Context-oriented Programming

Pascal Costanza (Vrije Universiteit Brussel, Belgium)
Robert Hirschfeld (Hasso-Plattner-Institut, Potsdam, Germany)

Programs are too static!

- Mobile devices
- Software agents
- Business rules
- Security
- Personalization
- Internationalization

Introduction to OOP.

```
class Rectangle {
  int x, y, width, height;
  void draw() { ... }
}


class Person {
  String name, address, city, zip;
  void display() { ... }
}
```

Context-independent behavior.

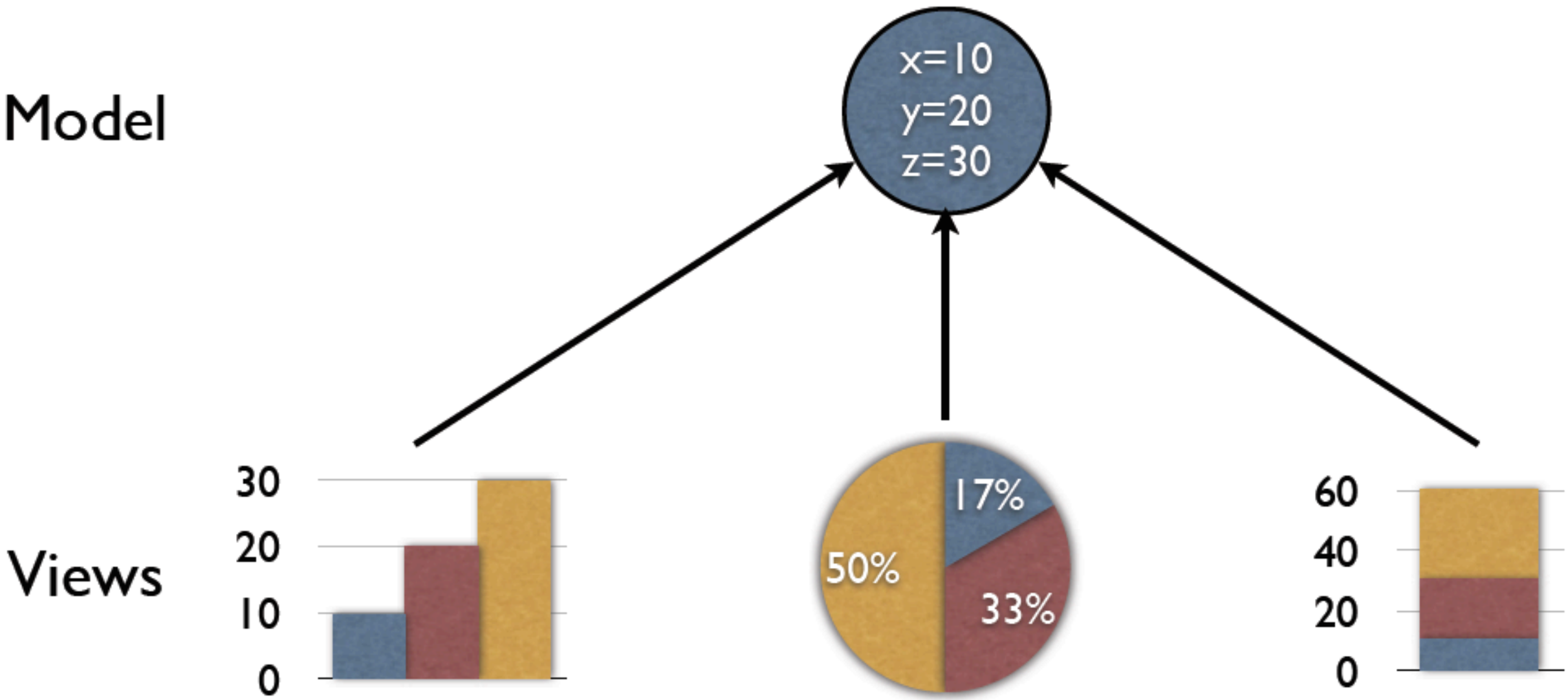
```
class Person {

   String name;


   void display () {
     println(name);
   }


}
```

Context-dependent behavior.

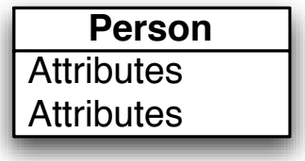
```
class Person {

   String name, address, zip, city;


   void display (... printAddress, printCity ...) {
     println(name);
     if (printAddress) { println(address); }
     if (printCity) { println(zip); println(city); }
   }


}
```

Model-View-Controller.

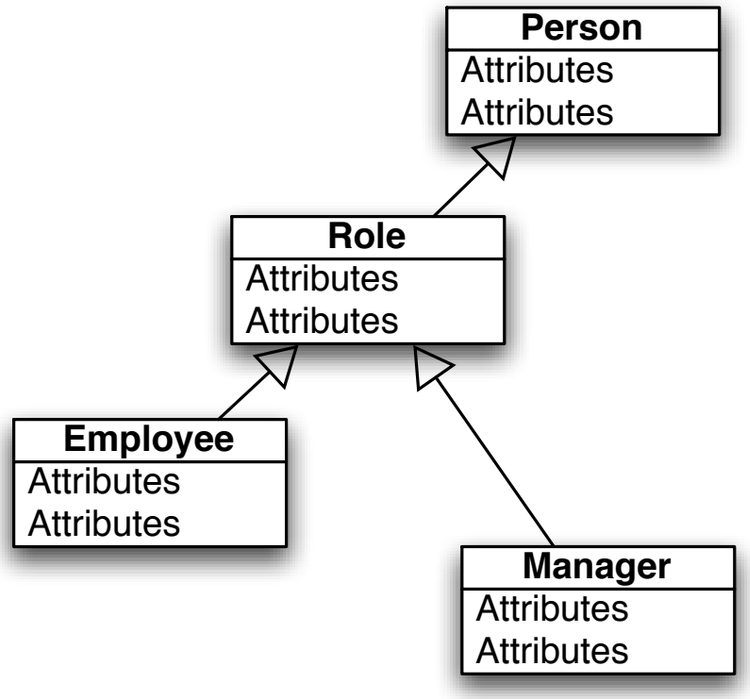
Model



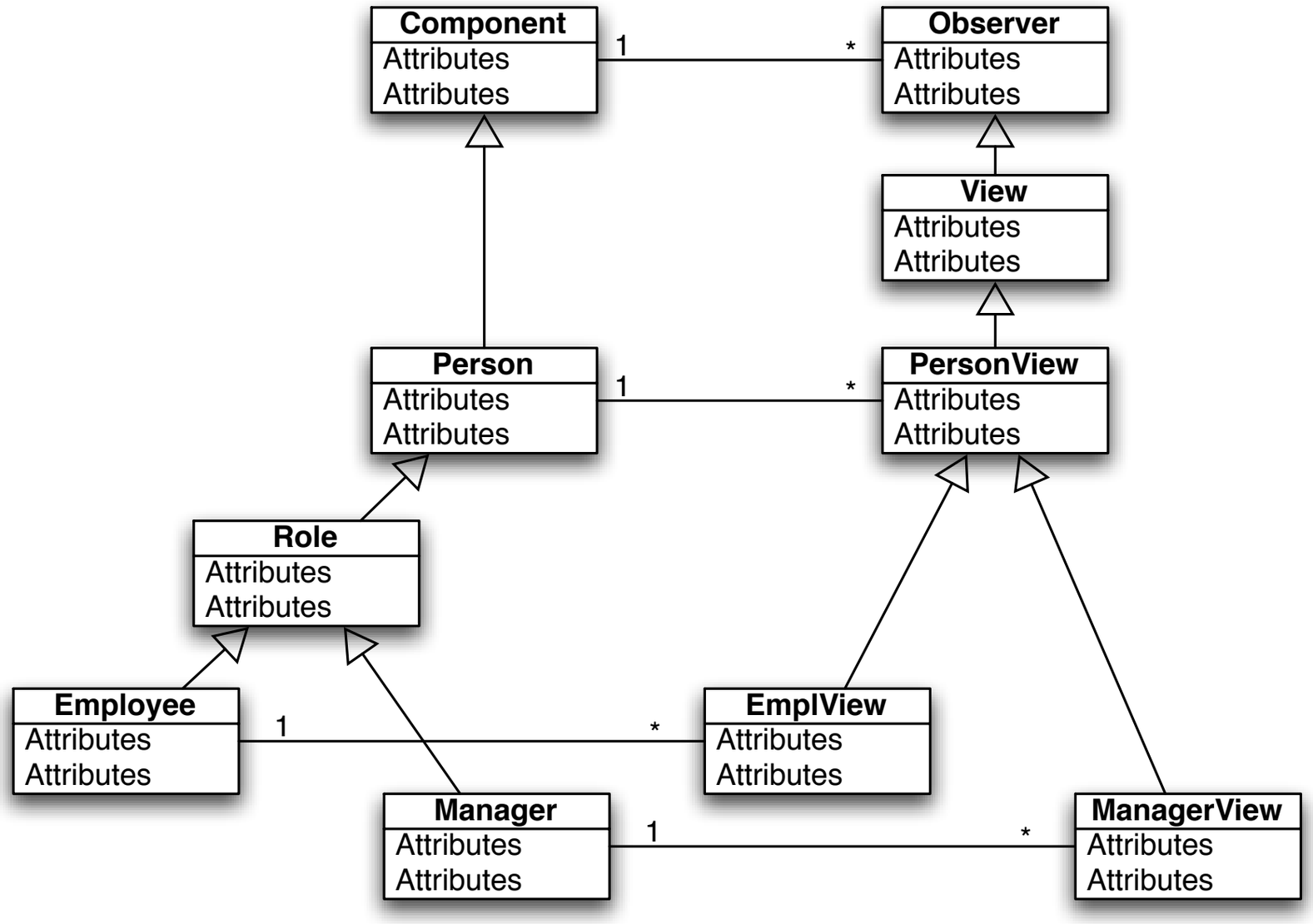
Increased Complexity.



