Improving Design Pattern Quality Using Aspect Orientation

Mario Luca Bernardi, Giuseppe Antonio Di Lucca
dilucca/mlbernar@unisannio.it
RCOST Research Centre on Software Technology
University of Sannio

Palazzo ex Poste, via Traiano, 82100 Benevento, Italy

ABSTRACT
Object Oriented (OO) implementations of Design Patterns (DP) may suffer of some problems due to deficiencies of (OO) languages affecting some quality attributes such as modularity, comprehensibility, maintainability and testability. Aspect Oriented Programming (AOP) provides powerful constructs able to better handle modularity and composition; these constructs can help to overcome some of the trade-offs in current OO implementations of DPs. An approach to re-implement DP’s by AOP is presented in this paper: some different AOP re-implementations of DPs in existing systems from real world have been performed to improve DP’s quality. A set of existing metrics has been used to evaluate the quality of the different AOP implementations. Such an evaluation helps to select the implementation to use/reuse in a specified context.

KEY WORDS
Software Engineering, Design Patterns, Aspect Oriented Software Development, Software Metrics

1. Introduction

A Design Pattern (DP) models a generic solution to a recurring design problem; a catalogue of Object Oriented DPs is provided in [1]. Object Oriented DPs are more and more used in the development of software systems, thus the way a DP is implemented may heavily affect the overall quality of a system by impacting the system structure. Indeed, the implementation of a DP, usually, is ‘invasive’: the code to implement a DP is scattered among the modules as well as the modules involved in a DP implementation have tangled code dealing with more than one concern. Thus it may be hard to identify which are the modules where the code implementing a DP actually is as well as to distinguish, inside a module, between the code of pattern instances and the code of the other system concerns.

The consequences on system quality attributes, such as the modularity and hence the comprehensibility, maintainability, testability are obvious. The implementation of concerns spreading across several modules is known as “Cross-Cutting Concerns”, because it cuts module boundaries and is fragmented across a number of modules. In [2, 4, 5] it is shown how several patterns from GoF catalog [1] introduce Cross-Cutting Concerns (CCC) that OO abstractions are often unable to well modularize. That is mainly due to some poorness of the composition and quantification constructs in OOP languages that do not allow a good modularization of the concerns. Indeed, by using OOP programmers are forced to mix the DP code (e.g., classes, interfaces, methods and attributes) with the code of the modules (e.g., packages, classes, etc.) resulting from the system dominant decomposition. All that causes code scattering and tangling by reducing the quality of some DP attributes. Aspect Oriented Programming (AOP) provide patterns’ developers with powerful quantification constructs to better modularize the DP concerns. In [2, 4, 6] AOP implementations of GoF patterns are provided; they have better values for attributes such as locality, (un)pluggability, composability and reusability that contribute to improve the modularity of the system. Moreover AOP implementation of the DPs allows to improve the composition transparency and optionality attributes that also affects the modularity. Thus, a way to improve the modularity of DPs implemented by OOP is to re-implement them by AOP techniques. AOP aims at solving these problems by new ways for CCC modularization: AOP languages provide new modularization units and more powerful referential, quantification and composition mechanisms than OOP languages. This issue is dealt in [5] where it is shown how AOP can help in restructuring GoF patterns to get more effective anticipation of future changes with a lower overhead. However, an OO DP may be re-implemented by AOP in several ways, each one differently affecting the quality of the AOP implementation.
adopted. The attributes of locality, (un)pluggability, composability, reusability, transparency and optionality, as well as the obliviousness (defined as a way to decouple the dominant decomposition of the base system from the other aspects), mainly affect the quality of aspect oriented designs and implementations [2, 3, 4, 13]. However, research in this field is not yet mature, and usually it has been related to a reduced set of patterns. A deeper investigation and discussion is required about these issues. In particular investigation is needed to understand how these attributes are related to coupling and cohesion.

