A Proposal For Classifying Tangled Code

Stefan Hanenberg and Rainer Unland

Institute for Computer Science
University of Essen, 45117 Essen, Germany
{shanenbe, unlandR}@cs.uni-essen.de

Abstract. A lot of different composition mechanisms claim to permit aspect-oriented software development because they are able to handle the problem of tangled code which was the original target of aspect-oriented programming (AOP). However, it becomes hard to compare those different approaches, because they usually focus on different kinds of tangled code. For comparing different approaches it is necessary to find some communalities between tangled code. Such communalities can be used to classify tangled code and in that way the impact of new aspect-oriented approaches can be determined in respect to what classes of tangled code they are able to handle. A general purpose aspect-oriented approach should be able to handle all kinds of tangled code.
Currently, there is no common agreement about how to classify tangled code. This paper outlines the need for such a classification and proposes a (preliminary) non-formal classification of tangled code.

1 Introduction

In the fundamental paper on aspect-oriented programming (AOP) Kiczales et al argue that "tangled code is extremely difficult to maintain, since small changes to the functionality require mentally untangling and then re-tangling it" [11] and propose aspect-oriented programming to solve the problem of tangled code. The problem of tangled code means that within software systems code exists, which is one the one hand redundant but on the other hand cannot be encapsulated by separate modules using regular techniques.

There are a lot of different approaches available which claim to permit aspect-oriented software development (AOSD) and therefore claim to be appropriate solutions for the problem of tangled code. Most of them mentioned in this context are AspectJ [2] implemented by those people who actuated the term aspect-oriented programming [11], HyperJ [14], which is an offspring of subject-oriented programming (SOP, [10]), DemeterJ [6] whose foundation came from adaptive programming [12] and composition filters [1]. More recent proposals include for example logical meta programming [16] or different mixin-mechanisms like destructive mixins [15] or per-object-mixins [13].

Although there are already some comparisons between different approaches like AspectJ and HyperJ (cf. e.g. [4]) there is currently no common accepted foundation available describing how to compare aspect-oriented mechanisms. This makes it hard to determine if a new proposal really supplies a new solution for untangling code which is not already available in any other approach. Furthermore, it makes it impossible for real software projects to determine what technique to use in a given situation. Moreover, it must not be forgotten that object-oriented programming already provides some techniques for sharing code and hence permits to untangle code in some way. So it is also necessary to determine if a needed mechanism is already available in OOP.

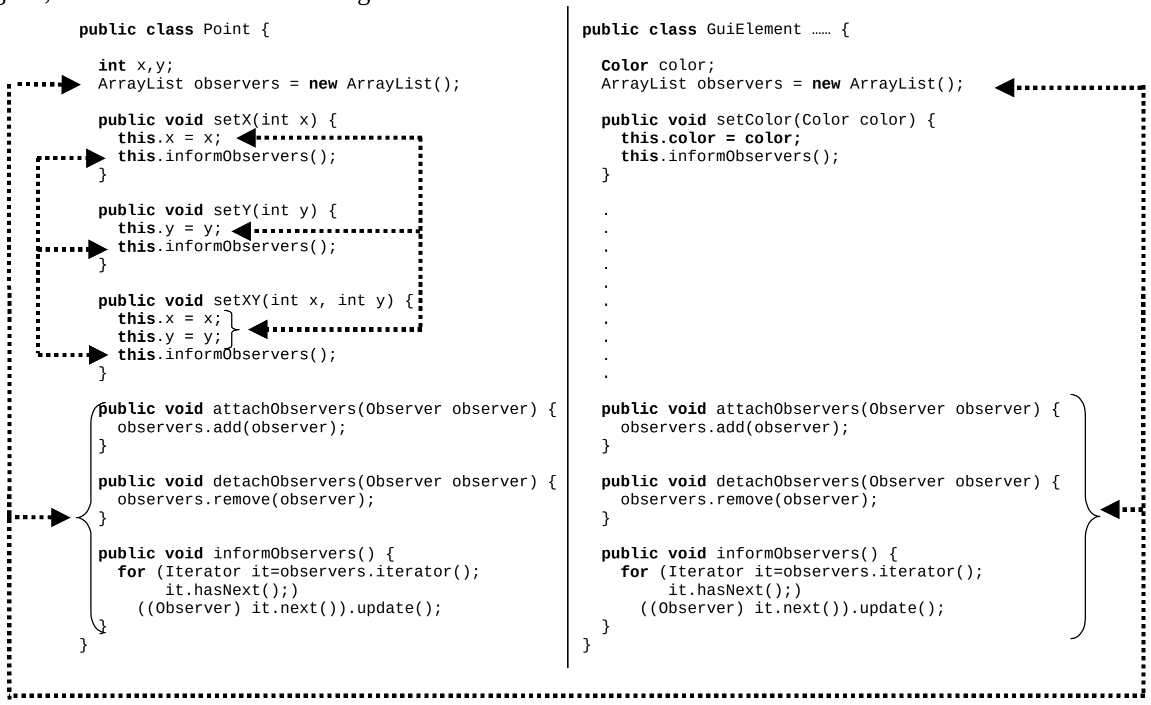
It is necessary to describe what kind of situation, i.e. what kind of tangled code exists, hence communalities between tangled code have to be analyzed. Those communalities can be utilized to classify different kinds of tangled code. Then it is possible to determine what techniques are able to handle what kind of tangled code. Because aspect-oriented programming claims to supply appropriate solutions for the problem of tangled code, a general purpose aspect-oriented approach should be able to supply appropriate solutions for all kinds of tangled code.