Increased Complexity.



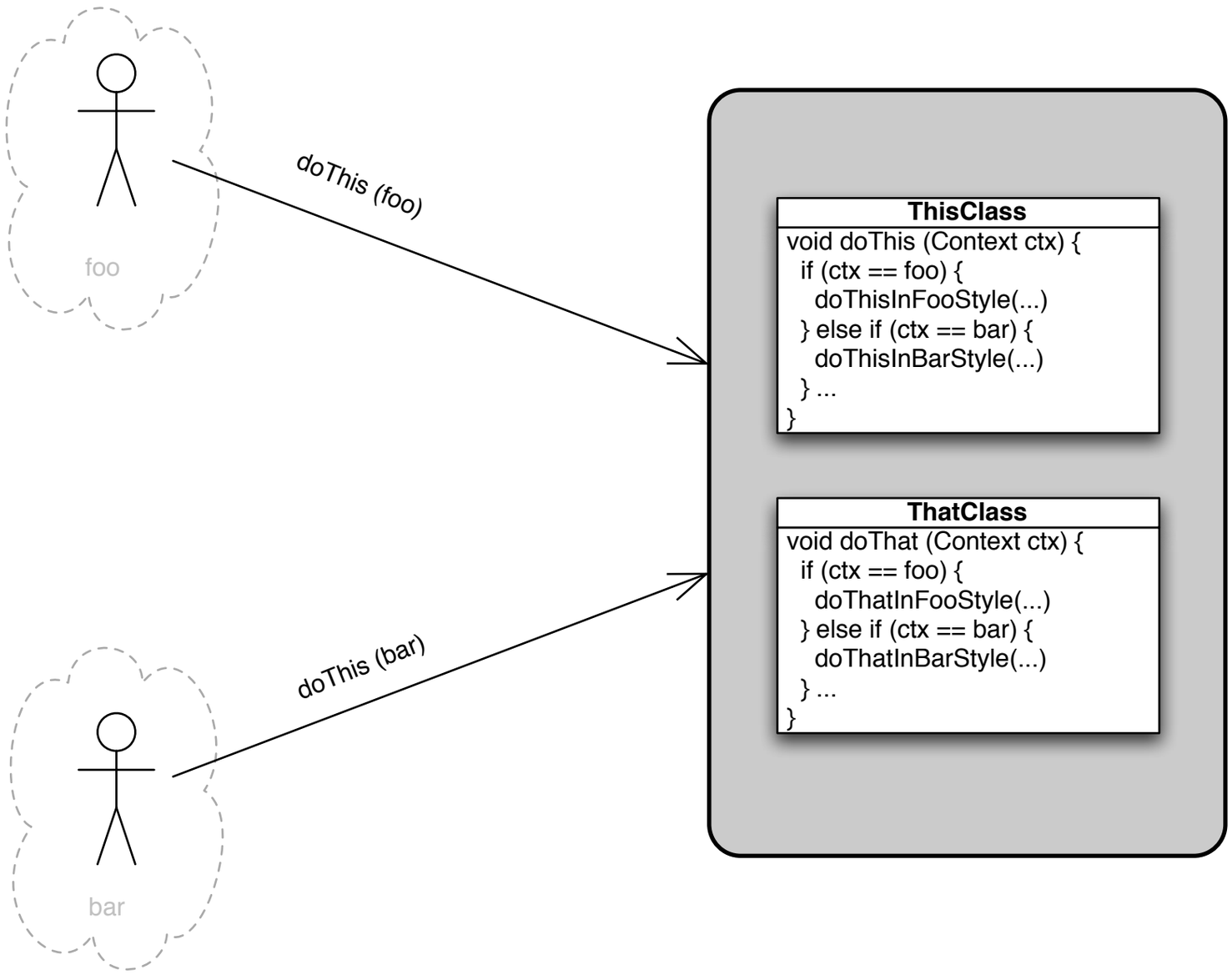
Increased Complexity.



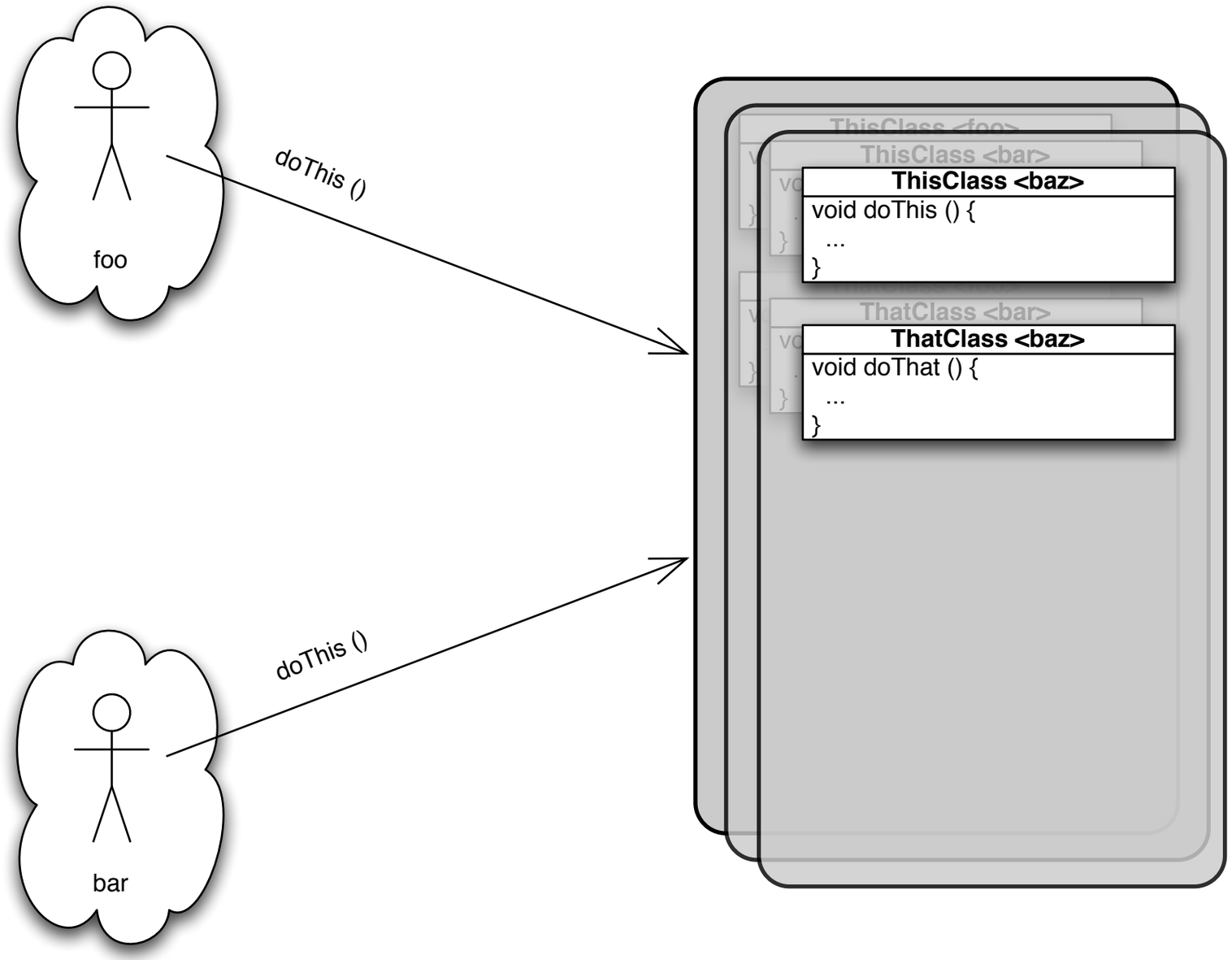
Manual Context Orientation.

- Context-dependent behavior spread over several classes!
- Secondary classes required just for plumbing!
- Basic notion of OOP broken: Objects don’t know how to behave!

Context-oriented Programming.



Context-oriented Programming.



Context-oriented Programming.

- Several language extensions in the works.
  (ContextL, ContextS, ContextJ, ...)

