Metaprogramming & Reflection

Pascal Costanza

Example: Handling scarce resources.

```
try {
   FileInputStream in = new FileInputStream(filename);
   doSomething(in);
} finally {
   in.close();
}
```

Example: Handling scarce resources.

```
FileInputStream in;
try {
   in = new FileInputStream(filename);
   doSomething(in);
} finally {
   in.close();
}
```

Example: Handling scarce resources.

```
FileInputStream in;
try {
   in = new FileInputStream(filename);
   doSomething(in);
} finally {
   in.close();
}
```

```
(let (in)
  (unwind-protect
    (progn
      (setf in (open filename))
      (do-something in))
    (close in)))
```

Example: Handling scarce resources.

```java
FileInputStream in;
try {
   in = new FileInputStream(filename);
   doSomething(in);
} finally {
   in.close();
}
```

```lisp
(let (in)
  (unwind-protect
    (progn
      (setf in (open filename))
      (do-something in))
    (close in)))
```

```lisp
(with-open-file (in filename)
   (do-something in))
```

Example: Handling scarce resources.

```java
FileInputStream in;
try {
   in = new FileInputStream(filename);
   doSomething(in);
} finally {
   in.close();
}
```

```lisp
(let (in)
   (unwind-protect
      (progn
         (setf in (open filename))
         (do-something in))
      (close in)))
```

```lisp
(with-open-file (in filename)
   (do-something in))
```

```lisp
(defmacro with-open-file ((var filename) &body body)
   `(let (,var)
      (unwind-protect
           (progn (setf ,var (open ,filename))
                  ,@body)
        (close ,var))))
```

Why is metaprogramming important?

- Productivity!!!
  - object-relational mappings
  - higher-order functions
  - map/reduce
  - program transformations
  - domain-specific languages
  - etc., etc.

Why is metaprogramming important?

- Productivity!!!
  - Don’t write programs yourself,
    but make your computer write programs for you!

What is metaprogramming?

- Every program has a domain.
  - For example, a financial application is about bank accounts, money, transfers, etc.
- A program with other programs as its domain is a meta-program.
  - Interpreters, debuggers, profilers, code coverage tools, compilers, program generators, etc.
- A program with itself as (part of) its domain is a reflective program.

Forms of metaprogramming.

	Lisp	Java
compile-time load-time	macros read macros	annotation processing bytecode transformation
runtime	higher-order functions metaobject protocols	inner classes dynamic proxies

Example: Hibernate with XML.

- public class Person {

```
    private Long id; // primary key in database
    protected String name;

    public Long getId() {return id;}
    public void setId(Long id) {this.id = id;}

    public String getName() {return name;}
    public void setName(String name) {this.name = name;}

}
```

Example: Hibernate with XML.

-
  ```
  <?xml version="1.0"?>
  <!DOCTYPE hibernate-mapping PUBLIC
          "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
          "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

  <hibernate-mapping>
     <class name="Person" table="Person" schema="PUBLIC">
          <id name="id" column="PERSON_ID">
              <generator class="increment"/>
          </id>

          <property name="name" column="NAME" />
     </class>
  </hibernate-mapping>
  ```

Example: Hibernate with Annotations.

-
  ```
  @Entity @Table(name=”Person”)
  public class Person {

     private Long id; // primary key in database
     protected String name;

     @Id @Column(name=”PERSON_ID”)
     @GenerateValue(strategy = GenerationType.AUTO)
     public Long getId() {return id;}
     public void setId(Long id) {this.id = id;}

     public String getName() {return name;}
     public void setName(String name) {this.name = name;}

  }
  ```

Important Concepts.

- Introspection: The ability to inspect a program.
- Intercession: The ability to change a program’s behavior.
- Reflective tower (more about that later).

Example: Map/Reduce.

- Mapping:
  Take a piece of code and apply it to each element of a collection of data.
  - Traditionally: Important form of iterating over data.
- Reduce:
  Take a collection of data and turn it into one result.
  - also known as “fold”
- Map/Reduce
  - Easy to parallelize, for example used at Google.

Example: Mapping.

```
class Mapper<T1, T2> {
  interface Fun<T1, T2> { public T2 call(T1 x); }

  public Vector<T2> map(Fun<T1, T2> f, Vector<T1> input) {
    Vector<T2> result = new Vector<T2>();
    for (T1 element: input) result.add(f.call(element));
    return result;
  }
  public static void main(String[] args) {
    Vector<Integer> input = new Vector<Integer>();
    input.add(1); input.add(2); input.add(3);
    System.out.println(new Mapper<Integer, Integer>().map(
      new Fun<Integer, Integer>() { public Integer call(Integer x) {return x+1;} },
      input));
  }
}
```

Example: Mapping.

```
(defun mapper (f list)
  (if (null list) ‘()
    (cons (funcall f (car list)) (mapper f (cdr list)))))

(defun test ()
  (mapper (lambda (x) (+ x 1)) (list 1 2 3)))
```

Example: Sorting.

-
  ```
  public static void main(String[] args) {
     java.util.Arrays.sort(args);
     print(args);
  }
  ```

Example: Sorting.

-
  ```
  public static void main(String[] args) {
    java.util.Arrays.sort(args);
    print(args);
  }
  ```

-
  ```
  public static void main(String[] args) {
    java.util.Array.sort(args,
      new Comparator<String> () {
        public int compare(String s1, String s2) {
          return s1.compareToIgnoreCase(s2);
        }
      }
    );
    print(args);
  }
  ```

Example: Sorting.

- (defun test (strings)
  (print (sort strings #‘string<=)))

Example: Sorting.

- (defun test (strings)
  (print (sort strings #‘string<=)))

- (defun test (strings)
  (print (sort strings #‘string-lessp)))

Observation: Program representation matters!

- In the Java examples, we have to define interfaces and classes, and have to create objects to represent computations (“pieces of code”).
    - Additional complexity because of static typing, but that’s a secondary issues.
- In the Lisp examples, lambda is a “direct” representation of computations.
- Better representations of programs make metaprogramming easier.

Necessary Ingredients.

- Good representation of programs
  + machine code?
  + bytecode
  + strings?
  + tokens?
  + ASTs
  + s-expressions
  + closures

- Decisions about phases
  + compile-time
  + load-time
  + run-time

Bytecode.

- System.out.println(i);

- getstatic java/lang/System.out:Ljava/io/PrintStream
  iload 1
  invokevirtual java/io/PrintStream.println:(I)V

- Note: symbolic information matters!

break

Reflection in Python (1).

```
class Person:
   def display(self):
         print self.name

p = Person()
p.name = “Bill Gates”

p.display() => Bill Gates

p.__dict__ => {'name': 'Bill Gates'}

p.__class__ => <class __main__.Person at 0x9bcc0>

p.__class__.__dict__ => {'__module__': '__main__',
                         'display': <function f at 0x978f0>,
                         '__doc__': None}
```

Reflection in Python (2).

```
x = 42
eval(”x + x”) => 84

def f():
   x = 2
   print eval(”x + x”)

f() => 4

x => 42
```

An example: Aspect-Oriented Programming.

```
class c:
    def foo(self):
         print "foo"

def adv(self, *args, **keyw):
    print "before"
    return self.__proceed(*args,**keyw)

wrap_around( c.foo, adv )

c().foo() => before
             foo
```

(A Light-weight Approach to Aspect-Oriented Programming in Python, Antii Kervinen, http://www.cs.tut.fi/~ask/aspects/index.shtml)

Reflection in “Early” Lisp (1).

```
(define display-person
   (lambda (self)
      (print (get self ‘name))))

(define p ‘(name “Bill Gates”))

(display-person p) => Bill Gates

p => (name “Bill Gates”)

(symbol-plist ‘display-person) => (... expr (lambda (self) ...) ...)
```

Reflection in “Early” Lisp (2).

```
(define x 42)
(eval ‘(+ x x)) => 84

(define f
   (lambda ()
      (let ((x 2))
        (print (eval ‘(+ x x))))))

(f) => 4

x => 42
```

An early example: The PILOT system.

“There are two ways a user can modify programs in this subjective model of programming: he can modify the interface between procedures, or he can modify the procedure itself. [...] Modifying the interface is called advising. Modifying a procedure itself is called editing. [...]
Advising consists of inserting new procedures at any or all of the entry or exit points to a particular procedure (or class of procedures). [...]
The principal advantage of advising is that the user need not be concerned about the details of the actual changes in his program, nor the internal representation of advice. He can treat the procedure to be advised as a unit, a single block, and make changes to it without concern for the particulars of this block. This may be contrasted with editing in which the programer must be cognizant of the internal structure of the procedure.”

(PILOT: A Step Toward Man-Computer Symbiosis,
Warren Teitelman, 1966)

An early example: The PILOT system.

“The user can affect the flow of control - from advice to procedure to advice - by returning a non-NIL value from a piece of advice. [...] [For example], the user can indicate that the original procedure is to be bypassed entirely. [...] one can specify advice for any recursive set of functions. For example, to determine whether or not the procedure in question has called itself more than twice, one need merely search the HISTORY list. [...] The HISTORY list is a globally available variable which contains information regarding computation in progress.”

(PILOT: A Step Toward Man-Computer Symbiosis,
Warren Teitelman, 1966)

An early example: The PILOT system.

- The basic ingredients of metaprogramming are already there.
  - Introspection:
    The HISTORY list can be inspected.
  - Intercession:
    Advice can return non-NIL values to influence the meta-level behavior.
- Secondary ingredient:
  - “Obliviousness:”
    If you want to change a program’s behavior,
    you can either change the program or change its interpreter.

Why Lisp?

“LISP differs from most programming languages in three important ways. The first way is in the nature of the data. In the LISP language, all data are in the form of symbolic expressions usually referred to as S-expressions, of indefinite length, and which have a branching tree-type of structure, so that significant subexpressions can be readily isolated. [...] The second distinction is that the LISP language is the source language itself which specifies in what way the S-expressions are to be processed. Third, LISP can interpret and execute programs written in the form of S-expressions. Thus, like machine language, and unlike most other higher level languages, it can be used to generate programs for further execution.”

