SOFTWARE: PRACTICE AND EXPERIENCE
Softw. Pract. Exper. (2012)
Published online in Wiley Online Library (wileyonlinelibrary.com). DOI: 10.1002/spe.2154

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

SUMMARY

This paper presents a study of the support for modularity of Object Teams, an aspect-oriented language backwards compatible with 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 advantages of Object Teams over Java with respect to modularity, module composition and reuse. We present a systematic comparison with a functionally equivalent collection of implementations in AspectJ, regarding five modularity properties: locality, reusability, composition transparency, (un)pluggability and extensibility. Object Teams yields broadly comparable results in terms of the first four properties. Object Teams yields better results as regards flexible module extensibility, composition at the instance level and enclosing multiple pattern participants into a larger, cohesive module. AspectJ is more successful than Object Teams in the *Singleton* pattern because of its ability to intercept constructor events. Copyright © 2012 John Wiley & Sons, Ltd.

KEY WORDS: aspect-oriented programming; design patterns; modularity; extensibility; reusability

1. INTRODUCTION

Aspect-oriented programming (AOP) is an emerging programming model primarily focused on the modularization of crosscutting concerns [1–5]. AOP is undergoing maturation, and many aspect-oriented programming languages (AOPLs) have been proposed [6]. Most of such languages are backwards-compatible extensions to existing languages, among which, Java features prominently. There is 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 Object Teams/Java (OT/J) are backwards-compatible extensions to Java and are both labelled aspect oriented. Yet, the mechanisms provided by the two languages are markedly different. Given the wide variety of proposals for AOPLs, it would be desirable that studies were made reporting on the relative strengths and limitations of the various languages. However, few studies were made comparing aspect-oriented languages. The few reports available are by the creators of the language concerned. For most AOPLs, there is a dearth of studies by independent authors. Although 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 [7–10]. There have been very few examples reporting on comparisons *among* AOPLs [11,12]. Among the AOPLs compared, AspectJ features prominently.

This paper presents a study of the AOPL OT/J [13–17]. It discusses OT/J by comparing it with Java – its base language – and AspectJ [18–20], which is used as the reference AOPL. The comparisons of OT/J with both languages are driven by implementations of the Gang-of-Four (GoF) design patterns [21].

*Correspondence to: Miguel P. Monteiro, CITI, Departamento de Informática, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, 2829-516, Caparica, Portugal.
†E-mail: mtpm@fct.unl.pt

Design patterns comprise a suitable material to assess and compare languages for various reasons. The GoF patterns are very well known and implemented in many different languages [22–24]. The GoF patterns present a significant variety and richness of composition and design problems even with relatively simple examples. Often, the composition effects illustrated by patterns come handy in real systems but are not directly supported by the language used. For instance, *Decorator* relates to mixin composition [25], and *Visitor* relates to double dispatch, none of which are directly supported by Java. Patterns provide commonly practised solutions on how to achieve those composition effects with languages that lack the appropriate features. Examining how such patterns can be implemented using AOPLs is a good way to evaluate them. Collections of functionally equivalent pattern implementations form a good basis for comparing them.

The potential of design patterns for illustrating and/or assessing the relative advantages of an AOPL was explored in the past [26–28,11,29], although few previous studies involve the full GoF collection of 23 patterns. To our knowledge, there are two systematic studies of AOPLs involving all GoF patterns: those by Hannemann and Kiczales (HK) on AspectJ [30] and those by Rajan on Eos [11]. Only the code examples from the HK study are publicly available. These were subsequently used as a basis for independent research [31,7,8,32]. Previous systematic studies based on GoF patterns use examples created by people with a stake on the language under study [30,11], 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 existing collections of Java examples freely available on the Web.[a] Both collections were posted long before the setup of the present study, by people that are independent of the present authors. The newly developed examples in OT/J are functionally equivalent to the corresponding original Java examples. Three examples from our collections were implemented in two different ways because OT/J provides several options for some patterns, some of which are more suitable than others for some specific comparisons.

The rest of this paper is structured as follows. Section 2 presents the study setup. Section 3 provides an overview of OT/J, with an emphasis on those language features and techniques that have an impact on the implementations. To illustrate the various features, an OT/J implementation of *Observer* is discussed. Section 4 establishes some parallels between OT/J and Java and AspectJ by proposing a set of guidelines on how to convert Java and AspectJ programs into OT/J. Section 5 discusses the pattern implementations, with a focus on the contributions that OT/J brings relative to Java and AspectJ. A few OT/J features and techniques of narrower applicability are also mentioned when discussing specific patterns. Section 6 presents a systematic comparison between the OT/J and AspectJ implementations, using as a basis the set of modularity properties from the HK study. Section 7 surveys related work, Section 8 outlines opportunities for future work and Section 9 concludes.

2. STUDY SETUP

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

- The collection by HK[b] was selected owing to the availability of functionally equivalent implementations of all GoF patterns in both Java and AspectJ [30]. The Java versions formed the basis for the OT/J implementations. The AspectJ versions are used in the comparisons discussed in Sections 5 and 6.
- The collection of Java examples by Cooper[c] [33]. Using a second complete collection was important for our study. Cooper's collection was selected on account of the challenges it poses. Almost all

[a]Full sources are available for download as an Eclipse/OTDT project at https://projectos.fct.unl.pt/attachments/download/590/OTJGoF4SPE.zip

[b]The original eclipse project with the AJDT plug-in for AspectJ is available at http://hannemann.pbworks.com/f/gof1.11.zip

[c]The refactored version used as a basis for this work is available as an eclipse project at https://projectos.fct.unl.pt/attachments/download/591/JavaGoFJamesCooper.zip

implementations are based on GUI classes from the standard Java *swing* library. The licence under which classes from Java standard APIs are available forbids modifying byte codes, which precludes some kinds of composition carried out by typical implementations of AOPLs. These hurdles make for slightly more realistic and interesting examples.

2.1. Terminology

Each design pattern prescribes *roles* for the concrete objects and/or modules (e.g. classes and Java interfaces) that participate in the pattern. Unfortunately, the term *role* is important in the context of OT/J as well [14]. To avoid confusion, this paper uses *role* to refer to the role classes of OT/J and *participant* or *pattern role* to refer to a role in a pattern [21].

Throughout the paper, 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 of employees. The HK study uses a very different scenario about traversing a tree structure. Each implementation of a given pattern requires a suitable scenario. We use the term *example* to refer to a specific implementation in a given language of a scenario for a pattern. Each scenario gives rise to at least one example for each different language. The collection from the HK study comprises 23 scenarios – one per pattern – and 46 examples – one each in Java and AspectJ. The collection by Cooper comprises 23 scenarios and examples – one for each pattern. We developed at least one OT/J example for each example, giving rise to 46 examples corresponding to one example for each of the 23 GoF patterns and for each of the collections. A scenario can give rise to more than one example even with a single language when it is possible to implement it in different ways. Alternative examples were developed for the scenarios of *Composite*, *Prototype* and *Visitor*. In each pattern, one of the examples mimics the AspectJ approach and is used for the comparative analysis presented in Section 6.

2.2. Preparation of the material for the study

The Java collection by Cooper was subject to a number of refactorings [34] prior to the study. This was important because good OOP decomposition is an important prerequisite for applying AOP effectively [35]. As the Cooper collection originally stood, many Main classes used as drivers for the example doubled as GUI class, for example, extending JFrame from the standard *swing* API and implementing *swing* interfaces such as ActionListener. The refactorings were mostly to ensure each class has a single set of responsibilities and comprised the extraction of a number of simple classes (e.g. for button actions), including a class representing the example's GUI. The observable behaviour of the examples was not changed.

Our approach 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 used in examples from both scenarios were classified as reusable. It was also established that modules with only abstract declarations would not count as a reusable module. This rule is important because when OT/J is used, it is easy to create abstract modules that can be used in multiple examples but add little functionality.

2.3. Classification of pattern roles as superimposed and defining

For the analysis of the various patterns and respective implementations, it is helpful to distinguish between the following two kinds of pattern roles:

- *Superimposed roles* are assigned to class modules that have functionality and responsibilities outside the pattern. In such cases, the participant class has a *core* set of responsibilities, and ideally, no other code should be found in it. The pattern role the class plays in the pattern is an *additional*, or *secondary*, role that should be moved to a separate module. Superimposed roles are indicative of the presence of crosscutting concerns, as the participation in a given pattern often cuts across multiple classes [30].
- *Defining roles* are completely defined by the pattern, with no functionality outside that context. Removing a pattern implementation from a system entails removing that class as well.

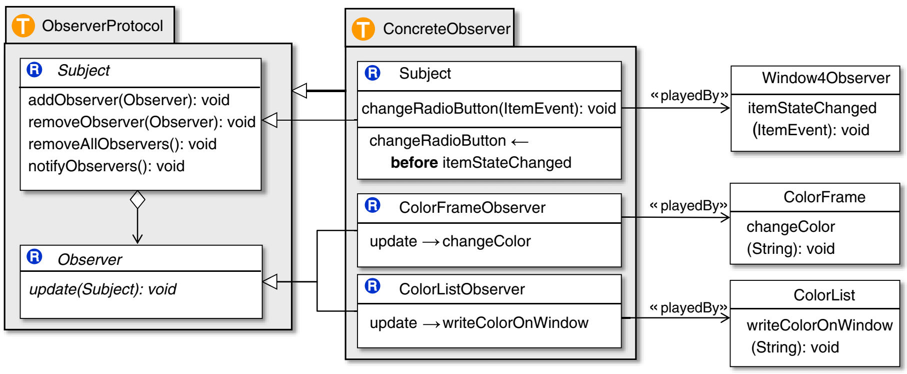

ObserverProtocol
Subject
addObserver(Observer): void
removeObserver(Observer): void
removeAllObservers(): void
notifyObservers(): void
Observer
update(Subject): void
ConcreteObserver
Subject
changeRadioButton(ItemEvent): void
changeRadioButton ← before itemStateChanged
ColorFrameObserver
update → changeColor
ColorListObserver
update → writeColorOnWindow
«playedBy»
«playedBy»
«playedBy»
Window4Observer
itemStateChanged (ItemEvent): void
ColorFrame
changeColor (String): void
ColorList
writeColorOnWindow (String): void

Figure 1. Example of a concrete scenario for *Observer*.

It is important to note that it is not always clear whether a pattern role should be classified as defining or superimposed. In general, a specific judgement influences the resulting design – see also Sections 4, 5 and 6.

3. THE OBJECT TEAMS PROGRAMMING MODEL

This section describes the OT/J language in some detail. Some parallels between OT/J and AspectJ are provided to clarify the various points. We assume the reader is familiar with AspectJ, as it is a well-known language, having been documented in many papers, tutorials and books (e.g. [19,20]). This section uses an example for *Observer* to illustrate the various features of OT/J. To distinguish between the modules supported by AspectJ and OT/J, the remainder of this paper uses the term 'aspect' to refer to the former and the term 'team' to the latter.

3.1. The language

The OT/J language was branded aspect-oriented from the start [15,36], but unlike AspectJ-like languages, it does not rely on an explicit notion of joinpoint and does not provide constructs for specifying or capturing joinpoints. Instead, OT/J provides language support to *role-based programming* [37,16]. Its approach can be characterized by multiple dimensions of polymorphism and support for composition at the instance level. In practice, this approach is equivalent to a joinpoint model comprising just method calls and field accesses.

Object Teams/Java adds to Java a new kind of module, the *team*, which roughly corresponds to AspectJ aspects. The most prominent feature of teams is that they enclose *roles*, that is, inner classes that represent the internal concepts of a collaboration of objects. Roles are *virtual classes* [38], that is, classes that are members of *instances* and subject to overriding and dynamic dispatch in the same way as methods. Teams provide the context within which a collaboration of roles takes place, which can include state and behaviour pertaining to the context of the collaboration. OT/J supports *family polymorphism* [39], that is, the ability to group a set of virtual classes into a larger package-like class – the team – such that consistency between all member classes and their instances can be enforced by the type checker. Team instances are used as type anchors to ensure that role instances from different team instances are not mixed.

