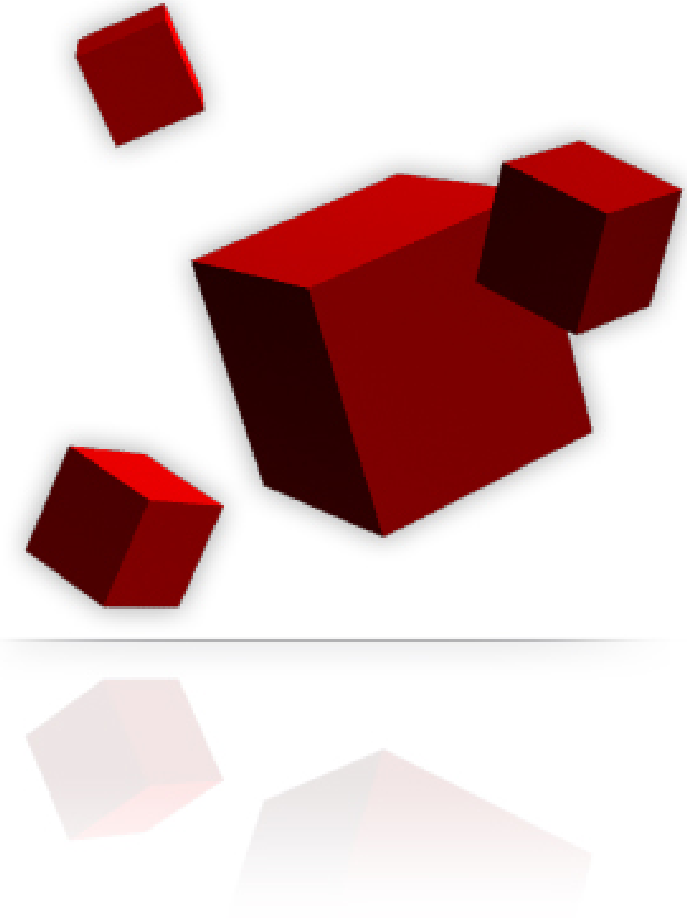
Dynamic AOP with Dynamic Classboxes and Friends



Alexandre Bergel
bergel@iam.unibe.ch

Software Composition Group
Universität Bern, Switzerland

Outline

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

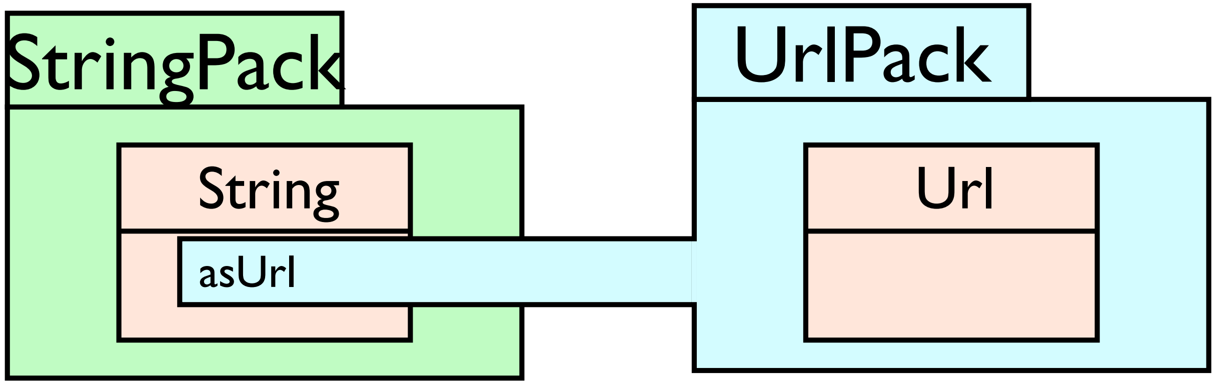
AspectJ: manipulating source code

- Sophisticated mechanism for source-code transformation.
- Weaving done before compile time.
- Aspects are “weaved” away.
- Aspect does not exist at run-time.
- Applying an aspect can break already existing clients.
- Aspects have a global impact.
- Does not fit to bring unanticipated changes on an running application!

Security and Aspects

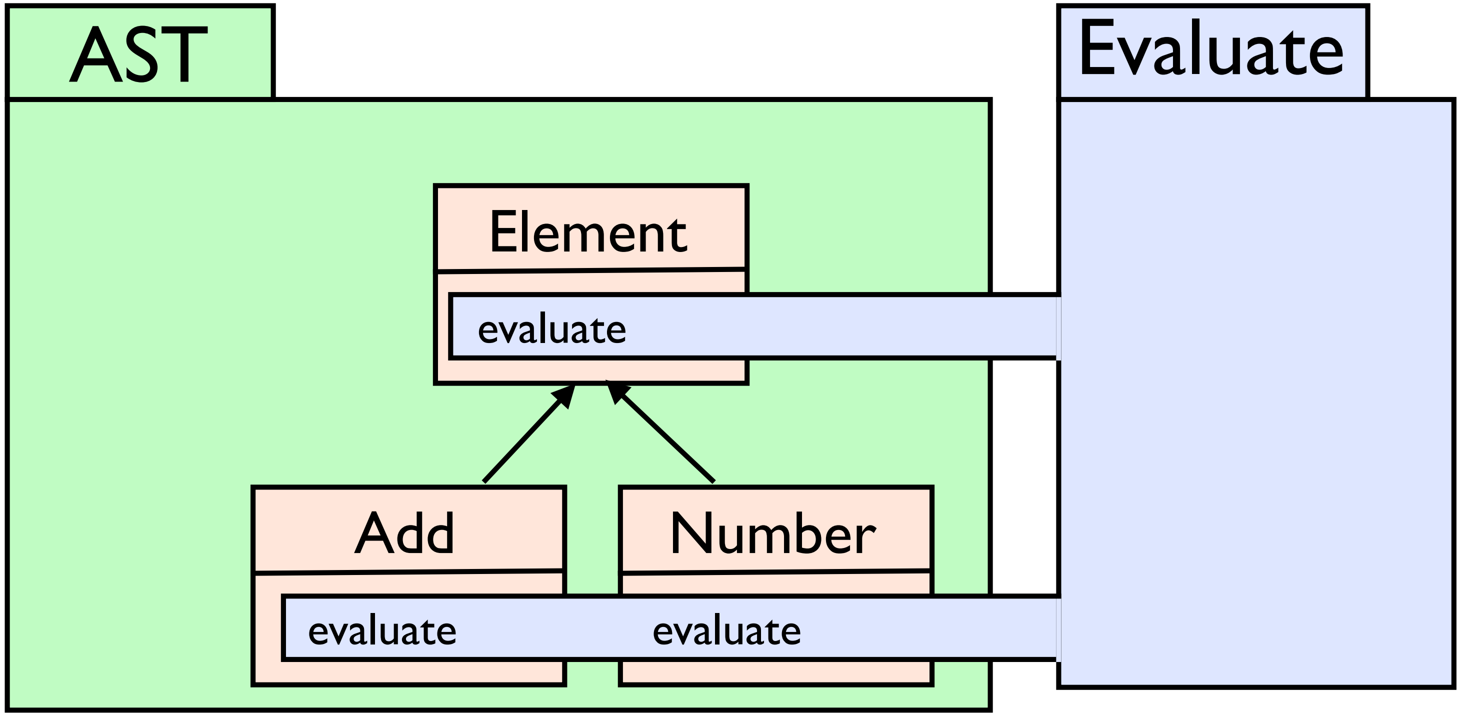
- With classboxes [9] security issues are addressed by emphasizing locality of aspects.
- Classboxes does not offer join-points such as before/after or around but use class extension to define aspects.
- Does not need any source source.
- Classboxes exist at run-time, and their configuration are completely dynamic.