(LISP 1.5 Programmer's Manual,1962)

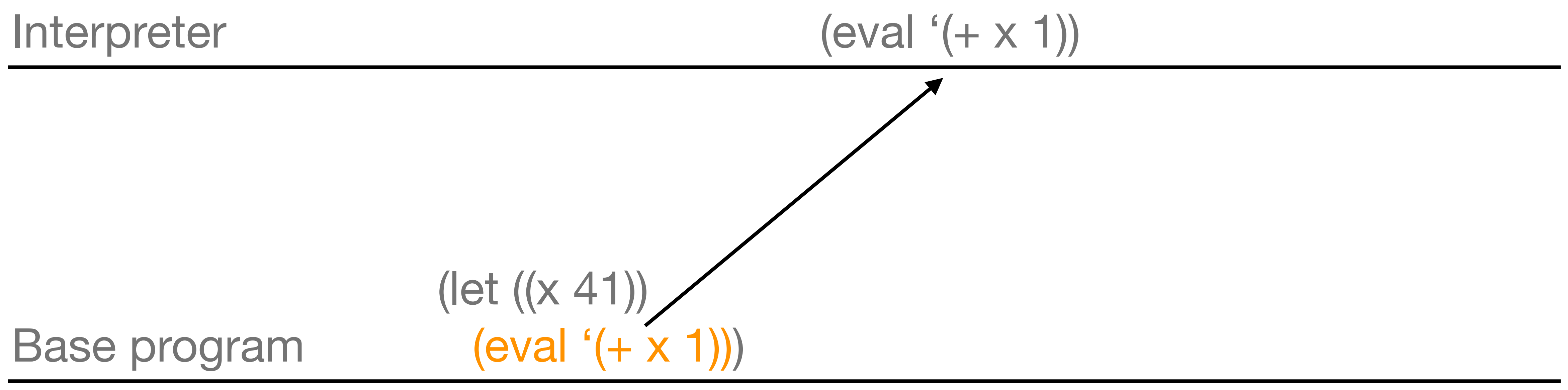
Programs = Data.

- (+ 3 4) is a program.
- (quote (+ 3 4)) is a list with three elements.
- Typically abbreviated as ‘(+ 3 4).
- So (+ 3 4) <=> (eval ‘(+ 3 4)).

Metacircular Interpreter.

-
  ```
  (defun eval (form)
     (cond ((symbolp form) (lookup form))
               ((atom form)       form)
               ((consp form)
                (let ((head (first form)))
                  (cond ((eq head ‘quote)    (second form))
                            ((eq head ‘atom)     (atom (eval (second form))))
                            ((eq head ‘eq)         (eq (eval (second form)) (eval (third form))))
                            ((eq head ‘car)        (car (eval (second form))))
                            ((eq head ‘cdr)        (cdr (eval (second form))))
                            ((eq head ‘cond)     (eval-cond (rest form)))
                            ((eq head ‘lambda) (eval-lambda (rest form)))
                            (t (let ((values (mapcar #‘eval form)))
                                 (apply (first values) (rest values)))))))))
  ```

Metacircular Interpreter.



Functions that don’t evaluate their arguments.

```
(define f (lambda (x) x))
(f (+ 2 2)) => 4

(define f (nlambda (x) x))
(f (+ 2 2)) => ((+ 2 2))

(define f (nlambda (x) (eval (first x))))
(f (+ 2 2)) => 4
```

Example: if statement in terms of cond.

```
(define if (nlambda (args)
                  (cond ((eval (first args)) (eval (second args)))
                           (t                          (eval (third args))))))


(let ((test 42))
  (if (number? test) ‘yes ‘no)) => yes
```

Problem: Dynamic Scoping.

```
(define if (nlambda (args)
                (cond ((eval (first args)) (eval (second args)))
                      (t                   (eval (third args))))))


(let ((test 42))
  (if (number? test) ‘yes ‘no)) => yes


(let ((args 42))
  (if (number? args) ‘yes ‘no)) => no
```

Environments.

```
(define if (nlambda (args env)
              (cond ((eval (first args) env) (eval (second args) env))
                    (t                       (eval (third args) env)))))
```

Macros.

```
(define if (nlambda (args env)
                (cond ((eval (first args) env) (eval (second args) env))
                      (t                       (eval (third args) env)))))

(defmacro if (args)
   `(cond (,(first args) ,(second args))
          (t            ,(third args))))
```

And now for something completely different!

-
  ```
  (let ((a 1) (b 2) (c 3) (d 4))
    (list a b c d))

  => (1 2 3 4) ; everything is evaluated
  ```

-
  ```
  (let ((a 1) (b 2) (c 3) (d 4))
    (list ‘a b c d))

  => (A 2 3 4) ; not everything is evaluated
  ```

Constructing lists.

-
  ```
  (let ((a 1) (b 2) (c 3) (d 4))
    (list ‘a ‘b ‘c d))

  => (A B C 4) ; very little is evaluated
  ```

- Here is a more concise way to write this:

  ```
  (let ((a 1) (b 2) (c 3) (d 4))
    `(a b c ,d))

  => (A B C 4) ; very little is evaluated
  ```

Backquote.

- `(a b c ,d) uses backquote
- ‘(a b c d) uses quote
- backquote allows evaluating parts of an expression explicitly marked with a comma
- you can’t do this with quote

Backquote.

- `` `(a b c) <=> ‘(a b c) ``
- `` `(a ,b c ,d) <=> (list ‘a b ‘c d) ``
- `` (let ((b 2)) `(a (,b c))) => (A (2 C)) ``

-
  ```
  (let ((a 1) (b 2) (c 3))
     `(a b ,c (‘,(+ a b c)) (+ a b) ‘c ‘((,a ,b))))

  => (A B 3 (‘6) (+ A B) ‘C ‘((1 2)))
  ```

More backquote.

-
  ```
  (let ((list ‘(1 2 3)))
    `(a b ,@list c d))

  => (a b 1 2 3 c d)
  ```

- ,@ (comma-at) splices into the surrounding list
  (so there must be a surrounding list!)

break

Macros.

- This is code: (+ 1 2 3)
- This is data: ‘(+ 1 2 3)
- Macros are like functions,
  but they take code as arguments and return new code.

Macro example.

-
  ```
  (defun while-fun (predicate thunk)
     (when (funcall predicate)
         (funcall thunk)
         (while-fun predicate thunk)))
  ```

-
  ```
  (defmacro while (expression &body body)
      (list ‘while-fun (list ‘lambda ‘() expression)
                       (list* ‘lambda ‘() body)))
  ```

Macro example.

-
  ```
  (defun qsort (vector low high)
      (let* ((left low)
             (right high)
             (pivot ...))
        ...
          (while (< (svref vector left) pivot)
             (incf left))
        ...))
  ```

Macro example.

- (funcall (macro-function ‘while)

```
    ‘(while (< (svref vector left) pivot)
         (incf left))
    environment)

=> (while-fun
       (lambda ()
          (< (svref vector left) pivot))
       (lambda () (incf left)))
```

Macro example.

-
  ```
  (defun qsort (vector low high)
    (let* (...)
      ...
        (while-fun
          (lambda ()
            (< (svref vector left) pivot))
          (lambda () (incf left)))
      ...))
  ```

Macros + backquote.

- This looks ugly:

```
(defmacro while (expression &body body)
    (list ‘while-fun (list ‘lambda ‘() expression)
                     (list* ‘lambda ‘() body)))
```

- You can also write that:

```
(defmacro while (expression &body body)
    `(while-fun (lambda () ,expression)
         (lambda () ,@body)))
```

Why macros?

- Use macros for syntactic abstractions.
- Question: Why not just say this?

```
(while (lambda () (< (svref vector left) pivot))
    (lambda () (incf left)))
```

Syntactic abstractions.

- The while function leaks: You need to know details about its implementation!
- That is, the fact that it uses closures.

Alternative implementations of while.

-
  ```
  (defmacro while (expression &body body)
     `(do () ((not ,expression)) ,@body))
  ```

-
  ```
  (defmacro while (expression &body body)
    `(tagbody
        10 (unless ,expression (go 20))
           ,@body
           (go 10)
        20))
  ```

Example: Mapping.

```
(defun mapper (f list)
   (if (null list) ‘()
      (cons (funcall f (car list)) (mapper f (cdr list)))))

(defun test ()
   (mapper (lambda (x) (+ x 1)) (list 1 2 3)))
```

Example: Mapping.

```
(defun mapper (f list)
  (if (null list) ‘()
    (cons (funcall f (car list)) (mapper f (cdr list)))))

(defun test ()
  (mapper (lambda (x) (+ x 1)) (list 1 2 3)))

(defmacro mapping ((x list) &body body)
  `(mapper (lambda (,x) ,@body) ,list))

(defun test2 ()
  (mapping (x (list 1 2 3))
    (+ x 1)))
```

Example: Mapping.

```
(defmacro mapping ((x list) &body body)
  (with-gensyms (result current)
    `(prog* ((,result (copy-list ,list))
             (,current ,result))

        10 (when (null ,current) (go 20))

           (setf (car ,current) (let ((,x (car ,current))) ,@body))
           (setf ,current (cdr ,current))
           (go 10)

        20 (return ,result))))

(defun test2 ()
  (mapping (x (list 1 2 3))
    (+ x 1)))
```

Abstractions.

- Syntactic abstractions hide implementation details,
  just like functional abstractions.
- Hiding implementation details allows you to change your mind later on,
  and allows the users to think purely in terms of what they care about.

Abstractions.

-
  ```
  (while-fun (lambda () (< (svref ...) pivot))
      (lambda () (incf left)))
  ```

  vs.

-
  ```
  (while (< (svref ...) pivot)
      (incf left))
  ```

Introduction of macros.

"In LISP 1.5 special forms are used for three logically separate purposes:
a) to reach the alist, b) to allow functions to have an indefinite number of arguments, and c) to keep arguments from being evaluated.
New LISP interpreters can easily satisfy need (a) by making the alist a SPECIAL-type or APVAL-type entity. Uses (b) and (c) can be replaced by incorporating a MACRO instruction expander in define. I am proposing such an expander.
1. The property list of a macro will have the indicator MACRO followed by a function of one argument, a form beginning with the macro's name, and whose value will replace the original form in all function definitions."

(MACRO Definitions for LISP, Timothy P. Hart, 1963)

Early forms of macros.

“A macro definition of our DESCRIBE special form might look like this:

```
(DEFUN DESCRIBE MACRO (X)
        (LIST 'PRINC
              (LIST 'LOOKUP-DOCUMENTATION
                    (LIST 'QUOTE (CADR X)))))
```

[...] MACLISP and LISPMACHINE Lisp [...] allows the following alternate syntax for DESCRIBE's definition as a macro:

```
(DEFUN DESCRIBE MACRO (X)
    `(PRINC (LOOKUP-DOCUMENTATION ',(CADR X))))”
```

(Special Forms in Lisp, Kent Pitman, 1980)

Quasiquotation.

