# Aspect-Oriented Programming to Improve Modularity of Object-Oriented Applications

Jose M. Felix
Principality of Asturias, Computer Science Department, Oviedo, Spain
Email: jmanuelfr@princast.es

Francisco Ortin
University of Oviedo, Computer Science Department, Oviedo, Spain
Email: ortin@lsi.uniovi.es

***Abstract*— The separation of concerns design principle improves software reutilization, understandability, extensibility and maintainability. By using the object-oriented paradigm, it is not always possible to separate into independent modules the different concerns of an application. The result is that the source code of crosscutting concerns are tangled and scattered across the whole application. Aspect-oriented programming offers a higher level of modularity, providing a solution for the code tangling and scattering problem. To show how aspect-oriented programming can be used as a suitable mechanism to improve the modularity of object-oriented applications, this divulgative article presents the implementation of a typical design pattern following both the object- and aspect-oriented paradigms. The two approaches are compared from the modularity perspective, establishing a discussion on the benefits provided and is current use.**

***Index Terms*—aspect-oriented programming, modularity, crosscutting concerns, AspectJ, separation of concerns**

## I. INTRODUCTION

Designing modular systems is fundamental for managing software complexity and improving its reusability, understandability, extensibility and maintainability [1]. At the implementation level, programming languages provide mechanisms to perform this modularization. Some common features of object-oriented languages used to facilitate the modularization of application abstractions are methods, classes, packages, namespaces and annotations. There exist some other modularization mechanisms, not directly supported by programming languages, which provide a higher level of abstraction. Examples of these mechanisms are components, design patterns, application frameworks, and architectural patterns.

In the software development process, there are cases when some system abstractions cannot be directly modularized with the mechanisms provided by a programming language [2]. A vector sorting algorithm can be implemented in a unique class method. However, functionalities such as error detection and correction, logging, persistence and security cannot be directly modularized with the mechanisms provided by common object-oriented languages [3].

The different (functional and non-functional) requirements demanded to an application are called software *concerns* [4]. The Separation of Concerns (SoC) design principle is aimed at separating a computer application into distinct modules, such that each one addresses a separate concern [4].

Some concerns cannot be directly modularized in classic object-oriented languages because those languages have not sufficient expressiveness to implement them in independent modules. In that case, the implementations of those concerns *cut across* multiple abstractions in a program. For this reason, these concerns are said to be *crosscutting* [5]. Figure 1 shows a real example. This figure presents the modularization of the Apache Tomcat application server implementation [6]. Each vertical bar shows one implementation module, and its size is proportional to the number of lines of code. The left-hand side of Figure 1 shows in red the lines of code whose concern is XML document parsing. It can be seen how that functionality is placed in one single module. The right-hand side of Figure 1 shows the source code distribution of the logging concern. This is an example of a crosscutting concern: its source code is not placed in a unique module (code scattering), and every module, including XML parsing, contains code of this concern (code tangling). This code scattering and tangling issues indicate that the implementation-level modularization is not appropriate, leading to reusability, understandability and maintainability limitations [4].

The main contribution of this divulgative paper is a practical analysis of how Aspect-Oriented Programming (AOP) provides an alternative mechanism to solve the code tangling and scattering problems in the implementation of crosscutting concerns. In order to do that, Section II introduces the main AOP concepts and AspectJ, one of the most widely used AOP tools nowadays [7]. Section III presents an example of a common design problem, comparing the object- and aspect-oriented implementations. Afterwards, the suitability of AOP for improving modularity is discussed

