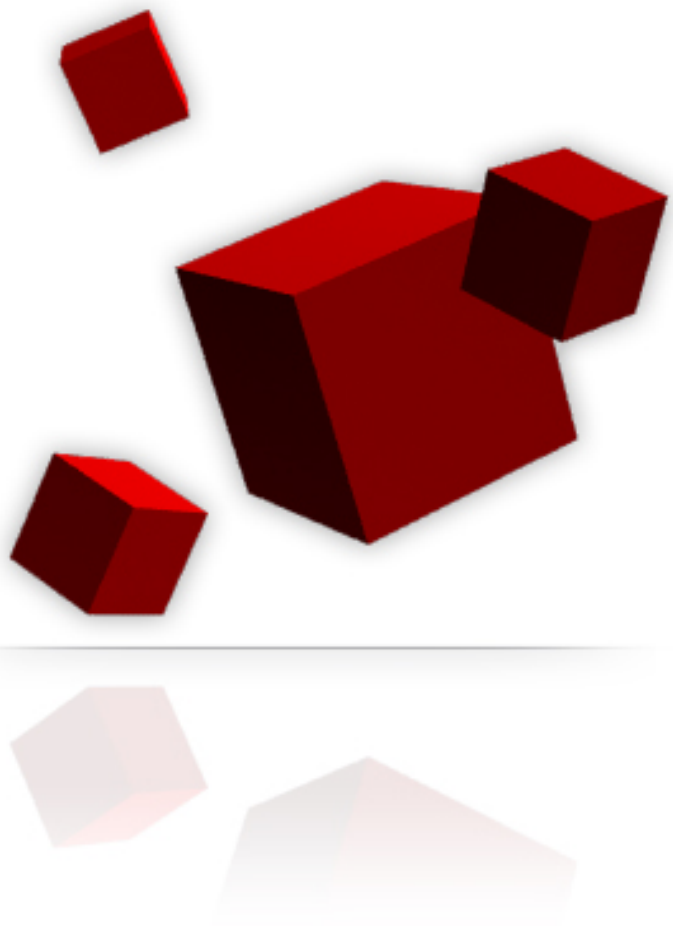
Dynamic Aspect-Oriented Programming

Alexandre Bergel
bergel@iam.unibe.ch

Software Composition Group
Universität Bern, Switzerland



Outline

1. Why do we need dynamic AOP?
2. Example
3. PROSE: Event-based and JIT compilation
4. Steamloom: Run-time speed as a major concern
5. AspectS: High flexibility
6. Classboxes: Aspect as class extension
7. Evaluation

Dynamically Adapting an Application

- A lot of application cannot be halted in order to be updated: financial system, real-time monitoring, embedded system, ...
- Dynamic adaptation of a running application allows the application's behavior to be changed without stopping and restarting it.

Why Dynamic AOP?

- Strategy Pattern helps to produce adaptable applications, however all the ways an application will have to be adapted cannot be anticipated.
- AOP helps to define cross-cutting changes.
- Adapting an application by applying dynamically some AOP techniques

Example: Embedded System in a Satellite

- HEDC is a satellite recently launched [4] intended to observe the sun and to build a catalog of events like sun flares.
- Data are accessible to scientists through a web service implemented with a java servlet.
- For each HTTP request a new session was created, leading to a performance degradation when the number of users was high.
- The system relies on a proprietary library, so the source code was not available.
- The fix was to replace the **new Session()** code.
- This example illustrates how useful DAOP can be.

PROSE 1

- The Java Virtual Machine Debugger Interface (JVMDI) triggers some execution events.
- PROSE 1 [3] is based on providing some notification handlers for events like: method entry, method exit, field access, field modification.
- Handlers can be added, removed and replaced at run-time.
- Managing events offers low performance.

Example with PROSE 2 (1/3)

Weaving location specific access control at the start of methods defined in AService:

```
class SecurityAspect extends Aspect{
   Crosscut accCtrl = new MethodCut(){
      public void ANYMETHOD(AService thisO, REST anyp){
       //Advice that check the access
      }
      {// ... && before m*(...) && instanceof(Remote)
       setSpecializer(
            (MethodS.BEFORE)           .AND
            (MethodS.named("m.*"))     .AND
            (TargetS.inSubclass(Remote.class)) );
      }
   }
}
```