Class extension



- Adding a new instance variable, a new method or redefining one on an already existing class is a *class extension.*
- Decoupling a class definition from field and method definitions
- Relevant: HyperJ, AspectJ, Smalltalk, CLOS, ...

Class extensions as cross-cutting aspect

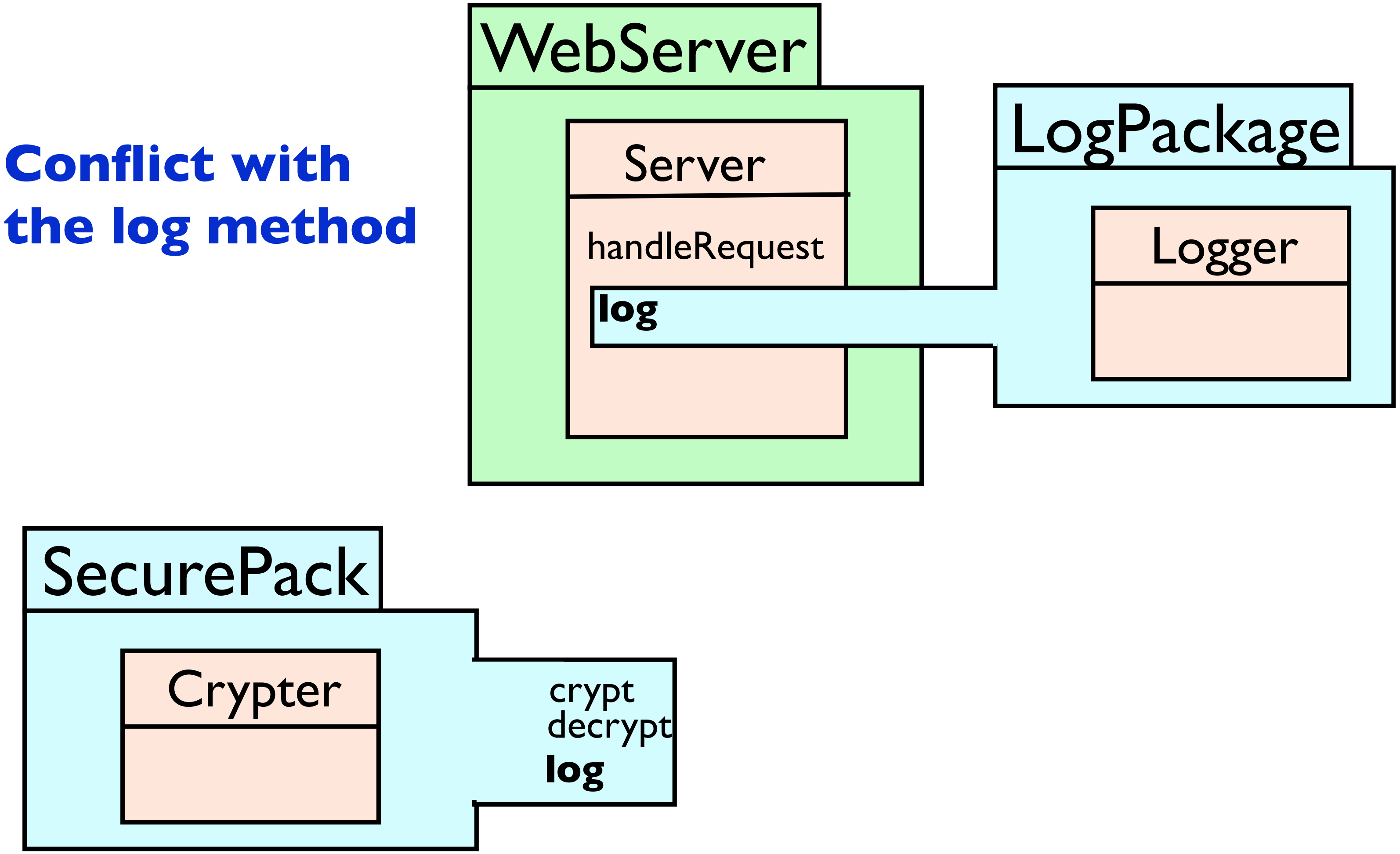


With AspectJ:
- Global Scope
- At Compilation Time

Consequences:
- Static Configuration
- Client might break
- Conflicts may appear

Resolving class-extension conflicts

Conflict with the log method



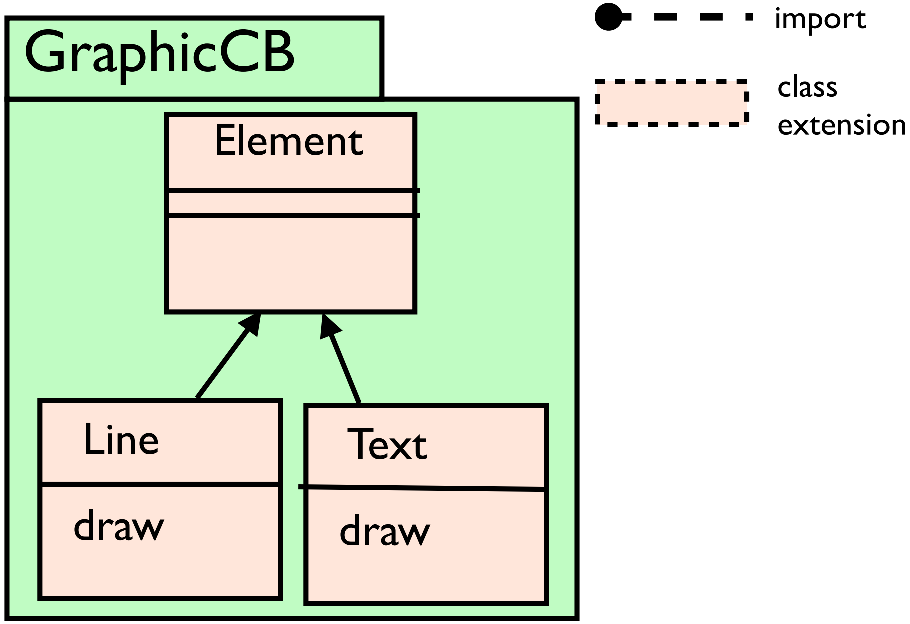
Aspects with Classboxes

- 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.

