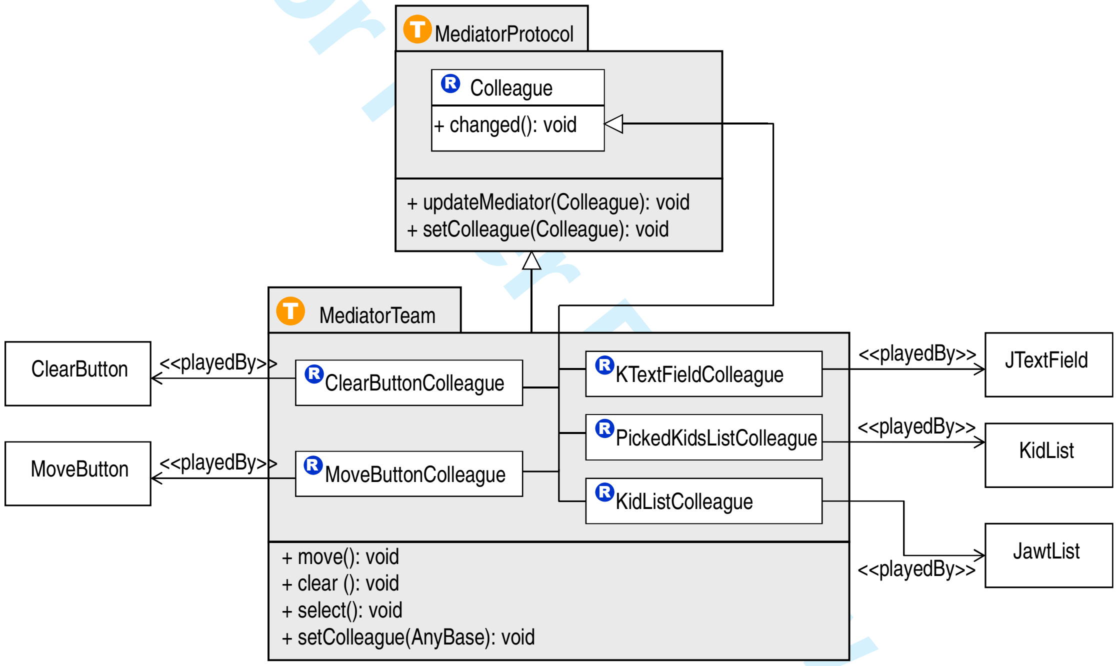


Figure 14. OT/J implementation of *Mediator* (Cooper scenario).

Obstacles to flexible extensibility: *Composite*

Composite defines the abstract participant component to expose a common interface (*component*) for both primitives and composites, represented by participants *leaf* and *composite* respectively. Clients always access *leaf* and *composite* objects through the interface exposed by *component*. Figure 15 shows the HK scenario to illustrate a typical Java implementation. Classes Directory and File represent *leaf* and *composite* respectively. The Java interface FileSystemComponent is the *component* and is implemented by both Directory and File.

Two different OT/J implementations were created for *Composite*. One was created in the first development phase and mimics the AspectJ approach, whose key components are a reusable abstract aspect that is based on inner marker interfaces (i.e., within the aspect) representing participants *component*, *composite* and *leaf*, and a hash map holding the mappings between components and their children. The remaining members of the aspect comprise a set of aspect methods for managing the hash map. Concrete sub-aspects use *declare parents* clauses to bind the marker interfaces to the specific classes of a given example. In a previous paper [44], we argued that the resulting logic is relatively complex for the simple task of managing the relation between composites and their leafs. The complexity is due in part to a protocol based on visitors through which operations on the elements of the composite structure can be carried out. We expressed doubts of whether the added complexity warrants the use of a reusable implementation of *Composite*.

The OT/J examples for *Composite* developed during the first development phase mimic the AspectJ approach, using roles instead of marker interfaces and the role playing relation instead of *declare parents* clauses. It is a bit simpler in that it eschews the protocol based on visitors for adding operations on the participants. However, one important point is that may seem contrived to treat all participants of *Composite* as additional roles that may be (un)plugged from the specific classes. In many cases, it may seem more natural to treat the structure of the composite as intrinsic to the class playing that role. For this reason, and also because teams seem natural candidates for being composites for their roles, a different implementation was created in the second development phase. In this approach, the team becomes the *composite* as the context-holding participant, which uses roles to represent the *leafs*. Figure 15 shows a scenario for Composite and Figure 16 shows the OT/J implementation of it. Role Leaf is bound to base class File while team DirectoryTeam replaces Directory as the composite. In accordance to the pattern, both the team and role implement a common Java interface (IDirectory) – the component participant.

Unfortunately, this approach is unique among the examples from the entire OT/J repository in that the resulting team module cannot be flexibly extended. An unexpected hurdle prevents the team to be extended with the same flexibility as with other teams in the collection. The Java interface representing *component* cannot be placed within the team, as a team cannot implement a Java interface enclosed within it (nor would that make much sense). Herein lies the problem: deploying IDirectory as a top-level, standalone Java interface places it outside of the family of types associated to the team object (which is their type anchor). As a consequence, the family of types cannot be extended with the usual flexibility and guarantees from the type checker. For instance, extending IDirectory through sub-interfaces does not propagate to roles in sub-teams of DirectoryTeam and extending the roles in sub-teams has no impact on code based on IDirectory. In practice, the interface exposed by IDirectory is fixed and the team looses much of the extensibility that is a general hallmark of family polymorphism [12, 13]. It is worth pointing out that this limitation applies to instances of the *Transparent Role* idiom, since these entail using a top-level Java interface. It is given more prominence at this point because the use of *Transparent Role* are optional in most cases, but the use of a top-level Java interface is intrinsec to this implementation of *Composite*.

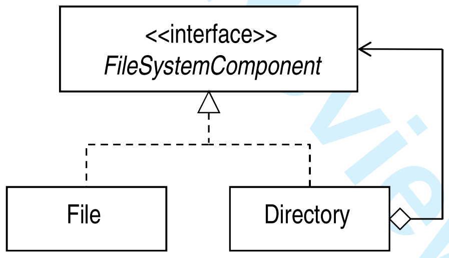


Figure 15. Java implementation of *Composite* (HK scenario).

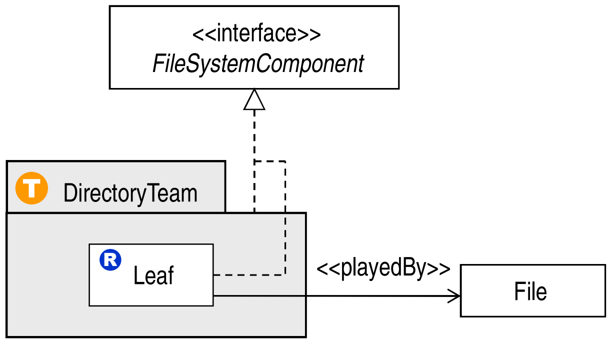


Figure 16. OT/J implementation of *Composite* (HK scenario).

Iterator
Two different approaches for *Iterator* in OT/J were implemented, one in each development phase. The one developed in the second phase fits the pattern from this group and is described next. The *aggregate* structure to be traversed lends itself to be represented by a team, with an unbound role for representing the iterator. The role object is returned by the team aggregate through a method declaring a Java interface type, using the *Transparent Role* idiom. Using this approach, implementations for both the HK and Cooper scenarios were created. The primary advantage is that the implementation of the iterator is completely encapsulated within the team. In theory, a similar approach can be taken with plain Java, using inner classes to represent the iterators, within an aggregate class. However, the system comprising the aggregate and its iterators cannot be flexibly extended through inheritance, since Java inner classes are not virtual.

3.6. Composing pattern roles through role playing: *Adapter*, *Decorator*, *Façade*, *Iterator*, *Prototype*, *Proxy*, *Strategy*

Regarding the patterns from this group, the specific contribution of OT/J is primarily due to a broader range of options to compose the modules that represent the pattern participants to the modules from the specific examples. Role playing comes to the fore as an alternative way to the traditional approaches to compose pattern roles to concrete participants (e.g, aggregation, inheritance and invasive changes on the source code).

Adapter
Adapter is about converting the interface of a class into another interface that clients expect. *Adapter* lets classes work together that otherwise could not, due to incompatible interfaces. The participants are a *target* object defining the specific interface that client objects are based on; an *adaptee* object that exports an interface that needs adapting, and the *adapter*, which is responsible for adapting the interface of the *adaptee* to that of *target*. The GoF book proposes two variants of this pattern: (1) a class variant – to be used with languages that support multiple inheritance – in which the *adapter* inherits from both *target* and *adaptee*, and an object variant in which the *adapter* inherits from just the *target* and uses aggregation for working with the *adaptee*. Though Java does not support multiple inheritance, it can support the class variant by making the class resort to both inheritance and interface implementation.
The Java example by Cooper uses the class variant and is based on classes from the *awt* and *swing* standard Java libraries. Cooper's scenario is based on the idea of a GUI that uses two list objects with incompatible interfaces. Scope for OT/J to make a significant impact in this example is not apparent: though the OT/J implementation is not identical to that in Java and uses teams with an *adapter* role, the role is unbound and ends up mimicking the approach of the Java *adapter* class. The HK scenario for *Adapter* is based on the object variant and the OT/J implementation uses role playing to bind an *adapter* role to an *adaptee* base class.

Decorator and Proxy
The intent of *Decorator* is to attach additional responsibilities to an object dynamically. The problem to be solved comprises an object that must be extended with multiple distinct functionalities without changing its interface. The additional functionalities give rise to many combinations and use of inheritance to support them gives rise to a combinatorial explosion of sub-classes. *Decorator* tackles this problem by providing a flexible alternative to inheritance for extending functionality. *Decorator* defines the *concrete component* as the object to be extended and *decorator* as the object that extends the functionality of the *concrete component*. The key principle of this pattern is that both *concrete component*s and *decorator*s declare a common *component* interface. In Java, the usual approach is to represent *component*s with Java interfaces. The *decorator* receives the requests through the *component* interface and forwards them to the *concrete component*. It adds its specific, additional logic before or after forwarding. One problem often mentioned is that the identity of the *component* object is split between *component*s and *decorator*s. Clients cannot rely on object identity when *Decorator* is used.
In the OT/J implementations of both scenarios of *Decorator*, roles are the decorators, bound through role playing to the Java classes to be decorated. As is usually the case, instantiation of roles is carried out implicitly, by the team, in a way that is oblivious to code outside the team. The sole exceptions are the drivers for the examples (i.e., the classes with the `main` method), which reside at a different conceptual level and are responsible for setting up the object structure and instantiating and activating the teams. Callins are used to trigger the additional functionality provided by the *decorator* roles. This way, clients of the decorated classes (excluding the driver for

the example) can be oblivious of the decorators and see only the *component* objects. This approach solves the problem of split object identities.

The two examples of *Decorator* illustrate the use of a number of OT/J features for providing fine-grained control over which objects are decorated and when the *decorator*s intervene. The example by Cooper is used next to illustrate. In both examples, the aim is to ensure that roles are created and used just for some selected instances of the base classes. A straightforward use of callins cannot be used, as it would trigger the instantiation of roles for *all* base instances. It is important to note that whenever the target method of a callin is executed, the control flow is passed to the team and, if a role corresponding to the base object does not exist, it is created at that moment. If we want a more fine-grained control, one option is to use the *Object Registration* idiom, which is the case here.

