Towards Reusable Aspects: the Callback Mismatch Problem

Maarten Bynens, Dimitri Van Landuyt,
Eddy Truyen and Wouter Joosen

DistriNet, Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Leuven, Belgium

{maarten.bynens,dimitri.vanlanduyt,
eddy.truyen,wouter.joosen}@cs.kuleuven.be

Abstract

Because software development is increasingly expensive and time-consuming, software reuse gains importance. Aspect-oriented software development modularizes crosscutting concerns which enables their systematic reuse. Literature provides a number of AOP patterns and best practices for developing reusable aspects based on compelling examples for concerns like tracing, transactions and persistence. However, such best practices are lacking for systematically reusing invasive aspects.

In this paper, we present the 'callback mismatch problem'. This problem arises in the context of abstraction mismatch, in which the aspect is required to issue a callback to the base application. As a consequence, the composition of invasive aspects is cumbersome to implement, difficult to maintain and impossible to reuse.

We motivate this problem in a real-world example, show that it persists in the current state-of-the-art, and outline the need for advanced aspectual composition mechanisms to deal with this.

Categories and Subject Descriptors D.2.13 [*Software Engineering*]: Reusable Software—Reusable libraries; D.2.11 [*Software Engineering*]: Software architectures—Information hiding,Languages,Patterns

General Terms Design, Documentation

Keywords reusable aspects, invasive aspects, aspect adapter

1. Introduction

Current AOP languages and approaches often result in aspects that are tightly coupled to the base classes they act upon. For example, it is a common technique to write advice code that involves join point reflection to find out the necessary contextual information [11]. Such advice code typically hard-codes assumptions about the structure and behavior of the base classes. This has a number of negative consequences: the aspect must be maintained together with the base, which makes it difficult to develop aspects and base in parallel, and leads to fragility of aspectual composition (lack of robustness). Additionally, the resulting aspects and their compositions are very specific to the scope of one application, and thus not reusable, for example in an aspect library.

To address these problems, the current state-of-the-art provides a number of techniques and patterns that involve introducing an abstraction layer between the base and the aspect. Examples of this are pointcut interfaces [6], annotations and marker interfaces. Introducing an abstraction layer enables the design of reusable aspects, in the sense that the required interface of the aspect (the elements it needs from the base to perform its function) can be specified uniquely in terms of abstractions that are relevant in the scope of the aspect itself. For example, the required interface of a reusable *authentication* aspect would be defined in terms of aspect-specific abstractions such as the *principal*, *credentials*, etc. To compose this *authentication* aspect to the base application, the developer must implement and provide these abstractions, by mapping elements of the base application (e.g. a *customer* in a web shop) to the aspect abstractions (the principal). Because the aspect is less tightly coupled to the base application, it can be reused more easily across applications.

A common problem in the design of reusable aspects is that of *abstraction mismatch*. This occurs when the elements of the base are not fully compatible with the abstraction required by the aspect. For example, the *credentials* abstraction may consist of a password that is encoded in MD5 —meaning that the *authentication* aspect expects passwords to be provided in MD5—, while the base offers the password in plain text. The solution to this is to introduce an adapter [5] that converts the base abstraction into the aspect abstraction. In the example, the adapter would be responsible for applying the MD5 hash function to the password that is provided by the base and providing the result to the aspect.

These techniques are sufficient to realize a loose coupling between aspect and base for both *spectative* and *regulative* aspects [9]; i.e. aspects that respectively observe the base application without affecting its functionality, or observe the base application and redirect or block the thread of execution in some cases. However, there is a lack of similar patterns or solutions for *invasive aspects* that issue callbacks to the base application to change its state or its behavior.

In this paper, we highlight this problem, which we call the *callback mismatch* problem. This problem arises (i) in the occurrence of abstraction mismatch, and (ii) when the aspect is required to issue a callback to the base application. As a consequence, the composition specification of such aspects becomes cumbersome to implement, difficult to maintain and impossible to reuse.