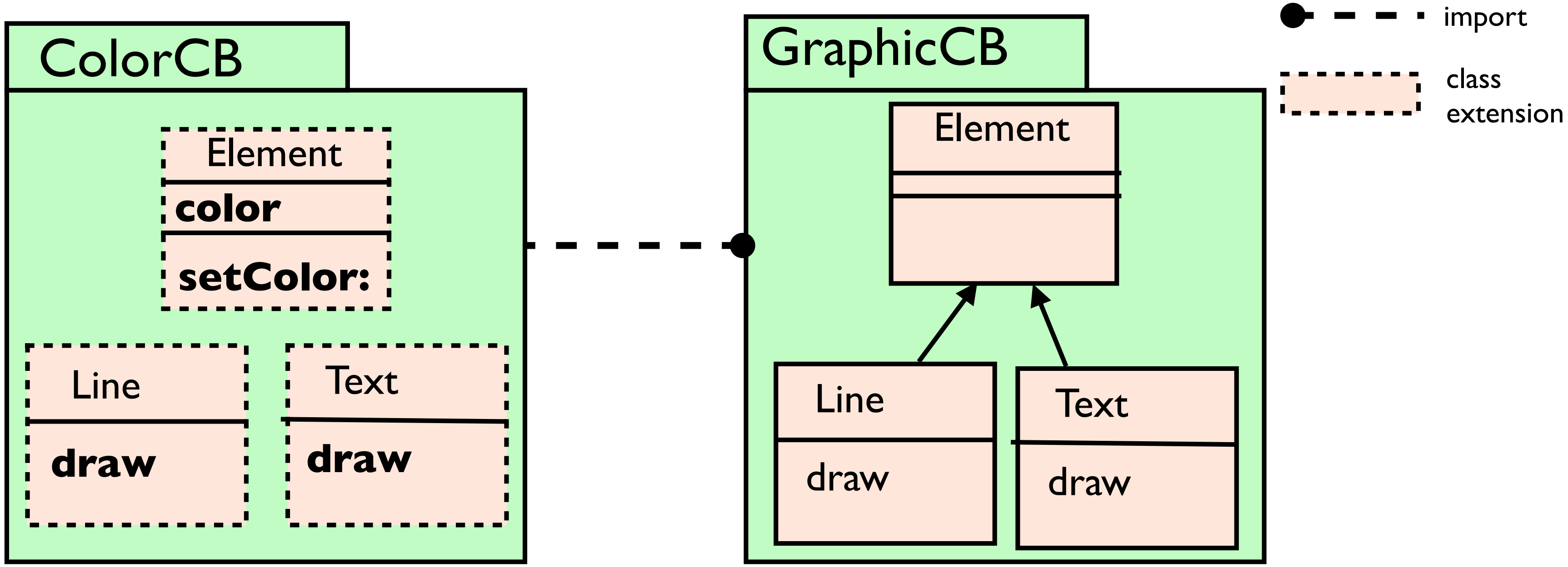
The classbox model

- A **classbox** is a unit of scoping (it behaves as a namespace).
- Within a classbox:
  - Classes can be defined
  - Classes can be imported from other classboxes
  - Methods and instance variables can be defined **on any visible class**
  - Dynamically installed and uninstalled
- Local methods redefinitions take precedence over previous definitions

