Implementation Techniques for AOP

Focus on Weaving

AOP workshop

05/09/01

Thomas Ledoux

Pierre-Charles David

Terminology

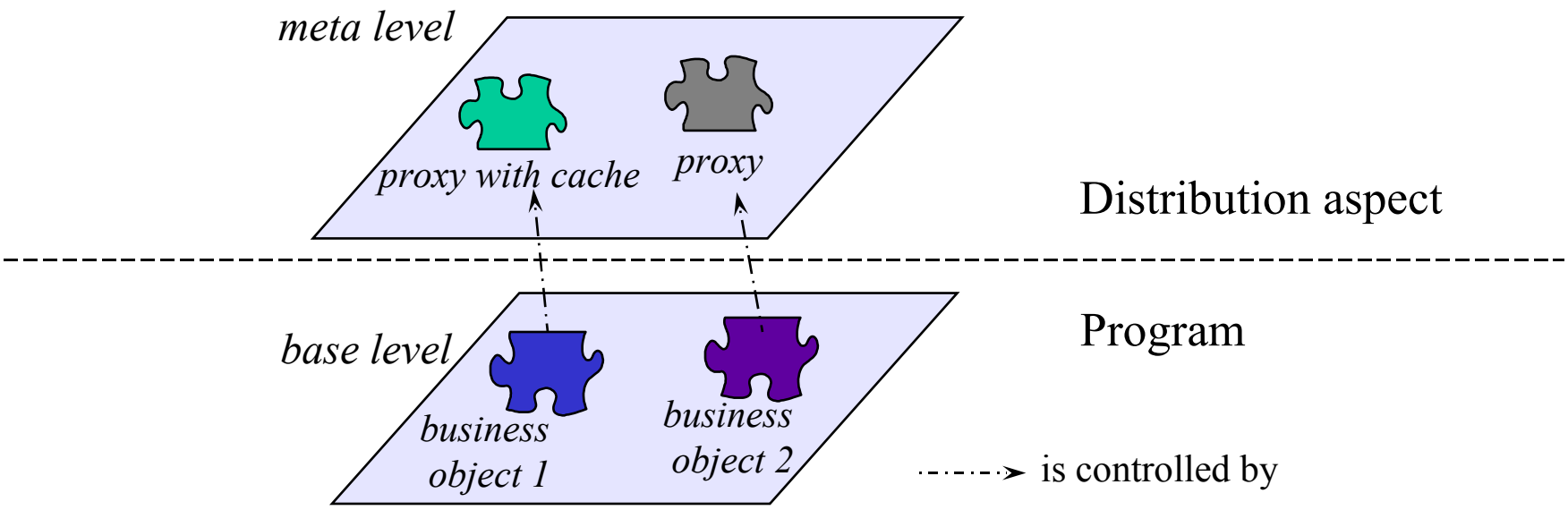
- Join points
  - principled points in the execution of the program
- Weaving
  - coordination process between aspect code and non-aspect (program) code at the appropriate join points

Weaving Issues

- How to weave? When to weave? are recurrent questions for AOP system designers
- How to weave?
  - first classification: Juxtapose, Merge, Fuse [Lamping97]
  - AspectJ: advice and pointcut constructs
- When to weave?
  - natural classification: compile-time, load-time, run-time
  - AspectJ: compile-time approach

