

Decomposition into Elementary Pointcuts: A Design Principle for Improved Aspect Reusability

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

{bertl,wouter}@cs.kuleuven.be

Abstract

In this position paper we propose the design principle *decomposition into elementary pointcuts*, which aims at improving the reusability of aspects that combine pointcuts and advices in one software module. This essential design principle can only be effectively applied if the following language features are supported: pointcut inheritance and explicit binding of aspects. In this paper we discuss the need for those features, and illustrate this with an example in AspectJ and CaesarJ. We also discuss alternative design decisions that ease the pain, when those features are missing.

1 Introduction

Aspect oriented software modules take many forms. In this paper we focus on aspects that combine pointcuts and advices in one software module, like in AspectJ[3] or CaesarJ[9]. An advice defines certain behaviour of the aspect and is associated with a pointcut, which defines the composition logic: i.e, where and when the behaviour has to be applied. Reusability of those aspect oriented software modules can be defined as the ability to reuse their behaviour (advices) and composition logic (pointcuts), and to reuse the aspects as prefabricated entities in executable form in a third party composition [1, 2]. This ability to reuse an aspect depends on some essential features of the *programming language*, but also on *key design decisions* to be made by the developer. In this paper we propose and discuss a key design principle to improve the reuse of aspects: decomposition into elementary pointcuts. This essential design principle can only be effectively applied if the following language features are supported: pointcut inheritance and explicit binding of aspects.

In aspect languages, we have identified the following features that support reusability of aspects. The discussed features exist in aspect languages, be it scattered over different technologies.

1. Aspect inheritance, that supports inheritance of concrete pointcuts as well as advices. This kind of aspect inheritance supports reuse of pointcuts and advices in child aspects and supports overriding pointcuts to re-define them. Child aspects can also add behaviour

(advice) to pointcuts defined in the parent aspect.

2. Explicit binding of an aspect: compiling or loading an aspect should not directly bind the aspect to the advised classes. The aspect is explicitly deployed at load time or runtime by means of a deploy operation. This is necessary when pointcut inheritance is supported, because it is possible the parent aspect is only *used* by the child aspect, without that the parent aspect is bound to classes.

Another important feature of the aspect technology is reusability of prefabricated, compiled aspects, because the source code is not always available. Within the scope of this paper, we only shortly discuss this feature. From a complete perspective on reusability, it is required, but we don't prioritize it in this paper.

These language features itself are not enough to ensure reusability of aspects. The developer of an aspect also needs to apply some design principles. This paper proposes the following design principle to improve the reusability of aspects: *decomposition into elementary pointcuts*. It is a design principle for languages that support pointcut inheritance and explicit aspect binding (e.g. CaesarJ). The key idea is to decompose pointcuts into multiple elementary pointcuts. Each elementary pointcut only defines *a restriction on one property of the joinpoint*, for example the joinpoint type (call or execution), the class, the members of the class, or the arguments ... The complete pointcut is built with the different elementary pointcuts by means of logic operators. In this way we decompose the pointcut for possible reuse.

The rest of the paper is structured as follows. In the second section - problem elaboration - we discuss the key features for reuse of aspects: pointcut inheritance, explicit binding and reusability of prefabricated aspects. We illustrate the need and show how they are supported in AspectJ and CaesarJ. We argue that the reusability of the aspects also depends on the design decisions of the developer. The language features only do not guarantee reusability of an aspect. In section number three, we discuss the design principle of decomposition into elementary pointcuts as a possible solution for the problems encountered in the second section. Afterwards, in section four, we evaluate the influence of the principle on other software engineering properties. Then we discuss related work and future work. Finally we conclude.

2 Problem Elaboration

First we define the working example of this paper. Then we work it out in AspectJ and CaesarJ. We discuss the features of each language for reuse of aspects. We illustrate the support for the features or the lack of support. In case a language lacks support, we propose a design strategy that could ease the pain.

In the end we illustrate and conclude that even the support for the features does not guarantee reusability of an aspect, and that applying design principles to aspects is needed to further improve the reusability.

2.1 The example

The setting of the example is the sector of e-finance. It is an example taken from a larger project but it has been abridged in this paper for conciseness. The key entity in the system is the *core bank account*, which has a unique id and balance. It allows to do two kinds of transactions: to withdraw and deposit an amount on the account. The account keeps a list of all transactions that have been done. A *transaction entity* consists of an amount (positive for deposits, negative for withdrawals) and a message describing the transaction. This message is printed on the online and printed statements for the customers. Account and Transaction are defined in listings 1 and 2.

