Using AOP to improve Design Patterns Modularity
Mario L. Bernardi, Giuseppe A. Di Lucca
RCOST - Research Centre on Software Technology
Dept. of Engineering - University of Sannio
dilucca/mlbernar@unisannio.it

ABSTRACT
The way Design Patterns (DP) are implemented may affect the modularity of a software system, due to some typical deficiencies of Object Oriented (OO) languages. Aspect Oriented Programming (AOP) provide patterns’ developers with powerful quantification constructs to better handle modularity and composition and to overcome some of the OO design trade-offs in current DP implementations.
In this paper a case study where AOP has been used to re-implement DPs in existing systems from real world is presented: some different AOP re-implementations of the DPs were performed and evaluated with respect to a set of selected metrics in order to drive a better AOP design of improved DP.

KEYWORDS
Software Engineering, Design Patterns, Aspect Oriented Software Development, Software Metrics

1 Introduction
Object Oriented Design Patterns provide the design of generic solutions to recurring problems [1]. However, DPs implementation may introduce in OO systems some problems affecting some quality attributes (such as modularity, understandability, maintainability, testability, reuse) that can be hard to solve within the object oriented paradigm. Usually, these problems are due to the indirection techniques forcing one or more key interfaces to be implemented (often introducing a greater overhead). Software systems (both object and function oriented) are usually designed thinking to a dominant decomposition, but it could be difficult to encapsulate into a specific module all the code implementing different aspects of a certain requirement or functionality (e.g. persistence, logging, memory management and concurrency).
In Aspect Oriented Programming (AOP) the implementation of requirements/functionalities spreading across several modules is called “Cross-Cutting Concerns”, because it cuts modules boundary and is fragmented across a number of modules.
The main negative effects produced by Cross-Cutting Concerns (CCC) presence are:
- code scattering: code for a concern is spread in several places;
- code tangling: modules deal with more than one functionality (i.e. low cohesion modules)
Thus the presence of CCC affects the modularity of a system, and thus maintainability, understandability, testability, and so on.

Recent researches have shown that many DPs involve Cross-Cutting Concerns (CCC). In [2, 4, 5] it was shown how several patterns from GoF catalog [1] may introduce crosscutting that OO abstractions are often unable to well modularize. That is mainly due to the poorness of the composition and quantification constructs in OOP languages that do not allow a good modularization of the concerns. Indeed, by using OO DPs programmers are forced to add classes, interfaces, methods and attributes inside the code of the components (i.e. classes, packages, etc.) derived from the dominant decomposition. The introduction of these elements will produce code scattering and tangling by reducing the quality of some DP attributes such as reusability, traceability, comprehensibility, maintainability.
AOP constructs are able to better modularize DP concerns. In [2, 4, 6] AOP implementations of GoF [2] patterns are provided; they have better values for properties such as locality, (un)pluggability, composability and reusability.
Thus, a way to improve the modularity of OO DP is 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 affecting differently the quality of the AOP implementation adopted.
In this paper we present and discuss a case study where some systems, from real world, developed by using DPs were analyzed to verify the presence of DP code scattering and tangling; different AOP re-implementations of some DPs affected by code scattering and tangling were performed; the different AOP DP implementation were evaluated with respect a set of appropriate metrics. Such an evaluation can be used to drive a better design of improved DP by AOP implementations.
The paper is structured as follow: in Section 2 the overall approach used in the case study is presented together with the set of metrics used to evaluate the different DPs’ AOP re-implementations; Section 3 presents the case study we carried out with reference to the results for one the analysed systems. Conclusive remarks and future work are finally presented in Section 4.
2 An overview of the approach used to drive the case study

The case study was carried out according a four steps process:

a) static source code analysis of the existing systems to identify DP’s code scattering and tangling (i.e. CCC);
b) CCC analysis;
c) different AOP re-implementations of the DPs affected by CCC
d) evaluation of the different AOP implementations of a same DP by a set of metrics and quality assessment.

Cross-Cutting Concerns is identified by using lexical matcher to identify the code implementing a specified concern. 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. The results of the CCC identification are then analysed to recognise the reasons generating the CCC.

The main typical issues in OO DP implementation generating CCC are:

1) *Patterns overlap*: this is when concrete classes participate to more than one pattern instance. This reduce pattern traceability and concrete classes reusability.

2) *Inheritance relationships*: this is when inheritance is used to assign roles to pattern participants. Since inheritance relationship is static, developers are forced to define pattern logic and possible participants to patterns ahead of time. Moreover, concrete classes contain pattern code thus being less reusable and understandable.

3) *Encapsulation violation*: this is when patterns force to expose internal 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 already having 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 DP do not introduce directly crosscutting concerns they can contribute to shape the principal decomposition of the base system. This means that secondary concerns will be scattered also across the patterns implementation.

Some of these issues (like the ones at points 1, 3, 4, 5) are well known and discussed in the literature, as well as effective AOP solutions have been proposed for those at points 1), 4), and 5) [5],[10], [15], [4]. The issue at point 2) has not been deep addressed while 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 issues of the previous bullet list affect each DP.

