A Metric Model for Aspects’ Coupling

Mario Luca Bernardi  
Department of Engineering – RCOST  
University of Sannio, Italy  
mlbernar@unisannio.it

Giuseppe A. Di Lucca  
Department of Engineering – RCOST  
University of Sannio, Italy  
dilucca@unisannio.it

ABSTRACT
Aspect Oriented Programming (AOP) introduces new types of coupling among the aspects and the components of the base system. Indeed, several and different new kinds of interactions among aspects and the other components can be introduced by the AOP constructs, allowing the alteration both of the structure, control and data flow of the components of the base system. These interactions can make higher the complexity of the overall system affecting its comprehension.

In this paper we present a proposal for a metric model to classify the types of coupling among aspects and the components of the base system. The model can be used to define how each kind of coupling affects the complexity, and thus the comprehensibility, of the system.

Categories and Subject Descriptors
D.2.8 [Software Engineering]: Metrics – complexity measures, product metrics.

General Terms
Measurement, Languages.

Keywords
Software Quality, Program comprehension, Aspect Oriented Programming

1. INTRODUCTION
The Aspect Oriented Programming (AOP) language constructs, structures, and mechanisms [4, 12] introduce a new kinds of coupling among an aspect and the other components of an AO system. Indeed, the code in the aspects is able to change both structural (static) properties of a component in the base system and its dynamic behavior at run time, thus creating coupling relationships, between aspects and the affected components, that are different from traditional ones.

New methods and fields can be added, by an aspect, to a class of the base system by using inter-type declarations as well as inheritance relationships can be altered, and interface implementations forced.

Aspects can define pointcuts specifying both the conditions triggering the execution of blocks of code (advice) in the aspects and the points (join points) in a program execution at which the code in the advice has to be executed. The code encapsulated in aspects, is woven into the code of the ‘traditional’ program units (e.g. classes in OO code) to build the overall system. The system resulting after weaving presents coupling relationships among aspects and ‘base modules’ that are implicitly and declaratively defined with respect to ‘traditional’ explicit and imperative ones: hence the system structure, and behavior, of the complete weaved system may be largely different from the ones of the base system (i.e. the system with no weaving ). While, on one hand, the AOP constructs better supports the separation of crosscutting concerns by encapsulating them into the aspects, on the other hand the new forms of coupling they introduce could make system comprehension more difficult. For example, it could be more difficult to identify and comprehend which are the portions of code directly or indirectly affected by aspects, or if a class/method is affected by an aspect, as well as which are the portions of code that will be actually run at a given point in the code.

Then, although AOP would potentially allow the development of better modularized systems, if the peculiar AOP constructs are used in undisciplined manner the resulting system could resemble a new kind of ‘spaghetti bowl’, mainly due to the new ways an aspect can be coupled with the other components.

It could be difficult to gain a full comprehension of the system: code reading and inspection activities could be unsuccessful because it would be hard to know if new fields or methods have been added to a given class, or if any inheritance relationship affecting a class has been altered, or to identify all the join points defined by a pointcut at which some advices could take place.

Each of the aspectual constructs (such as inter-type declarations, pointcuts, advices) interacts and alters the structure, control and data flow of the components of an AO system in a different way and with a different level of coupling.

This paper is an improvement on our previous work [16] in which a taxonomy of aspects’ interactions was proposed.

In this paper we propose a metric model, based on that taxonomy, representing the ways by which aspects can be coupled with the components of the base system. These ways have been analysed and classified; in particular we have been considered the coupling related to alterations of (i) the static structure of base components, (ii) the inter-procedural control flow, and (iii) the state of objects. The main aim was to identify a set of coupling forms among aspects and the components of the base system. This set can be used to understand which are the aspects in the system that are too coupled with the base system, hence affecting its quality (comprehensibility, maintainability and so on).
The metric model can be used to evaluate the degree of coupling introduced by aspects. However, at today the model is defined on a nominal scale, thus it is not yet possible to say which attributes affect complexity and comprehensibility more than others attributes.

The paper is structured as follows: in section 2 some related work are discussed; in section 3 the proposed model is presented and discussed, while the section 4 presents a case study where the model is used to classify the coupling of aspects in a system from the real world and to make some first considerations about how each kind of coupling can affect the comprehension of the system.

