On the need for *setpoints*[1]

Rubén Altman
Departamento de Computación,
Universidad de Buenos Aires
Buenos Aires, Argentina
raltman@dc.uba.ar

Alan Cyment
Departamento de Computación,
Universidad de Buenos Aires
Buenos Aires, Argentina
acyment@dc.uba.ar

Nicolás Kicillof
Departamento de Computación,
Universidad de Buenos Aires
Buenos Aires, Argentina
nicok@dc.uba.ar

ABSTRACT

Pointcuts are currently defined in most aspect-oriented frameworks either by way of laborious enumeration or by referring to some structural property of the code. This coarse way of quantifying reduces program evolvability, supposedly one of the advantages of AOP. We believe that a strong decoupling of concerns will only be achieved when pointcut definition mechanisms are provided that rely on system views other than the program code. Structure-based pointcuts must evolve into *setpoints,* semantic pointcuts. It is the main purpose of this paper to further explain this concept and to present SetPoint, an AOP environment developed as proof-of-concept for these ideas.

General Terms

Design, Experimentation, Languages.

Keywords

Aspect-Oriented Programming, Traceability, Semantics, Pointcut, Ontology.

1. INTRODUCTION

Quantification has been repeatedly pointed out as one of the pillars of AOP. As stated in [1], aspect oriented programming can be defined as the "desire to make programming statements of the form *In programs* P, *whenever condition* C *arises, perform action* A". It is on the universe over which predicate C ranges that we that we want to focus.

Quantification capabilities are represented by what is widely known as *pointcut declarations*, i.e. sets of well-known points during the execution of a program [2]. Pointcuts are currently defined in most aspect-oriented frameworks either by laborious enumeration or by referring to some structural property of the code (which can be a combination of other, more simple properties), such as particular naming conventions, coding conventions and coding patterns (e.g., "create a log entry before executing any method whose name matches the `get[(a-z)*]` regular expression"). [3] exposes the unexpected consequences of this coarse way of quantifying, concluding that "AOSD leads to software that should be more robust with respect to evolution because it offers better modularization, but paradoxically reduces the evolvability because it introduces tight coupling".

We believe that these coupling stems not only from the lack of obliviousness as presented in [1], (i.e. base-code programmers not knowing that aspects will be applied to it), but also from the fact that aspect writers need to be completely aware of base-code details and evolution. Therefore, we call the former requirement one-way obliviousness, and we extend the definition of obliviousness to consider both directions. In our view (two-way) obliviousness is the requirement that base-code programmers must not be aware of the future addition of aspects, and aspect programmers do not need to know structural and syntactic particulars of the base-code or adapt their pointcuts to changes in base-code structure or naming (refactoring).

Beginning with Section 2, we grow out of these ideas by describing archetypical scenarios of tight coupling caused by the use of structure-based pointcuts.

We outline in Section 3 what we term *semantic pointcuts*. Section 4 briefly introduces **SetPoint,** our first implementation of this idea. We conclude in Section 5, presenting conclusions and further work on the subject.

2. ARCHETYPICAL SCENARIOS

2.1 Frameworks

Some of the problems that arise when trying to use aspects with white-box frameworks have already been described in [3]. If aspects need to be applied to a program based on a black-box framework, the lack of knowledge of framework internals would prevent the use of structure-based pointcuts. Learning framework details is not an option, both because it would break framework encapsulation, and because it would not be an two-way oblivious solution. The only remaining option would be to apply aspects only on the framework façade.

2.2 "Name explosion"

[3] already argues against refactoring as a way of generating useful pointcut definitions. Evidently the problem becomes harder as more aspects come into play. What is more, the resulting base code no longer tackles only the main (i.e. functional) concern, but is also shaped so that crosscutting-concerns can be effectively weaved.