“[...] nothing resembles today's Lisp quasiquotation as closely as the notation in McDermott and Sussman's Conniver language.”

(Quasiquotation in Lisp, Alan Bawden, 1999)

- Alan Bawden refers here to the Conniver reference manual from 1972.

Common Lisp vs. Scheme.

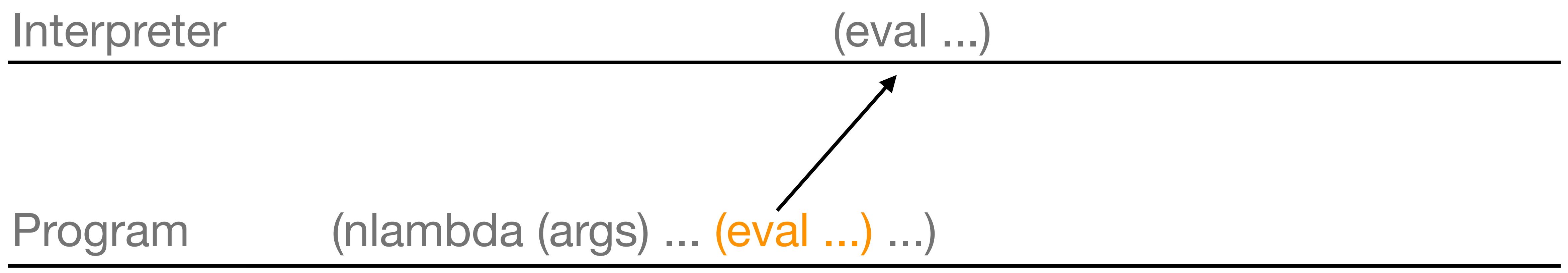
- For Common Lisp, the story ends here (almost):
  - Macro expanders are arbitrary Lisp functions.
  - Backquote/quasiquote is used to construct new forms.
  - There is limited support for destructuring and environment access.

- In Scheme, research has continued:
  - Macro hygiene: Avoid name clashes, similar to dynamic scoping problems.
  - Destructuring forms: syntax-rules / syntax-case.

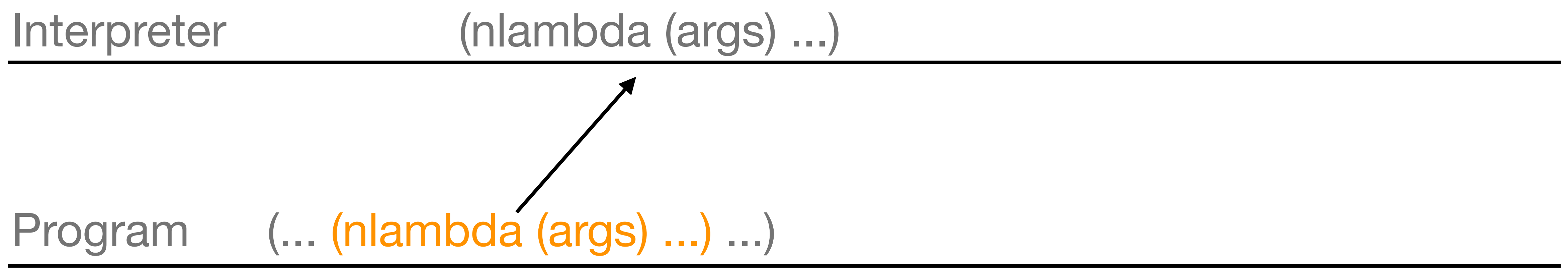
Intermission.

- What we have seen so far:
    - Early approaches for ad-hoc reflection.
    - Macros as the compile-time substrate of reflection.
- Next: More principled approaches to reflection.

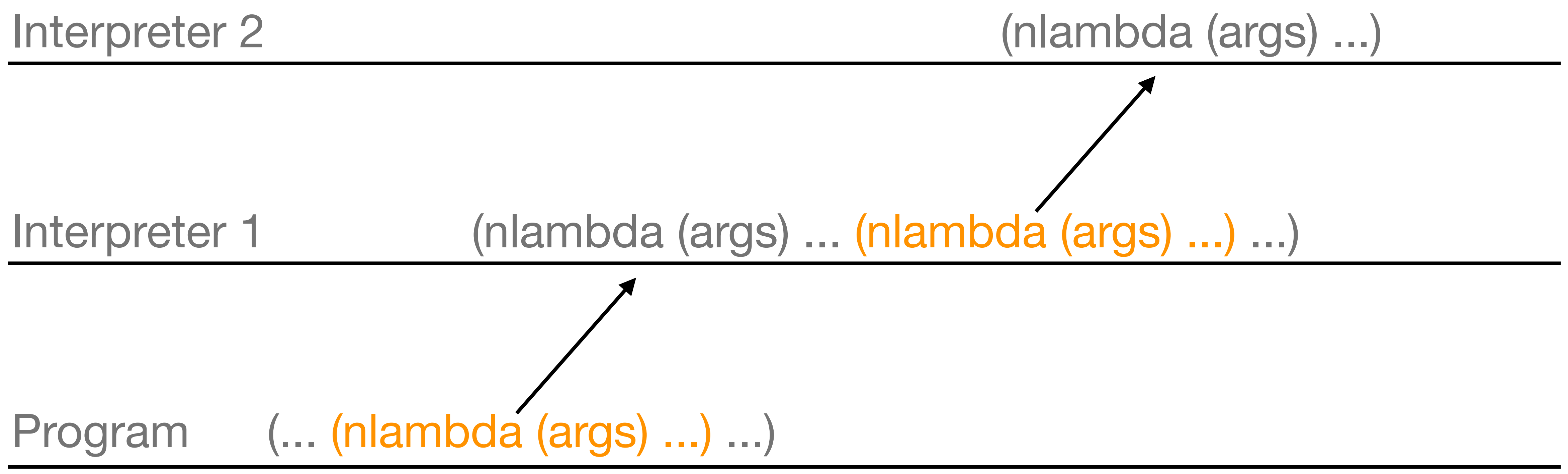
“Early” Lisps.



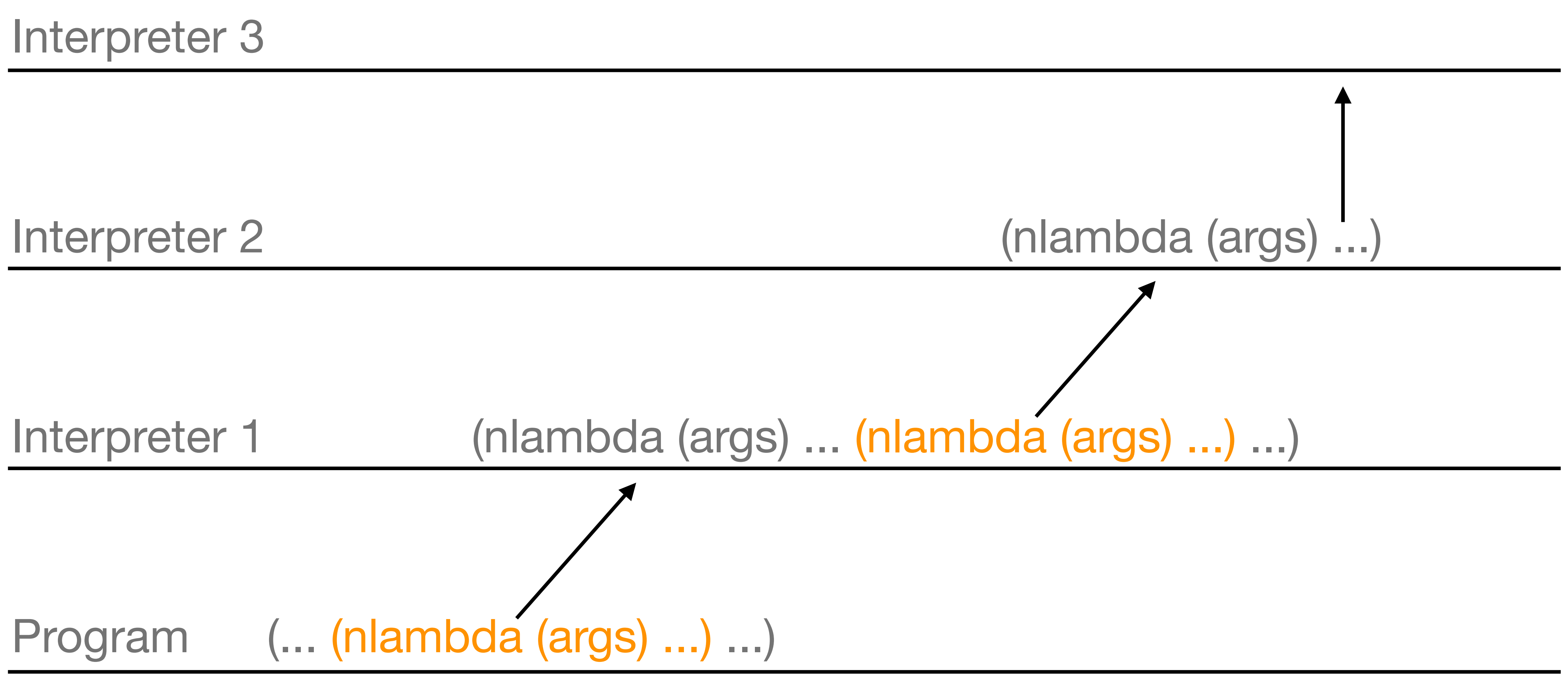
3-Lisp.



