Improving AOP systems' evolvability by decoupling advices from base code [1]

Alan Cyment, Nicolas Kicillof, Rubén Altman and Fernando Asteasuain
University of Buenos Aires
[acyment,nicok,raltman,fasteasuain]@dc.uba.ar

ABSTRACT

AOP systems evolvability is severely affected by the tight coupling between aspects and base code. This paper identifies the advice fragility problem, originated in the advice's need to access the application context while being oblivious to base code details. Proposed solutions to the pointcut fragility problem have generally put forward various mechanisms that decouple base code from aspects using some intermediate abstraction layer. We build on top of those proposals, introducing the concept of model-based aspects, and present a new version of our semantic pointcut framework, constituting a practical approach to address the advice fragility problem.

1.INTRODUCTION

Since the very inception of the AOSD family of concepts and technologies, the degree to which so-called aspect-oriented systems have successfully coped with the woes of software evolution has been thoroughly scrutinized [1]. One of the most debated ideas has been the *obliviousness* concept [3], first heralded by the community as a core prerequisite for considering a system truly aspect-oriented, but lately bashed by several authors [4] [23] as an obstacle to the successful evolution of AOSD systems.

Most of the existing research on this subject has so far focused on the *pointcut fragility* problem (coined by [2], and referred to by this and other names in [9], [8], [6], [5], [12]), which basically describes the dangerous coupling between an oblivious base code and a given *pointcut descriptor* (PCD) that heavily relies on the low-level structure of that code. But not much attention has yet been paid to the problem of maintaining *advices* synchronized with an evolving base code. Changing a method name can easily break advice code dependent on the structure of the class that has evolved (i.e., if no aspect-aware refactoring aide is used). This issue will hereafter be referred to as the **advice fragility problem**. Following the AspectJ convention, we use the term *aspect* to define the combination of advices and pointcuts. Hence, we will also refer to the **aspect fragility problem** to describe the compound issue posed by the two fragility problems so far described.

Proposed solutions to the pointcut fragility problem have generally put forward various mechanisms that decouple base code from aspects using some intermediate abstraction layer [4][10][11][9]. We build on top of those proposals, introducing the concept of **model-based aspects**. These are basically pointcuts and advices that, instead of relying on the low-level structure of base code, are decoupled from it by being defined in terms of an intermediate, more abstract, conceptual description of the domain modeled by the application.

The next section characterizes the advice fragility problem; section 3 presents our proposal for dealing with this problem, section 4 puts this proposal into perspective by showing our implementation of these ideas and a concrete example, and the remaining sections conclude the work.

2.ASPECT FRAGILITY = (ADVICE + POINTCUT) FRAGILITY

The obliviousness principle, originally considered by AOSD pioneers as one of the most rewarding features of this emerging technology, has recently been "considered harmful" by some researchers [4] [23]. Keeping base code completely unaware of the existence of aspects would undoubtedly make developers of the former happy. But recent research has shown that the natural evolution of the base code will very easily wreak havoc with the *aspect* side of the equation. Using existing AOP tools, if an aspect programmer were to follow the obliviousness principle, she would be forced to tightly couple her pointcuts to the base code, in order to completely avoid adapting the base code to the aspects.

An AOP system that evolves has both base code and aspects. Aspects are made up of pointcuts and advices. As we have mentioned, when base code evolves, pointcuts can easily become obsolete. But, what about advices? We claim that mechanisms that implement the obliviousness principle make advices excessively coupled to base code too. Advice is basically code written in a programming language. Its goal is to model, at least in part, a given domain, that happens to be a crosscutting concern of the complete application under development. But advice does not have a life of its own: it will only be executed at a joinpoint (i.e. a given point during the lifetime of the base code). Due to the quantification principle [3], advice code must be built so that it can successfully interact with heterogeneous base code. If the latter evolves (i.e. changes its structure), there is a clear risk of the original advice code not to fulfill the implicit *contract* intended by its original programmer. That is why we state that the fragility of advices lies in the way they **access context**.