Listing 1: Account

```
class Account{
  String id;
  double balance;
  List<Transaction> transactions;
  Account(string id){...}
  double GetBalance(){...}
  void Withdraw(double amount){...}
  void Deposit(double amount){...}
  List<Transaction> GetTransactionList(){...}
```

Listing 2: Transaction

```
class Transaction{ ...
  Transaction(int id, double amount, string msg){...}
  double GetAmount(){...}
  String GetMessage(){...}}
```

The amounts of the two transactional operations on the account - withdraw and deposit - are verified if they are strictly positive. This verification is implemented as an aspect *AmountVerification*. We define only the key idea of *AmountVerification* for now: if operation withdraw or deposit of Account is executed, and if the amount is not larger than zero, throw an exception. Further we discuss the possible options in AspectJ and CaesarJ to implement this aspect.

The idea of the core bank account is to reuse it for the different services the bank has to offer: basic banking services like current accounts and savings accounts, investment services, international bank accounts, ... In this paper we reuse it in a service for basic banking. This service, located on the bank's application servers allows to create new current accounts, withdraw money from an account, deposit money on an account and transfer money between accounts. This service is remotely accessible from the workstations of the bank employees in the branch

offices. The basic banking service is a composition scenario that uses the previous entities: Account, Transaction and AmountVerification (preferably, but not necessary, in executable form). The service is defined in listing 3.

Listing 3: BasicBanking service

```
class BasicBanking{
  List<Account> accounts;
  void CreateCurrentAccount(string id){...}
  void Withdraw(string id, double amount){...}
  void Deposit(string id, double amount){...}
  void Transfer(string from, string to,
               double amount){...}}
```

From the point of view of security - a key issue in e-finance - verification of the amount of the three transactions should happen as soon as possible. It is an important security guideline to do input validation first. For example, it decreases unnecessary resource usage at the server. So, we also need to verify the amount of the three transactional operations of BasicBanking; the fact that BasicBanking uses Account and thus AmountVerification will be applied anyway is not satisfactory.

Intuitively, we assume that reuse of the existing AmountVerification aspect to validate the amount in the BasicBanking service, should be very easy, because the operations are almost the same (syntactically and semantically). Only one extra operation should be verified, which is also an account transaction and has an amount involved. It seems, we actually only need to extend the pointcut with another operation where the aspect has to apply.

More concrete, we want to reuse the AmountVerification aspect to achieve the following results. The amounts of the three operations withdraw, deposit, transfer in BasicBanking are verified. And, as an optimization, when BasicBanking calls Account's withdraw and deposit, verification of the amount in Account is not necessary, because it already happened in the BasicBanking service. But this optimization may only occur when the BasicBanking service is the caller. In a typical deployment scenario, calls on Account coming from elsewhere, still need to be verified.

Before we elaborate on the example in AspectJ and CaesarJ, we explain our design choice for the basic banking service. The default design choice in a seemingly academic example like this, would be to let inherit current account from account. But the basic banking service and account entity should be considered simplified representations of coarse-grained components in a distributed setting. The core account entity is a composite component representing the persistent entities account and transaction, like the composite entity pattern [11] with EJB's in J2EE [12]. The basic banking service is a session facade [11] for the remote clients in the branch offices.

2.2 Reusability of aspects in AspectJ

We discuss first the support in AspectJ for aspect inheritance, then its support for explicit binding and last its support for reuse of prefabricated, compiled (binary) aspects.

Inheritance Reusing aspects by means of inheritance in AspectJ is limited to reusing abstract aspects. These are

aspects with one or more concrete advices defined on abstract pointcuts. Abstract aspects do not contain any concrete pointcuts, only abstract pointcuts. So, reuse of aspect behaviour is possible using abstract aspects, but reuse of composition logic defined in pointcuts is not possible by means of inheritance .

Explicit binding The AspectJ language does not support explicit binding of aspects. When a pointcut matches a certain joinpoint, the aspect is always bound. But, the AspectJ 5 runtime, offers the possibility to weave an aspect at load time. So the aspect is not bound at compile time. For this load time weaving, there is a configuration file that defines the distinction between abstracts aspects to be loaded and concrete aspects to be woven. This configuration file can (possibly) be used to load an aspect, without actually binding it. Nevertheless, without pointcut inheritance, loading an aspect without binding it would not really offer any advantages towards reuse.