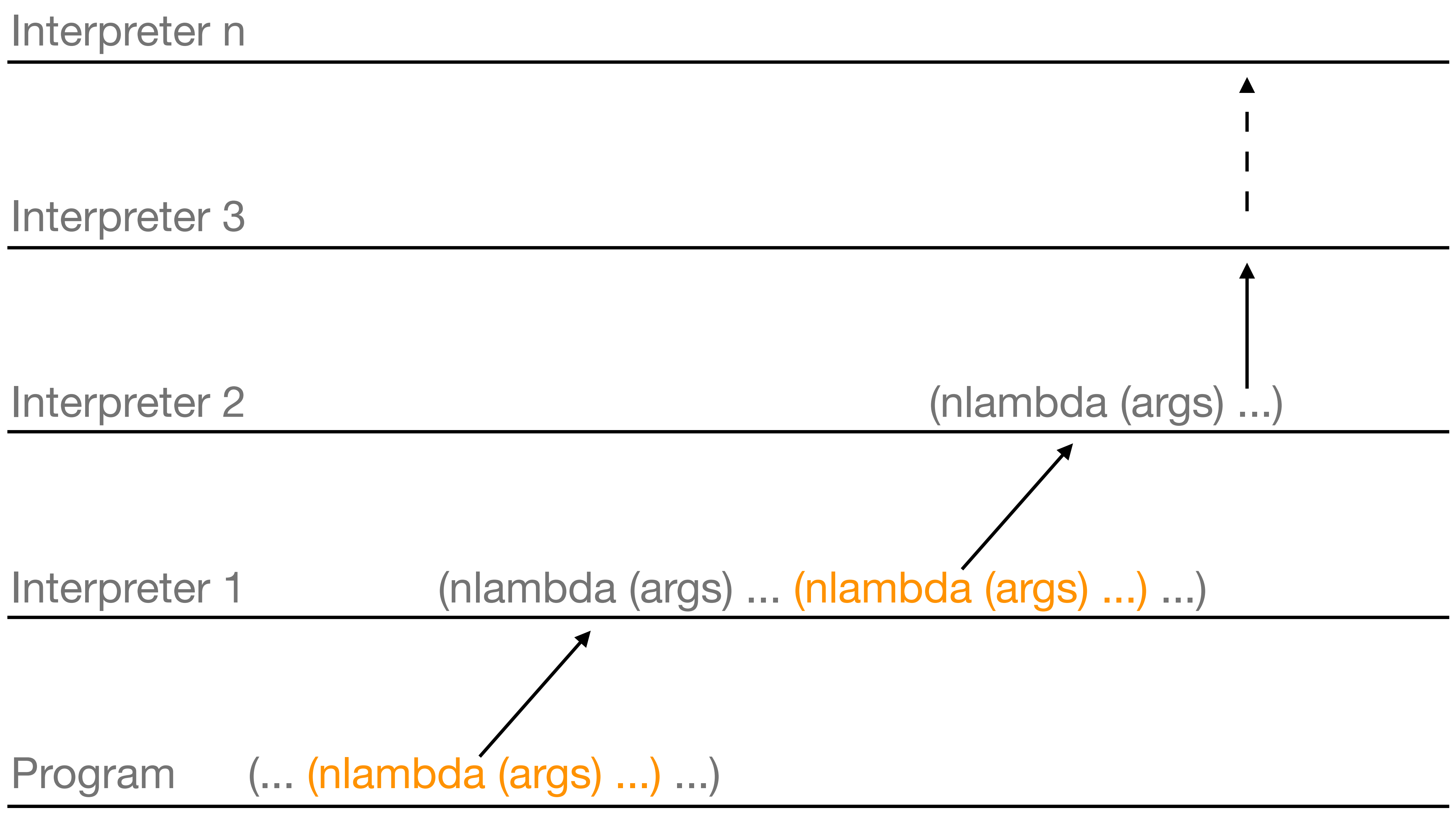
3-Lisp.


Interpreter 2
(nlambda (args) ...)
Interpreter 1
(nlambda (args) ... (nlambda (args) ...) ...)
Program
(... (nlambda (args) ...) ...)

3-Lisp.


Interpreter 3
Interpreter 2
(nlambda (args) ...)
Interpreter 1
(nlambda (args) ... (nlambda (args) ...) ...)
Program
(... (nlambda (args) ...) ...)

Reflective Tower.



Implementation of 3-Lisp.

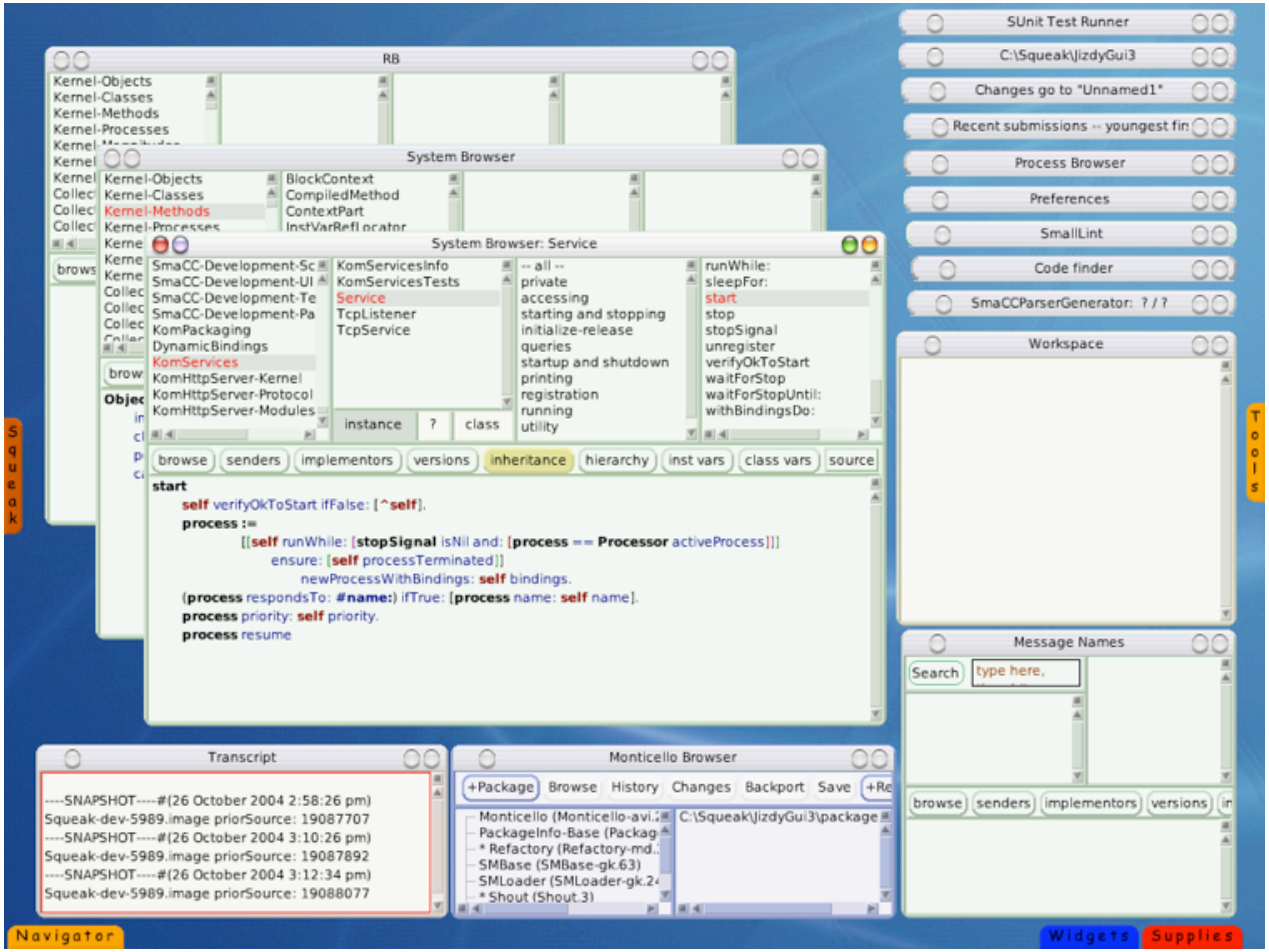
“The key observation is that the activity at most levels - in fact at all but a finite number of the lowest levels - will be monotonous: [...] From some finite level **k** all the way to the “top”, in other words, the tower will just consist of the processor processing the processor. [...] Call a processing level *boring* if the only expressions that are processed at that level [...] are kernel expressions. [...] Just as a correct implementation of recursion is not required to terminate when a procedure recurses indefinitely, a correct implementation of a procedurally reflective system need terminate only on computations having a *finite* degree of introspection. Tractable reflective programs, in other words, are those with a finite degree of introspection.”

(The Implementation of Procedurally Reflective Languages,
Jim des Rivières, Brian C. Smith, 1984)

Meanwhile...

- Alan Kay: “Everything is an object.”
- If that’s the case, then so are classes, methods, fields, stacks, ...

Really everything is an object!



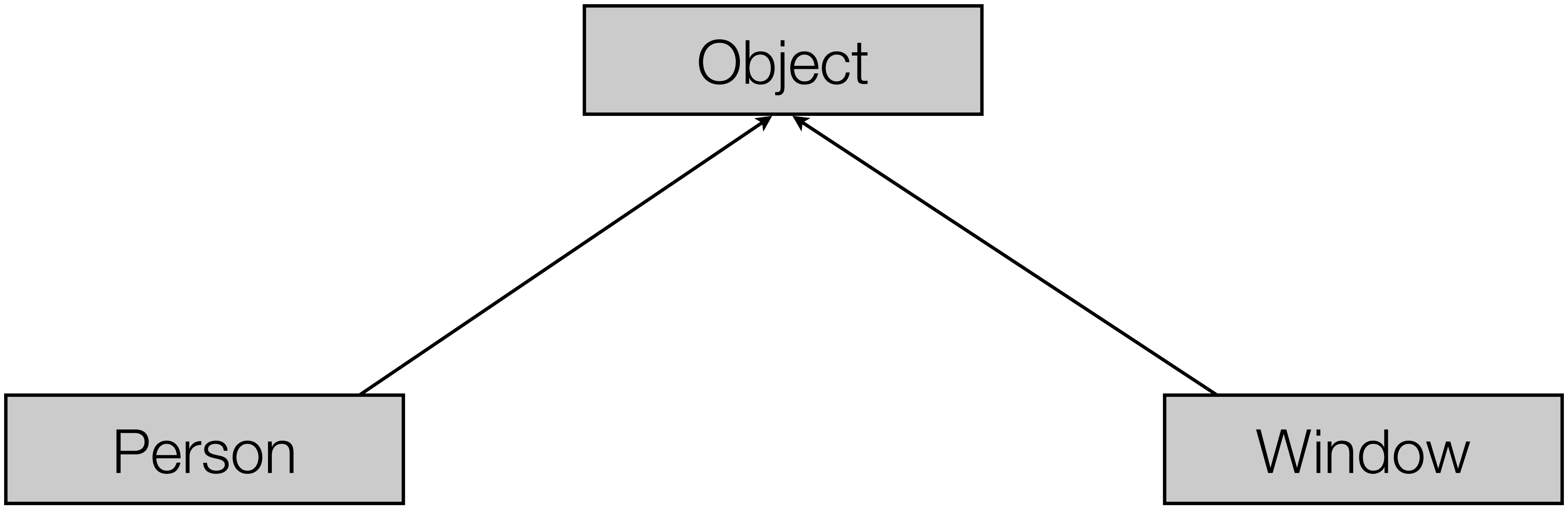
```
SmaCC-Development-Sc    KomServicesInfo     -- all --                runWhile:
SmaCC-Development-UI    KomServicesTests    private                  sleepFor:
SmaCC-Development-Te    Service             accessing                start
SmaCC-Development-Pa    TcpListener         starting and stopping    stop
KomPackaging            TcpService          initialize-release       stopSignal
DynamicBindings                             queries                  unregister
KomServices                                 startup and shutdown     verifyOkToStart
KomHttpServer-Kernel                        printing                 waitForStop
KomHttpServer-Protocol                      registration             waitForStopUntil:
KomHttpServer-Modules                       running                  withBindingsDo:
                                            utility
```

```
start
	self verifyOkToStart ifFalse: [^self].
	process :=
		[[self runWhile: [stopSignal isNil and: [process == Processor activeProcess]]]
			ensure: [self processTerminated]]
				newProcessWithBindings: self bindings.
	(process respondsTo: #name:) ifTrue: [process name: self name].
	process priority: self priority.
	process resume
```