Figure 1 outlines an example for *Observer*. Listings 1 and 2 show the code for the two teams from Figure 1. The notation used is UFA (UML for Aspects), an extension of UML proposed by Herrmann [40] to express the new kinds of module and module compositions that OT/J adds to Java. A key addition is the use of a package-like diagram representing a team – identified with 'T' at the top label. It aims to express a module capable of enclosing a set of collaborating classes. Figure 1 includes two teams: ObserverProtocol and ConcreteObserver. Any class within a team is a role, which is identified with an 'R' inside the name area for the class. In Figure 1, team ObserverProtocol encloses two roles – Subject and Observer. In Listing 1, these correspond to lines 04–11 and 12–16, respectively. The other key element is the role playing relationship, which is modelled as a dependence relationship with the <<playedBy>> stereotype.

```
  //The roles do not really need to be abstract. They are declared so to signal that
  //specific applications using the pattern should provide a refinement of these roles.
  public abstract team class ObserverProtocol {
    protected abstract class Subject {
      /** Registry of known Observers: */
      private LinkedList<Observer> observers = new LinkedList<Observer>();
      public void addObserver (Observer o) { observers.add(o); }
      public void removeObserver (Observer o) { observers.remove(o); }
      public void removeAllObservers() { observers.removeAll(observers); }
      public void changeOp() { for(Observer observer: observers) observer.update(this);
    }
    protected abstract class Observer {
      protected abstract void update(Subject s);
      public void start (Subject s) { s.addObserver(this); }
      public void stop (Subject s) { s.removeObserver(this); }
    }
  }
```

Listing 1. Example of a reusable team for the *Observer* pattern.

```
public team class ConcreteObserver extends ObserverProtocol {

  protected class Subject playedBy Window4Observer {
    private String selectedColor = "none";

    public String getSelectedColor() { return selectedColor; }

    public void changeRadioButton(ItemEvent e)
    //Guard predicate: if enabled, blocks callin when subject is in "Red" state)
    //when (((JRadioButton)e.getSource()).getText() != "Red")
    {
      if (e.getStateChange() == ItemEvent.SELECTED){
        selectedColor = ((JRadioButton)e.getSource()).getText();
        System.out.println(selectedColor);
        this.notifyObservers();
      }
    }
    changeRadioButton <- before itemStateChanged;
  }

  //Obtain String from the chain_of_responsibility chain of subject
  private String getColorFromSubject(Subject s){ return s.getSelectedColor(); }

  protected class ColorFrameObserver extends Observer playedBy ColorFrame {
    void update(Subject s) -> void changeColor(String selectedColor) with {
      getColorFromSubject(s) -> selectedColor
    }
    private abstract Color getColorFromColorFrame();
    getColorFromColorFrame -> get _color;

    private abstract void setColorFromColorFrame(Color c);
    setColorFromColorFrame -> set _color;
  }

  protected class ColorListObserver extends Observer playedBy ColorList {
    void update(Subject s) -> void writeColorOnWindow(String s) with {
      getColorFromSubject(s)-> s
    }
  }

  //Declared lifting: uses AnyBase to represent any base class playing Observer
  public <AnyBase base Observer>
  void addObserver(AnyBase as Observer obs, Window4Observer as Subject sub){
    sub.addObserver(obs);
  }

  public <AnyBase base Observer>
  void remObserver(AnyBase as Observer obs, Window4Observer as Subject sub){
    sub.removeObserver(obs);
  }
}
```

Listing 2. A concrete sub-team for a specific example of *Observer*.

In Figure 1, each role within the ConcreteObserver team has a role playing relationship with a traditional class. Classes involved in such relationships are base classes, but no special notation is used to distinguish them from non-base classes.

All potentially reusable logic pertaining to the collaboration between subjects and observers is enclosed in the team. Note that both roles have protected visibility. For roles, this means that only the team and sub-teams can see them. This is independently of the package where they are deployed: even other modules in the same package cannot see a protected role. Composing the team ObserverProtocol to a specific application is carried out by creating case-specific sub-teams. The present example includes classes Window4Observer as subject and classes ColorFrame and ColorList as observers (shown only in Figure 1). Composing ObserverProtocol to this example entails adding a sub-team such as that from Listing 2 (see also Figure 1).The sub-team encloses the case-specific parts, comprising bindings between the roles and example-specific classes. This approach is similar to that used with aspects [30], although AspectJ has limitations as concrete aspects cannot be extended. In OT/J, all possibilities of traditional class inheritance apply to teams *and* roles.

A role can extend another role from the same team or from some super-team along the team inheritance chain. For example, roles ColorFrameObserver (Listing 2, lines 24–33) and ColorListObserver (Listing 2, 35–39) inherit from ObserverProtocol.Observer using an explicit extends clause (Listing 2, lines 24 and 35). A role also inherits from roles of the same name in super-teams without using extends, a form of inheritance known as *implicit inheritance*. For example, ConcreteObserver.Subject (Listing 2, lines 3–19) inherits from ObserverProtocol.Subject (Listing 1, lines 4–11). Member overriding and late binding is supported in all cases.

A role can declare a *role playing relation* to a given *base class* – usually a plain Java class but can also be a team. This relation expresses a conceptual connection between the internal participant of the object collaboration represented within the team and some external module from a specific application. Roles Subject, ColorFrameObserver and ColorListObserver declare such relations – through the ‘playedBy’ keyword – to classes Window4Observer, ColorFrame and ColorList, respectively (Listing 2, lines 3, 24 and 35).

Each role can specialize only one base, but multiple roles can specialize the same base simultaneously. Role playing relations are inherited by sub-roles, independently of the variety of inheritance used, and can be refined, subject to typing rules. For technical reasons, proprietary classes such as those from the Java standard APIs cannot be bases. A role that declares (or inherits) a playedBy relation with some class is a *bound role*. A role can also be *unbound*, which is often the case of roles within abstract teams such as ObserverProtocol (Listing 1). Unbound roles are also used to represent concepts within the collaborations that have no counterpart outside the team.

Role playing resembles inheritance to some extent but differs in two important ways. First, role playing works at the instance level rather than at the class level. Roles can be added and removed to specialize 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 – essential to ensure the full substitutability that is prerequisite for type–subtype relationships. However, role playing relations do not require full substitutability. A role acquires base members *selectively*, and each acquisition is made explicitly. Role instances and base instances are substitutable in relation to code that uses those acquired members.

Although it has no effect on its own, a playedBy declaration provides the context for two kinds of bindings between role members and base members: *callouts* and *callins*. The ‘in’ and ‘out’ are to be seen from the perspective of the role. A callout binding (henceforth just ‘callout’) enables a role to acquire an implementation available in the base instance for one of its abstract declarations – the role object is said to ‘call out’ to the base object. Listing 2 shows two examples of callouts of fields (lines 29 and 32). The modifiers get and set serve to distinguish between field accesses for reading and for writing – similarly to the get and set pointcut designators of AspectJ. Thus, role methods getColorFromColorFrame and setColorFromColorFrame are bound to accesses to the field _color of base class ColorFrame to become a getter and a setter, respectively.

The notation for callouts to methods is often very simple. When the parameter lists and return types are identical, callouts can be specified without them:

roleMethodName → baseMethodName;

Object Teams/Java supports an optional ‘with’ {...} sub-clause within roles to specify mappings between role members and base members when names and/or signatures are different. They also can specify certain adaptations, for example, order of parameters, type conversions or some short glue-code expression. Listing 2 also shows two such cases (lines 25–27 and 36–38).

A concrete role must define all of its members but can leave some of them as abstract and acquire the corresponding definitions from the base by means of callouts. Role ColorFrameObserver (Listing 2, lines 24–33) provides an example: it acquires definitions for two of its methods (Listing 2, lines 29 and 32). By default, a role acquires nothing from its base: all acquisitions must be explicit.[d]

Callouts are a means of communication from role to base. *Callin bindings* (henceforth just 'callins') support communication in the opposite direction. Callins are also the means through which a role *overrides* base behaviour, by specifying that the methods of its role be implicitly called whenever certain events from the associated base object take place. Callins are the rough equivalent to AspectJ advice, although they work more narrowly, along the channel of communication between role and base. The notation for callins is similar to that for callouts, with the arrow pointing from base to role. See Listing 2 (line 18) for an example.

The execution events supported by callins are the execution of methods and access to fields. Whenever the target 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 implicitly created at that point. This is the default way to create role instances. Callins can distinguish between field accesses for reading and for writing, like in AspectJ pointcuts. Callins can execute before, after or instead of the captured event, similarly to AspectJ advice. To that effect, role methods bound through callins must have one of three modifiers 'before', 'after' and 'replace', respectively. Listing 2 line 18 shows an example of a before callin.

The callin modifier is required for each role method that overrides a base method, that is, is bound with the 'replace' modifier (not shown in Listing 2). *Callin methods* are a special kind of role method that can invoke the base method using a *base call*, using the 'base' keyword, in a way that resembles proceed calls within around advice in AspectJ. Note that the method name used in the base call is that of *role method* so that the namespace of the role is kept self-contained, which is essential to ensure a complete independence of teams and roles from specific bases:

callin *type* roleMethod(*parameter*) {
    //...
    ...=**base**.roleMethod();
    //...
}
type roleMethod() <– **replace** *type* baseMethod(*parameter*) ;

Listing 3 shows the driver for the *Observer* example (the rest of the class is not relevant). It illustrates that unlike aspects, teams can be freely instantiated using 'new'. For callins to take effect, the enclosing team instance must be *activated*, for which methods activate and deactivate are available to all teams. Method activate can be called without arguments, for events originating from the current thread only, or with the predefined constant org.objectteams.Team.ALL_THREADS (Listing 3, line 3) to make the team respond to events from any thread. This contrasts with aspects, whose advice cannot be activated or deactivated dynamically. This limitation tends to make the AspectJ implementation of some patterns more complex than they would otherwise be [41,11].

```
01    public static void main(String[] args) {
02      ConcreteObserverTeam observationTeam = new ConcreteObserverTeam();
03      observationTeam.activate(Team.ALL_THREADS);
04
05      // subject
06      Window4Observer mainWindow = new Window4Observer();
07
08      // observers
09      ColorFrame colorFrame1 = new ColorFrame();
10      ColorFrame colorFrame2 = new ColorFrame();
11      ColorFrame colorFrame3 = new ColorFrame();
12      ColorList colorList = new ColorList();
13
14      observationTeam.addObserver(colorFrame1, mainWindow);
15      observationTeam.addObserver(colorFrame2, mainWindow);
16      observationTeam.addObserver(colorFrame3, mainWindow);
17      observationTeam.addObserver(colorList, mainWindow);
18    }
```

Listing 3. Driver for the *Observer* example.

[d]The OT/J compiler can be configured to support *inferred callouts* so that it does not issue errors when finding undeclared references to base members. However, in general, that is not the recommended style.

Object Teams/Java also provides additional control for callin activation in the form of *guard predicates*, which comprise declarative clauses that restrict the effect of callins to specific situations. OT/J supports several levels of control: callin binding, role method, role class and team. Line 10 of Listing 2 shows a role method-level guard predicate (within a line comment).

Instances of role classes and base classes comprise separate object identities, but in some situations, their instances can be interchanged in ways that are transparent to the type checker – this is called *translation polymorphism* [36]. When the flow of control crosses the base–team boundary in the base–role direction (e.g. by way of callin), the role instance automatically replaces the base object. This translation 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*. It is possible to specify a lifting explicitly in team-level methods using a signature that specifies *both* the base and the role types. This is called *declared lifting* and opens the way for team-level methods that map base objects to their role instances while avoiding exposing roles to details from outside the team boundaries. Listing 2 (lines 41–45 and 47–50) shows two team methods using declared lifting. The base type can be a concrete, specific type, but in those examples, AnyBase is a type parameter whose instantiations must all be bases liftable to role Observer.

Finally, OT/J automatically synthesizes a special *lifting constructor* for each bound role, which accepts a base instance. It is used in cases that require an explicit instantiation of a bound role – although this is supposed to occur infrequently.

3.2. *Object Teams idioms*

As in most modern programming languages, a number of design problems surfaced that are not solved by a direct, straightforward use of OT/J's mechanisms but can be addressed through *idioms*, that is, pattern-like techniques that are specific to the features of a particular language. Three idioms are used in some of the pattern implementations – *Transparent Role*, *Object Registration* and *Double Dispatch*. The latter is discussed in the context of the *Visitor* pattern (Section 5.4).

The *Transparent Role* is the idiom most often used in our collections of examples. It addresses the problem of exposing role functionality to the outside of the team instance without exposing details of the role, which should be considered part of the team's private implementation. An option would be to use team-level methods to forward calls from outside the team to the role instance within the team. However, that can be verbose as it entails creating a different team-level method for each role method (possibly using declared lifting). The key idea of *Transparent Role* is to make both role and base implement a common top-level Java interface and make clients access the role only through that interface. This way, the role need never be exposed outside of the team.

The *Object Registration* idiom is about restricting the set of base objects that have role counterparts. When no explicit restrictions are programmed, a role instance is created for *every* base instance that crosses the team boundary in a program's control flow – usually when a callin is activated. *Object Registration* uses guard predicates – at the role level: there are other levels – to restrict activation of the callin to a set of registered base instances. A team method is used to explicitly register all intended base instances. Unregistered base instances do not trigger the callins.

Listing 4 illustrates the use of *Object Registration*, using the OT/J example for the Cooper scenario for *Decorator*. A team encloses two decorator roles. Team-level methods (Listing 4, lines 14–17 and 18–21) create the roles explicitly, using their lifting constructors (Listing 4, lines 15 and 19). To ensure callins are triggered for just those registered objects, the roles declare the appropriate guard predicates (Listing 4, lines 5 and 10). The example from Listing 4 also illustrates the use of a clause to control precedence between callins bound to the same target event (Listing 4, line 2). These are similar to aspect precedence in AspectJ, which provides a measure of control over the order of advice acting on the same joinpoint.

```
public team class ButtonDecoratorTeam {
  precedence CoolDecorator, SlashDecorator;

  public class CoolDecorator playedBy MyButton
  base when (ButtonDecoratorTeam.this.hasRole(base, CoolDecorator.class)) {
    //...
  }

  public class SlashDecorator playedBy MyButton
  base when (ButtonDecoratorTeam.this.hasRole(base, SlashDecorator.class)) {
    //...
  }

  public MyButton addSlashDecorator(MyButton c){
    new SlashDecorator(c);
    return c;
  }
  public MyButton addCoolDecorator(MyButton c){
    new CoolDecorator(c);
    return c;
  }
}
```

Listing 4. Illustration of the implementation of *Decorator* for the example by Cooper.

