

On Aspect-Oriented Technology and Object-Oriented Design Patterns

Ouafa Hachani and Daniel Bardou

Equipe SIGMA, LSR-IMAG
BP 72

38402 Saint Martin d'Hères Cedex, France
{Ouafa.Hachani, Daniel.Bardou}@imag.fr

Abstract. More and more works are done on design patterns and aspect-oriented programming. These works mainly propose to use aspect-oriented programming mechanisms to provide new implementations of object-oriented design patterns. This paper illustrates our own approach by presenting an aspect-oriented implementation of the GoF *Strategy* pattern, and claims implementation is not sufficient: aspect-orientation also needs to be considered at the design level in the description of design patterns. We also discuss different issues relative to the “aspectization” of object-oriented design patterns.

1 Introduction

More and more works are done on object-oriented design patterns [2], [3], [4]¹ and aspect-oriented programming [12]. The motivation of some of these works, such as [13], [18], is to establish how design patterns (*Visitor* in particular) can help in achieving aspect-oriented programming. We will not much consider such approaches in this paper but rather concentrate on the works related to provide aspect-oriented implementations of object-oriented design patterns such as [8], [9], [14], [15], [16] (see 4.1 for a bit more details on these works). These latter works all start from a quite simple basic report that we develop below.

The GoF design patterns were proposed to solve common problems in object-oriented design, and they were therefore described in terms of structures and solutions that are of course directly related to object-oriented programming techniques and mechanisms. While not refuting the usefulness of these patterns, they can be reconsidered by the advent of the more recently proposed aspect-oriented languages for at least the two following reasons.

- Design patterns are generally defined as collaborations between several objects (classes): they are thus a case of code scattering. Code scattering is one of the two main problems that can be solved by advanced separation of concerns [10] and aspect-oriented programming has been introduced in that purpose [12].
- Most of the currently proposed aspect-oriented programming languages build up on object-oriented programming languages. Aspect-oriented mechanisms should thus allow for new program designs that are out of reach of strict object-orientation and could possibly improve the structures and implementations that were initially proposed in design patterns.

The rest of this paper is organized as follows. Section 2 summarizes our approach and illustrates it with the example of the *Strategy* pattern. Section 3 provides some details on related approaches and contains a discussion on some open issues relative to the translation of object-oriented design patterns into aspect-oriented ones. The contribution of this paper mainly resides in this discussion that could be continued during the workshop.

2 Aspect-Oriented Implementation of Design Patterns

We briefly summarize our work on providing aspect-oriented solutions for the object-oriented design patterns [5], [6], [7]. We started from the identification of four problems raised by the use of the GoF patterns in their initially proposed implementations: confusion, indirection, encapsulation breaching and inheritance related problems. We further noticed that these four problems were special cases of the two problems of “code scattering” and “code tangling” that are addressed by advanced separation of concerns [10] and aspect-orientation [12]. We thus proposed AspectJ [11], [20] implementations for the 23 GoF patterns and showed how these implementations solve or avoid these problems. We tried to not systematically mimic the original object-oriented structure, but we rather were mainly inspired by the patterns intents. One of our main motivations in this work was to improve the traceability of design patterns by being able to localize them in the code, and

¹ For seek of simplicity we will only focus in the rest of this paper on the Gang-of-Four (GoF) design patterns [4], as they are well-known. This however does not put out of consideration other sets of object-oriented design patterns.

consequently improve the code readability, reusability, maintenance, and evolution. We consider below the GoF *Strategy* pattern to illustrate our approach in this section.

2.1 AspectJ Implementation of *Strategy*

The intent of *Strategy* is to define a family of interchangeable encapsulated algorithms [4]. In other words, it allows giving different interchangeable definitions of a method for the instances of the same class. Fig. 1 shows the object-oriented structure of this pattern. Each sub-class of the **Strategy** abstract class holds a different definition of a method named `algorithmInterface()` that is supposed to be applied on instances of **Context**.