We could then follow strict naming conventions, so that names describe a given concern. But problems arise. To begin with, we are suffering from an evident lack of obliviousness. Furthermore, the more aspects we have the more rules need to be followed to make method names belong to the proper pointcut. In the worst case, some rules can even be incompatible with one another. In a typical case (i.e., adding a substring at the end of the method name is usually enough), identifiers can become extremely long, dramatically decreasing code legibility. We call

[1] This work was partially funded by Microsoft Research

this problem “name explosion” because we find it similar to the one described in the decorator pattern [5].

Furthermore, forcing part of a method name to signal what aspect or aspects should be applied to it violates the **intention revealing naming rule**. It is considered a coding best practice by the Object Oriented Programming community that method names clearly represent the intention of the method code [6]. A good example might be taking a method named *transferAmount* (that naturally conveys the functional intention behind the method code), which becomes *transferAmount_concurrent_beginsTransaction_traced* when crosscutting concerns need to be applied on it.

2.3 Ambiguity

Any syntactic (i.e. structure-based) solution brings about ambiguity problems. Let us suppose that we use method names to distinguish between join points that belong to two different pointcuts: pointcut “A” (related to the persistence aspect) might have been written assuming that methods beginning with `set` are always setters, and pointcut “B” (related to the “user-interface” aspect) was specified considering that those same methods refer to graphics being settled on the screen. Coding conventions may be established to avoid this problem, but they will introduce the one described in sections 2.3 and 2.4.

2.4 Refactoring

Consider the following simple case study:

- Class “InternetSearcher” has two subclasses, “FastSearcher” and “CarefulSearcher”, which differ only in the searching algorithm.

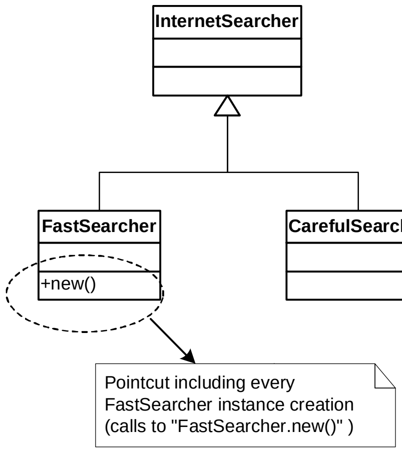

InternetSearcher
FastSearcher
+new()
CarefulSearcher
Pointcut including every FastSearcher instance creation (calls to "FastSearcher.new()")

Figure 3: Refactoring First Scenario

- After a couple of weeks, one of the designers in the development team realizes that using a model based on the Strategy pattern [5] would be a good decision. Hence, the model is changed through the use of refactoring. A “searchAlgorithm” attribute is added to the “InternetSearcher” class. Its value must be an instance of one of the classes in the “SearchAlgorithm” hierarchy. “CarefulSearchAlgorithm” and “FastSearchAlgorithm” specialize the new “SearchAlgorithm” abstract class, while the existing “InternetSearcher” subclasses are deleted.

Now, consider that during the development of the first stage someone wanted to improve the performance of “FastSearcher” by applying a “load-balancing” aspect on every “FastSearcher” instance creation. The pointcut would become completely useless after second-stage refactoring. The class “FastSearcher” no longer exists. Consequently, either the pointcut writer or the AOP environment would have to address this situation.

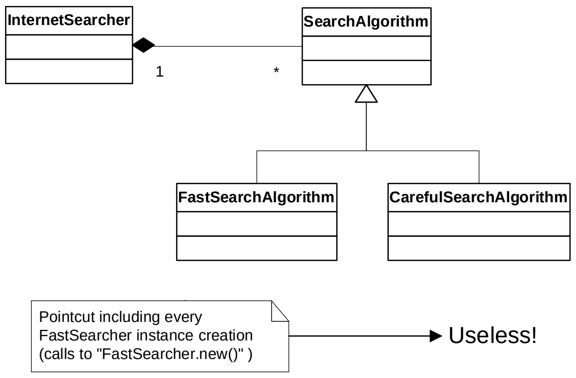

InternetSearcher
SearchAlgorithm
1
*
FastSearchAlgorithm
CarefulSearchAlgorithm
Pointcut including every FastSearcher instance creation (calls to "FastSearcher.new()")
Useless!