2. RELATED WORK

Few works have explicitly studied new kind of couplings providing a complete picture for them. The only work on the subject, to the author’s knowledge, that explicitly defines some coupling metrics is [15] in which an extension of the Chidamber and Kemerer metric suite for AOP is proposed. With respect to this work, our proposal is more oriented to provide a wider spectrum of all the kinds of couplings that could exists in an AOP system, rather then extend an existing metric suite.

Indeed, several papers, that are still relevant with our proposal, have dealt with the analysis of interactions among aspects and the base system’s components.

In [6], an aspect is considered harmful if it invalidates desired properties of the system to which it is applied; the focus is on the identification of aspect categories either based on static code analysis, or with transformation rules that allow deriving only aspects of a desired type. Aspect-aware interfaces are used to weaken obliviousness at the code level. The aspects enabled in a module have to be explicitly declared in these interfaces: this makes the effects of method activations more easy to comprehend for both modularity and analysis goals.

Additional categories of aspects have been defined in several works. The work in [5] defines an observer aspect (defined as an aspect that doesn’t affect the state of the base system). It points out issues of aliasing that are hard to solve by an automatic static analysis. Its also suggests a development methodology based on observers. In [9] a code analysis system is proposed for a simplified aspect language intended primarily for optimizations. Anyway the gathered information can also be used to identify those aspects that add behavior to the system without affecting the base system one.

The same issues are discussed in [10]: an extensive interference analysis is proposed for AspectJ-like programs, emphasizing the complications introduced by inheritance and multiple instances. The result is an effective methodology for identification of observer aspects (which is called ‘interference freedom’).

Other works [1, 2, 11] extend to aspects well-known programming slicing techniques based on dataflow analysis. These are used to identify the extent of influence of an aspect on the underlying system, and to identify potential conflicts among aspects. Such techniques can also be used to reduce the size of the model to be analyzed when model checking techniques are applied. In particular in [2] a slicing technique for AspectJ is proposed to identify the influence of each aspect. A similar approach has been proposed for Composition Filters in [8].

Finally in [7] aspect advices are represented by a state transition graph that is model checked to extend invariants proven true for the original system to the system with aspects applied on it.

While some works are starting to provide insights on aspect invasiveness and complexities introduced by them, the metric model proposed in this work is, to the authors’ knowledge, a first contribution that takes into account and organizes, in a in a well defined structure, the entire spectrum of kind of couplings the aspect are capable of. It provides a starting point for reasoning about:

- the ways the aspects modify and interact with the remaining of the system;
- the kinds of coupling generated by such alterations and interactions.

3. THE PROPOSED MODEL FOR ASPECTS’ COUPLING

Coupling is “The manner and degree of interdependence between software modules.” [17]

In this section we analyze and discuss the different (new) manners by which an aspect can interact with the other software components due to the specific AOP feature, i.e. the ways an aspect can be coupled to ‘traditional’ software components, such as classes. A metric model is then defined on the base of these coupling mechanisms.

The specific AOP language constructs allow an aspect to interact with the other components mainly to alter:

- their static structure of both code or system, e.g. by injecting fields or methods into a class, or by altering the hierarchy structure along inheritance relationships;
- their control flow, by selecting join points at which the control of the execution may be passed to an advice;
- their state of objects, e.g. by using or defining, along advice executions, data transferred at join points.

The first category is related to static crosscutting features of AOP languages while the others two are related to dynamic crosscutting ones.

Each of these three main ways of AO specific coupling can cause different degree of alterations of the (base) system, going from the absence of structural modifications to a deep invasive alterations affecting the behaviour or the structure of the system.

3.1 Notation

The following notation will be used in the remaining of the paper. By $T$ we refer to the set of all the types (i.e. classes, aspects, and interfaces) defined in an AO system: $T = I \cup A \cup C$ ,where $I$, $A$, and $C$ are respectively the set of Interfaces, Aspects and Classes defined in the system. Given a $t \in T$, $O(t)=\{o_1, o_2, ... , o_n\}$ is the set of the operations $o_i$ , i.e. methods of classes/interfaces or advices of aspects, declared in $t$, while $F(t)=\{f_1, f_2, ..., f_n\}$ is the set of the fields $f_i$ declared in $t$. Given an aspect $a \in A$, $Adv(a) = \{adv_1, adv_2, ... , adv_m\}$ is the set of advices declared by $a$, while $P(a)=\{p_1, p_2, ... , p_k\}$ is the set of the pointcuts designators $p_i$. 
3.2 Coupling due to interactions altering the static structure