- Here: ContextL, based on the Common Lisp Object System (CLOS).

root layer

```
(define-layered-class person
   ((name :initarg :name
          :layered-accessor person-name)))

(define-layered-function display (object))

(define-layered-method display ((object person))
   (print (person-name object)))
```

root layer

employment layer

```
(deflayer employment)

(define-layered-class employer :in-layer employment ()
   ((name :initarg :name
          :layered-accessor employer-name)))

(define-layered-class person :in-layer employment ()
   ((employer :initarg :employer
              :layered-accessor person-employer)))

(define-layered-method display
   :in-layer employment :after ((object person))
   (display (person-employer object)))
```

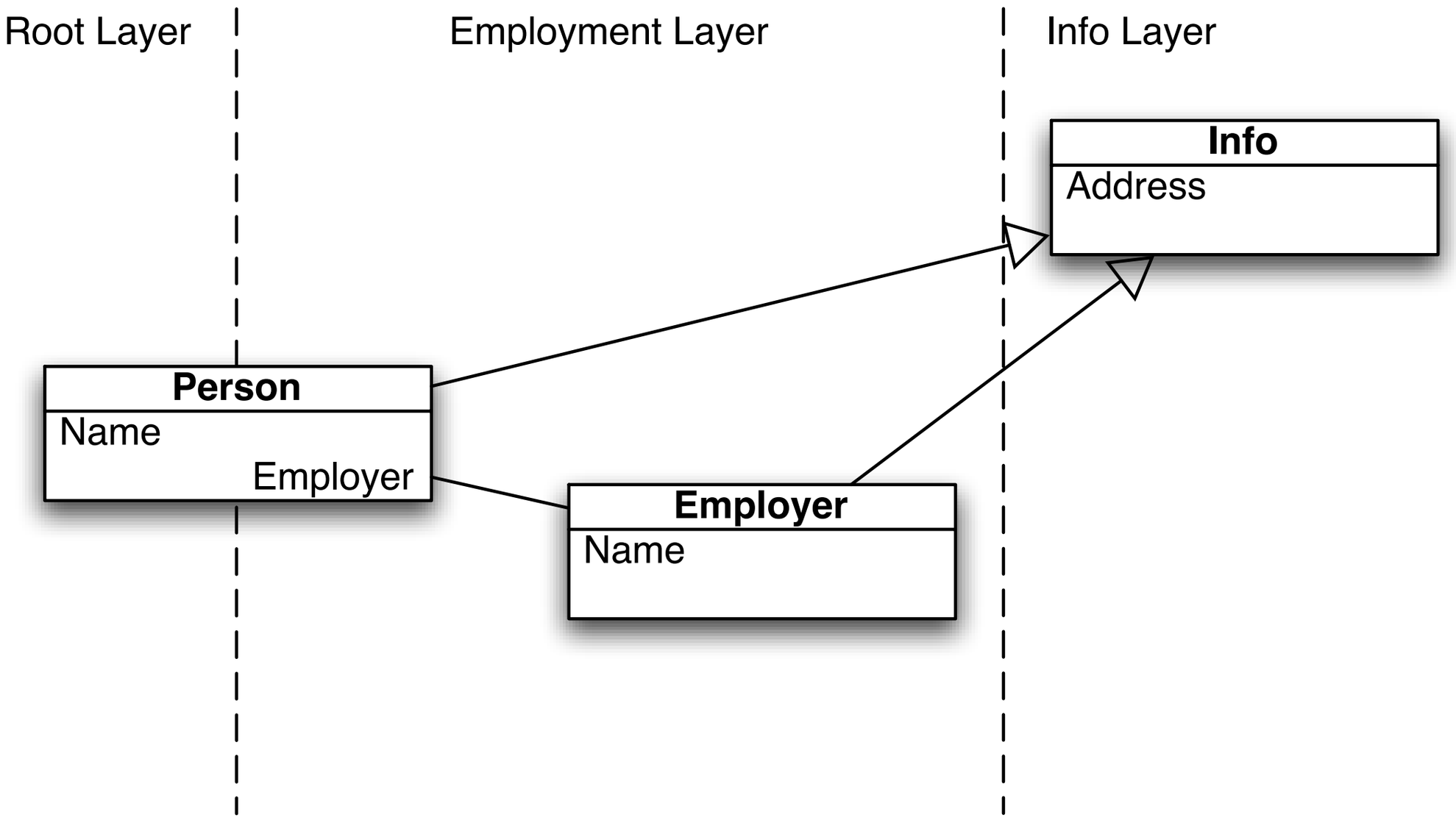
root layer

employment layer

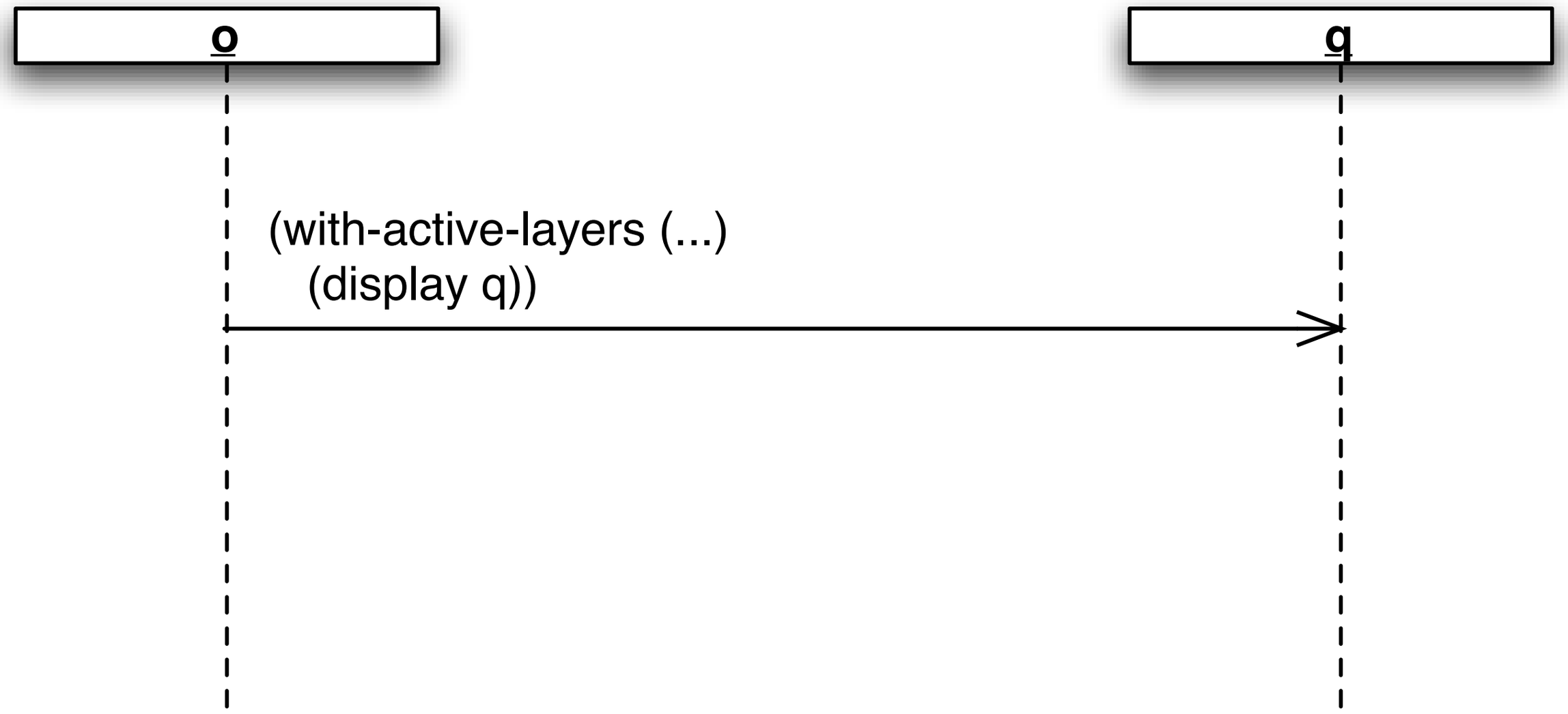
info layer

```
(deflayer info)

(define-layered-class info-mixin :in-layer info ()
   ((address :initarg :address
             :layered-accessor address)))

(define-layered-method display
   :in-layer info :after ((object info-mixin))
   (print (address object)))

(define-layered-class person :in-layer info (info-mixin)
   ())
(define-layered-class employer :in-layer info (info-mixin)
   ())
```

