

Managing semantic interference with aspect integration contracts

Bert Lagaisse, Wouter Joosen, Bart De Win
Distrinet, Department of Computer Science, K.U.Leuven

{bertl,wouter,bartd}@cs.kuleuven.ac.be

Abstract

Programming languages and environments that support AOP lack expressive power to manage the interference between components and aspects. We illustrate this problem in an example and identify the needed expressive power. We propose aspect integration contracts to fill the gap. These contracts specify the permitted interference between an aspect and a base component. We discuss the impact of our observations on language design, and the impact of a solution on software engineering properties.

1 Introduction

Through the different stages of the software development process, separation of concerns [4] is one of the important software engineering principles. But the consequence of separating concerns is that we have to compose the resulting modules into a complete software system that matches all the requirements of the stakeholders. This composition reveals essential challenges.

When composing multiple software components - i.e. modules that encapsulate specific concerns (such as classes, packages, components or aspects), one of the important issues is managing interference. One needs to express and control which modules may use and affect each other. In an object-oriented or component-based software design, each artefact can be equipped with a contract that specifies the provided functionality and the needed (required) functionality that describes the dependencies of a component on other components. Contracts in object-oriented systems have been introduced by Bertrand Meyer [5]. An extension for component-based systems has been described by

Andreas Raush in [7]. In principle, correct behavior can be guaranteed if a component has been designed defensively and if it strictly implements its contract. We observe that the state-of-the-art notion of a contract is no longer sufficient in an aspect-oriented programming [1] environment.

When a class or component is composed with aspects by means of superimposition, there is no expressive power to specify a number of interactions in contracts of the above mentioned type. Assume an aspect A is bound to a component C ¹.

1. The aspect should specify what it requires from component C and possibly from other software components.
2. The aspect also needs to specify in which way it affects the component C and the functionality it provides (if applicable).
3. The specification of component C must express which interference is permitted from certain (types of) aspects.

Aspect languages lack this type of expressive power. The result is a problem of uncontrolled semantic interference that can endanger the integrity of component software as existing contracts might break because of aspect binding, either intentionally or (and most often) unintentionally.

An example will illustrate that this problem can have various kinds of consequences, such as undesirable exposure of data, undesirable modification of data, undesirable exposure of behavior and undesirable modification of behavior of components, all resulting in some kind of contract breach.

The paper is structured as follows. We elaborate on the problem of uncontrolled semantic interference

¹In the context of superimposition, we may expect an aspect to be bound to multiple components as this essentially reflects a crosscutting concern, but this is not relevant at this stage of the discussion.

rence in section 2. In section 3, we introduce a solution based on aspect integration contracts (AICs). We also sketch language constructs that can express such contracts. In section 4, we discuss the impact of this type of language extensions on comprehensiveness, predictability and semantic interaction. We also sketch the impact on other software engineering properties. We discuss the status of this work in section 5. Then we conclude.

2 Problem Statement

Before we elaborate on the problem, we will first explain some terms used in the paper. We use Clemens Szyperski's widely accepted definition of a component: *A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties*[8]. This definition implies that a component must be well separated from its environment and from other components.

An aspect is a software module that encapsulates a crosscutting concern. The added-value of using an aspect is improved localization in the sense that behavior that implements the requirements that are related to a crosscutting concern is encapsulated in a cohesive module. Superimposition is the operation of coupling or binding such a module to multiple other modules (hopefully components) in the software system. An aspect thus includes behavior and coupling.

One can question whether an aspect typically is an artifact that should match the above mentioned definition of a component. We believe this should be the case in many if not most circumstances² but this is not crucial for our problem statement: our focus is on ensuring that adding aspects to a working application that is composed of traditional, state-of-the-art components will not interfere with the contractual obligations of the components that constitute the core application. Such interference is what we mean by *uncontrolled semantic interference*.

Further on in this section, we will illustrate our understanding of uncontrolled semantic interference by sketching an example. We use Java

²We do not intend to elaborate on this matter in this paper.

to describe the application semantics and pseudo-Aspect/J to sketch the role and impact of superimposition. Our motivation is independent of the languages that we use to illustrate the problem though.

Uncontrolled semantic interference and language design Even if a component is designed defensively: composing it with aspects can result in a component that does no longer meet all contractual obligations.

Superimposition of an aspect is typically described using a concept similar to pointcut definitions [2]. Join points are very powerful but they do not take in account the scope qualifiers of the members of the base component. These qualifiers are set by the developer of the base component, often assuming that the component is typically used in a plain object-oriented environment. Scope qualifiers are used to enforce a part of the contract of the base component. Data and behavior they do provide to other (client) components are *public*.