This paper proposes a (preliminary) non-formal classification of tangled code. We do not regard this classification to be complete. Instead it is the result of our observations we did during applying aspect-oriented programming, especially when applying AspectJ. Although the approach of aspect-oriented programming claims to be not limited to object-oriented programming, all of the above mentioned techniques are based on object-oriented concepts. Hence, the here proposed classification will also be based on the object-oriented paradigm.

In the next section we will motivate the need for a classification of crosscutting code. Afterwards we introduce a classification of crosscutting code and crosscuts. Afterwards we conclude the paper.

2 Motivation

Figure 1 and 2 contain some code examples, which obviously contain some tangled code. In figure 1 the tangled code results from an implementation of the observer pattern [8]. Both classes `Point` and `GuiElement` contain redundant methods for attaching and detaching observers. Moreover, the definition of the instance variable `observers` occurs in both classes. The intension of those definitions is the same and therefore it can be argued, that both classes contain tangled code.



```
public class Point {

  int x,y;
  ArrayList observers = new ArrayList();

  public void setX(int x) {
    this.x = x;
    this.informObservers();
  }

  public void setY(int y) {
    this.y = y;
    this.informObservers();
  }

  public void setXY(int x, int y) {
    this.x = x;
    this.y = y;
    this.informObservers();
  }

  public void attachObservers(Observer observer) {
    observers.add(observer);
  }

  public void detachObservers(Observer observer) {
    observers.remove(observer);
  }

  public void informObservers() {
    for (Iterator it=observers.iterator();
         it.hasNext();)
      ((Observer) it.next()).update();
  }
}
```

```
public class GuiElement …… {

  Color color;
  ArrayList observers = new ArrayList();

  public void setColor(Color color) {
    this.color = color;
    this.informObservers();
  }

  .
  .
  .
  .
  .
  .
  .
  .
  .
  .

  public void attachObservers(Observer observer) {
    observers.add(observer);
  }

  public void detachObservers(Observer observer) {
    observers.remove(observer);
  }

  public void informObservers() {
    for (Iterator it=observers.iterator();
         it.hasNext();)
      ((Observer) it.next()).update();
  }
}
```

Figure 1: Tangled Observer Implementation

There is already an object-oriented solution for this problem, because if both classes can have the same superclass containing the observer related implementation. So refactoring both classes using *extract superclass* [7] seems to lead to the desired result. Nevertheless, there might be reasons why it is undesired to use inheritance in that situation. On the one hand in object-oriented programming practices inheritance is often more than just a mechanism for code reuse and expresses a subtype relationship. Such a relationship might be undesired, because it expresses a common root of both classes. On the other hand extracting a superclass in programming languages which support only single inheritance is not that easy. So it seems to be worth thinking about (aspect-oriented) alternatives in such a situation.

The statement `this.informObservers()` occurs three times in different methods in class `Point` and one time in class `GuiElement` and therefore seems to be a potential example of tangled code. The intension of those statements stems also from the observer and defines that every time the state of an object changes all observers must be informed. The state of `Point` instance is describes by the coordinates `x` and `y`, the state of `GuiElement` objects is described by `color`. For such tangled code there is no object-oriented solution. Is must also be mentioned that this code depends on method `informObservers` and also in `this`. This dependency means that the code can only occur in (object-) methods whose class definition also contain method `informObservers` (assuming a class-based language).