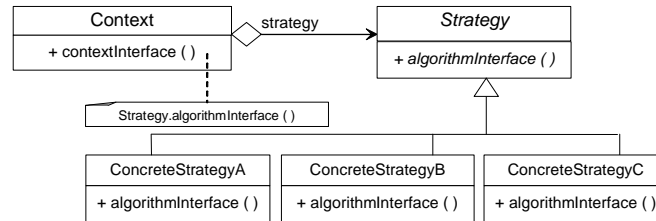


Fig. 1. UML class diagram of the *Strategy* structure (adapted from [4]).

Strategy's intent is to provide a polymorphic behavior to the instances of a given class. We propose to gather the various definitions of this behavior in one aspect for each instance of *Strategy*. The selection of the adequate method can then be ensured by using an advice catching calls to `contextInterface()`. Fig. 2 shows the aspect-oriented structure that we propose for *Strategy*, and fig. 3 gives the outline of the corresponding AspectJ code. The `strategy` field is used for the internal representation of the actual strategy chosen for an object. This field is statically introduced by the **Strategy** aspect, the same way the default strategy is set.

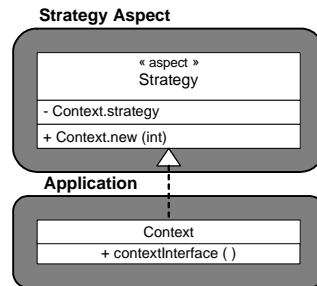


Fig. 2. Aspect-oriented structure of *Strategy*²

```

privileged aspect Strategy pertarget ( target (Context) && call (void Context.contextInterface ()) ) {
  private int Context.strategy = 0;
  public Context.new(int strategy) {
    this.strategy=strategy; }
  public void defaultStrategy() { /* definition of the default strategy' */ }
  void around (Context c : target (c) && execution (void Context.contextInterface ()) ) {
    switch (c.strategy) {
      case 1: // code to be executed for the strategy 1
        break;
      case 2: // code to be executed for the strategy 2
        break;
      default: defaultStrategy();
    }
  }
}
  
```

Fig. 3. Outline of the AspectJ code for the **Strategy** aspect

² We use here the UML extension proposed by [19] in which an implementation relationship (a dashed arrow with a triangular arrowhead) connecting a class to an aspect means that this class is impacted by this aspect.

2.2 Towards Aspect-Oriented Descriptions of GoF Patterns

In order to fully preserve the benefits of design patterns, it is necessary to not only consider aspect-oriented implementation, but also care of the evolution of their description. The GoF patterns are all described in the same format: each pattern description is organized in 14 sections [4]. As we tried to respect the intent and applicability of the original patterns, we see no reason to modify the corresponding sections. What is the case for the two sections “Name” and “Classification”. It is also obvious that the AspectJ implementations that we propose will be described in the “Structure” and “Sample Code” sections. We briefly explain below how other sections can be modified to provide new descriptions of the GoF patterns.

- **Participants:** we generally propose structures in which only one aspect is used to indicate the pattern as a whole, which usually results in a smaller number of participants.
- **Collaborations:** in some patterns, the use of AspectJ introductions [11] does not involve any collaboration. In most of the other patterns, communications are reduced and the collaboration protocol is localized into the aspect denoting the pattern.
- **Implementation:** there are often several possible alternatives for structuring patterns in AspectJ: for example we discuss some alternatives of the *Visitor* aspect-oriented structure in [7]. As more aspect-oriented implementations of patterns will be proposed in different languages (and not only AspectJ), this section will be more significant.
- **Consequences:** this section does not only describe the way a pattern meets his objectives, but it is also strongly related to the “Implementation” section since it discusses the impact of various alternative structures on the application.

Table 1 shows how the GoF description of *Strategy* can be modified. We only outline the sections that are actually modified (compared to the original description in [4]).