In general, an object-oriented programming environment is characterized by objects that encapsulate state and behavior. Behavior is triggered by messages that are sent between objects. Scope qualifiers like *public* and *private* are enforcing these basic characteristics in state of the art object-oriented programming languages. In languages such as Eiffel [6], it is even possible to specify to which other entities certain data and behavior is exposed.

Another characterization of class based object orientated languages is inheritance, where subclasses must have access to some of the members of their (parent) super classes. Scope qualifiers like *protected* and *private-protected* permit a relatively fine-grained expression and enforcement of this kind of access.

Composing aspects through superimposition does pose some new problems. We can question if -for example- access to private members of a class should be enabled for an aspect. As illustrated further, that kind of interference may be necessary in some cases, but definitely not always and it increases the risk of introducing errors.

If components are now composed in an environment with a new kind of dependencies, do they need to know about these dependencies? We believe that the answer to this question is clearly affirmative.

We need a new kind of expressive power to specify what kind of dependencies components accept for this new kind of first class citizens: aspects. It is uncertain however, whether this leads to extra language features for a number of reasons:

- It has appeared to be extremely difficult to define orthogonal support in object-oriented languages to support both encapsulation and inheritance[12].
- We do not intend to add scope qualifiers³ in an ad hoc way.

We believe that we should study this problem in a more systematic way, probably based on a suite of examples, in order to validate the precise needs for expressive power. We describe such an example in the next subsection.

An example. Suppose that an entity person is the key abstraction in a software system. A person has a certain social security number, a name and a birth date. Because of privacy reasons a person object should never expose its age or birth date. It is necessary to know whether a person is an adult, hence the birth date is a necessary property in the software system. The birth date also needs to be stored in a database. If the persistence feature is delivered as an aspect, then the persistence aspect needs access to the birth date property, while other software entities should not be able to access this property.

The software system will also require authentication of a user for calling the *setName* method of the person component. Changing the name of a person is only permitted to a privileged user. One of the public methods of *Person* is *authenticate*, which is used to verify the user name and password. A hash of the password is stored on the hard drive and the authentication method compares the stored hash with the hash of the password that has been entered. The authentication part of *Person* ensures that the password (in clear text) is never exposed to the rest of the software system.

This example is illustrated below in Java and pseudo-Aspect/J [2]:

```
public class Person{
private Date birthDate;
```

³E.g. similar to *friend* in C++[13]

```
private String ssn , name;
public Person(
    String ssn ,
    String name,
    Date birthDate){
//initialization
}
private void setSsn(String ssn){}
private void setBirthDate(Date bd){}
public void setName(String name){
    <get users credentials>
    if(authenticate(user , passwd))...
}
public String getSsn () {...}
public String getName () {...}
private Date getBirthDate {...}

public boolean isAdult (){
// derived from birthDate
}
//can also be used by other
//components
public boolean authenticate(
    String username,
    String passwd){
//verify username and password
}
```

```
Aspect PersonPersistence{
//on constructor exection
after(Person p):
    execution (Person.new(...))
    && this(p){
//store internal state
store(p.ssn , p.name, p.birthDate);
}
after(Person p):
    execution(* set*(...))
    && this(p){
//store internal state
store(p.ssn , p.name, p.birthDate);
}
private store (...){...}

public void refetch(
    String ssn ,
    Person p){

    p.ssn = ssn;
//fetch and set name
p.name = ...;
//fetch and set birthdate
p.setBirthDate (...);
}}
```

The contract of the person class will certainly specify that it provides the *isAdult* functionality. Towards other modules the birth date property will remain hidden. However, this property has to be exposed towards the persistence aspect, because that aspect requires *person* to expose encapsulated data that needs to be persistent. Therefore the contract of the persistence aspect must specify that it requires access to the encapsulated (i.e. private) state of a person object.

Suppose the data of a person object is updated in the database by an other program. Then a possible

solution is to overwrite the state of the object with the data in the database. Therefore the *refresh* operation is provided. To modify the internal state of the object, the *Persistence* aspect needs write access to the private data members of the object, or it has to use the private setters. Both alternatives are supported in the example.

The software system also includes a logging feature. A logging aspect records all public method calls. The arguments of the calls are also logged. If the logging aspect is composed with the base application, then the contract of the *authenticate* method is broken. Though unintentionally, the username and plain password can be logged.

```
Aspect Logging{
before():
execution(public * *.*(..)){
    String info = <method info>;
    System.out.println(info);
}}
```

In fact, *Person* must specify that it expects that no other parts of the software system send the password (in clear text) to any output medium. But the logging aspect will require access to the password argument of the authentication method. So granting this kind of access must depend on what the logging aspect will do with the password value. To resolve this type of interference the logging aspect must also specify in some form that it will write the password to the hard disk. That would enable detecting and avoiding the risk of the logging aspect breaking the contract of the authentication aspect.