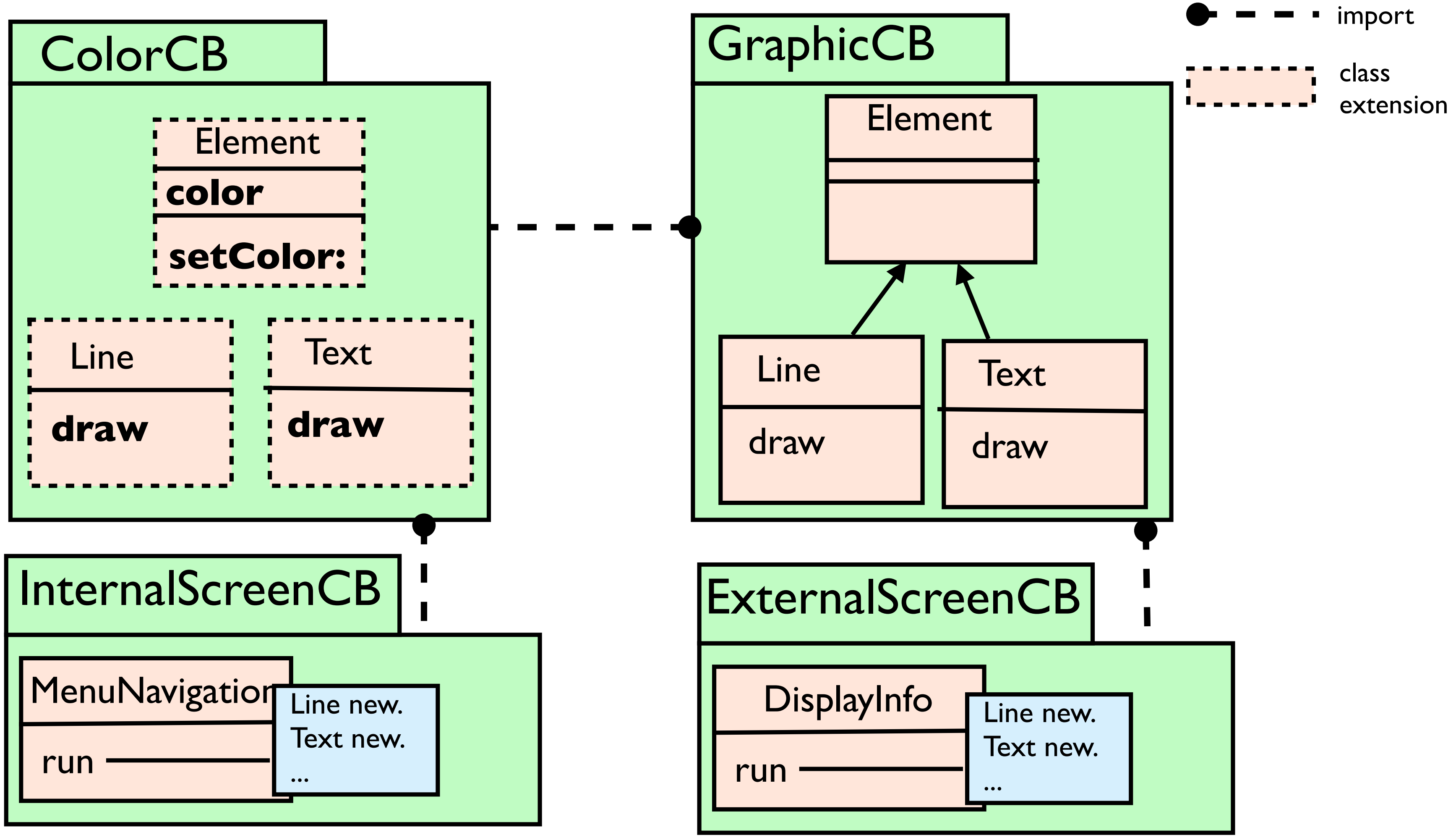
Cellphone Example



Cellphone Example



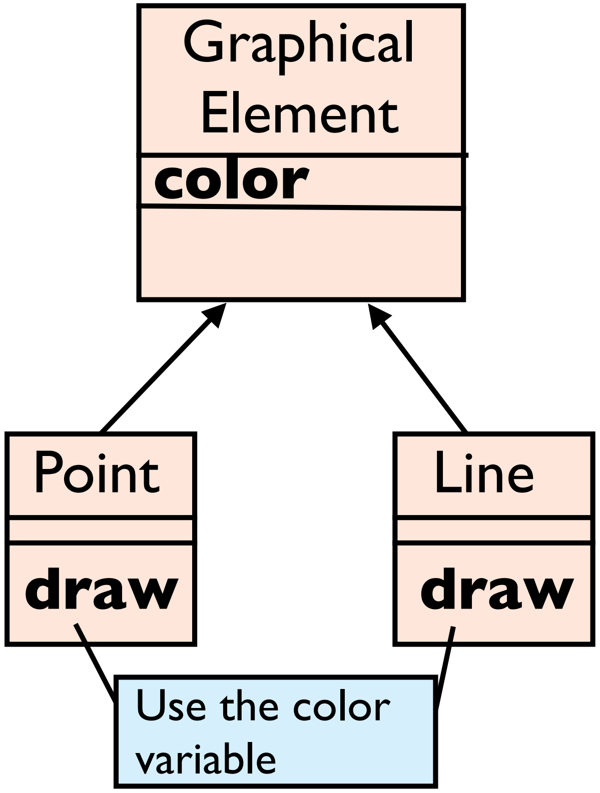
Cellphone Example



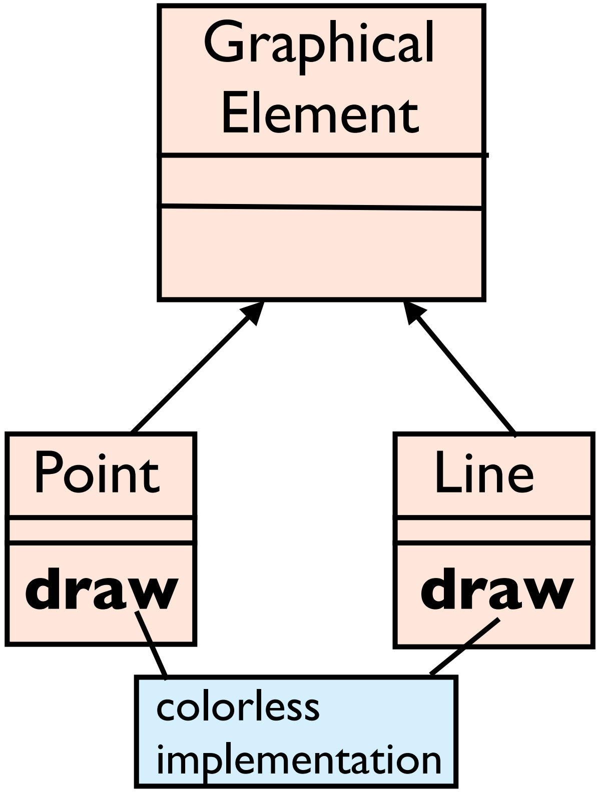
Cellphone Example

- 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.