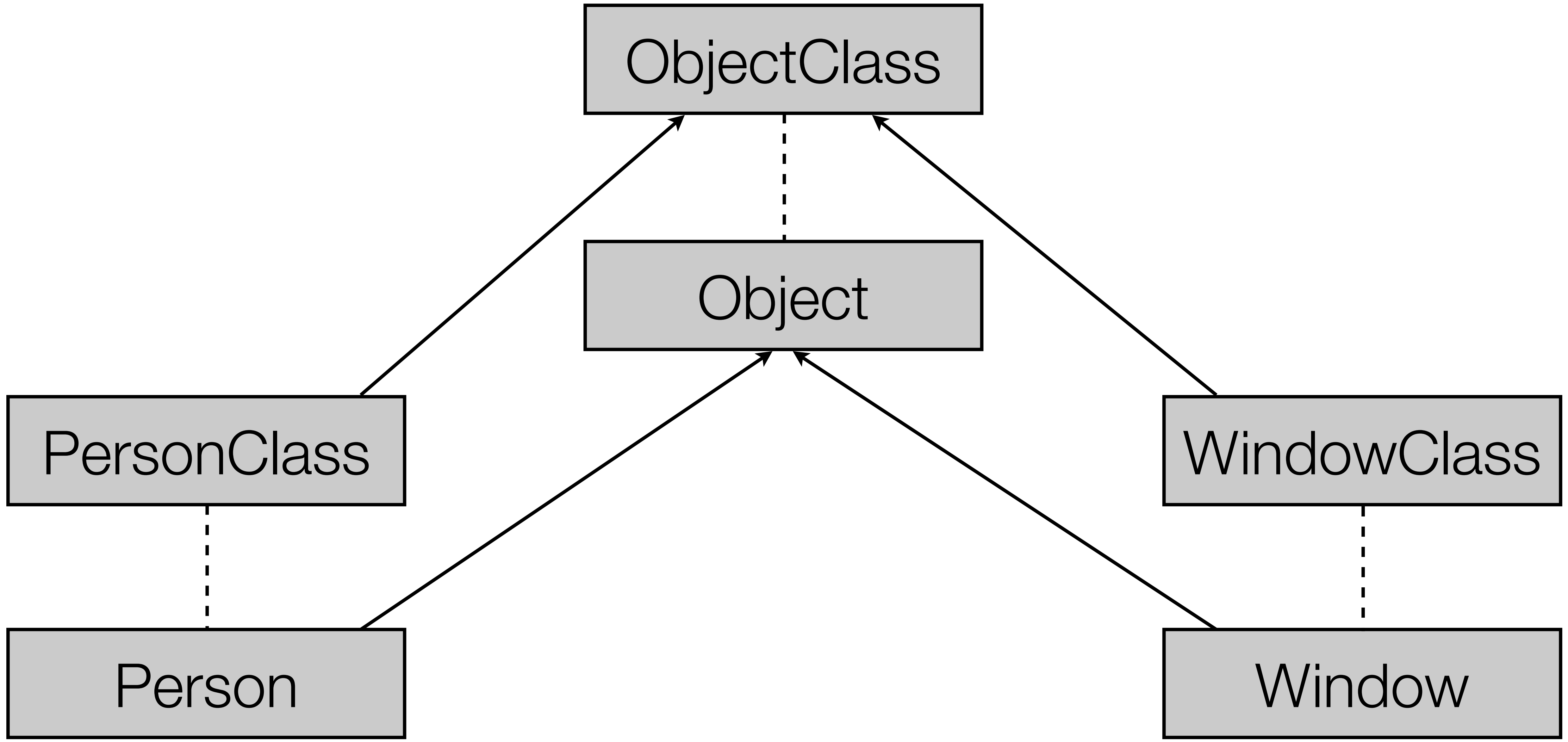
```
----SNAPSHOT----#(26 October 2004 2:58:26 pm)
Squeak-dev-5989.image priorSource: 19087707
----SNAPSHOT----#(26 October 2004 3:10:26 pm)
Squeak-dev-5989.image priorSource: 19087892
----SNAPSHOT----#(26 October 2004 3:12:34 pm)
Squeak-dev-5989.image priorSource: 19088077
```

```
Monticello (Monticello-avi.2
PackageInfo-Base (Packag
* Refactory (Refactory-md.
SMBase (SMBase-gk.63)
SMLoader (SMLoader-gk.24
* Shout (Shout.3)
```

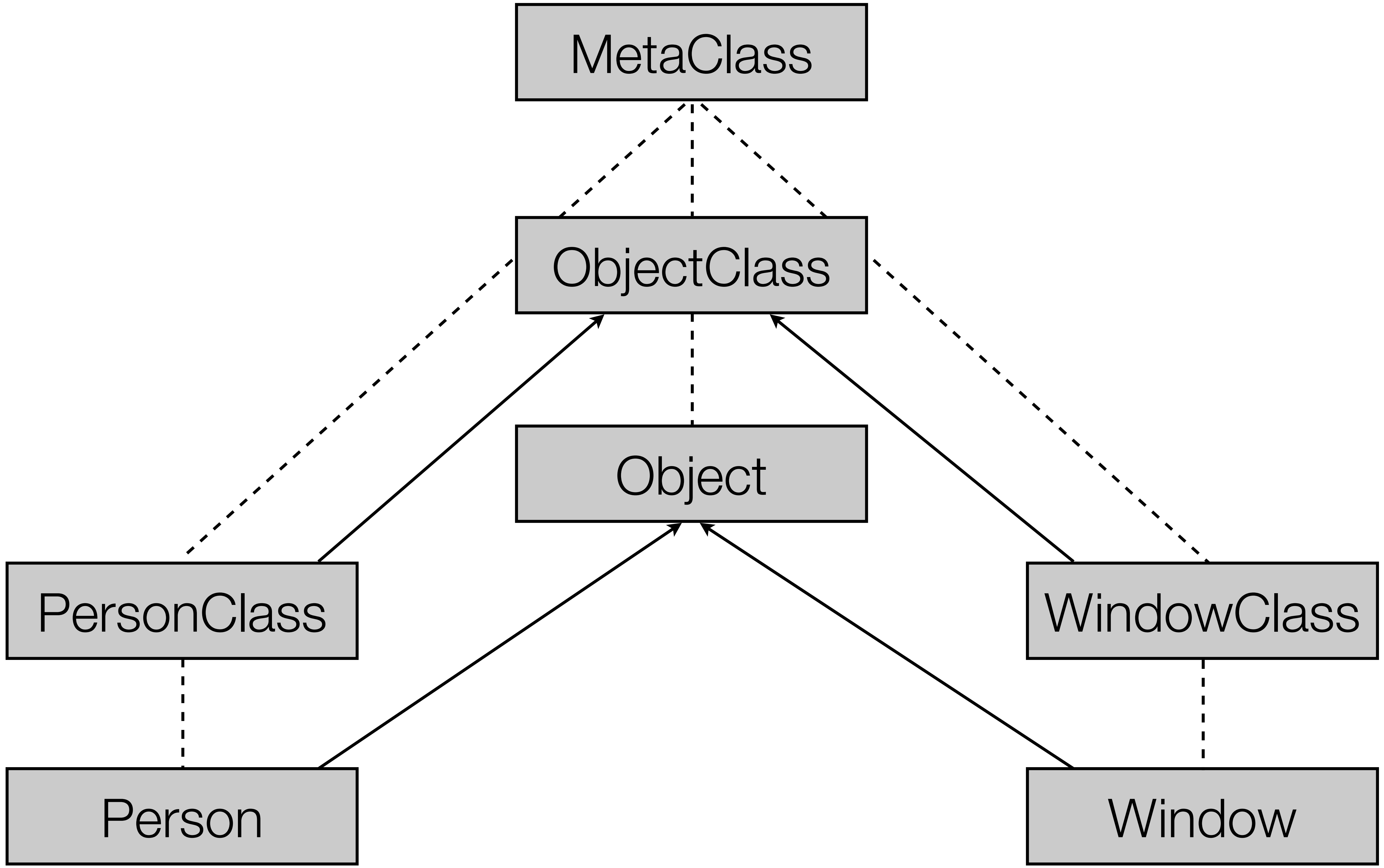
Smalltalk.



The Metalevel in Smalltalk.



The Metalevel in Smalltalk.



break

“Tower” issues in program transformation.

-
  ```
  public class C {

     public B b = new B();


     public void doSomething() {
        b.manipulate();
     }

  }
  ```

Add a counter for accesses of b.

-
  ```
  public class C {

    public B b = new B();
    private int b_counter = 0;

    public void doSomething() {
      b_counter++;
      b.manipulate();
    }

  }
  ```

Another transformation: Adding getters & setters.

- public class C {

```
    private B b = new B();

    public B getB()          {return this.b;}
    public void setB(B _b) {this.b = _b;}

    public void doSomething() {
      getB().manipulate();
    }

}
```

Both transformations combined...

- public class C {

```
public class C {
   private B b = new B();
   private int b_counter = 0;

   public B getB() {
      b_counter++;
      return this.b;
   }
   public void setB(B _b) {this.b = _b;}

   public void doSomething() {
      b_counter++;
      getB().manipulate();
   }
}
```

Both transformations combined...

```
• public class C {
    private B b = new B();
    private int b_counter = 0;

    public B getB() {
      b_counter++;
      return this.b;
    }
    public void setB(B _b) {this.b = _b;}

    public void doSomething() {
      b_counter++;
      getB().manipulate();
    }
  }
```

This insert depends on ordering of transformations!