Figure 4: Refactoring Second Scenario

The former approach forces the aspect writers to be aware of code changes, thus violating two-way obliviousness as we have defined it. The latter is highlighted by some works [7] as a way to achieve better pointcut definition mechanisms. We agree on the importance of better AOP-programming environments, but we feel that without a real change in pointcut definition mechanisms, better tools would become just a transient patch.

3. TOWARDS A SOLUTION

We strongly believe that the main cause of the preceding problems lies in the use of structure-based properties and syntactic conventions. Intuitively, when a software engineer thinks about applying an aspect, she thinks about *“messages sent to an object of the* [data layer] *handled by* [referentially transparent] *methods”* or “[numerical operations] *whose operands are* [money] *fields of* [business objects] *and which are performed within the scope of a* [transfer] *transaction”*, and not about “*methods with names satisfying the* `set*` *pattern”.* She is thinking in terms of program views and not particularly about code itself. It therefore seems sensible enough to provide pointcut definition mechanisms that predicate over these views. Two main challenges arise: defining program views and relating these views to the program code.

3.1 Program Views - Semantics

Software views are descriptions of a program that are focused on issues relevant to a specific stakeholder, and are thus written in a language she can understand. Thus they are often expressed in a notation that provides a higher level of abstraction than programming languages. Views may be seen as a way of clarifying a program, exposing its meaning (with an interpretation of *meaning* suitable to each specific stakeholder); that is to say, making the program semantics explicit.

We considered two different approaches for program views representation: general-purpose knowledge representation languages, and a set of domain-specific languages [8]. The pros and cons of these options are the usual ones when dealing with general vs. purpose-specific solutions. Despite the widespread adoption of notations to represent particular program views, such as use-cases, software architecture description languages or finite-state machines, we opted to explore the former option first, so as to gain flexibility in these early prototyping stages.

3.2 Program Annotations and *setpoints*

We chose to link p*rogram semantics or views* to base-code through metadata elements. Code annotation mechanisms such as the ones defined in the .NET and Java environments may be used for this purpose. Pointcut definitions will be based on these annotations, which must belong to a *semantic model*, thus relating the program to its meaning.

We call this new kind of pointcuts *setpoint*s (i.e. semantic-based pointcuts). In the following section we will describe why this approach solves the archetypical problems presented before.

3.3 Archetypical scenarios revisited

Incompatibility problems raised by different coding conventions imposed by an AOP framework combined with another **development framework** do not exist in the presence of setpoints, since the proposed AOP framework does not require coding conventions to be in place.

Semantically annotated **black-box frameworks** can be managed without any additional problem by the semantic AOP framework, no matter what their internal structure looks like. The only requirement posed by our AOP tool is that their code should be linked to well-known semantic models (either internally or externally).

Method names are clearly not affected by aspect application if a semantic-based AOP tool is used. This clearly solves the **name explosion** scenario.

Ambiguity is avoided as well; given the fact that program metadata will reference concepts that belong to a specific semantic model. Their meaning will then be uniquely defined.

Finally, as far as **refactoring** is concerned, pointcut definitions will no longer refer to a specific class name or relationship among classes, but to class semantics. Thus, usage of either inheritance or composition makes no difference to the AOP framework: following the example presented in the previous section, the pointcut definition would no longer specifically refer to class FastSearcher, but to any classes that have the purpose of searching the Internet prioritizing response-time. Any **refactoring** that preserves this semantics would not affect the aspect effect. Neither would the pointcut need to be altered if other classes implementing a faster algorithm were added, as long as they have a clearly defined semantics.

