

Aspects in a Prototype-Based Environment

Thomas Cleenewerck*, Kris Gybels[†] and Adriaan Peeters
Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, 1050 Brussel, Belgium

{thomas.cleenewerck, kris.gybels, adriaan.peeters}@vub.ac.be

<http://prog.vub.ac.be/>

March 5, 2004

Abstract

Most of the existing aspect-oriented technologies are founded on a class-based object-oriented language. A whole other paradigm of object-oriented programming is thus ignored: prototype-based programming. In this paper we explore the impact of the differences of prototype-based and class-based programming on the design of crosscut languages and the implementation of weaving. Crosscut languages for prototype-based programs will definitely need to depend more on dynamic properties of the running program. In cases where some of those properties don't change too often, we propose the use of dynamically modifying code.

1 Introduction

Up to now Aspect-Oriented Programming seems to have been considered mostly in the context of class-based object-oriented languages. Class-based programs exhibit large amounts of static information that is exploited in many aspect-oriented systems. The static information about the structure of classes, their methods etc. gives crosscuts "something to hold onto" when trying to determine the points in the code they need to hook into. Weaving can be optimized by exploiting the static information, and for certain aspect languages can be done purely statically.

But besides class-based programming languages another more dynamic way of doing object-oriented programming exists: prototype-based programming. In this paper we describe a first exploration of what impact a prototype-based environment has on our notions of how aspects should be modeled, crosscuts be written and how weaving is to be done.

*This research is partially performed in the context of the e-VRT Advanced Media project (funded by the Flemish Government) which consists of a joint collaboration between VRT, VUB, UG, and IMEC.

[†]Research assistant of the Fund for Scientific Research - Flanders (Belgium) (F.W.O.)

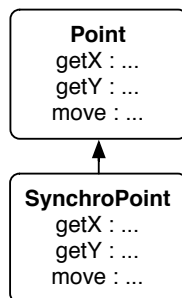


Figure 1: Point synchronization example with delegation.

2 Prototype-Based OO

Prototype-based programming differs markedly from class-based object-oriented programming in that objects are not grouped into classes that specify the methods and data members all the objects in that group support. Rather each object is entirely individual and contains its own methods and data members. Examples of prototype-based languages are JavaScript, Pic%, Self [3] and Kevo [7].

This does not mean however that prototype-based programming does not support a form of "code reuse". While it obviously does not support class-based inheritance, it has a mechanism known as object-based inheritance or better "delegation". An object usually has a link to some "parent" object. When a message is sent to the object, and the object does not have a method for the message, the message is automatically delegated to the parent. Note that delegation is different from mere forwarding in one important aspect: while it is handling the delegated request, any message the parent sends to itself is instead automatically sent to the child object [6]. Variations on this basic scheme are possible where for example objects can have multiple parents or a specific object to delegate to can be specified etc. It also may or may not be possible for the delegation link to be changed after the object is created etc.

Another difference with class-based programming is that objects are not created by instantiating a class. Rather they are constructed by cloning (or copying) an existing object. New or different methods and data fields can then be added to the new object.

3 Aspects with Delegation

Delegation is such a powerful mechanism it has been considered as a way of implementing certain aspects. The argument is that the concept of "putting code" before and after a method is well handled by delegation. Consider for example the situation depicted in figure 1. It depicts a synchronized version of a point delegating to the actual point. The methods move, getX and getY of the synchronized point simply perform the necessary synchronization steps and further delegate the message to the parent point object. The synchronized point object can easily be reused by serving as a prototype for other synchronized points. When the synchronized point is cloned - depending on the semantics of the clone operation but this can usually be overridden - the parent point is also

cloned and effectively a fully new synchronized point is delivered.

It is however fairly obvious that using delegation to implement aspects suffers from some of the same problems as using the concept of wrappers in class-based languages to do the same. It is for example difficult to use the synchronization code for other objects than points. Furthermore a particular point object may have new methods added to it at run-time. One would have to remember to make sure an appropriate synchronized method is added as well.

So we can state that despite some claims to the contrary, the essential concept of low-level aspect-oriented programming may make sense for prototype-based programming as well. It would be useful to have objects that can describe using a crosscut when exactly they are to be invoked. The problem is then what should a crosscut language for a prototype-based language be able to express?