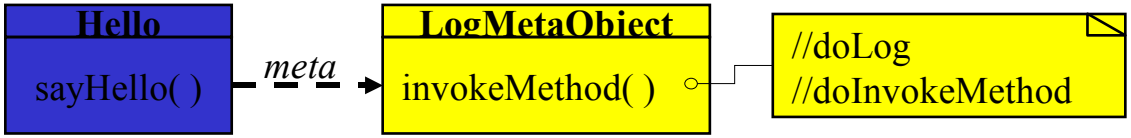
AOP and Reflection

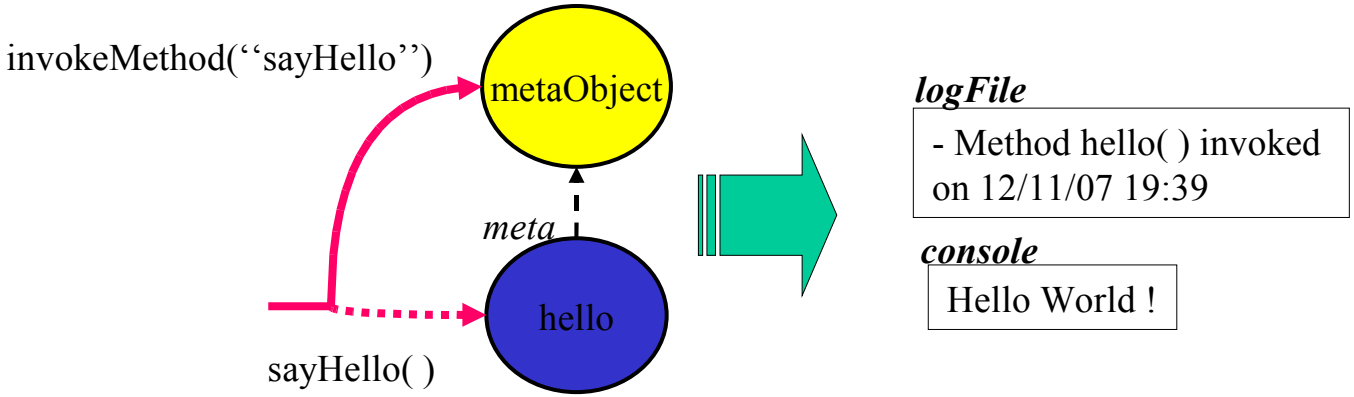
- Reflection allows separation of concerns
- Meta-objects can represent aspects



Weaving in Reflection (1/2)

- How to weave?
  - jump – via the meta-link – to the meta-level and execute meta-object code





Weaving in Reflection (2/2)

- When to weave?
  - compile-time, load-time, VM-based run-time, proxy-based run-time reflection [Caromel01]
    - classification of MOPs based on *when* meta-objects are actually in use
    - proxy-based MOP introduce *hooks* in the program in order to shift to the meta-level (⇨ program transformation)