In this paper we propose an approach to analyze OO implementation of DPs in existing systems to identify code scattering and tangling and to re-implement them with AOP to improve their quality. We have analyzed some DP implementations from different domains (structural decomposition, data access, communication, management and access-control patterns) selecting them from a DP set wider than the GoF catalog. After having identified the main issues generating code scattering and tangling, some different AOP re-implementations of DPs were performed. The quality of each implementation has been evaluated by a set of existing metrics well known in the literature. This evaluation can be used to drive the design of improved DPs by aspect-oriented implementations. A first experiment involving some DPs selected in existing systems ‘from real-world’ has been carried out: a comparison of the computed metrics for the different AOP implementations of DPs allowed to identify which ones showed the best quality characteristics. The paper is structured as follow: Section 2 discusses about the main characteristics of AOP; in Section 3 the overall approach is presented together with the set of metrics used to evaluate the different DPs’ AOP re-implementations; Section 4 presents and discusses the case study we carried out. Conclusive remarks and future work are finally presented in Section 5.

2. AOP main characteristics

The design of a software system (both object and function oriented) is usually thought according to a dominant system decomposition. This decomposition typically includes the modules implementing the concerns related to the application domain entities having the responsibilities of the business rules. Other concerns related both to functional and nonfunctional requirements (such as usability, reliability, security, distribution, persistence, concurrency and so on) cause more non-dominant decompositions interleaving with the dominant one. While the concerns included in the dominant decomposition can be usually encapsulated into well defined modules, it could be difficult to encapsulate into a specific module all the code implementing the concerns not belonging to the dominant decomposition. Thus their implementations can be scattered across more modules or tangled together in a single module. This is mainly because such concerns can be related to different aspects of a certain requirement or functionality. As a consequence the modularity of the system is negatively affected and its maintenance could be become very complex. Aspect Oriented Software Development (AOSD) provides methodologies, principles and techniques to identify, to modularise, to represent and to compose the concerns that cannot be effectively modularised using ‘traditional’ current software development techniques, such as the OO development. AOSD allows to avoid, or at least to reduce, CCC and code tangling. Most of the work in AOSD has been focused on the development of Aspect-Oriented Programming (AOP) languages, platforms and frameworks. AOP aims at providing ways to modularize CCC: AOP languages provide new modularization units and more powerful referential, quantification and composition mechanisms than OOP languages. With the “AOPWeaver” component, AOP environments provide new composition capabilities that open up the possibility to design and to implement a system keeping CCC well modularized: the aspect weaver has in charge the burden of composing aspect logic into the base system.

Two important AOP constructs are introductions and pointcuts. Introductions allow to modify the static structure of source code: for example, using the introductions the developers can change inheritance hierarchy, implementation
and extension relationships, insert members into classes. Pointcuts are a way to refer to “Join points” in programs (i.e., points in the system where some additional behaviour can be added; they contribute to define the structure of the aspects). With pointcuts/join-points a developer can refer particular points in the program control flow that can be intercepted during the execution. This allows, for example, capturing the creation of objects, the execution of a method and other events along with the context in which such events take place. Figure 1 shows a simple example of AOP code including pointcuts and introductions: the code defines an aspect HashablePoint that inject into class Point (not showed in the figure) two methods (hashcode() and equals()) to make it "hashable". Moreover, an abstract pointcut (to be defined in subaspects of the aspect hierarchy) and a pointcut capturing creation of objects of class MyClass are declared.

3. Approach overview

The proposed approach to improve the quality of OOP DPs re-implementing them by AOP consists of a stepped process driving the software engineer from the identification of CCC in the DPs to the evaluation of different AOP implementations among which to select the most suitable one for a specific context.

The process consists of the main following four steps:

1. static source code analysis of the existing system to identify code scattering and tangling (i.e., CCC) due to the implementation of DPs;
2. CCC analysis to identify the issues triggering the CCC;
3. different AOP re-implementations of the DPs affected by CCC;
4. evaluation of the quality of the different AOP implementations of a same DP by a set of metrics.

Table 1. The adopted metric set

<table>
<thead>
<tr>
<th>Metric</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>WOM</td>
<td>Number of weighted operation in a module</td>
</tr>
<tr>
<td>DIT</td>
<td>Length of the longest path from a given module to the deployment hierarchy root</td>
</tr>
<tr>
<td>RFM</td>
<td>Number of methods and advices potentially executed in response to a message received by a given module</td>
</tr>
</tbody>
</table>

Figure 2 reports the structure of this process.

