Operation-Level Composition: A Case in (Join) Point

Harold Ossher
Peri Tarr

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
{ossher, tarr}@watson.ibm.com

1. Introduction

Our work on subject-oriented programming [1, 2] has focused on two central issues that we believe to be at the core of research in the domain of "aspect-oriented" software development:

- Facilitating the identification and description of *cross-cutting* concerns in a software system—i.e., those aspects that affect more than one unit of functionality in the system, given some definition of "units of functionality" (objects, modules, functions, etc.).
- Facilitating the identification and integration of *join points* in the system. Join points are the locations in systems that are affected by one or more cross-cutting concerns. The process of integrating join points involves describing *how* a cross-cutting concern affects code at one or more join points. The integration process is referred to as *composition* or *weaving*.

The set of possible join points includes all locations in all system components, which we call *statement-level* join points to indicate that they can occur in any part of code. Subject-oriented programming, however, is predicated on the belief that a significant majority of join points of concern in software development are those represented by *operations*, and that the majority of cross-cutting issues that are of concern to developers are those involving capabilities that affect multiple operations. Additionally, we believe that a focus on operation-level joining is especially appropriate in an object-oriented context, because it adds the power of composition naturally within the object-oriented paradigm.

Operation join points do not necessarily imply specification (or coding) of all aspects as functions. We believe that many of the kinds of capabilities for which aspect-oriented programming might be used, which might at first appear non-functional, do, in fact, involve functional aspects and operation join points. However, there are undoubtedly cases in which non-functional specification of aspects, in different notations, is appropriate. Even in such cases, operation join points are often appropriate.

The remainder of this position paper explores the broad utility of functional aspects and operation join points, and discusses several issues that affect the feasibility of supporting general statement-level join points.

2. The Prevalence of Operation Join Points

Many kinds of cross-cutting concerns affect the definition of collections of operations that span multiple units of functionality. For example, many objects might support a `print` capability. The particular way in which this capability works in a given system might depend on one or more system requirements, such as "all print operations will send mail to the system administrator if they cannot complete successfully." Cross-cutting concerns that affect the behavior of groups of operations typically require the use of operation join points—e.g., to add a check for success after the invocation of each print operation, and to send mail as appropriate.

In our experience, a wide range of cross-cutting concerns that affect a system are, in fact, correctly described using operation join points, even when, at first glance, it is not obvious that this would be the case. This phenomenon arises because cross-cutting concerns very often are specified in terms of how they affect existing object behaviors (when they define new behaviors, they can be said to involve operation join points trivially), which makes them amenable to implementation via operation join points. Some examples of these kinds of cross-cutting concerns, and their description in terms of operation join points, are presented in the following paragraphs.

Persistence: Persistence tends to be a pervasive property of data in systems; thus, it would be appropriate to develop a "persistence aspect" that implements the persistence capability independently of any particular objects and compose this capability into those objects that are to become persistent. In deciding how to compose the persistence aspect with the objects to which it will apply, we note that retrieval of persistent objects from a database occurs upon object access, and update of persistent objects occurs upon object creation or modification. Thus, the necessary join points are operations: the "update" part of the persistence aspect affects the constructor and `set` methods of persistent objects, while the "retrieve" part affects `get` methods.

Error detection and handling, and fault tolerance: Some forms of error detection are intrinsic to the definition of a type of object; for example, it is always wrong to attempt to pop an item from an empty stack. Such "well-formedness" definitions are usually built into a type. Other kinds of errors, however, are context-specific—their presence depends on the requirements of the particular application in which the types are used. For example, a set of generic, reusable components (e.g., lists, stacks, sets) used in a compiler have considerably looser error handling and fault tolerance requirements than the same components used in a safety-critical system. In such cases, it is desirable to describe the error detection and handling behaviors as one or more separate aspects. Some of the most common kinds of non-intrinsic error handling mechanisms we have seen are those represented as pre- and post-conditions on modify methods. The join points in such cases are operations: pre- and post-condition checks, error-catching methods, and error-handling methods can all be joined to the methods that can cause or encounter the error conditions. There are other cases where one really needs to add additional error checks within existing code; in these cases operation join points are not sufficient (unless one is willing to duplicate code by replacing a function with a copy to which the additional checks have been added).

Logging, tracing, and metrics-gathering: Where and when logging, tracing, or metrics-gathering activities occur is frequently dependent on application-wide decisions that are determined, for example, by development phase or local policies. Ideally, code to perform these activities would be modeled as an aspect and composed selectively into the relevant parts of an application. All of these activities are usually associated with operations (e.g., to log entry into, and exit from, an operation) and could be composed using operation join points.