We now describe different ways in which advices access context in AspectJ-like tools, showing for each of them how the evolution of base code can bring about errors in the complete system. In all cases, whenever base code changes, the programmer is to review all existing advice code in order to prevent these errors from taking place. Therefore, the resulting coupling of advice and base code results in what we have termed *advice fragility*.

[1] This work was partially funded by Microsoft Research's Phoenix—Excellence in Programming RFP Awards and ANCyT PICT 11738

The first case comprises what we call the **pointcut signature** kind of access, although some authors have termed it *typed advice* or *pointcut parameter*. The advice receives objects of a known type as arguments. But advice code often expects these objects to be in a given state (e.g. already initialized) or to be related to one another in a certain way (e.g. the first parameter is always smaller than the second one). Moreover, evolution of base code may trigger a change of type in the parameters of both a pointcut and its associated advice. The second case involves **reflection**. The use of reflection to access context is mandatory in many AOP tools. If base code changes, it is very easy for advice code to invoke, for example, a non-existent method. The third and last case is **object casting**. This case is similar to the previous one, in that advice code can easily break (this time triggering a casting exception) as soon as there is a small change in base code.

3.MODEL-BASED ASPECTS

Most of the proposed solutions to the pointcut fragility problem [12][21][9][22] decouple PCDs from base code by making the former depend not on the low-level structure of the latter, but rather on an intermediate, more abstract layer that aims at describing the problem at hand in a more domain-oriented fashion than raw code. Following this idea, we propose that this layer must be conceived in such a way that advice code can benefit from the resulting decoupling as well. This is illustrated in Figure 1.

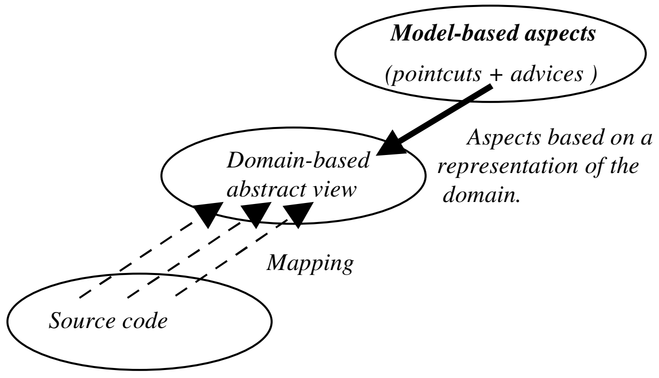
Model-based aspects
(pointcuts + advices)
Aspects based on a representation of the domain.
Domain-based abstract view
Mapping
Source code

Figure1: Model-based Aspect's schema

Let us now analyze, as a running example, an application the developer wishes to describe in *architectural* terms. The system consists of two *components*, connected via an *event handler*. Relationships between concepts are the basis of the architecture's configuration. They are undoubtedly significant, and should therefore be made explicit; the model should provide the means to describe these relations. Also, a relation must be established between base-code and the concepts they represent.

3.1 Relating the base code and the model

Mechanisms to relate the base code with the abstract layer can be broken down into those based on *predicates* (i.e. over base code) [9][4] and those using *annotations* [12][22]. Predicates are used for establishing *contracts* base code ought to comply with, and are by definition *intensional* (i.e. they are written once, but define a subset of the entire base code). Annotations (i.e. metadata), on the other hand, are associated to a single code entity, and can therefore be considered *extensional*. Predicates are therefore a better way of achieving the quantification property [3].

If context access is to be done in a conceptually higher level of abstraction, **advice code should be able to collaborate with living objects that represent entities from the conceptual model**. An object-oriented model is more naturally specified by extension rather than by intension. Object-oriented developers work by defining classes and instances, and only in some flavors of the paradigm do they use the concept of contracts. We have therefore chosen *annotations* as the means for relating the conceptual model to base code.