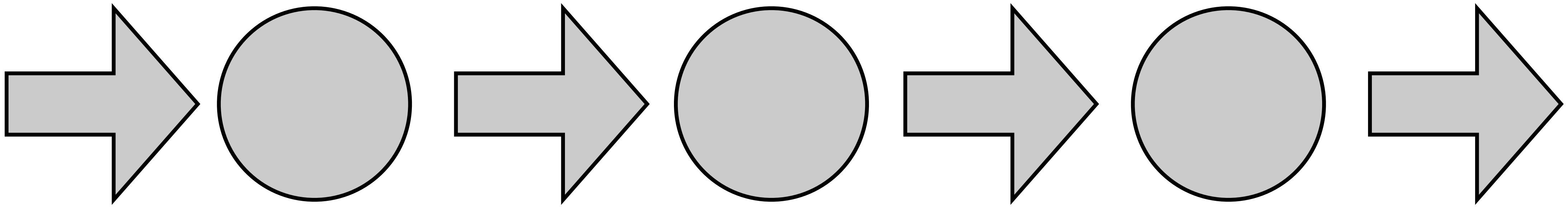
Both transformations combined...

```
• public class C {
     private B b = new B();
     private int b_counter = 0;

     public B getB() {
        b_counter++;
        return this.b;
     }
     public void setB(B _b) {this.b = _b;}

     public void doSomething() {
        b_counter++;
        getB().manipulate();
     }
  }
```

Should we add a read counter here as well?

This insert depends on ordering of transformations!

Solution: Define an ordering of transformations.



From 3-Lisp to Metaobject Protocols .

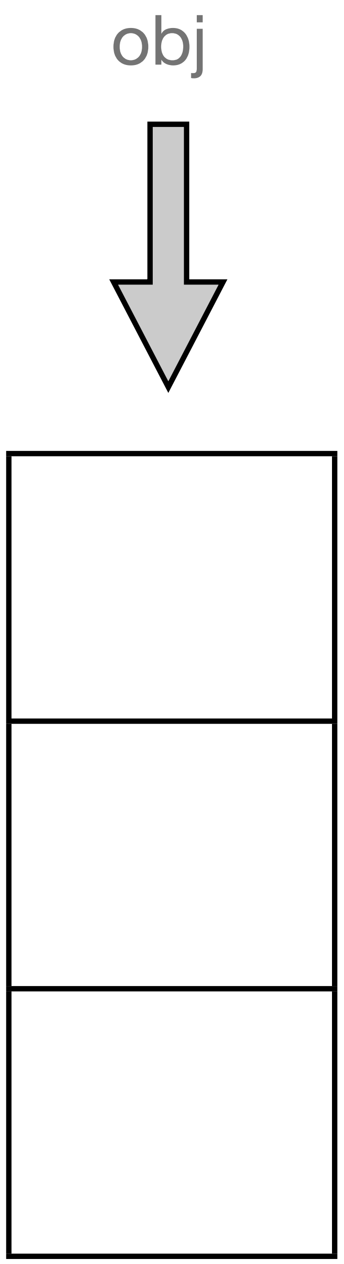
- 3-Lisp has a layered design in which the levels can interact.
- Class hierarchies are layered designs in which the levels can interact.
- Can this be combined?

What is an object?

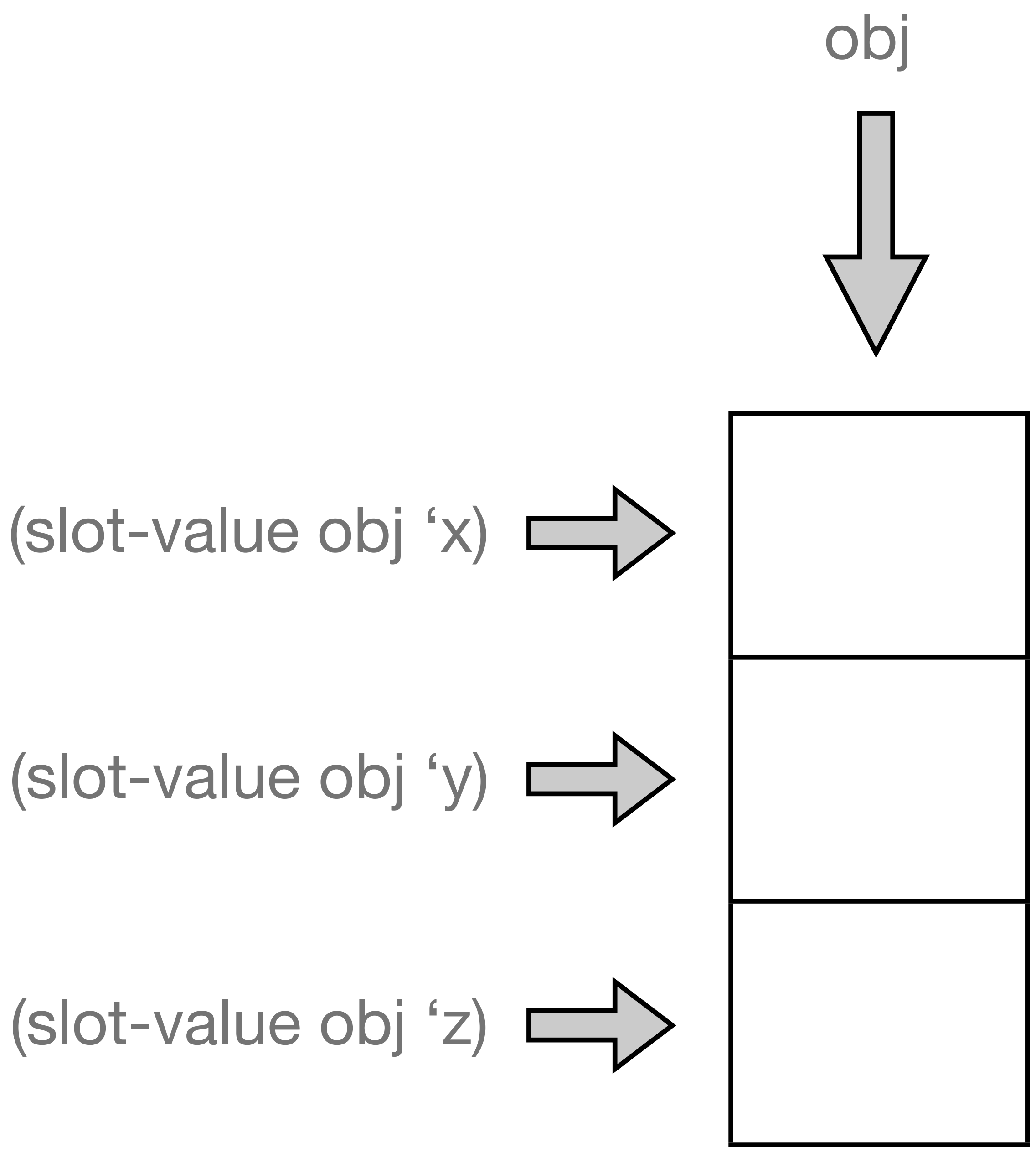
“An object has state, behavior, and identity.”

(Object-oriented Analysis and Design with Applications, Grady Booch, 1991)