Different view of a hierarchy

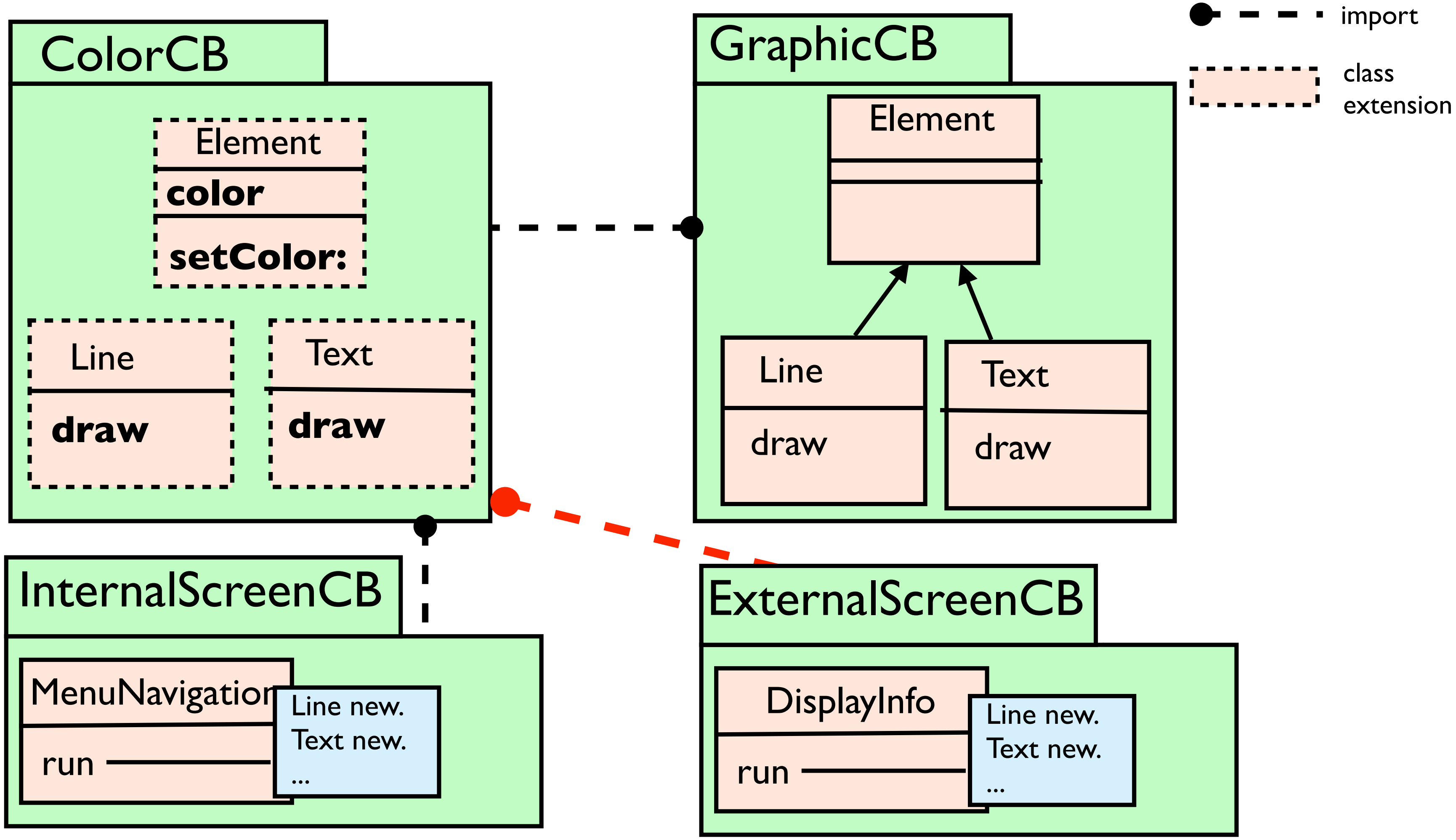


From a colored screen



From a colorless screen

Both Screens are Colored



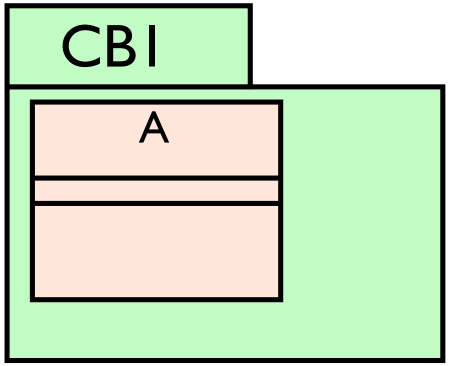
Alexandre Bergel

Implementation

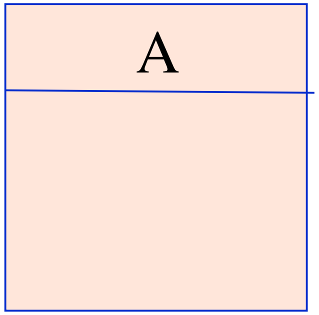
- In Squeak but applicable to other OO languages (Ruby, ...).
- New method lookup semantics.
- No need to modify the VM.
- No cost for method additions.
- Cache for redefined methods.
- Checking the cache validity need 5 extra bytecodes placed at the beginning of the redefined method.

Cache Mechanism (1/4)

In model



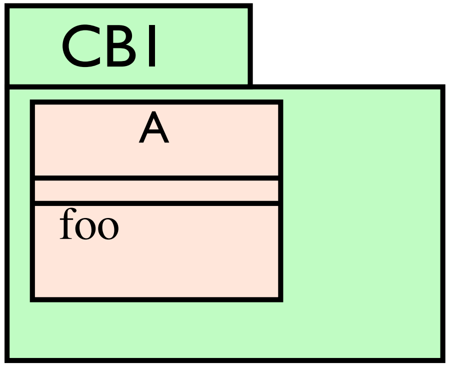
In memory



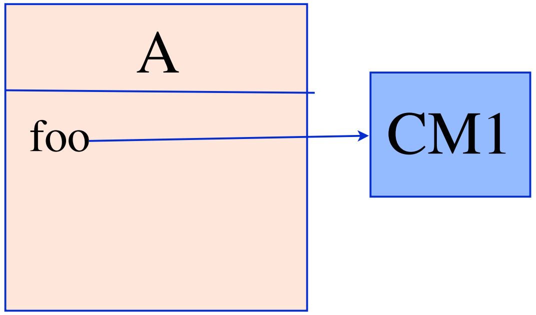
Class Definition

Cache Mechanism (2/4)