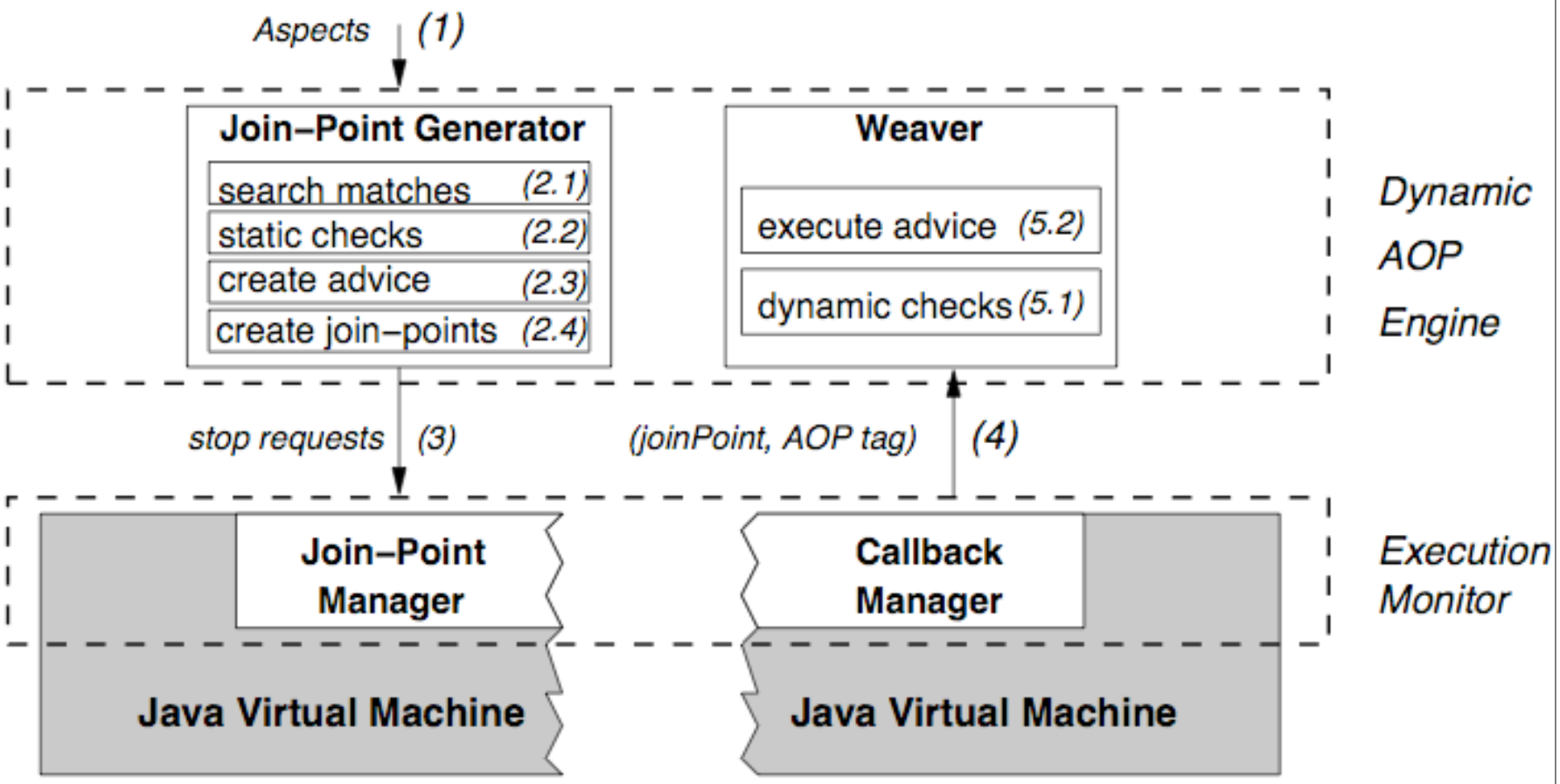

Example with PROSE 2 (2/3)

- Aspects are first-class entity
- An aspect extends the `Aspect` base class.
- It contains one or several crosscut objects.
- A crosscut object represents a modification that is applied on the base system when the aspect is installed.
- This crosscut object defines an advice and describes the join-points where the advice has to be executed.
- An advice is a piece of code executed when a join-point is reached during the execution of the base system.

Example with PROSE 2 (3/3)

- A join-point is a description of the code location where the execution must be interrupted in order to execute advice.
- The number and types of join-points defined by a crosscut object depend on the signature of the advice method.
- The specializer further restricts the set of join-points to entries in methods whose name matches the regular expression “m.*”.
- Specializers are composable using NOT, AND and OR.

PROSE 2: Architecture (1/2)



PROSE 2: Architecture (2/2)

- In the upper layer, the AOP engine accepts aspects (a) and transforms them into basic entities like join-point requests (2.1-2.4).
- It activates the join-point by register them to the *execution monitor* (3).
- When the execution reaches one of the activated join-points, the execution monitor notifies the AOP engine (4) which then executes an advice (5).

PROSE 2: Performances

- PROSE 2 [3] is based on a modified IBM Jikes JVM.
- Use a modified version of the baseline compiler to insert code that checks for the presence of advice at every possible join point.
- Hooks are inserted and called at every point that may be a joint point regardless of whether there is advice code associated with it or not.
- Decorated virtual method calls are slowed down up to 8.8 times!

How performance can be improved ?

- The cost of Prose is high because whenever a message is sent it has to be verified if an advice needs to be invoked or not.
- Performance is one of the main concern with Steamloom [3].
- It add a new keyword **deploy** in the the language.

Aspect Deployment with Steamloom

Steamloom [3] is an implementation of Caesar [6]. It introduces a new keyword to weave "locally" an aspect.

The execution of a **deploy** statement with an aspect as a parameter triggers aspect weaving, i.e., the hooks needed to execute advice is added and deleted at run-time.

```
deploy (anAspect) {
   // Weaving
   ...   // Code
   // Unweaving
}
```


Fibonacci Example (1/3)

```
public class App {
   public void run () {
      this.run(10);
   }

   public void fibstart (int n) {
      this.fib(n);
   }

   public int fib (int k) {
      return (k>1) ? fib(k-1)+fib(k-2) : k;
   }
}
```


Fibonacci Example (2/3)

```
public class FibonacciAspect {
   private int ctr = -1;

   before():
      execution (void App.fibstart(int)) {ctr = 0; }

   after():
      execution (void App.fibstart(int)) {
         System.out.println(ctr);}

   before():
      execution (int App.fib(int)) {ctr++; }
}
```


Fibonacci Example (3/3)

Applying the aspect

```
deploy public class DeploymentAspect {
   around(): call (void App.run()) {
      deploy (new FibonacciAspect())
         {proceed();}
   }
}
```

proceed() triggers the original definition of run. The deploy statement weaves the fibstart and fib function with the aspect (**new FibonacciAspect()**).

Scope of an aspect

- An aspect can either be local to a thread (advices are executed only for a particular thread, else they are ignored), or it can be attached to a particular instance.
- In the previous example, the FibonacciAspect is local to the thread that deploys it.
- A brief snippet of code is inserted before every call to advice functionality to check if it occurs on the right instance or in the right thread.