The *Object Registration* idiom entails defining team-level methods to create the roles explicitly, using the lifting constructor (Listing 7, lines 15 and 19). In the examples, these methods are called by the driver to the example, which is the only piece of code that needs to be aware of the decorators. To ensure that callins are triggered for just the registered objects (i.e., the base objects that already have an associated role), the roles declare the appropriate guard predicates (Listing 7, lines 5 and 10). The team also controls the precedence between two roles bound to the same base object, i.e., which role method executes first (Listing 7, line 2). In all cases, the identity of the original decorated object is preserved throughout the execution.

```
01 public team class ButtonDecoratorTeam {
02   precedence CoolDecorator, SlashDecorator;
03
04   public class CoolDecorator playedBy MyButton
05   base when (ButtonDecoratorTeam.this.hasRole(base, CoolDecorator.class)) {
06     //...
07   }
08
09   public class SlashDecorator playedBy MyButton
10   base when (ButtonDecoratorTeam.this.hasRole(base, SlashDecorator.class)) {
11     //...
12   }
13
14   public MyButton addSlashDecorator(MyButton c){
15     new SlashDecorator(c);
16     return c;
17   }
18   public MyButton addCoolDecorator(MyButton c){
19     new CoolDecorator(c);
20     return c;
21   }
22 }
```

Listing 7. Illustration of the implementation of *Decorator* for the example by Cooper.

Proxy bears many similarities to *Decorator* and though its purpose is different, is can be similarly implemented. *Decorator* differs from *Proxy* in that decorators add one or more responsibilities to an object, while the intent of *Proxy* is to provide a surrogate or placeholder for another object to control access to it. The pattern defines participants *subject*, which is the object whose access is to be controlled, and *proxy*, which provides an interface identical to subject's and maintains a reference to it. The OT/J implementations of both scenarios for *Proxy* use roles to represent the *proxy*, which are bound to subject base classes through role playing. Thus, OT/J composes a *proxy* to its *subject* the same way *decorator*s are composed to *component*s. In both examples of *Proxy*, the approach is similar to that for *Decorator*, but simpler because there is just one proxy for a given subject, while there can be several decorators for a component. Partly for this reason, the OT/J examples for *Proxy* do not use advanced features such as guard predicates and precedence control. Like in *Decorator*, the primary difference between the OT/J implementation and one in plain Java is the use of role playing to compose proxies to subjects.

Façade

The purpose of *Façade* is to provide a unified interface to a set of modules in a subsystem. *Façade* defines a higher-level interface that makes the subsystem easier to use. The pattern defines participants *façade* and *subsystem classes*. The *façade* knows which subsystem classes are responsible for a request and forwards client requests to appropriate subsystem objects. The contribution of OT/J to the implementations of *Façade* is to provide more flexibility in composing the *façade* to the subsystem classes. In the Cooper example no special advantage in using teams is apparent and traditional aggregation is used. Role playing is not an option in this scenario as the subsystems classes are Java interfaces from a Java standard API. In the HK example, role playing is used to bind the *façade* team (or more to the point: its roles) to the *subsystem classes*.

Iterator
In addition to the approach to *Iterator* described in section 3.5, both the HK and Cooper scenarios for this pattern were also implemented using a different approach that accords to the approach highlighted in this section. Like in the approach from section 3.5, the iterators are unbound roles that implement a Java interface through which clients traverse the aggregate, i.e., the *Transparent Role* idiom is again used. The approach differs in that here, the team is not the aggregate. In the HK example, another role represents the aggregate, which is bound through role playing to a base class. The Cooper example is different, where the iterator is an unbound role that again uses the *Transparent Role* idiom and whose constructor receives the aggregate object. Note that this role constructor is not a lifting constructor, as the role is unbound. The constructor is called by the team object, which gets the aggregate from a team method called from client code. This approach is an option when the aggregate cannot be modified and cannot be subject to role playing, as is the case of proprietary classes available only in binary form. In this case, no advantage is noticeable with respect to Java, except for the enhanced extensibility of the team module (even if the use of *Transparent Role* compromises extensibility a bit, due to the use of a top-level Java interface).

Prototype
Prototype is about specifying a category of objects that are to be created through the copying of a prototypical instance. The pattern defines a *prototype* participant that declares an interface for cloning itself and *concrete prototype* participants that extend *prototype* and implement the cloning operation. *Client* participants use a *concrete prototype* through the *prototype* interface to create new objects by asking the *concrete prototype* to clone itself. The GoF book highlights the dynamic nature of this pattern as one advantage: entities can be manipulated at the instance level during runtime, which is more flexible than working at the static, class level.
Two different implementations of *Prototype* were developed. The one developed during the first development phase mimics the AspectJ approach [23], as is often the case with the examples developed in that phase. The AspectJ approach is based on a reusable aspect that composes a clone method to target classes. The aspect module refers to these classes in abstract terms through an inner marker interface that is the target of the inter-type declarations that compose clone(). The aspect also calls the clone method and handles the checked exception CloneNotSupportedException that is potentially thrown upon calling clone. The first OT/J implementation is based on a team using the same approach, representing the concrete prototype as a role that sub-teams bind to some case-specific class through role playing. To facilitate a comparison between the AspectJ and OT/J reusable implementations of this common approach, Listing 8 shows both side by side.
Both Java examples use the java.lang.Cloneable Java interface as the *prototype*, which presently could cause hurdles to an OT/J implementation because the language has limitations as regards role playing for Java interfaces. For this reason, roles representing *prototype* in concrete sub-teams bind to the concrete base classes directly.
The OT/J implementation developed in the second phase does not (re)use the PrototypeProtocol team. Instead, it uses *Object Registration* idiom to ensure the intended cloning effect is narrowed to just the target instances rather than all instances of a given base class.

Strategy
Strategy is about defining a family of algorithms relating to a given operation, encapsulating each algorithm, and making them interchangeable. The purpose is to let the algorithm vary independently from clients that use it. The pattern defines a *strategy* participant that declares an interface common to all supported algorithms. Several *concrete strategy* participants extend the *strategy* and define a concrete algorithm for implementing the operation declared by the *strategy*. A *context* participant maintains a reference to the *concrete strategy* object through the *strategy* type.
The OT/J implementations of both scenarios use roles to represent the *context* and *strategy* participants, which are bound to case-specific classes through role playing. The two examples vary slightly in their minute design decisions. In the HK example, the team has both an abstract role for representing the *strategy* in abstract terms

and a group of concrete sub-roles for representing the various *concrete strategies*, while the Cooper example uses just a single *strategy* role.

```
public abstract aspect PrototypeProtocol
{
   protected interface Prototype {}

   public Object Prototype.clone()
        throws CloneNotSupportedException {
      return super.clone();
   }

  public Object cloneObject(Prototype obj){
      try {
        return object.clone();
      } catch(CloneNotSupportedException e){
        return createCloneFor(obj);
      }
   }

   protected Object
   createCloneFor(Prototype object) {
      return null;
   }
}
```

```
public abstract team class PrototypeProtocol
{
   protected abstract class Prototype {
      public Object clone()
      throws CloneNotSupportedException {
         return super.clone();
      }
      public abstract Object deepClone();
   }

   public Object createClone(Prototype obj) {
      try {
         return object.clone();
      } catch(CloneNotSupportedException e){
         return createCloneFor(obj);
      }
   }
   public Object
   createCloneFor(Prototype object) {
      return null;
   }
}
```

Listing 8. Reusable modules for *Prototype* pattern in AspectJ and OT/J.

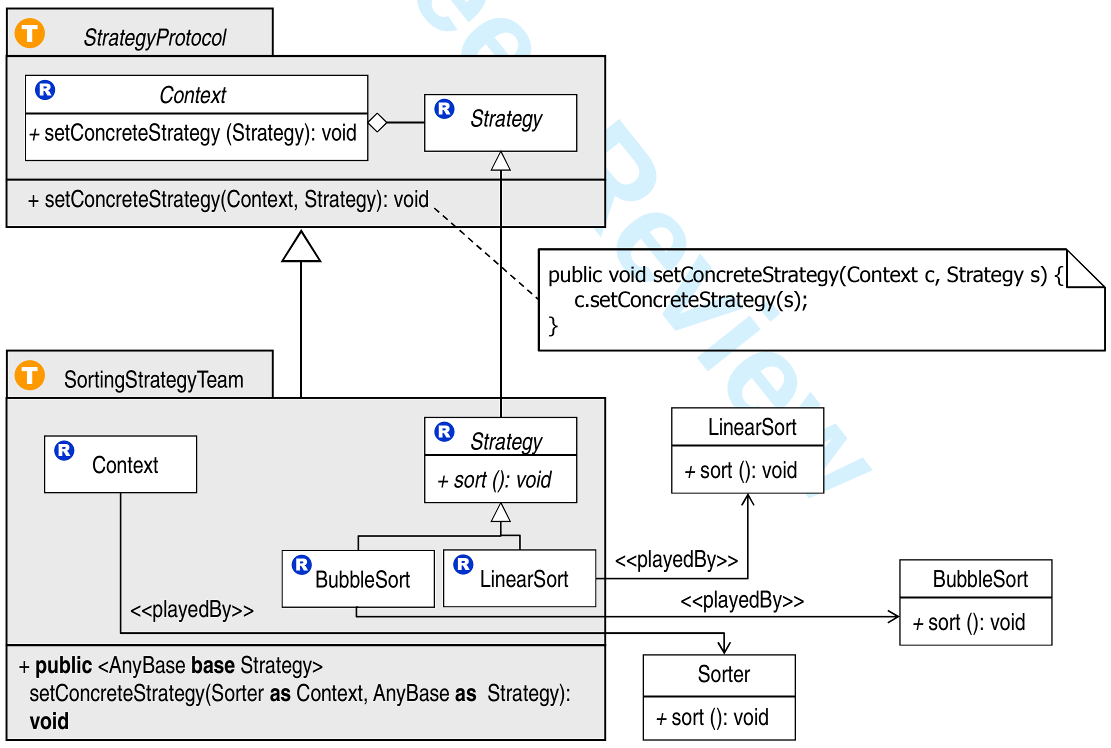

StrategyProtocol
Context
+ setConcreteStrategy (Strategy): void
Strategy
+ setConcreteStrategy(Context, Strategy): void
public void setConcreteStrategy(Context c, Strategy s) {
c.setConcreteStrategy(s);
}
SortingStrategyTeam
Context
Strategy
+ sort (): void
BubbleSort
LinearSort
<<playedBy>>
<<playedBy>>
<<playedBy>>
LinearSort
+ sort (): void
BubbleSort
+ sort (): void
Sorter
+ sort (): void
+ public <AnyBase base Strategy> setConcreteStrategy(Sorter as Context, AnyBase as Strategy): void

Figure 17. OT/J implementation of *Strategy* (HK scenario).

