

# The Elementary Pointcut Pattern

Maarten Bynens

Bert Lagaisse

Eddy Truyen

Wouter Joosen

DistriNet, KULeuven

<http://www.cs.kuleuven.be/cwis/research/distrinet/public/index.php?lang=en>

## ABSTRACT

Reuse of aspects, and pointcuts more specifically, becomes more and more important as AOP matures and libraries will include aspects. The Elementary Pointcut [2] design pattern aims to improve the reusability of aspects that combine pointcuts and advice in one module. This short paper will introduce this pattern using the GoF template [1].

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques

## General Terms

Design, Languages

## Keywords

Design Patterns, Aspect-Oriented Programming

## 1. INTENT

Define the general structure of a pointcut by decomposing it into elementary pointcuts. Elementary Pointcut lets sub-aspects override certain properties of the composite pointcut, without changing its overall structure. This way the composite pointcut, and hence the aspect in general, becomes more reusable.

The Elementary Pointcut pattern can be considered as the Template Method design pattern [1] applied to pointcut definitions.

## 2. MOTIVATION

Consider an e-finance framework that provides basic banking services. Accounts, the key entities in the system, have a unique id and a balance, and provide two kinds of transactions: to withdraw and deposit an amount on the account.

```
class Account{
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Workshop BPAOSD '07 March 12-13, Vancouver, British Columbia, Canada

Copyright 2007 ACM 1-59593-662-2/07/03 ...\$5.00.

```
String id;
double balance;
Account(string id){...}
double GetBalance(){...}
void Withdraw(double amount){...}
void Deposit(double amount){...}
}
```

Listing 1: The Account class.

The amount that is deposited or withdrawn should be strictly positive and this should be verified. This verification is implemented as an aspect AmountVerification.

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

Listing 2: The base aspect, verifying the transactions on Account (in CaesarJ).

The goal is to make the account and associated amount verification as reusable as possible for various banking services, like savings accounts, investment services, international bank accounts, ... Suppose for instance that the Account component is reused as part of a BasicBanking component that implements some basic banking services: creation of new accounts, withdrawal from an account, deposit on an account and transfer between accounts.

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

Listing 3: The BasicBanking class

From a security viewpoint, a key issue in e-finance, input

validation should happen as soon as possible. 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 reusing 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) and only one extra operation should be verified. More concrete, we want to reuse the AmountVerification aspect to achieve that the amounts of the three operations in BasicBanking (withdraw, deposit and transfer), are verified.

```

class BBAmountVerification extends
    AmountVerification{
    pointcut transactions(double amount):
        execution(* BasicBanking.WithDraw(..)
            && args(.., amount)
        || execution(* BasicBanking.Deposit(..)
            && args(.., amount);
        || execution(* BasicBanking.Transfer(..)
            && args(.., amount);
    }

```

**Listing 4: The subspect, verifying the transactions on BasicBanking (in CaesarJ).**

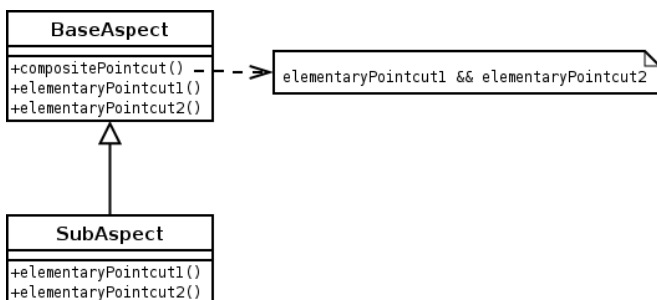
As AmountVerification is implemented right now, only the advice can be reused and all the common pointcut properties have to be repeated. Without a proper design of the AmountVerification aspect, reusing it is cumbersome and leads to duplication of pointcut expressions.

### 3. APPLICABILITY

Use the Elementary Pointcut pattern when you want to design a reusable aspect. The composite pointcut defines the invariant properties and structure of the pointcut and leaves it up to subspects to (re)define the variable parts.

To avoid duplication of pointcut expressions among several aspects, one can define a superaspect using Elementary Pointcut to express the common properties of the subspects.

### 4. STRUCTURE



**Figure 1: Structure of Elementary Pointcut**

### 5. PARTICIPANTS

- Base aspect - a base aspect with a composite pointcut decomposed into a number of elementary pointcuts. Ideally, the elementary pointcuts are orthogonal with respect to each other, so that they can be refined independently (e.g. each elementary pointcut puts a restriction on only one join point property and is least restrictive on all other join point properties, like a wildcard). The composite pointcut is then defined on the basis of the elementary pointcuts by means of logical operators.
- Subaspect - an aspect overriding some of the elementary pointcuts from the base aspect.

## 6. COLLABORATIONS

The subspect relies on the base aspect to define the invariant structure of the pointcut by means of elementary pointcuts.

## 7. CONSEQUENCES

The reusability of the superaspect is largely determined by the general structure that is defined by the composite pointcut. If each elementary pointcut only puts a restriction on one join point property (like call/execution, type, member, argument, ...)<sup>1</sup>, these properties become reusable separately, without duplicating the rest of the composite pointcut expression.

The 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 meaningful names. Using these meaningful names to define the composition of the final pointcut also makes the final pointcut easier to understand.

## 8. IMPLEMENTATION

Two aspect language features are necessary to benefit maximally from the advantages Elementary Pointcut offers:

- aspect inheritance with both advice and pointcut inheritance and pointcut overriding
- explicit aspect deployment.

Aspect inheritance is essential to be able to reuse pointcut expressions, preferably with the ability to refer to the inherited pointcut expression inside a redefinition (e.g. with *super*).

Explicit aspect deployment is needed to choose which aspects are active, in case base aspects are only used in the definition of subspects, without the necessity of being composed in the application themselves.

In AspectJ the reuse of aspects by means of inheritance is limited to abstract aspects, because of the absence of an explicit deployment operator. Also, only abstract pointcuts can be redefined, making Elementary Pointcut rather limited in AspectJ. An example of an aspect language that provides the necessary features is CaesarJ.

<sup>1</sup>This is not always possible, e.g. in AspectJ, the type property cannot be separated from the call/execution property.

## 9. SAMPLE CODE

The following code presents the benefits of using Elementary Pointcut in the basic banking example (in CaesarJ).

---

```
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 ;
  }
}
```

---

**Listing 5: The base aspect, using elementary pointcuts.**

The pointcut *transactions* is the composite, composed of the elementary pointcuts *classes*, *members* and *arguments*, each restricting one join point property (the class, the class member and the arguments respectively).

Without the execution join point property, these elementary pointcuts would be orthogonal w.r.t. each other.

---

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

---

**Listing 6: The subspect, reusing the base aspect.**

Except for the execution join point property, all common properties are reused by inheriting and overriding the elementary pointcuts. The class property is changed to BasicBanking and the member property is extended with Transfer.

## 10. REFERENCES

- [1] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [2] Bert Lagaisse and Wouter Joosen. Decomposition into elementary pointcuts: A design principle for improved aspect reusability. In *SPLAT*, 2006.