The structure of this paper is as follows. First, we define and illustrate the *callback mismatch* problem in a case study and we show that this is a realistic problem in the context of parallel development and reuse of aspectual modules. Then, we show that in the current state-of-the-art in aspect-oriented programming (AOP) and related techniques, patterns and notations, this problem persists and there is a need for advanced aspectual composition mechanisms to deal with this issue.

2. The callback mismatch problem

2.1 Problem definition

Pointcuts abstract not only from interesting join points in the base program but also expose relevant context data available at these join points. Abstraction mismatch is the problem where the representation of these abstractions in the base program is not compatible with the representation in the aspect. Dealing with abstraction mismatches is easy by employing a binding aspect that extracts the necessary information from the available base abstractions.

In the presence of callbacks however, specifying such a binding aspect becomes problematic. Callbacks happen when the reusable aspect uses the data and/or the behavior of the base application exposed by a pointcut to intervene in the normal control flow. As presented in Section 1, callbacks are mostly used to realize invasive aspects. To bind the callback to the base program, the binding aspect needs to include adapter functionality that routes the callback to the same base object that triggered the reusable aspect in the first place.

This problem is more complex to overcome than traditional problems with object-oriented libraries and frameworks (e.g. API mismatch). As the reusable aspect is never explicitly called from the base program, the adapter (or in this case the binding aspect) needs to adapt in both directions. It has to make sure that the relevant join points are translated to the aspect abstractions and that callbacks refer back to the original object. As a result, dealing with the callback mismatch problem takes more than solving the mismatch separately in both directions.

In summary, the *callback mismatch problem* leads to the following:

Problem summary. In the current state-of-the art of AOP languages, patterns and best practices, the required composition logic for dealing with both (1) *abstraction mismatch* and (2) *callbacks* is cumbersome to implement, difficult to maintain and impossible to reuse.

2.2 Motivating Example

To illustrate the problem outlined in this paper, we present a simplified example from the car crash management system (CCMS) [10, 16]. This is a large-scale and realistic distributed application that helps the authorities dealing with car crashes more efficiently by (i) centralizing all information related to a car crash, (ii) proposing a suitable crash resolution strategy, (iii) dispatching resource workers (e.g. first-aid workers) to the crash site, and (iv) reassessing the strategy in real-time when new information comes in.

To avoid wasting resources on prank calls and witnesses assuming a false identity, the correct and efficient functioning of the CCMS depends highly on *witness identity validation*, which is implemented in the CCMS as an aspect. More specifically, as long as the system has not successfully validated the identity of the witness, the CCMS will operate in *limited mode*, meaning that only a restricted set of resources can be assigned to that particular car crash.

Figure 1 presents this aspect in detail. The sequence starts when a witness calls the crisis center to report a car crash. The coordinator answering the call enters the name and phone number of the witness into the CCMS.

In this example, `Witness` represents the base abstraction: it provides the information needed by the aspect.

The *witness identity validation* aspect is provided in the form of a reusable identity validation aspect `IdentityValidation`. The required aspect abstractions in this example are `Person` and `ValidationReport`.

This illustrates *abstraction mismatch* in this example: the provided abstraction of the base application is the `Witness` which encapsulates the name, the phone number, and the validity state of the witness. On the other hand, the required interface of the aspect consists of (1) the `Person` abstraction which encapsulates first and last name and phone number, and (2) `ValidationReport` abstraction which encapsulates the validity state.

As pointed out in Section 2.1, this issue can be resolved by specifying a binding aspect with adapter functionality. In this example, we have implemented a *class adapter* which adapts the interface of the `Witness` object to match those of `Person` and `ValidationReport` (message 2).

After this, the aspectual composition with the *witness identity validation* aspect is realized. More specifically, the pointcut for this aspect is specified in terms of the `Person` interface (message 3). Both the `Person` and the `ValidationReport` are exposed through these join points.