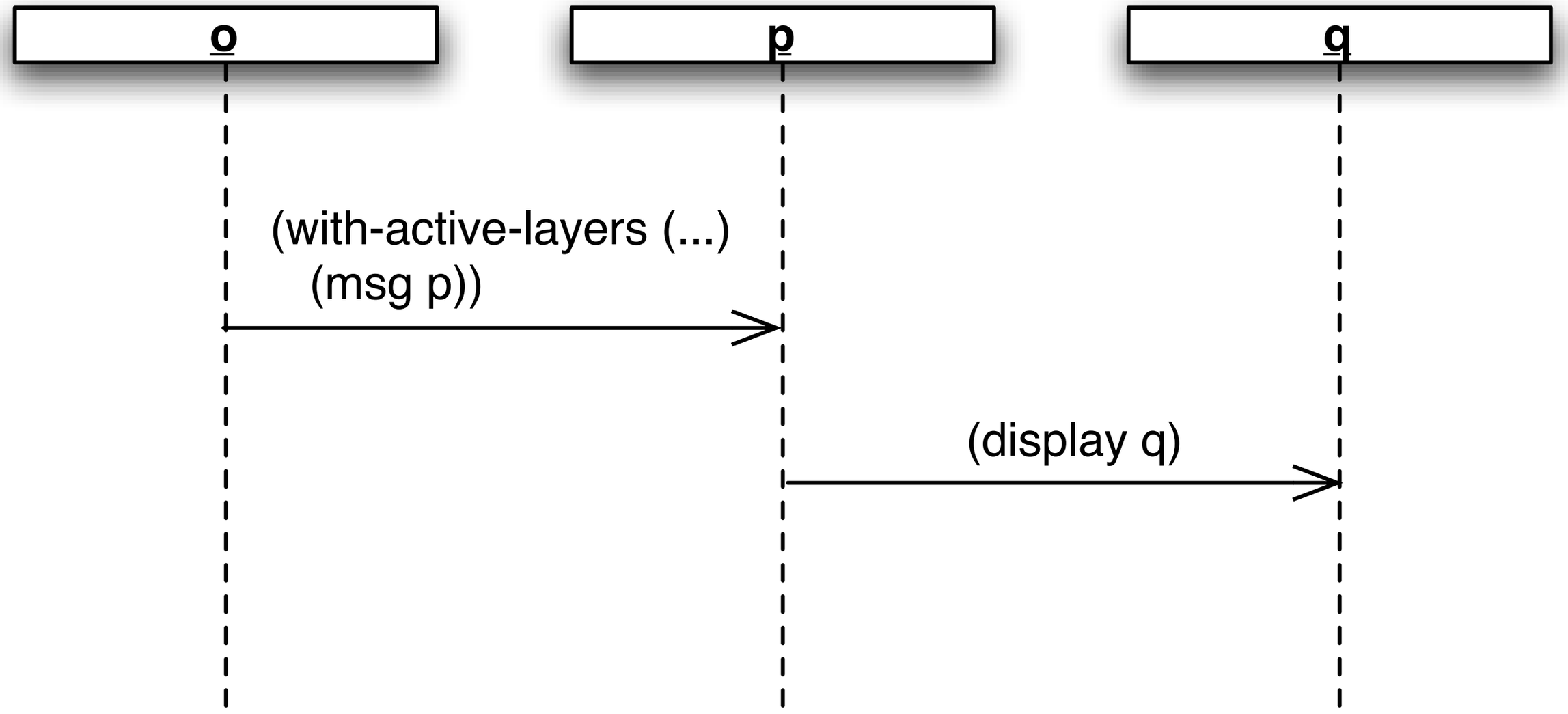
Example Classes.



Layer Activation.



Layer Activation.



Demo

Overview: Context-oriented Programming.

- Behavioral Variations: new or modified behavior.
- Layers: group related context-dependent behavioral variations.
- Activation: Layers can be activated and deactivated at runtime.
- Context: any information which is computationally accessible.
- Scoping: explicit control of effect of layer activation and deactivation.

Example uses.

- Multiple views.
- Coordination of screen updates.
- Report generation.
- Exception handling.
- Discerning of phone calls.
- Selecting billing schemes in cell phones.

The Figure Editor Example

- Class hierarchy of simple and composite graphical objects.
- Changing positions of graphical objects triggers updates on the screen.
- Typically used to motivate aspect-oriented programming.
  (“jumping aspects”)

root layer

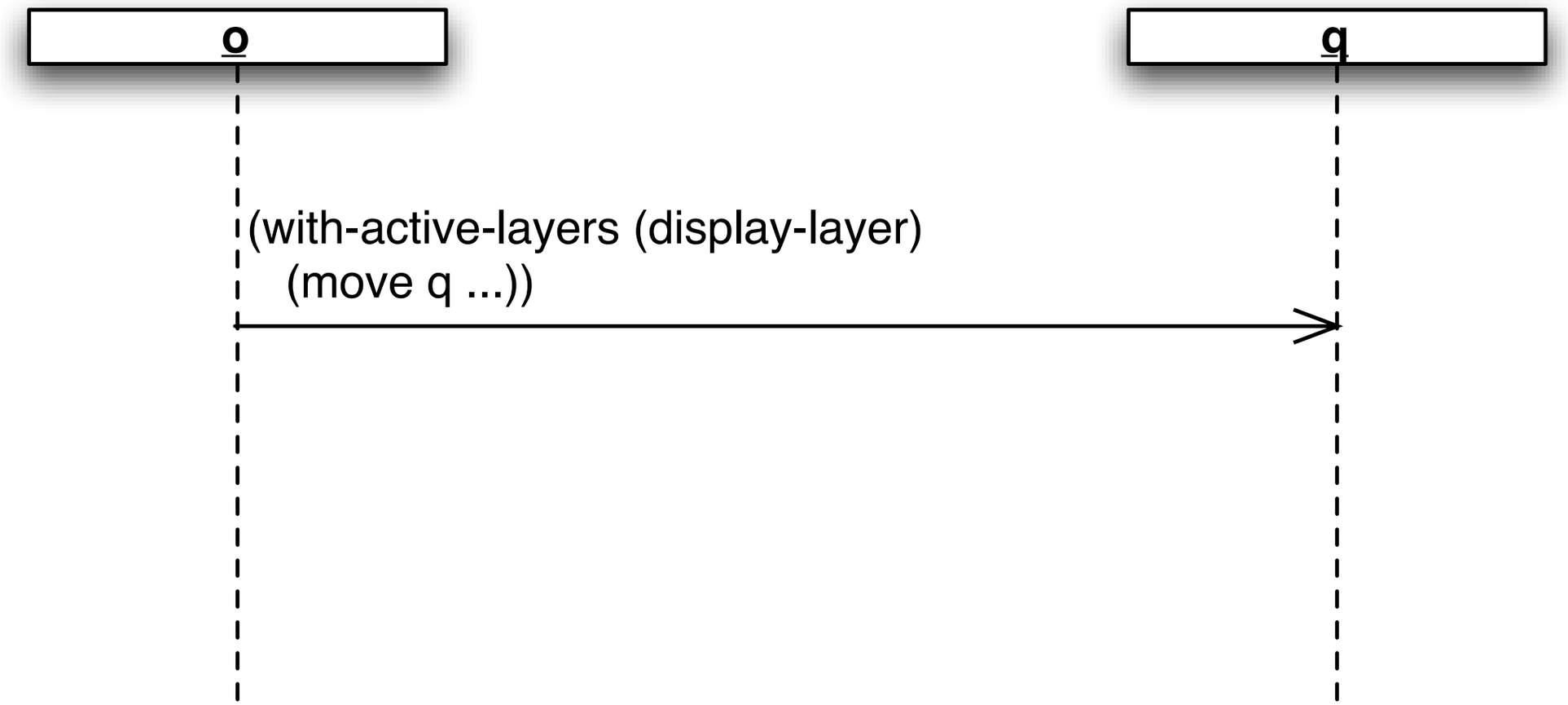
```
(define-layered-class point (figure-element)
  ((x :initarg :x :layered-accessor point-x)
   (y :initarg :y :layered-accessor point-y)))

(define-layered-method move ((elm point) dx dy)
  (incf (point-x elm) dx)
  (incf (point-y elm) dy))

(define-layered-class line (figure-element)
  ((p1 :initarg :p1 :layered-accessor line-p1)
   (p2 :initarg :p2 :layered-accessor line-p2)))

(define-layered-method move ((elm line) dx dy)
  (move (line-p1 elm) dx dy)
  (move (line-p2 elm) dx dy))
```