Table 1. Outline of an aspect-oriented description of *Strategy*

Structure	see fig. 2 p.3
Participants	<p>Context class: different strategies will be applicable on its instances.</p> <p>Strategy aspect:</p> <ul style="list-style-type: none"> - is based on a join point: call of <code>contextInterface()</code>; - gathers the various definitions of the algorithm; - introduces a strategy field used for the internal representation of the strategy associated to an instance of Context; - is responsible for the default strategy setting; - controls with an around advice the execution of <code>contextInterface()</code>; this control is also based on the strategy field.
Collaboration	<ul style="list-style-type: none"> - Strategy and Context interact to perform the chosen algorithm, with no communication. Calls to <code>contextInterface()</code> are transparently caught by Strategy which controls the adequate executions. - Clients interact with Context exclusively. They may choose a strategy when creating new Context's instances.
Implementation	<p>Consider the following implementation issues.</p> <ul style="list-style-type: none"> - If there exist some algorithm parts that are common to several strategies, intermediate methods may be defined in Strategy. ... - Strategies may be dynamically changed by just adding an introduction of a <code>setStrategy()</code> method into Context. - ...
Consequences	The proposed structure requires using a conditional statement for selecting the adequate algorithm. This makes Strategy somehow difficult to maintain, especially if it supports numerous algorithms.

2.3 Benefits

The main advantage of our proposed aspect-oriented implementations of the GoF patterns is to easily localize any instance of pattern in the code. This improves readability, traceability, adaptability and evolution of both the code relative to the pattern and the one relative to the classes on which it is applied. This also significantly reduces the number of participants involved in a pattern.

Aspect-oriented structures also include for eight of the GoF patterns one abstract aspect that can be inherited by aspects denoting several instances of the same kind of pattern: some pattern-specific code can thus be shared between several pattern instances.

Finally several code-level dependencies (generally introduced by indirection or inheritance) are removed. The number of communications is consequently also reduced.

3 Discussion

This section is a discussion on more or less open issues relative to the transformation of object-oriented design patterns into aspect-oriented ones. As we want this discussion to apply not only to our work but also encompass related approaches, we briefly cite them in preamble. The rest of this discussion is organized as a list of questions that we try to answer.

3.1 Related Works

Here are briefly summarized some approaches aiming to propose aspect-oriented implementations for object-oriented design patterns.

- [14] proposes AspectJ and HyperJ implementations for 3 of the 23 GoF patterns (*Observer*, *Composite* and *Adapter*). Creating an aspect instead of any class that would have been introduced in an object-oriented implementation is suggested. Additional matching aspects are then defined to establish relationships between those aspects and application classes. Although this transformation scheme is rather simple to understand and apply, an important drawback is that it introduces numerous aspects.
- Nordberg presents an interesting discussion on dependencies in [15] and [16] and shows how they can be reduced, or even better directed, by adopting an AspectJ design for 12 patterns.
- [8] is certainly the approach that is the most related to our work. It also considers the 23 GoF patterns and proposes AspectJ implementations of them. Those implementations are also related to a classification of the patterns in which the following criteria are retained: locality, reusability, composition transparency, (un)pluggability, and the kinds of roles involved. Our proposed implementations however slightly differ from those of [8], partly because we care of ending with an aspect denoting each instance of a pattern in the code in order to improve traceability.
- [9] is to our knowledge the first work proposing AspectS implementations of some of the GoF patterns. This work suggests it will soon be possible to provide new aspect-oriented descriptions of the GoF patterns including sample implementations in different languages and not only AspectJ.

The brief comparison of all these approaches is continued in the questions listed below.

3.2 Why Is it Useful? Could it Prove More Useful?

A remark that can be made when comparing these different approaches is that the benefits of using aspect-oriented designs and implementations are not presented in the same way by their authors. One can simply put forward the ability of applying object-oriented expertise to aspect-oriented software development, or discuss further on the reduction of dependencies, the improvement of reusability, modularity and maintainability. We mainly retain the ability to explicitly denote and thus localize patterns in the code to be very important, as design patterns mining is not easy [1], [17].³ This not only allows for patterns traceability but also open new research directions for metrics on patterns (from the simplest one that could be the count of each pattern instances, to more elaborated ones assessing the joint use of several patterns on some classes).