In the first step, static source code analysis is performed to recover the system architecture and to extract notable information to: (i) identify and isolate the DP implementations in the system; (ii) identify the CCC due to the implementation of each DP (i.e., identification of the modules affected by the CCC). The identification of the CCC is carried out by using a lexical matcher to identify the code implementing several secondary concerns with respect to specified implementation patterns. Tools such as Aspect Browser [12], Aspect Mining Tool [13], or FEAT [14] can be used to automatically support this step, otherwise the step is carried out ‘manually’ by a ‘human’ expert reading the code. In the second step, the results of the CCC identification are analysed to recognised the reasons generating the CCC. At this aim the code is furtherly (manually) inspected along with the eventual existing documentation. The typical main issues in OO DP implementation generating CCC are:

1. Pattern overlapping: this is when concrete classes participate to more than one pattern instance. This reduce pattern traceability and concrete class reusability.
2. Inheritance relationships: this is when inheritance is used to assign roles to pattern participants. Because inheritance relationship is static, developers are forced to define pattern logic and possible participants to patterns ahead of time. Thus, concrete classes contain pattern code resulting to be less reusable and understandable.
3. Encapsulation violation: this is when patterns force to expose the internal object’s state to other objects to get
the flexibility needed to handle requested computation. Notable examples are "Composite" and "Visitor" patterns.

4. **Object Oriented Indirection**: Indirection superimposes secondary roles (from pattern logic) to concrete classes that already have a well defined role in the system. This introduce 'smells' like 'multiple personality' [15] increasing CCC.

5. **Secondary concerns scattered over pattern classes**: even when DPs do not introduce CCC directly, they can contribute to shape the dominant decomposition of the base system. This means that secondary concerns will be scattered across the patterns implementation.

These issues are well known and discussed in the literature. Effective AOP solutions have been proposed for those ones at points 1), 4) and 5) [5],[10], [15], [4]. The issue at point 3) has not yet been deeply addressed, anyway AOP does not seems to be, at now, very effective to solve the problems related to Encapsulation violation. The result of the second step is a list reporting which of the previous issues affect each DP. In the third step, DPs affected by CCC are re-implemented by AOP.

AOP introduces a new kind of implicit coupling due to the aspect interception mechanisms at run-time. This implicit coupling can cause subtle misbehaviour due to assumptions that the aspectual code components make on the non aspectual ones. Moreover, since aspects may depend on, and interact with, all system components, it is very easy to use AOP in a poor uncohesive way.

We performed different AOP re-implementations of DPs to experiment and to evaluate the effects of different AOP design strategies on coupling and cohesion. In particular, the following three different kinds of AOP implementations have been considered:

- a 'Lazy' (L) AOP implementation: this just aims to get a better modularization of the scattered code in a simple way. It uses inter-type declaration to move to aspects all pattern-related members (method and fields) and re-inject them at their original places at load time. This approach allows a quick migration to AOP but with a low quality level. Indeed, the code is modularized just at source code level but at linking/loading/run time the structure of the system is unchanged with respect to the old OOP implementation. This 'Lazy' AOP implementation is not (un)-pluggable and the reusability degree is low.

- an (un)pluggable (U) AOP implementation: the code of DPs’ logic was encapsulated into aspects; however this implementation may suffer of a low reusability.

- a reusable (R) AOP implementation: this represents an enhancement of the U implementation where a two aspect-based layered structure and a larger use of dynamic properties of AOP are used to gain greater (un)pluggability and reusability.

In the fourth step, each AOP implementation is evaluated with respect a set of metrics. Being implicit coupling and poor cohesion a critical issue for AO systems, a set of existing metrics, whose effectiveness has been proven in the literature, to evaluate the coupling and the cohesion of the AOP components was accurately selected. Some more metrics to evaluate the size and complexity of AOP components were considered too. The following metrics are, then, used to evaluate each AOP re-implementation of the DPs affected by CCC:

- Weighted Operations in Module (WOM)
- Depth of Inheritance Tree (DIT)
- Coupling on Intercepted Modules (CIM)
- Coupling on Method Call (CMC)
- Coupling on Field Access (CFA)
- Coupling Between Modules (CBM)
- Lack of Cohesion in Operations (LCO)
- Response For A Module (RFM)

A short description of these metrics is in Table 1. The metrics were selected among the ones proposed in [10], as a redefinition of the OO Chidamber and Kemerer metrics[11] tailored for AOP.