Identifying specific problems. Based on the example, we identify some threats that compromise the integrity of a component when aspects are composed without controlling interference. We classify the interference that we need to manage. The solution is addressed in the next section.

First we can distinguish data related interference and behavior related interference. There's also a distinction between exposure of class members and modification of class members. This brings us to four possible ways of interference that we will try to control.

Data exposure. Encapsulation of data members controls undesirable exposure of the data to other objects. Aspects sometimes need access to this encapsulated data - but not always. This has

been illustrated with the *Persistence* aspect that should be able to access *birthDate*.

Even if data access is granted, then it is still necessary to control the flow of the data because other software components may not be permitted to gain access to that information. This has been illustrated by discussing the interference between the *Authenticate* method and the *Logging* aspect.

Data modification. An obviously more serious threat is undesirable modification of data, which can lead to misbehavior of the system. In the example data modification is illustrated in the *Persistence* aspect. This aspect needs for instance write access to *name*.

Behavior exposure. Certain behavior of a component needs to be exposed to an aspect. Specifying and controlling which behavior may be called by the aspect is also a necessity. For Example, the *Persistence* aspect needs access to *setBirthDate*.

Behavior modification. Intercepting method calls can lead to a breach in the control flow of the software and compromise critical algorithms. Therefore a software component has to be able specify whether modifications to its internal control flow are acceptable, and if so which one. For instance, interception of *authenticate()* is not acceptable for the *Logging aspect*.

3 A Solution based on Integration Contracts

A solution for managing complex interference is far from trivial. In this section we first sketch *aspect integration contracts*. Afterwards - by means of a simple example - a concrete format of such a contract is presented.

Aspects and contracts. We envisage an extension to the typical contracts for component composition. Such a contract contains specifications of provided functionality and of dependent components [7]. Provided functionality is also characterized by stating effects or results.

Assume an aspect A is bound with component C. The contract elements that we envisage are threefold:

1. A first part describes what an aspect requires. This part includes two elements
 - (a) what the aspect A requires from the component C it is bound to.
 - (b) what the aspect A requires from other components or from the environment.
2. A second part describes the functionality and effects of the aspect:
 - (a) for the clients of the component C
 - (b) for other entities, for instance aspects that are part of the same collaboration[9].
3. The third and most specific part is an *extension of the contract for component C*
 - (a) to specify the interference that is *permitted* by the base component,
 - (b) and (if needed) to specify further conditions to these permissions, i.e. the base component may express specific requirements about the behavior of an aspect before permitting interference.

Aspect requirements. Aspects must specify their requirements so that potential interference with other components can be identified. The developer of an aspect specifies what is needed. In this contract one needs a fine-grained specification expressing

1. component data exposure, the requested read access to encapsulated data of the base component;
2. component data modification, the requested write access to encapsulated data of the base component;
3. component behavior exposure, the need to call certain behavior of the base component;
4. component behavior modification, the methods of the base component that will be intercepted.

Aspect functionality and effects. The provided functionality of the aspect is required to express the results and effects generated by an aspect.

An important issue is the dataflow that the aspect will expose. For example, what is delegated or exposed to other components? Such a contract element can be specified in a general purpose notation, or alternatively one may prefer a domain specific language to express its effects. This will be discussed further on.

Specifying permitted interference. The third and most important element in the context of aspect integration is the specification of the interference that is permitted by a component towards aspects. We call this an aspect integration contract. Obviously this part of the component's specification is directly related to the aspect requirements discussed above. So in this part the component will grant interference in terms of data exposure or modification, and in terms of behavior exposure or modification.

Notice that component permissions can be condition to aspect behavior. The last part of the component specification expresses certain conditions in terms of the behavior of aspects the component may be integrated with. If the effects generated by such an aspect do not break these requirements of the component, then that aspect will be granted interference as discussed above. Obviously this part of the contract is directly related to the specification of aspect functionality.

A format for aspect integration contracts. As mentioned above, the notion of a typical component contract is affected in three ways:

1. aspect requirements must be specified;
2. aspect functionality and effects must be specified;
3. the aspect integration contract itself specifies interference that is permitted by the base component, possibly subject to aspect behavior.

From the component view, the contract extension concerning aspect integration consists of two sections: permitted interference and conditions in terms of aspect behavior. Interference is expressed in terms of the different parts of the component interface or class.

