The role of the base in aspect oriented programming

John Lamping

When should a concern be handled by aspect code or by base code? What is the difference between base code and aspect code? Sometimes it seems like there is no clear way to distinguish between what should be base code and what should be aspect code. Indeed, this has been cited as a weakness AOP compared to SOP. This note suggests that there is a clear difference between base and aspect, but that it is between basic *concepts* and aspect *code*. Both of those are crucial in both aspect-oriented programming and subject-oriented programming. The basic concepts are the common ground where different aspects or subjects (and concerns) meet. They, and the separation of responsibility among aspects, are the key design decisions in any multi-dimensional program

Programming style

There is a level deeper than the base program, a level of basic programming concepts, where subjects or aspects naturally meet concerns. Object orientation provides an example of this level. Object orientation provides both a way of decomposing a problem and a corresponding way of organizing a computation. These are the familiar design and coding parts of OO programming. The two phases correspond because an OO design and the corresponding OO program share the same basic operations — the messages that are sent between objects. It is not an accident that the first step of object oriented design methods is typically to write down what kinds of objects there should be and what operations they should support. This decides what kinds of message sends there will be, and thus decides on the vocabulary of the essential operations of both the design and the program, and on what that vocabulary should mean.

Each programming style has its analogue of message sends. For imperative programming, it is procedure call. For functional programming, it is function invocation. For logic programming, it is predicate satisfaction. In each case, the programming style provides a fundamental operation, and it provides a way of building up a vocabulary by naming different instances of the operation — different messages that can be sent, different functions that can be requested, different procedures that can be called for, different predicates whose satisfaction can be requested.

In all styles, the names of the operations are shared between the design and the implementation. The names, if the design is a good one, are ones that make sense in the problem domain, names like: “repaint”, “factorial”, “start_motor”, or “defends_piece”. Implementing the design means writing objects, functions, procedures, or predicates that can carry out the named behaviors.

Programming languages

That is where programming languages come in. A programming language provides a mechanism for defining implementations of the vocabulary defined in the design phase — a repaint method, a factorial function definition, a start_motor procedure definition, or a defends_piece predicate definition. In other words, a programming language is the tool to write definitions that will turn the conceptual vocabulary of the design into something that can actually run.

In addition, a programming language provides a way of organizing the vocabulary and the definitions. Traditionally programming languages organize them hierarchically. In Java, for example, method definitions are grouped into classes, which are grouped into packages. This hierarchical organization has two consequences. First, the same organizational structure must be used for all method definitions. Second, the entire definition of a method must be given in one place. (Inheritance does provide some

facilities for assembling method definitions, but it is a limited capability, primarily suitable for adding refinements.)

Aspects

With these concepts in hand, we can turn to aspect oriented programming. In aspect oriented programming, the basic concepts of the base programming style are still central. The fundamental operation of an AspectJ program, for example, is still a message send. Similarly, in designing an AspectJ program, the first questions to ask are still what the objects and messages should be.

While aspect oriented programming keeps the basic concepts of the programming style, it changes the program organization. It frees the programmer from the hierarchical program organization imposed by traditional languages, transcending both limitations of the hierarchy. Different structures can be used for different parts of a program, and different parts of the program can contribute to definition of a single vocabulary term.

This is possible because aspects cross-cut the base program. But since AOP keeps the basic concepts, the cross-cutting is always in terms of the vocabulary of the base design. An aspect can provide implementation definitions for vocabulary that would otherwise be scattered across the programming language hierarchy. (In AspectJ, introduce weaves can provide method definitions for several classes.) And an aspect can provide partial implementation definitions of vocabulary terms. (In AspectJ, advise weaves provide part of the implementation of the messages they advise.)

Subjectivity has similar affordances. And, it has a similar reliance on basic operations. Since correspondence between subjects allows for renaming, it doesn't require that basic operations be named the same in different subjects. But it preserves the more fundamental requirement that subjects are, in fact, talking about the same basic operations if they are to interact.

Concerns

A concern, in aspect orientation, could any coherent issue in the problem domain — having a GUI, thread safety, memory usage.

In some cases concerns align well with the basic vocabulary of the design. In this happy circumstance, only one concern is pertinent for each vocabulary term, and each vocabulary term corresponds to a different part of the concern. Notice that alignment is about how different concerns relate to the basic vocabulary. It doesn't make sense to talk about one concern being aligned with the basic vocabulary, but of concerns aligning together well. In this case, you don't want AOP.

But the world is usually more interesting than that. First of all, concerns can be superposed. Consider the concern of being able to sort a list and the concern for thread safety. The thread safety concern applies to sorting the list. They superpose. When list sorting is requested, both concerns necessarily come into play, which means that the implementation of list sorting will have to deal with both. There is no way around this kind of superposition; it is inherent in the problem domain. There is no way to fiddle with the basic vocabulary to get around it.

Second, concerns can have different granularities. Take the thread safety concern. It applies to all messages of the basic vocabulary. The programmer will want to specify different implementations of the concern for different messages, but there will probably be many messages for which the programmer wants to use the same thread safety strategy.

It is hard to implement these various concerns in a traditional program, because it requires a complete implementation for each vocabulary term to be defined one at a time, no matter how many concerns are

involved. AOP offers an alternative, because different aspects can combine to define the implementation of a single vocabulary term. And one piece of aspect code may contribute to the implementation of many vocabulary terms.

Design

So, what code should go in the base language and what should be put in aspects? If the granularity of a concern is the same as the granularity of the basic vocabulary, then it is a good candidate for implementing in the base language. There might still be reasons to put in an aspect, though, for program organization reasons. This is also one of the main motivations for subject oriented programming

But what concerns should be base and which should be handled by aspects is not an especially interesting question. Its answer won't affect the structure of the program much. The key questions in AOP, as in any multi-dimensional design are what the basic vocabulary should be, and how to get a clear separation of responsibility among concerns. The first question carries over from OO programming, and it is just as important in multi-dimensional programming, because multi-dimensional programming keeps the basic vocabulary. The second question is unique to multi-dimensional programming.

That question is the key to a successful multi-dimensional decomposition. Different dimensions can contribute code to the implementation of a single vocabulary term, and it has to be possible to combine that code into an implementation of that term. A good separation of responsibility among concerns is what makes that possible, because it means that the implementations contributed by the different dimensions are addressing different things and are thus less likely to conflict.

In summary, the multi-dimensional programmer's job is to design a good basic vocabulary and a good separation of responsibility among concerns. The language will then let the programmer separate the concerns in their code.