The AOP implementations are compared with respect to these metrics; this can help to select which implementation is better to use/reuse in a specified context.

4. **Case Study**

The case study involved two systems from the real world. The first system is the **Antenna Group Server communication system (AGS)**, a commercial system implementing a server to drive an antenna system. AGS was developed by using C and C++ languages, its size is about 20KLOC and it includes 37 classes. It receives control and status messages from remote devices (like control panels, consoles, environmental sensors), performs checks and issues commands to an advanced non-linear motion controller for an antenna group. The development of the system was based on a three-layer architecture: a Protocol Logic Layer, an Application Layer (grouping the software units related to Antenna and Motion servers) and a Physical Communication Layer. The system has to address several concerns like...
tracing, logging, debugging, message handling and protocol logic enforcement. It was designed and implemented using some well known DPs from several domains (GoF, communication, architectural, etc.). The following DP implementations were found in the system:

- Forwarder/Receiver (5 instances)
- Observer (3 instances)
- Command
- State
- Singleton (2 instances)
- Chain of Responsibility

In the following, due to space constraints, we will focus the discussion just on Forwarder/Receiver and Observer DPs.

4.1. Source code and CCC analysis

The code analysis revealed that most of the several concerns implemented in the two systems (such as tracing, logging, debugging, message handling and protocol logic enforcement, etc.) crosscut the system dominant decomposition. Moreover, the way the DPs were implemented highly contributed to CCC. For example, with reference to the AGS system, the tracing/debugging and logging concerns were largely scattered across the entire system. These are “orthogonal” concerns with respect to dominant system decomposition. Because OO implementation of DPs help to shape the dominant decomposition of the base system, also the DP modules contained scattered elements from all the secondary concerns.

4.2. Analyzing the DP implementations

In the following subsections we will provide some details about the DPs analysed in our case study focusing on the problems and issues highlighted in section 3.

Forwarders/Receivers. The AGS system drives the several devices (low and high speed serial lines, Ethernet, wireless LAN) it manages by dedicated interfaces. The Forwarder/Receiver (F/R) DP was adopted to handle communications over different media in order to decouple the message and protocol logic from the logic handling the remote interfaces. Distinct F/R abstract classes were used to inject into concrete classes the code implementing the different logic to handle the different roles played by each implemented F/R DP. This was due to the impossibility to handle, in the concrete system classes, different instances of the same pattern relationship by just using only inheritance, and interface implementation and composition. Thus concrete classes were forced to host pattern code by affecting negatively the DP reusability.

All that contributed to increase the crosscutting of the secondary concerns in the DPs. Figure 4 reports the crosscutting degree related to the data-integrity and marshalling secondary concerns in the F/R DP implementations. In the figure the boxes represent the physical source code containers, like implementation files or units.
In each box the coloured lines represent code lines, the colour of each line represents the concern the code line is associated to. Thus different lines with the same colour in different boxes indicates the presence of code scattering while the presence in a box of lines with different colours indicates code tangling. In the figure each box is identified by the name of the software unit implementing the several F/R instances in the AGS system. The figure shows that the code for marshalling and data integrity checks is scattered across the entire F/R DP implementations. As a consequence send/get-message pattern’s operations suffer of tangling (because they contain code to address scattered secondary concerns) and code duplication.

Command. In the JHD framework the Command pattern was implemented to handle all the kind of valid commands. A Command hierarchy was defined and implemented, by the developers at this aim; figure 5 shows an excerpt of this hierarchy. The CCC analysis of the Command hierarchy showed a high crosscutting degree related to the ‘Undo’ functionality (figure 6): the code implementing the ‘Undo’ functions is scattered across the Command classes which the ‘Undo’s are associated to.