4. CONVERTING JAVA AND ASPECTJ DESIGNS TO OBJECT TEAMS

This section sums up the topics about OT/J covered in the previous section, by proposing a set of guidelines on how to convert Java and AspectJ programs to OT/J. The guidelines are not intended to be precise or applicable in every situation – that would entail developing a catalogue of refactorings along the lines of [31], which is out of the scope of this paper. They should be approached as 'rules of thumb' about how the features of OT/J should be used to realize patterns in common scenarios.

4.1. Guidelines to convert Java to Object Teams

To convert plain Java programs to OT/J, we propose the following transformations. The guidelines are numbered so that some of them can be referred in subsequent sections. The numbering is not intended to imply a definite order for the transformations.

1 In general, inter-related classes from a collaboration of classes can be turned into roles within a team.
2 If the new roles are not used outside the team (a likely situation in e.g. *Interpreter* and *State*), they should become unbound roles with protected visibility. This situation corresponds to defining pattern roles. Often, the team serves as a facade for the roles.
3 If the original collaborating classes are used outside the team and each class has a core of responsibilities beyond the collaboration, keep the class standalone, as bases of the new (bound) roles. This situation corresponds to superimposed pattern roles and is likely to occur in instances of, for example, *Observer* and *Chain of Responsibility*.
4 When extracting a bound role, move pattern functionality from the class to the role as role methods. In the role, add callins to bind the role methods to the appropriate points in the base.
5 If the purpose of a class is to hold common context shared by a set of collaborating classes (as in many GoF patterns), move its state and behaviour to the team as team-level state and behaviour.

4.2. Converting AspectJ to Object Teams

Hannemann and Kiczales claimed that benefits from AspectJ are mainly felt in pattern implementations by *inverting dependence*, that is, making pattern code dependent of participants rather than the opposite and keeping code related to management of dependence within pattern aspect modules [30], thus making case-specific classes *oblivious* [42] from pattern roles. The OT/J implementations also follow this reasoning.

The AspectJ implementations from the HK study comprise a small set of techniques, which can be summarized as follows: (1) plain use of pointcuts and advice (e.g. *Decorator*); (2) use of inter-type declarations (a.k.a. *introductions* or the *open class mechanism*) to compose state and behaviour to participants; and (3) a more elaborate technique, based on *marker interfaces*, used in all the reusable aspects except *Command*. The technique uses empty marker interfaces to represent pattern roles, often enclosed within an abstract aspect. The aspect composes functionality to the interfaces, either through inter-type declarations or through aspect methods whose parameter types are the interfaces. Concrete sub-aspects use *declare parents* clauses to bind the marker interfaces to concrete, case-specific classes. Some of the reusable aspects also use pointcuts to capture the events of interest and establish the relevant connections among participants. The pieces of advice acting on those events also refer to participants through the marker interfaces. This technique is used in a number of patterns to attach a default implementation to a plain Java interface. It has the advantage to make class participants free to inherit from some other class [30].

As said in Section 3, advice is the equivalent to OT/J callins. Context capture performed by pointcuts and advice are roughly equivalent to callouts. The marker interfaces are the equivalent to OT/J roles, and *declare parents* clauses are equivalent to the *playedBy* binding between roles and base classes. Reflecting these parallels, we propose the following transformations to convert AspectJ programs to OT/J.

- Replace the aspect with a team.
- Turn inner, marker interfaces within an aspect into roles within a team.
- Turn inter-type declarations adding members to marker interfaces into role state and behaviour.
- Turn *declare parents* clauses binding concrete classes to marker interfaces into role playing (playedBy) between the roles and the concrete classes.
- Turn helper classes not used outside the aspect into protected unbound roles within the team.
- Turn before/after advice into role methods bound with before/after callins.
- Turn around advice into callin role methods bound with replace callins.
- Turn calls to *proceed* within around advice into base calls within callin methods.
- Replace context capture as performed by parameterized pointcuts with callouts and role method parameters.

We believe the above guidelines form a basis for transforming AspectJ into OT/, but they will not always yield an optimal OT/J design. We noticed that a number of patterns are better tackled using other approaches that have no counterparts in AspectJ. This is the reason why three of the patterns are implemented in two different ways using OT/J (see Section 6).

A major factor accounting for differences between AspectJ and OT/J is AspectJ's lack of a cohesion mechanism that associates the various aspect members together. That is why marker interfaces are often empty, and the members to which they are associated are often defined at the same level within the aspect, giving rise to a flat internal structure. Mezini and Ostermann [41] criticized AspectJ's design, pointing out that it gives rise to a rather procedural style of programming that seems contradictory to the fundamentals of OOP, according to which a type definition contains all methods that belong to its interface. Mezini and Ostermann also pointed out that it seems contradictory to the aspect-oriented vision of defining crosscutting modules in terms of their own modular structure. The flat structure makes aspects harder to reuse. In their study, HK conclude that benefits brought by the mechanisms of AspectJ are primarily felt when dealing with superimposed roles, whose code is straightforward to extract to a separate module. Defining roles, however, pose difficulties for AspectJ to improve over Java because there are no multiple roles to separate. Unlike AspectJ, OT/J has the option of using unbound roles to represent defining pattern roles and enclosing the whole object collaboration within a team.

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 supports guard predicates and the ability to (de)activate callins explicitly.

5. DESIGN PATTERN IMPLEMENTATIONS IN OBJECT TEAMS

This section describes the OT/J implementations. A number of recurring approaches based on the features of OT/J can be discerned, which are used to group the various patterns.

5.1. Polymorphic role constructors to support Factory Method

One important difference between Java and OT/J is that role constructors are polymorphic. Because the purpose of *Factory Method* is precisely to emulate polymorphic constructors, OT/J can be expected to provide direct language support for factory methods and indeed does so. The approach is to turn a hierarchy of *product* classes into a hierarchy of roles within a team (guideline 1). Calls to the factory methods are replaced by calls to role constructors. This approach can work independently of whether the roles are bound or unbound. It also works with proprietary classes whose source code is not available, as long as the classes are not final. Proprietary classes can be integrated by means of roles that inherit from them and which are used in their place. The OT/J example for the HK scenario of *Abstract Factory* is an example, involving roles that inherit from classes from the Java *swing* standard API (scenarios for *Abstract Factory* usually include a collection of factory methods).

Figures 2 and 3 illustrate *Factory Method* using the scenario by Cooper [33], in which a name is obtained from a GUI text field. The name comprises at least two words, which are extracted differently depending whether the field has a comma. If it has, the format "<last>, <first>" is assumed. If it does not have, it assumes that the first word in the field is the first name. The Java example (Figure 2) uses class Namer (participant *product*) to extract the first and last names. The two different ways to present the name are implemented by two subclasses of Namer – FirstFirst and LastFirst – which are *concrete products*. The *Factory Method* pattern also defines the roles *creator* and *concrete creator*, which are, respectively, abstract and concrete representations of the object providing the factory method. Both correspond to class NamerFactory, whose factory method instantiates the suitable subclass of Namer depending on the format of the text field.

Figure 3 shows the OT/J implementation. With OT/J, *products* usually are roles and the creator a team, which defines the factory method as a top-level team method that returns an instance of required concrete product. To prevent dependence to the roles outside the team, idiom *Transparent Role* is used: role instances are returned as instances of Java interface INamer. An alternative would be to keep the role instance hidden within the team, which is the option taken for the HK scenario. Although the OT/J design as shown in Figure 3 may seem more complex than that of Figure 2, the number of top-level modules is lower. Team NamerFactoryTeam replaces four classes from the Java version – Namer, FirstFirst, LastFirst and NamerFactory – and adds just interface INamer for use in the *Transparent Role* idiom. This illustrates how complexity tends to be better managed with OT/J.

Although some examples from this study include cases of polymorphic constructors, the approach does not work in all cases. Turning classes into roles may not be feasible in all cases. The factory method may receive arguments specifying the actual concrete class to instantiate, which cause hurdles. The selection logic associated to the arguments must still be placed somewhere, which ends up being a

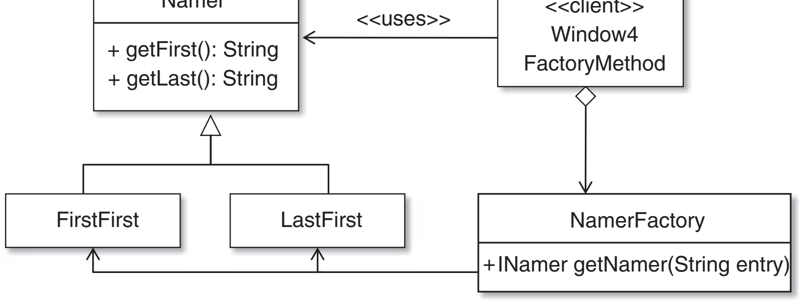
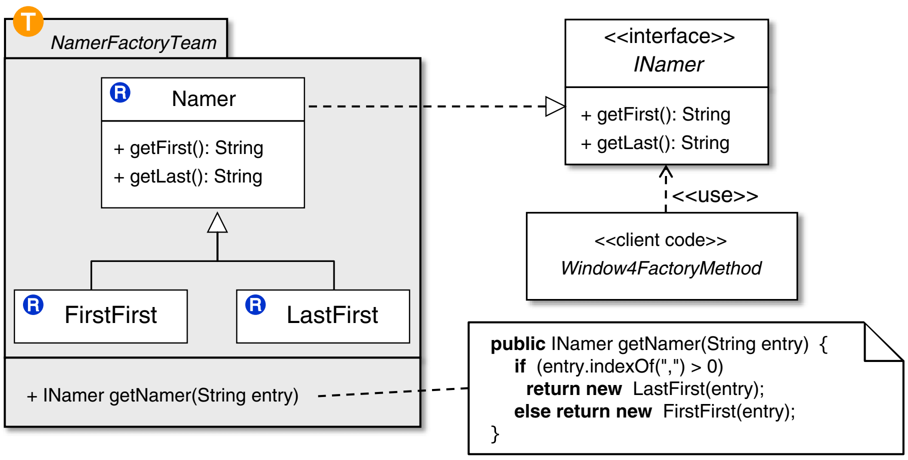


Figure 2. *Factory Method* in Java (Cooper scenario).



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

Figure 3. *Factory Method* in Object Teams (Cooper scenario).

kind of factory method. An OT/J implementation will not eliminate factory methods if they receive parameters, as is the case with the examples for *Factory Method* from the present study. Nevertheless, our collection includes a number of cases using factory methods that were made simpler by replacing them with polymorphic constructors (e.g. the examples of *Abstract Factory*). The AspectJ approach does not eliminate factory methods. It localizes them into aspects.

5.2. *Packaging multiple entities more cohesively behind a team boundary*

In several patterns, the various participants share a common, or *global*, context. Such patterns define a participant – often called *context* – for the express purpose of enclosing it. Thus, the course of action for all cases from this section entails applying guideline 5 to move common context to a team. Sometimes, it is possible to completely hide the participants from the team's public interface. This thinking is absent in the GoF patterns, because there is no construct unifying the concepts of class and package. AspectJ also lacks such a cohesion mechanism, and therefore, the approach taken in the HK study is very different. *Abstract Factory* and *Interpreter* are next used to illustrate.

5.2.1. Abstract factory. The purpose of the pattern 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 [21]. That is exactly the purpose of family polymorphism [39]. Team instances make natural factories for the role types they enclose and provide guarantees of consistency among object families. Indeed, OT/J provides direct support for *Abstract Factory* with respect to roles. The OT/J way to implement *Abstract Factory* is to turn product classes into roles whenever this is feasible (guideline 1 followed by guideline 2 or 3). To illustrate, Listing 5 shows the OT/J implementation of the HK scenario. Because of space constraints, the code for the GUI driver of the example is not shown.

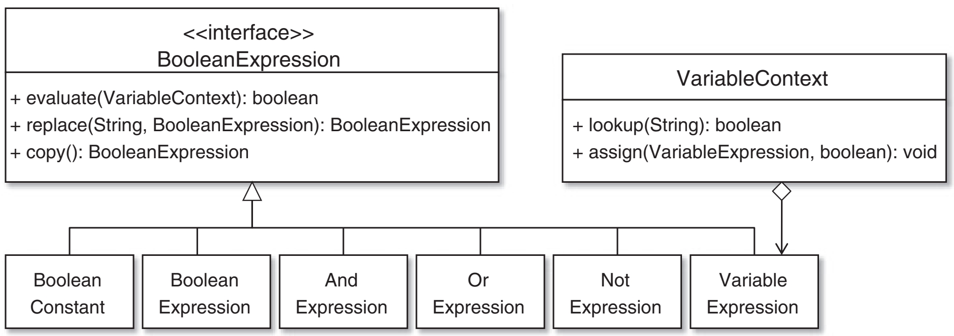
In this example, abstract team ComponentFactoryTeam (Listing 5, lines 1–13) defines default implementations for two concrete product classes as roles – Rlabel (Listing 5, lines 2–4) and RButton (Listing 5, lines 5–7) – as well as methods that instantiate and return them (methods createLabel and createButton in Listing 5, lines 8–10 and 11). The products are GUI objects from the Java standard API swing. They cannot be directly handled as roles, but roles can inherit from them (see Listing 5, lines 2 and 5). Sub-teams redefine the roles Rlabel and RButton – one is shown in Listing 5, lines 14–32. When methods createLabel and createButton are called on instances of those sub-teams, the code from those methods inherited from ComponentFactoryTeam is dispatched to the redefined roles.