In model

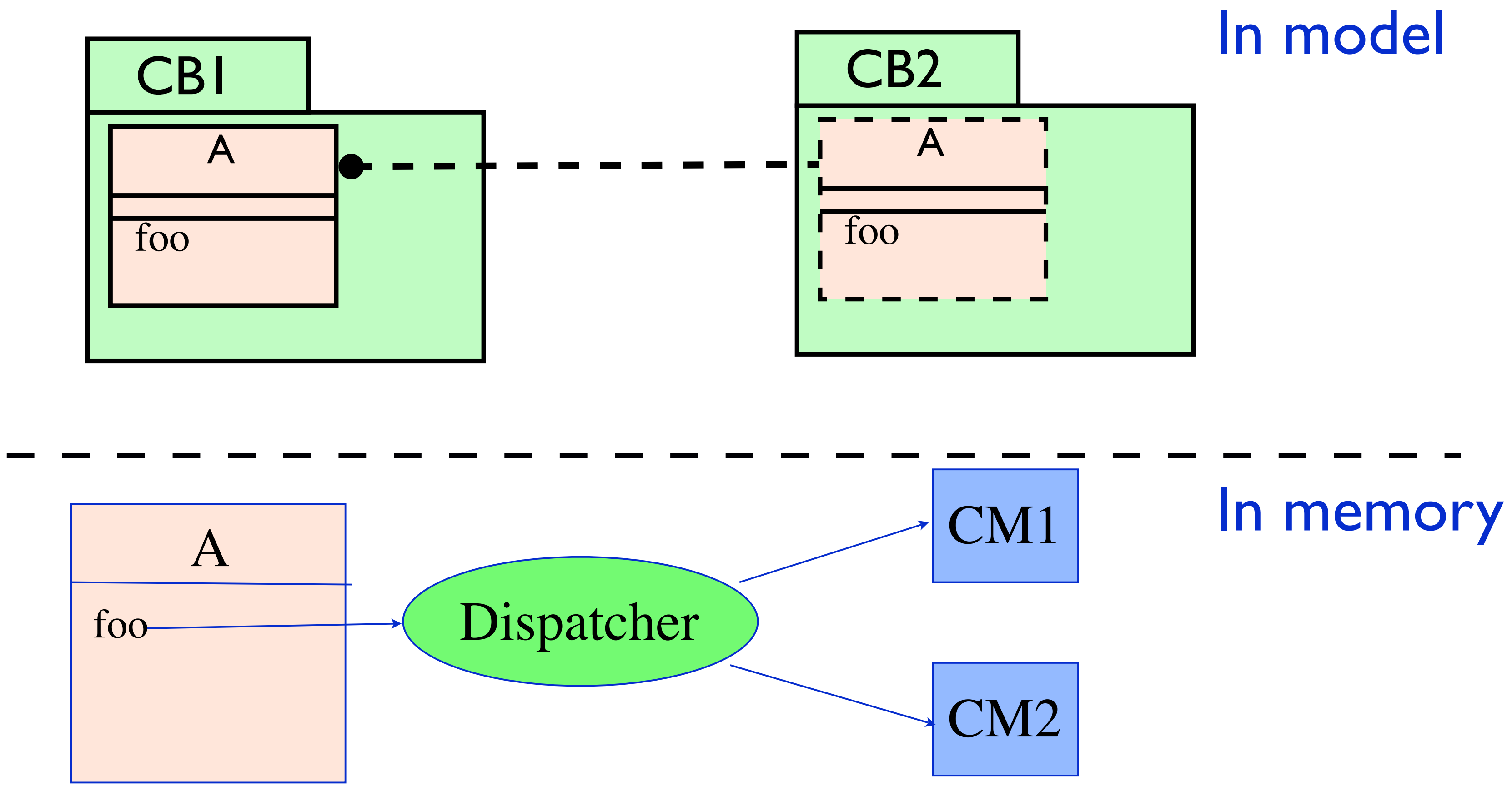


In memory



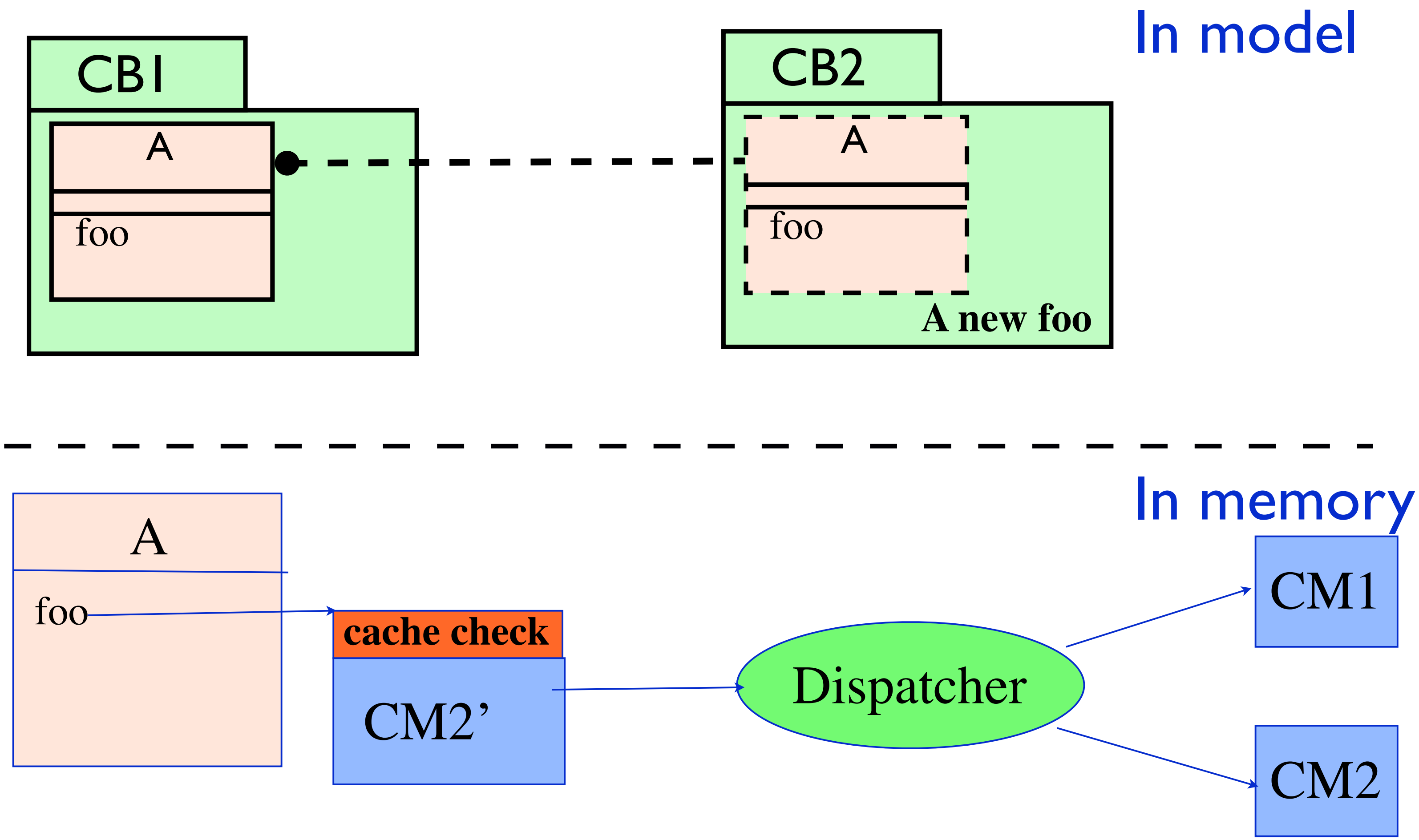
Method Definition

Cache Mechanism (3/4)



Method Redefinition

Cache Mechanism (4/4)



