The Paradoxical Success of Aspect-Oriented Programming

Friedrich Steinmann

Johan Östlund

Introduction

- AOP sets out to increase modularity and structure of code by enabling the modularization of cross-cutting concerns
- AOP is a promising new technology; in many ways like OO once was
- AOP is being adopted by increasing numbers, both in industry and academia
- AOP works against independent development and understandability of code, two of the primary purposes of modularization
- Thus, AOP's success as a means of achieving modularization is paradoxical

AOP
- a Moving Target

- Each AOPL comes with it's own (unambiguous) formal description of what AOP is
- No single definition that is
  - common to all AOPLs and
  - sufficiently distinguishes it from other, long established programming concepts
- There is though a common understanding what AOP is good for, namely modularizing cross-cutting concerns

The Aspect Formula

- The (probably) best known definition AOP is

  aspect-orientation = quantification + obliviousness

Obliviousness

- Obliviousness means that a program has no knowledge of which aspects modify it or when
- Obliviousness as a defining characteristic of AOP has been questioned by the AOP community
- Some say hat obliviousness is what distinguishes AOP from event-driven systems
- Obliviousness comes more as a side-effect of quantification

Quantification

- Quantification means that an aspect can affect arbitrarily many different points in a program
- Quantification is widely accepted as a defining characteristic of AOP

The Aspect Formula, cont'd

- The sentence "*In programs P, whenever condition C arises perform action A*" captures how an aspect (C, A) affects a given program P,
- but says nothing about P's knowledge of the aspect (C, A), and thus nothing about obliviousness
- As the context provided to an action A is provided by the aspect (C, A) and not by the program P the program is oblivious to which program elements an aspect relies on, as opposed to a function call where arguments are explicitly passed to the function

Interpretations of the Aspect Formula

- Translated in terms of AspectJ the parts of the formula read
  - P is the execution of a program, which includes the execution of advice
  - C is a set of pointcuts specifying the target elements of the aspect in the program and the context in which they occur (mostly variables, but also stack content)
  - A is a piece of advice that depends on the context captured by C; and
  - the quantification is implicit in the weaver

Playing with the Formula

- Using different formulations of the condition C we can investigate AOP, or really the above definition

Awareness Extreme

- Consider a condition C such as

  In programs P, whenever an aspect is referenced, perform its associated action A

- This expresses nothing more than the semantics of a standard procedure call

- This shows that quantification can be completely independent of obliviousness, as all places where condition C can arise are marked in the program text

- The programmer of P needs to now about which aspects are there, how they are named and how they work

- This is not AOP, but it shows that the "definition" of AOP is quite stretchable

Obliviousness Extreme

- Consider a condition C such as

  In programs P, whenever *Random* indicates it, perform action A

- This means that all points in a program are implicitly marked, but execution of A remains uncertain

- The programmer of P may be aware of AOP, but has no knowledge of the existence or behavior of any aspect

Taming Obliviousness

- The two previous examples are at the far extremes of the interpretation of the AOP formula
- There are, of course, less extreme interpretations of the formula

Annotations

- Consider a condition such as

  In programs P, whenever condition C arises
  where element B is referenced,
  perform action A

- B may be an abstract annotation

- Enables the programmer to deny aspects access where it is not wanted by not referencing B, but this means that the programmer must know of the aspects

- This is more or less equivalent to inserting a dynamically bound procedure call

- For massively crosscutting-concerns the annotations may very well turn out as annoying as the scattering of code that the aspect was to modularize

Annotations, cont'd

- Using annotations reduces obliviousness to a level where the programmer of P knows that aspects may interact with the points marked B in P, but not which aspects or when
- However, annotations can act as interfaces between the program and the aspects, translating some of the best practices of OOP to AOP

Annotations, cont'd

- Consider the following condition C

  In programs P, whenever condition C arises, add annotation B

- Obviously the aspect could add the advice directly, but that would mean going back to the original formula

Taming Quantification

- If and where aspects advice a program may very well seem random to a programmer
- Many propose tool support as a remedy to this, but tools can only mark the possible pointcut-"*shadows*" and not where and when advice are actually executed
- Keeping track of exactly where aspects advice an evolving program is not a trivial task
- One way of reducing this randomness is to use an explicit list of elements to be adviced

  In programs P, whenever execution reaches one of the points in $\{p_1, ..., p_n\}$, perform action A

- This is, of course, tedious and error prone for any interesting program

Taming Quantification, cont'd

- Generally the quantification property of AOP suffers from the the problem that conditions are extremely sensitive to changes in the program (known as *the fragile pointcut problem*)
- Some researchers expect that this can be addressed by using better languages for expressing conditions, i.e. semantic pointcut languages.
- However, for an aspect to be useful in any interesting way it needs to reference the program context, at which point a semantic pointcut language cannot help,
- unless automatic program understanding is invented, which would revolutionize programming as a whole and render AOP, as well as every other technique known today, obsolete

Modularity

- A module has a well defined interface which declares exactly what travels in and out of it
- This enables developers to work on different parts of a system (more or less) independently

Modules and Interfaces

- Interfaces form the border between modules
- Interfaces represent the coupling between modules
  - If the interface between two modules is empty, there is no coupling between them
- Interfaces should be made as explicit as possible to enable independent development

AOP and Modularity

- AOP breaks the modularity of the program by modularizing cross-cutting concerns
- One could argue that this is for a good cause - and thus worth it
- What happens when cross-cutting concerns crosscut each other? And as soon as an aspect is woven it is part of the program and thus is a candidate for weaving of other aspects

AOP and Modularity, cont'd

- Of course one could introduce annotations in the program to mark the places that should be adviced by aspects, but this makes AOP no different from a subroutine call
- It also reintroduces the very scattering of a concern that AOP was to avoid

Conclusion

- AOP sets out to modularize cross-cutting concerns, but it’s very nature breaks modularity
- It appears as this paradox cannot be resolved by tweaking the mechanics of AOP, because you end up with something which is very close to what we already have
- As a way of organizing code AOP does a good job by localizing a scattered concerns, but at the same time it breaks modularity of the program
- Thus, AOP’s success as a means of achieving modularization is paradoxical