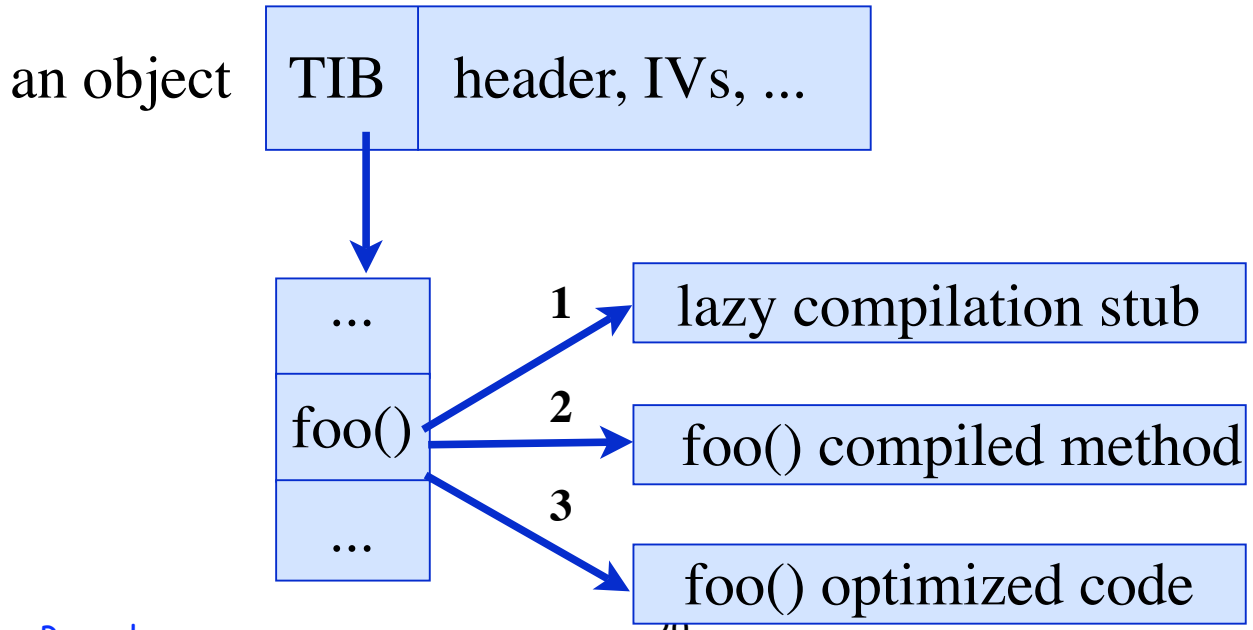
Aspectual Polymorphism

The instance passed to **deploy** (that represents an aspect) can be the result of a computation:

```
deploy class FibDeployment {
   around(): call (void App.run()){
     FibonacciAspect l = null;
     if (...)
       l = new FibonacciAspect();
     else
       l = new SubclassOfFibonacciAspect();
     deploy (l) {proceed();}
   }
}
```


Just-in-time and Lazy Compilation

- Performance is a major concern for Steamloom [3].
- First call triggers the compilation and second one the optimization.
- TIB = Type Information Block. It contains pointers to all virtual methods of the class.



Deployment of an instance-local aspect

- Deploying an aspect on an instance make this object point to a particular TIB

Performance

- 4% of overhead compare to the IBM's Java VM.
- Result from addition operations Steamloom performs at class-loading time and just-in-time compilation.

AspectS

- Mainly because of the static type system, dynamic method introduction are not allowed.
- Limited number of join-points can be hooked:
  - Prose does not handle *cflow*
  - Steamloom has some difficulty with *around*
- Better flexibility with a dynamic typed language.
- AspectS is implemented in Squeak, an open-source Smalltalk [7, 8].

AspectS

- An Aspect is a set of advices.
- An advice is a set of JointPoints and a qualifier
- A JointPoint refers to a class and one of its method.
- An AdviceQualifier used to restrict the advice to a subset of instances and to restrict the join point to a particular control flow.
- 5 kinds of advices: exception handler, before/after, around, introduction, cflow.

Example: Tracing a factorial

In Squeak, the factorial is implemented as:

```
Integer>>factorial
    self = 0 ifTrue: [^ 1].
    self > 0 ifTrue: [^ self * (self - 1) factorial].
    self error: 'Not valid for negative integers'.
```

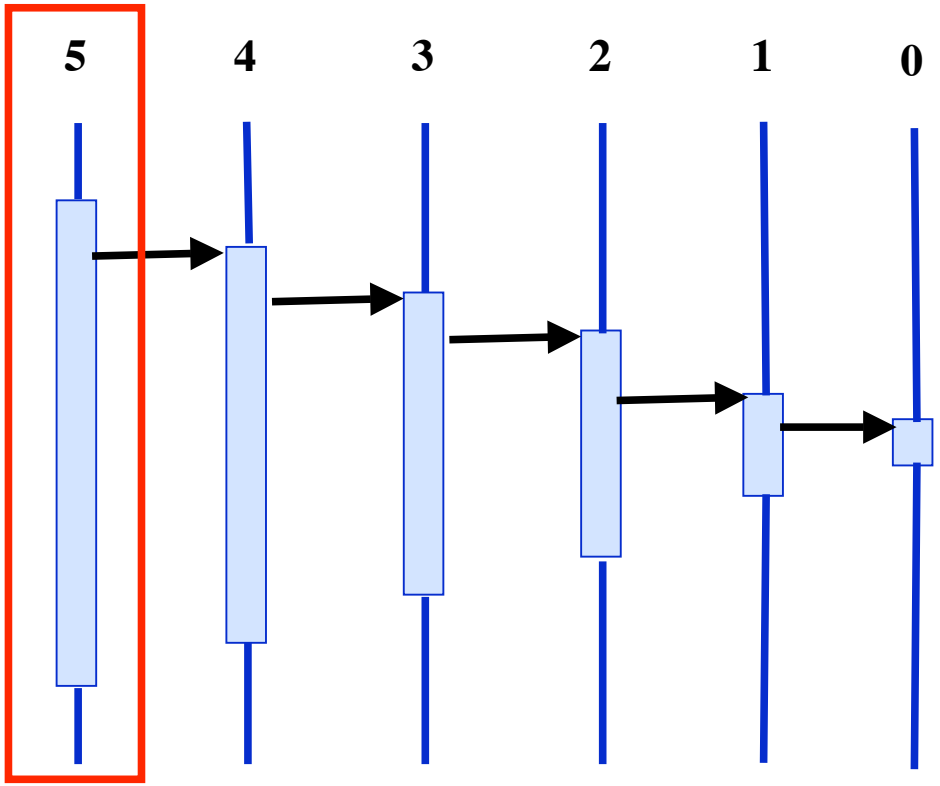
It is invoked by sending a message **factorial** to an integer