The way a connection between *context* and *strategy* is carried out also varies. In the HK example (Figure 17), it is made explicitly by a team-level method that uses declared lifting (called from the driver to the example). In the Cooper example, a callin in the Context sub-role triggers the assigning of a reference to the Strategy object to a field from the Context role. The abstract team StrategyProtocol is used in both scenarios (and examples, as there is just one example per scenario for this pattern). Team StrategyProtocol declares the relevant pattern roles and method setConcreteStrategy for assigning a strategy object to a context. Note that Context has access to Strategy through aggregation, which is inherited by sub-roles in sub-teams. The case-specific team SortingStrategyTeam is the most complex team of the two examples, as the Strategy sub-role heads a small role hierarchy. The strategy is used to vary a sorting algorithm, of which one instance is represented in each sub-role from the hierarchy. The module providing the *context* is class Sorter. Team SortingStrategyTeam can transparently compose the sort operations from either LinearSort or BubbleSort to the sort operation of Sorter.

3.7. No opportunities to improve over Java: *Singleton*, *Template Method*

In the two patterns from this section, no significant advantage over Java is apparent. The reasons vary, though. For this reason each pattern is presented separately.

Singleton

The purpose of *Singleton* is to ensure a class has just one instance and provide a global point of access to that instance. The usual approach is to give a private visibility the constructor or constructors of the class and make all clients to access the instance only through a method that returns the unique instance. Often, that method also creates the instance upon the first time it executes.

When using AspectJ, the usual approach – also used in the HK study – is to let the class be free of any code related to the singleton nature and for a poincut in an aspect to intercept the execution of the class' constructor, replacing the instance procuced by the constructor with a reference to the unique instance, which is kept by the aspect module. OT/J does not provide the means to replicate that approach, as callins do not work for class constructors. Therefore, the OT/J examples for *Singleton* are identical to those in Java. Indeed, *Singleton* is the sole pattern whose implementation in OT/J is identical to that in Java.

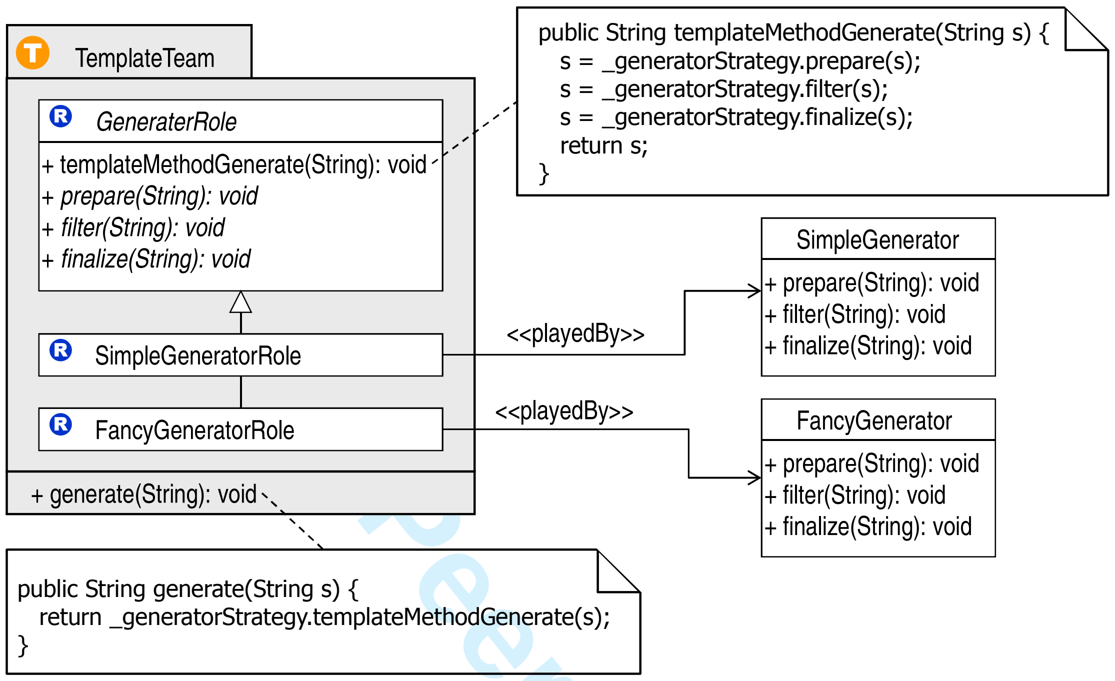
It is worth noting that it is debatable whether the singleton nature of a class should be separated into another module, as that nature is often considered intrinsic to the class. Nevertheless, the example is still useful to assess and illustrate the capabilities of a given language.

Template Method

The purpose of *Template Method* is to define the skeleton of an algorithm in an operation, deferring some steps to subclasses. An *abstract class* participant holds the method defining the skeleton (i.e., the template method), whose visibility may even be private, in which case it is inaccessible to sub-classes. The *concrete class* participants extend the *abstract class* and define methods for the deferred steps, often referred as *hook methods*. Using *Template Method*, subclasses are used to concretize or redefine the deferred steps of the algorithm without changing the algorithm's structure. In practice, *Template Method* amounts to using inheritance to avoid duplication in classes that have a common super-class, or that have enough commonalities to make it feasible to extract a common super-class, i.e., using the *Extract Super-class* refactoring [17]. The removal of duplication by factoring out common code to a super-class also provides the motivation for refactorings such as *Pull Up Method* and *Pull Up Field* [17]. Refactoring processes using such refactorings often yield instances of *Template Method*. Cooper argues that most uses of abstract classes entail simple forms of *Template Method* [10].

There is really no way to improve on the use of inheritance prescribed by *Template Method*. In their study, Hannemann and Kiczales recognize that the use of inheritance to distinguish different but related implementations is already nicely realized in OOP and their approach to implementing the pattern in AspectJ goes in a different direction, taking advantage of AspectJ's inter-type declarations to compose a default implementation to Java interfaces in place of abstract classes. The composed features are acquired by implementing classes, which is an advantage in languages supporting just single inheritance because the classes become free to inherit from some other class.

The OT/J implementations of *Template Method* do not improve on Java. The approach taken was to use the examples to illustrate how the pattern can be used along the role playing dimension instead of the dimension of traditional inheritance. The approach is similar to many other cases described in this paper. Roles declare abstract methods whose implementations are acquired through callouts from the associated base class. Thus, the roles are the *abstract class* participants and the base classes are the *concrete class* participants. This is illustrated in Figure 18. The abstract role GeneratorRole holds the template method (shown in a note from Figure 18) – templateMethodGenerate(String) – which calls three abstract methods also declared by that role. Those methods are (indirectly) concretized by base classes SimpleGenerator and FancyGenerator. Sub-roles SimpleGeneratorRole and FancyGeneratorRole establish the link between GeneratorRole and the base classes. The team method generate() (also shown in a note from Figure 18) enables the calling of the template method from outside the team.



```
public String templateMethodGenerate(String s) {
  s = _generatorStrategy.prepare(s);
  s = _generatorStrategy.filter(s);
  s = _generatorStrategy.finalize(s);
  return s;
}
```

```
public String generate(String s) {
  return _generatorStrategy.templateMethodGenerate(s);
}
```

Figure 18. Illustration of the *Template Method* pattern across the role playing dimension (HK scenario).

3.8. Object collaborations modularized: *Chain of Responsibility*, *Command*, *Observer*

All patterns from this section comprise reusable modularizations of entire collaborations between multiple participants. In contrast to the Java versions, the case-specific classes become oblivious to the parts they play in the patterns. The patterns and the collaboration they define are the following:

- *Chain of Responsibility* is meant to avoid coupling the sender of a request (*client*) to the receiver (*colleague*) by giving more than one object a chance to handle the request. The pattern prescribes the chaining of the receiving objects (*concrete colleagues*), which pass the request along the chain of multiple *concrete colleagues* until an object handles it.
- *Command* encapsulates a request as an object (*command*), thereby enabling clients to parameterize different requests, queue or log requests, and support undoable operations. In addition to *command* and *concrete commands*, the pattern also defines (1) the *invoker* participant that asks the *command* to carry out the request; (2) the *receiver* participant, which knows how to perform the operations associated with carrying out a request; (3) the *client*, which creates a *concrete command* and sets its *receiver*. In the examples developed, the participants that come most to the fore in object collaborations are *invoker*s and *concrete commands*.
- *Observer* is about defining a one-to-many dependency between objects so that when one object (*subject*) changes state, all its dependents (*observers*) are notified and updated automatically. As in most patterns, *Observer* defines abstract and concrete representations of the participants: *subjects* and *concrete subjects*, and *observer*s and *concrete observers*. The collaboration takes place between *subject*s and *observer*s so as to avoid direct dependencies between concrete participants.

In all the above patterns, the essence of the OT/J implementations is one team representing the collaboration between pattern participants in abstract terms, providing all the logic supporting the collaboration, in a way that is independent of any specific instance of the pattern. For each example, a sub-team adds the bindings between roles and the specific classes, as well as any glue code that may be required for the example. In all three patterns, the team representing the collaboration is potentially reusable and is indeed used in both scenarios. Figure 19 illustrates the approach, using the HK example of *Command*. In this example, participants Invoker and Command are represented by abstract roles that are concretized in a concrete sub-team, and bound to case-specific classes.

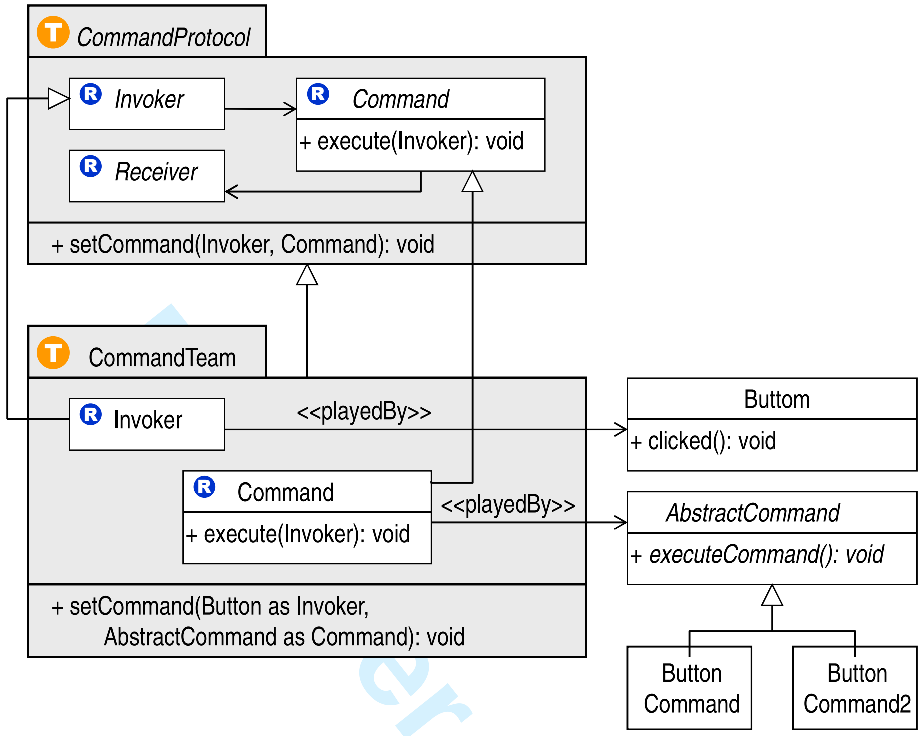


Figure 19. OT/J implementation of *Command* treating participants as superimposed.