The approach taken in the HK study is to use AspectJ's capability to emulate a limited form of multiple (mixin) inheritance to add default implementations to a Java interface representing the abstract factory.

5.2.2. Interpreter. The pattern defines a representation for the grammar of a simple language – often a syntax tree – along with an interpreter that uses the representation to interpret sentences in the language. The pattern declares an *abstract expression* participant that represents the various distinct nodes comprising the syntax trees, which is in turn divided into sub-types *terminal* and *non-terminal*. Figures 4 and 5 show the Java and OT/J approaches for the HK scenario for *Interpreter*, respectively. *Interpreter* declares a *context* participant to manage global information within the interpreter, which is represented by class VariableContext in the Java version (Figure 4) and which becomes InterpreterTeam team in the OT/J version – following guideline 5. The team also encapsulates a hierarchy of unbound roles representing the nodes – following guideline 2. The implementation enables a choice between enclosing all nodes within a single team or within a small hierarchy of teams. Here, one team encapsulates for terminals and a sub-team encapsulates non-terminals. Note that different partitions could have been made, as team hierarchies can evolve very flexibly [43]. A single interpreter team could have encapsulated all classes from Figure 4. The HK example uses an aspect with just a small set of inter-type declarations that add additional state and behaviour to the *terminal* and *non-terminal* participants.

```
public abstract team class ComponentFactoryTeam {
  protected class RLabel extends JLabel {
    public RLabel(String label) { super(label); }
  }
  protected class RButton extends JButton {
    public RButton(String label) { super(label); }
  }
  public JLabel createLabel() {
    return new RLabel("This Label was created by " + getName());
  }
  public JButton createButton(String label) { return new RButton(label); }
  public abstract String getName();
}
public team class FramedFactoryTeam extends ComponentFactoryTeam {
  protected class RLabel {
    public RLabel(String label) {
      super(label);
      Border raisedbevel = BorderFactory.createRaisedBevelBorder();
      Border loweredbevel = BorderFactory.createLoweredBevelBorder();
      setBorder(BorderFactory.createCompoundBorder(raisedbevel, loweredbevel));
    }
  }
  protected class RButton {
    public RButton(String label) {
      super(label);
      Border raisedbevel = BorderFactory.createRaisedBevelBorder();
      Border loweredbevel = BorderFactory.createLoweredBevelBorder();
      setBorder(BorderFactory.createCompoundBorder(raisedbevel, loweredbevel));
    }
  }
  public String getName() { return "Framed Factory"; }
}
```

Listing 5. Implementation in Object Teams of the Hannemann and Kiczales scenario for *Abstract Factory*.



<<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 4. Java implementation of *Interpreter* (Hannemann and Kiczales scenario).

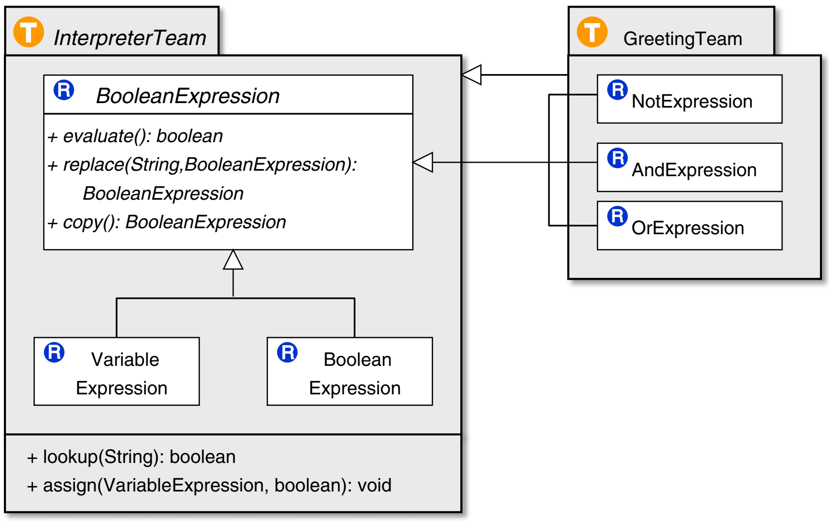

InterpreterTeam
BooleanExpression
+ evaluate(): boolean
+ replace(String,BooleanExpression): BooleanExpression
+ copy(): BooleanExpression
Variable Expression
Boolean Expression
+ lookup(String): boolean
+ assign(VariableExpression, boolean): void
GreetingTeam
NotExpression
AndExpression
OrExpression

Figure 5. Object Teams/Java implementation of *Interpreter* (Hannemann and Kiczales scenario).

The above OT/J implementation is also a case in which participants can be completely encapsulated within a team, as unbound roles, because participants correspond to defining pattern roles. The example of *Observer* shown in Section 3 illustrates the other case – superimposed roles – which are represented by bound roles.

We next provide an overview of the remaining patterns from this group.

5.2.3. Facade. The purpose of the pattern is to add a layer encapsulating a number of participants. This is another pattern whose approach using OT/J is straightforward. Indeed, teams often serve as a *Facade* for their roles. Unfortunately, the examples for *Facade* are not particularly suitable to illustrate this notion, although it can be found in many examples for other patterns. In the Cooper scenario, the *subsystem classes* comprise proprietary interfaces from the Java standard APIs. In the HK scenario, OT/J provides more flexibility in composing the *facade* to the *subsystem classes* by means of role playing. The AspectJ example from the HK study is identical to that in Java.

5.2.4. Flyweight. The pattern 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 by the *flyweight* participant, which declares a common interface through which the objects can receive and act on the shared state. The common context is supported by the *flyweight factory*, which is also responsible for creating the *flyweight*s and managing the way state is shared among them. In the OT/J implementation, a role represents the *flyweight* and the team is the *flyweight factory*. The AspectJ approach is to use an aspect acting as a *flyweight factory*. The *flyweight* pattern role is composed to concrete classes by means of a marker interface and inter-type declarations.

5.2.5. Iterator. The pattern defines an *aggregate* participant whose elements are to be accessed sequentially without exposing its underlying representation. The *iterator* defines an interface for accessing and traversing the elements of the *aggregate*. The *aggregate* provides the context in which the iterator operates. In the OT/J examples, it becomes a team that encloses the *iterator* as an unbound role. The iterator can be used outside the team without exposing a role by means of the Iterator interface. This is an instance of the *Transparent Role* idiom. The AspectJ example only differs from the Java version in that it places the *iterator* factory in an aspect instead of a plain Java class, which is composed to the *aggregate* participant by means of an inter-type declaration.

5.2.6. Mediator. The pattern defines an object that encapsulates how a set of objects interact and promotes loose coupling by preventing them from referring to each other explicitly. Participant *colleague* represents the collaborating objects. A *mediator* defines an interface for communicating with *colleague*s, knows and maintains the *colleague*s and implements the collaboration by coordinating them. The *mediator* manages the common context and becomes a team in the OT/J implementation. It encloses a number of roles bound to the colleague classes, which remain standalone but free of code related to the pattern (guidelines 1 and 3). The AspectJ approach treats *mediator* as a superimposed pattern role that is composed to a concrete class.

5.2.7. State. The pattern is about allowing an object to alter its behaviour when its internal state changes, giving the impression that it changes its class. *State* defines a *state* participant representing the internal states through which the object can go. 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. In the OT/J implementation, the team is the *context*, which encapsulates the various *state* participants as unbound roles (guidelines 1 and 2). The AspectJ approach to *State* is to use an aspect to enclose the logic for managing transitions between states. Related classes remain standalone with the consequence that the pattern implementation remains spread over multiple modules.

5.2.8. Builder. The pattern 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 *builder*, which declares an interface for the building process. The *director* participant constructs the *product* in stages, using the *builder* interface. In the OT/J examples, the *director* becomes a team that serves to provide the context for the *builder* and the building process. The AspectJ approach is to represent the *builder* as a Java interface to which an aspect attaches the default behaviour.

5.3. Composing pattern roles through role playing: Adapter, Bridge, Decorator, Prototype, Proxy and *Strategy*

In a number of patterns, the contribution of OT/J is primarily to provide role playing as an additional option to compose the pattern logic to the participant classes. In patterns *Adapter*, *Bridge*, *Decorator*, *Prototype*, *Proxy* and *Strategy*, role playing is used instead of aggregation and inheritance to compose secondary roles.

We illustrate the approach with the complete OT/J implementation of the HK scenario for *Adapter* – see Listing 6. In this simple example, some client code expects interface Writer (Listing 6, line 1), but the class used – SystemOutPrinter (Listing 6, lines 2–4) – does not conform to that interface. Team PrinterAdapter-Team (Listing 6, lines 5–11) defines a role Adapter (Listing 6, lines 6–8) bound to SystemOutPrinter and that implements Writer. In this example, a single callout (Listing 6, line 7) is sufficient for mapping *adapter* and *adaptee*. The team also provides a team-level method (Listing 6, line 10) that uses declared lifting to return the role instance corresponding to the base received as argument.

```
01  public interface Writer { public void write(String s); }
02  public class SystemOutPrinter {
03    public void printToSystemOut(String s) { System.out.println(s.toUpperCase()); }
04  }
05  public team class PrinterAdapterTeam {
06    public class Adapter implements Writer playedBy SystemOutPrinter {
07      write -> printToSystemOut;
08    }
09    //team-level method using declared lifting:
10    public Adapter getAdapted(SystemOutPrinter as Adapter adapter) { return adapter; }
11  }
12  public class Main {
13    public static void main(String[] args) {
14      Writer myTarget;
15      SystemOutPrinter adaptee;
16      System.out.println("Creating the Adaptee...");
17      adaptee = new SystemOutPrinter();
18
19      System.out.println("Creating the Adapter...");
20      final PrinterAdapterTeam pat = new PrinterAdapterTeam();
21      Adapter<@pat> adaptedClass = pat.getAdapted(adaptee);
22      myTarget = adaptedClass;
23
24      System.out.print  ("Adapter and Adaptee are the same object: ");
25      System.out.println(myTarget.equals(adaptee));
26
27      System.out.println("Issuing the request() to the Adapter...");
28      myTarget.write("Test successful.");
29    }
30  }
```

Listing 6. Implementation in Object Teams of the Hannemann and Kiczales scenario for *Adapter*.

Class Main (Listing 6, lines 12–30) is the driver for the example and also represents client code to an extent. The code for Main illustrates how a role can be used outside a team – note that the team instance must be declared final (Listing 6, line 20) to serve as a type anchor (e.g. Listing 6, line 21).

The AspectJ example uses an inter-type declaration to attach an adapting method to the *adaptee*.

We next provide an overview of the remaining patterns from this group.

5.3.1. Bridge. The *abstraction* hierarchy becomes a hierarchy of roles, bound to the hierarchy of *implementors* through role playing. The AspectJ example comprises an aspect that attaches a number of *implementor* members to a Java interface that represents the *abstraction*.

5.3.2. Decorator and Proxy. The decorators become roles bound through role playing to the Java classes to be decorated (guideline 1 and 3). The *Object Registration* idiom is used to ensure that only the desired objects are decorated. Contrary to traditional OOP implementations, the identity of the original *component* objects is preserved when they are decorated. The examples taken for *Proxy* are similar to those for *Decorator*, with *proxies* instead of *decorators*. The implementations of *Proxy* are simpler because they use neither guard predicates nor precedence control. The AspectJ example for *Decorator* is based on plain use of advice over *component* events. The AspectJ example for *Proxy* uses pointcuts and advice to control access to a *subject* participant. The AspectJ implementations for *Proxy* and *State* are based on advice, which operate statically rather than at the instance level and are therefore less flexible. In cases of *Proxy* in which the subject and proxy participants must be different classes, the AspectJ version would be identical to that in Java.

5.3.3. Prototype. Both Java examples use the java.lang.Cloneable Java interface as the *prototype*. The AspectJ example represents the *prototype* participant with a marker interface in an abstract aspect and uses a concrete sub-aspect to compose a clone operation by means of an inter-type declaration. Both OT/J examples for the scenario for *Prototype* use a bound role instead of a marker interface and *declare parents*. One example mimics the AspectJ approach and keeps the part relating to the clone operation in a reusable abstract team. The examples using this approach are the ones used for the comparison with AspectJ (Section 6). The other example is case-specific and uses the *Object Registration* idiom to narrow the cloning effect – implemented through callins – to just the intended objects.

5.3.4. Strategy. *Strategy* is about defining a family of algorithms for 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. The OT/J examples use role playing to compose the various strategies to the *context* objects that use them. The connection between the *strategy* and the *context* object can be made in different ways. In the HK example, it is made explicitly through a team-level method with declared lifting. In the Cooper's example, it is made implicitly by means of a callin. The AspectJ example uses a pointcut and advice to intercept the operation and provide the intended *strategy*, which is specified by means of an aspect method.