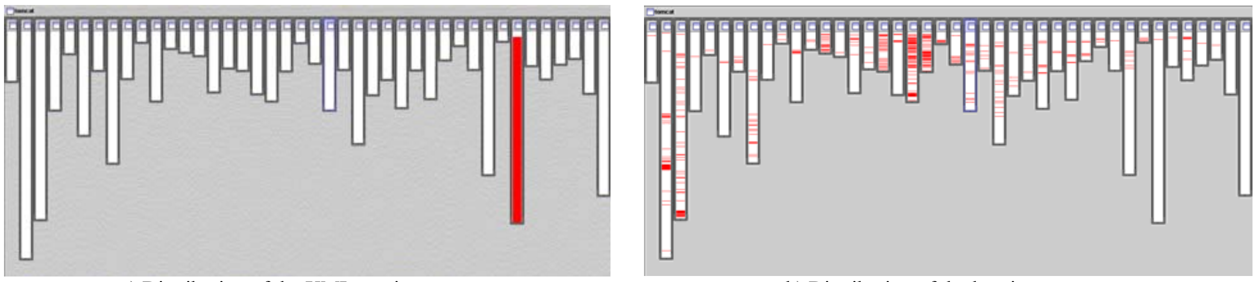
a) Distribution of the XML parsing concern b) Distribution of the logging concern

Figure 1. Distribution of concerns (source code per module) in the implementation of Apache Tomcat [6].

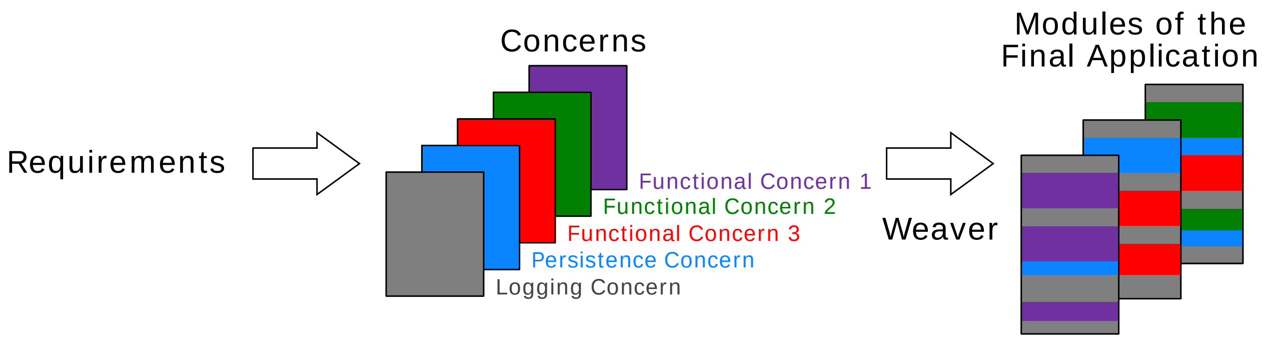

Figure 2. Weaving the different concerns identified from the application requirements.

in Section IV, and the conclusions and future work are presented in Section V.

## II. Aspect-Oriented Programming

Separating the different concerns of an application is an objective in all the steps of the software development process. Aspect-orientation is a particular mechanism to achieve the SoC goal. Aspect-Oriented Software Development (AOSD) focuses on the identification, specification and representation of cross-cutting concerns and their modularization in all the steps of the software development process. Therefore, aspect-orientation can be applied to requirement engineering, business process management, system architecture, modelling and design [8]. In this article we focus on aspect-oriented programming, which is centered on the programming techniques and tools to support the modularization of concerns at the level of the application source code [4].

As shown in Figure 2, the different concerns of an application are identified from its requirements. Before its implementation, the application concerns are conceptually separated. These concerns comprise both functional (i.e., problem domain) and non-functional (e.g., persistence or logging) requirements. The objective of AOP is to modularize all these concerns. The aspect *weaver* is the tool that takes the different concerns of an application and generates the target program. If the final application is coded in an object-oriented language, such as Java or C#, the target code of the different concerns may be tangled and scattered. Figure 2 shows how the final implementations of the functional concerns 1 and 2 are placed in one single module, tangled with the code of other concerns (e.g., logging). However, the rest of concerns are scattered among multiple modules, and tangled with the code of other concerns. Therefore, AOP raises the level of abstraction, offering the programmer a modularization mechanism superior to that provided by object orientation. The aspect weaver is the tool that translates the aspect-oriented abstractions into the object-oriented ones.