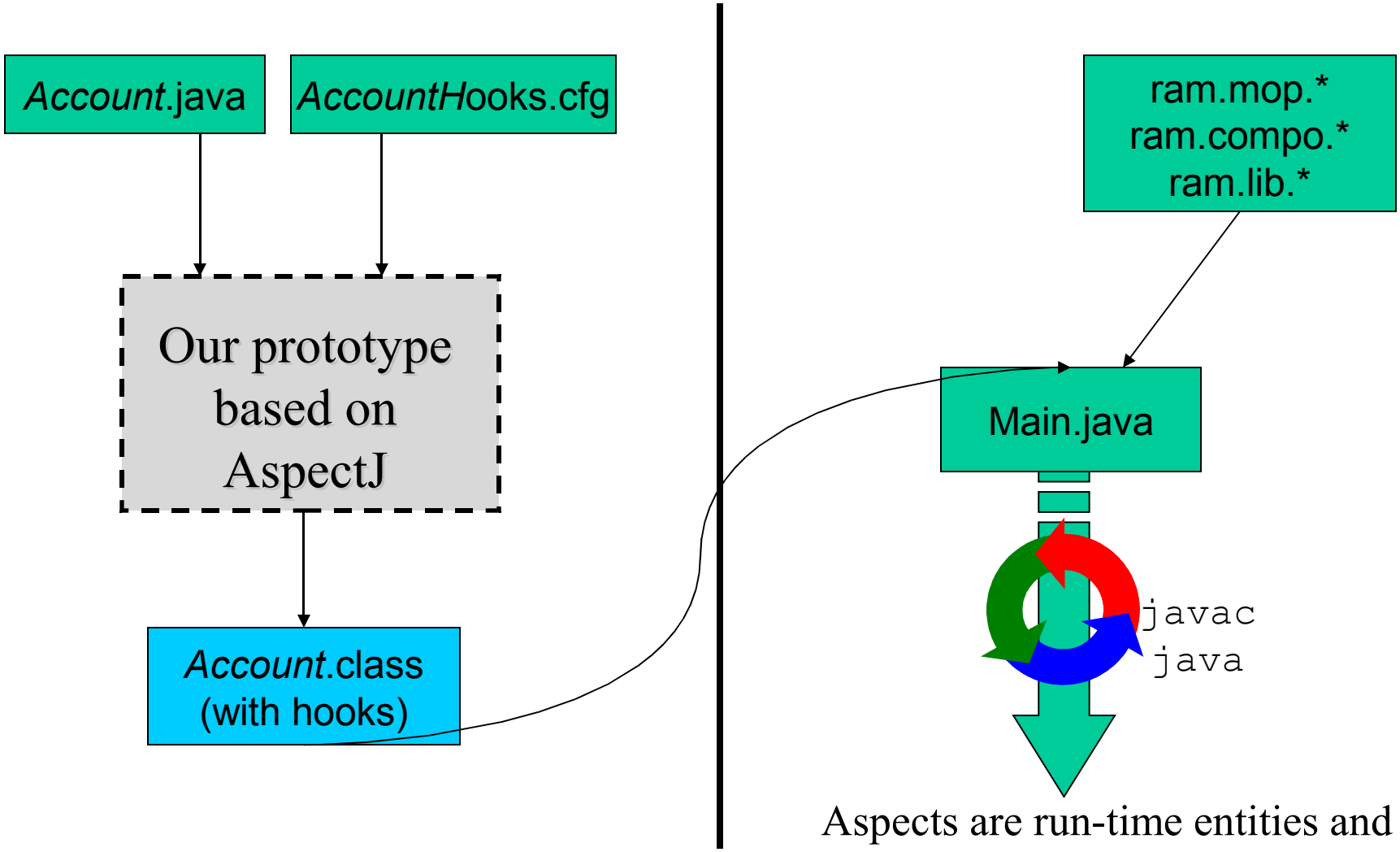
Current Work

- Explore the relationships between proxy-based run-time reflection in Java and AspectJ!
- Why?
  - We use run-time reflection in our current research projects
  - AspectJ proposes a rich set of join points where to introduce hooks
  - AspectJ is designed by Gregor ;-)