5.4. Implementing Visitor with the *Double Dispatch*

In traditional OOP, *Visitor* is a technique to add operations to instances of a pre-existing class hierarchy by means of separate *visitor* objects. Each class from the hierarchy (*concrete element*) declares an *accept* operation that receives visitors and uses their services through calls to a *visit* operation, which in turn receives the instance of the class and uses the suitable behaviour. Often, different classes (*concrete elements*) require different visit methods that are usually implemented through method overloading. The drawback of *Visitor* is that adding a new concrete element class to the hierarchy entails adding a new method for that class in every visitor class.

The *Double Dispatch* idiom is used in the examples of *Visitor*. The idiom is about emulating double dispatch by connecting two hierarchies – one of bases and another of roles – by means of a team method with declared lifting. Bases are the *concrete elements*, and roles are visitors. By building a suitable hierarchy of visitor roles within a team, dispatch to the intended visitor is performed without the need for explicit *accept* and *visit* operations. The root motivation for *Visitor* is to deal with the lack of double dispatch in OOP languages, so one would expect the implementation of the pattern to 'disappear' from the OT/J examples – and indeed it does.

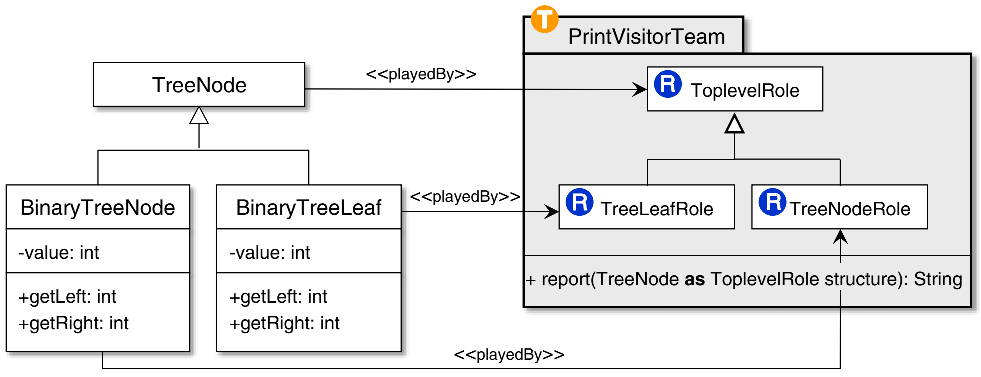


Figure 6. Object Teams/Java implementation of *Visitor* supporting double dispatch

To illustrate the use of *Double Dispatch* idiom, Figure 6 outlines the OT/J example for the HK scenario, which comprises an (empty) class TreeNode as *element* and subclasses BinaryTreeNode and BinaryTreeLeaf as *concrete elements*. The roles correspond to the visitor classes from the Java version and add operations to the instances of concrete elements. Both visitors accumulate values as the collection of concrete elements is traversed.

```
01  public team class PrintVisitorTeam {
02    protected class ToplevelRole playedBy TreeNode {
03      public String getRepresentation() {
04        return ""; //this method is not supposed to be called, ever
05      }
06    }
07    protected class TreeLeafRole extends ToplevelRole playedBy BinaryTreeLeaf {
08      public abstract String getRepresentation();
09      String getRepresentation() -> int getValue() with {
10        result <- Integer.toString(result)
11      };
12    }
13    protected class TreeNodeRole extends ToplevelRole playedBy BinaryTreeNode {
14      public abstract TreeNodeRole getLeft();
15      getLeft -> getLeft;
16      public abstract TreeNodeRole getRight();
17      getRight -> getRight;
18
19      public String getRepresentation() {
20       return "{"+getLeft().getRepresentation()+","+getRight().getRepresentation()+"}";
21      }
22    }
23
24    public String report(TreeNode as TreeNodeRole structure) {
25      return “>>> traversed the tree to:” + structure.getRepresentation();
26    }
27  }
```

Listing 7. Implementation in Object Teams of a concrete example of *Visitor*.

Listing 7 shows the OT/J code for the example from Figure 6. The team defines a role for each different class from the hierarchy (Listing 7, lines 7–12 and 13–22) plus a top-level team method *report* (Listing 7, lines 24–26) that receives a (*concrete element*) base object and translates it to its corresponding (*visitor*) role. The team-level method only needs the top-level types from the two hierarchies. When the example runs, it lifts the *concrete element* base instance to its corresponding *visitor* role instance. Late binding along the visitor role hierarchy dispatches to the appropriate visitor code.

The AspectJ solution to *Visitor* from the HK study is based on a reusable abstract aspect using marker interfaces to represent the *concrete element* and *visitor* participants and uses inter-type declarations to compose additional accept and visit methods to the interfaces. Concrete sub-aspects use *declare parents* clauses to bind the interfaces to example-specific classes, which thereby acquire the methods. No pointcuts or advice is used. One of the marker interfaces used by the abstract aspect must be top-level and implemented by the case-specific modules, meaning that the implementation is partly placed outside the aspects. A second OT/J example mimicking the AspectJ design was created using role playing with callouts to derive a similar binding outcome (it is used in the systematic comparison with AspectJ discussed in Section 6). Like the AspectJ example, it gives rise to a reusable module. It improves on the AspectJ implementation as regards locality because all parts relating to concrete participants are defined within the team and composed through role playing. Independent of the AOPL used, this approach to *Visitor* fits less naturally than the solution based on *Double Dispatch* because it does not avoid explicit *accept* and *visit* methods. It merely localizes them into an aspect.

5.5. Confined roles for Memento

Memento is discussed separately because its implementation in OT/J uses *confined roles*, a feature not mentioned elsewhere in this paper. The intent of *Memento* is to capture and externalize the internal state of an *originator* object without violating encapsulation so that the object can later be restored to the saved state (the *memento*) [21]. The object that manages the memento and restores it to the *originator* upon request is the *caretaker*. Many languages lack the fine-grained control of encapsulation and visibility required to ensure that mementos do not violate the encapsulation of the *originator*'s state. The Java option used by Cooper is to provide package-level visibility to the *memento*, with the drawback that all classes from the same package also have access to it. The AspectJ version uses a top-level interface to represent the *memento* participant and a marker interface within an abstract aspect to represent the *originator*. The abstract aspect also declares a few abstract methods for saving and restoring mementos. A concrete sub-aspect uses a *declare parents* clause to tag a concrete class as the originator and provides concrete implementations for the inherited declarations. This approach provides no guarantees against violations of encapsulation of either the *originator* or the *memento* – the *originator*'s state is given protected visibility, and the *memento* interface is intended to be reusable!

The approach taken with OT/J is to represent the *originator* as a role bound to the object concerned and to use callouts to circumvent the *private* visibility of its internal state, which is saved into an unbound role representing the *memento*. The *memento* role is a *confined role*, that is, a special variety of role that provides guarantees that no features of instances can be accessed outside the team, even those generally accessible through java.lang.Object [44]. All that a confined role allows is to obtain a reference to itself, which can be passed back to the team. A confined role is produced by making an unbound role with protected visibility to extend a Team.Confined role. The memento participant is represented this way, and all our attempts to leak it to outside the team did indeed give rise to compiler errors.

To illustrate this approach, Listing 8 shows the OT/J 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 of this example is the unbound role *memento* that extends Team.Confined.

```
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 8. Illustration of the use of confined roles to support *Memento* in Object Teams/Java (Hannemann and Kiczales 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 9. Illustration of the impact of confined roles on clients – driver for the Hannemann and Kiczales *Memento*.

To illustrate the impact of this technique on clients, Listing 9 shows the driver for this example. The lines in strikethrough illustrate attempts to use a 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. This means that although Memento no longer has the members traditionally acquired from java.lang.Object, care must still be taken not to expose Memento's state anew.

5.6. Obstacles to flexible extensibility: Composite

Composite is about setting up a unified and simplified way of interacting with a complex object (*composite*) and elements in its internal structure (*leafs*). To abstract from the differences between leafs and composites, all objects expose a common interface (*component*) to be used by clients. Because the *composite* participant holds the common context to all *leafs*, the natural OT/J way to implement *Composite* is for the team to be the *composite* and use roles to represent the *leafs*. Both the composite team and leaf roles implement a Java interface representing the *component* participant, which declares the interface common to all objects in the composition. This interface must be top-level, as a team cannot implement a Java interface enclosed within it. Figure 7 shows the Java and OT/J approaches for *Composite*. While in the Java version, all participants reside at the top level, in the OT/J version, the team (DirectoryTeam) encloses a Leaf role. Both the team and the role implement interface FileSystemComponent, which is deployed similarly in both versions.

Unfortunately, deploying it as a top-level standalone interface has the consequence that it is not a virtual type anchored to the team instance. Thus, the family of types represented by the team cannot be extended with the usual flexibility and type guarantees. If it is extended through sub-interfaces, the relevant type information associated to them does not propagate to roles in sub-teams, and extending the roles in sub-teams has no impact on the code based on the interface. The top-level interface freezes the component *participant*, and the team loses much of the flexible extensibility that is a hallmark of family polymorphism [39,43]. This implementation of *Composite* is virtually the sole case in which obstacles to extensibility can be observed. One could reason this limitation applies to all cases of *Transparent Role*, because they entail using a top-level Java interface. However, the use of a Java interface does not have an impact because only roles implement the interface, not the team.

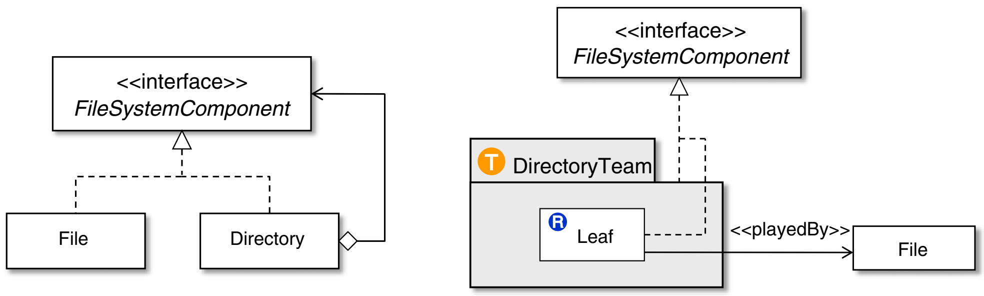


Figure 7. Implementations of *Composite* (Hannemann and Kiczales scenario) in Java (left) and Object Teams (right).

The AspectJ approach to *Composite* is very different, being based on an abstract reusable aspect that treats the *component*, *composite* and *leaf* participants as superimposed and represents them with marker interfaces. Concrete sub-aspects bind case-specific classes to those interfaces through *declare parents* clauses and compose any additional functionality. A second OT/J approach was developed for the two scenarios that mimic that approach (see also for the comparative analysis with AspectJ in Section 6). Unlike the first OT/J example described in this section, it is free of problems regarding reusability and extensibility.

5.7. *No advantages over Java: Singleton* and *Template Method*

In *Singleton* and *Template Method*, no significant advantages over Java were detected. The AspectJ approach to *Singleton* is to intercept the execution of the class' constructor and make it return the same instance each time it is called. This cannot be carried out with OT/J because the language does not support interception of constructor events. *Singleton* is the sole pattern whose OT/J examples are identical to those in Java.

There is really no way for OT/J to improve on the use of traditional inheritance prescribed by *Template Method*. In their study, HK recognized that the use of inheritance to distinguish different but related implementations is already nicely realized in OOP. Their approach to implementing the pattern in AspectJ is to use AspectJ's inter-type declarations to compose a default implementation of the template method to a Java interface, which is used instead of an abstract class. The gain is that classes become free to inherit from some other class.

The implementations in OT/J illustrate how *Template Method* can be used along the role playing dimension rather than the inheritance dimension. Roles correspond to the *abstract class* participant and declare abstract methods whose implementations are acquired through callouts from the associated base class, which becomes the *concrete class* participant. Listing 10 illustrates the approach. In this example, a template method modifies a string in three steps and returns the result.

Team TemplateTeam (Listing 10, lines 18–50) defines a role GeneratorRole (Listing 10, lines 19–29) that stands for participant *abstract class* and defines the template (Listing 10, lines 20–25). Subroles SimpleGeneratorRole (Listing 10, lines 30–34) and FancyGeneratorRole (Listing 10, lines 35–39) provide concrete implementations for the steps, obtaining them from their respective base classes – SimpleGenerator and FancyGenerator, respectively – which are the *concrete class* participants. Note that although their interfaces are identical, they need not implement a common interface. The two overloaded implementations to the team constructor (Listing 10, lines 41–43 and 44–46) use declared lifting to select the appropriate role, depending on the base instance passed. A team-level method (Listing 10, lines 47–49) calls the template method.