In the example just presented an object model can describe the architecture far more intuitively than a set of predicates. Moreover, architecture elements (i.e. components and event-handler) cannot easily be associated to base code using predicates. Code annotations offer in this case a better way to relate classes and methods to components and event-handlers.

3.2 The need for mappings

As mentioned before, advice fragility has its roots in the need for advice to access context. Context access can be thought of as the advice's code need to *collaborate* with (or refer to) objects originated in the base-code world. In contrast to pointcuts, where coupling to base code stems from the PCDs relying on base-code structure and execution flow, advices rely both on the naming and semantics (i.e. contract fulfillment) of base-code methods.

Probably the most widely adopted way of decoupling collaboration among objects is the *adapter pattern* [24]. What an adapter basically does is stand in the middle of a given collaboration, so as to decouple both ends. In the same fashion, a series of patterns, namely *façade* and *mediator*, decouple an object from a group of other objects that together pursue a given goal. Taking a look at what these patterns have in common**,** we claim that the intermediate layer should allow a **mapping of both behavior and structure between base code and the world of aspects**. Using object-oriented models for the abstract layer clearly offers a clean way of specifying this mapping. [10] and [11] use a similar idea to decouple collaboration between advice and base code.

Going back to the previous architecture model, let us suppose that, at base-code level, *events* are identified with numerical IDs, but at the architectural level they are named using mnemonic strings. This non-trivial mapping can easily be achieved by defining an ad-hoc mapping object.

3.3 Comparing *explicit* and *implicit* models

We have proposed to use annotations for relating base code to entities in a conceptual model. But depending on the complexity of the application, sometimes conceptual models need not be *explicitly* defined. When this happens, annotations will refer to concepts that are not defined elsewhere, they merely become loose labels. This can be done if there are no relevant relationships between concepts, and only when mappings from base-code elements to concepts are straightforward.

As an example of implicit conceptual models, let us assume a developer wants to enrich base code with a trivial model that classifies methods into those that *update* and those that *retrieve data*. Using this simple model, the developer could later write a simple *transaction* aspect that enlists only *updating* methods. Later on, a *caching* aspect could be added, in order to cache the returned value from all methods that *retrieve data*. No added value would come out from specifying the *data updating/retrieval* model in some knowledge-representation language, probably defining an abstract concept *data operation* to contain both kinds of methods. There is no relevant relationship between the concepts of the model (i.e. they are orthogonal) and no mapping behavior needs to be attached to them (i.e. they easily map to methods in base code on a one-to-one basis).

On the other hand, a more complex scenario like the architectural view presented before calls for making the conceptual model *explicit*. In this case, there are relevant relationships between model entities and, a mapping from base-code elements to concepts needs to be specified.

Due to this need to represent relationships among concepts and complex mapping to base-code elements, we have chosen explicit over implicit models for our framework.

4.PROOF-OF-CONCEPT IMPLEMENTATION

Following the work that began in [14], our implementation of the ideas presented in the previous section has been done using the SetPoint tool for .NET. SetPoint was one of the first AOP tools to explicitly attack the pointcut fragility problem. The context access problem was mentioned in [14] as potential future work, which naturally led to adding support for conceptual context access in the new version of the tool.

SetPoint was developed using the C# language. It works in two steps. First, assemblies are preprocessed, so that method calls can later be intercepted. Then, during runtime, the SetPoint engine (an assembly itself), analyzes each method call (i.e. join point) to determine if it belongs to any of the declared pointcuts (see [13] for a similar approach). This new version of SetPoint is being developed using Microsoft's Phoenix framework.

4.1 Intermediate model

SetPoint uses object-orientation as the knowledge representation formalism for the intermediate conceptual model. Together with code annotations, they constitute the mechanism for decoupling aspects from base code.