Chain of Responsibility differs from the other patterns in that the participants in the collaboration are instances of the *same* participant – the colleagues. In the other patterns, the participants are instances of *different* participants. Though the purpose of each pattern is different, *Chain of Responsibility* can be considered a simpler variant of *Observer* from a structural point of view. The difference is that defines just one participant (*colleague*) instead of two (e.g., *subject* and *observer* for *Observer*).

A more subjective difference between *Command* and *Observer* should be pointed out. For this discussion, we must distinguish between the following two kinds of pattern roles:

- *Superimposed roles* are assigned to class modules that have functionality and responsibility outside the pattern. Participant classes playing such roles contain code pertaining to more than one role, which is a form of tangling and a case of the *Double Personality* smell [45]. Superimposed roles are indicative of the presence of latent aspects.
- *Defining roles* are completely defined by the pattern, with no functionality outside the context of the pattern. Removing a pattern implementation from a system entails removing all code related to such roles as well.

Both *Command* and *Observer* have in common a collaboration between different participants. *Observer* defines *subject* and *observer*, while in *Command* it is (mostly) *invoker* and *command*. However, the participants defined by *Observer* are superimposed roles is practically *all* cases, while the participants of *Command* are defining in a significant number of cases. A frequent example of defining roles for *Command* is the action to be carried out in a Java program when a GUI object receives click events. In Java, it is usual for the classes representing such actions to be anonymous classes that wouldn't exist if it were not for the pattern. Such cases are too simple to warrant the use of teams or aspects.

One advantage in treating the *command* participant as superimposed is that participants do not need to depend to an interface common to *invoker*s and *command*s. This is because the team composes these pattern roles to case-specific classes transparently, like in the other patterns from this group. In cases in which the roles of *Command* should be treated as defining, the OT/J implementation is likely to be identical to that in Java.

4. Comparison with AspectJ

A key goal of AOP is to provide a more systematic support for modularity than previously attained through traditional approaches [51]. In line with that goal, this section presents a comparison between the support for modularity provided by AspectJ and OT/J respectively. Two factors make it feasible to perform direct comparisons: (1) part of the implementations developed with OT/J are based on the same scenarios as those used in the study on AspectJ by Hannemann and Kiczales, which means those examples are directly comparable [23]; and (2) the modularity properties used in the analysis from the HK study can be used in this study as well, which facilitates comparisons between the two studies. Thus, this paper uses the results from the HK study in the comparison, using criteria that include the modularity properties used in the HK study. The present study also uses the extensibility property.

Hannemann and Kiczales claim that benefits from AspectJ are mainly felt in pattern implementations by *inverting dependencies*, i.e., making pattern code dependent of participants rather than the opposite, and keeping code related to management of dependencies within pattern aspect modules. The corollary of applying this principle is to achieve code obliviousness (section 2.1) for the pattern functionality. The OT/J implementations also follow this reasoning.

The AspectJ implementations are mostly based on: (1) use of pointcuts and advice, (2) use of inter-type declarations, and (3) a more elaborate technique, based on *marker interfaces*, used in all the reusable aspect modules except *Command*. The technique uses empty inner interfaces to represent pattern roles, which are often placed within an abstract aspect. The latter also includes functionality associated to the interfaces, comprising either inter-type declarations that compose state and/or behaviour to the interfaces, or aspect methods whose parameter types are the interfaces. Concrete aspects inherit this logic and use *declare parents* clauses to bind the marker interfaces (and associated logic) to concrete, case-specific classes. Some of the reusable aspects also use pointcuts and advice.

The marker interfaces are roughly equivalent to OT/J roles and the *declare parents* clauses used to bind case-specific classes to marker interfaces are roughly equivalent to the *playedBy* binding between roles and base classes. However, OT/J roles and the role playing relation are more expressive, due to roles being fully-fledged classes that cohesively enclose some logic. Marker interfaces, by contrast, are usually empty constructs and the logic to which they are associated is often placed outside the interfaces and within the aspect, i.e., at same level as the interfaces. This flat structure was criticized in the past by Mezini and Ostermann. It results in a rather procedural style of programming that is contradictory to one of the fundamentals of OOP, according to which a type definition contains all methods that belong to its interface. It is also contradictory to the aspect-oriented vision of defining crosscutting modules in terms of their own modular structure [42]. In addition, it and makes the AspectJ aspects inflexible to reuse and extend. By contrast, the support for virtual classes and family polymorphism on the part of OT/J means that internals of team modules are organized hierarchically, which avoids the limitations of a flat internal structure and it is more intuitive as it is easier to obtain a modular structure that closely corresponds to structure of concerns of the problem, as well as their relations.

The differences in outcomes from the features of the two AOPLs can be illustrated through the implementations of *Interpreter*. The AspectJ module for *Interpreter* is an aspect with just a set of inter-type declarations that add additional state and behaviour to the participant classes. The OT/J module for *Interpreter* is team structured the usual way, enclosing several roles plus top-level methods used to manipulate role instances.

A different limitation of AspectJ stems from the static nature of its advice. In AspectJ, advice cannot be activated or deactivated dynamically, which has an impact on the implementation of some patterns (e.g., *Decorator*). In contrast, OT/J provides the ability to (de)activate the OT/J constructs that correspond to AspectJ advice.

Some of the underlying design choices that mimic the AspectJ approaches can be debatable, particularly as regards whether the pattern participants should be considered defining or superimposed. However, much the same can be said of the AspectJ examples [44]. Mimicking the AspectJ approach has the advantage that the resulting examples in OT/J can be directly compared with their AspectJ counterparts. It also demonstrates the capability of OT/J to replicate the effects achieved with AspectJ.

4.1. Modularity properties

In their study, Hannemann and Kiczales use the following modularity properties [23]:

- **Locality,** the ability to place all code pertaining to a given concern in a module separate from the other modules. When the concern is a given pattern, locality entails placing all pattern code in a module that is separate from all case-specific (class) modules that (may) participate in the pattern. This separation has the benefit that case-specific modules are free from (i.e., oblivious to) implementation of the pattern. Full source code locality for a given concern is a prerequisite for a successful modularization of that concern, as well as all the following properties.

- **Reusability.** Ability of a given module to be applied to distinct scenarios/examples without the need for invasive changes on its source code.
- **Composition transparency.** Ability to compose multiple instances on a given pattern in such a way that composing an instance does not interfere with the composition of other instances. Note that this property is considered from the side of class participants, not clients of the resulting compositions. Depending on the specific circumstances, clients may need to be aware of the involvement of a given module in the various pattern instances, namely to set up structures, select a specific variant from the set of available choices, or carry out configurations.
- **(Un)pluggability.** Ability to add or remove a given module (and associated pattern functionality) from the concrete pattern participants, enabling a choice between using and not using the pattern. This property is to be evaluated in terms of the impact on participants, not on specific client code.

Extensibility
In addition to the above properties, we find it insightful to include *extensibility* as well. We define extensibility as the ability to further extend a pattern implementation non-invasively, through the addition of new modules that extend the functionality provided by the existing module. Effects of overriding definitions should propagate polymorphically to the clients of the original module. This notion of extensibility as proposed here has some relation to the *open-closed principle* that states that modules should be open for extension, but closed for modification [41]. Thus, client code using the module to be extended can also use the extensions of that module without the need for invasive changes. Of course, this is on condition of client code not using new operations from the module extension. If it does, the client code itself is undergoes invasive changes and is also being extended. Extensibility opens the way to *incrementality* in the sense used by Ernst [13].

Direct language support
In the analysis of their results, Hannemann and Kiczales mention a group of patterns whose implementations "disappear" due to direct support from AspectJ – including *Adapter*, *Decorator*, *Proxy*, *Strategy* and *Visitor*. However, they also acknowledge that those implementations have inherent limitations. For instance, their advice-based implementation of *Decorator* is devoid of dynamic properties, namely the ability to dynamically reorder decorators or distinguish between different instances of the decorated (*component*) class. These limitations motivate some qualifications to the extent to which the modularity properties were attained with AspectJ.
Table 1 presents the modularity properties obtained from the two AOPLs, organized by pattern. The results for AspectJ are taken from the HK study [23] and the results for OT/J are based on the part of the OT/J collection developed in the first phase, for the scenarios from the HK study. Keep in mind that the approaches taken in those implementations closely follow those used with AspectJ. Table 1 also includes the classification of pattern roles proposed by Hannemann and Kiczales into *defining* and *superimposed* [23]. Entries to Table 1 state whether a given property holds for a given pattern – "yes" or "no". However, there are qualifications to be pointed out for a number of cases. The HK study classifies the results obtained with AspectJ for some modularity properties with "(yes)" (instead of plain "yes") to indicate that limitations of some sort apply. Unfortunately, details on the specifics limitation felt with respect to each pattern are not provided. The qualified "yes" entries for the OT/J implementations also have this meaning.
Since locality is a prerequisite for the remaining properties, it is to be expected that a "no" for that property is followed by "no" for all other properties. That is indeed the case, for both AOPLs. However, the HK study has some cases in which a qualified "yes" entry (i.e., "(yes)") is followed by unqualified "yes" entries for the following properties – *Command* and *Visitor*. The HK study does not clarify that issue. We conjecture that the entries for the properties after locality are classified for the case to which locality still applies, which are the parts of the pattern implementation that were effectively modularized. In the entries relative to the OT/J implementations we try to be consistent across all properties and in a manner that enables comparisons between the two studies.

Table 1. Pattern roles and modularity properties of the OT/J and AspectJ implementations.