Example: Tracing a factorial

To echo the **initial** reception of a factorial message.

```
adviceFactorialInFirst
  ^ BeforeAfterAdvice
      qualifier: (AdviceQualifier attributes:
                    {#receiverclassSpecific .#cfFirstClass})
      pointcut: [OrderedCollection with:
                    (JoinPointDescriptor
                       targetClass: Integer
                       targetSelector: #factorial)]
      beforeBlock:
        [:receiver :arguments :aspect :client|
          Transcript show: 'fac: ', self printString]
```


Example: Tracing a factorial



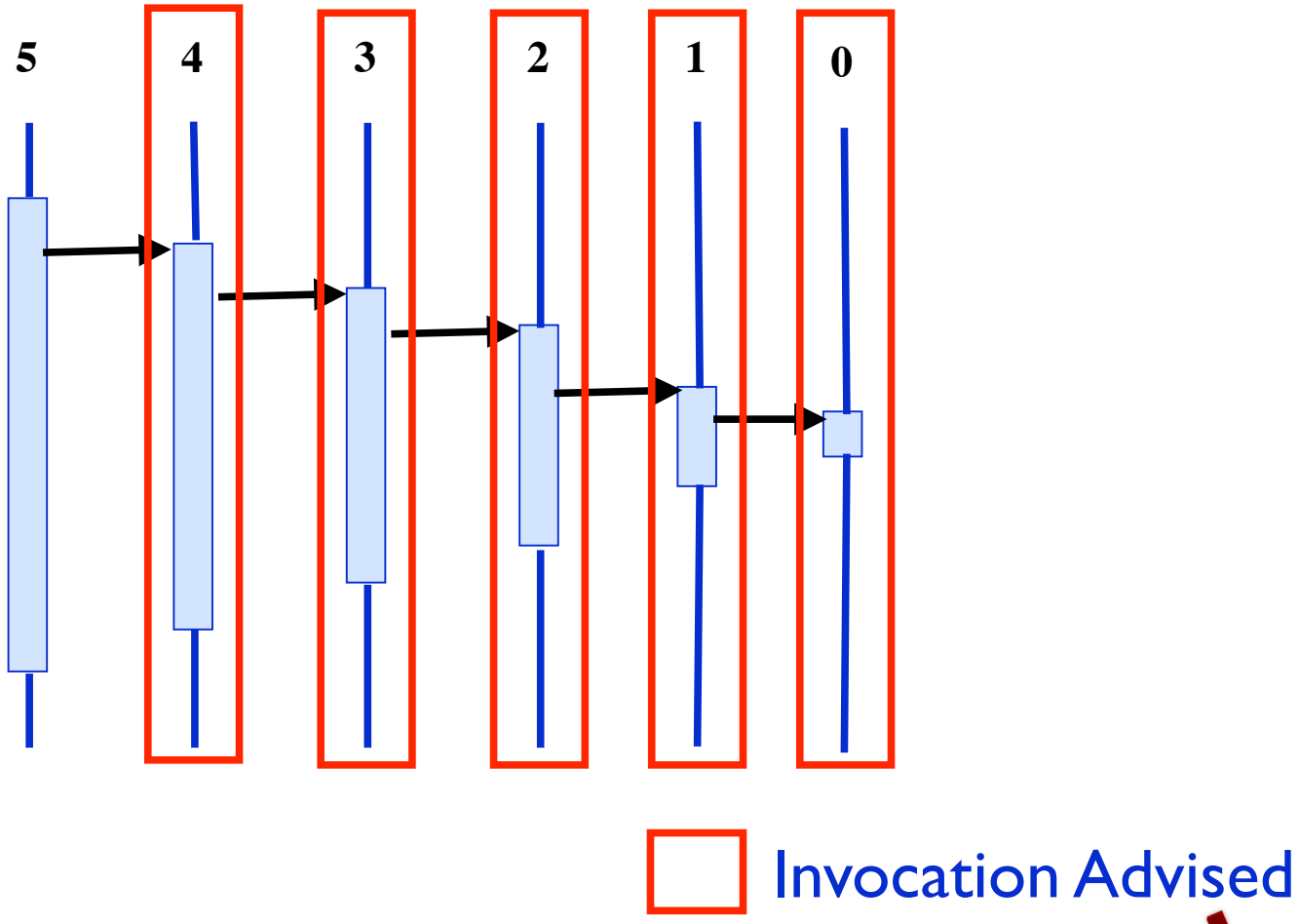
Invocation Advised

Example: Tracing a factorial

To echo the **initial** reception of a factorial message.