State. The State DP is implemented in JHD to handle the state of the several tools the users can select and use by a tool palette. The DP has been implemented by a generalization-specialization hierarchy (figure 7) rooted in the abstract class ‘AbstractTool’, an implementation of the interface ‘Tool’. The AbstractTool subclasses implement the state-specific behaviour for the concrete tools in the framework. This implementation of the State DP introduces scattered code in the subclasses to perform state transitions. The State DP is, in turn, affected by the CCC due to the observing relationships of the Observer DP and tool hierarchy related to the ‘Undo’ infrastructure in the Command DP.

Observer. Instances of the Observer patterns have been found in both AGS and JHD systems where they were adopted to handle multiple observing relationships. In the AGS the DP was adopted to issue commands to the motion controller. This took into account: the status of several hardware devices, the status of software system, and the weather conditions. Figure 8 shows the class diagram modelling the key observing relationships in the AGS system. Two abstract classes were defined to handle the subject and observer roles. The analysis of the Observer DPs implemented in the JHD system is just focused on the ones ‘to observe’ the events issued by Command and State DPs: the two ‘EventDispatcher’ classes in figure 5 and figure 7 were charged to send and to receive the events. The event logic to handle the observed events is spread across the Command and State hierarchies. Due to the two different observing relationships, two different Observer DP implementations were developed. This increased code duplication across the units implementing the Command and State DPs. Two abstract classes were defined to handle the roles of subject and
observer by introducing a double indirection. This generated roles superimposition overloading concrete classes that already have a role in the system (such as hardware sensors, servers, consoles and panels, in the case of AGS system), with secondary roles needed to enforce the pattern logic. The degree of CCC and code duplication was then highly increased, by reducing the understandability and reusability of the concrete classes. A further problem was due to the way the multiple observing relationships was handled. Indeed, by using inheritance, a concrete class could be involved, as subject or observer, just in one observing relationship. The AGS developers were forced to modify the implementation of pattern operations by adding extra code to implement the application logic needed to distinguish the different kinds of events to observe. That generated muddled pattern instances, with a high coupling and reduced understandability and traceability.

The JHD developers duplicated the pattern implementation to be able to use different event dispatchers to handle each type of observed relationships. That increased the code duplication in the system.

<table>
<thead>
<tr>
<th>Table 2. Main issues and affected patterns</th>
</tr>
</thead>
</table>

Table 2, summarizes, according to the list in Section 3, the identified issues generating CCC in the analysed DPs implemented in the AGS and JHD systems.
4.3. Re-implementing the DPs by AOP

To eliminate or reduce the crosscutting of secondary concerns in DP implementations, the DPs have been re-implemented by AOP using AspectJ as programming language. Each DP was re-implemented according to the three types of re-implementations (Lazy (L), (un)pluggable (U), reusable (R)) described in Section 3.

For sake of brevity just the AOP re-implementations of the DPs in the AGS system will be discussed in the following.

4.3.1. The AOP Observer DP re-implementation for AGS. The ‘Lazy’ implementation has the advantage that it is simple to do, but it leaves many issues still open (e.g., code duplication, unchanged reusability, pattern traceability). Indeed, because the implemented aspects inject the attach/detach and update/getState Observer’s methods into system concrete classes, the state of the observing relationship is anyway contained in such instances. In the (un)pluggable implementation the pattern logic is implemented into one aspect. Figure 9 shows a class diagram modelling the ‘U’ implementation for the Observer pattern: the roles of the pattern are associated to concrete classes by means of the ”‘declare parent’” construct using marker interfaces. This is enough, in this simple case, to obtain an (un)pluggable Observer [4]: the logic of the pattern is encapsulated into the aspect which capture the context for the Subject state changes and call update on registered Observers. However the ‘U’ implementation still lacks for reusability and it is not able to handle multiple observing relationships. The ‘R’ implementation was developed to obtain a more (un)pluggable and reusable Observer. The pattern was re-implemented by a three layered structure:

- Pattern logic concern layer: gathering all the components encapsulating the logic of the pattern;
- Base system class layer: gathering the concrete system classes derived from the dominant decomposition
- Mapping concern layer: gathering the components allowing the mapping between concerns; this is made by intercepting the instantiation of new objects and enforcing the Observer/Observed protocol between those instances that have to communicate each other.