Method execution: A new foo

PROSE 1

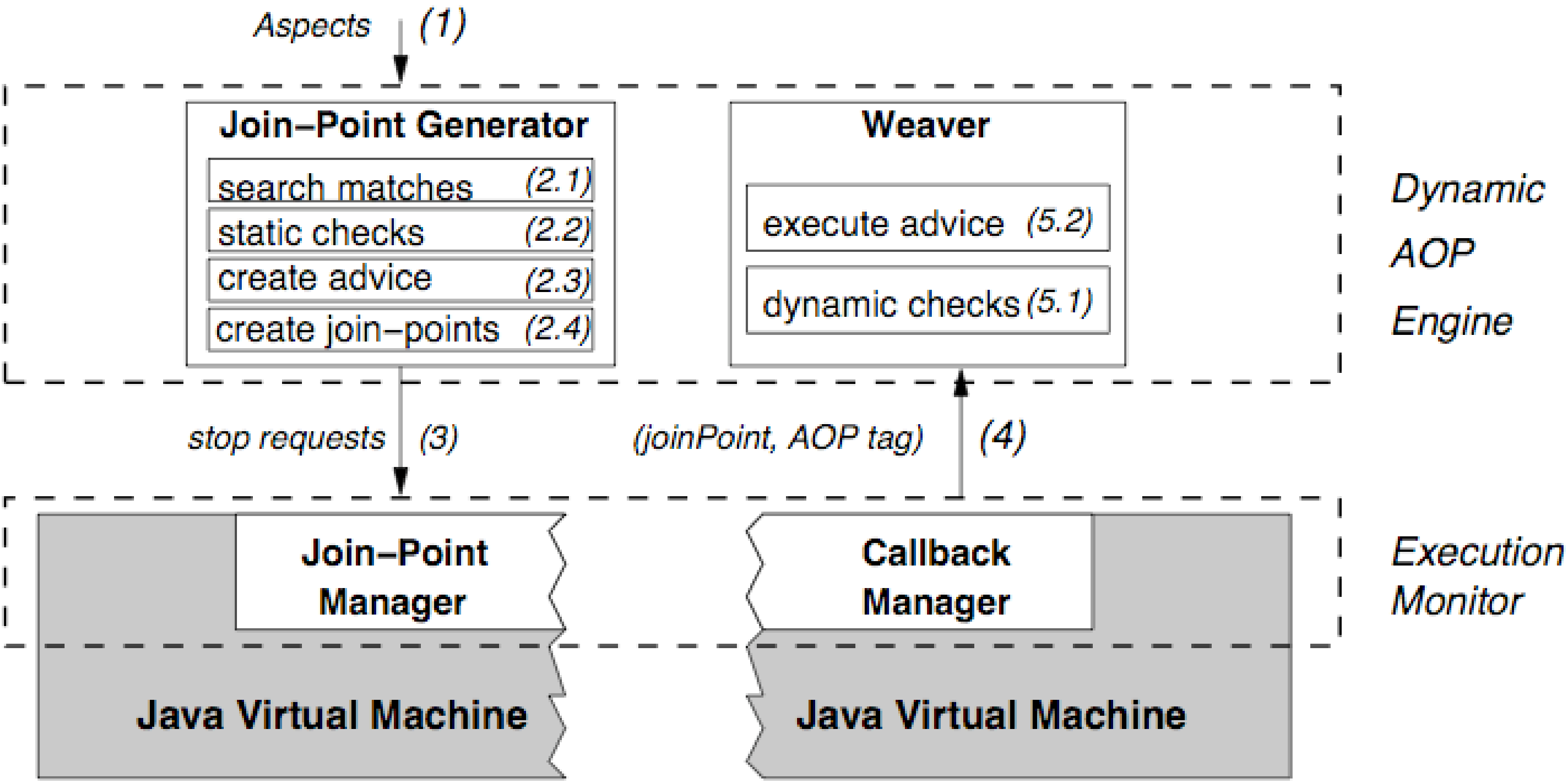
- The Java Virtual Machine Debugger Interface (JVMDI) triggers some execution events.
- PROSE 1 [3] provides some notification handlers for events like: method entry, method exit, field access, field modification.
- Handlers can be added, removed and replaced at run-time.
- Managing events offers low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

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.
- 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!

Steamloom

- Performance is one of the main concern with Steamloom [3].
- It add a new keyword **deploy(anAspect) {...}** in the the language.
- Aspects can either be local to a thread or to a set of instances.
- Details will be presented by Mira Mezini.

Dynamic and static types languages

- 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

- AspectS is implemented in Squeak, an open-source Smalltalk [7, 8].
- 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’.
```

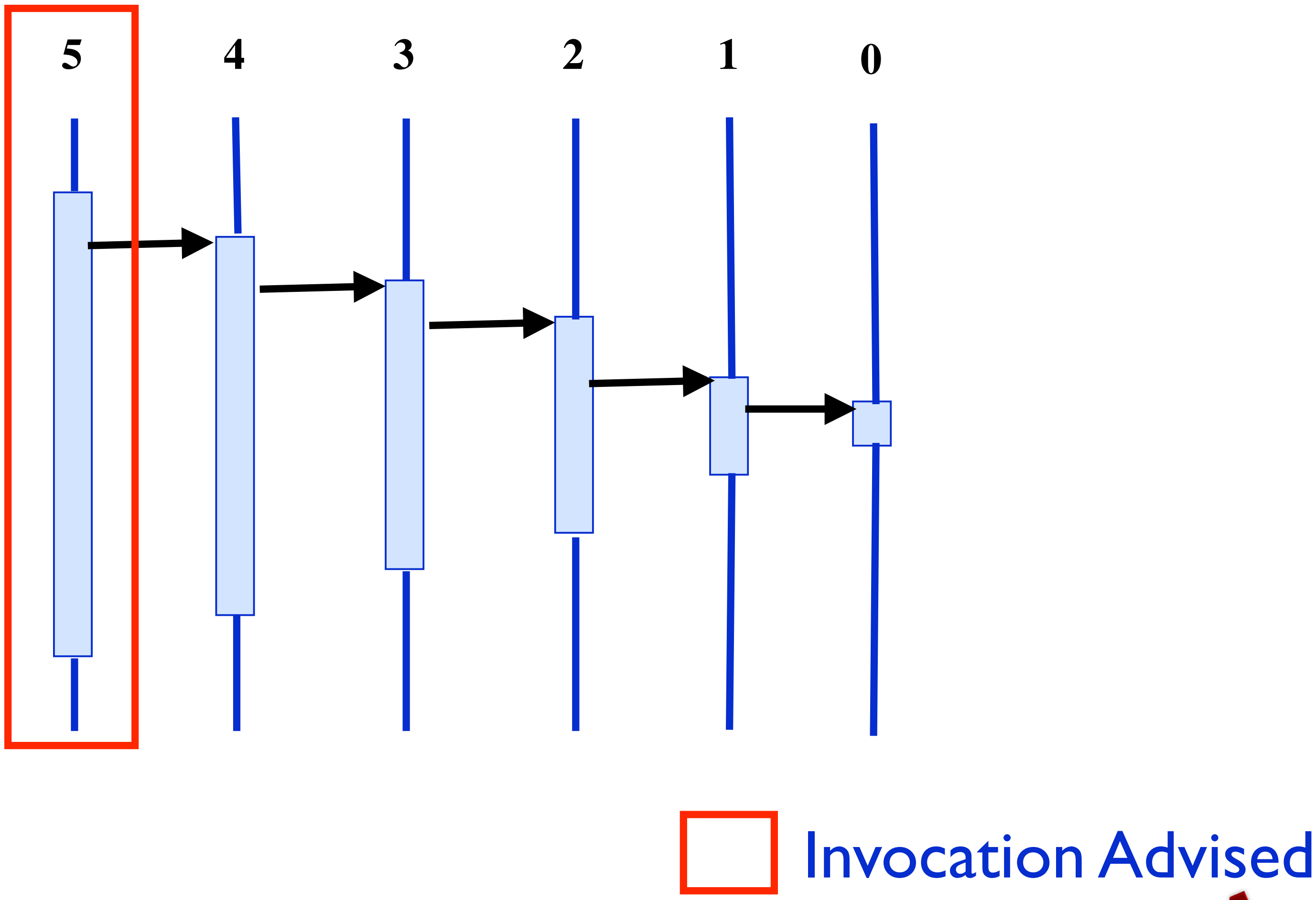
It is invoked by sending a message **factorial** to an integer

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

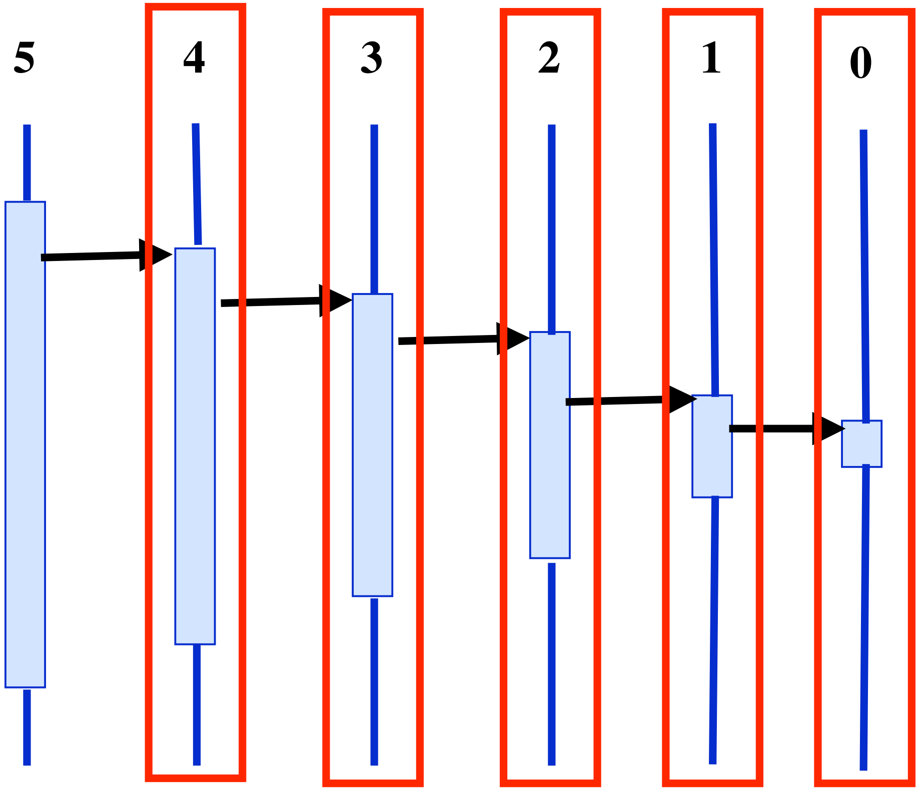


Example: Tracing a factorial

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]
```

