

# Metaprogramming & Reflection

---

# 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, compilers, etc.
- A program with itself as (part of) its domain is a reflective program.

# Important Concepts.

---

- Introspection.
  - The ability to inspect a program.
- Intercession.
  - The ability to change a program's behavior.

# 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 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 programmer 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)

# The Roots of Lisp

---

- Seven primitive operators: quote, atom, eq, car, cdr, cons, cond
- Denoting functions:
  - (lambda (p1 ... pn) e)
  - ((lambda (f) (f '(b c))) '(lambda (x) (cons 'a x))) => (a b c)
  - (label f (lambda (p1 ... pn) e)) ;; f can appear in e
- Some abbreviations: defun, cadr, caddr, ..., list
- Some functions: null, and, not, append, pair, assoc, eval, evcon, evlis

# Metacircular Interpreter.

---

Interpreter

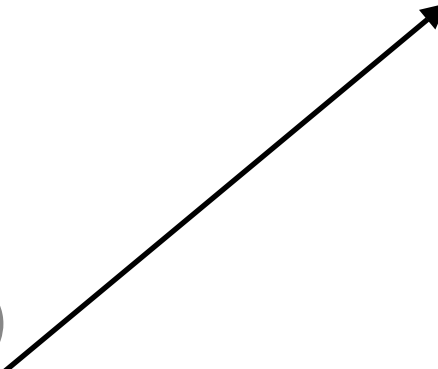
---

(eval '(+ x 1))

Base program

---

(let ((x 41))  
 (eval '(+ x 1)))



# 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
```

# Dynamic Scoping.

---

```
(define if (lambda (args)
  (cond ((eval (first args)) (eval (second args)))
        (else (eval (third args))))))
```

```
(let ((test 42))
  (if (numberp test) 'yes 'no)) => yes
```

```
(let ((args 42))
  (if (numberp args) 'yes 'no)) => no
```

# Environments.

---

```
(define if (lambda (args env)
  (cond ((eval (first args) env) (eval (second args) env))
        (else (eval (third args) env)))))
```

# Continuations...

---

# Macros.

---

```
(define if (lambda (args env)
  (cond ((eval (first args) env) (eval (second args) env))
        (else (eval (third args) env)))))
```

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



# 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 [...] allow 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.

# 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.

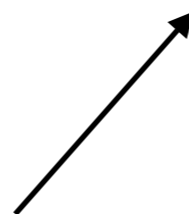
---

Interpreter

(eval ...)

Program

(nlambda (args) ... (eval ...) ...)



# 3-Lisp.

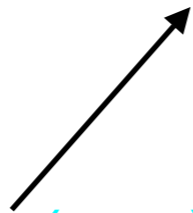
---

Interpreter (nlambda (args) ...)

---

Program (... (nlambda (args) ...) ...)

---



# 3-Lisp.

---

Interpreter 2

(nlambda (args) ...)

Interpreter 1

(nlambda (args) ... (nlambda (args) ...) ...)

Program

(... (nlambda (args) ...) ...)