root layer

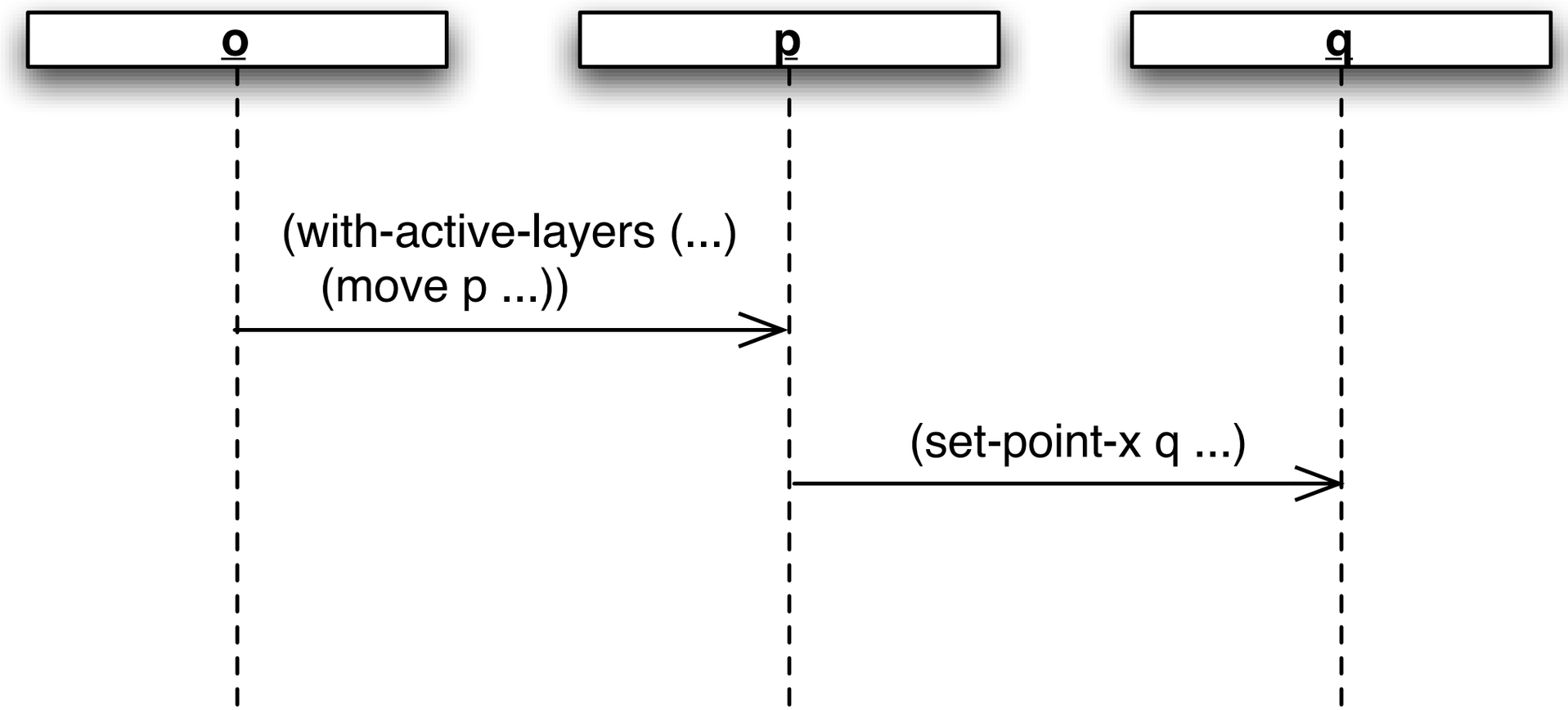
display layer

```
(deflayer display-layer)

(define-layered-method move
  :in-layer display-layer :after
  ((elm figure-element) dx dy)
  (update display elm))

(define-layered-method set-point-x
  :in-layer display-layer :after
  ((elm point) new-x)
  (update display elm))

... same for set-point-y, set-line-p1, set-line-p2 ...
```

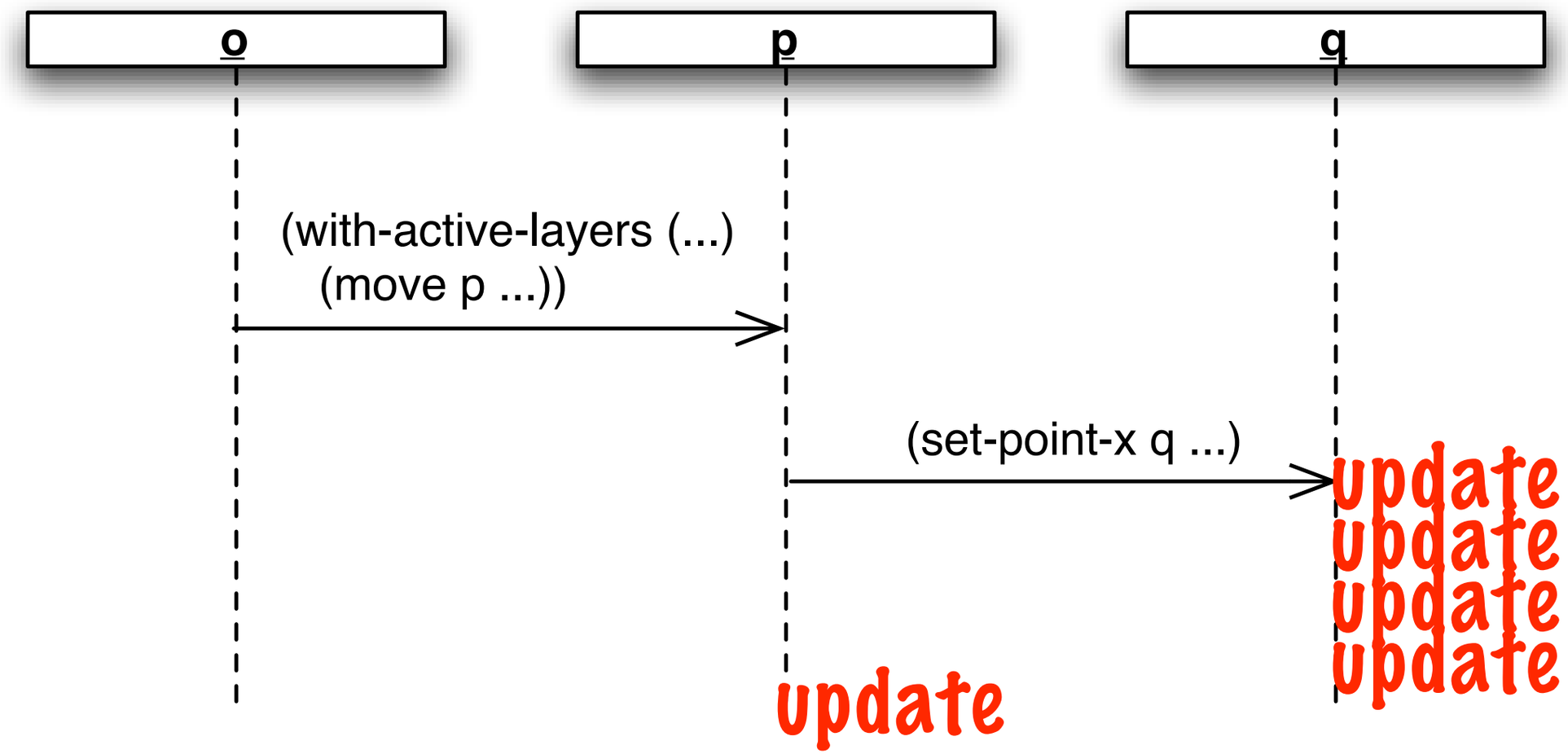
Layer Activation.



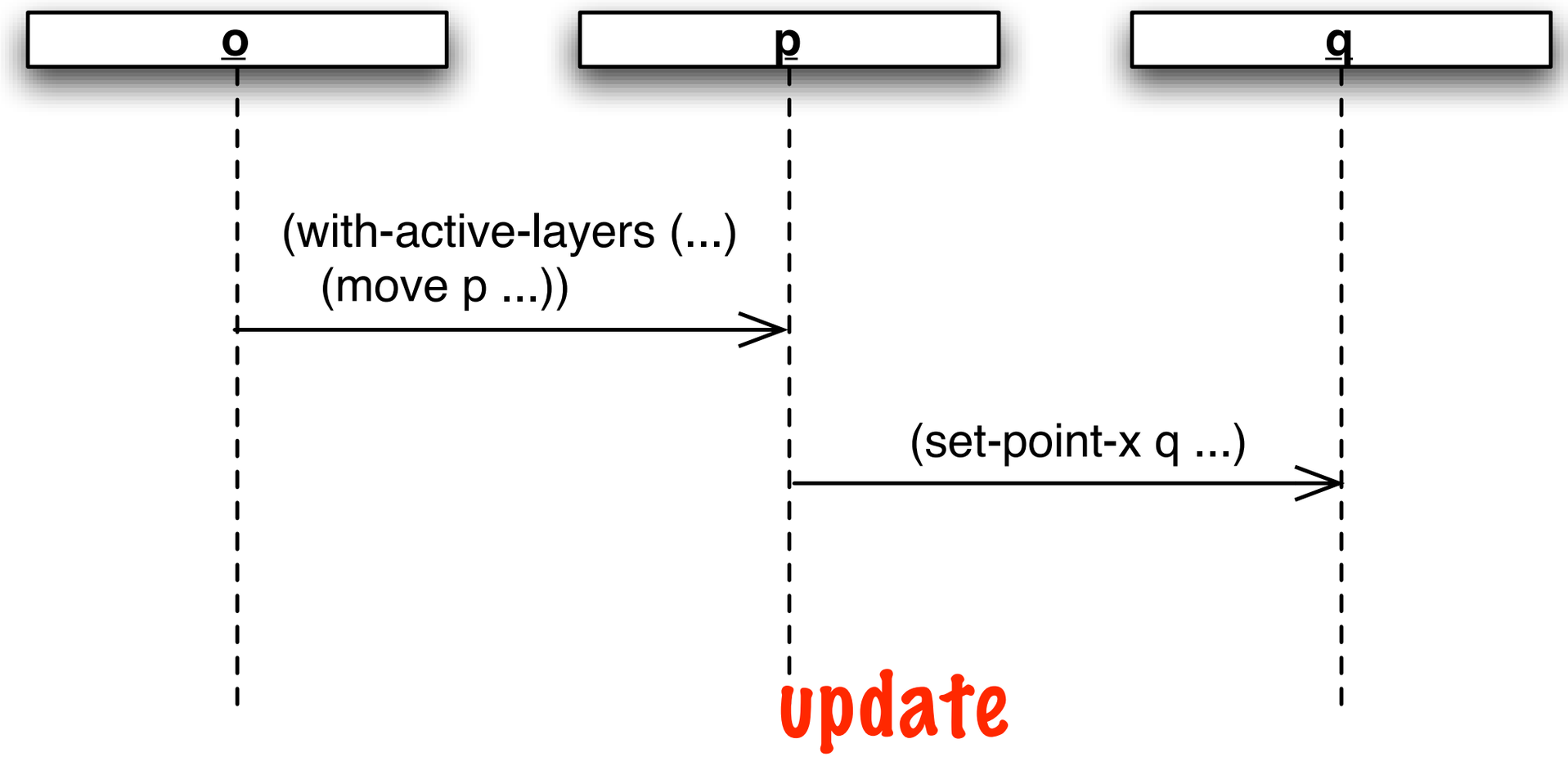
Layer Activation.



