

Towards Aspect Architectures and Remodularization

Mika Katara*

Department of Computer Science
The Technion, Haifa 32000, Israel
katara@cs.technion.ac.il

Shmuel Katz

Department of Computer Science
The Technion, Haifa 32000, Israel
katz@cs.technion.ac.il

1 Introduction

Separation of concerns is one of the key elements in the design of complex systems where software plays a major role. To cope with complexity, a large and complicated system is divided into smaller units, i.e. modularized, so that different concerns are localized. Examples of such units, or modules, include classes, components and processes.

However, there is no way to divide a system so that total separation of concerns would be satisfied. That is, it is always possible to find a concern that cuts across the structure imposed by the division. In conventional design notations and programming languages, for instance, concerns emerging from the requirements, such as features visible to the users, are usually scattered around the units and tangled with other features inside the units. It has been speculated that this kind of misalignment between requirements, design, and code, resulting in scattering and tangling, causes problems concerning traceability, comprehensibility, evolvability, low reuse, high impacts of changes and reduced concurrency in development [1].

In practice, addition of new features, or changing features, requires invasive treatment affecting many modules in the existing design. In effect, the maintenance requires a vast amount of resources, notably time and money, during the software life cycle.

Conventional notations and languages allow good separation of concerns only along a *dominant dimension of decomposition* [6]. That is, for instance, in object-oriented methods, the units of the decomposition are the objects and in functional languages they are the functions. What is missing, is the capability to decompose simultaneously along some other dimensions.

*On leave from Institute of Software Systems, Tampere University of Technology, Finland

Furthermore, as pointed out by Ossher and Tarr [3], the set of relevant concerns is context-sensitive and changes in the course of the software life cycle. They suggest that it should be possible to *remodularize software on-demand*.

Recently, a number of new design approaches and programming languages have been introduced to alleviate the above problems. These support advanced separation of concerns, or *aspect-orientation*, enabling designers to address cross-cutting concerns in a modular fashion and to make additive, instead of invasive, changes to existing designs.

While offering a considerable advantage over conventional notations and languages, current aspect-oriented approaches leave the relationships between different aspects implicit. Consider having two aspects, describing different cross-cutting concerns of the same system. Several questions immediately arise, primarily whether the aspects are orthogonal. That is, are they related and if they are, how are they related and are there some common *sub-aspects* interesting in their own right? The effects on one aspect caused by the changes in the other raise more questions.

Questions like these call for making the dependencies between aspects explicit. There should be an *aspect architecture* defining the dependencies and enabling to focus only on some subset of the aspects and their mutual relationships. The architecture would also define the space for remodularizations.

To obtain such an architecture, we propose lifting *augmentations to existing designs* to first class entities. That is, to encapsulate design augmentations that introduce new details to existing design incrementally. Naturally, the augmentations can cut across any existing module division. While there might be concerns that cut across any module division, we think that useful concerns rarely cut across *design augmentations*.

In this paper, we describe our view of aspect architecture and outline how to extract it from an existing system.

2 Aspect-orientation

We adapt a view of [5] and emphasize the distinction between concerns and software artifacts. Concerns are conceptual matters of interest. We use the term aspect to mean any software artifact that “addresses, embodies, or exhibits manifestations” of a cross-cutting concern, in a modular way.

In Figure 1 the scattering and tangling are illustrated in the case of two concerns A and B whose corresponding software artifacts overlap. The design or code treating concern A cross-cuts three modules and that of B two modules. Moreover, in one of the modules, A and B overlap, meaning that, for example, some pieces of code, in part, address both concerns.

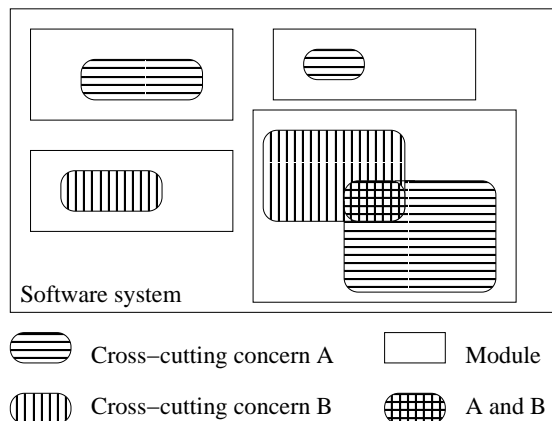


Figure 1: Overlapping, cross-cutting concerns.