```
graph BT; P["Program (... (nlambda (args) ...) ...)"]; I1["Interpreter 1 (nlambda (args) ... (nlambda (args) ...) ...)"]; I2["Interpreter 2 (nlambda (args) ...)"]; P --> I1; I1 --> I2;
```

# 3-Lisp.

---

Interpreter 3

---

Interpreter 2

---

Interpreter 1

---

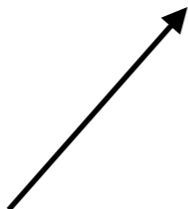
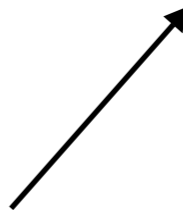
Program

---

(nlambda (args) ...)

(nlambda (args) ... (nlambda (args) ...) ...)

(... (nlambda (args) ...) ...)





# Reflective Tower.

---

Interpreter n

---

Interpreter 3

---

Interpreter 2

---

Interpreter 1

---

Program

---

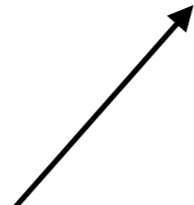
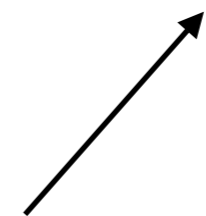
↑  
|  
|  
|

↑

(nlambda (args) ...)

(nlambda (args) ... (nlambda (args) ...) ...)

(... (nlambda (args) ...) ...)



# 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)

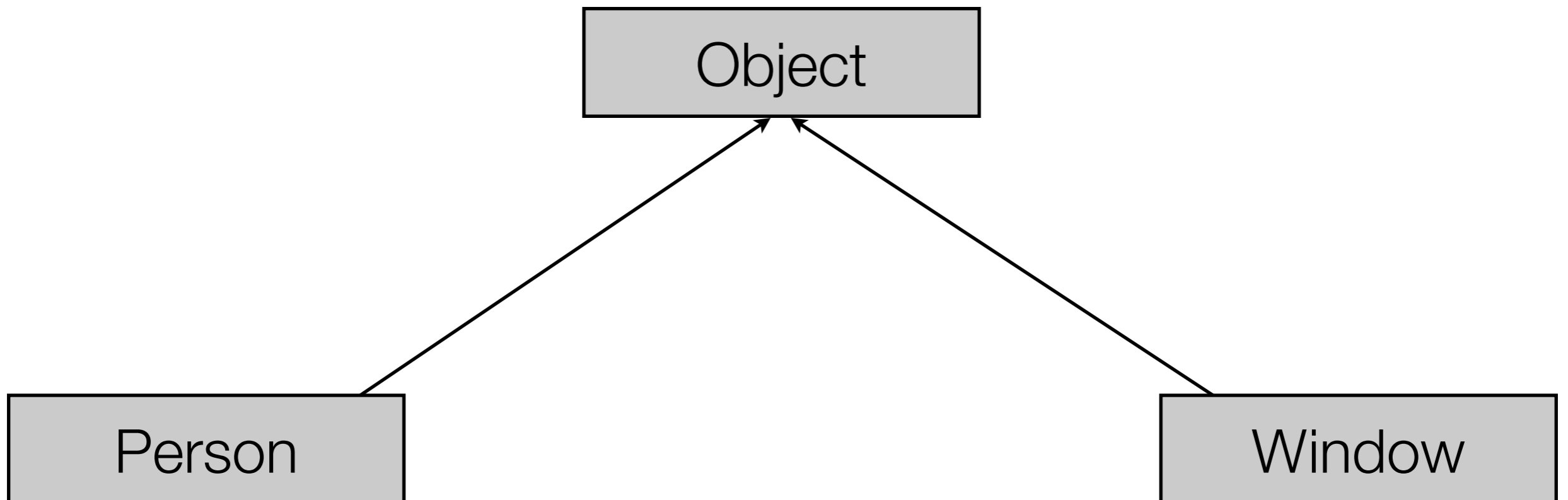