The model consists of what we call *concepts*. We represent them with C# *interfaces*. There are basically two kinds of concepts: *actions* and *entities*. Entities have *entity properties*; actions, in turn, have *action roles* (played by entities). Actions represent system-wide events in the model. They are basically the conceptual equivalent of *join points,* allowing aspect developers to refer to points in the flow of a program in a more conceptual way, decoupled from low-level implementation details. Entities represent concepts in the application domain. The interfaces that model actions and entities merely define their action roles and entity properties, respectively.

All transformations from base code to the model are made by *mappers*, C# *objects* that implement the above-mentioned interfaces. *Action mappers* implement, in terms of *low-level join points*, the .NET properties[2] that represent action roles . Low-level join points are constructs that reify runtime events, such as method calls or constructor calls: they contain the message sender, receiver, arguments and the selector (following pure object oriented terminology). *Entity mappers*, in turn, wrap base-code objects or groups of objects. They therefore have access to all their public members, in order to implement getters and setters for properties in the interface.

Lastly, code *annotations*, implemented by .NET attributes (stored as program metadata), relate base-code elements (such as methods or classes) to the model. Attributes are actually special classes, so whole attribute hierarchies can be defined, with different behavior for each of their members. As we did with mappers, we have chosen to differentiate entity and action annotations.

5.EXAMPLE

We will use a reduced banking application as a simple example. The application domain can informally be specified as follows (bracketed words and phrases correspond to semantic concepts):

```
There are [accounts].
Each [account] has an [account number].
An [account number] is a [string].
Each [account] has a [balance].
A [balance] is a [rational number].
There are [operations] that [update] an
[account]'s [balance].
Each [operation] has a [name].
A [name] is a [string].
A [savings account] is a kind of [account]
```

The requirement that we intend to implement as an aspect is conceptually defined as follows:

```
After an [operation] that [updates] a [savings
account]'s [balance], if that [savings account]'s
[balance] is negative then write in the system
event log the [account number], the [operation]'s
[name] and the [balance].
```

As said, the domain is modeled using interfaces. This means that, each *entity* or *action* is represented with a .NET interface. As an example, we show here the interface that conceptually represents an operation that modifies an account's balance:

```
interface IBalanceUpdate : IAction {
        public IAccount Account {get;}
        public string Name {get;}
        public float Amount {get; }
}
```

Similarly, the following couple of interfaces specify the savings account concept:

```
interface IAccount : IEntity {
        public string AccountNumber {get;}
```

[2] .NET Properties are a native construct of .NET similar to fields, except that getter and setter methods must be defined to access their value. They behave analogously to their EJB counterpart.

```
        public float Balance {get;}
}
interface ISavingsAccount : IAccount { }
```

Interface `IBalanceUpdate` has three .NET properties to represent action roles, namely an account, a name and a balance, as specified in the informal domain description. Interface `ISavingsAccount` has two properties: its number and its balance (all inherited from interface `IAccount`). The mapping from a base code to the resulting model is implemented via annotations and mappers. A possible base-code scenario is the following:

```
class SA {
        float Balance;
        string AccountNumber;
        void Withdraw (float Amount) {…}
        …
}
```

A straightforward way provided by SetPoint to map this class to its conceptual counterpart is to make class *SA* to directly implement interface `ISavingsAccount`. This would require adding to the class .NET properties with the corresponding name. A less intrusive mechanism is to create a mapper and associate it to the class through an *entity annotation*. The following code shows an example for the latter option:

```
[EntityAnnotation("ISavingsAccount", "SAMapper")]
class SA {…}
```

`Withdraw` is an operation that changes an account's balance. It should therefore be annotated, so as to relate it with the corresponding concept in the abstract model. The annotation's attribute must state that this is an *action* of type `IBalanceUpdate` and also point to the *mapper* to be used for the abstraction. We should now annotate the entities that correspond to this action's roles. Roles *Account* and *Amount* can be directly annotated in the base-code: the instance *receiving* message *Withdraw* should be mapped to role *Account*; role *Amount* can be mapped by annotating the `Amount` parameter of the aforementioned method:

```
class SA {...
  [ActionAnnotation(
        "IBalanceUpdate",
        "BalanceUpdateMapper"
  )]
  [ReceiverRoleAnnotation(
        "IBalanceUpdate",
        "Account"
  )]
  void Withdraw(
        [RoleAnnotation(
                "IBalanceUpdate",
                "Amount")
        )] float Amount
  )
        ...
}
```