```
public class SimpleGenerator {
  public String prepare(String s) { return s; }
  public String filter(String s) { return s.toUpperCase(); }
  public String finalize(String s) { return s + "."; }
}
public class FancyGenerator {
  public String prepare(String s) { return s.toLowerCase(); }
  public String filter(String s) {
    s = s.replace('a', 'A');
    s = s.replace('e', 'E');
    s = s.replace('i', 'I');
    s = s.replace('o', 'O');
    s = s.replace('u', 'U');
    return s;
  }
  public String finalize(String s) { return (s+".\n(all vowels capitalized)"); }
}
public team class TemplateTeam {
  public abstract class GeneratorRole {
    public String templateMethodGenerate(String s) {
      s = _generatorStrategy.prepare(s);
      s = _generatorStrategy.filter(s);
      s = _generatorStrategy.finalize(s);
      return s;
    }
    public abstract String prepare(String s);
    public abstract String filter(String s);
    public abstract String finalize(String s);
  }
  public class SimpleGeneratorRole extends GeneratorRole playedBy SimpleGenerator {
    prepare -> prepare;
    filter -> filter;
    String finalize(String s) -> String finalize(String s);
  }
  public class FancyGeneratorRole extends GeneratorRole playedBy FancyGenerator {
    prepare -> prepare;
    filter -> filter;
    String finalize(String s) -> String finalize(String s);
  }
  public GeneratorRole _generatorStrategy;
  public TemplateTeam(SimpleGenerator as SimpleGeneratorRole simple){
    _generatorStrategy = simple;
  }
  public TemplateTeam(FancyGenerator as FancyGeneratorRole fancy) {
    _generatorStrategy = fancy;
  }
  public String generate(String s) { //Forwards to the template method
    return _generatorStrategy.templateMethodGenerate(s);
  }
}
```

Listing 10. Implementation in Object Teams of the Hannemann and Kiczales scenario for *Template Method*.

5.8. *Summing up*

Results obtained with the use OT/J to implement the GoF patterns are as follows:

- The *Factory Method* and *Abstract Factory* patterns are directly supported by OT/J owing to their support for virtual classes [38] and family polymorphism [39], respectively. That approach requires that participant classes be turned into roles within a team. In practice, OT/J factory methods that receive arguments specifying which class to instantiate cannot take advantage of these features.
- In *Memento*, OT/J can completely encapsulate the *memento* participant through use of *confined roles*.
- OT/J enables more flexible implementations of *Visitor* than with Java and AspectJ, by means of the *Double Dispatch* idiom.
- The capability of teams to enclose the context for their roles enables highly cohesive implementations for *Builder*, *Composite*, *Flyweight*, *Interpreter*, *Iterator*, *Mediator* and *State*.
- Role playing provides additional options to compose pattern roles in *Adapter*, *Decorator*, *Facade*, *Iterator*, *Prototype*, *Proxy* and *Strategy*.

- The examples for *Chain of Responsibility*, *Command* and *Observer* demonstrate the capability of OT/J to modularize entire collaborations of objects.
- Object Teams brings no special advantages for *Singleton* and *Template Method*.

6. COMPARISON WITH ASPECTJ

This section presents a systematic comparison between OT/J and AspectJ on the basis of the examples from the repository that are directly comparable: the scenarios from the HK study, for which implementations are available in both AOPLs. The criteria used in the analysis comprise the four modularity properties used in the HK study (see Section 6.1), which are equally valid for the present study. Using them here as well facilitates comparisons between the two studies. We also use extensibility as an additional criterion.

Three patterns were implemented in OT/J in two different ways: *Composite*, *Prototype* and *Visitor*. The way that seems the most natural for OT/J features is described in Section 5. The alternative approaches closely mimic the AspectJ approach for these patterns – and thus demonstrate the capability of OT/J to replicate many of the effects achieved with AspectJ. They are used in the comparison from this section because they yield reusable modules, whereas none of the implementations mentioned in Section 5 do. This has an impact on the comparison with AspectJ because one of the evaluation criteria is reusability. The alternative examples also make the comparisons more straightforward to carry out.

6.1. Modularity properties

In their comparative study of AspectJ and Java, HK used four modularity properties as a basis [30], which are defined as follows:

- **Locality**. The ability to place all code pertaining to a given concern in a module separate from other modules. When the concern is a pattern, locality is about placing all pattern code in a module separate from case-specific participants, which should be completely free from pattern code. Full source code locality for a given concern is a prerequisite for a successful modularization of that concern, as well as all the properties mentioned next.
- **Reusability**. The module can be applied to multiple, different scenarios/examples without resorting to duplication or invasive changes on the module's source code.
- **Composition transparency**. It is possible to compose multiple instances of the module on a given system in such a way that the composition of one instance of the module does not interfere with the composition of other instances.
- **(Un)pluggability** is the ability to add (plug) or remove (unplug) a module from a system without the need for invasive changes on the source code of the system's remaining modules. This enables a choice between using and not using the module in the system.

When reasoning with the above properties, we distinguish between classes that *participate* in a given composition and *clients* of the outcome of that composition. Clients reside at a different level from participants in that they use the composite functionality resulting from the composition. They often need to be aware of both the pattern functionality and the participants' core functionality, for example, to set up structures, select specific variants or perform configurations. Each pattern example used in this study includes an example-specific client Main class serving as the driver for the example. When assessing dependence between modules, the present study does not consider clients. This is consistent with the criteria used in the HK study.

In addition to the above properties, we find it insightful to include *extensibility*. Our notion of extensibility is very close to the *open-closed principle* [45,46]. We define extensibility as the ability to extend a module's functionality without changing the module's source code. In practice, this entails the creation of new modules that acquire the functionality of the extended module, add functionality of their own and can polymorphically substitute the extended module without impact on its client modules.

6.2. Discussion

Table 1 presents the modularity properties obtained from the two AOPLs, organized by pattern. All results for AspectJ are taken from the HK study [30], and the results for OT/J are all based on the implementations described in Section 5, except for *Composite*, *Prototype* and *Visitor*. In the latter three patterns, the analysis is based on alternative OT/J implementations that mimic the AspectJ approach. Table 1 also includes the classification of pattern roles proposed in the HK study into *defining* and *superimposed* [30] (Section 2.3). HK acknowledged that the distinction between defining and superimposed roles is not always clear-cut. For this reason, they signal unclear cases by placing the role names within parentheses in either or both categories.

When in doubt, the HK study leans on approaching pattern roles as superimposed. This way, they take advantage of AspectJ's ability to separate additional functionality from the core concern of a class. The classification of the *singleton* role in *Singleton* as superimposed is arguably the most controversial. The singleton nature can be considered *intrinsic* to the class involved, in which case the *singleton* participant should be classified as defining. Although the separation of the *singleton* nature to an aspect does not translate into *explicit* dependence to the aspect at the code level, it may give rise to subtle dependencies in the way clients are programmed – either different calls to the constructor are expected to return different object identities or calls are expected to always return the same identity. These assumptions are not (un)pluggable. Most of the modularity properties discussed do not seem to apply to *Singleton*; for example, it does not make sense to talk of composing the pattern multiple times (see also Section 6.6).

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 – although details on each specific limitation are not provided. We aim to give a similar meaning to the qualified 'yes' entries for the OT/J implementations.

6.3. Locality

In their paper, HK noted 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 five patterns – *Abstract Factory*, *Bridge*, *Builder*, *Factory Method* and *Interpreter* – is due to the participants in those patterns being all unambiguously defining. *Facade* is special case in that the Java and AspectJ implementations are identical but can be included in this group. In addition, limitations are indicated for five other patterns: *Command*, *Proxy*, *State*, *Template Method* and *Visitor*. In all these cases, either the pattern includes a role that is unambiguously defining or it is debatable whether a given role is superimposed.

Because locality is a prerequisite for the remaining properties, a 'no' for that property is followed by 'no' for all other properties.

The limitations of AspectJ that are an obstacle to locality are not felt with OT/J, due to a large extent to OT/J's capability to cohesively package multiple components into a team. In most cases, teams can enclose all meaningful participants as roles along with the object collaboration they embody. This enhanced cohesion brings broad-ranging benefits. For instance, it explains why the *locality* property applies in *all* OT/J cases in Table 1 – albeit with qualifications in the cases of *Command* and *Template Method*. The examples for *Command* in both AspectJ and OT/J differ from those in Java only because participants are treated as superimposed. Otherwise, the examples would be identical in all languages. The OT/J examples for *Template Method* are used to illustrate how role playing can be used the same way as inheritance for the purposes of this pattern. The OT/J example for *Visitor* achieves full code locality, despite being based on the AspectJ example, which does not.

6.4. Reusability

Regarding reusability, the two languages yield broadly comparable results overall. Table 1 includes 12 patterns giving rise to reusable modules in AspectJ (which in the case of *Iterator* is a Java interface) as

Table I. Pattern roles and modularity properties of the Object Teams/Java (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 [30].
*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.

opposed to 10 patterns with OT/J (teams in all cases). However, a number of subtle points should be considered. The entries in Table 1 for OT/J display a 'yes' for reusability whenever the module implementing the pattern is used in both scenarios. The module must also have at least one concrete member to be considered. These criteria are perhaps a bit more demanding than those used by HK. Their study comprises just one implementation per pattern, and the lack of concrete members would arguably warrant a 'no' for *Iterator* and *Memento*.

The primary advantage brought by OT/J for *Iterator* is greater cohesion, by packaging the *aggregate* and *iterator* participants together within a team. In *Memento*, the only non-abstract member is a top-level exception class.

The OT/J implementation of *Command* mimics the AspectJ approach in treating the *command* participant as superimposed and attains similar results. Therefore, the results displayed in Table 1 are identical. HK noted that reusability in the AspectJ example for *Visitor* is restricted to the cases in which the structure to be traversed has just two levels (i.e. terminal and non-terminal nodes). The OT/J implementation based on the AspectJ approach seems capable of dealing with more levels, as teams can be flexibly extended. OT/J fails to improve on the Java approach to *Singleton* – contrary to AspectJ – because its composition model does not cover constructor events.

6.5. Composition transparency

The HK study does not consider composition transparency to be applicable to *Interpreter*, *Singleton* and *State* ('n/a' in Table 1). In general, it also seems hard to apply to cases in which the approach is to enclose the participants into a cohesive whole and the pattern roles are defining, which to some extent is also the case of *Builder* and *Iterator*. Thus, Table 1 displays a 'no' for the OT/J implementation of *Builder* and a qualified 'yes' for *Iterator*.

6.6. (Un)pluggability

In a few cases, OT/J implements a pattern directly with some of its features, in which case the issue of plugging and unplugging the pattern is really about whether the language feature should be used. In such cases, reasoning in terms of modularity properties is not really applicable. This issue is further discussed in Section 6.8.

Differences in outcomes for *Bridge* are due to the lack of cohesion mechanism in AspectJ. The AspectJ example for *Bridge* is a set of inter-type declarations targeting case-specific classes, whereas with OT/J, the hierarchy of *abstractions* is encapsulated within a team, which is composed to *implementors* through role playing. The OT/J examples for *Command* mimic the AspectJ approach and attain similar results. *Facade* is the sole pattern from the HK study in which the AspectJ implementation is identical to that in Java. In this case, the Java implementation can be considered unpluggable. Naturally, the OT/J examples for *Facade* are unpluggable as well. The HK study indicates that the AspectJ example for *Proxy* has limitations, remarking that their approach – based on pointcuts and advice – does not handle cases in which the subject and proxy participants are two different objects. The OT/J can be used in such cases. However, HK also referred to the case of remote and virtual proxy: it is not clear at this point how well OT/J can handle such cases.

In Section 6.2, we argue that the pattern role of *Singleton* should be considered intrinsic to the class that implements it. The *singleton* nature gives rise to dependencies in terms of the way clients use a class. For this reason, we are sceptical of the 'yes' for unpluggability of the AspectJ *Singleton* in Table 1.

The OT/J example for *State* is based on unbound roles and yields a more cohesively structured implementation than the AspectJ approach, which is based on pointcuts and advice. However, it is also 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* is unpluggable without limitations, unlike the AspectJ approach it mimics. This is possible because it attains full locality.

6.7. Extensibility