Reuse of binaries Reuse of the prefabricated entities produced by the AspectJ compiler used to be quite complex, due to compile time weaving. It was unclear what the resulting binary provided as public interface: which were the types that it provided in its public interface. What were the resulting classes in the binary named? Were the aspects still present as defined in the source code, or were they completely woven into the other Java classes? All these problems made it difficult or even impossible to reuse the compiled classes and the aspects as a prefabricated entity in a third party composition. But, load time weaving and reweavable aspects in AspectJ 5 will improve the reuse of classes and aspects that are compiled by the AspectJ compiler.

Conclusion Due to the lack of pointcut inheritance and explicit binding, reuse of aspects in other aspects is impossible. But, reuse of the original aspect with its pointcut and advices in a third party composition with new classes is possible, even when it's already compiled. In this context, we define a design principle to improve the reusability of aspects in third party compositions: the least restrictive pointcut approach.

2.3 The example in AspectJ

We show the example in AspectJ using different design strategies. First, we use an aspect with one advice and an anonymous pointcut. Second, we use an abstract aspect with advice on a named abstract pointcut. And third, we discuss the design strategy of *least restrictive pointcut expression*. This design principle realizes the reuse of pointcuts for the example we defined.

A first design option for the AmountVerification aspect is a solution with an anonymous enumeration pointcut [4]. The pointcut is defined in the advice and enumerates the transactions of Account (see listing 4).

Listing 4: The first approach

```
aspect AmountVerification{
  before(double amount):
    execution(void Account.WithDraw(..)
      && args (.., amount) ||
    execution(void Account.Deposit(..)
      && args (.., amount) {
    if (amount <= 0) throw new Exception();}}
```

We must conclude that reusability of this aspect in the third party composition with the basic banking service is nonexistent. We can't reuse its pointcut or advice for the BasicBanking service. Possible solutions to increase its reusability are : (1) reuse of the advice only, using an abstract aspect or (2) pointcut and advice reuse, using the design principle of least restrictive pointcut. These approaches can be applied to reweavable aspects, but also to source code reuse of aspects without source code modification.

Reuse of advice only In AspectJ, a common approach to make an aspect more reusable is actually making two aspects: one abstract aspect, with an abstract pointcut, only implementing the advice. And a second (concrete) aspect inheriting from the first one and specifying the pointcut. The first abstract aspect can then easily be reused in third party composition scenarios by inheriting from it and defining the pointcut for that scenario. Applied to the example, the abstract aspect Verification is defined in listing 5. The concrete AmountVerification aspect for Account is defined in listing 6.

Listing 5: Abstract Verification aspect

```
abstract aspect Verification{
  abstract pointcut transactions(double amount);
  before(double amount): transactions(amount){
    if (amount <= 0) throw new Exception();}}
```

Listing 6: AmountVerification for Account

```
aspect AmountVerification extends Verification{
  pointcut transaction(double amount):
    execution(void Account.withdraw(..)
      && args (.., amount) ||
    execution(void Account.deposit(..)
      && args (.., amount);}
```

The two aspects above are reused in the third party composition with the basic banking service. The definition of the aspect *basic banking amount verification* without the optimizing is in listing 7.

Listing 7: BBAmountVerification for BasicBanking

```
aspect BBAmountVerification extends Verification{
  pointcut transaction(double amount):
    execution(void BasicBanking.WithDraw(..)
      && args (.., amount) ||
    execution(void BasicBanking.Transfer(..)
      && args (.., amount) ||
    execution(void BasicBanking.Deposit(..)
      && args (.., amount);}
```

An aspect TotalVerification, inheriting from the abstract Verification aspect, that would do the optimized verification for Account and BasicBanking, only makes sense if the AmountVerification aspect is not yet bound to Account. This is only possible if the AmountVerification aspect is compiled for load time weaving, and is not loaded

by the third party. Before AspectJ 5 it was actually impossible to do the optimization, because AmountVerification was always bound to Account at compile time. Because in AspectJ TotalVerification is also only an extension of the abstract aspect Verification, without any reuse of composition logic, we postpone its definition until we discuss CaesarJ.

We conclude that abstract aspects do offer the possibility to reuse aspect behaviour, but it requires that the developer of the original aspect splits up his aspect in an abstract aspect defining the behaviour and a child aspect defining the composition logic. Next, we inquire into reuse of pointcuts in third party compositions.