There exist two approaches for representing concerns in AOP. Asymmetric AOP distinguishes the base code (traditional classes in the classical object orientation) from the aspects. Aspects represent crosscutting concerns that, due to the aspect weaver, can be modularized. Therefore, in asymmetric AOP, an aspect must be used to be woven with a class (or another aspect). AspectJ is an example tool that provides asymmetric AOP [7]. On contrary, symmetric AOP is based on the unique concept of class/aspect. Any class can act as an aspect and be woven with any other class (or aspect). Hyper/J is an example of symmetric AOP [9].

In general, aspect weaving is performed statically, before application execution. AspectJ also provides load-time aspect weaving, when classes are about to be loaded into memory by the virtual machine. There are also AOP tools that allows aspect weaving and unweaving at runtime, when the application is being executed [10]. These dynamic weavers adapt running applications at stable points of execution. Examples of these dynamic AOP tools are the JAsCo, PROSE and DSAW platforms [11].

Aspect-oriented programming has been included in widespread application server frameworks such as JBoss or Spring. There are other implementations, such as JAC,

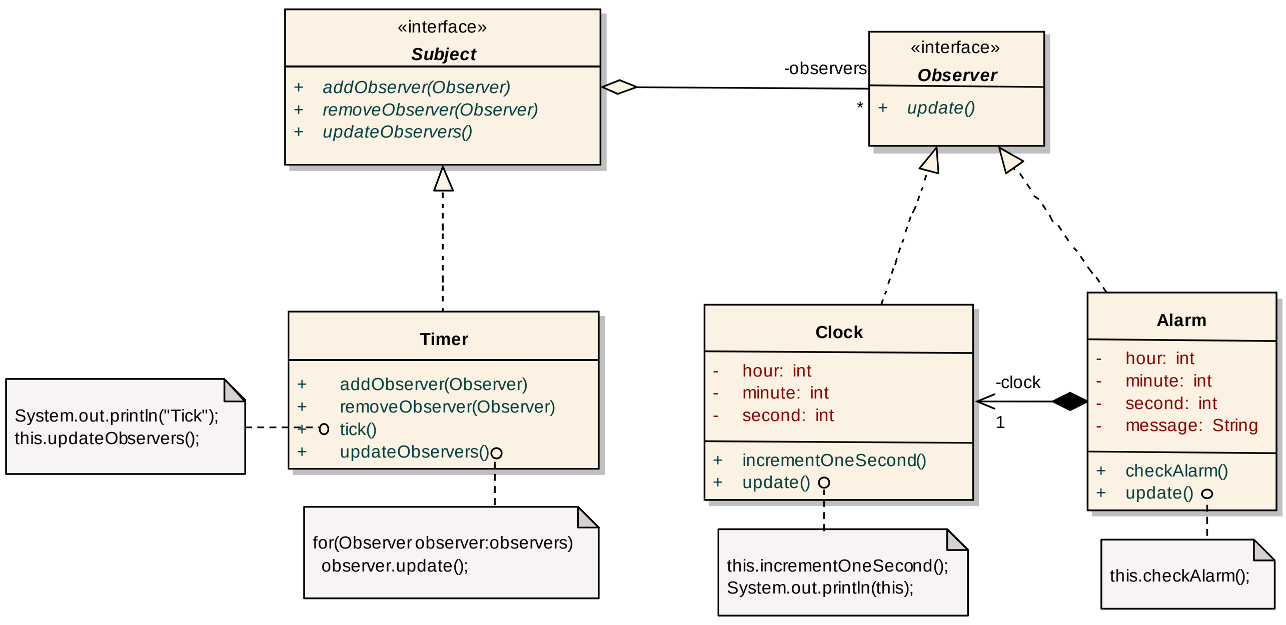

Figure 3. Object-oriented implementation of the Observer design pattern.

aimed at developing an aspect-oriented middleware layer. Another criterion to classify AOP tools is their domain. They could be used for general purposes, such as AspectJ, Compose* or CaesarJ; or specific for one domain: DiSL, for dynamic program analysis; AGOL, for multi-agent systems; and LLDSAL for dynamic code-generation and program modification.

## III. A Higher Level of Modularity

We show an example of how AOP provides a higher level of modularity compared with object-oriented programming. For this purpose, we first identify the different concerns in a classical design problem, and implement them in the Java programming language (Section III.A). Afterwards, we solve the same problem in AOP using AspectJ, comparing the modularization of this approach with the object-oriented one (Section III.B).

