A Taxonomy of Interactions Introduced by Aspects

Mario Luca Bernardi, Giuseppe Antonio Di Lucca
Research Centre on Software Technology (RCOST), University of Sannio, Italy
{dilucca, mlbernar} @unisannio.it

Abstract
Aspects have a large impact on the static structure and dynamic behaviour of the system they belong. This is due to the intrinsic intrusive nature of aspects and the woven process allowing the alteration of the structure, the control and data flow of the components of the base system in Aspect Oriented (AO) systems. Several and different types of interactions among aspects and the other components can be introduced according to the different mechanisms provided by AO Programming. These interactions can make higher the complexity of the overall system affecting its comprehension.

In this paper we propose a taxonomy to categorize these types of interactions among aspects and the components of the base system. The taxonomy can be used to understand how each type of interaction affects the complexity, and thus the comprehensibility, of the system.

Keywords: Software Quality, Program comprehension, Aspect Oriented Programming

1. Introduction
Aspect Oriented Programming (AOP) explicitly supports crosscutting concerns by means of program units called aspects [5, 8, 9] encapsulating the crosscutting code. The code in the aspects is able to change both structural (static) properties of the base system and its dynamic behavior at run time. 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 relationships among aspects and traditional modules that can define a system behavior and structure that may be largely different from the one shown by the base system (i.e. the system with no weaving). This can make the comprehension of a software system implemented by AOP more difficult than a traditional one. Indeed, while on one hand AOP allows to simplify and enhance the design and the code structure, on the other hand it could be very difficult to identify and comprehend which are the portions of code directly or indirectly affected by aspects, as well as which are the portions of code that will be actually run at a given point in the code.

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 intrusion. Each kind of interaction, and then of the alteration it causes in the base system, affects in some way the complexity of the system thus affecting the comprehensibility of the overall system.

In this paper we propose a taxonomy to categorize the ways by which aspects interact with the components of the base system. These ways have been analysed and classified; in particular interactions causing alterations of: (i) the static structure of base components, (ii) the inter-procedural control flow, and (iii) the state of objects have been considered. The main aim of the proposed taxonomy is to define a set of attributes able to characterize the interactions and dependencies among aspects and the components of the base system; this set can be used to understand how each kind of interaction affects the complexity, and thus the comprehensibility, of the system. A metric model based on the proposed taxonomy can be developed in turn to evaluate the degree of interactions/alterations introduced by aspects.

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

2. Related Work
Several papers have dealt with the complexity that aspects introduce into a system due to their interactions with the other components.

In [7], 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.

Additional categories of aspects have been defined in several works. The work in [6] 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. It’s also suggests a development methodology based on observers. In [12] is proposed a code analysis system to identify aspects that add behavior to the system without affecting the base system.

The same issues are discussed in [13]: 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, 14] extend to aspects well-known programming slicing techniques based on dataflow analysis. They are used to identify the extent of influence of an aspect on the underlying system, and to identify potential conflicts among aspects. A similar approach is proposed for Composition Filters in [11].

Finally in [10] aspect advices are represented by a state transition graph that is (model) checked in order to extend invariants proven true for the original system to the system with aspects applied to it.

While some works are starting to provide insights on invasiveness and complexities introduced by aspects, the taxonomy 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 spectrum of interactions the aspect is capable of. It provides a precise and solid starting point for reasoning about interactions among aspects and the remaining of the system.

3. The proposed taxonomy of aspect interactions

An aspect is able to interact with other components by altering mainly:

- the 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;
- the control flow, by selecting join points at which the control of the execution may be passed to an advice;
- the 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 while the others two are related to dynamic crosscutting features of AOP languages.

Each of these three main categories of interactions can cause different kinds 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

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 $t \in T$, $O(t) = \{o_1, o_2, \ldots, o_n\}$ is the set of the operations $o_i$ (i.e. methods of classes/interfaces/aspects or advices of aspects) declared in $t$, while $F(t) = \{f_1, f_2, \ldots, f_n\}$ is the set of the fields $f_i$ declared in $t$. Given an aspect $a \in A$, $Adv(a) = \{adv_1, adv_2, \ldots, adv_n\}$ is the set of advices declared by $a$, while $P(a) = \{p_1, p_2, \ldots, p_n\}$ is the set of the pointcuts designators $p_i$ declared in $a$. Given a pointcut designator $p \in P(a)$, $JP(p) = \{jp_1, jp_2, \ldots, jp_n\}$ is the set of all join point shadows $jp_i$ potentially matching $p$. For a given $p$, restrictions of $JP(p)$ to specified modular units in the system (i.e. methods, types and packages) can be defined. For instance $JP(p)|_{\text{mod}(O(t))}$ restrict the set of join point shadows, potentially matching the pointcut designator $p$, just within the operations $o$ of the type $t$.

3.2 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 [9], as well as most of the AO languages, provide the declare error statement to define access constraints.