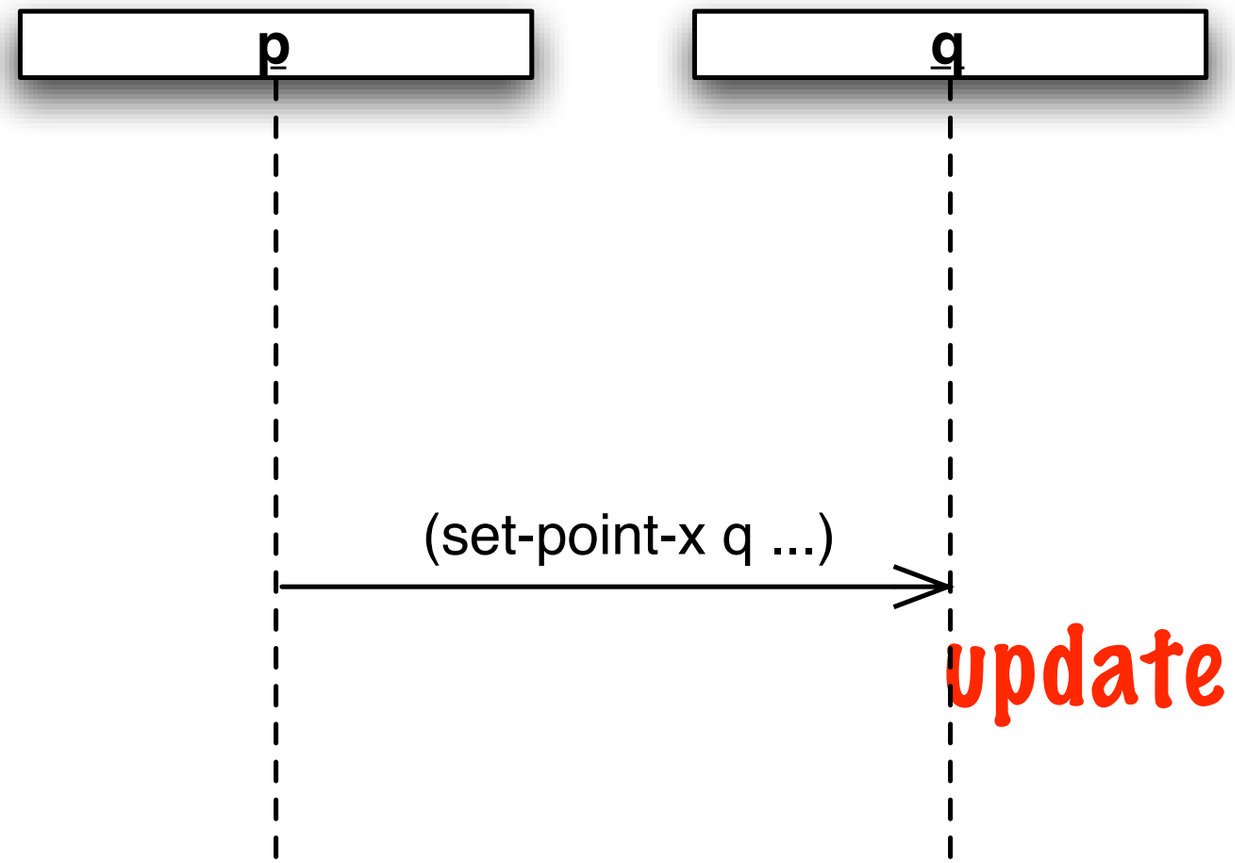
When to update?



When to update?



When to update?



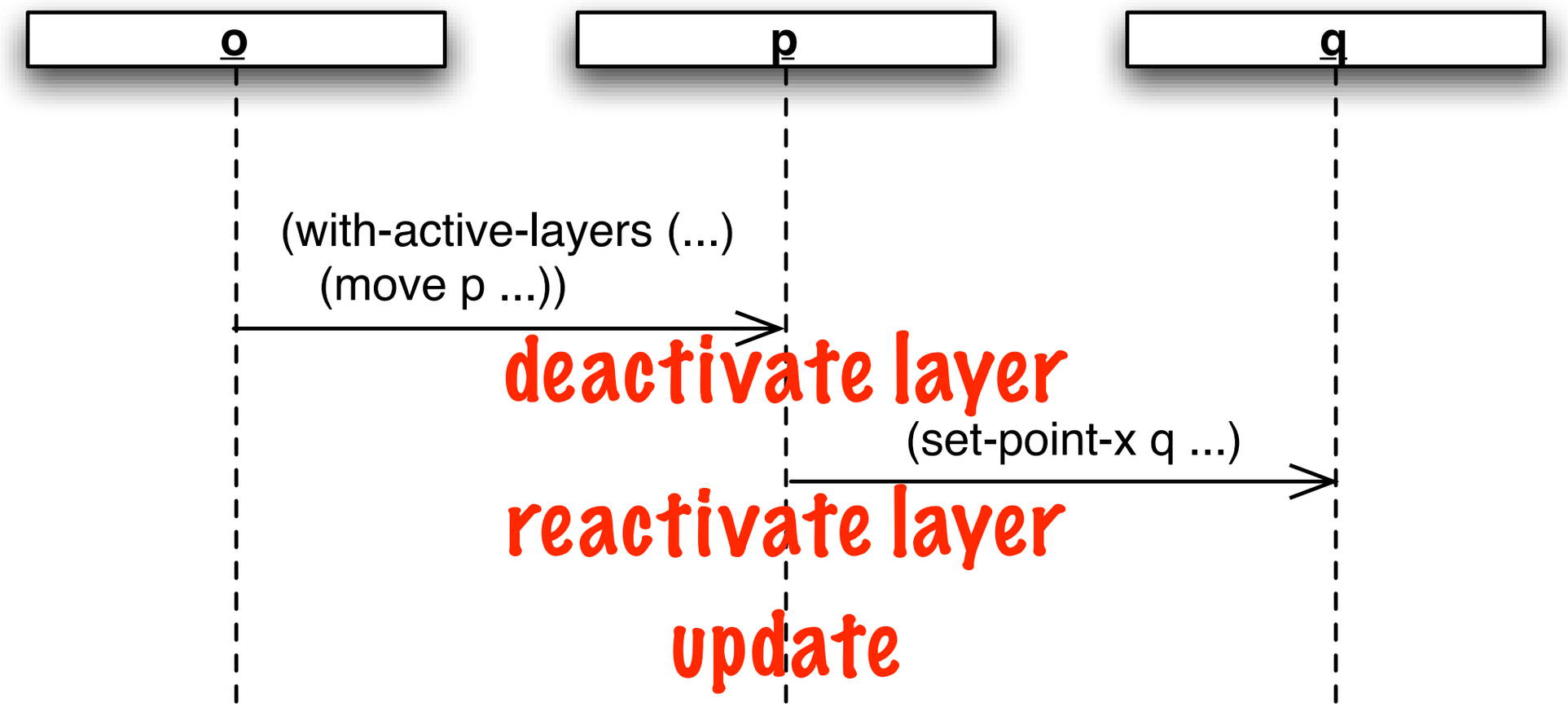
only top-level moves

DisplayUpdating v4

```
aspect DisplayUpdating {

  pointcut move(FigureElement fe):
    target(fe) &&
    (call(void FigureElement.moveBy(int, int)) ||
     call(void Line.setP1(Point))              ||
     call(void Line.setP2(Point))              ||
     call(void Point.setX(int))                ||
     call(void Point.setY(int)));

  pointcut topLevelMove(FigureElement fe):
    move(fe) && !cflowbelow(move(FigureElement));

  after(FigureElement fe) returning: topLevelMove(fe) {
    Display.update(fe);
  }
}
```

aspectj.org

When to update depends on context!



root layer

display layer

```
(deflayer display-layer)

(define-layered-method move
  :in-layer display-layer :around
  ((elm figure-element) dx dy)
  (with-inactive-layers (display-layer)
     (call-next-method))
  (update display elm))
```

... same for set-point-x, set-point-y, set-line-p1, set-line-p2 ...

root layer

display layer

```
(deflayer display-layer)

(defun call-and-update (change-function object)
  (with-inactive-layers (display-layer)
    (funcall change-function))))
  (update display object))

(define-layered-method move
  :in-layer display-layer :around
  ((elm figure-element) dx dy)
  (call-and-update (function call-next-method) elm))

(define-layered-method layered-slot-set
  :in-layer display-layer :around
  ((elm figure-element) writer)
  (call-and-update writer elm))
```

Dynamically scoped layer activation.