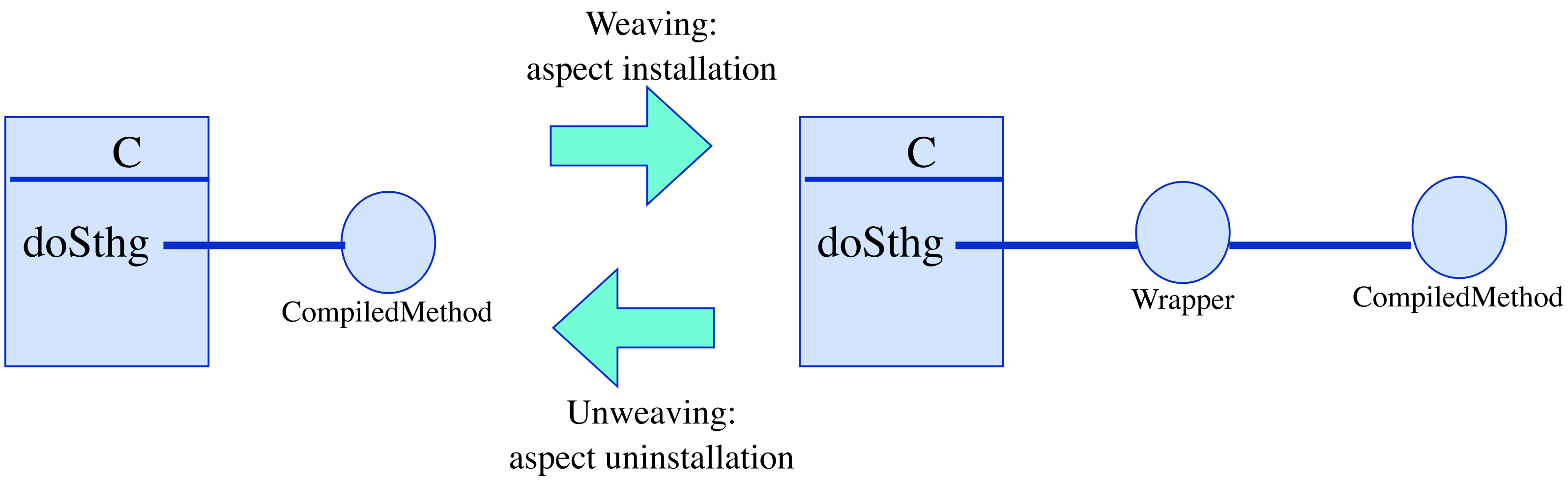
Example: Tracing a factorial



☐ Invocation Advised

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.



Taxonomy (inspired from [4])

- Common characteristic:
  - Time of change: run-time
- Classboxes:
  - Language model: use of reflection
  - Object of change: class extension (variable addition, method addition or redefinition)
  - Scope: classbox
  - Kind of evolution: atomic modification of a group of classes
- PROSE
  - Language model: VM + dynamic Support
  - Object of change: before/after field and method access
  - Scope: global
  - Kind of evolution: advises

Taxonomy

- Steamloom
  - Language model: VM Support
  - Object of change: before/after field and method access +cflow
  - Scope: a set of instances and a thread
  - Kind of evolution: advises
- AspectS
  - Language model: use of reflection
  - Object of change: before/after/around field and method access + cflow
  - Scope: global
  - Kind of evolution: advises + class modificiations

Lesson learnt

- Some kind of applications require to be updated without being stopped and then restarted.
- Classboxes limit the impacts of aspects defined as a set of class extensions (variable addition, method additions and redefinitions).
- Dynamic AOP requires to have a first class representation of aspect (different than AspectJ and HyperJ).
- Many issues with static type languages (no introduction and limited number of join-points).
- It is always a compromise between flexibility (e.g., AspectS) and speed performances (e.g., Steamloom).

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. Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, Günter Kniesel: *Towards a Taxonomy of Software Change*. To appear in: Software Maintenance and Evolution

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