We call access restriction an intertype declaration that defines access constraints to some elements of the base system, according to the following definition:

**Definition 3.2.1** An intertype declaration that defines access constraints to a set of code elements is called access restriction (AR). A given aspect $a \in A$ contains an access restriction for a source code element $e \in T \cup O(t) \cup F(t)$ if and only if:

1. the aspect $a$ contains a declare error statement associated with a pointcut designator $p$;

- 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 [9], as well as most of the AO languages, provide the declare error statement to define access constraints.

We call access restriction an intertype declaration that defines access constraints to some elements of the base system, according to the following definition:

**Definition 3.2.1** An intertype declaration that defines access constraints to a set of code elements is called access restriction (AR). A given aspect $a \in A$ contains an access restriction for a source code element $e \in T \cup O(t) \cup F(t)$ if and only if:

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 \( j \in \text{JP}(p) \).

The case b) is related to the usage of intertype declarations in an aspect to add operations or fields to the original code of a type \( t \). The injection of fields or operations can cause the overriding of the (eventually) existing ones (by modifying class, state and behaviour).

We call \textit{injective} an intertype declaration altering the original code of a type \( t \) according to the following definition:

\textbf{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, is called \textit{injective} (\( I \)). The “\textit{data}” and “\textit{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.

We call \textit{interface marker} an intertype declaration forcing the implementation of an interface by a class or aspect, according to the following definition:

\textbf{Definition 3.2.3} An intertype declaration compelling a type \( e \in \text{C}\cup\text{A} \) to implement the interface \( e \in I \) is called \textit{interface marker} \(^2\) (\( IM \)) of type \( t \) by interface \( i \).

A particular case is when \textit{injective} and \textit{interface marker} declarations are in the same aspect involving some of the injected methods are declared by the interface: in this case we call this set of related declarations as \textit{role providing} declarations, according to the following definition:

\textbf{Definition 3.2.4} If an aspect \( a \in \text{A} \) contains an interface marker declaration for an interface \( i \in I \) and injective declarations for some of the operations declared in \( i \), then this set of related declarations is said \textit{role providing} (\( RP \)).

The semantic difference between \textit{interface markers} and \textit{role providing} is related to the ways the aspectual components organize and structure the collaborations of classes to realize scenarios involving the crosscutting concerns they implement. To involve classes into collaborations where each class plays a well defined role, aspects use interfaces (usually private to them). Moreover aspects can leave up to each class the responsibility to fulfill the interface implementation relationship or can even provide an (eventually partial) implementation.

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 \( t \).

---

\(^2\) The term ‘\textit{interface marker}’ has not to be confused with the ‘\textit{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.

We call \textit{gen-spec alteration} an intertype declaration that alters a generalization/specialization structure according to the following definition:

\textbf{Definition 3.2.5} An intertype declaration changing the generalization/specialization relationship existing among a type \( t \in \text{C}\cup\text{A} \) its super-type \( spt \in \text{C}\cup\text{A} \) and/or its sub-type(s) \( sbt \in \text{C}\cup\text{A} \) is called \textit{gen-spec alteration} (\GSA{}).

3.3 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. The kind (around, before, after) of an advice, the precedence order, and the way it is structured can determine if the code affected at the join point is actually executed or not. Thus we consider the following cases of 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 \textit{adding} such an interaction according to the following definition:

\textbf{Definition 3.3.1} An interaction is said \textit{adding} (\( A \)) 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 \textit{after} advice is always added to the one of code at join points, while for an \textit{around} advice it is added if all of its paths in the control flow contain a \textit{proceed} statement.

In the case of \textit{around} advices, the code at join point could not be executed if no \textit{proceed} statement is the code of the advice. In this case it is as if the advice ‘replaces’ the computation at join point, by replacing the original behaviour with the one encapsulated in the advice. We call \textit{replacing} such an interaction, according to the following definition:

\textbf{Definition 3.3.2} An interaction is called \textit{replacing} (\( R \)) if there are no execution paths in the control flow graph of the involved advice that contains a \textit{proceed} statement: this means that the computation at join points can never take place.

Anyway, in an \textit{around} advice the execution of \textit{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 \textit{added} or \textit{replaced} according to the actual path to be execute on the predicate value. We call \textit{conditioned replacing} an interaction of this kind, according to the following definition:

\textbf{Definition 3.3.3} An interaction is called \textit{conditioned replacing} (\( CR \)) 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. it does not throw exceptions) being involved, in that case, in a conditioned replacing interaction.

### 3.4 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. 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 interactions that alter the values of 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, is said independent, observing, or conflicting according to the following definitions:

**Definition 3.4.1** An interaction is called independent (I) 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** An interaction is called observing (O) 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** An interaction is called conflicting (C) if the involved advice adv defines (or re-defines) some fields belonging to the context of the operation associated to the matched join point.

The figure 1 summarizes the proposed taxonomy. In the figure ISS stands for “Interactions altering the Static Structure”, ICF for “Interactions altering the Control Flow”, and IOS for “Interactions altering Object’s State”.