³ Note that having the instance of patterns explicitly identified in the code does not make obsolete those works: they even could be adapted to provide automatic tools for translating existing object-oriented code into aspect-oriented one.

3.3 What Kind of Decomposition?

Designing aspect-oriented solutions amounts to choose how to decompose the code into classes and aspects. There are several possibilities for patterns implementation and the best design is certainly related to the main motivation of the approach. As we want to improve traceability of patterns, we care of having exactly one aspect to denote a pattern instance in the code, while keeping some abstract aspects in order to improve reusability. One may also choose to group all instances of the same pattern into one aspect for better modularity, or to design aspects to denote only functional concerns of the application and thus involving several pattern instances of different kinds. One may even think of organizing aspects into a layered architecture. There is however a couple of important remarks that can be made since aspects are used to implement design patterns.

- This is a case of using aspects for the separation of functional concerns and it differs somehow from numerous works on aspect-orientation (including the initial proposal [12]) that were mainly directed to deal with non-functional concerns.
- As pattern instances are also small parts of the whole design of an application, it also has for consequence to likely end up with numerous thin aspects (crosscutting much fewer classes than an aspect denoting a non-functional concern that may impacts the whole set of classes).

Having thinner aspects amounts more thinking to compose them into an application, and suggests we should put more efforts on studying aspect-level relationships that can be considered in the design. Aspect inheritance of aspects is already supported by some aspect-oriented languages, but is it really comparable to class inheritance at the design level: can we speak of aspect generalization? And would it make sense to think of aspect associations, aggregations and compositions? We think one may not give trivial answers to these last questions as aspects are not supposed to denote entities of the application domain, just like objects do.

3.4 Do some Patterns Disappear?

One may think some of the object-oriented design patterns (*Visitor* for example) just disappear when aspect-oriented programming mechanisms are used, because their implementation becomes trivial. We do not agree with such a statement, not only because we want to improve patterns traceability and such keep a trace of any pattern instance, but also because we think that expertise capitalization is probably the most important benefit of patterns. As we can consider the object-oriented design patterns to be today familiar to most of the software architects, we think they come easily to their minds when they care about the design of an application. The single name of a design pattern is also generally sufficient to resume some part of the design that could be quite large. For these reasons, we think the name and description of the patterns should always subsist.

3.5 Language Specific or not Language Specific

Most of the works cited in this paper use AspectJ as primary aspect-oriented language. It is however important to end with new aspect-oriented descriptions of design patterns that are not language specific. We began our work by using only AspectJ, and are now trying to consider other aspect-oriented languages. The translation from AspectJ to AspectS is rather obvious as the basics of these two languages are close to each other. The translation from AspectJ to HyperJ is more difficult in the other hand. This might be a point against the process of trying to translate object-oriented design patterns into aspect-oriented ones, or this might suggest that more work must be done on the definition of the common minimal basics and requirements of aspect-oriented languages.

3.6 Are We Going to Find “Real” Aspect-Oriented Design Patterns?

Directly related to the previous question, one can also note that all the works cited in this paper have been only considering object-oriented design patterns so far. It is not very amazing that aspect-oriented programming mechanisms that have been introduced with the hope to provide better programming techniques than the previously existing object-oriented ones can improve object-oriented design patterns. Can we however claim that we are considering aspect-oriented design patterns then? Object-oriented design patterns have been proposed to identify and collect solutions to recurring problems in object-oriented design: “real” aspect-oriented design patterns should then deal with the limitations of aspect-orientation rather than those of object-orientation. Finding new patterns directly related to aspect-orientation still an important research direction.

