A Pattern Enforcing Compiler (PEC) for Java: Use Cases for Multiple Dispatch

Authors: Howard C Lovatt, Anthony M Sloane, and Dominic R Verity
howard.lovatt@iee.org
http://pec.dev.java.net

The talk outlines use cases for multiply dispatched methods. The examples given use an extended Java compiler to implement multiple dispatch on an unmodified JVM. We term our extended compiler a A Pattern Enforcing Compiler (PEC) for Java and amongst the patterns it supports is multiple dispatch. The compiler is available for free download under LGPL.

Multiple Dispatch is found in Common Lisp and other languages and is therefore not a new design pattern. The reasons for wanting Multiple Dispatch or something similar in the context of adding methods to a class without modifying the class are well noted by many authors including Matthias Zenger and Martin Odersky and are also explained in the talk in the form of a Numeric Example. The inspiration for this implementation Multiple Dispatch comes primarily from Relaxed MultiJava but also from Nice, Sprintabout, and Matthias Zenger and Martin Odersky. This latter reference may seem strange in the context of the Multiple Dispatch pattern because it is not concerned with multiple dispatch but is concerned with adding methods to a class without having to modify the class. Adding methods to a class without modifying the class is one of the big advantages of Multiple Dispatch and also this reference discusses the important point of having a default method to call if a closer match cannot be found. Therefore this paper has influenced the design of Multiple Dispatch presented below in two key areas: having default methods and also automatically providing a default method if none is given by the programmer.

The use cases discussed in the talk are:

Simplicity: With single dispatch it is not obvious which of a series of overloaded methods will be called because of the interaction of dispatch on the receiver and static binding on the other arguments. With multiple dispatch the semantics are much simpler.

Argument Symmetry: When symmetry of arguments is expected for an operation, a Numerical Example is a given because many standard mathematical operations are symmetrical.

Increase Decoupling: When you may be tempted to use a Visitor or Double Dispatch patterns, e.g. when walking an expression tree in a compiler and generating code. See Walking an Expression Tree example below.

Specialized methods: When the user needs to add new methods that are special cases, a Shape intersect example is given. Intersect is a good example since better algorithms than a general purpose algorithm may exist for specific combinations of shapes and when the user adds a new shape a new algorithm may also be desirable.

Ultimate Extendability: When you want the ultimate in extendability; i.e. the ability to add special case methods *and* special case types, see Extensible Algebraic Datatypes with Defaults. This paper discusses the desirability of adding special case methods and special case types and presents a proposed extension to Java. Multiple-dispatch is a simpler to use alternative to that proposed in the paper; which is in essence a series of `instanceof` tests, but with some syntactic sugar in the form of an extended `switch` statement. this use case is illustrated by a Numeric Example.

Avoiding Type Specific Code: When you are using casts and/or `instanceof` operations, see Avoiding Casts example below.

Instead of Pattern Matching: In functional languages pattern matching is common; however in OO languages it isn't. The reason for this is that pattern matching and inheritance are not easily reconciled. The talk shows how multiple dispatch is a viable alternative.

Briefly, at the end of the talk, when not to use multiple dispatch is discussed.