Pointcut and advice reuse by least restrictive pointcut approach The AspectJ developer of AmountVerification could have developed his aspect taking reuse in account, by specifying his pointcuts in a more generic way. E.g: by replacing the Account with *, in the original AmountVerification aspect, it would verify the amount of the withdraw and deposit operation of BasicBanking (with load time weaving). But, the transfer operation would still be a problem.

In general, the condition defined in the pointcut to evaluate on a joinpoint should be the *least restrictive expression* within the scope of the developer. This means: the pointcut expression should define restrictions on as less possible parts of the joinpoint. The pointcut typically takes the form of a pattern pointcut [4] with wildcards on the parts of the joinpoint that do not require any restriction. This approach works when you want to reuse an aspect provided as a compiled reweavable aspect or provided as source code.

Applying this to the example, we actually can achieve complete reuse. The definition of the AmountVerification aspect is as in listing 4 or 6, but with the pointcut defined as follows:

```
execution(* *(..) && args (.., amount))
```

In the third party composition, the AmountVerification aspect is woven into the basic banking class at load time. So, in this particularly case of Account and BasicBanking, the least restrictive pointcut approach offers a kind of pointcut reuse, be it without the optimization.

Discussion Actually, in some circumstances, this approach can cause a lot of problems. This least restrictive pointcut above means : any operation with a double as last argument. This will definitely involve a lot of unwanted side effects in third party compositions. It is then the responsibility of the developer reusing the aspect in a third party composition to restrict the pointcut to the least restrictive definition for his composition. But this involves that the developer needs to have the ability to access the members of the aspect's pointcut, and reuse them in a new aspect replacing the original one, or he needs to have the ability to modify the pointcut in the original aspect. The latter approach is not always possible, e.g. when using a compiled reweavable aspect. It is also not desirable when

reusing an open source aspect; editing the original code is not considered a clean reuse strategy. The former approach, reusing parts of the original pointcut in a new aspect replacing the original one, seems to offer a more structured solution. AspectJ does not offer the feature to reuse pointcuts in new aspects, for example by means of pointcut inheritance, but CaesarJ does.

2.4 Reusability of aspects in CaesarJ

CaesarJ supports more advanced inheritance of aspects. Reuse of aspect oriented composition logic (pointcuts) is supported in CaesarJ. By means of inheritance, concrete pointcuts are inherited. They can be used in the child aspect and can even be overridden. For example, a security aspect defines a pointcut with the sensitive operations of an application. An authentication aspect and an authorization aspect inherit from this security aspect. In that way they reuse the composition logic from the security aspect to add the specific behaviour for authentication or authorization.

Explicit binding of aspects is supported in CaesarJ by means of the *deploy* operation. It is then essential to leave out the *deployed* keyword in the aspect definition.

CaesarJ only supports reuse of source code, not compiled executables.

2.5 The example in CaesarJ

In CaesarJ, the AmountVerification can be defined as follows :

```
cclass AmountVerification{
    pointcut transactions(double amount):
        execution(void Account.WithDraw(..)
            && args (.., amount) ||
            execution(void Account.Deposit(..)
            && args (.., amount));
    before(double amount): transactions(amount){
        if (amount <= 0) throw new Exception();}
```

CaesarJ allows to override a pointcut in an inheriting aspect. Caesar also allows to use inherited pointcuts in the pointcut definitions of the inheriting aspects. But with the current implementation of Caesar, when overriding a pointcut, we were not able to refer to the overridden pointcut of the parent aspect (e.g. using *super*). Nevertheless, we assume that this feature will be present in the implementation in the near future. In the overriding of the pointcut, one can then refer to the parent pointcut with *super*, and add additional conditions to the pointcut. For example:

```
cclass BBAmountVerification extends AmountVerification{
    pointcut transaction(double amount):
        super.transaction(amount)
        && <some other condition>}
```

This feature allows to reuse pointcuts and advices of AmountVerification in BBAmountVerification. But still, it is not possible to state that it should be BasicBanking to which the aspect should be applied, instead of Account, and that it should also be applied to the transfer operation of BasicBanking. We actually end up with a complete redefinition of the pointcut. This incorporates as much reuse

of the pointcut as the approach with abstract pointcuts in AspectJ.

The problem is that we have to reuse the pointcut of the parent aspect as a whole. This is a consequence of defining the pointcut as one regular expression. We would actually like to reuse some parts of it, and redefine other parts. By applying the design principle of *decomposition into elementary pointcuts* to the parent aspect, this is possible.