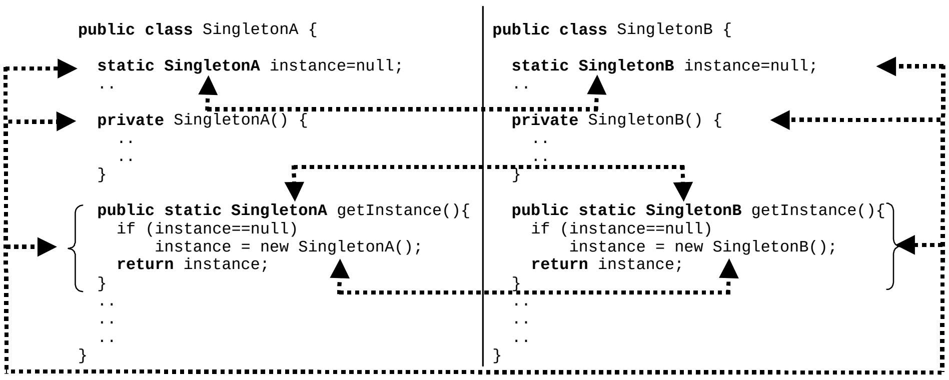
The assignments `this.x = x` and `this.y = y` exist in method `setXY` in addition to `setX` and `setY` in `Point`. In some way this kind of tangling seems to be more complex than the ones before. The observers should be informed exactly once whenever a method is called which changes a point's state. For this reason `setXY` cannot invoke `setX` or `setY`, because this would inform the observers twice. In this situation is does not seem to be clear what code is tangled: just by comparing lines of code the answer is that the assignments are tangled. On the other hand they appear more than once because of the property to be observable. It seems to be obviously that this is a different kind of tangling than the `this.informObserver` statement.[1]

Another often discussed example of tangled code is a singleton implementation [8] which can be found in figure 2. It is not that easy to identify the tangled code. One the one hand the body of both `getInstance` methods is (almost) the same. So it could be argued that the method bodies contain the tangled code. On the

[1] In fact this kind of code tangling was already exhaustively discussed in the context of AspectJ (cf. [3], [5] and [9]).

other hand the declarations of `getInstance` only differ in their return types. So it seems to be more appropriate to determine the whole method to be an example of tangled code. Programming languages like Java which allow neither to declare covariant methods nor generic types do not supply any appropriate solution to untangle this code.

The definition of `instance` also seems to be redundant but differs in both classes concerning the type. Moreover, the singleton design pattern is usually implemented using private constructors. So "declaring the constructor private" also seems to be kind of tangled code.



```
public class SingletonA {

  static SingletonA instance=null;
  ..

  private SingletonA() {
    ..
    ..
  }

  public static SingletonA getInstance(){
    if (instance==null)
        instance = new SingletonA();
    return instance;
  }
  ..
  ..
  ..
}
```

```
public class SingletonB {

  static SingletonB instance=null;
  ..

  private SingletonB() {
    ..
    ..
  }

  public static SingletonB getInstance(){
    if (instance==null)
        instance = new SingletonB();
    return instance;
  }
  ..
  ..
  ..
}
```

Figure 2: Tangled Singleton Implementation

The examples showed that there are a lot of different kinds of tangling. One the one hand there is tangled code which is easy to identify and which might exist on statement level (like `this.informObservers`). On the other hand there is code which is not that easy to identify, because the object-oriented realization already hides the details responsible for the tangling (like the implementation of method `setXY` in class `Point`). Furthermore, there is code somehow depends on the locations it crosscuts (like the type of `instance` in `SingletonA`).

One of the main cognitions of those examples is, that there is a difference between the tangled code itself and the way how this code crosscuts existing structures. Therefore we distinguish between crosscutting code and the way of crosscutting.

3 Crosscutting code

This section classifies the crosscutting code. The classification is based upon the following observations.

First, there is code which can be added to almost arbitrary classes or methods because the code does not expect any parameters from the location it is woven to. A typical example is a collection of methods, which do not are strongly coherent to each other, but on the other hand to not expect anything from their object except those methods. Such methods can be added to almost any class (restrictions are given e.g. if the underlying language does not support covariant methods).