4. SETPOINT!

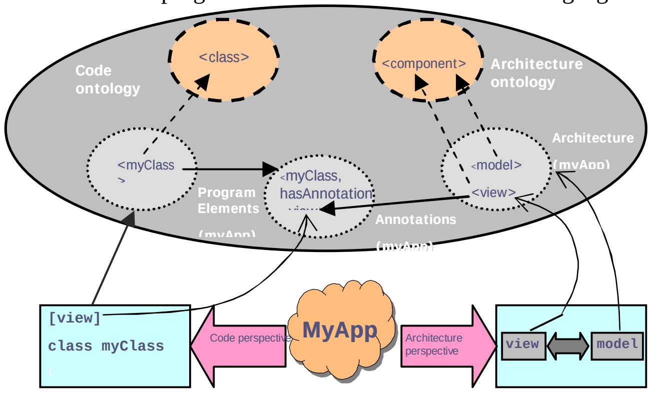
The Setpoint Framework, built using Microsoft's .NET platform, has been developed as a proof of concept for our ideas on semantic pointcuts.

Formal ontologies [9] were used as the means to represent different program views in a uniform way. The framework relies on widespread W3C standards to represent ontologies (OWL [10] and RDF [11]). These standards were adopted to minimize the risk in the choice of a general-purpose language: since OWL and RDF are designed to be used throughout the semantic web [12], they have been widely tested in the representation of dissimilar knowledge-bases. The existence of a number of tools designed to manage W3C models also makes it easier for us to deal with these standards, isolated from the development of the required machinery. As ontologies are usually applied to add semantic information to Web pages, they seem a natural choice to do the same to program code. Concepts described in the ontologies are later referenced in program annotations using .NET custom attributes.

4.1 The semantic model

We rely on RDF/OWL as the means to define different program views. Thus, every knowledge model used to represent these views must be described in terms of RDF ontologies. The models must then be instantiated for a specific program. This is achieved through code annotations: the framework takes as inputs a program and its annotations and creates the proper RDF representations. This process, called *semantication*, is further explained in the architecture section.

Previously mentioned ideas about different semantic models for one program are summarized in the following figure:



One distinguished program view is always present: the code itself. Like any other view, it must be translated into a proper RDF representation. The specific resources that make up this model are called *program elements.*

4.2 LENDL

A Domain Specific Language called LENDL was developed to make it easier for developers to declare advices and pointcuts in our framework. The following code shows an example of a pointcut definition:

```
pointcut MyPointcut {
   sender is [semantics://anOntology#concept1];
}
```

This statement is translated into C# using [13] and finally parsed into a machine executable RDF query language [14]. This underlying language allows querying RDF models for the existence of relationships (either explicit or inferred ones). Therefore, the sample pointcut is a query for the existence of an RDF resource that relates the *program element* that represents the current join point sender and the concept “`concept1`”, which must have been previously defined in the ontology `semantics://anOntology`. The keyword `is` represents a built-in RDF concept, already defined in the environment.

The `receiver` and `message` keywords are also included in the language, in order to predicate about the corresponding program elements of the current join point.

4.2.1 Aspects

Any program can potentially be inserted at some point in another program’s execution; we say then that the former program is assuming the role of an aspect. The running code of an *aspect* is therefore written using any .NET compliant language. LENDL has a construct for declaring the protocol of the program that will be used as an aspect, thus exposing it to the framework as such. For example:

```
aspect MyAspect {
   event oneRelevantEvent;
   event anotherRelevantEvent;
}
```

The events must match the name of a method in the class MyAspect, so that the runtime environment can find them via reflection mechanisms.

The advices will then relate events and specific pointcuts:

```
advice myAdvice : myAspect {
   trigger oneRelevantEvent after myPointcut1;
   trigger anotherRelevantEvent before myPointcut2;
}
```

The syntax is straightforward: event `oneRelevantEvent` must be executed after the execution of any join point belonging to pointcut `myPointcut1`, and event `anotherRelevantEvent` must run before the execution of any join point belonging to `myPointcut2`.