### *A. The Observer Design Pattern*

The Observer design pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [12]. Figure 3 shows an example instance of the Observer design pattern. The `Timer` class plays the role of the object which state changes (called `Subject` in the design pattern). `Timer` is associated to a collection of `Observers`. `Timer` objects execute their `tick` method every second.

There are two types of `Observers`. `Clock` objects increment their state (`hour`, `minute` and `second`) in one second every time the `incrementOneSecond` method is called. An `Alarm` object utilizes one `Clock` to check the current time. When that `Clock` reaches the `hour`, `minute` and `second` saved as the state of the `Alarm` object, the corresponding `message` is shown in the console.

Each `Subject` holds a collection of `Observers` by means of the `observers` field, and the `addObserver` and `removeObserver` methods. Although there may be different types of `Subjects`, in this example we only consider the `Timer` class. Whenever a change in the state of a `Subject` occurs, this object notifies the associated `Observers` of this change by calling the `updateObservers` method. As shown in Figure 3, `Timer` performs this notification at the end of the `tick` method implementation (every second). The execution of `updateObservers` implies passing the `update` message to each of the associated `Observer` objects registered for that particular `Subject`. For `Clocks`, `update` means incrementing its current time in one second; for `Alarms`, the time of its clock is compared with its own state and, if both are equal, its `message` is shown in the console (i.e., the alarm is triggered).

In the Observer design pattern, the `Subjects` trigger the events that may occur dynamically (`updateObservers`), and each type of the registered `Observer` defines how to respond to these events (implementation of the `update` method). These are the functional concerns we have identified in this example:

1. The `Timer`, `Clock` and `Alarm` domain entities, modularized in different classes.
2. The `Subject` and `Observer` roles, to be played by the existing entities of the problem domain. In the object-oriented approach presented, these roles are modularized in two interfaces (Figure 3). However, the identification of the roles played by each entity requires a subtle modification of each entity module: declaring that the entity (class) `implements` the role (interface).
3. The collection of `Observers` associated to a `Subject`. The concern of managing this collection is modularized inside the `Timer` class

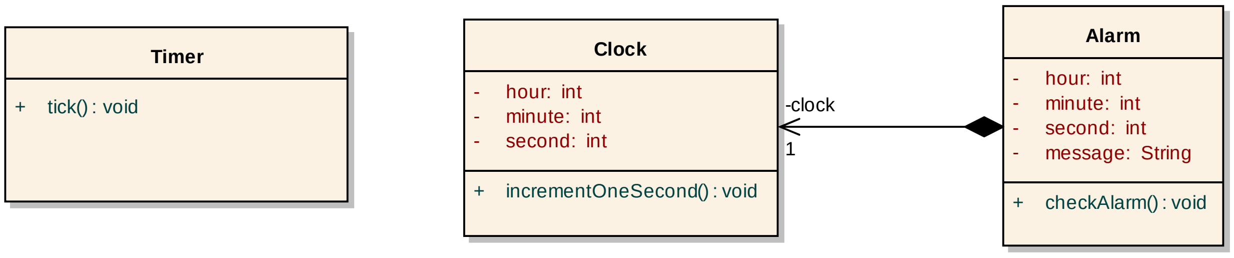

Figure 4. Domain entities without concerns of the Observer design pattern.

```
01: package observer.aspectj.aspects;
02:
03: import java.util.Set;
04: import java.util.HashSet;
05:
06: abstract aspect ObserverDesignPattern {
07:
08:   private Set<Observer> Subject.observers =
                                  new HashSet<Observer>();
09:
10:   public synchronized void Subject.addObserver(
                                       Observer observer) {
11:     if (this!=null) {
12:       this.observers.add(observer);
13:       observer.setSubject(this);
14:     }
15:   }
16:
17:   public synchronized void
          Subject.removeObserver(Observer observer) {
18:     this.observers.remove(observer);
19:   }
20:
21:   public synchronized
                        void Subject.updateObservers() {
22:     for(Observer observer:observers)
23:       observer.update();
24:   }
25:
26:   abstract pointcut stateChanges(Subject s);
27:
28:   after(Subject subject) returning :
                                 stateChanges(subject) {
29:     subject.updateObservers();
30:   }
31: }
```

Figure 5. Aspect-oriented implementation of the Observer design pattern.