### 4. A case study

The proposed taxonomy has been used to classify the interactions among aspects and classes in the AJHotDraw v0.3 system. The analyzed system was made up 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 interactions among aspects and the components of the base system. In particular, the Interprocedural Aspect Control Flow Graph (IACFG) [3,4] 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 types of structural, control flow and object state interactions identified and classified with respect the taxonomy described in the section 3.

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 ISS, ICF, and IOS interactions found for each of the considered aspects (each column header reports the acronym of the type of interaction it refers).

The identification of the different ISS interactions 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.

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 upper part of the table 1 we can note that the aspect CmdCheckViewRef, implementing the Design by Contract concern, is the one with the highest...
The total number of interactions. This aspect, while has no interaction with the static structure of the base system, alters both control flow and the state of objects, due to the interactions at 18 join points corresponding to execution regions for the methods `execute()` of 18 classes (the concrete command classes derived from AbstractCommand).

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

The remaining 6 aspects in the table 1 implement the Persistence concern. All the 6 aspects show only interactions with the static structure of the base system. The one with the highest number of interactions is the PersistentFigure aspect that injects into Figure class hierarchy read/write methods also forcing the Storable interface for all figures. While considering just the direct relationships of this aspect the number of the interactions is small, when the indirect interactions are considered the value becomes very high since the Storable role is injected on an entire complex hierarchy: it is hard in this case to understand if and where binding interactions arise in the system.

In order to find any relationship between the number and the type of the interactions an aspect introduces in a system and the complexity of the aspect itself, three software engineers were asked to analyse the AJHotDraw system with the aim to understand the three crosscutting concerns implemented by the 10 aspects and to make a ranking of the 10 aspects with respect to the complexity and comprehension difficulty due to the interactions introduced by the aspects themselves. Table 2 reports the final ranking.

All the 18 interactions are *conditioned replacing* ones with respect to the control flow and *observing* ones with respect the state of objects. This means that the aspect doesn’t directly modify the state of the base system objects but under certain conditions it could prevent the execution of the commands (skipping the computation at join points).

The aspects referred in the rows from 2 to 4 implement the Figure Selection concern. The two aspects `FigureSelectionSubjectRole` and `FigureSelectionObserverRole` have the 2nd and 3rd highest total number of interactions, mainly due to the interactions with the static structure of the base system. Indeed they mainly acts as *role providing* and *gen-spec alteration*. The `FigureSelectionObserverRole` aspect does not interact with the control flow and the state of objects, while the `FigureSelectionSubjectRole` has *adding* interactions with respect the control flow and *conflicting* ones with the state of objects. If we consider the indirect relationships for the `FigureSelectionObserverRole` the number of the interactions raises because the alterations it introduces are propagated to 54 classes (instead of the 5 ones directly affected).

### Table 1: Number of each type of interaction identify for each aspect

<table>
<thead>
<tr>
<th>DIRECT RELATIONSHIPS</th>
<th>ISS</th>
<th>ICF</th>
<th>IOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CmdCheckViewRef</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FigureSelectionSubjectRole</td>
<td>0</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>ISS</th>
<th>ICF</th>
<th>IOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASPECTS</td>
<td></td>
<td></td>
<td></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 2: Software engineers’ complexity ranking of the aspects in AJHotDraw

<table>
<thead>
<tr>
<th>ASPECTS</th>
<th>ISS</th>
<th>ICF</th>
<th>IOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CmdCheckViewRef</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>FigureSelectionSubjectRole</td>
<td>0</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>

The main consideration we can made looking at the table 2 is that the software engineers found the aspects interacting with the dynamic behaviour introduce a greater complexity in system (hindering comprehension) than the ones interacting with the static structure. From this point of view the identification and classification considering only the direct interactions gives better results since aspects that interacts with the static structure that *indirectly* affects a lot of classes (like the PersistentFigure aspect) are still considered as not introducing too much complexity. However taking into account the structural indirect interactions the aspects are classified as more complex ones since they introduce alterations on a higher number of classes, thus they are ranked as more complex than actually they are.
The results in table 2 show that the software engineers classified the aspects implementing the Design by Contract and Figure Selection concerns as the hardest one to comprehend due to the number and the types of interactions.

As said, in the evaluation of the software engineers the aspects that alter the dynamic behavior were considered introducing a complexity greater than the one altering the static structure. This suggests that the different types of interactions affect in different ways the system comprehension, and in particular the ICF and IOS interactions seem to be the ones making harder the comprehension of an AO system.

5. Conclusions and Future Work

While Aspect Oriented Programming supports new and powerful program structuring constructs allowing improvement in system modularity, their interactions with the base system can make harder the system comprehension. In this paper a taxonomy to classify the interactions between aspects and classes has been proposed. The model considers interactions introduced by the aspects with the static structure, the control flow, and the state of objects of the base system. The taxonomy allows to identify which are the kinds of interactions that largely affect the comprehension of the system. A first case study has shown that interactions altering the control flow, and the state of objects are those ones making harder the system comprehension.

Future work will be devoted to develop a metric model based on the proposed taxonomy to assess the complexity of an aspect due to the interaction it establishes with the components of the base system.

References