Intertype declarations in the aspects allow to modify the structure of a type \( t \in T \), in several ways:

a) by defining constraints, i.e., restricting or forcing, the access to types, fields, or methods;

b) by adding fields or methods to \( t \);

c) by forcing a class to implement an interface.

d) by altering the inheritance relationships existing among \( t \) and the other types;

The most of AO languages, provide static directives to define access constraints that are resolved at compile time, thus they are identifiable by code static analysis. The AspectJ AO language [14], as well as most of the AO languages, provides the declare error statement to define access constraints. This statement is associated with a pointcut designer and allows to define:

- the restriction contexts, namely the elements under which constraints are to apply; these are specified by means of join points matching against well defined source code regions using the within and withincode pointcut designators.

- the restricted elements, namely the elements that have restricted access to. In this case all the other statically determinable pointcut designators can be used.

Figure 1 shows an example of an aspect with a declare error statement. In this example the classes whose name start by the string “QuickSort” represent the restriction context and the operations in them whose names end by the string “swap” are the restricted elements: an error will be issued if and only if in the “QuickSort*” scope there are calls to methods whose name ends with “swap”.

An intertype declaration that defines access constraints to some elements of the base system originates a form of coupling that we call access restriction, according to the following definition:

\[
\text{aspect Simple} \{
\text{declare error:}
\text{within (QuickSort*) & & call (*swap(...));}
\}
\]

Figure 1: A declare error statement.

Definition 3.2.1 An intertype declaration that defines access constraints to a set of code elements originates an access restriction coupling. A given aspect \( a \in A \) contains an access restriction for a source code element \( e \in T \cup O(t) \cup F(t) \) iff:

1. the aspect \( a \) contains a declare error statement associated with a pointcut designator \( p \);

2. \( e \) is an element corresponding at least to one join point shadow \( jp \in JP(p) \).

The case b) is related to the usage of intertype declarations to add operations or fields to the original code of a type \( t \), i.e., new elements are added to \( t \) by injecting in it more code forming the added fields or operations that alter its original structure and behaviour. The injection of fields or operations can cause the overriding of the (eventually) existing ones (modifying class, state and behaviour). Figure 2 shows an example of an aspect injecting some fields and methods into a class.

```java
class A {
    aspect LockingAspect {
        int A.lock;
        void acquireLockRecursive() { A.lock++; … }
        void releaseLockRecursive() { A.lock--; … }
    }
}
```

Figure 2: An aspect injecting fields and methods into a class.

We call injective the coupling due to an intertype declaration altering the code of a type \( t \) according to the following definition:

Definition 3.2.2 An intertype declaration that augments the state or the behaviour of a type \( e \in T \) by injecting fields or operations into it originates injective coupling. The “data” and “behaviour” qualifiers are used for those declarations injecting respectively fields and methods.

The case c) happens when an intertype declaration compels a class or an aspect to implement an interface, i.e. a class (or an aspect) is forced to implement an interface. The figure 3 shows an example of an aspect compelling a class to implement an interface.

```java
class A {
    void m() { // implementation of m() }
}
aspect MarkerAspect {
    interface I {
        void m();
    }
    declare parent: A implements I;
}
```

Figure 3: An interface marker aspect.

We call interface marker the coupling due to an intertype declaration forcing the implementation of an interface by a class or aspect, according to the following definition:

Definition 3.2.3 An intertype declaration compelling a type \( t \in C \cup A \) to implement the interface \( i \in I \) gives origin to interface marker\(^1\) coupling.

A particular case of coupling is when injective and interface marker intertype declarations are in the same aspect and the

\(^1\) The term ‘interface marker’ is not to be confused with the ‘marker interface pattern’ one providing the latter a mean to associate metadata to a class for those languages that have not an explicit support for declaring metadata on language elements.
injected methods are those declared by the interface: in this case we distinguish a new kind of coupling called role providing, according to the following definition:

**Definition 3.2.4** If an aspect aeA contains an interface marker declaration for an interface tεI and injective declarations for some of the operations declared in a, a role providing coupling is originated.