In a general case, it is impossible to modularize a system into disjoint modules each addressing a separate concern. If it would be possible, concerns A and B would be localized. Then, except for the most trivial cases, there exists some concern C that deals with artifacts both in A and in B, and whose treatment would then be scattered to modules addressing A and B, which leads to a contradiction. As an example, consider mutual exclusion which is clearly a cross-cutting concern of processes trying to enter a critical section.

To alleviate the problems caused by such cross-cutting concerns a number of new pro-

gramming languages and design approaches have been recently introduced. These support advanced separation of concerns, or aspect-orientation, enabling designers to address cross-cutting concerns in a modular fashion and to make additive, instead of invasive, changes to existing designs. On the one hand, to avoid scattering, a module contains *all* those parts of the system that address the concern in question. On the other hand, to avoid tangling, the module contains *only* those parts. For example, AspectJ [7] enables good separation for many concerns that would otherwise be scattered around the classes given in the Java language.

In [6], Tarr et. al. laid the foundations of so called *multi-dimensional separation of concerns*. They introduced the *Hyperspaces* model which enables addressing overlapping concerns in a modular fashion independently of the notation or language used. The basic idea is to capture each aspect in a separate module called a hyperslice. The artifact described by a slice need not be well-defined, or complete, in the notation used. Slices need not respect any module division dominant in the notation.

The slices can be partly overlapping by describing the same units from different perspectives or by using different units to describe the same concepts. The slices can be grouped to hypermodules which define how the incorporated slices should be composed to form well-defined, complete artifacts. Hypermodules produce valid slices, so they can be nested.

3 Aspect architecture

In our opinion, there are certain weaknesses in the Hyperspaces model. On the one hand, because relationships between different aspects are given only implicitly by the overlapping parts of the slices, composition is very hard to do automatically. In fact, complex reconciliation may be needed when the slices are composed. On the other hand, there might be some common subaspects interesting in their own right. The subaspects would be good candidates for generic aspects when building an aspect library. However, this would mean that it should be possible to define them explicitly.

To alleviate the above problems, we call for making the dependencies between aspects explicit. There should be an aspect architecture defining the dependencies and enabling to focus

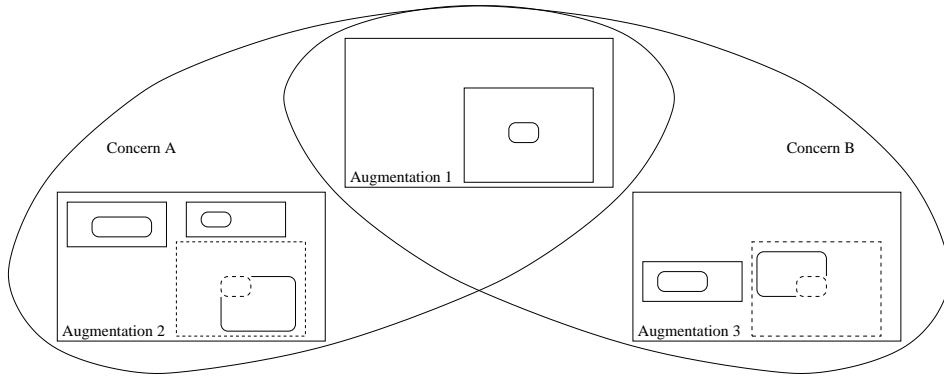


Figure 2: Augmentation steps 1, 2 and 3 addressing concerns A and B.

only on some subset of the aspects and their mutual relationships. Naturally, any duplication of information should be avoided. That is, the overlapping parts of non-orthogonal aspects, and interesting sub-aspects in particular, should be described explicitly and only once.

We suggest encapsulating design augmentations that introduce new details to existing design. These augmentations are applied in an incremental and additive fashion. Naturally, the augmentations can cut across any existing module division.

Each augmentation consists of *uses* and *defines* parts. On the one hand an augmentation *uses* those parts of the original design to which it adds new details. On the other hand, the augmentation *defines* the new details to be added to the original parts.

Each concern is addressed by a collection of augmentations. However, as augmentations can be composed, we can compose the collection into one augmentation. In [4] two operations were introduced for composing *superimposition steps*, sequential composition and parallel composition. These are the two basic operations that are needed to build complex augmentations from existing ones. In sequential composition, one augmentation is applied before another so that the *uses* part of the latter can be satisfied, perhaps only partly, by the *uses* and *defines* parts of the former. In parallel composition, the steps are considered to be applied independently, meaning that the *uses* and *defines* parts of the result are the union of the *uses* and *defines* parts of the components, respectively.

In Figure 2 an abstract aspect architecture is depicted. It consists of three augmentations.