# Meanwhile...

---

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

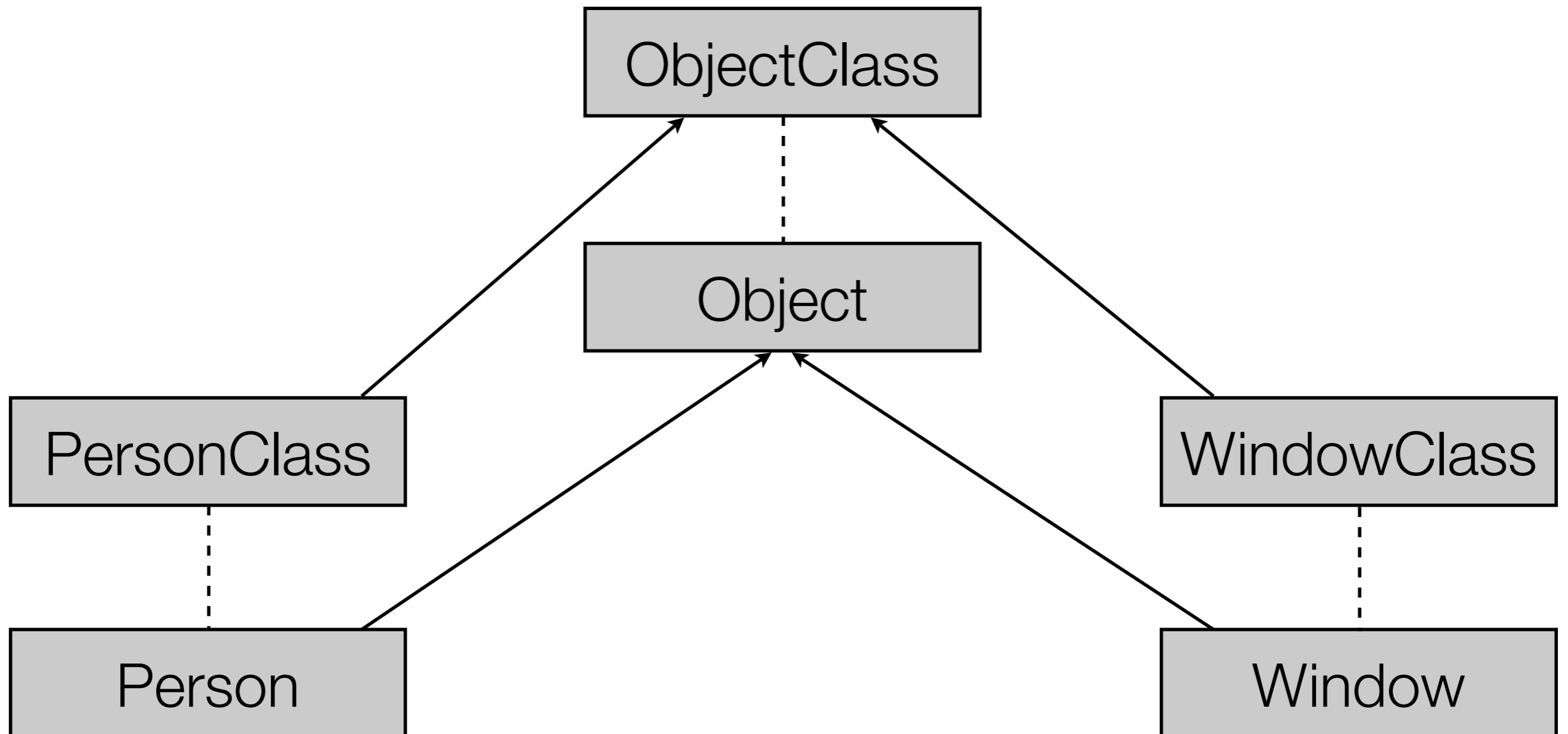
# Smalltalk.

---



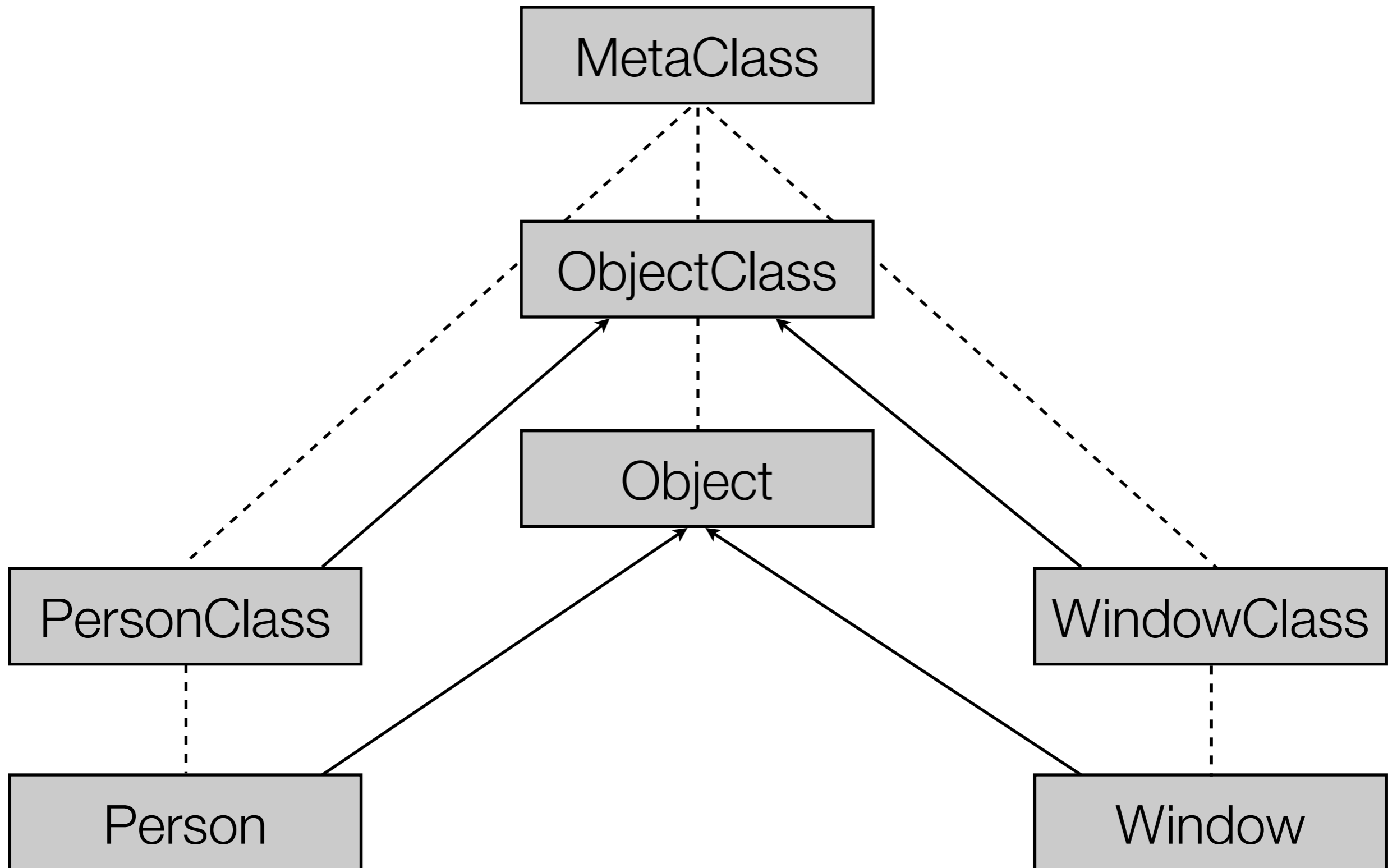
# The Metalevel in Smalltalk.

---



# The Metalevel in Smalltalk.

---



# Really everything is an object!

The image displays the Squeak Smalltalk IDE interface. The main window shows a code editor with the following code:

```
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
```

The interface includes several tool windows:

- System Browser:** Shows a hierarchy of objects, with **Kernel-Methods** selected.
- System Browser: Service:** Shows the source code for the **Service** class, including methods like **start**, **runWhile**, **sleepFor**, **stop**, **stopSignal**, **unregister**, **verifyOkToStart**, **waitForStop**, **waitForStopUntil**, and **withBindingsDo**.
- Workspace:** A large empty area for editing objects.
- Message Names:** A search window for finding messages.
- Transcript:** A window showing system messages and snapshots.
- Monticello Browser:** A window for managing packages, showing a list of packages like **Monticello**, **PackageInfo-Base**, **Refactory**, **SMBase**, **SMLoader**, and **Shout**.

On the right side, there is a vertical stack of tool windows, including **SUnit Test Runner**, **C:\Squeak\JizdyGui3**, **Changes go to "Unnamed1"**, **Recent submissions -- youngest fir**, **Process Browser**, **Preferences**, **SmallLint**, **Code finder**, and **SmaCCParserGenerator: ? / ?**.

At the bottom, there are two buttons: **Navigator** and **Widgets Supplies**.

# From 3-Lisp to Metaobject Protocols.

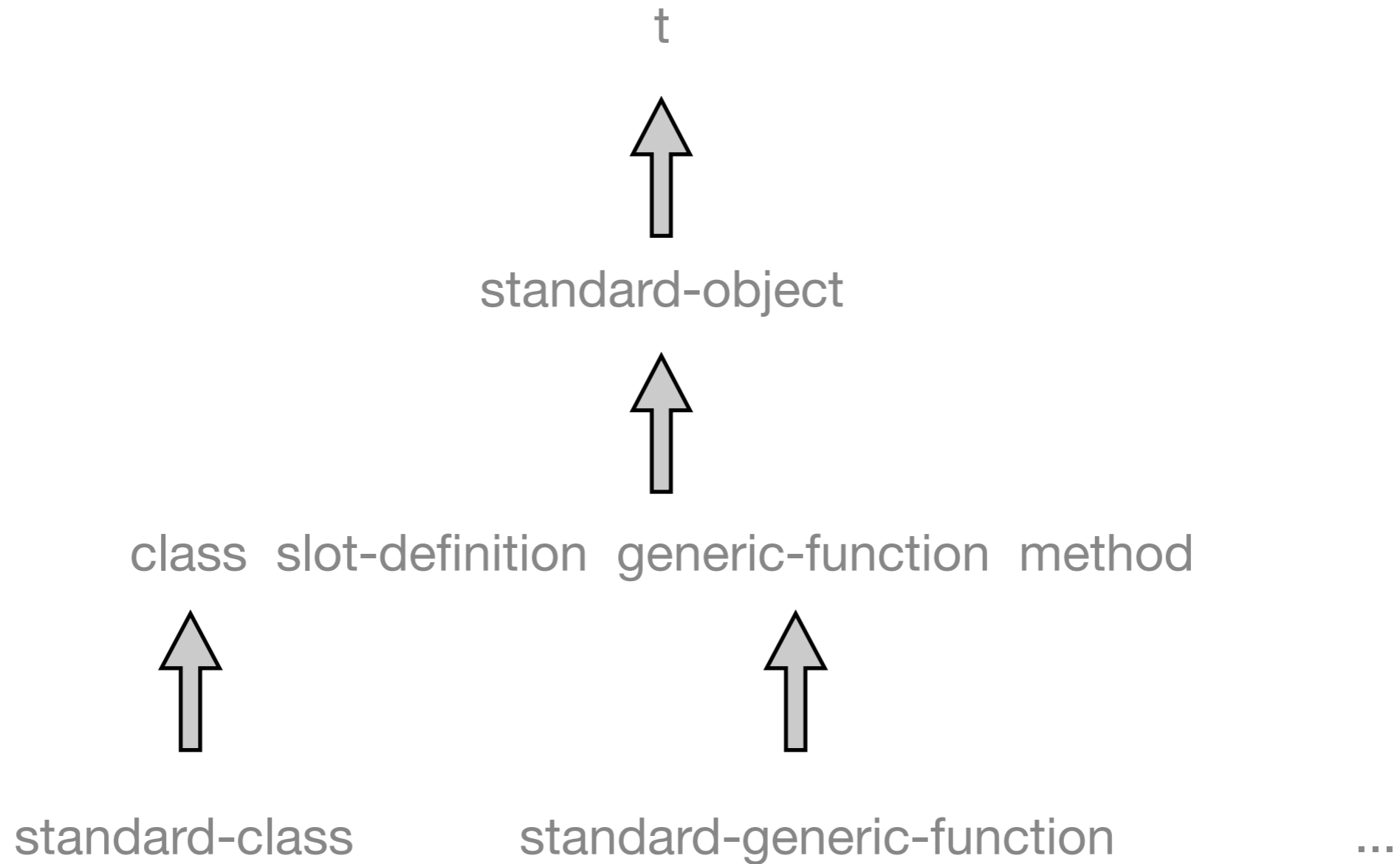
---

- 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?



# Hierarchy for metaobject classes.

---



# Separation between base and meta level.

---

- 3-Lisp: nlambda shifts up, eval shifts down
- Lisp macros: quote shifts up, replacing the original code shifts down.
- CLOS: class-of shifts up, returning from meta-level code shifts down.

# 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 ...))
```

# Limitations.

---

- Macros are not first-class entities at runtime.
- The CLOS MOP does not provide interception of argument evaluation.
- That's why in our delay/force example, delay must be implemented as a macro, and delayed-function and forced-function must be implemented as generic function classes.
- Note: Whenever an approach is simplified, expressive power might get lost!

# 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.

# 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>
- Paul Graham, The Roots of Lisp.  
<http://www.paulgraham.com/rootsoflisp.html>
- 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 Rivières, Control-related meta-level facilities in LISP. In P. Maes and D. Nardi, editors, *Meta-Level Architectures 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.