3 Decomposition into elementary pointcuts

A possible solution for the problem of the previous section is to decompose the pointcut in multiple elementary pointcuts. Each elementary pointcut only defines a *restriction on one property of the joinpoint* in the pointcut. The complete pointcut is built with the different elementary pointcuts by means of logic operators. In this way we decompose the pointcut for possible reuse. First we explain the basic concepts of this principle and illustrate them, then we define the final design approach to realize reuse of the original AmountVerification aspect including the optimization.

Basic concepts A pointcut defines a restriction on a property of a joinpoint if it narrows the matching joinpoints from all to less than all, based on that property only. An elementary pointcut defines only a restriction on one specific property, for example the class, the members or the arguments. Or if it is a call or execution. The syntax of pointcuts in CaesarJ does not allow us completely to do that. For example, in AmountVerification in listing 8, the pointcut *classes* defines two restrictions : execution as joinpoint type and Account as class. Ideally this would be decomposed into two pointcuts.

Listing 8: Decomposed into elementary pointcuts

```
cclass AmountVerification{
  pointcut classes(): execution(* Account.*(..));
  pointcut members():
    execution(* *.Withdraw(..) ||
    execution(* *.Deposit(..));
  pointcut arguments(double amount):args(.., amount);
  pointcut transactions(double amount):
    classes() && members() && arguments(amount);
  before(double amount): transactions(amount) {
    if amount <= 0 throw new Exception ;}}}
```

When inheriting from this aspect, one can override one of the pointcuts, to redefine one of the parts of the pointcut. The aspect BBAmountVerification in listing 9 reuses the aspect AmountVerification by means of inheritance to check the amount of transfer, withdraw and deposit on BasicBanking. It uses pointcut inheritance to adapt the composition logic for BasicBanking. The pointcut *classes* is redefined for executions on BasicBanking. The pointcut *members* also matches on the transfer operation.

Listing 9: Improved reuse in BBAmountVerification

```
cclass BBAmountVerification extends AmountVerification{
  pointcut classes(): execution(* BasicBanking.*(..))
  pointcut members():
    super.members() || execution(* *.Transfer(..);
}
```

In the resulting third party composition the two aspects are deployed : the original parent aspect and the child aspect. The original parent aspect still verifies the amounts of transactions on Account and the new child aspect verifies the amounts on transactions of BasicBanking. The latter also happens when the call comes from BasicBanking. So the optimization is not implemented here.

The e-finance example revisited If we want to be able to realize the optimization, that the amounts of transactions on Account don't have to be verified when the call comes from BasicBanking, there are two possible designs.

(1) Two child aspects of AmountVerification: one for Account (AAmountVerification) and one for BasicBanking (BBAmountVerification). AAmountVerification is defined in listing 10. BBAmountVerification is defined in listing 9. The two new child aspects are deployed in stead of the original parent aspect AmountVerification. Executions of transactions on account will verify for executions of transactions from BasicBanking in the control flow.

(2) One child aspect of AmountVerification for as well Account as BasicBanking, deployed in stead of the parent aspect. This aspect is defined as TotalVerification in listing 11. The *classes* pointcut is here extended to the least restrictive pointcut. Because this aspect is also bound to BasicBanking we need to use *cflowbelow* for the optimization here.

Listing 10: Optimized AAmountVerification

```
cclass AAmountVerification extends AmountVerification{
  pointcut fromBasicBanking():
    cflow(execution(* BasicBanking.*(..));
    //call is from BasicBanking
  pointcut transactions(double amount):
    super.transactions(amount) && !fromBasicBanking();}
```

Listing 11: Optimized TotalVerification

```
cclass TotalVerification extends AmountVerification{
  pointcut classes(): execution(* *.*(..));
  pointcut members():
    super.members() || execution(* *.transfer(..));
  pointcut fromBasicBanking():
    cflowbelow(execution(* BasicBanking.*(..));
  pointcut transactions(double amount):
    super.transactions(amount) && !fromBasicBanking();}
```

4 Influence on other properties

In this section we briefly discuss the improvement of other software engineering properties when applying decomposition into elementary pointcuts. These -ilities we discuss are comprehensibility, evolvability and maintainability. Comprehensibility of pointcuts defined as one long regular expression is far from optimal. Regular expressions are often cryptic and difficult to understand. Defining elementary pointcuts breaks up these complex long pointcuts into easy understandable pointcuts with a meaningful name (see listing 8). Using these meaningful names to define the composition of the final pointcut also makes the final pointcut easier to understand.

Evolvability of an aspect is improved because of the improved reusability and comprehensibility. Maintainability is improved because of the improved comprehensibility.