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 – although with limitations in a few cases, especially *Composite*. The reason for the generalized 'no' for AspectJ is due to a constraint on aspect inheritance – concrete aspects cannot be extended. In fact, the design of AspectJ explicitly eschews polymorphism in most forms, save for the cases that AspectJ 'inherits' from Java [47]. Another constraining factor is the lack of a cohesion mechanism in AspectJ [41,48]. In contrast, any non-final module in OT/J can be further extended through inheritance. *Interpreter* provides a good illustration of team extensibility. The example for *Interpreter* from the HK scenario is organized as a super-team and a sub-team, but a single team or several sub-teams could have been used instead. The chosen division into two layers reflects a conceptual division of the nodes between terminal and non-terminal. In most cases, teams can 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 [43]. The only exception is the approach for *Composite* discussed in Section 5.6, because of the team implementing a top-level interface. Some limitations can be expected in implementations using the *Transparent Role* idiom, although they are not likely to have as much impact because it involves just roles, not the team.

6.8. Direct language support

Some patterns are closely associated to specific language features. *Factory Method* is closely related to the idea of polymorphic constructors, and *Abstract Factory* is closely related to virtual classes and family polymorphism. Because OT/J supports these features, implementing the patterns is a matter of using the features in the appropriate way. Because the new features are specific to teams and roles and do not apply to existing plain Java classes, the claim in Table 1 of 'direct language support' appears within parentheses.

In many circumstances, direct language support can be considered a better result than a successful modularization. However, the present analysis also illustrates a tension between language support and modularity properties. For instance, directly supporting *Abstract Factory* by using teams and roles decreases the count of cases that show unpluggability. This issue must be taken into consideration when analysing Table 1.

In the analysis of their results, HK mentioned a group of patterns whose implementations 'disappear' because of direct support from AspectJ – including *Adapter*, *Decorator*, *Proxy*, *Strategy* and *Visitor*. They nevertheless acknowledge that their implementations have inherent limitations. For instance, the advice-based implementation of *Decorator* is devoid of dynamic properties, namely the ability to dynamically reorder decorators or to distinguish between different instances of the decorated (*component*) class. These limitations motivate some qualifications to the extent to which AspectJ can be said to directly support any given pattern.

From the present study, we conclude that OT/J provides direct language support for *Abstract Factory* and *Factory Method* when participants are turned into roles. *Factory Method* further requires that the factory method be devoid of explicit arguments specifying which class to instantiate.

6.9. Summing up

Regarding the four modularity properties also used in the HK study, results can be summarized as follows:

- AspectJ yields the results similar to Java for *Facade* and OT/J yields results similar to Java for *Singleton*.
- The AspectJ implementations fail to yield code locality for six patterns (*Abstract Factory*, *Bridge*, *Builder*, *Facade*, *Factory Method* and *Interpreter*). OT/J yields code locality in *all* 23 patterns. This is because there is locality in the implementation of *Singleton*, although it is identical to that in Java.
- The AspectJ implementations yield 12 reusable modules, as opposed to 10 reusable modules for OT/J. However, one of the reusable modules from the AspectJ collection (*Iterator*) is a Java interface. OT/J also provides direct language support for *Factory Method* and *Abstract Factory*.
- OT/J attains composition transparency in 16 patterns, compared with the 14 for AspectJ. Note that we consider this property as not applicable to *Interpreter*, *Singleton* and *State*. If we exclude those patterns plus those for which OT/J provides direct support, OT/J 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 – although the exact set of patterns differs.

The relative advantages over one language to another in terms of the four aforementioned modularity properties do not seem particularly pronounced. However, there are a number of important issues that are not expressed in Table 1. Regarding extensibility, none of the AspectJ aspects are extensible [47] because of (seemingly intended) limitations on the design of AspectJ. In contrast, all OT/J teams are extensible with the exception of one of the variants for implementing *Composite* because the team implements a top-level Java interface (Section 5.6). In terms of direct language support for specific patterns, OT/J is the winner, on account of its support for *Factory Method* and *Abstract Factory* (when classes are turned into roles). The advantage of OT/J over AspectJ is also clear regarding how flexibly the resulting modules can be adapted or extended to new situations. One of the contributions of OT/J is to enrich the set of variation points in a program's code, easing program extension in ways that follow the open-closed principle [45].

Object Teams/Java also has the advantage over AspectJ that its features 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, yielding simpler solutions. This advantage is important as many of the GoF patterns relate to relationships between individual instances rather than classes. However, OT/J requires additional code in a number of implementations to pass objects from the team to instances of its roles.

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* because of AspectJ's support for constructor joinpoints.

In general, OT/J seems better suited to implement the GoF patterns than AspectJ.

7. RELATED WORK

We divide work on the assessment of programming languages and models in two groups: those that are based on design patterns like the present study and those based on different assessment approaches.

7.1. Assessment approaches based on design patterns

Many works exist that focus on the impact of a given set of language features on the implementation of design patterns [49–51,32]. Some studies have been carried out that focus on a restricted set of patterns [36,52,53] or even a single pattern (e.g. *Observer* [54,55] and *Visitor* [56]). The present study is focused on specific language implementations of the well-known GoF patterns as a means for an evaluation. A few studies are in this vein. For instance, Schmager *et al*. carried out an assessment of the Go language using patterns and a framework [57].

The results presented in this paper are an update on the results presented in our SAC/OOPS paper [58] and supersede it. The previous work does not provide an in-depth analysis and does not take into account the implementations of some of the Cooper scenarios, whose collection in OT/J was incomplete at the time.

The study by Rajan [11] 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 an alternative model to AspectJ that unifies the notions of class and aspect in a module construct called the *classpect*. 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*). 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 or to ensure that only selected participant objects are subject to aspect compositions. Because OT/J also supports instance-level composition, it yields similar advantages over AspectJ.

The study by Kuhlemann *et al*. [32] is another work based on the code examples from the HK study. Its aim is to assess and explore the perceived duality between AOP and *feature-oriented programming*

(FOP). Kuhlemann *et al*. observed that 19 of the 23 examples from the HK study can be transformed straightforwardly into equivalent FOP examples. They provide a set of general rules on how to transform AOP programs into FOP programs and discuss their experiences regarding these transformations. They also illustrate similarities and differences between AOP and FOP solutions. The description of the transformations bears some resemblances to the set of guidelines we proposed in Section 4.2, although the work by Kuhlemann *et al*. is more detailed. The collection of refactorings by Monteiro and Fernandes to transform Java code into AspectJ is also in a similar vein and more detailed still [31,59].

Hirschfeld *et al*. [28] discussed design pattern implementation in AspectS, using two patterns to illustrate – *Visitor* and *Decorator*. AspectS [60] 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. Instead, it makes use of meta-object composition and method-call interception. Contrary to the mechanisms of AspectJ, those of AspectS can work at the instance level. In their discussion, Hirschfeld *et al*. distinguished 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 but without a fundamental redesign to leverage AOP constructs. The benefits of this approach are better code locality, reusability, composability, implementation modularity and comprehensibility. The latter is a redesign based on AOP-specific constructs. Hirschfeld *et al*. claimed that native AOP solutions avoid certain drawbacks such as 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. Native AOP solutions also eliminate 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 5.4) is a native AOP solution – avoiding explicit *accept* and *visit* methods – whereas the implementation that mimics the AspectJ example from the HK study (Section 6) – 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*. [7] and Cacho *et al*. [8] use adapted versions of the material used in the HK study [30] to perform comparisons between the OOP and AOP implementations on the basis of a set of quantitative metrics. The present study does not use 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 are likely to require new metrics, for example, to account for virtual classes (roles) and family classes (teams), plus the various new kinds of polymorphism supported. An exploratory study of the metrics supported by the OTDT Eclipse plug-in was explored by Lima *et al*. [61], using the subset of examples in Java and OT/J presented in our SAC/OOPS paper [58]. The study concludes that the metrics' support is unsuitable for comparisons between Java and OT/J, as the plug-in does not take into account the constructs specific to OT/J.

Sousa and Monteiro reported on a preliminary study of CaesarJ that is also based on design patterns, although only a few of the patterns are implemented and analysed [12]. The CaesarJ model [41,62] has many similarities with the two AOPLs compared here. It supports virtual classes and family polymorphism and uses pointcuts and advice to compose aspect components to specific applications and software systems. CaesarJ decouples a software component's implementation from the component's binding to the remaining parts of the system by means of separate inheritance hierarchies that are composed through multiple inheritance. Both hierarchies initiate at a *collaboration interface* module providing the contract through which implementations and bindings share common operations. Pointcuts and advice come less to the fore than with AspectJ and are used mainly to compose the component to other parts of a system. The approach taken in that study is similar to that reported here, relying on the implementation in CaesarJ of pre-existing pattern examples in Java. Family polymorphism brings benefits similar to those reported with OT/J, including direct language support for *Abstract Factory*.

Nordberg [27] proposes a set of principles for managing dependences between modules in complex systems, which lead to more stable module structures, easier to understand and more maintainable. The principles state that (1) a dependence must not form cycles; (2) modules should depend on abstractions (e.g. declarations of interfaces), not on concrete elements; and

(3) the direction of dependence between modules should always be from less stable to more stable. Nordberg analyses the dependence originated by several design patterns and identifies cases in which traditional OOP implementations fail to meet the principles. For instance, traditional OOP implementations of *Visitor* violate all three principles: the dependence 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 such dependences so as to follow the principles. The present study does not use the principles proposed by Nordberg in the analysis of the OT/J implementations it describes, but very similar principles are followed in both the AspectJ and OT/J collections of implementations.

7.2. Other assessment approaches

Lopez-Herrejon *et al.* presented a study of five technologies in light of their support for modularization and composition of *features* [48] – AspectJ, Hyper/J, Jiazzi, Scala and AHEAD. Product line design provides the underlying motivation – a product line is a family of related programs, each program is a unique combination of features and a feature is an increment in feature functionality. They adopted a variant of the *expression problem* [63,64] to drive their study, which is treated as a canonical problem in product line design. Like the present study, they also remarked on the lack of a cohesion mechanism in AspectJ to group all elements of a feature together and manipulate them as a single entity. They concluded from the study that none of the technologies assessed provide a satisfactory solution to the problem of building product lines. They argued that product line architects reason about programs in terms of their features, not in terms of their code or implementing technologies. They attempted to express the reasoning in an abstract model of features that equates compositional reasoning with algebraic reasoning in order to abstract from code or specific implementing technologies.

Greenwood *et al.* reported on an empirical study on the impact of AOPLs on design stability of a software system in the context of an evolution scenario [9]. The study covers 10 releases of the system and coded in three languages – Java, AspectJ and CaesarJ. However, they used more than one AOPL just to ensure that their derived conclusions are broad and agnostic to specific AOPL features. No comparisons between AOPLs are made. The study analyses modularity, change propagation, concern interaction and identification of architectural ripple effects. They concluded that (1) the AOP implementations tend to have a more stable design, particularly when a change relates to a crosscutting concern; (2) changes tend to be much less intrusive and more simplistic when AOP is used; (3) greater architectural ripple effects occur with the OO design when persistence-related exceptions are introduced. In certain circumstances, aspectual decompositions do perform worse, which tends to occur when evolutionary scenarios target patterns such as *Command* and *State* because applying these patterns violate pivotal design principles such as narrow interfaces and low coupling. Even though the AOP implementations tend to require less invasive changes, sometimes the modifications propagate to components that are not the direct target of the change scenario. Regarding the design stability and concern interaction, Greenwood *et al.* concluded that aspect decomposition narrows boundaries of concern dependence but gives rise to tighter and intricate interactions.

Lämmel and Ostermann [65] analysed the language concept of *type classes*, as supported by the functional programming language Haskell. The analysis is made in light of the contribution that type classes can be expected to bring to software extension and software integration. To drive the analysis, Lämmel and Ostermann used formulations of a number of extension or integration problems, namely the *expression problem* [63,64], the *framework integration problem* [66], the *tyranny of dominant decomposition* [3], *scattering* and *tangling* [2] and the *component integration problem* [67]. By means of an analysis of the published state of the art, Lämmel and Ostermann concluded that type classes provide a principled mechanism for software extension and integration from which useful insights can be derived. They pinpoint several limitations of type classes and argue that further research on this concept is likely to contribute to a better understanding of the relation between advanced OO features and functional programming as regards the challenges of software extension and integration.