Pattern	Kinds of roles		Language	Locality	Reusability	Composition Transparency	Unpluggability
	Defining	Superimposed					
Abstract Factory	Factory, Product	–	OT/J	Direct language support			
			AspectJ	no	no	no	no
Adapter	Target, Adapter	Adaptee	OT/J	yes	no	yes	yes
			AspectJ	yes	no	yes	yes
Bridge	Abstraction, Implementor	–	OT/J	yes	no	yes	yes
			AspectJ	no	no	no	no
Builder	Builder, (Director)	–	OT/J	yes	no	no	no
			AspectJ	no	no	no	no
Chain of responsibility	–	Handler	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Command	Command	Invoker, Receiver	OT/J	(yes)	(yes)	yes	yes
			AspectJ	(yes)	yes	yes	yes
Composite	(Component)	(Composite, Leaf)	OT/J	yes	yes	(yes)	(yes)
			AspectJ	yes	yes	yes	(yes)
Decorator	Component, Decorator	Concrete-component	OT/J	yes	no	yes	yes
			AspectJ	yes	no	yes	yes
Façade	Façade	–	OT/J	yes	no	yes	yes
			AspectJ	Same implementation for Java and AspectJ			
Factory Method	Product, Creator	–	OT/J	Direct language support			
			AspectJ	no	no	no	no
Flyweight	Flyweight-factory	Flyweight	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Interpreter	Context, Expression		OT/J	yes	no	n/a	no
			AspectJ	no	no	n/a	no
Iterator	(Iterator)	Aggregate	OT/J	yes	no	(yes)	yes
			AspectJ	yes	yes	yes	yes
Mediator	–	(Mediator), Colleague	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Memento	Memento	Originator	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Observer	–	Subject, Observer	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Prototype	–	Prototype	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	(yes)	yes
Proxy	(Proxy)	(Subject)	OT/J	yes	no	yes	yes
			AspectJ	(yes)	no	(yes)	(yes)
Singleton		Singleton*	OT/J	Same implementation for Java and OT/J			
			AspectJ	yes	yes	n/a	yes
State	State	Context	OT/J	yes	no	n/a	no
			AspectJ	(yes)	no	n/a	(yes)
Strategy	Strategy	Context	OT/J	yes	yes	yes	yes
			AspectJ	yes	yes	yes	yes
Template Method	(Abstract-class), (Concrete-class)	(Abstract-class), (Concrete-class)	OT/J	(yes)	no	no	(yes)
			AspectJ	(yes)	no	no	(yes)
Visitor	Visitor	Element	OT/J	yes	yes	yes	yes
			AspectJ	(yes)	yes	yes	(yes)

In general, (yes) for a property means that some restrictions apply [23].
(*) The exact classification from the HK study is reproduced in the table but we do not subscribe to the view that the singleton role should be classified as superimposed.

4.2. Superimposed vs. defining pattern roles

Regarding the classification of pattern roles, the HK study acknowledges that the distinction between defining and superimposed roles is not always clear-cut. For this reason, they signal ambiguous cases by placing the role names within parentheses in either or both categories. These details are also reproduced in Table 1. When in doubt, Hannemann and Kiczales lean on approaching pattern roles as superimposed and taking advantage of AspectJ's composition capabilities to separate additional functionality from the core of the concern.

We have a disagreement over the classification proposed in the HK study as regards the classification of the *singleton* role in *Singleton* as superimposed. We regard the singleton nature of a class to be intrinsic to that class and for this reason we would classify the singleton role as defining, contrary to what appears in Table 1. It is worth noting that though the singleton nature of a given module may not translate into *explicit* client dependencies at the compilation level, it gives rise to subtle dependencies in terms of the programming style with which clients use the singleton module. Most of the modularity properties outlined above do not seem applicable to *Singleton*. For example, it does not make sense to talk of composing the pattern multiple times or even of plugging or unplugging *Singleton* from a given system. For this reason, we are skeptical towards the "yes" for unpluggability of *Singleton* in Table 1.

In their study, Hannemann and Kiczales conclude that benefits brought by the mechanisms of AspectJ are primarily felt when dealing with superimposed roles. Superimposed roles provide the opportunity to extract to a separate module the code associated to the pattern role, yielding some or all the properties mentioned above. In the presence of defining roles, however, there aren't multiple roles to separate, which poses difficulties for AspectJ to improve over Java. In contrast, OT/J has the option – unavailable to AspectJ – of using unbound roles to represent defining pattern roles and enclosing the whole object collaboration within a team.

4.3. Locality

In their paper, Hannemann and Kiczales note that a qualified "yes" for locality means that the pattern is localized in terms of its superimposed roles but the implementation of the remaining defining roles is still scattered throughout other, separate modules (e.g. state classes for *State*). The failure of AspectJ to yield the locality property (i.e., "no") for 5 patterns – *Abstract Factory*, *Bridge*, *Builder*, *Factory Method*, *Interpreter* – is due to the participants defined by those patterns being all unambiguously defining. *Façade* could be added to this list though it is a special case in that the Java and AspectJ implementations are identical. In addition, limitations are indicated for 5 other patterns: *Command*, *Proxy*, *State*, *Template Method* and *Visitor*. In all these cases, the pattern either includes a role that is unambiguously defining or it is debatable whether a given role is superimposed. As pointed out in section 3.8, the AspectJ implementation of *Command* applies only to cases in which it is feasible to treat the pattern roles as superimposed. The implementations for *Proxy* and *State* rely on advice, which operate statically and therefore entail some loss of flexibility. In cases of *Proxy* in which the subject and proxy participants must be different classes, the AspectJ version is identical to that in Java. *Template Method* is primarily about the use of traditional class inheritance and for this reason the implementations in both AOPLs are focused on side issues.

The AspectJ implementation of *Visitor* is based on inter-type declarations that compose the additional operations to the classes of the elements of the structure to be traversed. The design is based on an abstract aspect that declares several marker interfaces representing terminal and non-terminal nodes from the structure. Connections between the interfaces and case-specific classes are specified in concrete sub-aspects. One of the Java interfaces defined within the abstract aspect must be explicitly implemented by the case-specific modules, meaning that parts of the implementation are placed outside the aspect modules and therefore the pattern implementation as a whole is not fully localized. The OT/J variant based on this approach improves on the AspectJ implementation as regards locality because the parts explicitly defined by concrete participants are composed by the team through the role playing relation. The second OT/J approach supporting double dispatch (section 3.2) is more suitable but even the approach that mimics the AspectJ implementation achieves full code locality.

Many of the situations that are an obstacle to locality when using AspectJ can be circumvented with OT/J due to the capability of team modules to package multiple components into a single cohesive scope. This capability – provided by family polymorphism and virtual classes – has a wide-ranging impact on the OT/J implementations. In all patterns specifying more than one meaningful role, the teams provide the option to enclose pattern roles within a single module. Independently of other considerations specific to a given pattern, this enhanced cohesion brings benefits of its own. Thus, the implementation of most patterns can theoretically comprise a team with (possibly abstract) roles representing the pattern roles. This explains why the *locality* property applies in *all* OT/J cases in Table 1. With OT/J, there is always the option to package and encapsulate participants in a larger module.

Two patterns have a qualified "yes" for locality as regards the OT/J implementations. The OT/J *Command* is different from that in Java in just the cases in which participants can be treated as superimposed. The OT/J implementations *Template Method* is used with the role playing relation instead of traditional inheritance.

4.4. Reusability

Regarding the OT/J implementations, our criterion for including a "yes" for the reusability property in Table 1 is that the module aiming to represent the pattern be used in more than one scenario. Note that this is a more demanding criterion than that used by Hannemann and Kiczales, since their study comprises just one implementation per pattern. The HK study classifies some AspectJ implementations as reusable by deeming modules that are free from case-specific details as reusable. Note, however, that further assessing the actual reusability of the AspectJ implementations is out of the scope of this article. Our study additionally requires that modules have at least one concrete member (state and/or behaviour) to justify its existence in an implementation. In principle, that criterion would disqualify a "yes" for the *Iterator* in AspectJ, as all that is reusable is a Java interface – java.util.Iterator.

Our second development phase produced variant implementations for a number of patterns (not covered in this section), which virtually always entailed precluding the reusable module in preference to a more intuitive approach – the case of *Composite*, *Prototype* and *Visitor*. Since the implementations from the first phase are used in both scenarios, OT/J implementations of these patterns are classified as reusable in Table 1. The first OT/J approach to *Visitor* has the benefit full locality is achieved, which opens the way to achieve reusability as well (see also the considerations on locality in the previous section). Note also that the AspectJ approach only deals with two different kind of nodes from the structure to be traversed: terminal and non-terminal nodes. In cases where the structure contains more types of nodes, this approach cannot be used without modifications. The OT/J implementation based on the same approach seems capable to deal with a broader range of situations, as the team module can be flexibly extended. However, it is not clear at this point the extent to which OT/J can improve the AspectJ approach as regards scalability.

As regards *Command*, it can be argued that the OT/J implementation, which also mimics the AspectJ approach, is less intuitive than that in plain Java (section 3.8) and not really applicable to all cases. For this reason, Table 1 indicates that restrictions apply for *Command*.

The two languages yield broadly comparable results overall regarding reusability. Table 1 includes 12 patterns implemented by reusable modules (which in the case of *Iterator* is a Java interface) to be contrasted with 10 patterns in OT/J (teams in all cases). There are two groups of patterns which yield disparate results in terms of reusability: patterns whose results with AspectJ seem better than those with OT/J – *Abstract Factory* and *Factory Method* – and two patterns with the opposite outcome – *Iterator* and *Singleton*. The differences for *Abstract Factory* and *Factory Method* are explained by the fact that OT/J provides direct language support to these patterns, a result which we consider actually better than reusability. The difference for *Singleton* is explained by the joinpoint model of AspectJ covering constructors while the callins of OT/J cannot be used with the constructors of base classes.

The HK study includes *Iterator* in a group of patterns for which a reusable implementation was successfully derived. However, uniquely in the group of reusable AspectJ examples, there is no reusable aspect for *Iterator*. The difference between the AspectJ and Java versions is that a factory method in the iterator class from the Java version is instead placed in an aspect composing it to the original iterator class through AspectJ's inter-type declarations. The sole aspect module in the implementation of *Iterator* is a concrete aspect that depends on case-specific modules. The only module from that implementation that promises to be reusable is Java interface java.util.Iterator. With OT/J, the primary advantage brought by the language is the ability to package together the *aggregate* and *Iterator* pattern roles into a common team module.

4.5. Composition transparency

In neither of the studies is the claim to composition transparency tested with actual examples created specifically for the purpose. The classification according to composition transparency is based on the assumption that it would indeed be that case in actual systems.

Unsurprisingly, Table 1 shows a "no" for the AspectJ implementations for all patterns for which also failed to yield locality, which includes *Interpreter*. There are 3 patterns for which the composition transparency property is deemed not applicable ("n/a" in Table 1): *Interpreter*, *Singleton* and *State*. We agree with the classification but point out that composition transparency also seems hard to apply to *Builder* and *Template Method* in the general case. Table 1 also shows a "no" for the OT/J implementations for those two patterns.

A clear case in which composition transparency of an OT/J implementation is not attainable is the case in which the team holds the context for the pattern or collaboration (section 3.5) *and* the roles are unbound. That is the case of *Builder*, *Iterator* and *State*. Actually, a variant of *Iterator* was created whose roles are bound to a base class, in which case composition transparency is attainable. However, that specific implementation seems more contrived than the variant based on an unbound role – admittedly more limited in terms of composability. For this reason, Table 1 signals that composition transparency applies with limitations for *Iterator*.

4.6. (Un)pluggability

The property of pluggability is assessed from the point of view of the class participants. By this, we mean that if a given modular representation of a pattern is deemed unpluggable if it can be separated from the system and the concrete participants without invasive changes, in such a way that the participant class modules remain consistent and usable. This criterion applies mostly to superimposed pattern roles, as modules that play defining roles will be separated from the system along with the rest of the pattern implementation. Thus, pluggability is a property that is strongly dependent on whether a pattern prescribes superimposed roles, defining roles or a combination of both. This does not mean that unplugging the pattern will not have an impact on the rest of the system. In theory, there are client systems that depend on the combination of the class participants with the pattern implementation composed on them, while others are based on just the classes devoid of pattern logic. The point here is to assess whether removing the pattern from participants yields consistent modules that still make sense by themselves.