4 A Crosscut Language

Our view of a crosscut language is that it is a query language to select join points from the set of all join points that occur in a program [4]. In order to be able to be specific about which join points one intends, the crosscut language should not just allow one to express conditions on the join points themselves, but also on any of their associated objects like message arguments, or even the state of any other objects in the system. This is especially true for a crosscut language in a prototype-based environment. To see how exactly such an environment impacts the crosscut language we will start from the AspectJ crosscut language.

As is well known, the AspectJ crosscut language has primitive predicates for picking out join points based on their type, such as method calls, method executions, static initializers etc. It also has predicates for expressing conditions on the static extent of the join points, such as the class in which it is executed and from which class that class inherits. The join point type predicates also rely in a way on static code elements such as the types of arguments. As we will discuss, this is not possible in prototype-based languages.

4.1 Class

The most fundamental difference between prototype-based languages and class-based languages is the absence of classes. Let us describe the resulting repercussions.

1. The most obvious repercussion is that static initialization is non existing in prototype-based languages.
2. Classes group similar objects together. Reasoning about a class is reasoning about all its objects, and changing its class is changing all the objects. These are two ways in which aspect languages use classes. The reasoning is done in the pointcut language and the changing is performed in the weaving engine through advices. So aspect languages basically use classes to respectively *select* and *reach* the objects. In many of the AspectJ pointcut primitives, classes are used for this purpose. Consider the following pointcut: `call(* Point.setX(int))` which selects all the calls to the method `setX(...)` of class `Point`.

Expressing this pointcut in a prototype-based environment is not so straightforward because there are only point objects available. Each instance containing its own set of methods and datamembers. Clearly we need some mechanism to reach or select the desired set of point objects.

3. Classes are the blueprints of objects. In a class the methods that the object support are written, they are thus known in advance (at compilation time) and also determined in advance and never change. Reconsider the class `Point`, it contains one method `setX(...)` at compilation time. Because that never changes, all the objects of class `Point` will also be able to receive the method `setX(...)`. Therefore we can write the pointcut to select all the calls to the method `setX(...)` to all objects of class `Point` by the following pointcut expression `call(* Point.setX(int))`. The assumption in this pointcut is that in class-based languages we know in advance how all objects will behave.

In prototype-based languages this is not the case. Each object is built ex-nihilo or by cloning a prototype object. Hence there is no static structure defining how the objects of a certain kind should look like. Each object may change the number and kind of messages it can receive and redefine its behavior. So objects are solely defined by their own set of messages they implement.

4. Classes in the case of statically typed object-oriented languages are also used as a type. There are a number of primitive pointcuts that are especially there to select join points where the currently executing object or the target object is of a particular type respectively using `this(Point)` and `target(Point)`. Used in combination with the `call` primitive pointcut, the type of variables are used to determine which calls at the caller site are included in the joinpoint and which are not.

Since the bulk of the prototype-based languages are dynamically typed, statically determining which calls to a certain object belong to a pointcut is not possible. Statically we may only rely on the signature of the method, checking if a variable is of a certain type must be performed at run-time.

5. In class-based languages objects are created by instantiating a class. Aspect languages provide primitives to select join points of instance creation or object initialization `initialization(Point.new(int, int))` or `call(Point.new(int, int))`.

Prototype-based languages however, need a very different way of creating objects since the lack of classes. Objects are either created ex-nihilo or are created by cloning another known object. In these cases it is impossible to specify a joinpoint to intercept the creation of a new kind of object.

4.2 Inheritance

A second element of the static structure is inheritance. Similar to classes, inheritance hierarchies are also known and determined at compile-time and do not change during the execution of the program. The `within` primitive pointcut is one of the primitives that is based on that property. Because hierarchies do not

change during the execution of a program, the hierarchies are a solid set and reliable set of information.

Since there are no classes in Prototype-based languages, it is not possible to define a static hierarchy based on the interrelationships of classes such as in Class-based languages. Another mechanism to specify interrelationships exists however: object inheritance. It can perfectly mimic class-based inheritance but it is much more flexible. Hierarchies can be formed, changed and broken at run-time. Moreover since these hierarchies are composed out of objects, it is also hard to refer to properties of these hierarchical structures.

