1. Reflective Programming and Open Implementations

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Winter Semester 2000-2001

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

Goal of this Lecture

You will learn about

- Open Implementations
- Reflection: Intercession and Introspection
- Reflective Architectures and Kernels (SOM, Smalltalk, CLOS)
- Meta Object Protocol: Powering End-Users
- Metaclasses
- Message Passing Control

Side Effects

- Program with a reflective system
- Let you implement your own micro kernel
- Deeply understanding OO
- Experiment with different OO models

Outline of the Lecture

- (C) Introduction, Concepts, Definitions, Examples, Meta Object Protocol, Open Implementations
- ok (C) The Study of an Object-Oriented Reflective Kernel (ObjVLisp)
- ok(Lab) ObjVLisp Implementation (1)
- ok(Lab) ObjVLisp Implementation (2)
- ok check @@(C) Metaclass Composition Issues
- ok(Lab) Metaclass Programming with ObjVlisp
-
- ->~(C) Analysing CLOS and its MOP
- You : (C) Reflection in OO Languages (Clos, Smalltalk, Java Comparison)
- ->>>(Lab) Interface Browser
- ->>>(C) Message Passing Control in Smalltalk
- (Lab) Implementing Actalk
- >>>You : (C) Presentation of papers
- (Lab) Scaffolding Patterns

What we could have made...

- MetaCircularity and Infinite Tower: Lisp in Lisp
- Different reflective paradigm (relational, actors...)
  - ☞ We will focus on OO reflective programming

History,Concepts,Definitions and Examples

Why Do We Need Reflective Programming?

```
>Does anyone know why CLOS does not provide a copy protocol?
>Has anybody implemented an inheritance method “a la Eiffel”?
>We need a method dispatch that take into account an external context of execution?
```

[Tutorial MOP & OI OOPSLA’93]

Some problems:

- data structure allocation, optimization
- control of language entities (feedback, trace, analysis...)
- UI and API definition
- language semantics

In summary

- Optimization
- Language extensions (control, debugging)
- Semantics change

Why Do We Need Reflective Programming?

"As a programming language becomes higher and higher level, its implementation in terms of underlying machine involves more and more tradeoffs, on the part of the implementor, about what cases to optimize at the expense of waht other cases.... the ability to *cleanly* integrate something outside of the language's scope becomes more and more limited" [Kiczales'92a]

- Why instances do have to have the same internal representation?
  - for Point => maximum speed needed, all instance variables used
  - ☞ array like representation
  - for Person => minimize space, few instances used
  - ☞ hash-table like representation
- Why can't I control internal representation or attribute accesses?
- Why can't I query the language representation instead of scanning, parsing code?
- Why can't we tune a language to fit our needs from the language itself and not by inventing yet a new language or rebuilding a dedicaced compiler ?

Traditional vs Reflective Answers

Traditional Answers at Language Level:

- Illusionary complete language
- Library of extensions: Eiffel
- Macroes: C, Lisp

But do not cover language extensions or semantics changes

Traditional Answers in Software development:

What happens if the language does not support our need [Kiczales92,92b,92c]?

- buy a new one that fits your today need and change tomorow!
- buy an illusionary complete language
- code between the lines (danger for portability)
- create your own layer (probleme with integration issues)

Reflective Answers

- Propose an extensible language or system
- Give the power to the end-user (meta-programmer)
  - customize your reflective or open language to fit your need

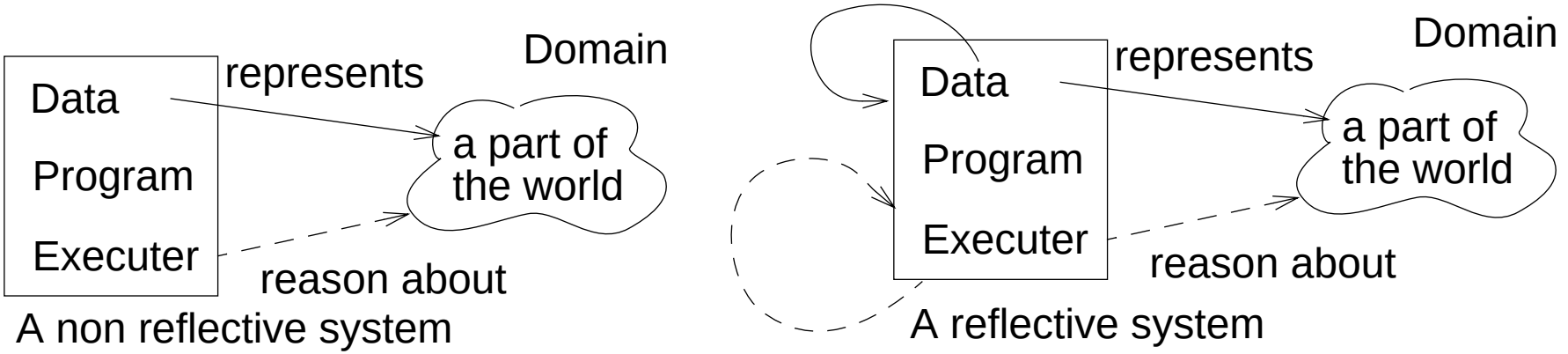
Role ot Reflective Prog in Software Engineering

