

Separating Behavioral Concerns With Predicate Dispatch,
or,
If Statement Considered Harmful

Doug Orleans
College of Computer Science
Northeastern University
360 Huntington Ave, 161CN
Boston, MA 02115, USA
dougo@ccs.neu.edu

Category: B. Programming languages design and implementation.

The behavior of a program is implemented in units that specify two things: what to do, and when to do it. The degree of flexibility that the programming language allows in specifying the second part determines how well behavioral concerns can be separated. For example, in a procedural language, you can assign names to procedures so that a procedure runs when its name appears in the operator position of an application expression; if a procedure needs to do different things depending on the run-time types of its arguments, you have to dispatch on the type explicitly (with a switch statement or the like). In a typical object-oriented language, you can assign names to methods and attach them to a classes, so that when a message matching a method's name is sent to an instance of its class, the method is executed; if a method needs to do different things depending on the run-time types of the message arguments, you again need to do this dispatch explicitly (the binary methods problem). In an OO language with multiple dispatch, you can specify a list of specializer types for the arguments of a method and attach it to a generic function, so that when that function is invoked with arguments that are instances of the specializer types, the method is executed. Ernst et. al. [1] generalized this even further with predicate dispatch: a method attached to a generic function can have an arbitrary predicate over its arguments that determines when the method should be executed; for example, the "equals 1" and "integer greater than 1" clauses of a factorial function can be separated into two methods. By allowing more precise declaration of the "when to do it" part of a behavioral unit, we remove the tangling that would otherwise occur if the dispatch had be implemented explicitly. (One benefit of separation of concerns is that we can easily add a new behavioral unit, e.g. a "less than 1 or not an integer" method that raises an error, without modifying the existing behavioral units.)

A complication with this strategy is that there might be multiple

behavioral units that want to run at the same time; for example, in single-dispatch OO, methods with the same name on a subclass and a superclass are both applicable when a message with that name is sent to an instance of the subclass, because it is also an instance of the superclass. In this case, the method on the subclass has precedence, and is executed; however, this method can resend to the superclass method. (Note that this method combination mechanism is another form of tangling removal, because otherwise the code from the superclass method would have to appear in the subclass method as well.)

Predicate dispatching generalizes the precedence relation between methods from receiver subtype to logical implication of predicates: if one method is always applicable when another method is run, but not vice versa, then the second method is executed first. If two methods are applicable, and neither predicate implies the other (or they both imply each other, i.e. the "when to do it" conditions are the same), then a "message ambiguous" error is raised during method dispatch. This leaves us with another source of tangling, however; if both predicates imply each other, then the methods must be combined into one method, and if neither predicate implies the other, then another method must be added to explicitly dispatch between the two. Even worse, suppose a method A wants to run before either of two other methods B and C, whose predicates are disjoint: since A is always applicable when either B or C is applicable, both B's and C's predicates must imply A's (and not vice versa), thus they will both take precedence over A. Instead the code in A must be copied into both B and C.

A simple change that fixes many of these kinds of situations is to have two different kinds of methods, plain methods and "around" methods. We can then extend the precedence relation so that all "around" methods take precedence over all plain methods. This captures the notion that some behavioral concerns modify others, rather than just providing functionality.

By now you've probably figured out the punchline of this paper: with a simple extension to predicate dispatching, we can support the clean separation of cross-cutting concerns, which is one of the goals of aspect-oriented programming. (Another goal is the modularization of cross-cutting concerns, which is more than just separation; modularizing predicate dispatch methods is an active area of research, and I will not cover it in this paper.) To further illustrate this connection, here is an implementation of a simplified form of the instance counting aspect from the AspectJ primer [2] in the language of Ernst et. al. [1] extended with the "around" method modifier (and the "proceed" construct):

```

predicate CountedType(c@Class)
  when c == Foo or c == Bar or c == Goo or c == Doo;
around method new(c@CountedType) {
  let instance := proceed();
  incCounter(InstanceCounter, c);
  return instance;
}

class InstanceCounter { table:HashTable };
method incCounter(ic, c@Class) when ic == InstanceCounter {
  put(ic.table, c, getCount(ic, c) + 1);
}
method getCount(ic, c@Class) when ic == InstanceCounter {
  get(ic.table, c);
}

```

Now, taking some cues from AspectJ, we can generalize predicate dispatching even further. Instead of methods having predicates over the list of arguments to the generic function application, we can give them predicates over join point objects, which contain the generic function, the list of arguments, the enclosing method, and the previous join point object. Whenever a generic function is applied, a join point is created, and a method is selected that is applicable to that join point and has precedence over the other applicable methods. A predicate over join points is roughly equivalent to a pointcut designator in AspectJ, while around methods are like around advice. With the information available in a join point object, predicates can express pointcuts equivalent to those designated by the "receptions", "calls", "cflow", "within", and "instanceof" primitive pointcut designators from AspectJ, as well as the wildcards of signature patterns.

The "executions" designator is a bit problematic, though. We only have one kind of join point, for receptions of messages (invocations of generic functions). Once a method has been selected by the dispatch process, there is no hook for "executions" predicates to attach to. I haven't seen a compelling example of the need for an "executions" designator, so I haven't thought too hard about how to add it, but it could be done by adding a second stage of dispatching, after a method has been selected to execute, where it again looks for methods applicable to the execution join point. It would have to be careful to avoid infinite regression though, because executing an around method that applies to an execution join point is itself an execution join point. In a sense this second stage of dispatch is at the meta level, while the regular method dispatch over reception join

points is still at the base level (although it involves some information that is usually considered meta level, like the generic function being invoked). The fact that "aspectual" dispatch uses the same mechanism as plain method dispatch seems like a nice unification-- for example, there's no distinction between components and aspects, they're both just modules of methods-- and adding a second kind of dispatch might muddy the waters a little too much.

Another important feature of AspectJ is the ability to bind variables in a pointcut. This in fact is identical to the binding mechanisms in the predicate language. A predicate can bind values to variables, either implicitly through a pattern match against a record value or explicitly with a let expression; these bindings are passed to the method body just like pointcut context is exposed to advice. Predicate abstractions allow predicates to be specified separately from a method, which is similar to named pointcuts.

To conclude: I have shown the relationship between predicate dispatching and separation of behavioral concerns, and outlined a few small extensions of predicate dispatching that allow separation of cross-cutting concerns in a manner much like pointcuts and advice in AspectJ. By unifying advice with regular methods, we have a simple semantics of program execution-- when a function is invoked, the applicable method with highest precedence is executed-- that can support very fine-grained divisions of behavior without involving weaving, wrapping, or similar mechanisms. In addition, the implementation techniques for efficient predicate dispatch [3] can be applied to aspect-oriented languages.

[1] Michael Ernst, Craig Kaplan, and Craig Chambers, "Predicate Dispatching: A Unified Theory of Dispatch". ECOOP '98, pp. 186-211.
<http://www.cs.washington.edu/research/projects/cecil/www/Papers/gud.html>

[2] "The AspectJ(TM) Primer, A Practical Guide for Programmers".
<http://aspectj.org/doc/primer/>

[3] Craig Chambers and Weimin Chen, "Efficient Multiple and Predicate Dispatching". OOPSLA '99
<http://www.cs.washington.edu/research/projects/cecil/www/Papers/dispatching.html>