4.3 Selecting Objects

As we have discussed in the previous section, crosscuts expressed in terms of the classical AspectJ pointcut primitives heavily rely on rather static structural information. In prototype-based languages the whole program structure is highly dynamic. Specifying pointcuts using these primitives is very difficult if not sometimes impossible because the only structural information that is left is the signature of messages. So the problem of specifying a point in prototype-based languages becomes actually the problem of selecting the desired objects.

Selecting the objects by merely the messages they support is often not sufficiently restrictive enough. Many popular messages are supported in many semantical totally different objects. Selecting objects using such messages would thus result in far too a heterogeneous set. For example, the following set of messages `add:`, `contain:`, `size:` are part of many different kind of objects like sets, bags and even recordsets. To further narrow a search we must therefore incorporate state information. The state of an object determines the relationship the objects has with the rest of the objects and from the perspective of the object-oriented paradigm thus also contributes to the semantics of the object.

5 A Weaving Model

It seems we should at least try to make it possible to express conditions like "when a message send joinpoint occurs to an object whose parent supports the message m". The problem with such crosscuts is of course how to weave their associated advices efficiently, since the delegation relationships, the state of objects or even the messages they support can change dynamically in a prototype-based environment.

A problem with the crosscut language for a prototype-based environment is that almost none of the conditions can be resolved statically. In some prototype-based environments, delegation relationships can even change after an object was created. New slots can be added to objects for new data fields or even methods, meaning the object's type can even be changed. This implies weaving needs to be highly dynamic as well.

The most naive way of doing weaving when one is confronted with highly dynamic crosscuts is to do all weaving at run-time. As almost any execution of a statement in a program is usually considered to be a join point for weaving, this would imply that at *every* statement the dynamic weaver would have to go through the set of *all* crosscuts to see if any match with the current joinpoint and

the current state of the program. This would obviously have an unacceptable performance impact.

5.1 Two-Phase Weaving

In previous work we and others already considered optimizing crosscuts that depend on dynamic properties of the application using techniques drawn from partial evaluation [4, 5]. The idea is basically to split weaving into a dynamic and a static phase. In the static phase the set of crosscuts that need to be checked at a join point is reduced as much as possible. Because every join point is related to some statement in the code, information about the join points at that statement that can be statically derived from that statement can be used to check whether the crosscuts can *ever* match those join points. A simple example is when one has a crosscut that captures all message send join points where the message "test" is sent with a single argument greater than three. It is very simple to let a weaver figure out that a statement where the message "nottest" is sent can never lead to join points that match the crosscut. On the other hand, a statement where the message "test" is sent could lead to join points that match the crosscut, whether they actually do needs to be checked dynamically because the argument to the message is only known at run-time. When even more static information can be derived and is actually used in the crosscut, the number of crosscuts to check at each statement can be greatly reduced.

To actually implement the two-phase weaving model one can simply rely on some of the same techniques of purely static weaving. It is not necessary to really have an explicit dynamic weaver that intercepts every statement and checks the associated set of remaining potentially matching crosscuts. When most of those sets are empty, it is much better to simply wrap statements that have a non-empty set of potentially matching crosscuts. The code that would be wrapped around those statements specifies the set of crosscuts to be checked, and tells the dynamic weaver to check those. The weaver would then of course execute the necessary advices for the crosscuts that match.

5.2 Jumping Aspects Revisited

While generally applicable, the two-phase weaving model for turning a naive weaver into a more efficient one is too simple to handle most jumping aspects. The well-known jumping aspects problem refers to the problem that whether an aspect applies at a specific message send join point may depend on the calling context of the join point [2]. The `cflow` construct was added to AspectJ's crosscut language to deal with this problem. When used in a crosscut, a `cflow` specifies another (sub)crosscut and expresses that a join point needs to be preceded by another join point on the call stack that matches the sub-crosscut. The straightforward way of implementing `cflow` in a naive weaver is to let it go through the call stack and check every preceding join point against the sub-crosscut until one is found that matches or the bottom join point is reached. Turning such a weaver into a two-phased one does not optimize this process, yet a pretty simple optimization is possible. Instead of going through the call stack to see whether a join point is to be found matching the sub-crosscut, one

can set a flag when such a join point actually occurs and simply check the flag later on [5].

While the jumping aspects problem was recognized for control-flow like problems, it really applies to any kind of problem where the crosscut requires one to postpone weaving to the dynamic phase. In a sense the static location of such aspects also "jumps around" depending on conditions over the dynamic state of the program, whether it is the control-flow relationship between join points, the state of objects, their delegation relationships etc.