- Allow migration of software:
  Ex: Nichimen Corp (http://www.nichimen.com/) 15 years of Lisp development
  (Flavors -> CLOS, From Symbolic Machine -> Indigo Silicon Graphics)
- Adaptation to new technologies
- Adaptation to new needs

Team organization

- Not everybody is changing the language semantics or introducing his own constructs
- One meta-programmer implements new semantics or adapts language semantics to the needs of the other developers

Definitions (I)



Reflection: a process's integral ability to represent, operate on, otherwise deal with itself in the same way that it represents, operates on and deals with its primary subject matter.

B.C Smith (OOPSLA' 90 Workshop on Reflection and MetaLevel Architectures)

"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: *introspection* and *intercession*.

Introspection is the ability for a program to observe and therefore reason about its own state.

Intercessory is the ability for a program to modify its own execution state or alter its own interpretation or meaning.

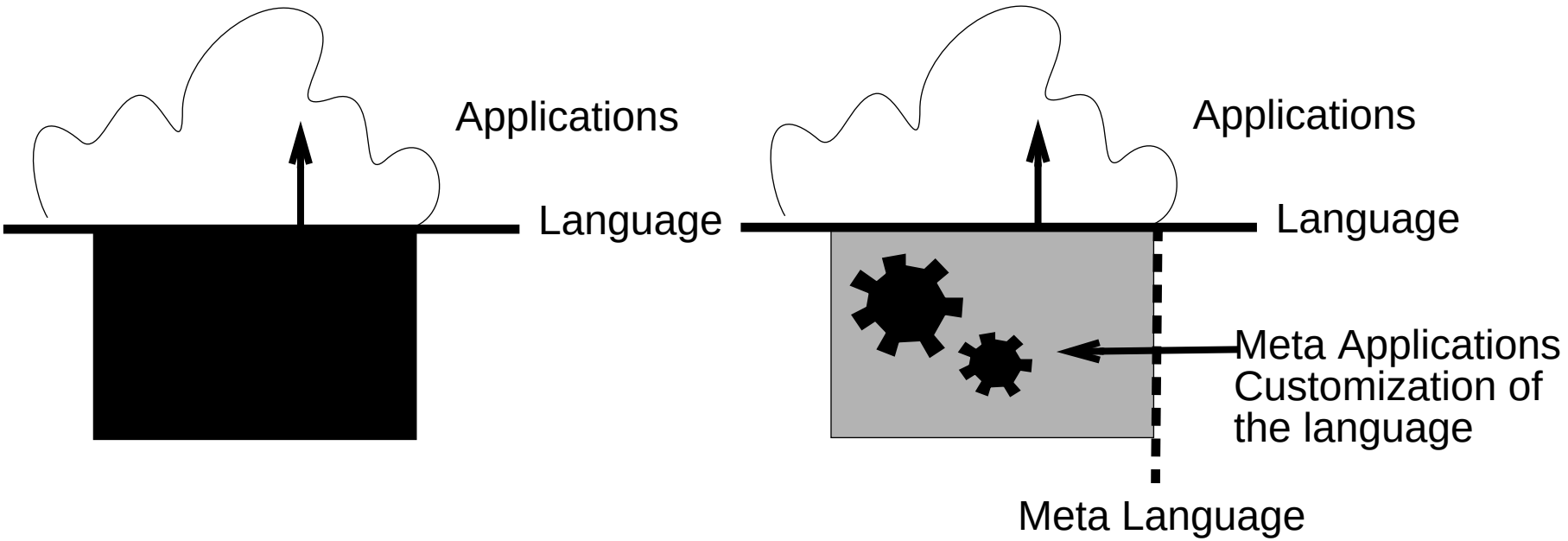
Both aspects require a mechanism for encoding execution state as data: providing such an encoding is called *reification*." [Bobrow, Gabriel and White in Paepke'92]