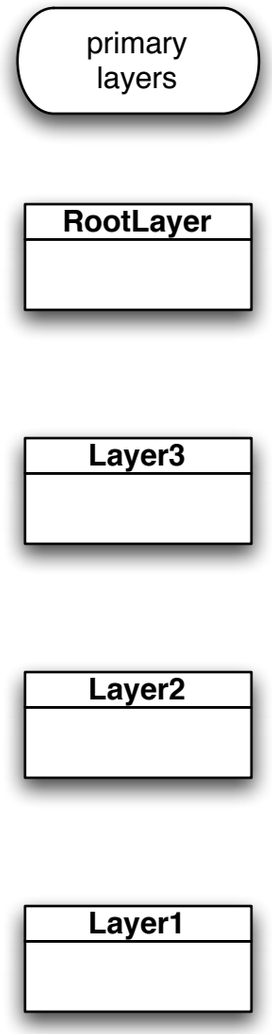
- with-active-layers
    - activates layers for the current thread.
    - does not interfere with other threads.
    - automatically deactivates on return.
- with-inactive-layers
    - deactivates layers for the current thread.
    - ...

Challenge.

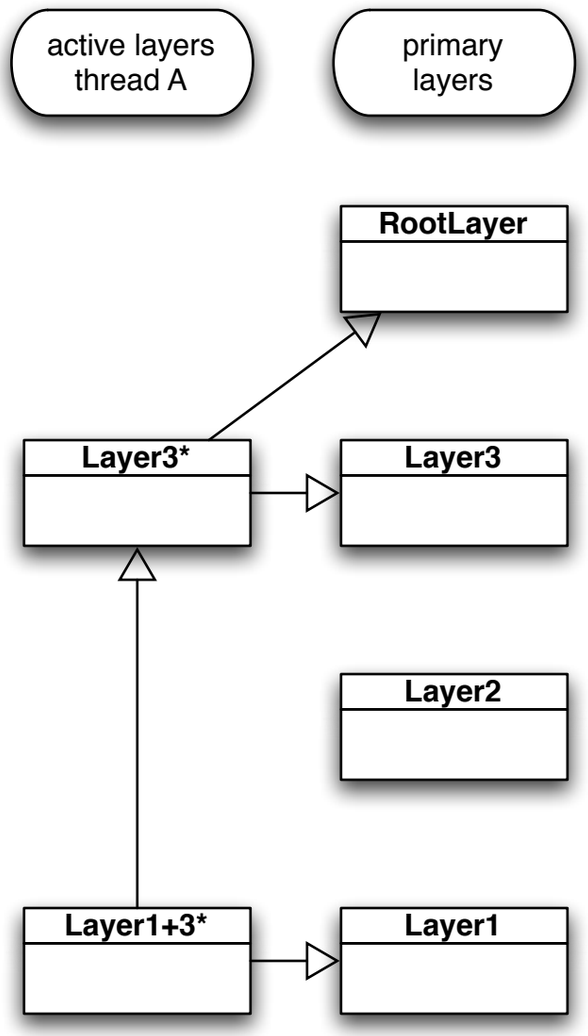
- Such examples require repeated layer activation and deactivation.

- Can this be implemented efficiently?

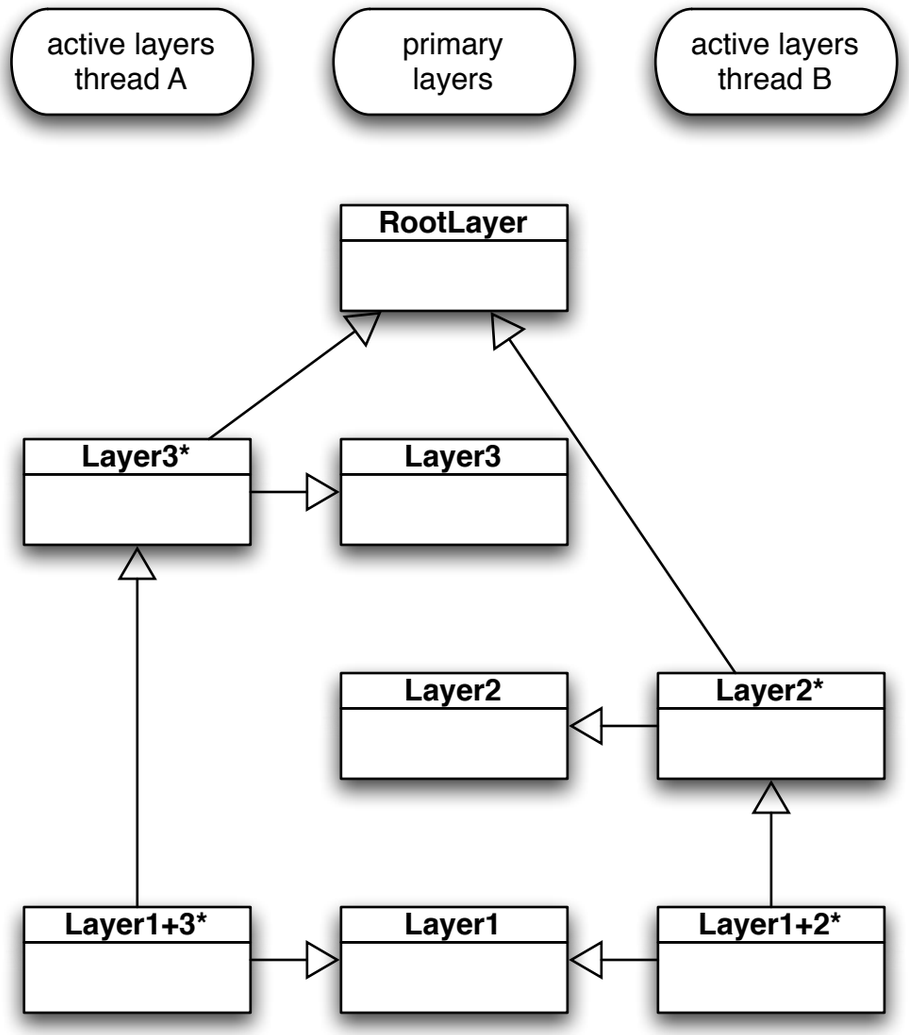
Implementation: Layers represented as classes.



Implementation: Layers represented as classes.



Implementation: Layers represented as classes.



Layers passed via another implicit argument.

- object.message(x, y, z) => object.message(object, x, y, z)
- (move elm x y) => (move layers elm x y)
- Methods are dispatched on layers, and possibly on further arguments.

Implementation: Key ingredients.

- Layer combinations via multiple inheritance.
- Layered dispatch via multiple dispatch.
- Efficient caches for layers (in ContextL).
- Efficient method dispatch (in CLOS).

Demo

Benchmark results.

Implementation	**Platform**	**Without Layers**	**With Layers**	**Overhead**
Allegro CL 7.0	Mac OS X	2.292 secs	2.540 secs	10.82% slower
CMUCL 19b	Mac OS X	0.7812 secs	0.7361 secs	6.13% *faster*
LispWorks 4.4	Mac OS X	3.0928 secs	3.1768 secs	2.72% slower
MCL 5.1	Mac OS X	2.3506 secs	2.6412 secs	12.36% slower
OpenMCL 0.14.3	Mac OS X	2.2448 secs	2.5066 secs	11.66% slower
SBCL 0.9.4	Mac OS X	0.8363 secs	0.7795 secs	7.29% *faster*
CMUCL 19a	Linux x86	0.76 secs	0.836 secs	10% slower
SBCL 0.9.4	Linux x86	0.5684 secs	0.638 secs	12.24% slower

Layer dependencies.

- start-phone-call and end-phone-call as layered functions.

- (deflayer phone-tariff)

  (define-layered-method start-phone-call :in-layer phone-tariff :after (number)
    ... record start time ...)

  (define-layered-method end-phone-call :in-layer phone-tariff :after ()
    ... record end time & determine cost ...)

- What if there are several alternative phone tariffs?

Layer inheritance.

- (deflayer phone-tariff)

  (define-layered-method start-phone-call :in-layer phone-tariff :after (number)
    ... record start time ...)