In the illustration below, parts are class members, i.e. methods or instance variables. So the additional part of the contract can contain the following interference rules related to the given class members:

1. Exposure of an instance variable, makes the variable readable to the aspect.
2. Exposure of a method, permits the aspect to call that method.
3. Modification of an instance variable, permits the aspect to write to the variable.
4. Modification of a method, permits the aspect to intercept calls to that method and potentially to alter the control flow of the base component.

A simple example is given in the code snippet below. We currently limit the possibilities for identifying aspects to the name of the aspect. For example, even if the persistence aspect does not expose the encapsulated data of the person to other components, reading the encapsulated data is permitted.

```
//defined aspects: Persistence, Logging
//defined classes: Person.
Component Person
IContract(Persistence){
Permit:
expose name, birthDate;
modify public * set*(..);
}
```

The aspects contract consists of two parts. The requirements from the base components and the results or effects. The contract below shows required behavior of the *Person* component.

```
Aspect Persistence
Require from Person{
expose name, birthDate;
modify public * set*(..);
}
```

The next listing summarizes a syntax for aspect integration contracts. We also show a similar syntax to specify aspect requirements.

```
<AIC_Component> ::=
    IContract ( <aspects> ) { <body> }
<aspect> ::=
    aspect_name | aspect_name,<aspects>
<body> ::=
    permits: <permission>*
<permission> ::=
    <action> <members> ;
<action> ::=
    expose | modify
<members> ::=
    <member> | <member>, <members>
```

```
<member> ::=
    own_method_signature | own_class_var
```

```
<Requirements_Aspect> ::=
    Require From class_name { <body> }
<body> ::=
    require : <require_clause>*
<require_clause> ::=
    <action> <members> ;
<action> ::=
    expose | modify
<members> ::=
    <member> | <member>, <members>
<member> ::=
    method_signature | class_var_name
```

In ongoing work we also examine more types of join points to make the possible interference rules potentially more finely grained. Exposure of arguments when intercepting method calls is but one example.

4 Impact on software engineering properties

First we sketch how aspect integration contracts can impact comprehensibility, predictability and evolvability. Then we briefly discuss some other software engineering properties.

Comprehensibility. Aspect integration contracts extend the component contract, thus describing a set of assumptions the component programmer made - and that are often not documented in a state-of-the-art programming environment. We believe that this is a clear win in terms of comprehensibility.

Predictability. Aspect integration contracts somehow reduce the degrees of freedom for the aspect programmer. This may seem a limitation in terms of flexibility for the code developer at first sight, but it enables to foresee the potential compositions that are anticipated in the base application. In a way, one can consider this to be an increased degree of predictability in terms of composition scenarios - though we are aware of a much broader notion of the term predictability.

Evolvability. We do not clearly see the impact on evolvability: on the one hand, aspect integration contracts may restrict the potential evolution of a software component as it is another specification one needs to be upward compatible with. On the other hand, it will reduce the cost of generating

the next version as testing and quality assurance should be less cumbersome.

Semantic interactions. Clearly, the simple notion of an aspect integration contract as sketched in the former section is but a first step towards the adequate management of semantic interactions. In our opinion, a key challenge is to define domain specific languages to express requirements and effects of components. A language to describe the component architecture and interaction could give meaning with stronger semantics to the specifications in the examples.

Other properties. Aspect integration contracts have impact on other software engineering properties like correctness, integrity, verifiability, robustness, extendibility, integrity and maintainability.

Correctness can be defined as compliance of a component to its contract. But there's a lack of expressive power to completely specify the contract of a component in terms of the permitted interference by a special kind of components: aspects. Aspect Integration Contracts tackle this problem. When this new kind of specifications in the components contract can be enforced through language support, assumptions about the correctness of a software component can be extended to the level of aspect interference.

As mentioned earlier, the contract of a component can be breached in its implementation when used in a composition with aspects. Even if the implementation was correct. This is a threat that compromises the integrity of a component. If enforcement of aspect integration contracts is supported, the integrity of a component can be ensured. This also increases the robustness of the software system.

Verifiability is supported for the interference between the aspect and the component. The aspect cannot interfere with an aspect in way that is not specified in the *permission section* of the component contract

Maintainability increases because of the increased comprehensibility. At first sight, extendibility seems like a property on which aspect integration contracts have negative influence. The possible extensions are limited to what the contract permits. But these limitations restrict the extensions in a way they are not able to endanger the properties mentioned above: correctness, verifiability, integrity and maintainability.

5 Discussion

New expressive power will specify what kind of dependencies components accept for entities that implement a crosscutting concern. We have introduced the notion of an integration contract to tackle this problem. We are currently studying the overall challenge in a more systematic way, in order to validate the precise needs for expressive power. Also, aspect integration contracts are part of a renewed notion of design-by-contract that anticipates the relevance of AOSD. A more extensive description of our work in this area can be found in [14].