The artifact addressing concern A is built using sequential composition from augmentations 1 and 2. Similarly, the one addressing concern B is sequentially composed of 1 and 3. The *uses* parts of the augmentations are given in dashed lines and the *defines* parts in solid lines.

In the architecture, the different artifacts corresponding to the concerns and their relations are explicit. On the one hand, if we would have to replace concern B, for instance a feature, with some new concern, e.g. another feature, the architecture would guide us in doing so. In particular, we should leave augmentation 1, possibly a common sub-aspect, untouched because it also addresses concern A. On the other hand, we could build an artifact addressing a composite concern AB from the ones addressing A and B. This could be done by first applying parallel composition to augmentations 2 and 3. The result would be sequentially composed with augmentation 1.

4 Remodularization

As pointed out by Ossher and Tarr [3], the set of relevant concerns is context-sensitive and changes in the course of the software life cycle. They suggest that it should be possible to *remodularize software on-demand*.

On-demand remodularization means that a designer, or another stake holder, should be able to view an artifact that reflects whatever concern of the system interests him/her, whenever needed. This calls for an ability to reorganize the aspect architecture, including creating new augmentations and aspects.

In principle, it should be possible to decom-

pose an existing system into augmentations each of which would define exactly one detail. These augmentations would then correspond to the Lego bricks that one could compose using sequential and parallel composition to build a complete system. The *uses* and *defines* parts of the augmentations define the “interfaces” between the bricks that must be respected.

For example, consider a system where (unrelated) objects have methods that update local values. The system also was designed with a concern of fault-tolerance by developing an aspect that duplicates instances of the objects on different processors (and guarantees their consistency), and another concern of security of key variables with an aspect for encrypting and decrypting them at sensitive points (like messages that activate methods).

The developers also guaranteed that no variable can lead to overflow, by checking in advance within each method that the needed data manipulations stay within a fixed range of values. Although overflow was *not* addressed as a separate concern at the design phase, when interest turns to it, we would like to isolate the treatment by creating an aspect, both so it can be analyzed, and to allow potential changes in its implementation. Thus we would like to modularize, extracting the treatment of overflow from the underlying data manipulation, and showing its interactions with other aspects.

After the modularization, it becomes clear that the treatment of overflow is independent of the fault-tolerance, but does interact with the security, in that the encryption can assume values within those provided by the code implementing the overflow concern, and needs to provide encrypted values within those limits. So, overflow and security are closely intertwined. They have at least one common augmentation, and possibly even a common sub-aspect.

5 Discussion

In [2] the close relationship between aspects and superimposition techniques was first described. Similarly to Hyperspaces, the aspect architecture approach needs be instantiated for some artifact development formalism before becoming applicable. The DisCo method [8] can be seen as a restricted instantiation of our approach in which the focus is on capturing joint behavior of objects using multi-object actions.

We have described some weaknesses of current aspect-oriented approaches to software development. An aspect architecture is proposed making explicit the relationships between different aspects by lifting augmentations to existing designs to first class entities and emphasizing overlapping parts. The overlapping can help to identify useful sub-aspects. Moreover, we have outlined how to modularize a system into augmentations. These augmentations can be composed using sequential and parallel composition to match the different concerns of the system.

References

- [1] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: towards improved alignment of requirements, design, and code. *ACM SIGPLAN Notices*, 34(10):325–339, Oct. 1999.
- [2] S. Katz and J. Gil. Aspects and superimpositions. Position paper in Aspect Oriented Programming workshop in ECOOP’99, Lisbon, Portugal, June 1999.
- [3] H. Ossher and P. Tarr. On the need for on-demand modularization. Position paper in ECOOP 2000 workshop on Aspects and Dimensions of Concerns, Sophia Antipolis and Cannes, France, June 2000.
- [4] M. Sihman and S. Katz. A calculus of superimpositions for distributed systems. In *Proc. 1st International Conference on Aspect-Oriented Software Development*, Enschede, The Netherlands, Apr. 2002.
- [5] S. M. Sutton, Jr. and P. Tarr. Aspect-oriented design needs concern modeling. Position paper in AOSD 2002 workshop on Aspect-Oriented Design, Enschede, The Netherlands, Apr. 2002.
- [6] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proc. 21st International Conference on Software Engineering*, pages 107–119, Los Angeles, CA, USA, May 1999. ACM Press.
- [7] AspectJ WWW site. At <http://aspectj.org> on the World Wide Web.
- [8] DisCo WWW site. At <http://disco.cs.tut.fi> on the World Wide Web.