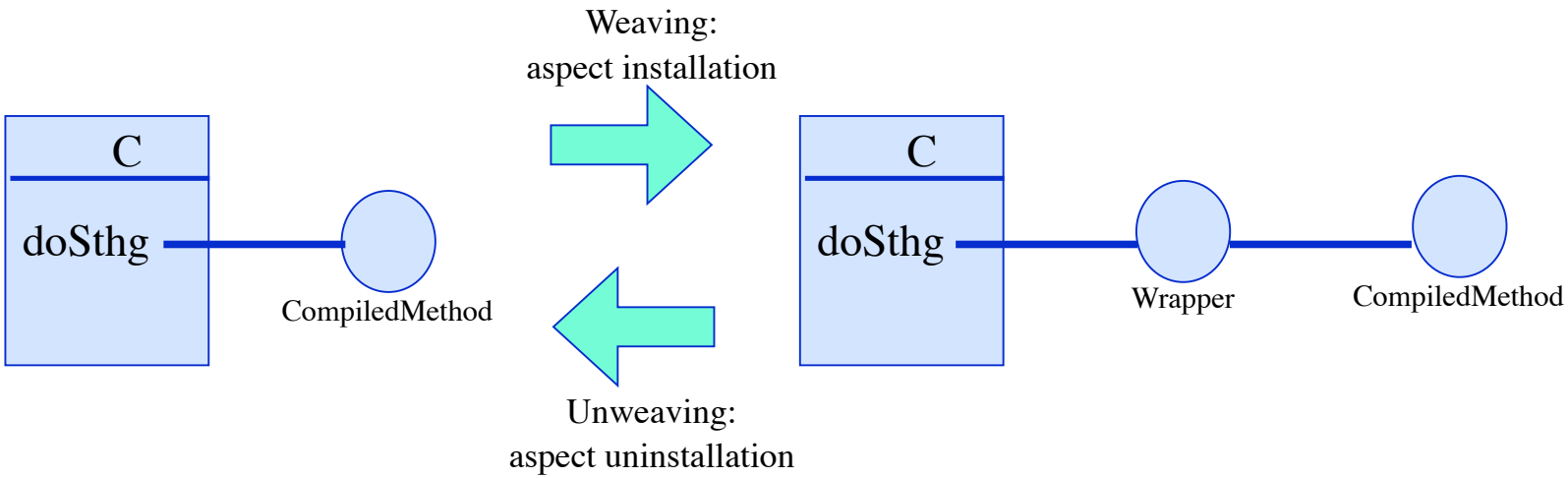
```
adviceFactorialInFirst
  ^ BeforeAfterAdvice
      qualifier: (AdviceQualifier attributes:
        {#receiverclassSpecific .#cfAllButFirstClass})
      pointcut: [OrderedCollection with:
                  (JoinPointDescriptor
                     targetClass: Integer
                     targetSelector: #factorial)]
      beforeBlock:
        [:receiver :arguments :aspect :client|
          Transcript show: 'fac: ', self printString]
```


Example: Tracing a factorial



Implementation

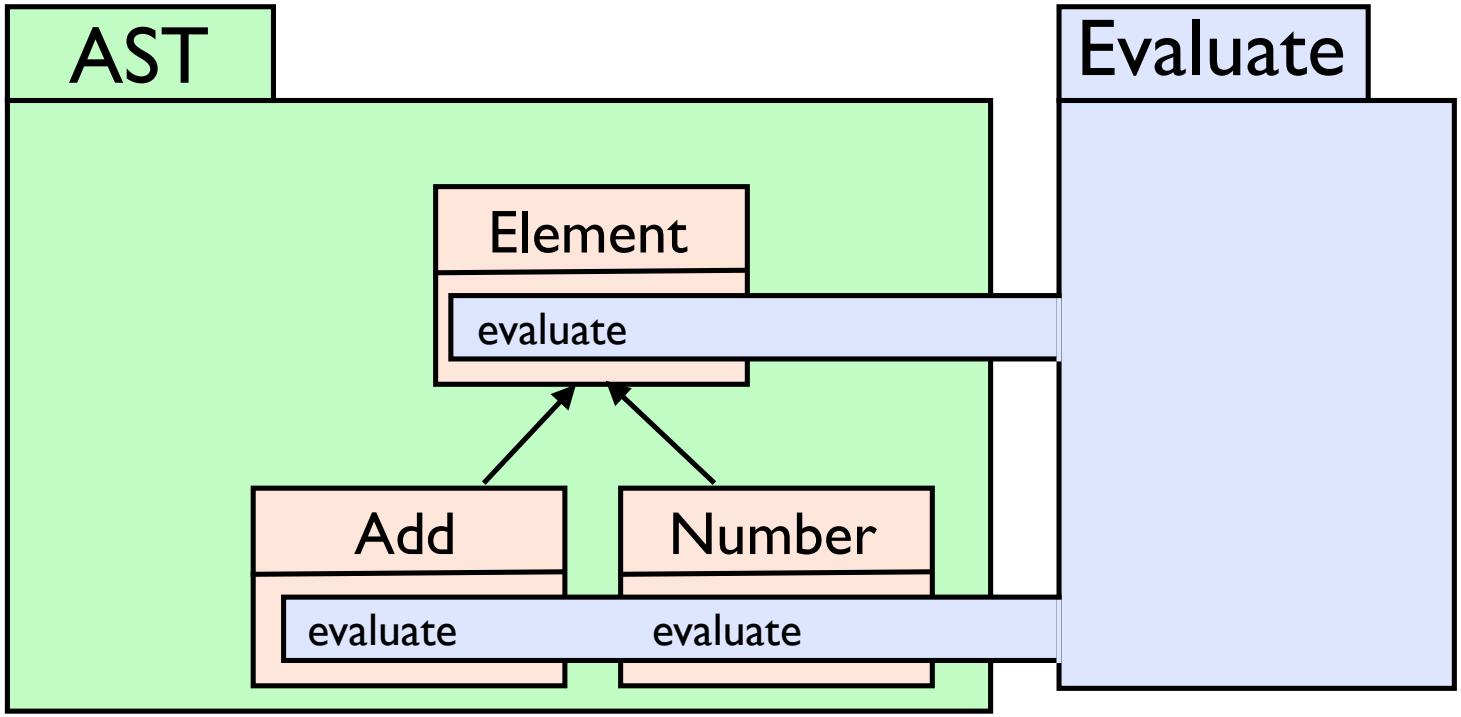
- Based on John Brant's *method wrapper*, a mechanism to add behavior to a compiled Smalltalk method.
- Sending the uninstall message.
- Weaving and unweaving at run-time.



Security and Aspects

- Steamloom can bound the visibility of an aspect to a set of objects or to a particular thread.
- Does not modify the flow of the original application.
- Classboxes does not offer join-points such as before/after or around but use class extension to define aspects.