The figure 4 shows a simple example involving the classes A, B, the aspect X, and the interface I, while figure 5 depicts a graphical representation of that code. The aspect X acts as an interface marker of class A by the interface I1 while being a role provider of A by the interface I2. The class B is in a situation in the middle of these two: the aspect provides a partial implementation of the interface I2 letting B to “complete” and customize the role implementation. In this case X is an abstract role provider for B.

```
class A { void n() { // code } }
class B { void p() { // code } }
aspect X {
    interface I1 {
        void n();
    }
    interface I2 {
        void m();
        void p();
    }
    void I2.m() { // code }
    void A.p() { // code }
    declare parent:
        A implements I1,I2;
}
```

---

**Figure 4: An interface markers and role providers**

The case d) is when an aspect affects the hierarchy of a generalization/specialization structure modifying it by sub-typing the parents of a target type t.

We call gen-spec alteration the coupling due to an intertype declaration that alters a generalization/specialization structure according to the following definition:

**Definition 3.2.5** An intertype declaration changing the generalization/specialization relationship existing among a type tεC\A its super-type sptεC\A and/or its sub-type(s) sbtεC\A originates gen-spec alteration coupling.

According to the previous definitions, the coupling due to aspects’ interactions altering the base system static structure can be classified as follows:

- access restriction (AR);
- injection (I);
- interface marker (IM);
- role providing (RP);
- gen-spec alteration (GSA).

Since an aspect may contain more different kinds of intertype declarations, their number and types will determine the level of coupling of the aspect, i.e. the level can be represented by a function f() of the kinds and number of the intertype declarations: \( \text{CLSS}(a) = f(\alpha, AR(a), \beta, I(a), \gamma, IM(a), \sigma, RP(a), \omega, GSA(a)) \)

where:

- CLSS(a): Coupling Level of the aspect a related to the Static Structure of the base system
- AR(a): total number of access restriction intertype declarations in a
- I(a): total number of injective intertype declarations in a
- IM(a): total number of interface marker intertype declarations in a
- RP(a): total number of role providing intertype declarations in a
- GSA(a): total number of gen-spec alteration intertype declarations in a
- \( \alpha, \beta, \gamma_1, \sigma, \omega_1 \): different weights by which the occurrences of each kind of declaration are to be evaluated according to an established ordered scale of values.

### 3.3 Coupling due to interactions altering the control flow

Aspects can introduce alterations in the control flow of an operation because of the inter-procedural relationships among operations established at join points. When, during the execution of the code of an operation, a join point matches one or more pointcut expressions, the control flow is transferred to the advice(s) associated to the matched pointcuts following well defined precedence rules. The code of the triggered advice can be executed either before, after or around the intercepted join point, thus modifying the behavior of the original operation by adding, or skipping, or replacing some computation (with respect to the original one) as well as forcing or softening exceptional termination. The kind (around, before, after) of an advice, the precedence order, and the way it is structured can determine if the code affected by the join point is actually executed or not. Thus we consider the following cases of coupling due to interactions altering the control flow.

A first case is when both the code at join points and the selected advices are executed. In this case the behaviour of advices is added to the one of code at join points; we say adding such kind of coupling, according to the following definition:

**Definition 3.3.1** Coupling is said adding if for each path in the control flow graph of the involved advice the computation at joint points is granted to be executed at least once.

The behaviour of an after advice is always added to the one of code at join points, while for an around advice it is added if all of its paths in the control flow contain a proceed statement. In the
case of around advices, the code at join point could not be executed if no proceed statement is the code of the advice. In this case it is as the advice ‘replaces’ the computation at join point, by replacing the original behaviour with the one encapsulated in the advice. We call replacing the coupling due to such an interaction, according to the following definition:

Definition 3.3.2 Coupling is called replacing if there are no execution paths in the control flow graph of the involved advice that contains a proceed statement: this means that the computation at join points can never take place.

Anyway, in an around advice the execution of proceed statements could be conditioned by any predicate and in this case the code at join points would be executed or not according to the truth value of the predicate, i.e. the advice behaviour could be added or replaced according to the actual path to be execute on the predicate value. We call conditioned replacing the coupling due to an interaction of this kind, according to the following definition.

Definition 3.3.3 Coupling is called conditioned replacing if there are some paths in the control flow graph of the involved advice that grant the computation at joint points, i.e. there are some proceed statements whose execution is conditioned by some predicate.