Second, the crosscutting code not only depends on its environment in the sense, that it expects any parameters, but the code itself has to be adapted when woven to the new environment. For example C++ templates expect class parameters, so the code itself will be transformed to match the needs at hand. Nevertheless, the transformations which are necessary to transform the crosscutting code might be more complex than "just" transform type information.

Third, the crosscutting code can transform the environment it is woven to. The most typical example is a delayed exception declaration in java sources, i.e. the exception is part of the crosscutting code.

Unit Dependency

This characteristic describes, if the code depends on the unit which is affected by the crosscutting or where the crosscutting is located in. We distinguish between *unit dependent* and *unit independent crosscutting code*. Unit dependent crosscutting code needs some input from the units it crosscuts. For example the statements `this.informObservers()` which can be found in both classes of figure 1 are examples for *object dependent crosscutting code*, because they depend on the variable `this` and assume the object referred by `this` to contain a method `informObservers()`. On the other hand they are *method independent*, because they do not depend on any method related information. Method related information might be any parameters from the method or any variables local to the method. Although the crosscutting code is object dependent it does not mean, that every occurrence of the code is related to the same object. Obviously, `this` in `GuiElement` relates to a different object than `this` in `Point`.

Constant vs. Variable Crosscutting Code

There is a difference between constant and variable crosscutting code, i.e. the difference is if there are any variabilities within the code or not. Examples of constant crosscutting code can be found in figure 1. The statements `this.informObserver()` are exactly the same (although `this` might be related to different objects) , so the crosscutting code is constant. Also the methods for attaching and detaching observers are examples for constant crosscutting code. In contrast to that, the singleton implementation contains variabilities: the return type of method `getInstance` is a changeable. Likewise the method body of `getInstance` is variable, because in `SingletonA` it create a new instance of `SingletonA` and in `SingletonB` a new instance of `SingletonB`.

Variable crosscutting code needs a more complex composition technique, because the code changes when applied to a specific context.

Transforming Crosscutting Code

This characteristic describes, if the crosscutting code changes the declaration of the element it is applied to. For example, the methods for attaching and detaching observers are *class transforming crosscutting code* because they change the interface of the class those methods are applied to. On the other hand, `this.informObservers()` is an example for *non-class transforming crosscutting code*, because is neither transforms the declaration of the methods it is located in nor does it change the interface of any class.

Transforming crosscutting code can either increase or decrease the possible usage of the element the code is applied to. For example code which declares several classes to extend a certain class increases the possible usage of those classes. All existing clients of those classes are still able to use them. On the other hand code which declares a "delayed" exception of a method decreases the possible usage, because clients are enforces to catch those exceptions.

4 Crosscutting

The crosscutting describes at what positions code crosscuts existing structures. It is a certain strategy applied to existing code to add crosscutting code. While the section above classified "what" crosscuts given code, this section describes "how" it crosscuts. Classifying crosscutting is based upon the following observations.

First, the crosscutting might be described constantly or variable. Constant crosscutting means, that the strategy how the crosscutting code is added to existing code is fix and cannot be used in any other context. Second, crosscutting code is added to certain locations. The question arises if the crosscutting really crosscuts such location or if it just affects the location. The difference between both is, that a crosscut location is a common root for several affected locations. It is something like a selector for identifying several affected locations. Third, the relationship between the crosscutting code and the affected location might be static or dynamic. Static means, that this relationship only depends on the location's lifetime.

Constant vs. Variable Crosscutting

Constant crosscutting means that the way certain code crosscuts a given structure is described completely without any variabilities. In other words, the strategy applying the crosscutting code to the existing one is entirely defined without any hooks which allow the crosscutting to be used in any other context. In contrast to that a *variable crosscutting* contains variabilities which allow to add the crosscutting code to a new context. Referring to the introducing examples the singleton implementation crosscuts existing structures variable. The implementation can be attached to any class. Therefore the class to which the implementations are applied to are the variable part of the crosscutting. Variable crosscutting is an indication for reusability of the crosscutting code, because it allows to apply the crosscutting code to new contexts without changing crosscutting itself.

A constant crosscutting is usually applied if the corresponding code is highly coherent to a certain location and highly specialized for a (usually location dependent) purpose and thus cannot be reused for any other purposes. Examples for constant crosscutting can be found e.g. in the *simple telecom simulation* which is part of the AspectJ documentation (cf. [2]). Such applied crosscutting code is highly connected to a application specific needs and therefore the crosscutting cannot be used in any other context.