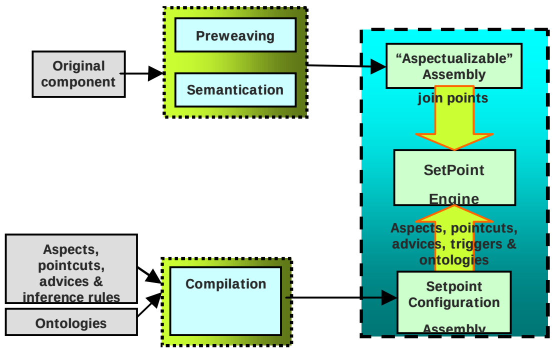
4.2.2 Inference Rules

LENDL allows the addition of inference rules that affect the specified RDF/OWL models. They modify the models by defining new relations provided some conditions are met. The following example shows the definition of a given inference rule:

```
declare CTS alias
semantics://programElements/objectOriented/CTS;
rule annotationTransitivity{
   infer B [CTS#hasAnnotation] X
      when A [CTS#hasProgramElement] B and
         A [CTS#hasAnnotation] X;
}
```

4.3 SetPoint Architecture

The following figure summarizes the architecture of the SetPoint AOP framework:


Original component
Preweaving
Semantication
“Aspectualizable” Assembly
join points
SetPoint Engine
Aspects, pointcuts, advices, triggers & ontologies
Aspects, pointcuts, advices & inference rules
Ontologies
Compilation
Setpoint Configuration Assembly

A process called *preweaving* and the already-mentioned *semantication* must be applied to assemblies intended to run under the AOP framework. An API called PERWAPI [15] is used as the basis to read, instrument and rewrite assemblies. These two processes achieve the following main goals:

- Create an assembly RDF representation from its type information (including attributes in their role of semantic annotations). The resulting representation is included in the assembly as an internal resource.

- Inject code to delegate message send execution to the AOP weaver on each method call.

LENDL code must be compiled together with the ontologies to be used (RDF/OWL models), so that they can serve as the main input to the runtime environment.

The weaver decides in runtime which aspect must be applied on each method call: every pointcut is evaluated to determine whether it includes the current join point or not. The weaver is also responsible for loading needed RDF-resources into memory. Sesame [16], a Java knowledge database that has been migrated to .NET by our group relying on IKVM [17], was chosen as the appropriate library to manage models in memory.

This runtime architecture implicitly describes two important design decisions we made, based on strong theoretical arguments:

- The weaving is performed dynamically at runtime.
- The only known join points are messages being sent and received.

4.4 Sample application

The SetPoint framework was tested by adding unexpected requirements to an application developed a year before starting our work on aspect-oriented programming. The application consisted of a traditional Senku game; its architecture was devised as simple model-view components that communicate with each other through events/method calls. The new (non-functional) requirements involved were as follows:

1. Logging all the messages sent between objects that belong to the Model component and objects that belong to the View component.
2. Caching some specific calculations made during the game, involving possible movements.
3. Distributing the Model and View components into different OS processes.

The Senku code was to be altered only by adding the corresponding annotations relating the source code to the defined semantic models. To achieve this, a reference to the namespace that defined the semantic annotations had to be manually added[2].

The semantic models defined using RDF/OWL include the previously mentioned Architectural view, an "Information" view, which in our case was useful to know which messages were referentially transparent, and a "Game" view, which describes properties intrinsic to a game, such as which messages imply the beginning of a new game. The Code view was also used to define the necessary pointcuts.

The following LENDL code applies the intended aspects to the Senku game:

```
declare architecture alias
semantics://perspectives/architecture;

/***********************************/
/* Logging Aspect */
/***********************************/
pointcut ViewToModelMessages{
   sender is [architecture#view];
   receive is [architecture#model];
}
aspect LoggingAspect{
   event startLogging;
   event endLogging;
}
advice LogEvents : LoggingAspect{
   trigger startLogging before {ViewToModelMessages};
   trigger endLogging after {ViewToModelMessages};
}

/***********************************/
/* Caching Aspect */
/***********************************/
pointcut ReferentiallyTransparentMessages{
   message is
      semantics://perspectives/
      information#referentiallyTransparent];
}
pointcut GameBeginningMessages{
   message is
      [semantics://perspectives/functional#startGame];
}
aspect CachingAspect{
   event lookUp;
   event flush;
   event afterLookUp;
}
advice CacheMessages : CachingAspect{
   trigger lookUp before
       {ReferentiallyTransparentMessages};
   trigger afterLookUp after
      {ReferentiallyTransparentMessages};
   trigger flush after {GameBeginningMessages};
}

/***********************************/
/* Distribution Aspect */
/***********************************/
pointcut ModelObjectsCreationMessages{
   sender is [architecture#view];
   receiver is [architecture#model];
   message [rdf#type] [cts#Constructor];
   receiver [cts#isDelegate] [cts#false];
}
aspect DistributionAspect{
   event createRemoteObject;
}
advice DistributeComponents : DistributionAspect{
   trigger createRemoteObject before
      {ModelObjectsCreationMessages};
```