Caching behavior: It is often desirable to consider the issue of caching intermediate results in a complex computation as an aspect separate from the algorithm itself. In particular, an algorithm written without caching might need to be modified to include caching upon observation of inadequate performance. In the important case where the computation traverses a network of objects, processing each node to compute some value(s), operation join points are natural. The "process" operation in the algorithmic aspect just performs the computation. The caching aspect of this same operation maintains a cache of the computed value(s), probably in the node itself. It either returns the cached value or invokes the algorithmic aspect, depending on the currency of the cache. Clearly, the composition in this case must give the caching aspect control.

Other common cross-cutting issues that affect many applications include serializability, atomicity, replication, security and visualization. We believe that support for these and many other features turn out to be well represented using operation join points.

Clearly, functional aspects and operation join points are required and useful for describing and integrating a wide range of important cross-cutting concerns in software systems. For this reason, the subject-oriented programming paradigm supports "aspect-oriented programming" based on functional aspects and operation join points. Part of our ongoing research includes the exploration and validation of the variety of functional and non-functional aspects to which operation-level joining applies.

3. On General, Statement-Level Join Points

A key element of subject-oriented programming is flexible, domain-independent, generic points at which composition is to occur, and specify the details of the composition desired. It is important that rules continue to work even as the inputs evolve, within reason. We excluded statement-level join points for several reasons:

- We have not yet found convincing evidence that the additional power resulting from such join points is of general use, particularly in light of the concomitant increase in complexity. This is particularly true in light of the broad spectrum of circumstances under which operation join points appear to be applicable.
- The tractability of the problem of defining general-purpose statement-level weavers is questionable. Even stable references to join points in rules present serious problems in the light of evolving inputs.
- We are extremely concerned about the degree of unpredictability that results from statement-level weaving. Changing a statement in a piece of code changes both the data- and control-flow properties of that code; any guarantees that might have been made about the code are negated. In fact, much work in the area of software analysis and testing has attempted to identify the impact of changes to code, in an attempt to identify new errors and to help select test cases whose results have been invalidated by the changes. Unfortunately, data- and control-flow analyses are inherently exponential, further suggesting the difficulty involved in understanding the effects of statement-level composition.
- One of the significant contributions of object-oriented software development was to make changes *additive* rather than *invasive* (see, for example, [3]). This is a particularly important property because of the well-documented, extremely adverse effects on software systems of invasive changes. The notion of general statement-level joins represents invasive software change, violating the additive-changes principle.

In light of these concerns, we do not believe that conclusive evidence yet exists to suggest that the additional power provided by arbitrary statement-level joining is of sufficiently broad and practical use to justify its potential disadvantages. Indeed, the work on aspect-oriented programming we are aware of has concentrated on weaving in specific domains, and in each case the kinds of join points, though not always operations, are carefully circumscribed. Further research is required to characterize systematically the circumstances under which statement-level joining may be practical and justified, despite its drawbacks.

4. Conclusion

We believe that the ability to describe capabilities associated with concerns that cut across multiple parts of a system and to specify how those capabilities affect the system, without having to physically intersperse the code that realizes such capabilities, is an extremely important part of support for programming-in-the-large. Used correctly, this capability, which defines aspect-oriented programming, can reduce the complexity and improve the maintainability of a system considerably.

Identifying the points at which a cross-cutting concern affects a system is an important part of aspect-oriented programming. While the set of potential "join points" is large and could, potentially, include every statement and expression within a system, our research suggests that focusing on operation-level joining can successfully address many common composition needs, for both functional and non-functional aspects. Further, we believe that general statement-level joining raises numerous technical and methodological concerns that may render it intractable, infeasible, or undesirable in the general case. Further work is needed to test and extend this conclusion.

Subject-oriented programming is an approach to aspect-oriented programming that is based on operation-level joining. Subject-oriented programming thus avoids the problems inherent in general statement-level joining, while providing the ability to describe many kinds of cross-cutting concerns separately and to compose cross-cutting capabilities into object-oriented programs. It also preserves and extends many of the desirable features of the object-oriented paradigm. The choice of join points has made it possible to develop a general-purpose, domain-independent compositor, which is central to our tool support [4].

Acknowledgements

Stan Sutton provided many very helpful comments on earlier drafts of this document.

5. Bibliography

[1] William Harrison and Harold Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*, pages 411–428, Washington, D.C., September 1993. ACM.

[2] Harold Ossher, William Harrison, Frank Budinsky, and Ian Simmonds. Subject-oriented programming: Supporting decentralized development of objects. In *Proceedings of the 7th IBM Conference on Object-Oriented Technology*, Santa Clara, CA, July 1994, IBM. Available as Research Report RC 20004, IBM T.J. Watson Research Center, Yorktown Heights, NY, March 1995.

[3] John Vlissides. Subject-Oriented Design. In *C++ Report*, February 1998.

[4] http://www.research.ibm.com/sop