Using annotations and conceptual definitions as input, SetPoint statically generates the C# code that implements mappers, automatically generating mapping behavior when the necessary information is available (i.e. when it is not, the developer must manually write the mapping code). In this case, SetPoint automatically constructs the `BalanceUpdateMapper` class using the information provided by the *entity annotation*. Using *role annotations*, direct mappings are generated for the `Account` and `Amount` properties. On the other hand, the implementation of property `Name` must be manually added:

```
class BalanceUpdateMapper:JoinPoint,
        IBalanceUpdate{
   public IAccount Account{
        get{
                //Automatically generated
                return new SAMapper(this.Receiver);
        }
   }
   public string Name{
        get{
           //Manually added
           return this.Message.Name;
        }
   }
   public string Amount{
        get{
                //Automatically generated
                return  this.GetParameterValue(1);
        }
   }
}
```

Action wrappers inherit from the *JoinPoint* class. This class abstracts low level context-access knowledge such as *receiver*, *sender* and *message*.

So far, we have shown how the domain model is defined and how elements in the base code are mapped to conceptual entities and actions. The next step is to present how aspects interact with these structures. We therefore specify a pointcut, an advice, and the corresponding aspect. It's worth mentioning at this point that we choose to use a familiar and known AspectJ-like notation since SetPoint's notation is still under refinement.

```
aspect BalanceLogging{
        pointcut SavingsAccountBalanceUpdate(
                out IBalanceUpdate bu
        ){
                Action(bu) &&
                Role[Account](ISavingsAccount)
        }
        after(IBalanceUpdate bu):
                SavingsAccountBalanceUpdate(bu){
                ValidateSABalance(bu)
       }
        void ValidateSABalance (
                IBalanceUpdate bu
        ){
                if(bu.Account.Balance < 0){
                    Debug.WriteLine(
                        String.Format(
                            "After operation {0}
                            the balance of
                            account {1} is {2}",
                            bu.Name,
                            bu.Account.AccountNumber,
                            bu.Account.Balance
                        )
                    );
                }
        }
}
```

Pointcut `SavingsAccountBalanceUpdate` establishes that whenever a message annotated as *performing a balance modification action* is reached, the corresponding mapper will be instantiated. The balance-logging aspect is what we have termed model-based aspect, since both pointcuts and advices are defined in terms of a conceptual domain representation. Context access (in this case, account balance update) is specified at a high level of abstraction, alleviating the advice fragility problem presented before.

Let us assume now that the base code evolves, and in a future version the account number is not directly represented by a field in class SA, but obtained as the concatenation of two fields, namely CustomerID and AccountSuffix. All we would need to do is adapting the corresponding property getter in class SAMapper. Both pointcut and advice would remain oblivious to this change, alleviating the advice fragility problem.

6.RELATED & FUTURE WORK

6.1 Context access

Reflection is used as the mechanism for context exposure and composition of PCDs in Josh [5]. But the metaprogramming skills needed to write context access code may become quite complex. In this respect, [17] makes an interesting point about the relationship between AOP and reflection: AOP engines should be built on top of reflection libraries, so that metaprogramming becomes intuitive. Our approach follows this line, thus we use reflection, but we avoid exposing its inherent complexity to the aspect programmer.

6.2 Intermediate Layer

Other approaches also intend to solve the pointcut fragility problem by increasing the expressive power and abstraction level of pointcuts [6][9][8], but the lack of a more abstract and semantic view than the base code itself greatly limits the power of the proposed solutions.