5.3 Blurring the Phase Distinction?

Simply postponing checks of such dynamic conditions by statically weaving if-conditions or the setting of flags into the code however may not be the only way to approach this problem. As we've already mentioned, an interesting property of most prototype-based systems is that they allow for easy replacement of methods at run-time. This would allow us to blur the strict separation of having a static weaving phase where code is modified and a dynamic phase where some of the dynamic conditions in the added code are checked. Instead, code could also be changed at runtime as a way of making the static location of aspects "jump around", quite literally in fact.

The way this would work for a `cflow` example is as follows. Recall the `cflow` is a condition of some crosscut, and the `cflow` itself specifies a sub-crosscut. As described earlier this can be woven with code that sets a flag when a join point is encountered that matches the sub-crosscut, and code that would check that flag at the static locations of join points that might match the whole crosscut (as well as any other dynamic conditions of that crosscut). We could change this to work like this: instead of setting a flag when the sub-crosscut is matched, we could change *at run-time* the code where we would need to check for the match to the whole crosscut as before, minus the check for the flag.

While this approach of literally making code jump around is quite a bit of overkill for something like a `cflow` it may be interesting to consider for other kinds of dynamic conditions. The problem with the `cflow` is that it would probably require too many code jumps as any `cflow` condition is one that quite often switches between being true and false. However, other kinds of dynamic conditions, such as for example delegation relationships between objects in prototype-based programming may not change that often. Let us consider a crosscut that would express something like "all message send join points of the message M to an object whose delegation parent is X" as a rather abstract but useful example. In this case it might be more interesting to weave this by changing the message M of any object that has X as parent, as well as changing this weaves dynamically when X gains or loses new delegation children, than to weave a check at every method M because any object might have X as parent at some point.

6 Position

While we realize our discussion of aspects in prototype-based environments is very preliminary, we do feel we have valuable contributions to make to the workshop's topic of discussion:

- We offer a "fresh" perspective on the relationships between aspects, objects and dynamic joinpoint models by considering the more dynamic variant of the object-oriented paradigm, prototype-based programming. Taking this paradigm as a starting point for further exploration may lead us to automatically consider more dynamic features for crosscut languages.
- We are interested in discussing what exactly the impact is of not having classes or types for crosscuts to grab onto will do to how we specify crosscuts. Will they naturally need to be more dynamic or can anyone offer a counter-view? Has aspect-oriented programming not been too depending so far on having static, rigid lexical structures available to specify crosscuts? One potential area of investigation is the use of query languages drawn from the field of object-oriented database systems [1]. Another potential area is to use inductive logic programming techniques to classify objects.
- Is the idea of literally making "jumping aspects" jump around by doing dynamic code changes for aspects that don't require jumping too much but would otherwise require a lot of dynamic checks a worthwhile technique to further explore? What support would such weaving exactly require from the run-time environment?

Acknowledgments

We would like to thank Jessie Dedecker, Johan Fabry and Dirk Deridder for the interesting discussions and insights they provided us.

References

- [1] Query by example, <http://www.db4o.com/>.
- [2] Johan Brichau, Wolfgang De Meuter, and Kris De Volder. Jumping aspects. In Peri Tarr, Maja D'Hondt, Christina Lopes, and Lodewijk Bergmans, editors, *International Workshop on Aspects and Dimensional Computing at ECOOP*, 2000.
- [3] Randall B. Smith David Ungar. Self: The power of simplicity. *LISP AND SYMBOLIC COMPUTATION*, 1991.
- [4] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the Second International Conference of Aspect-Oriented Software Development*, 2003.
- [5] Hidehiko Masuhara, Gregor Kiczales, and Chris Dutchyn. Compilation semantics of aspect-oriented programs. In Gary T. Leavens and Ron Cytron, editors, *Foundations of Aspect-Oriented Languages Workshop at AOSD 2002*, number 02-06 in Tech Report, pages 17–26. Department of Computer Science, Iowa State University, 2002.
- [6] Klaus Ostermann and Mira Mezini. Object-oriented composition untangled. In *Proceedings OOPSLA '01, Tampa Bay, FL*, 2001.

- [7] Antero Taivalsaari. *A Critical View of Inheritance and Reusability in Object-oriented Programming*. PhD thesis, University of Jyväskylä, 1993.