Dynamic Scoped Aspects with Classboxes [9]



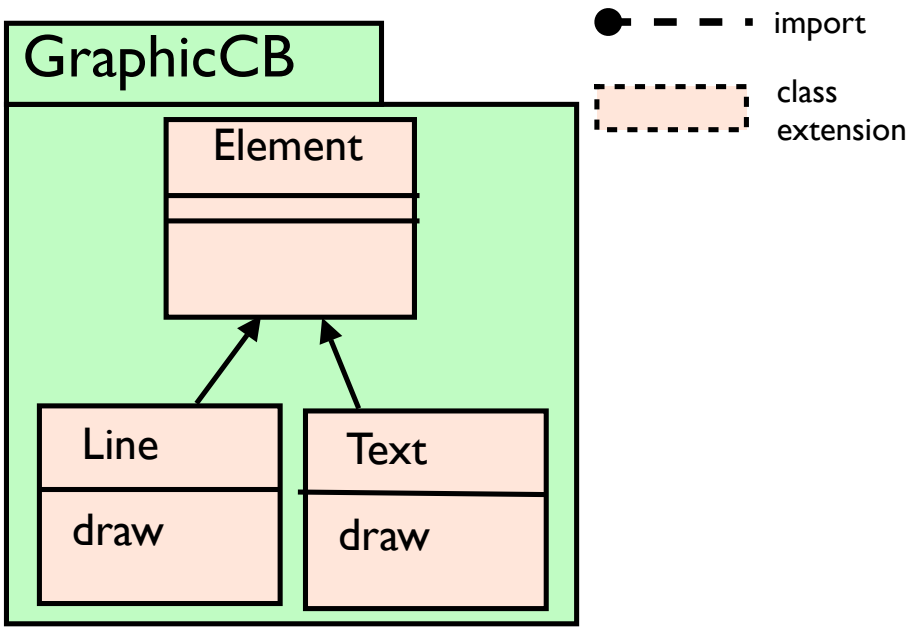
With AspectJ:
- Global Scope
- At Compilation Time

Consequences:
- Conflicts
- Static Configuration
- Client might break

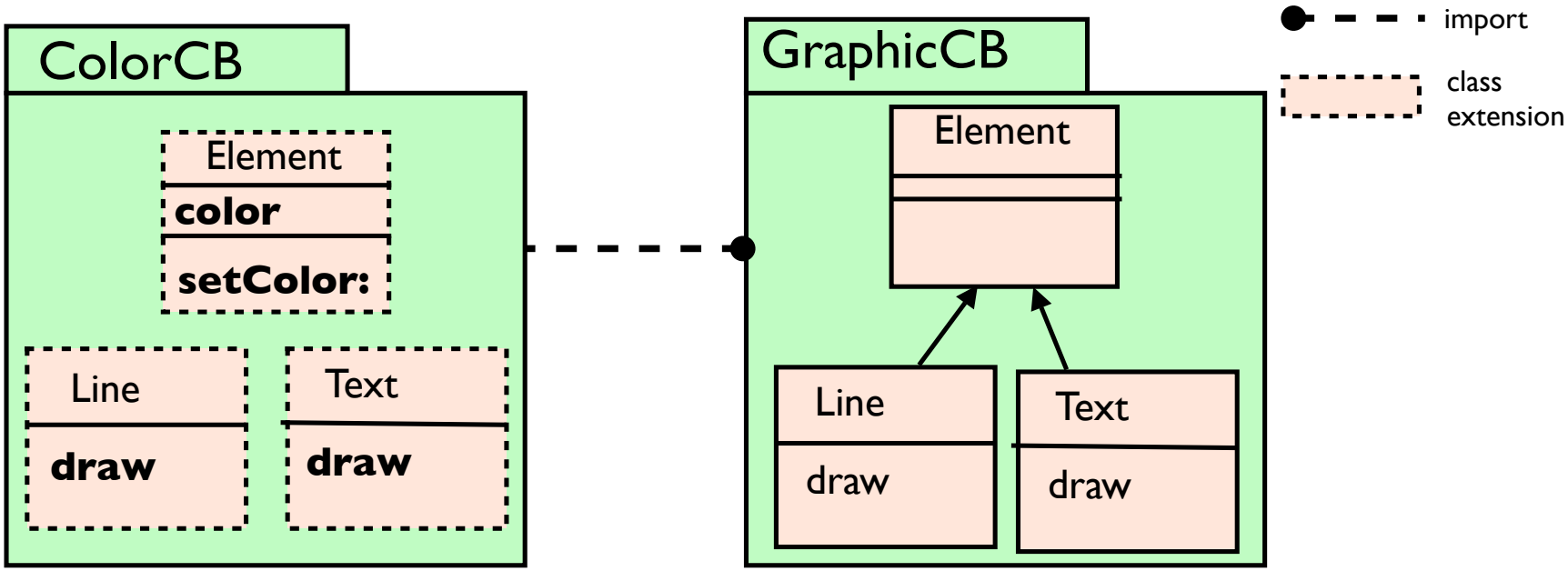
Aspects with Classboxes

- An aspect is a set of definitions (classes) and extensions (methods, instance variables).
- Can be dynamically installed and uninstalled.
- Class extensions are visible **only** in the classbox that define them and in other classboxes that import the extended class.
- Applying an aspect does not break former clients.
- Two aspects cannot conflict with each other.