The behaviour of a before advice is added when it doesn’t terminate the execution in exceptional way (i.e. does not throw exceptions) being involved, in that case, in a conditioned replacing interaction. According to the previous definitions the coupling of an aspect due to interaction altering the control flow can be categorized as:

- adding (A);
- replacing (R);
- conditioned replacing (CR);

Since an aspect may contain more different kinds of advices, their number and the way each one can alter the control flow will determine the level of coupling of the aspect related to the alterations of the control flow, i.e. the coupling level is a function $f$ of the kinds (adding, conditioned replacing, replacing) of the advices and of the number of advices for each kind:

$$CLCF(a) = f(\alpha_2 A(a), \beta_2 CR(a), \gamma_2 R(a))$$

where:

- $CLCF(a)$: Coupling Level of the aspect $a$ related to the alterations of the Control Flow of the base system
- $A(a)$: total number of added advices in $a$
- $CR(a)$: total number of conditioned replacing advices in $a$
- $R(a)$: total number of replacing advices in $a$
- $\alpha_2$, $\beta_2$, $\gamma_2$: the different weights by which the occurrences of each kind of advice are to be evaluated according to an established ordered scale of values.

3.4 Coupling due to interactions altering the state of an object

Alterations in the state of an object are due to the data flow at join points and the way advices can modify the data they can access at join points. Anyway, these alterations are the typical ones we can observe in usual inter-procedural data flow among the ‘traditional’ components (methods, functions, etc.) of a system when a component calls another one. But in the case of the aspects there is not an explicit transfer of control and data by a call statement. Indeed, the control and data flow transfer between an operation and an advice takes place automatically and implicitly at join points matching the corresponding pointcut expression, thus it could be not well understood and known which data the triggered advice can have access to, as well as the way the advice (eventually) modifies them.

Our interest is about the coupling due to interactions that alter the data corresponding to fields of objects because any change of the value of an object’s field can affect the overall state of the object and then its behaviour. With respect to the fields of an objects, an interaction involving an advice adv, with an associated pointcut designator pcd and with respect to a join point shadow jp, originates coupling that is said independent, observing, or conflicting according to the following definitions:

Definition 3.4.1 Coupling is called independent if the involved advice adv does not refer any field belonging to the context of the operation associated to the matched join point;

Definition 3.4.2 Coupling is called observing if the involved advice adv just only uses some fields, while does not define any field, belonging to the context of the operation associated to the matched join point;

Definition 3.4.3 Coupling is called conflicting if the involved advice adv defines (or re-defines) some fields belonging to the context of the operation associated to the matched join point.

According to the previous definitions the coupling of an aspect due to interactions altering the state of an object can be categorized as:

- independent (I);
- observing (O);
- conflicting (C);

Since an aspect may contain more advices each one affecting the state of an object in a different way, the number of the advices and the way they can alter the state of an object will determine the coupling level of the aspect, i.e. the level is a function $f$ of the number of the advices and of the way they can alter the state of an object:

$$CLSO(a) = f(\alpha_3 I(a), \beta_3 O(a), \gamma_3 C(a))$$

where:

- $CLSO(a)$: Coupling Level of the aspect $a$ related to the alterations of the State of an Object of the base system
- $I(a)$: total number of independent advices in $a$
- $O(a)$: total number of observer advices in $a$
- $C(a)$: total number of conflicting advices in $a$
- $\alpha_3$, $\beta_3$, $\gamma_3$: different weights by which the occurrences of each kind of advice are to be evaluated according to established ordered scale of values.

3.5 Evaluating the Coupling Level of an Aspect

The total level of coupling for an aspect will be a function of the ones related to the structural alterations, to the control flow alterations, and state variables data flow alterations:
model. The figure CSST stands for “Coupling due to Static kind of coupling.” The figure 6 summarizes the proposed metric number of interactions existing in an aspect that originate for all the weights, i.e., each coupling level is given just by the number of interactions existing in an aspect that originate that kind of coupling. The figure 6 summarizes the proposed metric model. In the figure CSS stands for “Coupling due to Static Structure” alterations, CCF for “Coupling due to Control Flow” alterations, and COS for “Coupling due Object’s State” alterations.