Crosscutting Location

The crosscutting location describes what entities are crosscut by the crosscutting code. It is necessary to distinguish between *crosscutting location* and *crosscutting affected units*. The first one describes the location to which the crosscutting is applied to, the latter one describes those units which are affected by the crosscutting. A crosscutting location is a common root for several affected locations. It is something like a selector for identifying several affected locations. We distinguish between class, method and object located (and affecting) crosscutting.

Class located crosscutting describes that kind of crosscutting which is restricted to a certain class. The observer implementation from figure 1 is an example of a class located crosscutting: the statements `this.informObserver()` are spread over methods of a certain class which represent the subject. On the other hand the crosscutting responsible for introducing the methods for attaching and detaching observers is not class located: although the methods are part of a certain class they do not crosscut the class (they occur exactly once in each classes). Hence, the crosscut which is responsible for attaching those methods to the class is *class affecting*, but not class located.

Object located crosscutting classifies that kind of crosscutting, that is restricted to certain objects. All statements `this.informObservers()`are object located, because they crosscut certain objects (which have the same type). The reason why the crosscutting is both class located and object related is that the underlying programming language of the examples is a class-based object-oriented programming language. For such a kind of programming languages every object located crosscutting is also a class located crosscutting (but not the other way around because of static methods and attributes). In prototypical languages which also allow the notion of classes it is possible that an object located crosscutting is not class located at the same time.

Method located crosscutting describes that kind of crosscutting that is restricted to certain methods. None of the examples above contains a method located crosscutting. A typical example of method located crosscutting is such code which is inserted into a certain method for debugging purposes like `System.out.println(..)` after every line of code. Although the statements `this.informObservers()`from the observer implementations are located in certain methods they are not method located, because the choice if a certain method should contain the code or not is not determined by the method, but by the enclosing object. So the crosscutting for the mentioned statements is *method affecting*.

The above enumerated kinds of crosscutting are related to object-oriented constructs. In that way the classification is already prescribed by the application code, e.g. all statements which are part of a certain method are classified by this method. In that way a classification "method located crosscutting" is according to the enclosing object-oriented construct. On the other hand a crosscutting might group constructs, which are not grouped by the application code. For example the methods for attaching and detaching observers crosscut a collection of classes which are not related in the application code. So the location of that crosscutting is a collection of the classes which is defined by the crosscut itself. The same situation is given for the `this.informObservers()` statement. The statement does not crosscut a single method, but a collection of methods which are defined by the (object located) crosscut itself. Such kind of crosscutting we call *selective*, e.g. the methods for attaching and detaching observers are called *selective class located crosscutting*.

Static vs. Dynamic Crosscutting

If the connection between crosscutting code and affected or located units depends only the unit's lifetime we call such crosscut *static*. In any other case the crosscut is *dynamic*. An example for a static crosscut are the methods for attaching and detaching observers or `getInstance` from the singleton implementations. They exists as long as the affected class definitions exist.

Likewise, it seems as if the statements `this.informObservers()` do statically crosscut the given structures, because they are always located in the methods which change an object's state. Nevertheless, this statement is an example for dynamic crosscuts. The assignments of `x` and `y` in method `setXY` are tangled, because it was not possible to invoke `setX` or `setY` without informing the observers twice. Therefore the object-oriented solution was to implement the assignment operations redundantly. The desired behavior of the observer implementation is to inform the observers exactly once, so `informObservers()` should be invoked after `setX`, `setY` or `setXY` but never more than once. This is what in [3] is called a *jumping aspect*. The connection between the crosscutting code and the crosscut location depends on the context the method was invoked. So the underlying crosscutting is dynamic.

AspectJ distinguished between static and dynamic crosscutting in a different way (cf. [2]). Static crosscutting describes mainly those kinds of crosscutting, which transform the interface of a class (also known as *introduction*), respectively "crosscutting concerns that do operate over the static structure of type hierarchies" [2]. Although this terminology seems to be somehow intuitive, it contains some inconsistencies: those crosscutting concerns, that in AspectJ terminology are called dynamic can operate over the static structure. E.g. code, which is executed right before a certain method is invoked is called "dynamic crosscutting". Nevertheless, there is nothing dynamic in it: the weaver uses static type information for connecting the crosscutting code and nothing during runtime unweaves that code. Nevertheless, we agree with the AspectJ terminology that construct like `cflow`- or `if`-pointcuts allow to express dynamic crosscutting.