4 Conclusion

We have considered in this paper several recent works aiming to propose new aspect-oriented implementations for design patterns that were originally described in the context of strict object-orientation. We briefly presented our own approach and we have claimed that not only implementations but also descriptions should be adapted in an aspect-oriented fashion. We have finally listed some interesting open questions that are relevant to all of the related approaches. Those questions could serve as a basis of an actual discussion between the attendees of the workshop.

References

1. Antoniol, G., Fiutem, R., Cristoforetti, L.: Using Metrics to Identify Design Patterns in Object-Oriented Software. In: Proceedings of the Fifth International Symposium on Software Metrics (METRICS '98). IEEE Computer Society (1998) 23–34
2. Buschman, F.: What is a pattern?. In: Object Expert, vol. 1, n°3 (1996) 17–18
3. Gamma, E.: Object-Oriented Software Developments based on ET++: Design Patterns, Class Library, Tools, PhD thesis, University of Zürich (1991)
4. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley (1995)
5. Hachani, O.: Utilisation de la programmation par aspects dans l'implémentation de patrons de conception. Mémoire de DEA, Université Grenoble 1 (2002)
6. Hachani, O., Bardou, D.: Using Aspect-Oriented Programming for Design Patterns Implementation. In: OOIS 2002 Workshop on Reuse in Object-Oriented Information Systems Design (2002)
7. Hachani, O., Bardou, D.: « Aspectisation » de patrons de conception – Exemples de transformation des patrons *Observateur*, *Visiteur* et *Stratégie*. In: Actes du congrès INFORSID 2003, Hermès Sciences (2003)
8. Hannemann, J., Kiczales, G.: Design Pattern Implementation in Java and AspectJ. In: Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA 2002). SIGPLAN Notices, Vol. 37, N°11, ACM (2002) 161–173
9. Hirschfeld, R., Lämmel, R., Wagner, M.: Design Patterns and Aspects – Modular Designs with Seamless Run-Time Integration. In: Proceedings of the 3rd German Workshop on Aspect-Oriented Software Development (AOSD-GI 2003) (2003)
10. Hürsh, W., Lopes, C.: Separation of Concerns. Technical report NU-CCS-95-03, College of Computer Science, Northeastern University (1995)
11. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Proceedings of ECOOP 2001, Lecture Notes in Computer Science, Vol. 2072, Springer (2001) 327–353
12. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J.: Aspect-Oriented Programming. In: Proceedings of ECOOP'97, Lecture Notes in Computer Science, Vol. 1241, Springer (1997) 220–242
13. Lorenz, D.H.: Visitor Beans: An Aspect-Oriented Pattern. In: ECOOP'98 Workshop on Aspect-Oriented Programming (1998)
14. Noda, N., Kishi, T.: Implementing Design Patterns Using Advanced Separation of Concerns. In: OOPSLA 2001 Workshop on AsoC in OOS (2001)
15. Nordberg, M.E.: Aspect-Oriented Dependency Inversion. In: OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems (2001)
16. Nordberg, M.E.: Aspect-Oriented Indirection – Beyond Object-Oriented Design Patterns. In: OOPSLA 2001 Workshop “Beyond Design: Patterns (mis)used” (2001)
17. Paakki, J., Karhinen, A., Gustafsson, J., Nenonen, L., Verkamo, I.: Software Metrics by Architectural Pattern Mining. In: Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress) (2000) 325–332
18. Parigot, D., Courbis, C., Degenne, P., Fau, A., Pasquier, A., Fillon, J., Held, C., Attali, I.: Aspect and XML-oriented Semantic Framework Generator: SmartTools. In van den Brand, M., Lämmel, R. (eds.): Proceedings of the second workshop on Language Descriptions, Tools and Applications (LDTA '02), satellite event of ETAPS 2002. Electronic Notes in Theoretical Computer Science, Vol. 65, N°3, Elsevier Science Publishers (2002)
19. Suzuki, J., Yamamoto, Y.: Extending UML with Aspects: Aspect Support in the Design Phase. In: ECOOP'99 Workshop on Aspect-Oriented Programming (1999)
20. AspectJ Web Site: <http://www.aspectj.org/>