(the `observers` field, and the `addObserver` and `removeObserver` methods). The implementation of this concern could be placed in an abstract class, and be inherited by those classes requiring it. Although this technique is widely used, the class will not be able to derive from another different class (Java is a single-inheritance object-oriented language).

4. Event notification, modularized in the `Timer` class. In this class, the `updateObservers` invocation (event notification) is tangled with the functional concerns of the `Timer` entity (e.g., the `tick` method).
5. Responses to events. The different responses to the events triggered are implemented in the modules playing the `Observer` role (`Timer` and `Alarm`), by overriding the `update` method.

## *B. Aspect-Oriented Implementation*

In the previous example, the different concerns identified in the problem were not modularized separately using the Java object-orientated language. The different concerns of the Observer design pattern, such as event triggering and the distinct responses to these events, are tangled with the functionalities of the domain entities. In this section, we show how a higher level of modularization can be achieved by using AspectJ. We present the domain entities, an aspect implementing the Observer pattern, and an instantiation of this pattern for the particular scenario described in Section III.A.

Figure 4 shows how AOP provides the modularization of the domain entities (concern 1 in Section III.A), without including any of the concerns of the design pattern (concerns 2 to 5). In Figure 4, the `Timer` class has no concern related to the observer collection management (`observers`, `addObserver` and `removeObserver` in Figure 3) or to event notification (`updateObservers`); neither `Clock` nor `Alarm` describe their response to events (`update` in Figure 3); and roles are not specified by means of interface implementation. With AOP, each concern is placed in a separate module.

Figure 5 shows an implementation of the Observer design pattern using AspectJ. The `ObserverDesignPattern` aspect is declared as `abstract` (line 6), because it models a generalization of the Observer design pattern (not an instance of that pattern for a specific scenario). Later on, we will see how to use inheritance to apply this abstract aspect for a particular problem (Figure 6). Lines 8-24 in Figure 5 define the members that the aspect weaver will add to the classes derived from `Observer`: `observers`, `addObserver`, `removeObserver` and `updateObservers`. In this way, the `ObserverDesignPattern` aspect modularizes the observer collection management (concern 3) and the event notification (concern 4) concerns identified in Section III.A. These two concerns could have been implemented in two separate aspects, but we have merged them in the same aspect for the sake of simplicity.

Before describing the rest of the code in Figure 5, we define the concepts of join point, pointcut and advice used in AspectJ [4]. A *join point* is a point of execution in the control flow of a program. A join point specifies when, in the execution of a program, the aspect code should be executed. Example join points offered to the

```
01: import observer.aspectj.components.*;
02:
03: aspect TimerAspect extends ObserverDesignPattern {
04:   declare parents: Timer implements Subject;
05:   declare parents: Clock implements Observer;
06:   declare parents: Alarm implements Observer;
07:
08:   pointcut stateChanges(Subject subject):
09:       target(subject) &&
10:       call(void Timer.tick());
11:
12:   public void Clock.update() {
13:     this.incrementOneSecond();
14:     System.out.println(this);
15:   }
16:
17:   public void Alarm.update() {
18:     this.checkAlarm();
19:   }
20:
21:   private Timer = new Timer();
22:
23:   pointcut observerCreation(Observer observer) :
24:       execution(Observer+.new(..)) &&
25:       this(observer);
26:
27:   after(Observer observer) returning :
                         observerCreation(observer) {
28:     timer.addObserver(observer);
29:   }
30: }
```

Figure 6. Implementation of a particular aspect using the Observer design pattern.

```
package observer.aspectj.aspects;

public interface Subject {
    void addObserver(Observer observer);
    void removeObserver(Observer observer);
    void updateObservers();
}
```

```
package observer.aspectj.aspects;

public interface Observer {
    void update();
    void setSubject(Subject subject);
    Subject getSubject();
}
```

Figure 7. The two roles identified in the Observer design pattern.