### Figure 6: The proposed metric model

\[
\text{CLA}(a) = f(\alpha_1 \text{CLSS}(a), \beta_1 \text{CCF}(a), \gamma_1 \text{COS}(a))
\]

where:
- \(\text{CLA}(a)\): is the total coupling level for the aspect \(a\);
- \(\text{CLSS}(a), \text{CCF}(a), \text{COS}(a)\) are the coupling levels discussed in the previous subsections
- \(\alpha_1, \beta_1, \gamma_1\): the different weights by which the computed coupling level are to be considered.

The weights \(\alpha_1, \beta_1, \gamma_1\) (as well as the weights in the previous expressions for \(\text{CLSS}, \text{CCF}, \text{COS}\)) are to be still defined. This will require empirical experiments to be able to understand and define their values (or range of values). At today, in our first experiments and case studies, we have assumed a value equal to 1 for all the weights, i.e., each coupling level is given just by the number of interactions existing in an aspect that originate that kind of coupling. The figure 6 summarizes the proposed metric model. In the figure CSS stands for “Coupling due to Static Structure” alterations, CCF for “Coupling due to Control Flow” alterations, and COS for “Coupling due Object’s State” alterations.

### 4. A CASE STUDY

The proposed metric model has been used to identify and evaluate aspects coupling in the AJHotDraw v0.3 system. The analyzed system was made of 216 types, 31 of them were interfaces and 10 aspects; the classes in the contrib packages were not considered in the case study. Static source code analysis was performed to identify and classify the kinds of couplings among the aspects and the components of the base system. In particular, the Interprocedural Aspect Control Flow Graph (IACFG) [3,13] was used to represent and analyze the system.

In the analyzed version of the system three crosscutting concerns are implemented by means of 10 aspects as follows:
- Design by Contract concern (1 aspect)
- Figure Selection concern (3 aspects)
- Persistence concern (6 aspects)

The concerns were analysed and their contribution to all the kinds of structural, control flow and data flow couplings were identified and classified with respect the metric model described in the section 3. The Table 1 reports the overall results of the analysis. In the table, the rows refer the several aspects (each one identified by its name in the system) while the columns report the number of each kind of CSS, CCF, and COS coupling found for each of the considered aspects (each column header reports the acronym of the type of type of coupling it refers).

The identification of the different CSS couplings was made considering both direct (a) and indirect (b) interactions towards the elements affected by inter-type declarations, where the case (b) considers the indirect injections due to inheritance relationships, the presence of polymorphism, and interface implementation properties of OOP languages. The Table 1 is divided in two parts: the first one, in the upper part, reports the values computed considering only the direct relationships, while the second one, in the lower part, reports the values computed considering also the indirect relationships (in this case just the aspects for which resulted different values for ISS interactions have been reported in the table). Looking at the table 1 (upper part) we can note that the aspect CmndCheckViewRef, implementing the Design by Contract concern, is the one with the highest coupling. This aspect, while has no interactions with the static structure of the base system, alters both the control flow and the state of objects, being coupled by means of 18 join points, corresponding to execution regions for the “execute()” methods, with 18 classes (the concrete command classes derived from AbstractCommand). Just conditioned replacing and observing types of coupling interest this aspect. This means that the aspect doesn’t directly modify state of the base system objects (because of its observing nature) but under certain conditions it could prevent the execution of the commands (skipping the computation at join points, i.e., the execution of Commands’ “execute(...)” methods).

The aspects referred in the rows from the 2nd to the 4th implement the Figure Selection concern. The two aspects FigureSelectionSubjectRole and FigureSelectionObserverRole have the 2nd and 3rd highest coupling, mainly due to modifications to the system's static structure. Indeed they mainly acts as role providing and gen-spec alteration.

The FigureSelectionObserverRole aspect has no coupling due to interaction with the control flow and the state of objects, while the FigureSelectionSubjectRole has adding coupling with respect the control flow and conflicting ones with respect to the state of objects. If we consider the indirect relationships for the FigureSelection-ObserverRole the number of the couplings raises because the alterations it introduces are propagated to 54 classes (instead of the 5 ones directly affected).

The SelectionChangedNotification aspect only contains dynamic crosscutting constructs: it intercepts calls to methods affecting figure selection and fires the corresponding event to interested listeners.

It’s interesting to point out that all of them (only considering the direct relationships) show a coupling lower than the one reported for CmndCheckViewRef aspect. This is not coherent with the fact that both apply to several places in the system: in the case of the figure selection concern the interactions are hidden by the indirect propagation of implementation by means of inheritance relationships and late binding. Hence in this case indirect analysis can be more effective to reveal the actual couplings among aspects and classes. The remaining 6 aspects in the Table 1 implement the Persistence concern. All the 6 aspects show only coupling due to
interactions with the static structure of components in the base system.