The experiment was successful: the aspects were applied without the need to make any additional changes to the Senku code[3].

5. CONCLUSION

The SetPoint framework was successfully used to add distribution, performance and logging requirements to a Senku game by developers who were oblivious of the later addition of aspects to the code they had written. It was a good first attempt towards the construction of a semantics-based AOP framework. Nevertheless, the framework is still in its early stages; indispensable improvements need to be made in order to reach an industrial-strength tool:

- Access to Context: It would come in useful to predicate about join points that have already been executed (i.e. being able to refer to the call stack), as well as having access to other contextual elements, such as message parameters or globally accessible variables. Needless to say, we aim at approaching this subject under the very same principles that have guided us so far: quantification and obliviousness as a means to achieving decoupling. The goal will then be to avoid code structure-based context access (e.g. *log both the method's first parameter and returning value, only when they're both unsigned integers*), most probably by referring to the intermediate semantic layer defined so far.
- Reliabilty: The framework must be tested with a larger set of applications, to ensure that the preweaving and semantication processes work flawlessly on every single case.
- Performance: Performance issues were not prioritized in this first version, but they will undoubtedly be a priority in further releases (it is important to note that, despite not much attention was payed to this item, the test application worked within quite acceptable response levels).
- IDE and Development Tools: Better tools must be offered to software developers in order to facilitate their work. These tools include integration with current environments, debuggers, etc…
- Join Point Model: Despite message sends and receives would seem as the proper join points in a pure object-oriented environment, other primitive instructions should be considered in a hybrid platform like .NET. Exception handlers and assignments are some examples.

[2] Adding this reference will not be necessary in further versions.

[3] Actually, the Senku classes were modified in order to apply the distribution aspect, but just because of .NET remoting needs, not because a need of the AOP environment.

6. ACKNOWLEDGMENTS

We thank very much Diego Garbervetsky and Víctor Braberman for their valuable comments on the paper.

7. REFERENCES

[1] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, Minneapolis, USA, October 2000.

[2] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-oriented programming. *Proceedings European Conference on Object-Oriented Programming (ECOOP'97)*, pages 220–242, Finnland, June 1997. LCNS 1241, Springer-Verlag.

[3] On the Existence of the AOSD-Evolution Paradox, Tom Tourwe, Johan Brichau, Kris Gybels. In *AOSD 2003 Workshop on Software-engineering Properties of Languages for Aspect Technologies,* 2003

[4] R. Johnson. The dynamic object model architecture. Only available at http://st-www.cs.uiuc.edu/users/johnson/papers/dom/

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides,*Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[6] Kent Beck, *Smalltalk Best Practice Patterns*. Prentice Hall, 1996.

[7] Shimon Rura, *Refactoring Aspect-Oriented Software*. Bsc. Thesis, Williams College, 2003.

[8] Niels Christaensen, *Domain specific language in software development, and the relation to partial evaluation*. PhD. Thesis, University of Copenhagen, 2003,

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

[10] http://www.w3c.org/2001/sw/WebOnt

[11] http://www.w3c.org/RDF/

[12] Tim Berners-Lee, James Hendler y Ora Lassila, *The Semantic Web*, Scientific American, 2001

[13] http://codeworker.free.fr/

[14] http://www.openrdf.org/doc/users/ch06.html

[15] http://www.plas.fit.qut.edu.au/perwapi/Default.aspx

[16] http://www.openrdf.org

[17] www.ikvm.net