Finally, the `IdentityValidation` component contacts a third-party telecom operator to check whether the presented person is indeed listed under the given phone number. The result of this verification activity is set via the `ValidationReport` interface. Because the `Witness` object has previously been adapted to this interface by the adapter, the callback ends up at the witness (message 4).

Section 2.3 illustrates in further detail how the adapter code is affected by the callback mismatch problem.

If the adapter is implemented incorrectly, the CCMS itself will remain in limited mode, and thus addresses the car crash inefficiently, if at all. The fact that the correct functioning of the entire application depends fully on the correct realization of the callback stresses the importance of writing an adapter that realizes the desired behavior in a comprehensible, maintainable and reusable manner.

2.3 Minimal solution in AspectJ

An example implementation of the scenario is included in the appendix. The pointcut `personIdNeedsChecking` in aspect `IdentityValidation` is defined in terms of types `Person` and `ValidationReport`. `Person` contains the data that needs to be checked and `ValidationReport` captures the result of the validation. Since these types are not directly supported by the base code, an adapter needs to be written to bind the aspect to the application. Listing 1 shows the adapter.

```
public aspect Adapter extends IdentityValidation{                    1
                                                                      2
  declare parents: Witness implements                                 3
      ValidationReport;
  public void Witness.validation(boolean b){                          4
    validate(b);                                                      5
  }                                                                   6
                                                                      7
  declare parents: Witness implements Person;                         8
  public void Witness.setFirstName(String s){}                        9
  public void Witness.setLastName(String s){}                         10
  public String Witness.getFirstName(){                               11
    return getName().split(" ")[0];                                   12
  }                                                                   13
  public String Witness.getLastName(){                                14
    return getName().split(" ")[1];                                   15
  }                                                                   16
  void around(Person w): execution(*                                  17
      Witness.setName(String)) && this(w){
    proceed(w);                                                       18
    w.setFirstName(w.getFirstName());                                 19
    w.setLastName(w.getLastName());                                   20
  }                                                                   21
                                                                      22
  pointcut report(ValidationReport report):                           23
      this(report);
}                                                                     24
```

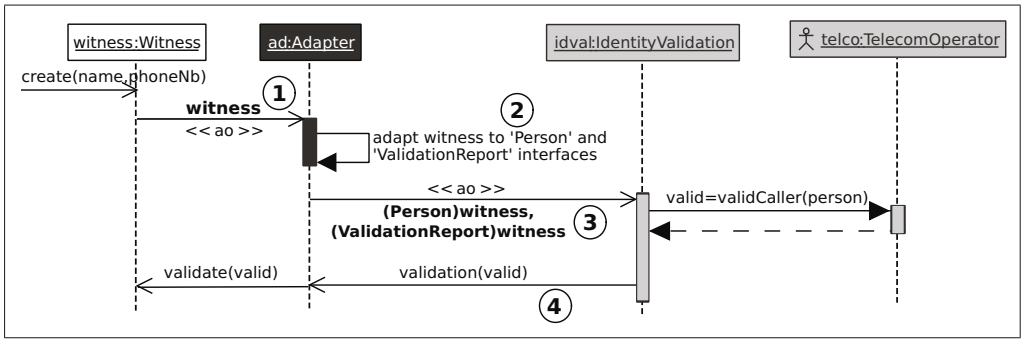

Figure 1. UML sequence diagram to illustrate the role of the adapter (in dark gray), and the *witness identity validation* aspect (in grey).

Listing 1. Example implementation of the adapter

In this scenario, the adapter has two responsibilities. Firstly, it needs to make sure that the callbacks through `ValidationReport` and `Person` are reified in the witness object. Therefore, the class `Witness` is made to implement the types `ValidationReport` (lines 3–6) and `Person` (lines 8–16) by means of *declare parents* and inter-type declarations (methods `setFirstName` and `setLastName` do not need an actual implementation because they are not used as a callback). Secondly, it needs to propagate the relevant join points on `Witness` as required join points on `Person`. This is achieved by around advice that calls *proceed* and additionally calls the appropriate methods (lines 17–21). Because there is a mismatch in the sense that `Person` has separate concepts for first name and last name, extra mapping functionality is required.