8. FUTURE WORK

The pattern implementations used for the present study 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 OOP frameworks. Another front is the assessment of the impact of AOPLs on framework design and development. In particular, *pattern density* [68] (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 OOP 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 *et al*. [7] and Cacho *et al*. [8] and similar studies. Material used as a basis for the present study comprises single snapshots. It could be complemented with studies involving evolution scenarios, of which the work by Greenwood *et al*. [9] is an example. One interesting possibility would be to extend that study to cover Object Teams and other AOPLs.

Another front is to carry out the aforementioned kinds of study to other advanced programming languages. For instance, Scala [69] is a language that also supports virtual classes and family polymorphism, and it has a number of features that directly support a number of GoF patterns. In addition, it seems capable of emulating a number of AspectJ-like features, including AspectJ-like advice [56]. It would be interesting to assess the extent to which Scala can emulate the various well-known AOP features associated to AspectJ.

9. CONCLUSION

This paper describes a study of the Object Teams programming language [13,14], which is compared with Java and AspectJ. The basis for comparisons comprises two complete collections of implementations in Object Teams of all 23 GoF design patterns. Both collections existed prior to the present study – both in Java and one in AspectJ as well. The analysis of results is made in two parts. The first assesses the contributions that Object Teams brings to the implementation of the design patterns when compared with Java and AspectJ. The second part is a systematic comparison with the AspectJ examples, using a set of modularity properties used in a previous study focusing on Java and AspectJ [30], to which we add extensibility.

Although Object Teams [15] was designed with goals different from those of AspectJ [18], we conclude from the present study that Object Teams can mimic the design and programming styles of AspectJ in most cases. The *Singleton* pattern is the main exception, because of a lack of support for constructor interception. Object Teams also enables different alternative solutions to a few patterns, most notably *Memento* and *Visitor*. In some circumstances, Object Teams provides direct language support for *Factory Method* and *Abstract Factory*.

A limitation of AspectJ that comes much to the fore in the comparisons is a lack of a cohesion mechanism, which accounts for the failure to modularize a number of patterns. In contrast, virtual classes and family polymorphism provide Object Teams with an effective cohesion mechanism, enabling Object Teams to modularize virtually all patterns. In general, the AspectJ implementations are not extensible because of the absence of inheritance for concrete aspect modules. In contrast, the modules of Object Teams can be flexibly extended in most cases.

ACKNOWLEDGEMENTS

We thank Stephan Herrmann for providing insightful feedback on a draft of this paper.
This work is partially supported by projects PRIA (UTAustin/CA/0056/2008) and GasPar (PTDC/EIA-EIA/108937/2008) funded by Portuguese FCT/MCTES and FEDER.

REFERENCES

1. 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.

2. Kiczales G, Lamping J, Mendhekar A, Maeda C, Lopes C, Loingtier J, Irwin J. Aspect-oriented programming. In *Proceedings of 11th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science vol. 1241. Springer-Verlag, 1997; 220–242.
3. Tarr P, Ossher H, Harrison W, Sutton Jr SM. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*. ACM Press. 1999; 107–119. DOI: 10.1145/302405.302457.
4. Rashid A, Moreira A. Domain models are not aspect free. In *Proceedings of Model Driven Engineering Languages and Systems*. Springer. 2006; 155–169. DOI: 10.1007/11880240_12.
5. Filman RE, Elrad T, Clarke S, Aksit M (eds). *Aspect-Oriented Software Development*. Addison-Wesley: Reading, MA, 2005. ISBN: 0321219767 978-0321219763.
6. Brichau J, Haupt M. Report describing survey of aspect languages and models. *AOSD-Europe Deliverable D12*, AOSD-Europe-VUB-01, 2005.
7. Garcia A, Sant'Anna C, Figueiredo E, Kulesza U, Lucena C, Staa A. Modularizing design patterns with aspects: a quantitative study. In *Transactions on Aspect-Oriented Software Development I*. Lecture Notes in Computer Science vol. 3880. Springer-Verlag: Berlin, Germany, 2006: 36–74. DOI: 10.1007/11687061_2.
8. Cacho N, Sant'Anna C, Figueiredo E, Garcia A, Batista T, Lucena C. Composing design patterns: a scalability study of aspect-oriented programming. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development*. ACM Press, New York. 2006; 109–121. DOI: 10.1145/1119655.1119672.
9. Greenwood P, Bartolomei 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. In *Proceedings of the 21st European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science vol. 4609. Springer-Verlag, 2007; 176–200. DOI: 10.1007/978–3–540–73589–2_9.
10. Xin B, McDirmid S, Eide E, Wilson CH. A comparison of Jiazzi and AspectJ for feature-wise decomposition. Technical Report UUCS-04-001, University of Utah, March 2004.
11. Rajan H. Design pattern implementations in Eos. In *Proceedings of the 14th Conference on Pattern Languages of Programs*. ACM Press. 2007; 9:1–9:11. DOI: 10.1145/1772070.1772081.
12. Sousa E, Monteiro MP. Implementing design patterns in CaesarJ: an exploratory study. In *Proceedings of the 2008 AOSD Workshop on Software Engineering Properties of Languages*. ACM Press, 2008; 6:1–6:6. DOI: 10.1145/1408647.1408653.
13. The Object Teams at eclipse. (Available from: http://www.eclipse.org/objectteams/). [24th July 2012].
14. Herrmann S, Hundt C, Mosconi M. *ObjectTeams/Java Language Definition version 1.3* (OTJLD). Technical Report 2009/08, Technische Universität Berlin, 2009.
15. Herrmann S. Object teams: improving modularity for crosscutting collaborations. *Proceedings of International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, 2002; 248–264. ISBN: 3-540-00737-7.
16. Herrmann S. A precise model for contextual roles: the programming language ObjectTeams/Java. *Applied Ontology* 2007; **2**(2): 181–207.
17. The Object Teams web original site. (Available from: http://www.objectteams.org/). [24th July 2012].
18. Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J, Griswold WG. An overview of AspectJ. In *Proceedings of 15th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science vol. 2072. Springer-Verlag, 2001, 327–335. ISBN: 3-540-42206-4.
19. Laddad R. *AspectJ in Action*, (2nd edn). Manning: Greenwich, 2009. ISBN: 1933988053 9781933988054.
20. Colyer A, Clement A, Harley G, Webster M. *Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools*. Addison-Wesley: Reading, MA, 2004.
21. Gamma E, Helm R, Johnson R, Vlissides J. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley: Reading, MA, 1995.
22. Alpert S, Brown K, Woolf B. *The Design Patterns Smalltalk Companion*. Addison-Wesley: Boston, 1998.
23. Harmes R, Diaz J. *Pro Javascript Design Patterns*. Apress: New York, 2008. ISBN: 159059908X 978-1590599082.
24. Olsen R. *Design Patterns in Ruby*. Addison-Wesley: Reading, MA, 2007. ISBN: 978-0321490452.
25. Bracha G, Cook W. Mixin-based inheritance. In *Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM Press, New York, 1990; 303–311. DOI: 10.1145/97946.97982.
26. The Demeter/Java project web site. (Available from: http://www.ccs.neu.edu/home/lieber/Demeter-and-Java.html) [24th July 2012].
27. Nordberg III M. Aspect-oriented dependency management, Chapter 24. In *Aspect-Oriented Software Development*. Addison-Wesley: Reading, MA, 2005; 557–584.
28. Hirschfeld R, Lämmel R, Wagner M. Design patterns and aspects – modular designs with seamless run-time integration. *3rd German GI Workshop on Aspect-Oriented Software Development*, 8 pages, 2003.
29. Lesiecki N. Enhance design patterns with AspectJ, Part 2, AOP@Work series at developerWorks, IBM, 2005. (Available from: http://www.ibm.com/developerworks/java/library/j-aopwork6/index.html) [7th April 2012].
30. Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, 2002; 161–172. DOI: 10.1145/582419.582436.
31. Monteiro MP, Fernandes JM. Towards a catalogue of refactorings and code smells for AspectJ. In *Transactions on Aspect-Oriented Software Development I*. Springer-Verlag, LNCS 3880, 2006; 214–258. ISBN: 3-540-32972-2 978-3-540-32972-5.

32. Kuhlemann M, Rosenmüller M, Apel S, Leich T. On the duality of aspect-oriented and feature-oriented design patterns. In *Proceedings of 6th Workshop on Aspects, Components, and Patterns for Infrastructure Software*. ACM Press, 2007. DOI: 10.1145/1233901.1233906.
33. Cooper J. *Java Design Patterns: A Tutorial*. Addison-Wesley: Reading, MA, 2000.
34. Fowler M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley: Reading, MA, 1999.
35. Yuen I, Robillard M. Bridging the gap between aspect mining and refactoring. In: *Proceedings of the 3rd Workshop on Linking Aspect Technology and Evolution*. ACM Press, 2007. DOI: 10.1145/1275672.1275673.
36. Herrmann S, Hundt C, Mehner K. Translation Polymorphism in Object Teams. Technical Report 2004/05, Fak. IV, Technical University Berlin, 2004.
37. Reenskaug T. *Working with Objects – The OORAM Software Engineering Method*. Prentice Hall: New Jersey, 1996. ISBN: 0134529308 978-0134529301.
38. Madsen OL, Moller-Pedersen B. Virtual classes: a powerful mechanism in object-oriented programming. In Proceedings of the *OOPLSA '86* Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications. ACM Press, 1989; 397–406. DOI: 10.1145/74877.74919.
39. Ernst E. Family polymorphism. In *Proceedings of the 15th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science vol. 2072, Springer-Verlag, 2001; 303–326.
40. Herrmann S. Composable Designs with UFA. *Workshop on Aspect-Oriented Modeling with UML*, 2002.
41. Mezini M, Ostermann K. Untangling crosscutting models with Caesar Chapter 8,. In *Aspect-Oriented Software Development*. Addison-Wesley: Reading, MA, 2005; 165–199.
42. Filman RE, Friedman DP. Aspect-oriented programming is quantification and obliviousness, Chapter 2. In *Aspect-Oriented Software Development*. Addison-Wesley: Reading, MA, 2005; 21–35.
43. Ernst E. Higher-order hierarchies. In *Proceedings of the 17th European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science vol. 2743. Springer-Verlag, 2003; 303–329. DOI: 10.1007/b11832.
44. Herrmann S. Confinement and Representation Encapsulation in Object Teams. Technical Report 2004/06, Fak. IV, Technical University Berlin, 2004.
45. Meyer B. *Object-Oriented Software Construction*, (2nd edn). Prentice Hall: New Jersey, 1997. ISBN: 0-13-629155-4.
46. Martin RC. The Open-Closed Principle. C++ Report, vol. 8, Jan. 1996.
47. Ernst E, Lorenz DH. Aspects and polymorphism in AspectJ. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development*. ACM Press, 2003; 150–157. DOI: 10.1145/643603.643619.
48. Lopez-Herrejon R, Batory D, Cook W. Evaluating support for features in advanced modularization technologies. In *Proceedings of the 19th European conference on Object-Oriented Programming*, Lecture Notes in Computer Science vol. 3586. Springer-Verlag, 2005; 169–194. DOI: 10.1007/11531142_8.
49. Baumgartner G, Läufer K, Russo VF. On the Interaction of Object-Oriented Design Patterns and Programming Languages. Technical report CSD-TR-96-020, Purdue University, 1996.
50. Sullivan GT. Advanced programming language features for executable design patterns: better patterns through reflection. *Artificial Intelligence Laboratory Memo AIM-2002-005*, Artif. Intel. Lab, MIT, 2002.
51. Gibbons J. Design patterns as higher-order datatype-generic programs. In *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*. ACM Press, 2006; 1–12. DOI: 10.1145/1159861.1159863.
52. Kouskouras K, Chatzigeorgioua A, Stephanides G. Facilitating software extension with design patterns and aspect-oriented programming. *Journal of Systems and Software* 2008; **81**(10): 1725–1737 DOI: 10.1016/j.jss.2007.12.807.
53. Monteiro MP, Fernandes JM. Pitfalls of AspectJ implementations of some of the Gang-of-Four design patterns. In *DSOA'2004 Workshop at JISBD 2004* (IX Jornadas de Ingeniería de Software y Bases de Datos), Málaga, Spain, 2004.
54. Piveta EK, Zancanella LC. Observer pattern using aspect-oriented programming. *3rd Latin American Conference on Pattern Languages of Programming*, 2003. Available from: http://www.cin.ufpe.br/~sugarloafplop/final_articles/20_ObserverAspects.pdf [24th July 2012].
55. Bernardi ML, Lucca GA. Improving design pattern quality using aspect orientation. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*. IEEE press, 2005; 206–218. DOI: 10.1109/STEP.2005.14.
56. Oliveira B, Wang M, Gibbons J. The visitor pattern as a reusable, generic, type-safe component. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*. ACM Press, 2008; 439–456. DOI: 10.1145/1449764.1449799.
57. Schmager F, Cameron N, Noble J. GoHotDraw: evaluating the Go programming language with design patterns. In *2nd Workshop on Evaluation and Usability of Programming Languages and Tools*. ACM Press, 2010; 10:1–10:6. DOI: 10.1145/1937117.1937127.
58. Gomes J, Monteiro MP. Design pattern implementation in Object Teams. In *Proceedings of the 25th Symposium on Applied Computing – Special Track on Object Oriented Programming Languages and Systems*. ACM Press, 2010; 2119–2120. DOI: 10.1145/1774088.1774534.
59. Monteiro MP, Fernandes JM. An illustrative example of refactoring object-oriented source code with aspect-oriented mechanisms. *Software: Practice and Experience* 2008; **38**(4): 361–396. DOI: 10.1002/spe.835.
60. 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*, 2003; 216–232.
61. Lima A, Goulão M, Monteiro MP. Evidence-based comparison of modularity support between Java and Object Teams.*1st Workshop for Empirical Evaluation of Software Composition Techniques*, 2010.
62. Aracic I, Gasiunas V, Mezini M, Ostermann K. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*. Springer LNCS vol. 3880, 2006; 135–173. DOI: 10.1007/11687061_5.

63. Wadler P. The expression problem. Message originally posted on the Java Genericity mailing list. 1998. Available from: http://www.daimi.au.dk/~madst/tool/papers/expression.txt [24th July 2012].
64. Torgersen M. The expression problem revisited. Four new solutions using generics. In *18th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science vol. 3086. Springer-Verlag, 2004; 123–143. ISBN: 3-540-22159-X.
65. Lämmel R, Ostermann K. Software extension and integration with type classes. In *Proceedings of the 5th International Conference on Generative Programming and Component Engineering*. ACM Press, 2006; 161–170. DOI: 10.1145/1173706.1173732.
66. Mattson M, Bosch J, Fayad ME. Framework integration – problems, causes, solutions. *Communications of the ACM* 1999; **42**(10): 80–87.
67. Hölzle U. Integrating independently-developed components in object-oriented languages. In *Proceedings of the 7th European Conference on Object-Oriented Programming*. Lecture Notes in Computer Science vol. 707. Springer-Verlag, 1993; 36–56. ISBN: 3-540-57120-5.
68. Riehle D, Brudermann R, Gross T, Mätzel KU. Pattern density and role modelling of an object transport service. ACM Computing Surveys, vol.32. ACM Press, 2000. DOI: 10.1145/351936.351946.
69. 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.