The Model-Based Pointcuts approach [9] also advocates for relying on an intermediate abstract layer in order to decouple aspects from base code. In this model, source-code entities that address the same concern are grouped together in views using logic programming concepts such as predicates, instantiation and pattern matching. AOP technology can be used on top of this model, making PCDs less dependent on the low-level structure of base code. However, these views are generated directly from the base-code syntax, rather than live the semantic world (i.e. domain representation) as in SetPoint. Our approach makes evolution much easier because it represents a higher level of abstraction.

We are not aware of any work that has yet focused on what we have called the advice fragility problem.

6.3 Other metadata approaches

The authors of Compose*[16], from the Composition Filters approach, propose the use of metadata entries, to tag base code elements with design information. The main difference with our approach is that this proposal offers no structure for metadata. There are no relationships between tags, so no complex domain model can be expressed.

The notion of collaborations, roles, and composition of different views is also exploited in the collaboration-based design approach Object Teams [11]. A new kind of collaboration module called Team is introduced so as to capture multi-object collaboration. This new module basically combines properties of packages and classes, containing inner classes where each element implements a role in the collaboration. Teams are composed from the base code by binding each of their roles in collaboration to a base class through an explicit mapping. In SetPoint, it is also possible for two or more base-code entities to collaborate to form a single concept in the semantic world with a proper mapping.

More similar to SetPoint, but in a different context, Tuna [18] refer to tags that are part of a model that lets the annotator make use of knowledge-representation semantics. Instead of AOP, metadata is used in this case to enrich program semantics, so as to, according to the authors, bridge the gap that exists between MDA and the XP development methodology. In the same spirit, Chris Welty's PhD dissertation [19] presents a source code *ontology* [20], which should allow maintenance coders to more easily browse an application they had never seen before, taking into account code entities annotated by original developers using this ontology.

6.4 Entities and mappers

There are several works introducing the use of interfaces in order to obtain more reusable and general aspects [15][4][10]. In [10], aspect implementation and aspect binding are specified in different modules, and interfaces glue them together to form reusable aspects. These interfaces, called collaborative-interfaces (CI), specify what the aspect provides to the context in which it is applied, and also what the aspect expects from that context. Although CIs help in decoupling base code from aspect code, they do not constitute an intermediate layer as in SetPoint. That is to say, interfaces in SetPoint define a semantic world, playing a totally different role in the development process.

6.5 Future work

The next challenge for SetPoint is to keep refining the interfaces, wrappers and annotations model. We will continue investigating the expressive power of interfaces as a domain representation. We are also improving automatic wrapper generation.

To the best of our knowledge, there is currently no AOP tool based on an intermediate layer that uses both predicates and metadata for relating base code to the abstract model. We think a combined approach could be beneficial and are therefore considering the possibility of adding contract-like predicates to our modeling formalism.

7.WORKSHOP RELEVANCE

We have first described the advice fragility problem, which makes AOP applications hard to *evolve*. This problem lies in the advice code need for accessing context.

As proposed by [17], we make use of *reflection*, but hide its inherent complexities from the base-code developer's eye.

The solution we have proposed relies on an intermediate, more conceptual, layer that stands between the aspects and base code. This layer is linked to the base code through annotations and mappers, which constitute the *metadata* needed to perform model-based AOP.

8.CONCLUSION

Decoupling aspect code from base code is crucial to improve aspect-oriented software evolution. In this respect, several approaches claim the need for aspects to refer to base code from an intermediate, more abstract point of view. Keeping this in mind, we had developed the first version of SetPoint, where the abstract view was based on a representation of the domain. However, this version was not expressive enough to accommodate context access in the domain-based abstract view; this model did not allow behavior mapping. As a consequence, it was not suitable for addressing the advice fragility problem. In this direction, we now present a new version of SetPoint, where we propose a practical approach to this problem. A more complex mapping mechanism consisting of annotations and mappers provides a domain-based abstract view with context-access capabilities. In this model, not only do pointcuts reside on this view, but advices too, resulting in what we call model-based aspects.