Next, remarks about the patterns for which the two AOPLs yield different results are provided. Two patterns are directly supported by language mechanisms: *Factory Method* and *Abstract Factory*. Therefore, in most cases the issue of plugging and unplugging those patterns amounts to a choice between using and not using the language features themselves. In these cases, it does not make much sense to reason in terms of modularity properties.

Differences in outcomes for *Bridge* are due to differences in language features. As regards *Command*, The HK study deems the AspectJ implementation as unpluggable without limitations while we consider there are some limitations in the OT/J implementation. The precise reasons for the classification of the AspectJ implementation are not clear and we conjecture that differences in outcomes for this pattern may be explained by different interpretations of a qualified "yes" entry. *Façade* is the sole pattern from the HK study in which the Java and AspectJ implementations are identical. However, it does not necessarily follow that a Java implementation is not unpluggable. The Java implementation of *Façade* from the HK study actually is. Naturally, the OT/J implementations of *Façade* are unpluggable as well. The AspectJ implementation of *Proxy* is classified as having limitations, while the OT/J is not. Hannemann and Kiczales remark that their implementation – based on pointcuts and advice – has inherent limitations, felt when the *subject* and *proxy* participants are two different objects. The OT/J can be used in such cases. However, Hannemann and Kiczales refer to the case of remote and virtual proxy: it is not clear at this point how well can OT/J cope with such cases. The differences in outcomes for *Singleton* are the same as mentioned in relation to reusability (section 4.4). The OT/J implementation of *State* resorts to unbound roles to yield an implementation that is more richly structured than the AspectJ implementation based on pointcuts and advice. However, using a team with unbound roles has the drawback that the implementation is case-specific and not unpluggable.

Considering that the OT/J implementation of *Template Method* is not applicable to the common cases but just to some cases involving the role playing relation, it may be surprising that Table 1 claims that *Template Method* supports unpluggability even if with limitations. However, in the limited cases in which *Template Method* can be used with the role playing relation, the pattern is indeed unpluggable, the same way a subclass is unpluggable from the superclass it extends. This is the general case of teams with bound roles. The OT/J implementation of *Visitor* that mimics the AspectJ approach can be deemed unpluggable, since it attains full locality.

4.7. Extensibility

Table 1 does not include a column for the extensibility property due to space constraints, but also because results for extensibility are surprisingly simple and regular, for both AOPLs. If such a column were included in Table 1, it would show a “no” for all cases of AspectJ and a “yes” for all cases of OT/J – though with limitations in some cases. The reason for the generalized “no” for AspectJ is due to that concrete aspect modules cannot be further extended through aspect inheritance. In fact, the design of AspectJ explicitly eschews polymorphism in most forms, save for the cases that AspectJ “inherits” from Java [14]. Another factor that imposes constraints on extensibility is that pattern roles are mostly represented by empty inner interfaces that reside at the same level as the members that act on them [42].

In contrast, any module in OT/J can be further extended through inheritance provided the final keyword is not used on its members. *Interpreter* provides a good illustrating example of this extensibility. The implementation of *Interpreter* from the HK scenario is organized as a super-team and a sub-team, but a single team could have been used instead. The division into two layers was made to reflect an equivalent conceptual division of the nodes. Other combinations of super and sub modules could easily have been produced. In most cases, teams can always be extended through team inheritance, and team instances enclosing role objects can be used polymorphically. In sum, OT/J has the capacity to support the incremental building of class hierarchies [13].

The sole exception we detected to such unfettered building of role hierarchies is in the case in which some class or interface is placed outside the team (i.e., the family class). The most notable case is arguably the implementation of *Composite* described in section 3.5 (not covered in the comparison with AspectJ), because the team itself implements a top-level interface. Some limitations apply to all implementations using the *Transparent Role* idiom, though they are not likely to be less problematic as it is a role, not the team that implements a top-level interface. In addition, those cases always have the option of forgoing the interface and exposing the role to the outside of the team. Nevertheless it is fair to qualify the extensibility property for those cases (in terms of Table 1 that would mean a “(yes)”).

4.8. Summing up

Table 2 summarizes the results of both languages as regards the modularity properties. Results suggest a slight advantage of OT/J over AspectJ, though it is fair to say that no language emerges as a clear winner overall. In terms of for direct language support for specific patterns, OT/J seems to emerge as the winner, on account of its support for *Factory Method* and *Abstract Factory*. However, an advantage in just two patterns does not seem pronounced enough for making a definite overall judgement. Definite judgements may also depend on the relative importance attached to individual patterns.

Table 2. Summary of both languages regarding number of patterns with a given modularity property

	Direct language support	**Locality**	**Reusability**	**Composition Transparency**	**Unpluggability**
OT/J	2	20	10	16	17
AspectJ	0	17	12	14	17

The advantage of OT/J over AspectJ is clearer as regards extensibility and in general, of how flexibly the resulting modules can be adapted or extended to new situations. The aspects of AspectJ are generally not extensible, while OT/J teams seem to be always extensible in a very flexible way. The few observed limitations are due to the specifics of a given pattern. OT/J also has the advantage over AspectJ that the mechanisms it supports work at the instance level rather than at the class level. For this reason, in OT/J there is less need for managing data structures that map objects to object-specific functionality, which yields simpler solutions. This advantage is important in the examples described in this paper, as many of the GoF patterns relate to relationships between individual instances rather than classes.

The introduction of this paper notes that though AspectJ and OT/J are classified as aspect-oriented, the approaches supported by the languages are very different. The primary distinguishing mechanism of AspectJ is pointcuts and advice, while OT/J relies on multiple dimensions of polymorphism. AspectJ yields better results than OT/J in cases that require a more wide-ranging and/or fine-grained joinpoint model. In the present study, the clear example is *Singleton*, in which AspectJ enables us to go further than OT/J due to its support for constructor joinpoints. However, we do not consider *Singleton* to be a convincing case for these capabilities, as argued in section 3.7. It is worth pointing out that the absence of quantification of constructor joinpoints on the part of OT/J may seem a limitation but it is defensible as the interception of object creation events comprises a very intrusive change on the semantics of the base language. *Singleton* is an illustration of the differences in design philosophy between the two AOPLs. Polymorphism is a well-known mechanism and its characteristics and benefits are also well-known. It enables some elements of a software system to be variation points that can be extended in a way that follows the open-closed principle [41], through the addition of new modules. The contribution of OT/J is basically to expand the supported dimensions of polymorphism and thus enrich the points in existing software systems that can vary.

To sum up the differences between the two languages, AspectJ and OT/J seem geared for different purposes. AspectJ is known to yield very good results when used for applications that perform "highly crosscutting" tasks of the kind provided by profilers, monitoring and instrumentation tools. The fine-grained joinpoint model of AspectJ is suitable for such tasks, which often provide a valuable contribution during development but are not always included in the product to be shipped to clients. However, AspectJ seems less suitable for the long-term support and evolvability of large architectures. OT/J is the opposite: it seems unsuited for the former but is very promising for the latter.

5. Related work

Many works exist, focusing on the impact of a given set of language features on the implementation of design patterns [5, 56, 20]. Some studies have been carried out on results obtained when using advanced language features for implementing specific patterns [36, 44], e.g., *Observer* [49, 6] and *Visitor* [48]. The study described in this paper is focused on a specific language rather than on the patterns. A few studies are in this vein. For instance, Schmager *et al.* carry out an assessment of the Go language using patterns and a framework [54]. The present study uses implementations of the well-known Gang-of-Four design patterns to perform an assessment of modularity and composition capabilities of Object Teams.

This work is a continuation of our previous work, which reports on the results obtained from an earlier version of the collection of OT/J implementations [21]. However, that work does not provide an in-depth analysis and does not take into account the implementations of some of the Cooper scenarios, whose development was incomplete at the time. The present paper is based on two complete collections, which enables thorough comparisons for all 23 GoF patterns in terms of reusability. The collection of implementations analysed here also includes a number of variants that generally yield more intuitive solutions at the expense of reusability.

A study closely related to ours is the study by Hannemann and Kiczales based on AspectJ implementations of all Gang-of-Four patterns. The analysis carried out by Hannemann and Kiczales is based on qualitative modularity properties used to compare the object-oriented (Java) and aspect-oriented (AspectJ) implementations. Our study reuses the scenarios created by Hannemann and Kiczales as part of our own study material and takes advantage of this common base to carry out a systematic comparison, using the same properties, between the results obtained with OT/J and those with AspectJ. However, the selection of those modularity properties was made with Java and AspectJ in mind and using just those four properties risks yielding biased results. For this reason, the present study includes the extensibility property in the analysis of results.

The study by Rajan [50] describes the implementations in the Eos language of all 23 GoF patterns using the HK scenarios and presents a comparative analysis of results. Eos is an aspect-oriented language that was developed to illustrate a model alternative to that of AspectJ. The model is based on the idea that the notions of class and aspect, which are clearly distinct in AspectJ (in terms of what each kind of module is able to express), can be unified in a module construct called the *classpect*. Eos is a proof-of-concept implementation of that model, built on top of the .Net platform and C# language. The main distinguishing characteristics of Eos are (1) a single, unified concept of classpect module that has the capabilities of both classes (such as explicit creation of first-class instances using new) and aspects (e.g., pointcuts and advice); (2) support for named methods only, eschewing nameless advice blocks, and a separate and explicit binding between joinpoints and methods; (3) a generalized, instance-level, advising model that supports implicit method invocation using before and after advice and overriding using around advice.

Rajan presents a comparative analysis of his Eos implementations and the AspectJ implementations from the HK study based of the same four modularity properties. Rajan claims a significant reduction in code size and number of members in some patterns (*Chain of Responsibility*, *Command*, *Composite*, *Mediator*, *Strategy* and *Observer*) and a closer and more precise support of the pattern's original intent (*Command*, *Composite*, *Decorator*, *Mediator*, *Observer* and *Strategy*). The improvements are due to a combination of instance-level advising and first-class aspect instances, which replace AspectJ's use of data structures for supporting mappings between objects and aspect behaviour and ensures only the participant objects and subject to aspect compositions. In contrast, it sometimes happens that the AspectJ implementation advises all instances of a class when just a subset of its instances participate in a pattern. Since OT/J also supports instance-level composition, it also yields similar advantages over AspectJ.

Hirschfeld *et al.* [31] discuss design pattern implementation in AspectS, using two patterns to illustrate – *Visitor* and *Decorator*. AspectS [30] is an extension of the Squeak/Smalltalk environment that extends the Smalltalk meta-object protocol to support a number of AspectJ-like constructs including pointcuts, advice and inter-type declarations. It does so without changing Smalltalk's syntax or its virtual machine and instead makes use of meta-object composition and method-call interception. Contrary to those of AspectJ, the mechanisms of AspectS can work at the instance level. In their discussion, Hirschfeld *et al.* distinguish between an *AOP Representation of Design Pattern Solution* and a *Native AOP Solution*. The former is an implementation of what is essentially the original, object-oriented approach using aspect-oriented constructs. No fundamental redesign to leverage AOP constructs is carried out. The benefits of this approach are better code locality, reusability, composability, implementation modularity and comprehensibility. He latter is a redesign based on AOP-specific constructs. Hirschfeld *et al.* claim that native AOP solutions avoid certain drawbacks like model bloat and messaging-overhead caused by indirection levels, and context-dependent change of identity as a consequence of placing intermediate objects mediating between two instances. AOP native solutions also eliminate of glue code that the new language mechanisms render obsolete, which simplifies design and implementation. In this paper, we present two approaches to implementing *Visitor* that correspond to these two categories. The implementation supporting double dispatch (section 3.2) is a native AOP solution (avoiding explicit *accept* and *visit* methods), while the earlier implementation (section 4.3) mimicking the AspectJ implementation from the HK study (where the *accept* and *visit* methods are still in place) is an AOP representation of a design pattern solution.