Although it does not seem to be obvious, there is a difference between dynamic crosscutting and variable crosscutting. E.g. in AspectJ it is possible to define constant, dynamic crosscuts. Also it is possible to define variable, static crosscuts.

5 Conclusion and Further Work

This paper discusses the ability to characterize tangled code and proposed a preliminary classification of tangled code motivated by some introducing examples. It distinguishes between the crosscutting which describes where and how the crosscutting code affects existing structures and the code itself. Mainly the proposal distinguishes code because of its *location dependency*, *variability* and *transformational property*. The crosscutting is distinguished by its *variability*, *location* and its *dynamics*.

We regard the proposed classification to be neither complete nor mature. Instead we think that this proposal is the beginning of some work which is concentrating on different kinds of tangled code which are observable in practice and which unites those appearances using a common classification schema.

The proposed preliminary classification of crosscutting and crosscutting code needs much refinement. For example it does not distinguish between different kinds of unit dependencies in crosscutting code. Furthermore, it is necessary to determine what kind of variabilities are possible within the crosscutting. Such an awareness is necessary for building general purpose aspect languages which can provide adequate hooks for building reusable aspects.

The categories highly depend on the underlying object-oriented programming language. In static typed, class-based languages every object belongs to exactly one class for the whole lifetime. This means, that object located crosscutting code must be contained in a class definition. So in this case object related crosscutting code is always class located. In (class-based) single dispatching languages a method belongs to exactly one class. This means that every method located crosscutting is also class located. So in the future the impact of the underlying language on the classification must be determined.

In future works it is necessary to apply the here proposed classification to different kind of tangled code which occur in practice. Although the classification has been reasoned by the introducing example it must be observed in what situations the classification fails or is too rough to describe the observable occurrences adequately. It should be emphasized that the typical code examples which lead to the proposed classification came from the context of AspectJ. Afterwards the classification has to be applied to different aspect-oriented techniques.

References

1. Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A. Abstracting Object-Interactions Using Composition-Filters. In: object-based distributed processing, R. Guerraoui, O. Nierstrasz and M. Riveill (Eds.), LNCS, Springer-Verlag, (1993), pp. 152-184
2. AspectJ-Team, The AspectJ Programming Guide, http://aspectj.org/doc/dist/progguide/
3. Johan Brichau, Wolfgang De Meuter, Kris De Volder. Jumping Aspects, Position paper at the workshop Aspects and Dimensions of Concerns, ECOOP 2000, Sophia Antipolis and Cannes, France, June 2000
4. Chavez, C.F.G.; Garcia, A.; Lucena, C. "Some Insights on the Use of AspectJ and Hyper/J". Tutorial and Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, UK, August 23-24, 2001
5. Pascal Costanza. Vanishing Aspects, Workshop on advanced separation of concerns at OOPSLA 2000, Minneapolis, Minnesota, USA, 2000
6. Demeter/Java Web page, http://www.ccs.neu.edu/research/demeter/releases/
7. Martin Fowler, Refactoring: Improving the Design of Existing Code, Addison Wesley, 1999
8. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995
9. Stefan Hanenberg, Rainer Unland, Using and Reusing Aspects in AspectJ, OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, 2001
10. William Harrison, Harold Ossher, Subject-Oriented Programming (A Critique of Pure Objects), in: Pro-ceedings of the OOPSLA 1993, pp. 411-428
11. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwing, J.. Aspect-Oriented Programming. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag, 1997, 220-242
12. Karl J. Lieberherr, Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns, PWS Publishing Company, Boston, 1996
13. Gustaf Neumann and Uwe Zdun: Implementing Object-Specific Design Patterns Using Per-Object Mixins, Proceedings of the Second Nordic Workshop on Software Architecture (NOSA'99), Ronneby, Sweden, Aug. 12-13 1999
14. Peri Tarr, Harold Ossher, Hyper/J™ User and Installation Manual, 2001
15. Kris de Volder, Inheritance with Destructive Mixins for Better Separation of Concerns, Workshop on Advanced Separation of Concerns at OOPSLA 2000
16. Kris De Volder, Theo D'Hondt, Aspect-Oriented Logic Meta Programming, Proceedings of Meta-Level Architectures and Reflection, Second International Conference, Reflection'99. LNCS 1616, pp. 250-272, Springer-Verlag, 1999