AspectJ programmer are method invocation (`call`), method execution (`execution`), object creation (`new`), field access (`get` and `set`) and exception handling (`handler`). A *pointcut* is a set of join points specified with some syntax, commonly including regular expressions. Whenever the program execution reaches one of the join points described in a pointcut, a piece of code associated to that pointcut is executed. These pieces of Java code are called *advice*. An advice also indicates if its code should be executed `before`, `after` or instead of (`around`) the intercepted join point.

The AspectJ code in lines 26-30 (Figure 5) shows how events are notified (concern 4 in Section III.A). Line 26 defines the `stateChanges` pointcut. This pointcut is `abstract`, meaning that derived aspects will define the particular join points associated to this pointcut. Lines 28-30 implement the advice for event notification: the `updateObservers` method of the `Subject` must be invoked `after` executing the join points defined by the `stateChanges` pointcut.

The source code in Figure 5 is the implementation of an aspect representing a generalization of the Observer design pattern. Figure 6 shows how to use that aspect for the particular scenario described in Section III.A. In line 3, `TimerAspect` inherits from the `ObserverDesignPattern`. Lines 4-6 indicate the role that each domain entity will play in the Observer pattern (concern 2). The `Subject` and `Observer` interfaces (Figure 7) are implemented in the `observer.aspectj.aspects` package, as part of the aspect-oriented implementation of the design pattern. They are used as a mechanism to implement the pattern, but they are not included in the problem domain model. With the aspect-oriented approach, the domain entities (Figure 4), the two roles identified in the design pattern (Figure 7), and the assignment of the roles played by each entity (lines 4-6 in Figure 6) are implemented in different modules.

Lines 8-10 define the event that triggers the update of the `Observers` associated to a `Subject`. The abstract `stateChanges` pointcut is instantiated with a concrete join point: the `tick` method `call` of any `Timer` instance (concern 4). Lines 12-15 and 17-19 implement the event responses of the `Clock` and `Alarm` objects, respectively (concern 5).

The last part of Figure 6 (lines 21-29) shows how `Observers` are registered in the `Subjects`. In our example, one instance of the `Timer` class is created (line 21). All the instantiated `Observers` will be registered in that `Timer` upon creation. The `observerCreation` pointcut embodies those execution points representing the creation (`execution` of the constructor, i.e. `new`) of an instance derived from `Observer` (`+` indicates derived from). The advice in lines 27-29 registers all the `Observers` in the `Timer` instance created in line 21, `after` their construction (the `observerCreation` pointcut).

## IV. Discussion

The example presented in Section III illustrates how AOP provides a higher level of modularization, compared to object-orientation. AOP allows modularizing concerns that commonly cut across different abstractions in an object-oriented program. AOP provides meta-programming mechanisms to indicate how existing classes should be extended. These meta-programming services supported by the aspect weaver are the basis for providing a higher level of abstraction and modularity.

A discussion to be established about the use of AOP – and AspectJ in particular– is regarding the complexity derived from modularizing crosscutting concerns. New programming elements such as join point, pointcut and advice, plus a new language syntax, must be understood by the AspectJ programmer. The inherent complexity of these new programming elements has raised doubts about whether this additional complexity is worth the benefits obtained [13]. This complexity, together with the execution of the advice code associated to the intercepted join points, can make it difficult to debug aspect-oriented applications [14]. Another problem derived from join point interception is the conflict resolution of multiple advice intercepting the same join point [15]. Therefore, the impact of the aspect-oriented paradigm has not been as significant as initially expected [16]. However, its current use in enterprise applications (Spring), Web and application servers (OSGi), application frameworks (Spring Roo) and monitoring tools (Glassbox) is evident [17].

Currently, different dynamic programming languages support meta-programming features, improving their modularity capabilities. For example, introspection (the reflective inspection of program structure) allows implementing generic code to process any first-class entity (object, class, module…) regardless its type [18]. Intercession (dynamic modification of program structure) allows modifying the program entities at runtime [19], as done by the aspect weaver [20]. Dynamic code generation facilitates the extension and adaptation of a running application [21]. These meta-programming features supported by different languages provide a higher level of modularity than AOP [20].