This example shows that even in this simple (almost trivial) case, defining the adapter is already a cumbersome task. One that needs to be repeated for every mismatch.

3. Approaches

This section gives an overview of existing AOP languages, techniques and patterns that are related to the problem and briefly argues that none of them sufficiently addresses the callback mismatch problem.

3.1 Explicit Pointcut Interfaces

Approaches like pointcut interface[6], XPI[15] and explicit join points[8] do not help to define bidirectional adapters more easily. The problem is that the aspect will always use a type description to be able to issue callbacks. This type should then be mapped to a concrete type in the base code. The approaches mentioned describe join points and not types and thus cannot be used in this mapping.

In the simplified case, the aspect specifies an abstract pointcut and the callback is issued on one of the exposed parameters. An explicit pointcut interface can help with implementing this abstract pointcut, but the mapping of the callback to the base code still needs to be done. A standard unidirectional adapter is sufficient in this case.

3.2 Type parameters

At first sight, type parameters seem to solve the callback mismatch problem, since an instantiated type parameter will behave as an alias for a concrete type of the base code. Unfortunately, for the aspect to be able to issue callbacks, it needs to refer to an actual type (and e.g. use it as a bound for the type parameter). As a result we end up with the same problems as before.

3.3 Caesar

Caesar supports on-demand remodularization to integrate independent components. Its model is object-based and uses virtual types, mixin composition and the concept of wrapper recycling [12]. As a result, Caesar provides a means to specify expressive, reusable adapters. However, Caesar does not support remodularization of aspect abstractions. In Caesar, the aspect composition is part of the binding and requires manual object wrapping (assisted by dynamic wrapper selection and wrapper recycling) [1, 13]. We can conclude that Caesar doesn't offer a solution to the callback mismatch problem as it not aims to bind abstract aspect compositions.

3.4 Subject-oriented programming

Subject-oriented programming [7] and its descendants Hyper/J and Theme[4] (which all involve Multi-Dimensional Separation of Concerns (MDSOC)) represent a more symmetrical approach to AOSD, meaning that each concern is developed independently. One of the key features of these approaches is *declarative completeness*, meaning that each concern explicitly defines the structure and behavior of the classes it depends on. To assemble an application, these concerns are composed using composition rules. Composition directives includes mechanisms for name-based merging of classes and methods, and support for renaming, overriding, ...

Because these mechanisms are nondirectional, they are inherently adequate for specifying callbacks. However, the composition mechanisms are not expressive enough to resolve sophisticated abstraction mismatches that can only be resolved with complex adapters involving more than renaming, overriding and merging classes and methods. Therefore these approaches do not solve the abstraction mismatch problem.

4. Conclusion

This paper introduces the *callback mismatch* problem. In essence, this problem is triggered by two key elements: (i) *abstraction mismatch* which is resolved by applying the Adapter design pattern [5], and (ii) *invasive* aspects [9], i.e. aspects that issue a callback to the base application to change its state or behavior. This situation leads to composition logic that is cumbersome to implement, difficult to maintain and impossible to reuse.

We have illustrated the problem in a minimal example from a realistic case study. Additionally, we have presented a number of factors that deteriorate this problem. Finally, we outline a number of related approaches in which this problem persists.

From this, we conclude that the current state-of-the-art is currently is not capable of solving the callback mismatch problem adequately. In our opinion, there are three distinct research directions

to be explored for a solution to this problem: (i) next-generation language constructs that allow the described adapters to be defined more elegantly and concisely (e.g. inspired by Caesar and SOP that provide disjoint sets of constructs that solve the problem partly), (ii) middleware-based solutions and framework-specific services that are capable of hiding most of the described adapter complexity, or (iii) AOP design patterns that provide reference solutions to this problem.