An obvious consequence of the analysis presented in this paper is that aspect should be treated as components indeed. They should be units of composition with contractually specified interfaces and explicit context dependencies only. In fact, the requirements that have been listed in section 3 cover two essential elements. First, aspect functionality and aspect requirements should be clearly specified: this makes aspect specification subject to design-by-contract. Secondly, any component should be equipped with an integration contract that considers aspect interference.

Nowadays AOP has not sufficiently considered aspects as units of composition with contractually specified interfaces and explicit context dependencies only. We refer to relatively well-known aspect oriented languages: Aspect/J [2] and Hyper/J [3].

A word on Aspect/J. Aspect/J [2] offers the notion of privileged aspects. Normal aspects do not have access to private members of a Java class. If the aspect developer declares the aspect *privileged*, then the aspect will be permitted access to all private members of all classes. First of all, this means that the developer of the class has no control of possible interference. The decision is completely on the side of the aspect developer. Also the lack of granularity is obvious: the privileged aspect has access to *all* private members of *all* classes, though this is probably unnecessary and possibly leads to uncontrolled side effects and faulty behavior.

A word on Hyper/J. When integrating multiple concerns in Hyper/J [3], one can specify whether a given class is *composable* or *uncomposable*. Uncomposable classes can not be modified by the entities that implement other concerns. This declaration of a class (being composable or not) is part of the specification of the integrated application,

and not localized at the level of an individual component (i.e. hyperslice). We believe that this is a good approach. However, when a class is composable it can be affected by all other concerns in the hypermodule. Again, we observe a lack of expressive power.

Some programming languages have (without referring to aspects as such) built-in support to express in some way interaction between unrelated components. The notion of *export* in Eiffel[6] is a relevant example.

Finally, we mention[11]. This work addresses to a large extent the same problem. However, the solution is tightly coupled to AspectJ specifics and to our understanding, not considering this problem from a component perspective.

6 Conclusion

It is obvious that in the current state of the art AOP technologies, the base components are not able to specify or restrict the impact of superimposition. This is caused by a lack of expressive power to control the interference between base components and aspects. To solve this uncontrolled interference we have proposed a solution which introduces the notion of *aspect integration contracts*. These contracts specify the permitted interference between an aspect and a base component, possibly conditional to the specific behavior of the aspect. We have sketched a simple notation for aspect integration contracts and we have discussed the potential of such a feature.

References

- [1] Kiczales, G. et al. *Aspect-Oriented Programming*. In Proc. of ECOOP, Springer-Verlag (1997).
- [2] Gregor Kiczales, et al. *An Overview of AspectJ*. In Proc. of ECOOP, Springer-Verlag (2001).
- [3] Harold Ossher, Peri Tarr. *Using multidimensional separation of concerns to (re)shape evolving software*. Communications of the ACM, v.44 n.10, p.43-50, Oct. 2001.
- [4] E.W. Dijkstra. *A Discipline of Programming*. PrenticeHall International, 1976.
- [5] Bertrand Meyer. *Design by contract: building bug-free O-O software*. In Hotline on Object-Oriented Technology, volume 4, Number 2, December 1992, pages 4-8.
- [6] B. Meyer. *Eiffel the language*. Prentice Hall, 1992.
- [7] Andreas Rausch, *Design by Contract + Componentware = Design by Signed Contract*. Journal of Object Technology, In Proc. of Tools Usa, 2002.
- [8] Clemens Szyperski, *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, 1998.
- [9] E. Truyen, et al. *Dynamic and Selective Combination of Extensions in Component-Based Applications*. In Proceedings of the 23rd International Conference on Software Engineering (ICSE'01), Toronto, Canada, May 2001.
- [10] Erik Ernst. *Family Polymorphism*. In Proceedings ECOOP 2001, LNCS 2072, pages 303-326, Budapest, Hungary, June 2001. Springer-Verlag.
- [11] David Larochelle, et al. *Join Point Encapsulation*. In AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies, Boston, Massachusetts, March, 2003
- [12] Mira Mezini, *Dynamic object evolution without name collisions*, in European Conference on Object-Oriented Programming, Jyvaskyla, Finland, 1997, pp. 190-219, Springer Verlag
- [13] Stanley B. Lippman, *C++ Primer*, Addison-Wesley, 1991.
- [14] Bart De Win and Wouter Joosen. *Design by contract for AOSD* In Internal Report, Department of Computer Science, K.U.Leuven, Belgium, January 2004.