In the third step, DPs affected by CCC are re-implemented by AOP: different AOP implementations are made, each one aiming to improve some AOP quality attributes. In particular the attributes of (un)-pluggability, reusability were considered.

Each AOP implementation is evaluated with respect to the following set of metrics:

- Weighted Operations in Module (WOM)
- Depth of Inheritance Tree (DIT)
- Number Of Children (NOC)
- Crosscutting Degree of an Aspect (CDA)
- Coupling on Advice Execution (CAE)
- 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)

These metrics were selected among the ones proposed in [10], being them a redefinition of the OO Chidamber and Kemerer metrics [11] tailored for AOP.

The selected metrics (aiming to evaluate the CCC, the coupling, the cohesion and the size of the AOP code) are usually good indicators for assessing the goodness of a module. The AOP implementations are then compared with respect to these metrics; this can help to select which implementation is better to use/reuse in a specified context.

3 Case study

The case study involved three systems from real world. For the sake of brevity, in the following we present and discuss the results related to one of the analyzed systems: 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 include 37 classes. It receives control and status messages from remote devices (like control panels, consoles, environmental sensors), performs checks and issue commands to an advanced non-linear motion controller for an antenna group. The development of the system was based on a three layers 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, and it was designed and implemented using well known DPs from several domains (GoF, communication, architectural , etc.). The following patterns were implemented 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. The four steps of the process described in section 2...
were carried out.

3.1 Source code and CCC analysis
The code analysis revealed that most of the different concerns (tracing, logging, debugging, message handling, and protocol logic enforcement, etc.) were crosscutting with respect to the system principal decomposition and, moreover, the used DPs increased crosscutting concern presence in several ways. In particular, there were explicit requirements for AGS server about tracing/debugging and logging concerns: these concerns are largely scattered across the entire system. They are “orthogonal” concerns, with respect to base system decomposition Figure 2 reports the crosscutting degree for logging and tracing concerns in the AGS system. The figure reports, for each one of the architectural layers, the software In the figure the boxes represent the physical source code containers, like implementation files or units, grouped according to the architectural layers they belong. In each box the coloured lines represent code lines, the colour of each line represent 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 of different line with different colours indicate code tangling.

The figure highlights that orthogonal concerns are scattered across all the AGS’ modules.
Of course since OO DPs help to shape the principal decomposition of the base system, they contain, like all other modules, scattered elements from all the secondary concerns including those identified above.

3.1.1 Analyzing the Forwarder/Receiver DP implementation
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 the different media in order to decouple the message and protocol logic from the logic to handle the remote interfaces.

Figure 1 shows the class diagram (recovered by reverse engineering) modelling the structure of the F/R patterns as implemented in AGS system. To make more readable the diagram, just the class related to the receiver are shown (however the forwarder classes are symmetric to receiver ones). Distinct F/R abstract classes were used to inject into concrete system classes the code implementing the different logic to handle the different roles that each implemented F/R DP has in the system. This is due to the inability to handle different instances of same relationships by explicit use of inheritance, interface implementation and composition in concrete system classes definition. But, that compromises DP reusability since concrete classes are forced to host pattern code. All that affects also the crosscutting of the secondary concerns in DPs by increasing such a CCC. Figure 3 reports the cross-cutting for the secondary concerns related to the data-integrity and marshalling concerns in the F/R DP implementations: marshalling and data integrity checks are scattered across the entire F/R implementations. In the figure each box is identified by the name of the software units implementing the several F/R instances in the AGS system.
3.1.2 Analyzing the Observer DP implementation

The AGS server must issue commands to motion controller taking into account the status of several hardware devices, the status of software system, and the weather conditions. The Observer DP was thus adopted at this aim.

Two abstract classes were defined to handle the subject and observer roles (double object oriented indirection). These indirections generate roles superimposition overloading concrete classes, that already have a role in the system (such as hardware sensors, servers, consoles and panels), with secondary roles needed to enforce the pattern logic. This increases the crosscutting degree of concerns, and code duplication by reducing the understandability and reusability of concrete classes.

A second issue is related to the handling of multiple observing relationships. Due to the usage of inheritance, a concrete class can be involved, as subject or observer, into just one observing relationship. AGS developers were forced to modify the implementation of patterns operations adding extra code to implement the application logic needed to distinguish among different kind of observing events. All that generated confusing pattern instances, with an increased coupling by reducing traceability and understandability.

Table 1, summarizes, according to the list in Section 2, the identified issues generating CCC in some of the DPs implemented in the AGS system.

<table>
<thead>
<tr>
<th>DP overlaps</th>
<th>Inheritance</th>
<th>Encapsulation</th>
<th>Object-Oriented</th>
<th>Indirection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4 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 and using AspectJ as programming language.

Three different kinds of AOP implementations were performed:

- a ‘Lazy’ (L) AOP implementation: this just aims to a simple better modularization of the scattered code. 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 also this implementation still suffers 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 were used to gain greater (un)-pluggability and reusability.