9.REFERENCES

[1] P.Tarr, M.D'Hont, L.Bergmans and C.V.Lopes. *Requirements on, and Challenge Problems For, Advanced Separation of Concerns*. Workshop on Aspects and Dimensions of Concern. ECOOP 2000.

[2] C.Koppen and M.Stoerzer. *Pcdiff: Attacking the fragile pointcut problem*. EIWAS 2004

[3] R.Filman and D.Friedman. *Aspect-oriented programming is quantification and obliviousness*. Advanced Separation of Concerns. OOPSLA 2000.

[4] W.G. Griswold, K.Sullivan, Y.Song, M.Shonle, N.Tewari, Y.Cai and H.Rajan et al. *Modular Software Design with Crosscutting Interfaces*. IEEE Software, vol. 23, no. 1, pp. 51-60, Jan/Feb. 2006.

[5] S.Chiba and K.Nakagawa. *Josh: An Open AspectJ-like Language*. AOSD 2004.

[6] M.Eichberg, M.Mezini and K.Ostermann. *Pointcuts as Functional Queries.* APLAS 2004.

[7] K.Gybels and J.Brichau. *Arranging Language Features for More Robust Pattern--Based Crosscuts*. AOSD 2003.

[8] H.Masuhara and K.Kawauchi. *Dataflow Pointcut in Aspect-Oriented Programming*. APLAS 2003.

[9] A.Kellens, K.Mens, J.Brichau, and K.Gybels. *Managing the Evolution of Aspect-Oriented Software with Model-based Pointcuts*. ECOOP 2006.

[10] M.Mezini and K.Ostermann. *Conquering Aspects with Caesar*. AOSD 2003.

[11] S.Herrmann. *Object Teams: Improving Modularity for Crosscutting Collaborations*. Proc. of International Conference NetObjectDays. 2002.

[12] R.Altman, A.Cyment and N.Kicillof. *On the need for SetPoints*. EIWAS 2005.

[13] R.Douence and M.Sudholt. *A model and a tool for event-based aspect-oriented programming (EAOP)*. Technical Report 02/11/INFO, Ecole des Mines de Nantes. 2002.

[14] R.Altman and A.Cyment. *SetPoint: a semantic approach for the pointcut resolution in AOP*. Msc. Thesis, Universidad de Buenos Aires. 2004.

[15] G.Kiczales and M.Mezini. *Aspect-oriented programming and modular reasoning.* ICSE '05.

[16] C.Noguera García. *Compose * A Runtime for the .Net Platform.* Msc. Thesis, University of Twente, 2003.

[17] G.T.Sullivan. *Aspect-oriented programming using reflection and meta-object protocols.* Comm. ACM, 44(10):95–97, 2001.

[18] C.Zimmer and A.Rauschmayer. *Tuna: Ontology-Based Source Code Navigation and Annotation*. Workshop Ontologies as Software Engineering Artifacts in OOSPLA 2004.

[19] C.Welty. *An Integrated Representation for Software Development and Discovery*. Ph.D. Thesis, Rensselaer Polytechnic Institute. 1996.

[20] T.Gruber. *Toward Principles for the Design of Ontologies Used for Knowledge Sharing*. International Workshop on Formal Ontology, 1993.

[21] W.Cazzola, S.Pini and M.Ancona. *Evolving Pointcut Definition to Get Software Evolution*. RAM-SE'04-ECOOP'04 Workshop on Reflection, AOP, and Meta-Data for Software Evolution. 2004.

[22] I.Nagy, L.Bergmans, W.Havinga and M.Aksit. *Utilizing Design Information in Aspect-Oriented Programming*. Proc. of International Conference NetObjectDays, NODe2005. 2005.

[23] C. Clifton and G. T. Leavens. *Obliviousness, modular reasoning, and the behavioral sub typing analogy*. Technical Report TR03-01a, Iowa State University. 2003.

[24] E.Gamma, R.Helm, R.Johnson and J.Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley. 1994.