The studies by Garcia *et al.* [19] and Cacho *et al.* [8] use adapted versions of the material produced by Hannemann and Kiczales [23] to perform comparisons between the object-oriented and aspect-oriented implementations based on a set of quantitative metrics. The present study does not include quantitative metrics, but like the examples from the HK study, the examples in OT/J open the way for subsequent quantitative studies. Quantitative studies focusing on OT/J code are likely to require new metrics, e.g., to account for the (relatively) novel cases of virtual classes (roles) and family classes (teams), plus the various new kinds of inheritance and polymorphism to which these give rise. An exploratory study of the metrics supported by the OTDT Eclipse plug-in was explored as part of its preparation [39], using the subset of examples in Java and OT/J that is presented in our SAC/OOPS paper [21]. Unfortunately, the study concludes that the metrics support is unsuitable for comparisons between Java and OT/J, as the plugin does not take into account the constructs specific to OT/J.

Sousa and Monteiro report a study of CaesarJ [55] that has many similarities with the present study, though it it less comprehensive and developed. The approach is similar, relying on the implementation in CaesarJ of pre-existing pattern examples in Java. As acknowledged by the authors, this study is preliminary in nature, covering just seven patterns and therefore does not seem sufficient for systematic comparisons of aggregate results. Family polymorphism as supported by CaesarJ brings benefits similar to those reported here, including direct language support for *Abstract Factory*.

The CaesarJ model [42, 4] bears many similarities with the two language models compared in this paper. In terms of language design and features supported, CaesarJ can be said to lie between the AspectJ and OT/J. Like OT/J but unlike AspectJ, CaesarJ supports virtual classes and family polymorphism. Like AspectJ but unlike OT/J, CaesarJ uses pointcuts and advice to compose aspect components to specific applications and software systems. In addition, CaesarJ supports the ability to polymorphically switch between alternative implementations of a given component, without impact on the remaining modules that make up that component. That ability results from CaesarJ's decoupling of a software component's implementation and the binding of that component to the remainder of the software system. This is achieved through a form of multiple inheritance yielding separate inheritance hierarchies for the component's implementation and bindings respectively. Both hierarchies initiate at a *collaboration interface* module that declares the abstract operations for which the implementation hierarchy provides concrete implementations and the binding hierarchy provides bindings. The collaboration interface provides the contract through which implementations and bindings share common concepts and operations. A final, often very simple module inherits from all hierarchies to yield an instantiatable component. Pointcuts and advice come less to the fore in this model and are used mainly to compose the component to concrete software systems and applications. This model based on collaboration interfaces and multiple inheritance is neither supported by AspectJ nor OT/J.

Nordberg [46] proposes a set of principles for managing dependencies between modules in complex systems, which lead to module structures that more stable, easier to understand and more maintainable. Summarily, the principles state that (1) dependencies must not form cycles, (2) modules should depend on abstractions (e.g., declarations of interfaces) and not on concrete elements and (3) the direction of dependencies should always be of less stable modules depending of more stable modules. Nordberg next provides an analysis of several design patterns in light of the dependencies they give rise to and identifies several cases in which traditional OOP implementations fail to meet the principles and points out a number of AOP-specific solutions that promise to solve or ameliorate the problems. For instance, traditional OOP implementations of *Visitor* violate all three principles: the dependency from the *visitor* participant to *concrete element* goes from abstract to concrete, goes from stable to less stable, and forms a cycle. Nordberg discusses ways in which AOPLs can invert those dependencies so as to follow the principles. However, the discussion is made in terms of the AspectJ-like kind of AOPL rather than the kind of solution supporting double dispatch described in section 3.2. The present study does not use the principles proposed by Nordberg in the analysis of the OT/J implementations it describes, but it may be fruitful to include them in future analyses.

6. Future work

The aforementioned works have that material used are toy examples. An obvious next step is to use the insights derived to study more complex and realistic systems, particularly systems whose designs use the patterns assessed in this paper, namely object-oriented frameworks. Another front is the assessment of the impact that AOPLs have on framework design and development. In particular, *pattern density* [53] (i.e., classes participate in a significant number of different patterns simultaneously) comprises a negative symptom that AOP languages seem well positioned to tackle. It would thus be interesting to assess how AOP constructs and languages can ameliorate or remove the symptom of pattern density from object-oriented frameworks.

The analysis presented in this paper is qualitative in nature. It should be complemented with quantitative analyses, in the same vein as those by Garcia, Cacho *et al.* [19, 8] and similar studies. Opportunities for future work should not only on the modularity of composability attained through the language, but also on observable benefits derived. For instance, Greenwood *et al.* describe a study on the impact of AOPLs on design stability of a software system in the context of an evolution scenario [22]. The study covers ten releases of the system and using versions in three languages – Java, AspectJ and CaesarJ respectively. Similar studies focusing Object Teams could also be carried out. One particularly interesting prospect would be to extend the study described by Greenwood *et al.* to cover Object Teams as a fourth language.

Another front is to extend the aforementioned kinds of study to other advanced programming languages. For instance, Scala [47] is a language that also supports virtual classes and family polymorphism and it has a number of mechanisms that promise to directly support a number of GoF patterns. In addition, it seems capable of emulating a number of AspectJ-like features, even AspectJ-like advice [48]. It therefore comprises an interesting subject for comparative assessment.

7. Conclusion

This paper describes a study of the Object Teams programming language [29, 3] based on two complete collections of implementations of all 23 Gang-of-Four design patterns [18]. An analysis of the code examples is provided, comparing them with functionally equivalent examples in Java, the base language of Object Teams:

- The *Factory Method* and *Abstract Factory* patterns are directly supported by Object Teams owing to supporting virtual classes [40] and family polymorphism [12] respectively.
- Object Teams also provides partial support for *Memento* and *Visitor*. The *memento* participant in *Memento* can enjoy stricter encapsulation rules than is the case with Java.
- *Visitor* is also well supported due to Object Teams' capacity to support double dispatch.
- The capability of teams to enclose the context for their roles enables highly cohesive implementations of 7 patterns: *Builder*, *Composite*, *Flyweight*, *Interpreter*, Iterator, *Mediator* and *State*.
- Role playing provided additional options to compose pattern roles in 7 patterns: *Adapter*, *Decorator*, *Façade*, *Iterator*, *Prototype*, *Proxy* and *Strategy*.
- Object Teams does not seem to provide special benefits for implementations of *Singleton* and *Template Method*. In case of *Singleton*, this is due to Object Teams not supporting the interception of constructor

calls or executions. In the case of *Template Method*, it is due to this pattern being about the use traditional inheritance, though the pattern can also be used along the role playing dimension.
- Object Teams is capable of modularizing entire object collaborations, which is demonstrated in the examples for *Chain of Responsibility*, *Command* and *Observer*.

A comparison with AspectJ [33] is presented, using implementations in Object Teams of the same complete collection that was used in a previous study of AspectJ [23]. The scenarios from that collection are now available in three languages: Java, AspectJ and Object Teams. A comparative analysis of Object Teams and AspectJ is presented, in terms of modularity properties including code locality, reusability, pluggability, composition transparency and extensibility. We summarize the results obtained as follows:
- AspectJ yields the results similar to Java for *Façade* and Object Teams yields results similar to Java for *Singleton* and *Template Method*.
- The AspectJ implementations fail to yield code locality for 6 patterns (*Abstract Factory, Bridge, Builder, Façade*, *Factory Method* and *Interpreter*). Object Teams yields code locality in *all* 23 patterns. Note this is because there is locality in the implementation of *Singleton*, though it is identical to that in Java.
- The AspectJ implementations number 12 reusable modules, as opposed to 10 reusable modules for Object Teams. However, one the reusable modules from the AspectJ collection is a Java interface rather than an aspect module (*Iterator*) and Object Teams also provides direct language support for two creational patterns: *Factory Method* and *Abstract Factory*.
- Object Teams attains composition transparency in 16 patterns, compared to 14 for AspectJ. Note we consider this property is not applicable to *Interpreter*, *Singleton* and *State*. If we exclude those patterns plus those for which Object Teams provides direct support, the language fails to provide composition transparency for just *Builder* and *Template Method*.
- The number of pattern implementations found to be (un)pluggable is the same for both languages – 17 – though the exact set of patterns differs.
- Due to (seemingly intended) limitations on its design, none of the AspectJ modules is extensible [14]. By contrast, all team modules are extensible with the exception of one of the variants for implementing *Composite*, to which limitations apply due to the team implement a top-level Java interface.

In addition, support for composition at the instance level on the part of Object Teams tends to yield somewhat simpler implementations when compared with those in AspectJ, whose constructs work at the static class level. A number of AspectJ implementations need to include additional state and behaviour to manage instance-level relationships. However, Object Teams requires additional code in a number of implementations to pass objects from the team to instances of its roles.

Summing up, AspectJ and OT/J seem geared for different purposes. AspectJ seems suitable for applications that perform "highly crosscutting" tasks that take a program as its input data, e.g., profilers, monitoring and instrumentation tools. However, it seems unsuitable for the support of large architectures and long-term evolvability. OT/J is the opposite: it seems less suited for the former category of tasks but is very promising for the latter.

Acknowledgements

We thank Stephan Herrmann for providing insightful feedback on a draft of this paper.
This work is partially supported by projects AMADEUS (POCTI, PTDC/EIA/70271/2006) and PRIA (UTAustin/CA/0056/ 2008) funded by Portuguese FCT/MCTES and FEDER.

References

1. The Demeter/Java project web site. www.ccs.neu.edu/home/lieber/Demeter-and-Java.html [14 July 2011].
2. The Object Teams web original site. www.objectteams.org/. [1st August 2011].
3. The Object Teams at eclipse. www.eclipse.org/objectteams/. [1st August 2011].
4. Aracic I., Gasiunas V., Mezini M., Ostermann K. An Overview of CaesarJ. *Transactions on Aspect-Oriented Software Development I*. Springer LNCS vol. 3880, 135-173, 2006. DOI: 10.1007/11687061_5.