4.1 The AOP Observer DP re-implementation

The ‘Lazy’ implementation used inter-type declaration to move into an aspect the member defined for OO Observer pattern. The advantage of this implementation is just that it is simple to do, but it leaves too much issues still open (code duplication, unchanged reusability, pattern traceability). Indeed, for example, since aspects inject the attach/detach and update/getState Observer's methods into system concrete classes, the state of the observing relationship is still maintained in such instances.

In the (un)-pluggable implementation the pattern logic is implemented into the aspect. Figure 4
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 ‘U’ implementation still lacks for reusability and it is not able to handle multiple observing relationships.

An ‘R’ implementation was developed to obtain a more (un)-pluggable and reusable Observer. The pattern was re-implemented by a three layered structure (see Figure 5):

- 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 base decomposition
- Mapping handling concern layer: gathering the components allowing the mapping between concerns by intercepting the instantiation of new objects and enforcing the Observer/Observable protocol between those instances that have to communicate each other.

Commonalities among different pattern instances can be factored in the ‘Pattern logic concern’ layer while multiple observing relationship can be more easily solved in the ‘Mapping handling concern’ layer by associating two observable aspects to the same concrete class. This implementation is showed in Figure 6.

4.2 The AOP Forwarder/Receiver DP re-implementation

In the case of the F/R DP just the ‘L’ and ‘R’ AOP implementation were performed. The ‘L’ implementation present the same drawback and issues discussed for the Observer one. Figure 5 reports the CCC degree for the F/R Lazy implementation: with respect to the figure 3, there is no scattered and tangled code in the units implementing the F/R pattern, but such a code is all encapsulated into the new aspects (Data marshalling and Data Integrity). Figure 8 show the class diagram modelling the F/R pattern ‘R’ implementation: in this case some aspects are defined to modularize the secondary concerns related to Data Marshalling and Data Integrity.

4.3 Evaluating the AOP implementations

The several AOP implementations of each DP was evaluated with respect to the set of metrics indicated in section 2. The table in figure 7 reports the values of those metrics for the F/R and Observer patterns. By analysing the values in the table we note that the cohesion of the ‘U’ and ‘R’ implementations is better that ‘L’ ones (in some cases ‘U’ values are better than ‘R’ ones):

- 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 for higher for the ‘L’ implementation, than the ‘U’ and ‘R’ ones too.

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. The main reason is that having encapsulated the code into aspects to eliminate the CCC, such modules will have a greater number of static/dynamic links 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). Thus there may be a greater number of links from the aspects to the other system components, but the modularity is better anyway because the code now lays just in a single unit thus resulting into a better understandability, maintainability, flexibility, 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 intercepts the operations of another modules (those explicitly included in pointcuts' definitions). High values of CIM indicate high coupling of the aspect towards the other referenced software components (hence a lower generality/reusability). But this is not surprising because the Lazy implementation has a lower value for CIM since it is not based on pointcuts (being it structured by an OO-style design that inject logic in concrete classes).

Similar considerations can be done for the other coupling metrics. All that confirm us that AOP is able to improve the modularity of current OO DP implementation already by simple AOP implementation as the Lazy one we used in the case study. When advanced AOP features like (un)pluggability, reusability, flexibility, and switch-ability of pattern instances are considered and would be improved a compromise between cohesion and coupling has to be met to get a more flexible and reusable design.

Moreover new metrics to highlight AOP design flaws (e.g. coupling due to complex events interception, fragile pointcuts declarations, aspect laziness, OO-style decentralised design) will be used or defined.

6 References
[1]. Gamma, E., Helm, R., Johnson, R., Vlissides, J., 'Design Patterns: Elements of Reusable Object-Oriented Software', Addison-Wesley (1995)
[6]. Magiel Bruntink, Aspect Mining using Clone Class Metrics, Proc. of Workshop on Aspect Reverse Engineering 2004 (WARE '04)
[14]. Martin P. Robillard, Gail C. Murphy, “Capturing Concern Descriptions During Program Navigation”, a position paper for the workshop on Tool Support for Aspect Oriented Software Development of the Object Oriented System Languages and Applications 2002 (OOPSLA '02)

Figure 8: F/R layered dynamic implementation

5 Conclusions and future works
AOP can improve some quality attributes of DP implementations such as reusability, flexibility, (un)pluggability, modularity. In the paper a case study where OO DP implementations, from real world systems, have been re-implemented by AOP and assessed with respect to a set of metrics has been presented. The study highlighted several the main problems in OO DP implementations (such as CCC, overlap, indirections, inheritance dependencies) and the results show that AOP actually can improve the modularity of DP patterns. The AOP paradigm allows different design solutions each actually can improve the modularity of DP patterns.

Future work will be addressed to more experimentation involving non-GoF DPs from several domain. Further properties such as optionality and patterns switch-ability will be investigated to provide an overview of the design tradeoffs.