Logging, tracing and authentication are aspects addressed pervasively throughout AOSD research. Based on the impact that these aspects have on the base application, they are characterized as either *spectative* or *regulative* [9]. The large body of research into these particular aspects classes suggests that they are well-known, and they can sufficiently be dealt with by current AOP techniques. As AOP matures, it is our opinion that the research focus should shift from *spectative* and *regulative* aspects towards more *invasive* aspects, which represent the most challenging class of crosscutting concerns. We believe that the problem brought to the forefront in this paper is a key hurdle in the road towards advanced AO languages, middleware and patterns that deal with these types of aspects in an efficient, maintainable, and reusable manner.

Acknowledgments

This research is supported by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, by the Research Fund K.U.Leuven.

References

[1] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. An overview of caesarj. [14], pages 135–173.

[2] Maarten Bynens and Wouter Joosen. Towards a pattern language for aspect-based design. In *PLATE '09: Proceedings of the 1st workshop on Linking aspect technology and evolution*, pages 13–15, New York, NY, USA, 2009. ACM.

[3] Maarten Bynens, Bert Lagaisse, Eddy Truyen, and Wouter Joosen. The elementary pointcut pattern. In *BPAOSD'07: Proceedings of the 2nd workshop on Best practices in applying aspect-oriented software development*, pages 1–2, 2007.

[4] Siobhán Clarke and Robert J. Walker. Generic aspect-oriented design with Theme/UML. pages 425–458.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, illustrated edition edition, November 1994.

[6] Stephan Gudmundson and Gregor Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[7] William H. Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *OOPSLA*, pages 411–428, 1993.

[8] Kevin Hoffman and Patrick Eugster. Bridging java and aspectj through explicit join points. In *PPPJ '07: Proceedings of the 5th international symposium on Principles and practice of programming in Java*, pages 63–72, New York, NY, USA, 2007. ACM.

[9] Shmuel Katz. Aspect categories and classes of temporal properties. [14], pages 106–134.

[10] Jörg Kienzle, Nicolas Guelfi, and Sadaf Mustafiz. Crisis management systems: A case study for aspect-oriented modeling. Technical Report SOCS-TR-2009-3, School of Computer Science, McGill University, 2009. `http://www.cs.mcgill.ca/research/reports/2009/socs-tr-2009-3.pdf`.

[11] Ramnivas Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., Greenwich, CT, USA, 2003.

[12] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 52–67, New York, NY, USA, 2002. ACM.

[13] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM.

[14] Awais Rashid and Mehmet Aksit, editors. *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*. Springer, 2006.

[15] Kevin J. Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/SIGSOFT FSE*, pages 166–175, 2005.

[16] Dimitri Van Landuyt, Eddy Truyen, and Wouter Joosen. Discovery of stable domain abstractions for reusable pointcut interfaces: common case study for ao modeling. Technical report, Department of Computer Science, K.U.Leuven, 2009. `http://www.cs.kuleuven.be/publicaties/rapporten/cw/CW560.abs.html`.

A. Entire example

The full source code is available at `http://www.cs.kuleuven.be/~dimitri/callbackmismatch.zip`.

```
public abstract aspect IdentityValidation {

  pointcut personIdNeedsChecking(Person
      person, ValidationReport report):
    (execution(new(..)) || execution(void
        Person.set*(..))) && this(person) &&
        report(report);
  abstract pointcut report(ValidationReport
      report);

  Object around(Person person, ValidationReport
      report):personIdNeedsChecking(person, report){
    Object res = proceed(person, report);
    report.validation(TelecomOperator.validCaller(
        person.getFirstName() + " " +
        person.getLastName(),
        person.getPhone()));
    return res;
  }
}
```

Listing 2. IdentityValidation.aj

```
public interface Person {

  public String getFirstName();
  public String getLastName();
  public String getPhone();
  public void setFirstName(String fname);
  public void setLastName(String lname);
  public void setPhone(String phone);

}
```

Listing 3. Person.java