However, most of the languages that provide the meta-programming features mentioned above are dynamically typed (e.g., Python, Ruby or JavaScript). These kind of languages commonly detect few type errors at compile time [22], and usually show a lower runtime performance [23]. Therefore, statically typed AOP tools such as AspectJ can be somehow considered as a balance between both approaches. They are not as flexible and expressive as reflective dynamic languages; but they provide a good level of modularity, with earlier type error detection and better runtime performance than dynamic languages.

## V. Conclusions and Future Work

Aspect-oriented programming provides a mechanism to improve the modularization of object-oriented applications. It avoids the code tangling and scattering problems caused by crosscutting concerns. By analyzing a typical design problem, we have seen how the different crosscutting concerns in a Java object-oriented implementation have been modularized with the use of AspectJ. In order to achieve this goal, new programming concepts such as join point, pointcut and advice are introduced. AspectJ also defines an extended syntax. However, these new elements may lead to a higher complexity in the implementation of applications. Compared to reflective dynamic languages, statically typed AOP tools represent a good trade-off between meta-programming capabilities, and compile-time type error detection and runtime performance.

We are currently working in the design of an aspect-oriented API for Java, which allows programmatic aspect weaving at runtime. The first benefit is providing aspect weaving without introducing a new syntax. It is symmetric, establishing no distinction between classes and aspects. Weaving is performed at runtime, making use of the JINDY library [24]. This library provides access to the new Java `invokedynamic` opcode from the Java language, obtaining a significant performance benefit compared with reflection [21]. We are also performing a performance evaluation of weaving [25], using the DSAW aspect weaving tool for .NET [26].

The source code of the two implementations used in this article are available for download at www.reflection.uniovi.es/dsaw/download/2014/iccet.zip

## Acknowledgment

This work has been funded by the Department of Science and Innovation (Spain) under the National Program for Research, Development and Innovation. Project TIN2011-25978, entitled *Obtaining Adaptable, Robust and Efficient Software by Including Structural Reflection in Statically Typed Programming Languages*.

## References

[1] B. Meyer, *Object-Oriented Software Construction*, 2nd edition, Prentice-Hall, ISBN 0-13-629155-4, 1997.

[2] D.L. Parnas, “On the criteria to be used in decomposing systems into modules,” *Communications of ACM*, vol. 15, issue 12, pp. 1053-1058, 1972.

[3] F. Ortin, B. Lopez, J. B. G. Perez-Schofield, “Separating Adaptable Persistence Attributes through Computational Reflection,” *IEEE Software*, vol. 21, issue 6, November 2004.

[4] W.L Hürsch, and C.V. Lopes, “Separation of concerns,” Technical Report NU-CCS-95-03, Northeastern University, Boston, February 1995.

[5] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin, “Aspect-Oriented Programming,” in *European Conference on Object-Oriented Programming* (ECOOP), Jyväskylä, Finland, pp. 220-242, June 1997.

[6] G. Kiczales, “The fun has just begun,” in *Aspect-Oriented Software Development* (AOSD), Boston, Massachusetts, March 2003.

[7] The Eclipse Foundation, “AspectJ, crosscutting objects for better modularity,” http://www.eclipse.org/aspectj, November 2013.

[8] Aspect-Oriented Software Association, “Aspect-Oriented Software Development,” http://www.aosd.net, November 2013.

[9] H. Ossher, and P. Tarra, “Hyper/J: Multi-dimensional separation of concerns for Java,” in *International Conference in Software Engineering* (ICSE), Toronto, Canada, pp. 821-822, May 2001.

[10] F. Ortin, L. Vinuesa, and J.M. Felix, “The DSAW aspect-oriented software development platform,” *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, issue 7, pp. 891-929, November 2011.