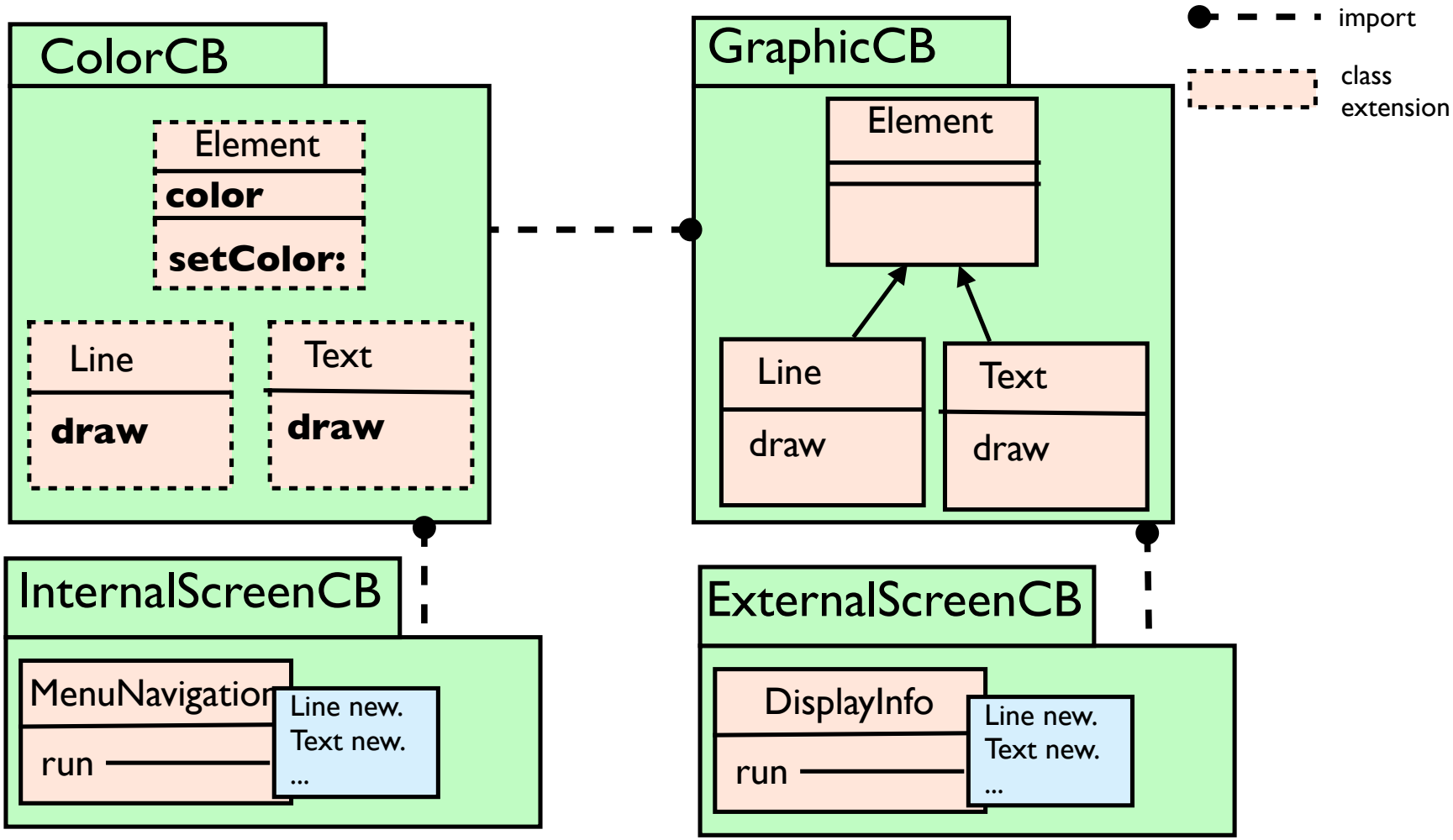
Cellphone Example



Cellphone Example



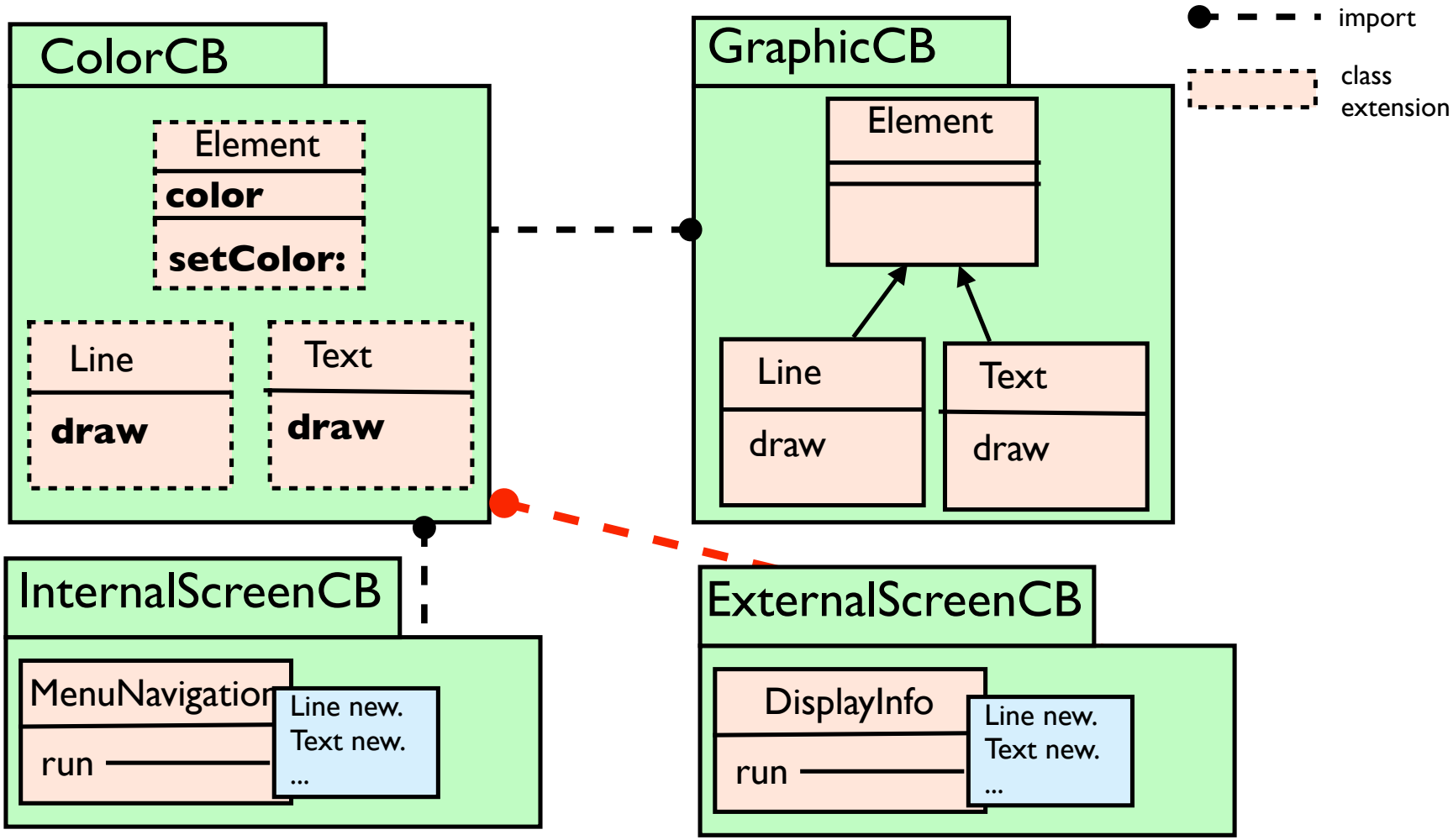
Cellphone Example



Cellphone Example

- There is one hierarchy of graphical elements
- Which is extended with a color concern. **But these extensions are scoped**.
- From the point of view of the internal screen elements are colored
- But from the point of view of the external one they are colorless.