![Image](image-url)

<table>
<thead>
<tr>
<th>DIRECT RELATIONSHIPS</th>
<th>CSS</th>
<th>CCF</th>
<th>COS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPECTS</td>
<td>CR</td>
<td>R</td>
<td>I</td>
</tr>
<tr>
<td>CmdCheckViewRef</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FigureSelectionSubjectRole</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FigureSelectionObserverRole</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SelectionChangedNotification</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentAttributeFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentCompositeFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentDrawing</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentImageFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentTextFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(a) only direct interactions considered

<table>
<thead>
<tr>
<th>INDIRECT RELATIONSHIPS</th>
<th>CSS</th>
<th>CCF</th>
<th>COS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPECTS</td>
<td>CR</td>
<td>R</td>
<td>I</td>
</tr>
<tr>
<td>FigureSelectionObserverRole</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentAttributeFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentCompositeFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentDrawing</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentImageFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PersistentTextFigure</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(b) also indirect interactions considered

Table 1: Evaluations of each type of interactions

By looking at the kind and levels of coupling of an aspect (CLA), developers are much more aware of the real impact of the aspects on the overall system. Moreover different kinds of couplings are very useful to quickly understand the different kinds of the aspects’ behaviours. For instance developers can find those aspects that act as observing (or as independent) with respect to the control flow and adding with respect to data flow: such aspects even if highly coupled with the base system, are not performing alteration on it and are easier to understand and to modify since they adds new behaviours that are orthogonal to existing ones. This happens, for instance, for the SelectionChangedNotification aspect (belonging to the Event Handling concern) that has a medium level of coupling for both direct and indirect cases: in this case a developer working on the aspect knows that it cannot alter the behaviours of the base system (hence he can focus only on the aspect code). Conversely developers can also identify those aspects that cooperate or even override existing behaviours in many places (directly or indirectly affecting classes, methods or reading/writing classes’ fields). In modifying or comprehending such aspects it’s crucial to evaluate the impact on the base system. This happens, for instance, for DesignByContract aspects and for the Observer role of the Event Handling concern: as we can see both such aspect scored a high degree of coupling due to mainly complex alterations for both structural (RP, GSA) and behavioural (CR, C).

In order to find a (first) relationship among the the type of the coupling an aspect introduces in a system and the complexity of the aspect itself, we asked to some software engineers to analyse the AHotDraw system with the aim to understand the three crosscutting concerns implemented by the 10 aspects and to indicate which ones they considered to be the most difficult to comprehend. The software engineers found the aspects showing a coupling due to interactions with the dynamic behaviour (i.e. CLCF and CLSO) introduce in the system a greater complexity (hindering comprehension) than the ones interacting with the static structure (CLSS). From this point of view the identification and classification, considering only the direct interactions, gives better results since aspects that interact with the static structure that indirectly affects a lot of classes (like the PersistentFigure aspect) are still considered as not introducing too much complexity.
The fact that the software engineers considered the aspects that alter the dynamic behavior the ones introducing a complexity greater than the one altering the static structure, suggests that the different types of coupling affect in different ways the system comprehension, and in particular the CLCF and CLOS interactions seem to be the ones making harder the comprehension of an AO system. Similar more experiments will contribute to define the value to assign to the weights to evaluate CLA(a).

5. CONCLUSIONS AND FUTURE WORK
While AOP supports new and powerful program structuring constructs allowing improvement in system modularity, the new types of coupling, with the base system components, it introduces can make harder the system comprehension. In this paper a metric model to classify the couplelings among aspects and classes has been proposed. The model considers the coupling due to interactions introduced by the aspects with the static structure, the control flow, and the state of objects of the base system. A first case study has shown that the proposed metric model is effective in identifying aspects that are coupled with too many classes affecting in an intrusive way both control flow of methods and object’s state. These aspects are considered to be more difficult to comprehend than the ones with lower coupling.

Future work will be devoted to better assess the metric model on real world aspect-oriented systems, larger than the one used in the case study, and to define the weights to use to differentiate the different types of coupling. Until today, this has been difficult due to the lack of available real-world medium and large aspect-oriented applications.

6. REFERENCES