5. Baumgartner G., Läufer K., Russo V. F. On the interaction of Object-oriented Design Patterns and Programming Languages. Technical report CSD-TR-96-020, Purdue University, Feburary 1996.
6. Bernardi M.L., Lucca G.A. Improving Design Pattern Quality Using Aspect Orientation. Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice, 2005; IEEE 206 – 218. DOI: 10.1109/STEP.2005.14.
7. Brichau J., Haupt M. Report describing survey of aspect languages and models. AOSD-Europe Deliverable D12, AOSD-Europe-VUB-01, May 2005.
8. Cacho N., Sant'Anna C., Figueiredo E., Garcia A., Batista T., Lucena C. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proceedings of the 5th International Conference on Aspect Oriented Software Development (AOSD 2006), ACM press, 109-121, 2006. DOI: 10.1145/1119655.1119672.
9. Colyer A., Clement A., Harley G., Webster M. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools. Addison-Wesley 2004.
10. Cooper J. *Java design patterns: a tutorial*, Addison-Wesley: Reading, MA, 2000.
11. Elrad T (moderator) with panelists Aksit M, Kiczales G, Lieberherr K, Ossher H. *Discussing aspects of AOP*, Communications of the ACM 2001; 44(10):33–38. DOI: 10.1145/383845.383854.
12. Ernst E. Family polymorphism. Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP 2001). Springer-Verlag LNCS vol. 2072, 303-326, 2001. ISBN: 3-540-42206-4.
13. Ernst E. Higher-Order Hierarchies. Proceedings of the 17th European Conference on Object-Oriented Programming (ECOOP 2003), Springer-Verlag LNCS vol. 2743, 303-329, 2003. DOI: 10.1007/b11832.
14. Ernst, E. and Lorenz, D. H. Aspects and polymorphism in AspectJ. Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD 2003), ACM press, 150-157, 2003. DOI: 10.1145/643603.643619.
15. Filman R.E., Elrad T., Clarke S., Aksit M (eds). *Aspect-Oriented Software Development*, Addison-Wesley: Reading, MA, 2005.
16. Filman RE, Friedman DP. Aspect-oriented programming is quantification and obliviousness. Chapter 2 of: [15], Addison-Wesley: Reading, MA, 2005, 21–35.
17. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading, MA, 1999.
18. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
19. Garcia A, Sant'Anna C, Figueiredo E, Kulesza U, Lucena C, Staa A. Modularizing design patterns with aspects: a quantitative study. *Transactions on Aspect-Oriented Software Development I*. Springer-Verlag LNCS vol. 3880, 36–74, 2006. DOI: 10.1007/11687061_2.
20. Gibbons J. Design patterns as higher-order datatype-generic programs. Proceedings of the 2006 ACM SIGPLAN workshop on Generic programming, ACM press 2006; 1-12. DOI: 10.1145/1159861.1159863.
21. Gomes J., Monteiro M.P. Design pattern implementation in Object Teams. Proceedings of the 25th Symposium on Applied Computing - Special track on Object Oriented Programming Languages and Systems (SAC/OOPS 2010). ACM Press, 2119-2120, 2010. DOI: 10.1145/1774088.1774534.
22. Greenwood P., Bartolomai T., Figueiredo E., Dosea M., Garcia A., Cacho N., Sant' Anna C., Soares S., Borba P., Kulesza U., Rashid A. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP 2007), Springer-Verlag, LNCS vol. 4609, 176-200, DOI: 10.1007/978-3-540-73589-2_9.
23. Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ. Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002). ACM press, 161–172, 2002. DOI: 10.1145/582419.582436.
24. Herrmann, S. A precise model for contextual roles: the programming language ObjectTeams/Java. Applied Ontology, vol.2 (2), 181-207, IOS Press, 2007.
25. Herrmann, S. Composable Designs with UFA. In Workshop on Aspect-Oriented Modeling with UML at 1st International Conference on Aspect Oriented Software Development, Enschede, The Netherlands, 2002.
26. Herrmann S. Confinement and Representation Encapsulation in Object Teams. Technical Report 2004/06, Fak. IV, Technical University Berlin, 2004.
27. Herrmann S. Object Teams: Improving Modularity for Crosscutting Collaborations. Proceedings of Net.ObjectDays, 248-264, 2002.

28. Herrmann S., Hundt C., Mehner K. Translation Polymorphism in Object Teams. Technical Report 2004/05, Fak. IV, Technical University Berlin, 2004.
29. Herrmann S., Hundt C., Mosconi, M. ObjectTeams/Java Language Definition version 1.3 (OTJLD). Technical Report 2009/08, Technische Universität Berlin, 2009.
30. Hirschfeld, R. AspectS – Aspect-Oriented Programming with Squeak. Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, 216-232, 2003.
31. Hirschfeld R., Lämmel R., Wagner M. Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. In 3rd German GI Workshop on Aspect-Oriented Software Development, 8 pages, 2003.
32. Ingalls D. A simple technique for handling multiple polymorphism. Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'86), ACM SIGPLAN Notices, 21(11), 347-349, 1986. DOI: 10.1145/960112.28732.
33. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An Overview of AspectJ, Proceedings of 15th European Conference on Object-Oriented Programming (ECOOP 2001), Springer-Verlag, LNCS vol. 2072, 327–335, 2001.
34. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect-oriented programming. Proceedings of 11th European Conference on Object-Oriented Programming, Jyväskylä, Finland, 1997 *(Lecture Notes in Computer Science*, vol. 1241), Aksit M, Matsuoka S (eds.). Springer: Berlin, Germany, 1997; 220–242.
35. Koppen C, Störzer M. PCDiff: attacking the fragile pointcut problem. *Proceedings of the Interactive Workshop on Aspects in Software*, Berlin, Germany, 2004.
36. Kouskouras K., Chatzigeorgioua A., Stephanides G. Facilitating software extension with design patterns and Aspect-Oriented Programming. Journal of Systems and Software Vol. 81 (10), October 2008, 1725-1737. Selected papers from the 30th Annual International Computer Software and Applications Conference (COMPSAC), Chicago, September 7–21, 2006. DOI:10.1016/j.jss.2007.12.807.
37. Laddad R. AspectJ in Action, second edition. Manning 2010.
38. Lesiecki N. *Enhance design patterns with AspectJ*, Part 2, AOP@Work series at developerWorks, IBM, 2005.
www.ibm.com/developerworks/java/library/j-aopwork6/index.html [25 April 2007].
39. Lima A., Goulão M., Monteiro, M.P. *Evidence-Based Comparison of Modularity Support Between Java and Object Teams*. First workshop for Empirical Evaluation of Software Composition Techniques (ESCOT 2010), Rennes France, 2010.
40. Madsen O. L., Moller-Pedersen B. *Virtual classes: a powerful mechanism in object-oriented programming*. OOPSLA'89, New Orleans, Louisiana, USA, 1989.
41. Meyer B., *Object-Oriented Software Construction*, second edition, Prentice Hall, 1997.
42. Mezini M, *Ostermann K. Untangling crosscutting models with Caesar*. Chapter 8 of: [15], Addison-Wesley: Reading, MA, 2005; 165-199.
43. Monteiro M.P., Fernandes J.M. *An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms*. Software: Practice and Experience 38 (4), John Wiley & Sons, 361-396, 2008. DOI: 10.1002/spe.835.
44. Monteiro M. P., Fernandes J. M. *Pitfalls of AspectJ Implementations of Some of the Gang-of-Four Design Patterns*. DSOA'2004 workshop at JISBD 2004 (IX Jornadas de Ingeniería de Software y Bases de Datos), Málaga, Spain, November 2004.
45. Monteiro MP, Fernandes JM. Towards a catalogue of aspect-oriented refactorings. *Proceedings of the 4th International Conference on Aspect-Oriented Software Development*, Chicago, Illinois, 2005. ACM Press: New York, NY, 2005; 111–122. DOI: 10.1145/1052898.1052908.
46. Nordberg III M. Aspect-oriented dependency management. Chapter 24 of: [15], Addison-Wesley: Reading, MA, 2005; 557-584.
47. Odersky M., Altherr P., Cremet V., Dragos I., Dubochet G., Emir B., McDirmid S., Micheloud S., Mihaylov N., Schinz M., Stenman E., Spoon L., Zenger M. *An Overview of the Scala Programming Language, 2nd Edition*. Technical Report LAMP-REPORT-2006-001, École Polytechnique Fédérale de Lausanne, 2006.

48. Oliveira B., Wang M., Gibbons J. The Visitor Pattern as a Reusable, Generic, Type-Safe Component. *Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages and Applications*, 2008; ACM press, 439-456. DOI: 10.1145/1449764.1449799.
49. Piveta E.K., Zancanella L.C. Observer Pattern using Aspect-Oriented Programming. 3rd Latin American Conference on Pattern Languages of Programming (SugarLoafPLOP 2003), 2003.
50. Rajan H., Design pattern implementations in Eos. Proceedings of the 14th Conference on Pattern Languages of Programs (PLoP '07), 2007; ACM 9:1-9:11. DOI: 10.1145/1772070.1772081.
51. Rashid A., Moreira A. Domain Models Are NOT Aspect Free. Proceedings of Model Driven Engineering Languages and Systems (MODELS'06), 2006; Springer 155-169. DOI: 10.1007/11880240_12.
52. Reenskaug T. Working with Objects – The OORAM Software Engineering Method. Prentice Hall, 1996.
53. Riehle, D., Brudermann, R., Gross, T. and Mätzel, K.-U. Pattern density and role modelling of an object transport service. ACM Computing Surveys, vol.32, ACM press, 2000. DOI: 10.1145/351936.351946.
54. Schmager F., Cameron N., Noble J. GoHotDraw: evaluating the Go programming language with design patterns. 2nd workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2010), ACM press, 2010; 10:1-10:6. DOI: 10.1145/1937117.1937127.
55. Sousa, E. and M. P. Monteiro. Implementing design patterns in CaesarJ: an exploratory study. Proceedings of the 2008 AOSD workshop on Software engineering properties of languages (SPLAT 2008), ACM press, 2008; 6:1-6:6. DOI: 10.1145/1408647.1408653.
56. Sullivan G. T. Advanced programming language features for executable design patterns: Better patterns through reflection. Artificial Intelligence Laboratory Memo AIM-2002-005, Artificial Intelligence Lab, MIT, Mar. 2002.
57. Sullivan K, Griswold W., Song Y., Cai Y., Shonle M., Tewari N., Rajan H. Information Hiding Interfaces for Aspect-Oriented Design. Proceedings of the 5th joint meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'05). ACM press, 2005; 166-175. DOI: 10.1145/1095430.1081734.
58. Wloka J., Hirschfeld R., Hänsel J. *Tool-supported Refactoring of Aspect-oriented Programs*. Proceedings of the 7th International Conference on Aspect-Oriented Software Development, Brussels, Belgium, 2008. ACM Press, 2008; 132-143. DOI: 10.1145/1353482.1353499.
59. Bin Xin, Sean McDirmid, Eric Eide, and Wilson C. Hsieh. A comparison of Jiazzi and AspectJ for feature-wise decomposition. Technical Report UUCS–04–001, University of Utah, March 2004.
60. Yuen I., Robillard M. Bridging the Gap between Aspect Mining and Refactoring. Proceedings of the 3rd workshop on Linking Aspect Technology and Evolution (LATE 2007). ACM press, 2007. DOI: 10.1145/1275672.1275673.
61. Wampler D. Aspect-Oriented Design Principles: Lessons from Object-Oriented. Industry track paper presented at AOSD'07, Vacouver, Canada, March 2007. http://www.aosd.net/2007/program/industry/I6-AspectDesignPrinciples.pdf [1 August 2011].