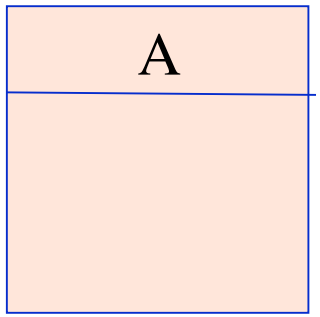
Both Screens are Colored



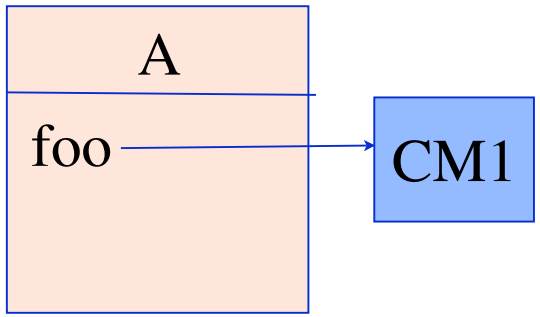
Implementation

- In Squeak but applicable to other OO languages (Ruby, ...).
- New method lookup semantics.
- No need to modify the VM.
- No cost for method additions.
- Cache for redefined methods.

Cache Mechanism (1/4)



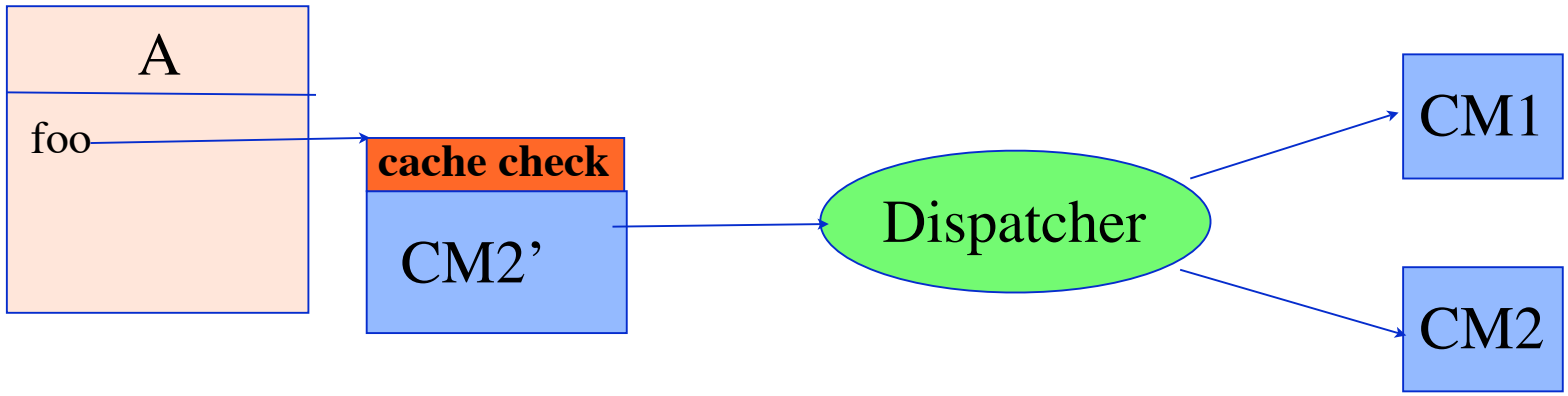
Cache Mechanism (2/4)



Cache Mechanism (3/4)

Method redefinition

Cache Mechanism (4/4)



Method execution: A new foo

Dynamic AOP

- Dynamic AOP requires to have a first class representation of aspect.
- Many issues with static type languages (no new method introduction, limited number of join-point)
- Use in Software Architecture Evolution [5]

Different Approaches to Dynamic AOP

- Pre-runtime instrumentation
  - Use of the EAOP preprocessor (EAOP, JAC, JBoss AOP, PROSE 2)
  - Load-time (JAC, JBoss AOP)
  - Just-in-time compiler (PROSE)
- Run-time event monitoring
  - PROSE 1
- Run-time weaving
  - Wool
  - AspectS

Bibliography

1. Robert Hirschfeld: *AspectS - Aspect-Oriented Programming with Squeak*. International Conference NetObjectDays 2002.
2. Andrei Popovici, Thomas Gross, and Gustavo Alonso: *Dynamic weaving for aspect-oriented programming*. AOSD'01
3. Christoph Bocksich, Machael Hapt, Mira Mezini, Klaus Ostermann. *Virtual Machine Support for Dynamic Join Points*. AOSD'04
4. E. Stolte and G. Alonso. *Efficient Exploration of Large Scientific Databases*. Intl. Conf. on Very Large DataBases (VLDB), 2002

Bibliography

5. Paolo Falcarin, Gustavo Alonso: *Software Architecture Evolution through Dynamic AOP*. 1st European Workshop on Software Architectures (EWSA) -- ICSE'04.
6. Mira Mezini, Klaus Ostermann: *Conquering Aspects with Caesar*. AOSD'03.
7. Dan Ingalls, Ted Kaehler, John Maloney, Scott Walace, Alan Key: *Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself*. OOPSLA'97.
8. Squeak Home Page: http://www.squeak.org
9. Alexandre Bergel, Stéphane Ducasse: *Dynamically Scoped Aspects with Classboxes*. JFDPA'04