5 Related work

A lot of aspect technologies define pointcuts outside the aspect in a deployment descriptor, for example JBOSS AOP [10], Lasagne [13] or JAC (when using the configuration files) [8]. This approach can be compared to abstract aspects in AspectJ. Only behaviour of the aspect is reused, not the composition logic. Another possibility to define pointcuts in JAC is using the *pointcut* operation of aspects, for example in the constructor of the aspect. The parameters of this operation are strings, defining the host, class, object and members to which the aspect has to be bound. By defining these strings as properties of the aspect, a similar effect as pointcut decomposition can be achieved. Aspects in JAC are normal Java classes and thus support inheritance. So, pointcut inheritance can be simulated in JAC by defining the strings as properties. It is then possible to reuse or override them in subclasses.

In [4], the authors consider several AspectJ implementation methods for a set of concerns about display updating. They evaluate the implementations' degree of separation of concerns and locality. Therefore they also consider an evolution scenario with reusability. They reuse the pointcut defining state change of the display to add logging. They add the logging as an advice to the original aspect, and so change the source code. We do not consider that a clean reuse strategy. Using CaesarJ with pointcut inheritance would offer a more modularized solution and cleaner reuse strategy. By applying decomposition into elementary pointcuts to the pointcut that defines state changes, they could have achieved a better evolvability of the aspects. For example, when changing the display background color is a new change of the display state. This does not map to a set-operation on a shape. We have applied the design principle of decomposition into elementary pointcuts to this example and to other, more complex examples in [14].

In this paper we only discussed reuse of composition logic based on syntactic pointcuts. Reusability of metadata based approaches and semantic pointcuts is part of our ongoing work.

Important related work on design principles and AOP has been done in the domain of implementing design patterns with AspectJ [5, 7]. That research mainly focused on implementing existing design patterns from GoF etc [6]. It did not really introduce new design patterns.

6 Conclusion

New paradigms like AOSD lead to new design problems and design errors, when the designs are evaluated on certain software engineering properties. Today a lot of best practices are described for programming with AspectJ. We believe that when explicit binding and pointcut inheritance is introduced into mainstream AOP languages, best practices for correct use of pointcut inheritance will arise. New design patterns for better designs with aspects will grow from those best practices.

In this paper we introduced a design principle for im-

proving the reusability of aspects. We defined reusability of aspects as the ability to reuse their behaviour (advice) and composition logic (pointcuts). Applying decomposition into elementary pointcuts improves the reusability of the composition logic, but it requires some essential features of the aspect language: pointcut inheritance and explicit binding of aspects. Applying the principle to base aspects allows the developer of the child aspect to perform fine grained adaptations of the composition logic, for his specific composition scenario.

References

- [1] Clemens Szyperski, Component software: beyond object-oriented programming. Second Edition. ACM Press/Addison-Wesley Publishing Co., New York, NY, 2002.
- [2] Heineman et al. Component-based Software Engineering. Addison-Wesley Publishing Co.
- [3] Kiczales, G. et al. An Overview of AspectJ. In Proc. of ECOOP 2001.
- [4] Gregor Kiczales and Mira Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. ECOOP, Springer LNCS, 2005.
- [5] Hannemann and Kiczales. Design Pattern Implementation in Java and AspectJ, In Proc. of OOPSLA 2002.
- [6] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Design patterns: elements of reusable object-oriented software, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1995.
- [7] Alessandro Garcia, et al. Modularizing design patterns with aspects: a quantitative study, In Proc of AOSD 2005.
- [8] R. Pawlak et al. JAC: A Flexible Solution for Aspect-oriented Programming in Java. In 3rd International Conference on Meta-level Architectures and Separation of Concerns (Reflection), volume 2192 of LNCS, p1-p25. Springer-Verlag, 2001.
- [9] Mira Mezini et al, Conquering aspects with Caesar. In proc. of AOSD 2003.
- [10] JBoss Homepage, <http://www.jboss.org/>
- [11] J2EE Patterns: <http://java.sun.com/blueprints/patterns/>
- [12] J2EE Homepage: <http://java.sun.com/javaee/>
- [13] E. Truyen, et al. Dynamic and Selective Combination of Extensions in Component-Based Applications. In Proc. of ICSE'01.
- [14] Lagaisse B, et al. Improving Aspect Reusability With Decomposition into Elementary Pointcuts. In Internal Report, Department of Computer Science, K.U.Leuven, Belgium, February 2006. To be published.