Aspects are technical, and they are few

Friedrich Steimann
Institut für Informationssysteme
Universität Hannover
Appelstraße 4, D-30167 Hannover
steimann@acm.org

Object-oriented programs consist of classes some of which are technical, some of which are domain-specific (or application-specific) in nature. Technical classes such as `String`, `Vector`, and `Exception` are used in literally all programs: they are general purpose ("helper") classes whose use is not restricted to any particular problem. Other classes such as `Account`, `Invoice`, or `PatientRecord` represent the concrete problem being solved by the program: they are domain specific in the sense that they represent items of that part of reality in which the problem is situated (the "problem domain").

Proponents of aspect orientation suggest that aspects are concepts that are discovered in a domain analogous to the way classes are discovered in it. In fact, they have successfully seeded the impression that just as there are general purpose technical classes and domain-specific classes, we will find that there are general purpose technical aspects as well as domain-specific aspects. This would make aspects a truly general notion, one that bears the potential of revolutionizing software engineering in an analogous way objects did, since it can be applied to all stages of software development including the soft ones such as requirements elicitation and modelling.

Frankly, I doubt whether this is really the case. Instead, I would conjecture that every aspect discovered with a given problem is technical in nature and as such part of the solution of the problem rather than part of the problem itself; that in fact all aspects are aspects of programming rather than aspects of the domain for which is being programmed. Since programming is a technical matter, it follows that aspects are technical.

One might argue that being technical does not preclude aspects from being domain-specific; this is correct, but in this case domain specific means "specific to a particular technology employed to solve a particular class of problems", as for instance specific to compiler construction, to middleware, or to web services (which are not to be confused with the domain they are being applied in, for instance a Java compiler, a data warehouse system, or e-banking). In order to distinguish the different meanings of *domain specific*, I will use the term *domain level* in the sequel and mean "found in a domain as part of the original problem, not of its (technical) solution".

My opposition to aspect-orientation as a generally useful paradigm of software engineering is condensed in the following two hypotheses:

Hypothesis 1: The number of aspects is not only finite, but also fairly small.

Hypothesis 2: Aspects are inherently technical.

If these hypotheses are true, aspects are not as universally useful as for instance classes; in particular, they will not be found in conceptual models.

Obviously, both hypotheses need further explication. As for hypothesis 1, what does "fairly small" mean? Large object-oriented software systems today have tens of thousands of classes. Given that aspects crosscut a system, I would conjecture that the number of aspects in any one system is in the tens. But even if not restricted to a single problem and its solution (an application), I would assume that the total number of (conceptually different) aspects is in the hundreds (growing with, yet bounded by the technological possibilities of programming, i.e., the different means we have to put a certain functionality into practice), whereas that of classes is at least as

in: *EIWAS'04 – European Interactive Workshop on Aspects in Software* (Berlin, September, 23-24, 2004).

large as the number of concepts we have (considering that the English language has some 400,000 words, we can reasonably expect it to be in the hundreds of thousands).

As for hypothesis 2, what does "technical" mean? If the domain is technical, are not all concepts identified in it necessarily technical? Here, we take "technical" to mean "added to the problem as part of its solution", i.e., something invented (a design or implementation artefact) to solve a certain problem. For instance, `Serializable` is a technical concept, since serialization relates to the solution (the way we store or transport data), whereas `Addressable` (meaning "having an address") is on the domain level. Note that both concepts are *roles* rather than *aspects*.

I have attempted elsewhere to prove that there can be no domain-level aspects in the aspect-oriented sense. If we accept that "not domain level" means "technical", then the second hypothesis follows immediately from this proof, with the first following indirectly, since the number of technical aspects is bounded by technology. My proof attempt of the non-existence of domain-level aspects is somewhat lengthy; it is based on the observation that aspects are either

a) roles (of an object) and hence should be modelled/implemented as such, or
b) second-order constructs and hence one level above (and not in) the domain being modelled/programmed.

I do not intend to repeat this proof in greater detail here, but rather want to call for the proponents of aspect orientation to attempt its falsification, simply by giving convincing counter-examples. As a matter of fact, hypotheses 1 and 2 will be proven wrong if we can find an element of a model or program

- that is an aspect in the aspect-oriented sense,
- that is not an artefact of the technical solution, but can be seen as representative of an element in the underlying problem domain, and
- whose choice has a certain arbitrariness about it so that the element can serve as prototype for other aspects in the same or other domains.

The search is not as simple as one might expect, since there are many aspects out there that are not really aspects in the aspect-oriented sense, or that are so special that they do not generalize. What follows is a list of seemingly obvious counter-examples that I consider unsuitable to prove the hypotheses wrong:

Roles A role is a named type comprising a set of properties whose definition depends on collaborations with other roles. The implementation of a role is typically polymorphic, since objects of different classes fill the roles differently. An aspect on the other hand is not a type, its definition is independent of other aspects, and its implementation is the same for all objects/classes having the aspect. Hence, roles are not aspects.

Domain classes with aspect names Sometimes a domain comes with elements that are best described by names of popular aspects, for instance `Transaction` and `Log` in a banking application. However, these elements are typically classes rather than aspects; in particular, they do not crosscut the domain.

Aspects as the subject matter Quite obviously, if aspects themselves are (part of) the subject matter, then we have aspects in the domain. This is the case when developing tools for AOSD. However, this example is rather singular; I doubt there are more of this kind.

To initiate the search, I will conduct a wiki-based seminar next semester in which we will try to collect all aspects discussed in the literature to date, and classify them into the categories *technical/general*, *technical/domain-specific*, and *domain-level* (if any). The seminar is open to students from other universities and, as is the nature of a wiki, to contributions from anyone who is interested. Hopefully, by next spring we have reached more clarity concerning the potential aspects have in revolutionizing the whole of software engineering, not only programming.