[11] L. Vinuesa, F. Ortin, J.M. Felix, and F. Alvarez, “DSAW - A Dynamic and Static Aspect Weaving Platform,” in *International Conference on Software and Data*

*Technologies* (ICSOFT), Porto, Portugal, pp. 55-62, July 2008.
[12] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Professional, ISBN 0-20-163361-2, 1994.
[13] F. Steimann, "The paradoxical success of aspect-oriented programming," in ACM SIGPLAN Conference on *Object-Oriented Programming Systems, Languages and Applications* (OOPSLA), Portland, Oregon, pp. 481-497 October 2006.
[14] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger, "Debugging aspect-enabled programs," in *International Conference on Software Composition*, Braga, Portugal, pp. 200-215, March 2007.
[15] R. Duoence, P. Fradet, and M. Südholt, "Detection and resolution of aspect interactions," Technical Report No. 4435, INRIA, France, ISSN 0249-6399, 2002.
[16] A. Popovici, "The impact of aspect-oriented programming on future application design," Information and Communication Research Group Seminar, ETH Zurich, January 2001.
[17] R. Laddad, "A real-world perspective of AOP," *Transactions on Aspect-Oriented Software Development* vol. VIII, pp. 108-115, 2011.
[18] F. Ortin, M. Garcia, J.M. Redondo, and J. Quiroga, "Achieving multiple dispatch in hybrid statically and dynamically typed languages," *Advances in Intelligent Systems and Computing*, vol. 206, pp. 703-713, 2013.
[19] F. Ortin, D. Diez, "Designing an Adaptable Heterogeneous Abstract Machine by means of Reflection," *Information and Software Technology*, vol. 47, issue 2, pp. 81-94, February 2005.
[20] F. Ortin, J.M. Cueva, "Dynamic Adaptation of Application Aspects," *Journal of Systems and Software*, vol. 71, issue 3, pp. 229-243, May 2004.
[21] F. Ortin, P. Conde, D. Fernandez-Lanvin, R. Izquierdo, "Runtime Performance of invokedynamic: Evaluation through a Java Library," *IEEE Software*, in press, doi.ieeecomputersociety.org/10.1109/MS.2013.46, April 2013.
[22] F. Ortin, M. Garcia, "Union and intersection types to support both dynamic and static typing," *Information Processing Letters*, vol. 111, issue 6, pp. 278-286, February 2011.
[23] F. Ortin, J.M. Redondo, J.B.G. Perez-Schofield, "Efficient virtual machine support of runtime structural reflection," *Science of Computer Programming*, vol. 74, issue 10, pp. 836-860, August 2009.
[24] P. Conde, F. Ortin, "Jindy: a Java library to support invokedynamic," *Computer Science and Information Systems*, in press, October 2013.
[25] M Garcia, F. Ortin, D. Llewellyn-Jones, and Madjid Merabti, "Performance cost evaluation of aspect weaving," in *Australasian Computer Science Conference* (ACSC), Adelaide, Australia, pp. 79-86, February 2013.
[26] M. Garcia, D. Llewellyn-Jones, F. Ortin, M. Merabti, "Applying dynamic separation of aspects to distributed systems security: a case study," *IET Software*, vol. 6, issue 3, pp. 231-248, June 2012.

**Jose Manuel Felix**, 1972, is a part-time PhD student that works as a civil servant for the Computer Science Department of the Spanish Principality of Asturias. He received his BSc degree in Computer Science in 1994. In 1998 he was awarded an MSc in Computer Engineering, and an MSc in Web Engineering in 2008. His PhD thesis is focused on defining, DSAW, an aspect-oriented platform for .NET that supports both dynamic and static weaving with important runtime performance optimizations.

**Francisco Ortin**, 1973, is an Associate Professor of the Computer Science Department at the University of Oviedo, Spain. He is the head of the Computational Reflection research group (http://www.reflection.uniovi.es). He received his BSc in Computer Science in 1994, and his MSc in Computer Engineering in 1996. In 2002 he was awarded his PhD entitled *A Flexible Programming Computational System developed over a Non-Restrictive Reflective Abstract Machine*. He has been the principal investigator of different research projects funded by Microsoft Research and the Spanish Department of Science and Innovation. His main research interests include dynamic languages, type systems, aspect-oriented programming, computational reflection, and runtime adaptable applications. Contact him at http://www.di.uniovi.es/~ortin