Common elements among different pattern instances can be factored in the ‘Pattern logic concern’ layer while multiple observing relationship can be more easily solved in the
4.3.2. The AOP Forwarder/Receiver DP re-implementation. In the case of the F/R DP just the ‘L’ and ‘R’ AOP implementations were performed. The ‘L’ implementation presented the same drawbacks and issues discussed for the Observer one. Figure 11 reports the CCC degree for the F/R Lazy implementation: with respect to the figure 4, there is no scattered and tangled code in the units implementing the F/R pattern, but now the code is all encapsulated into the new aspects (Data marshalling and Data Integrity).

4.3.3. Evaluating the AOP implementations. The AOP implementations of each DP was evaluated with respect to the set of metrics shown in section 3. The table in figure 12 reports the computed values of those metrics. The ‘O’ rows in the table report the values of some metrics applicable both for the OO and AOP implementations (‘NA’ means that the metric was not applicable to OO implementation). We can observe that the AOP values are not worse than the OO ones; the values for ‘R’ implementation are always better than ones of OO implementations. By analysing the values in the table we can note that:

- the WOM metric values become better and better going to the ‘L’ to ‘R’ implementations: indeed, the WOM metric values decrease moving from the Lazy AOP implementation (that are more based on static relationships) to the layered and reusable one (that are more based on dynamic relationships);
- the LCO values are higher for the ‘L’ implementation, than the ‘U’ and ‘R’ ones;
- the RFM values are higher for the ‘L’ implementation, than the ‘U’ and ‘R’ ones too.

This indicates that the cohesion of the ‘U’ and ‘R’ implementations is better that ‘L’ ones (in some cases ‘U’ values are better than ‘R’ ones). On the other hand, the values of the metrics related to coupling would seem to be worse moving from the ‘L’ implementation to ‘U’ and ‘R’ ones. This is due to the greater number of static/dynamic links from aspects, where the code generating CCC was encapsulated, to the other modules where the scattered/tangled code was.

This is mainly because AOP developers, due to the characteristics of AOP constructs, tend to use largely expressive referential mechanisms for quantification over program elements. As a consequence a greater number of links from the aspects to the other system components may exist. Anyway, the modularity is improved because the code now lays just in a single unit, that will be more understandable, maintainable, and so on. Thus, the CIM metric values increase from Lazy version to the Reusable one. The CIM values quantify the coupling focusing on the aspects that intercept the operations of another modules, those explicitely included in the definitions of the pointcuts. High values of CIM indicate high coupling of the aspect towards the other referenced software components (hence a lower generality/reusability). The Lazy implementation has a lower value for CIM because it is not based on pointcuts (being it structured by an OO-style design that injects logic in concrete classes). Similar considerations can be done for the other coupling metrics.

All that shows that AOP is able to improve the modularity of current OO DP implementation also by simple AOP
implementation as the Lazy one we used in the case study.

5. Conclusions and future work

OO implementations of DPs presents several problems (such as CCC, overlap, indirections, ...) affecting negatively the software quality, and mainly the modularity. Re-implementing DPs by AOP is an effective way to avoid the CCC due to the DPs OO code scattering. AOP allows different design solutions to re-implement a DP; each solution has to be evaluated to assess quality attributes. In particular, coupling and cohesion are a critical issue for AOP implementation, due to the AOP interception mechanisms at run time. The selected set of metrics allowed to evaluate effectively the coupling and cohesion of different AOP implementations and showed conflicts among some attributes too.

Future work will consider experimentation involving more non-GoF DPs from several domains as well the definition of new design solutions to improve coupling and cohesion of AOP components. Moreover, further research about properties like optionality and patterns switch-ability is also needed to provide an overview of the design tradeoffs in the AOP context. Finally the definition of new metrics is needed to highlight AOP design flaws, such as coupling due to complex events interception, fragile pointcuts declarations, aspect laziness, OO-style decentralised design.

References