Our Solution: two step-weaving with Reflection using AspectJ

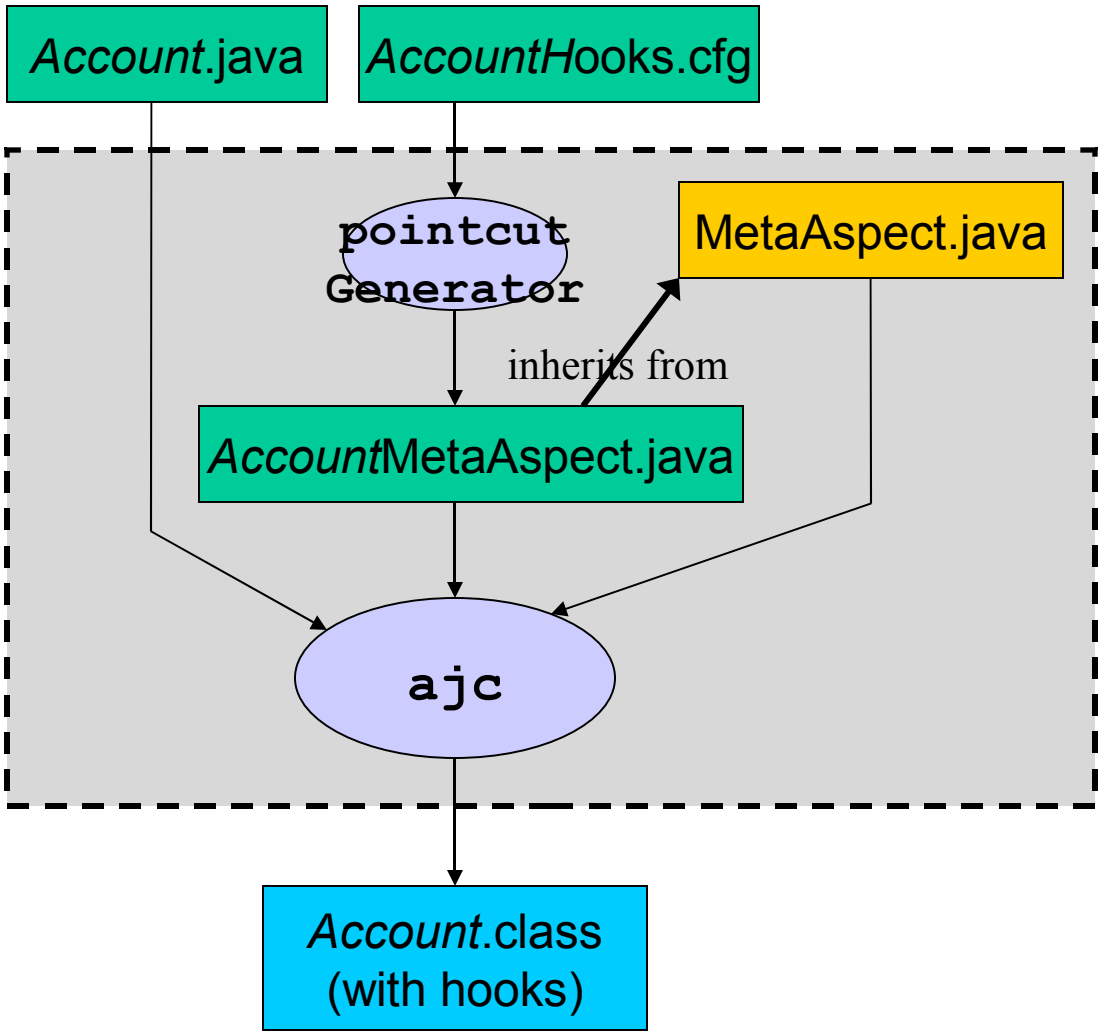
- Idea: a loose weaving with two steps
  - 1st step : in AspectJ
    - *compile-time*
    - weave at the appropriate join points in a class code a *generic aspect* introducing an indirection to our MOP
  - 2nd step : in RAM (our Java MOP)
    - *run-time*
    - attach dynamically a meta-object – implementing a particular aspect – to an instance of this class
    - adapt this attachment when needed: re-weaving!

Illustration



Aspects are run-time entities and can be changed dynamically

Our Prototype



Implementation Details

- Hook declaration
  - declare the appropriate hooks in the program code in a configuration file
- In AspectJ
  - MetaAspect
    - defines abstract pointcuts
    - uses thisJointPoint to have a reflective access to the current join point
    - implements generic advices calling our MOP
  - sub-aspect of MetaAspect (*automatically generated*)
    - concretizes the abstract pointcuts in order to introduce hooks at the pointcuts in the program code

A Support for Dynamic Adaptation of Middleware

- Dynamically adapt the execution policies of middleware platforms
  - mobile multimedia applications
  - examples: replication policy, persistence policy, etc.
- Middleware for *context-aware applications*
  - applications have to be aware of, and adapt to, variations in their execution context
  - examples: detect and react to high variability of network bandwidth, temporary loss of network connectivity, etc.

4 Elements of AOP
[Kiczales this morning]

- Evaluation of our solution
  - Join point model
    - same as AspectJ
  - Means of designating join points
    - a subset of AspectJ pointcuts (constructor & method call receptions, field sets/gets)
  - Means of affecting behavior at join points
    - Meta-objects code
  - Means of structuring the above
    - dispersed in a two step-weaving

Conclusion

- AspectJ experiments with run-time reflection: the best of both worlds!
  - Fine-grained poincuts declaration
  - Run-time weaving
- Prototype implemented in a few days!

Open Issues

- How to choose the relevant join points in the program code?
- Poincuts are weaved at compile-time!
- Source code is needed (cf. AspectJ)
- Run-time overhead and security issues?

"Meta aspect" + meta-object *i* + program specialization

=

aspect *i* in AspectJ

?