- (deflayer phone-tariff-a (phone-tariff))
  (deflayer phone-tariff-b (phone-tariff))

- ...allows sharing of common behavior.
  But this is not enough: Tariff a and b should be mutually exclusive!

Layers as metaobjects.

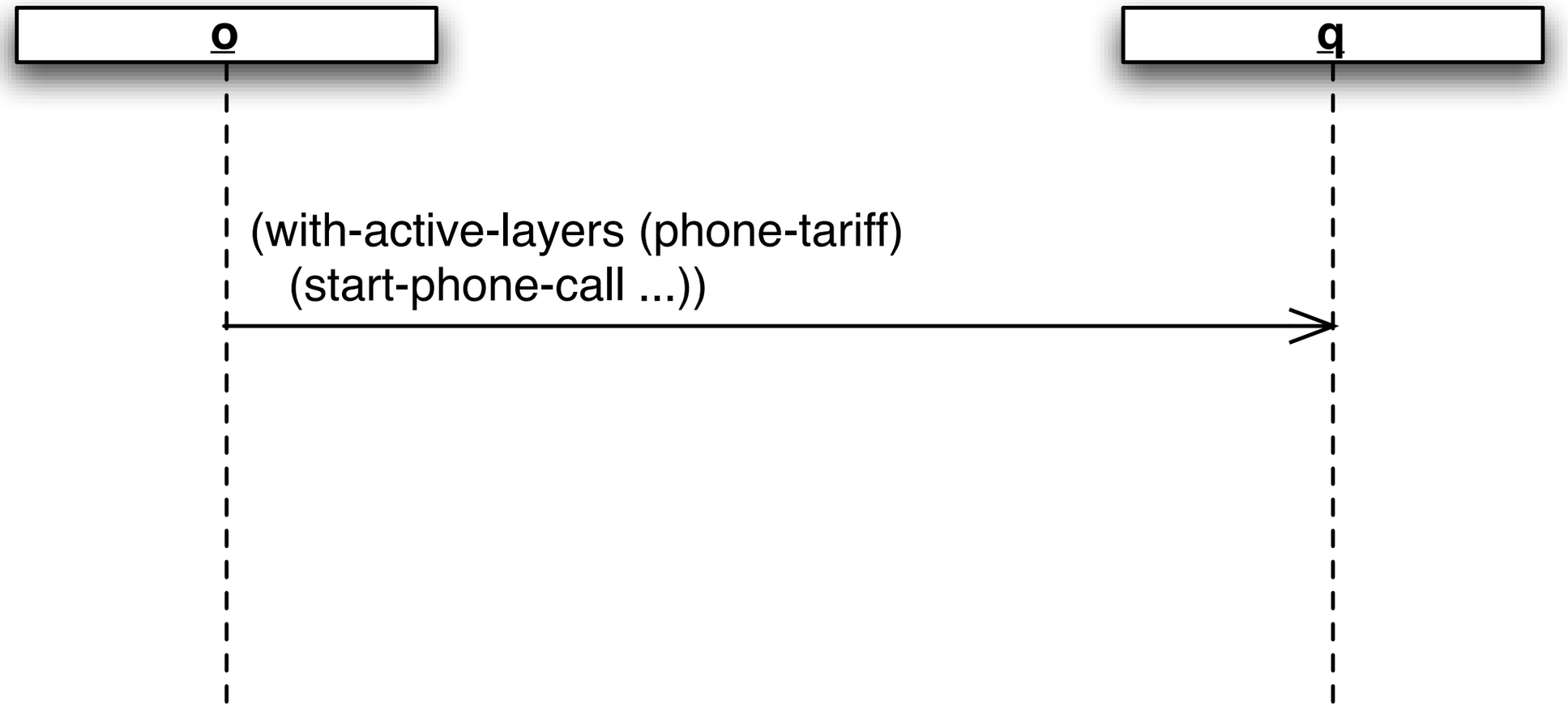
- Reflection = introspection and intercession.
- Metaobject protocols = OOP-style organization of the reflective API.
- Here: Layers are instances of layer metaobject classes.

Intercession of layer activation.

- (defclass tariff-base-layer-class (standard-layer-class)
  ())

  (deflayer phone-tariff ()
  ()
  (:metaclass tariff-base-layer-class))

Intercession of layer activation.



- Internally calls (adjoin-layer-using-class <phone-tariff> ...) and uses the result as the set of new active layers.

Intercession of layer activation.

- (defclass tariff-base-layer-class (standard-layer-class)
  ())

  (deflayer phone-tariff ()
  ()
  (:metaclass tariff-base-layer-class))

- (define-layered-method adjoin-layer-using-class
  ((layer tariff-base-layer-class) active-layers)
  (if (layer-active-p ‘phone-tariff active-layers)
  active-layers
  (let ((tariff (ask-user “Select tariff ...”)))
  (adjoin-layer tariff active-layers))))

Layer dependencies.

- Conditional or unconditional blocking of layer activations.

- Inclusion dependencies:
  Activation of a layer requires activation of another.

- Exclusion dependencies:
  Activation of a layer requires deactivation of another.

- Also: dependencies on layer deactivation.

Efficiency.

- Goal: Only incur a cost when necessary.

- (define-layered-method adjoin-layer-using-class
  :in-layer block-managed-layers
  ((layer managed-layer-class) active-layers)
  (values active-layers t))

Benchmark results.

- Without reflective layer activation (JMLC '06).

Implementation	**Platform**	**Without Layers**	**With Layers**	**Overhead**
Allegro CL 7.0	Mac OS X	2.292 secs	2.540 secs	10.82% slower
CMUCL 19b	Mac OS X	0.7812 secs	0.7361 secs	6.13% *faster*
LispWorks 4.4	Mac OS X	3.0928 secs	3.1768 secs	2.72% slower
MCL 5.1	Mac OS X	2.3506 secs	2.6412 secs	12.36% slower
OpenMCL 0.14.3	Mac OS X	2.2448 secs	2.5066 secs	11.66% slower
SBCL 0.9.4	Mac OS X	0.8363 secs	0.7795 secs	7.29% *faster*
CMUCL 19a	Linux x86	0.76 secs	0.836 secs	10% slower
SBCL 0.9.4	Linux x86	0.5684 secs	0.638 secs	12.24% slower

- With reflective layer activation (SAC PSC '07).

Implementation	**Without Layers**	**With Layers**	**Overhead**
Allegro CL 8.0	2.544 secs	2.650 secs	4.17% slower
CMUCL 19c	0.77 secs	0.744 secs	3.49% *faster*
LispWorks 4.4.6	3.128 secs	3.2374 secs	3.50% slower
MCL 5.1	2.187 secs	2.4358 secs	11.38% slower
OpenMCL 1.0	2.3788 secs	2.5938 secs	9.04% slower
SBCL 0.9.16	0.9138 secs	0.8708 secs	4.94% *faster*

Summary.

- Context-oriented Programming provides
  - layers with partial classes and methods
  - that can be freely selected and combined
  - without interfering with other contexts.

Summary.

- COP is independent of the organization of the source code.
    - Essential contribution is layer activation / deactivation at runtime.
- It can be beneficial to activate / deactivate layers anywhere.
- COP is compatible with a higher-order reflective programming style.

Summary.

- Some examples require repeated activation / deactivation of layers.
  (For example, the figure editor.)

- Efficient implementation
  - multiple inheritance & multiple dispatch
  - efficient caches
- Should also be doable in Java-style languages

ContextL Summary.

- Layers
- Layered classes
  - Layered slots, special slots
- Layered functions, layered accessors
- Dynamically scoped layer activation / deactivation

ContextL.

- Available for 7 major Common Lisp implementations:
  Allegro, CLisp, CMUCL, LispWorks, MCL, OpenMCL, SBCL.

  - This means: BeOS, FreeBSD, HP-UX, IBM AIX, IRIX, Linux x86, Linux PowerPC, Mac OS X, NetBSD, NeXTstep, OpenBSD, Solaris SPARC, Tru64, Windows, ...

- Implemented using the CLOS MOP.

- Apparently no serious runtime overhead!

- Source code with MIT/BSD-style license at
  http://common-lisp.net/project/closer/

Major achievements so far...

- **Language Construct for Context-oriented Programming - An Overview of ContextL**
  Dynamic Languages Symposium 2005 (with Robert Hirschfeld)

- **Efficient Layer Activation for Switching Context-dependent Behavior**
  Joint Modular Languages Conference 2006 (with Robert Hirschfeld & Wolfgang De Meuter)

- **Reflective Layer Activation in ContextL**
  ACM Symposium on Applied Computing 2007 (with Robert Hirschfeld)

- **The Context-Dependent Role Model**
  International Conference on Distributed Applications and Interoperable Systems 2007 (Jorge Vallejos et al.)

- **Context-Oriented Domain Analysis**
  International and Interdisciplinary Conference on Modeling and Using Context 2007 (Brecht Desmet et al.)

- **Context-oriented Programming**
  Journal of Object Technology, March/April 2008 (with Robert Hirschfeld & Oscar Nierstrasz)

Thank you.