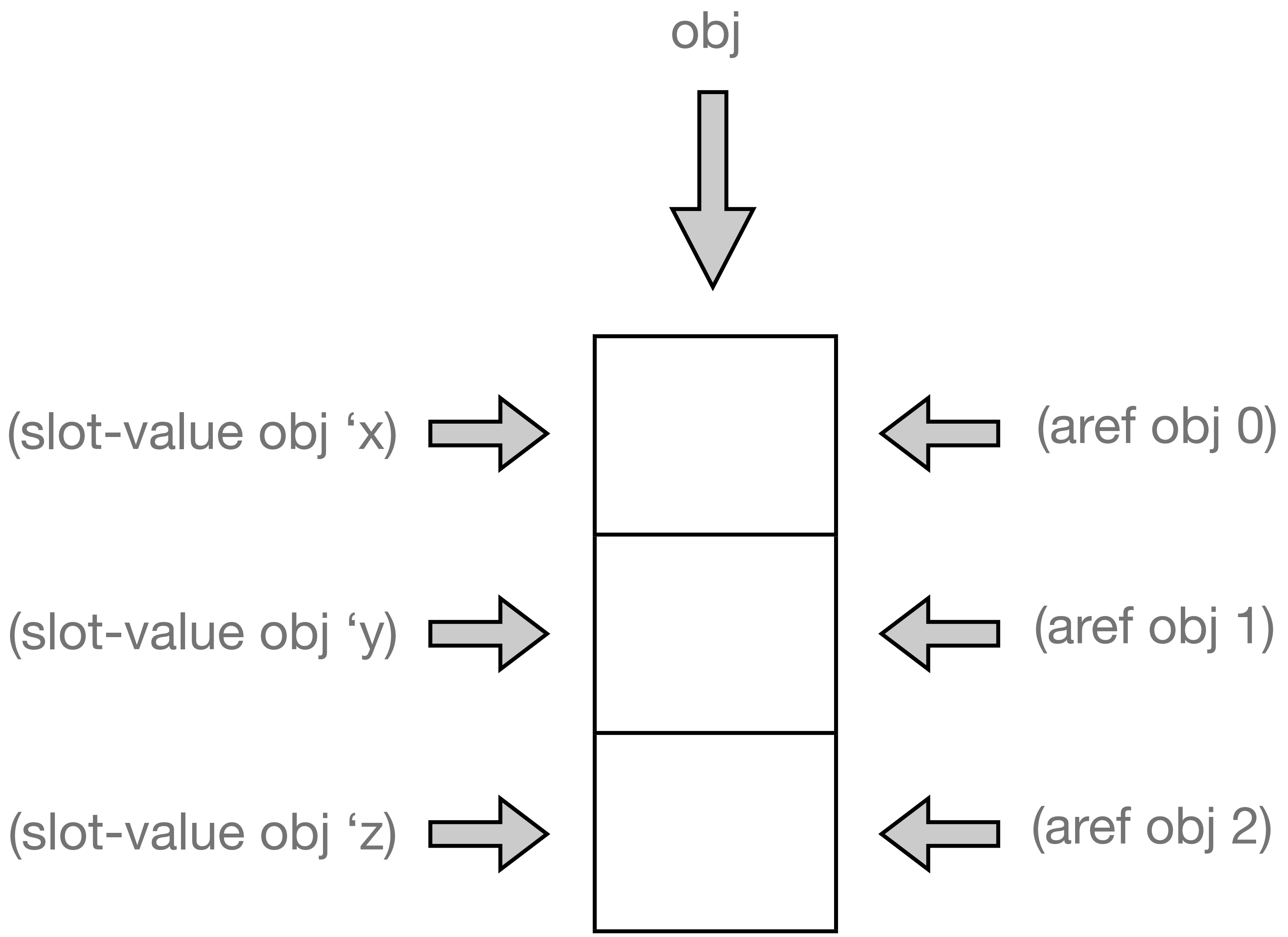
State



State

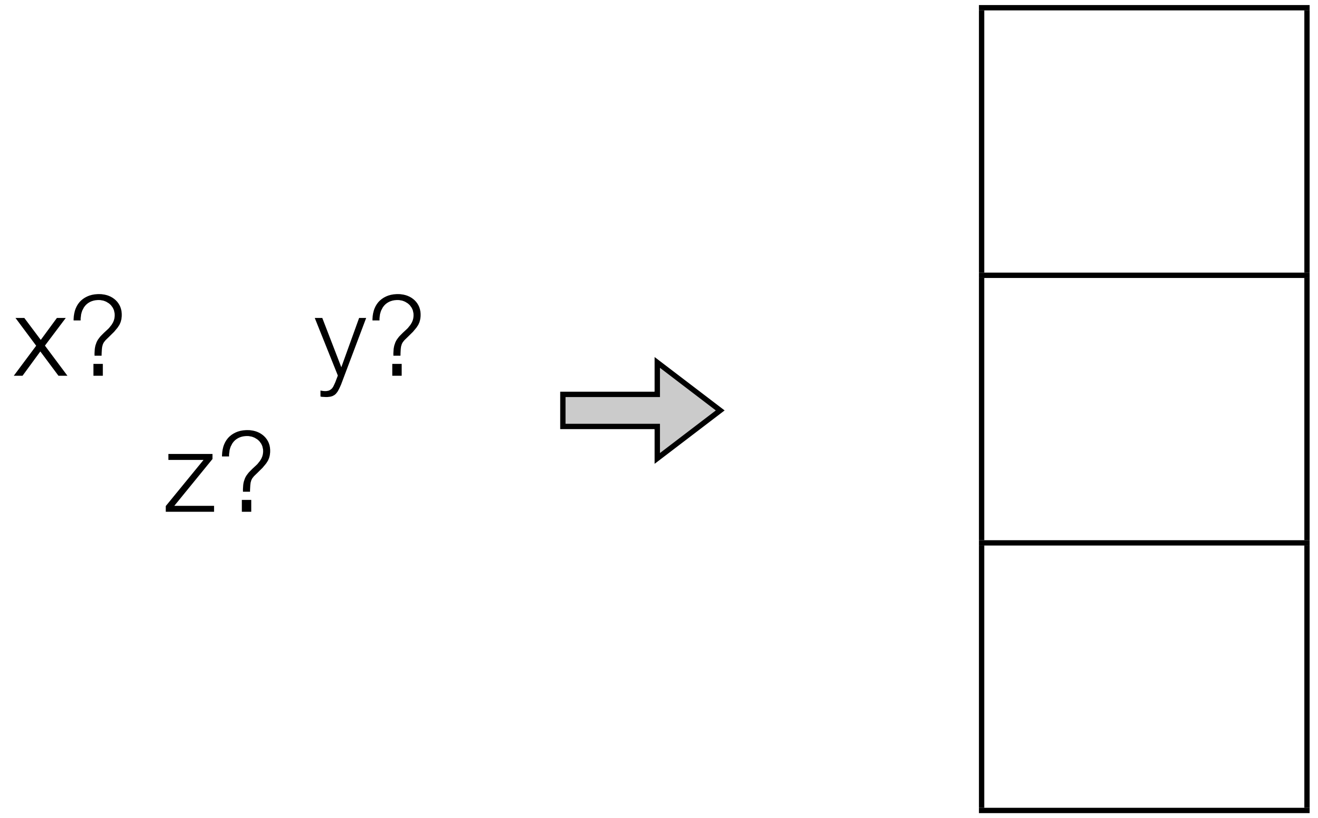


State


obj
(slot-value obj ‘x)
(slot-value obj ‘y)
(slot-value obj ‘z)
(aref obj 0)
(aref obj 1)
(aref obj 2)

How to map slots?

```
(defclass point ()
   (x y))

(defclass point-3d (point)
   (z))
```



How to map slots?

1. compute class precedence list
2. compute slots
3. determine slot locations

How to map slots?

1. (compute-class-precedence-list ...)
2. (compute-slots ...)
3. (slot-definition-location ...)

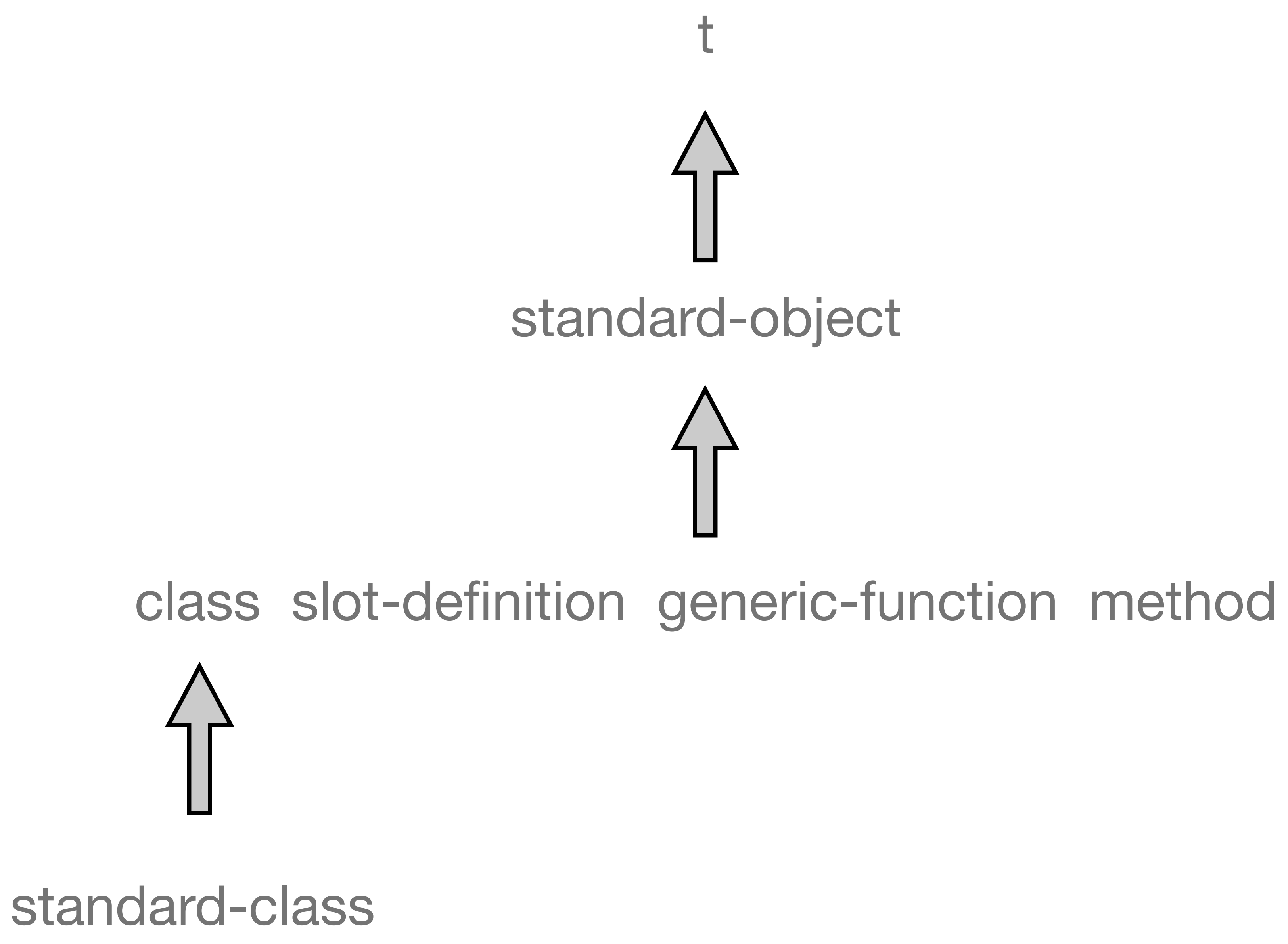
The CLOS Metaobject Protocol.

- Make compute-class-precedence-list, compute-slots, etc., generic functions!
- Allow changes to the CLOS object model!
- Question: How to distinguish between standard and non-standard behavior?
- [CLOS = Common Lisp Object System.]

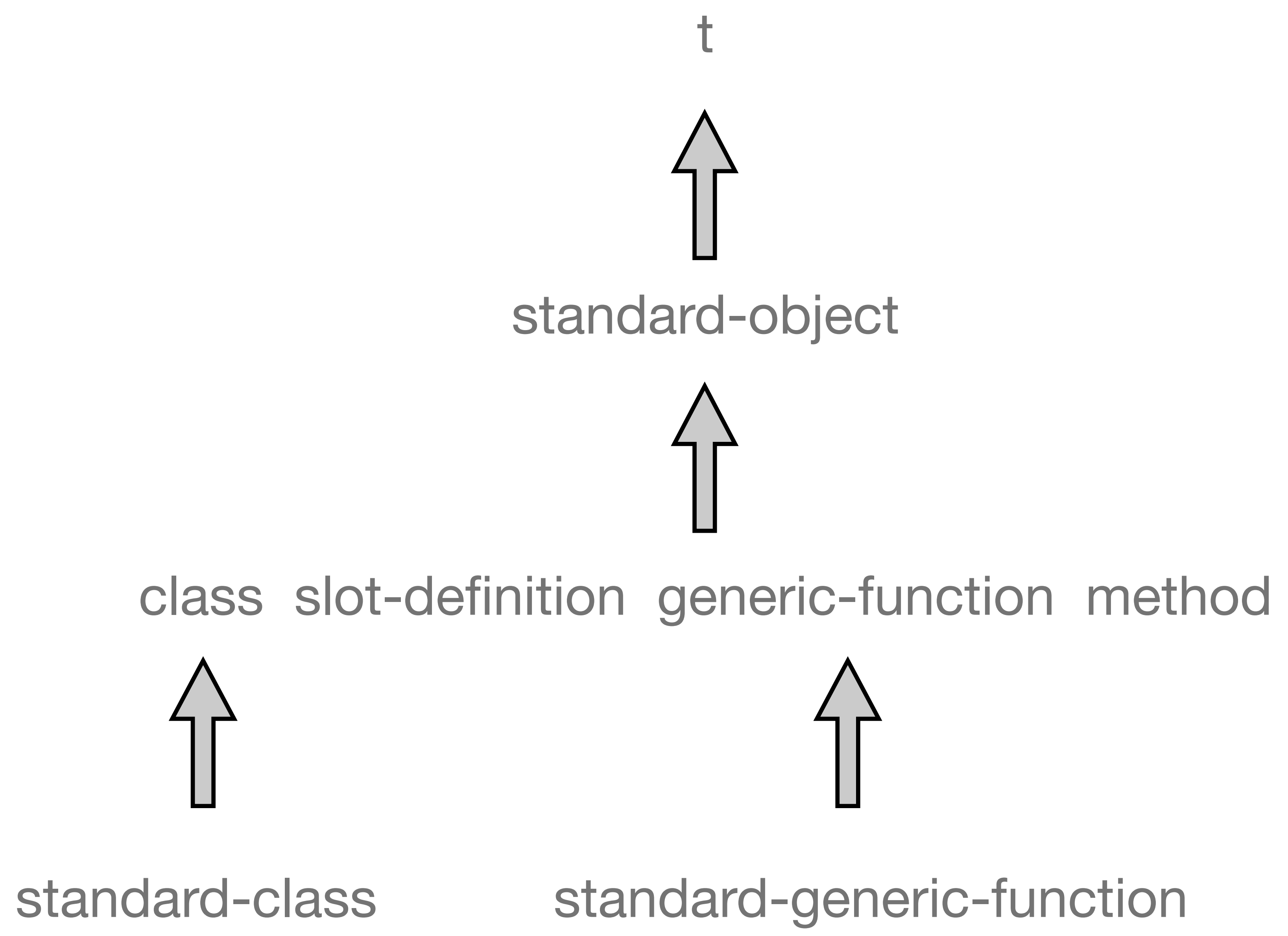
Hierarchy for metaobject classes.

$$\begin{tikzcd}
\text{t} \\
\text{standard-object} \arrow[u] \\
\text{class\ \ slot-definition\ \ generic-function\ \ method} \arrow[u]
\end{tikzcd}$$

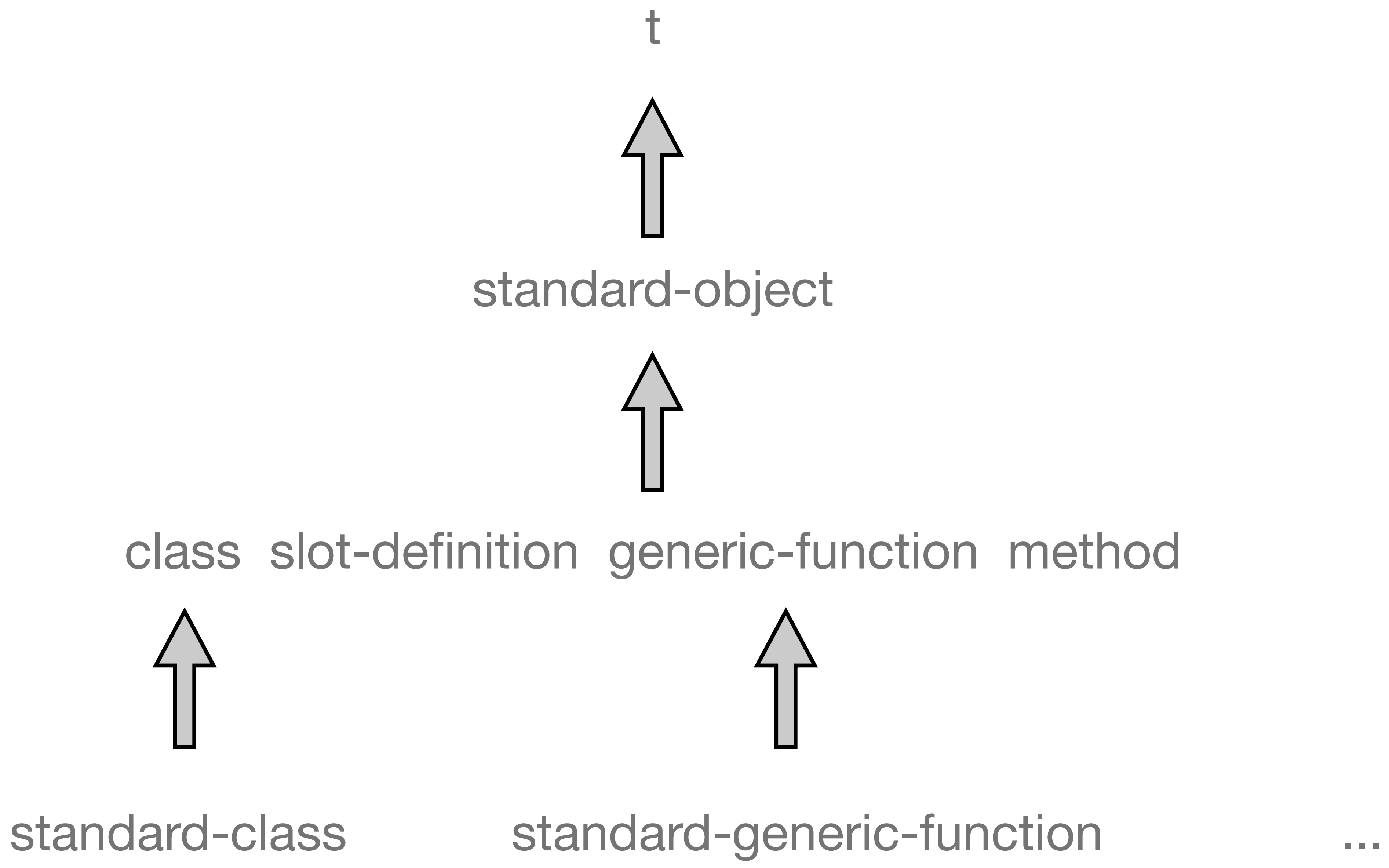
Hierarchy for metaobject classes.



Hierarchy for metaobject classes.


t
standard-object
class slot-definition generic-function method
standard-class
standard-generic-function

Hierarchy for metaobject classes.


t
standard-object
class slot-definition generic-function method
standard-class
standard-generic-function
...

Class metaobject classes.

```
(defclass persistent-class (standard-class)
  ((database-connection ...)))

(defclass person ()
  ((name ...)
   (address ...))
  (:metaclass persistent-class))
```

The Instance Structure Protocol.

```
(defmethod person-name ((object person))
  (slot-value object ‘name))

(defun slot-value (object slot)
  (slot-value-using-class
    (class-of object) object slot))

(defmethod slot-value-using-class
  ((class standard-class) object slot)
  (aref ...))
```

The Instance Structure Protocol.

```
(defmethod slot-value-using-class
   ((class persistent-class) object slot)
   (fetch-slot-from-database ...))
```

Separation between base and meta level.

- In 3-Lisp, nlambda performs an explicit shift to the meta-level.
- In CLOS, class-of performs an explicit shift to the meta-level.

Limitations.

- Macros are not first-class entities at runtime.
- The CLOS MOP does not provide interception of argument evaluation.
- Note: Whenever an approach is simplified, expressive power might get lost!
- Motivation: Efficiency!

(The Theory of Fexprs is Trivial,
Mitchell Wand, 1998)

Limitations of reflective systems.

"Reflective systems are intended to be open enough to allow the user to extend and modify them easily. In this paper we show that the openness and extensibility of a reflective system depends to a great degree on the choice of its underlying representations. Giving the language the necessary expressive power requires forethought in the design stage of the kinds of extensions the user might wish to make."

(A Reflective System is as Extensible as its Internal Representations: An Illustration, John W. Simmons II, Daniel P. Friedman, 1992)

- See also the notion of open implementations.

“Modern” Forms of Metaprogramming & Reflection

- XML + hand-written evaluators
- Annotations in Java and C#: syntactic abstractions
- Java’s ClassLoader architecture
- Java Dynamic Proxy Classes
- Template metaprogamming
- Program transformation frameworks
- ...

Task

- Pick a language of your choice.
- Describe its facilities for metaprogramming & reflection.

  + When can you do metaprogramming?
    (compile-time, load-time, runtime?)

  + What can you do? (introspection, intercession)

  + How are programs represented? (bytecode, data structures, closures, etc.)

  + How can you achieve what you want?
    What are typical examples and actual uses?

  + Bonus: Is there a tower? How are “tower issues” dealt with?

Next lesson.

- Advanced Lisp metaprogramming.
  + Generic functions.
  + CLOS, CLOS MOP.

- Advanced Java metaprogramming.
  + Annotation processing.
  + Class loading.
  + Dynamic proxies.

End of presentation.

References: “Early” Lisp

- John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, Michael I. Levin. Lisp 1.5 Programmer's Manual. MIT Press, Cambridge, Massachusetts, 1962. http://community.computerhistory.org/scc/projects/LISP/book/LISP%201.5%20Programmers%20Manual.pdf

- Warren Teitelman, PILOT: A Step Toward Man-Computer Symbiosis. PhD Thesis. MIT-AI-TR-221, MIT, September 1966. http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-032.pdf

References: 3-Lisp / Procedural Reflection

- Jim des Rivieres, Control-related meta-level facilities in LISP. In P. Maes and D. Nardi, editors, Meta-Level Archtitectures and Reflection, North-Holland, 1988.
- Jim des Rivières and Brian Cantwell Smith. "The implementation of procedurally reflective languages". 1984 ACM Symposium on LISP and functional programming. August 1984.
- John Wiseman Simmons II and Daniel P. Friedman. "A Reflective System is as Extensible as its Internal Representations: An Illustration". Computer Science Department, Indiana University. October 1992.
- See http://library.readscheme.org/page11.html for the latter two and more references.

References: CLOS MOP

- Andreas Paepcke. User-level language crafting - introducing the CLOS metaobject protocol. In Andreas Paepcke, editor, Object-Oriented Programming: The CLOS Perspective. MIT Press, 1993. http://www-db.stanford.edu/~paepcke/shared-documents/mopintro.ps

- Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow. The Art of the Metaobject Protocol. MIT Press, 1991.