Consequences

A system having itself as application domain and that is *causally* connected with this domain can be qualified as a reflective system [Pattie Maes]

- A reflective system has an internal representation of itself.
- A reflective system is able to act on itself with the ensurance that its representation will be causally connected (up to date).
- A reflective system has some static capacity of self-representation and dynamic self-modification in constant synchronization
- A system is said reflective if it has an introspection protocol and an intercessory protocol

Meta Programming in Programming Language Context



The meta-language and the language can be different: Scheme and an OO language

The meta-language and the language can be same: CLOS

=> metacircular architecture

Three Approaches

1. Tower of Metacircular Interpreters
   - ☞ every level is interpreting and controlling the next level

   ex: 3-Lisp, SRI

2. Meta entities control language entities

   ex: Smalltalk, CLOS, FOL, Meta-Prolog, ...
   ABCL/R, ACT/R (Concurrent languages)
   meta-rules controlling unification in prolog

3. Open Implementation
   - ☞ The implementation specifies some entry points allowing the future modification of the system. (often based on meta entities)

   ex: CLOS MOP (Meta Object Protocol)

Infinite Tower of (Meta)Interpreters

- 3-Lisp: a metacircular interpreter that can evaluate itself
- Scheme like based on continuations
- Theory, Basis for reflection
- Experimentation with language extension, various semantics

Passing from one level to another one is done using reifier
special functions with three non evaluated arguments

- current expression
- environment
- continuation

Interpreter 0 reifies and interpretes interpreter 1
Interpreter 1 reifies and interpretes interpreter 2....

Reflective Languages

CLOS, Smalltalk, Self

- Language written in itself
- MetaEntities controlling the languages (Class, Method, InstanceVariables...)
- Really powerfull, full control

In Smalltalk

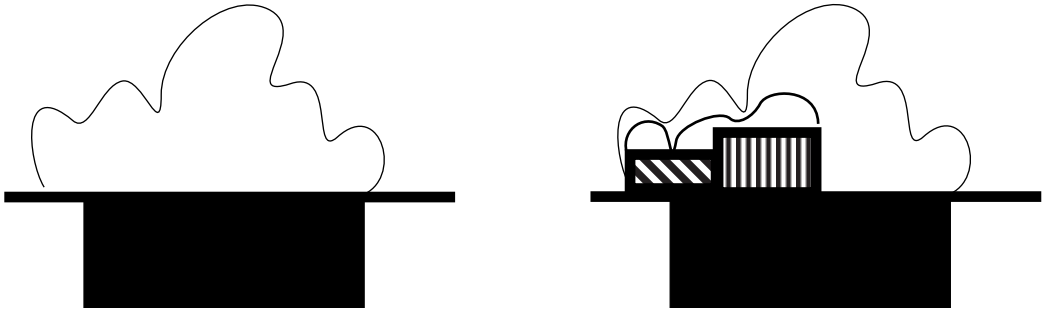
- everything is an object
- causally connected: a change in an object impact the semantics
  - ☞ Class, Method
  - ☞ Scanner, Parser, Compiler, Decompiler, ...
  - ☞ Scheduler, Process, Semaphore

But

- Did not make the effort of specifying a Meta Object Protocol
  - – Too much to do for the base programmer
  - – Not enough freedom to optimize for the language implementor
  - ☞ A solution: declarative model of the base level language and the meta level

Open Implementation and MOPs

The Basic Claim of Open Implementation



It is **impossible to hide** all implementation issues behing a module interface because not all of them are **details**. Instead, some involve **crucial** strategy issues that inevitably bias the performance of the resulting implementation. We call these issues *strategy dilemmas*, because they involve a choice about how to implement a higher-level functionality in terms of a lower level one. Strategy dilemmas can be broken down into *resource allocation dilemmas* where the implementor has to decide how much of a shared resource to allocate to each client, and *mapping dilemmas*, where the implementor has to decide how to map the functionality ther are implementing onto the lower-level functionality.

Despite black-box abstraction's appealing goal that a module should present a simple interface that exposes only functionality, there is a great deal more to a module than acknowleged by that interface.

Our claim is that module implementations must somehow be opened up to allow clients control over these issues as well. We call this the need for open implementations. From http://www.xerox.../oi/ (@@)

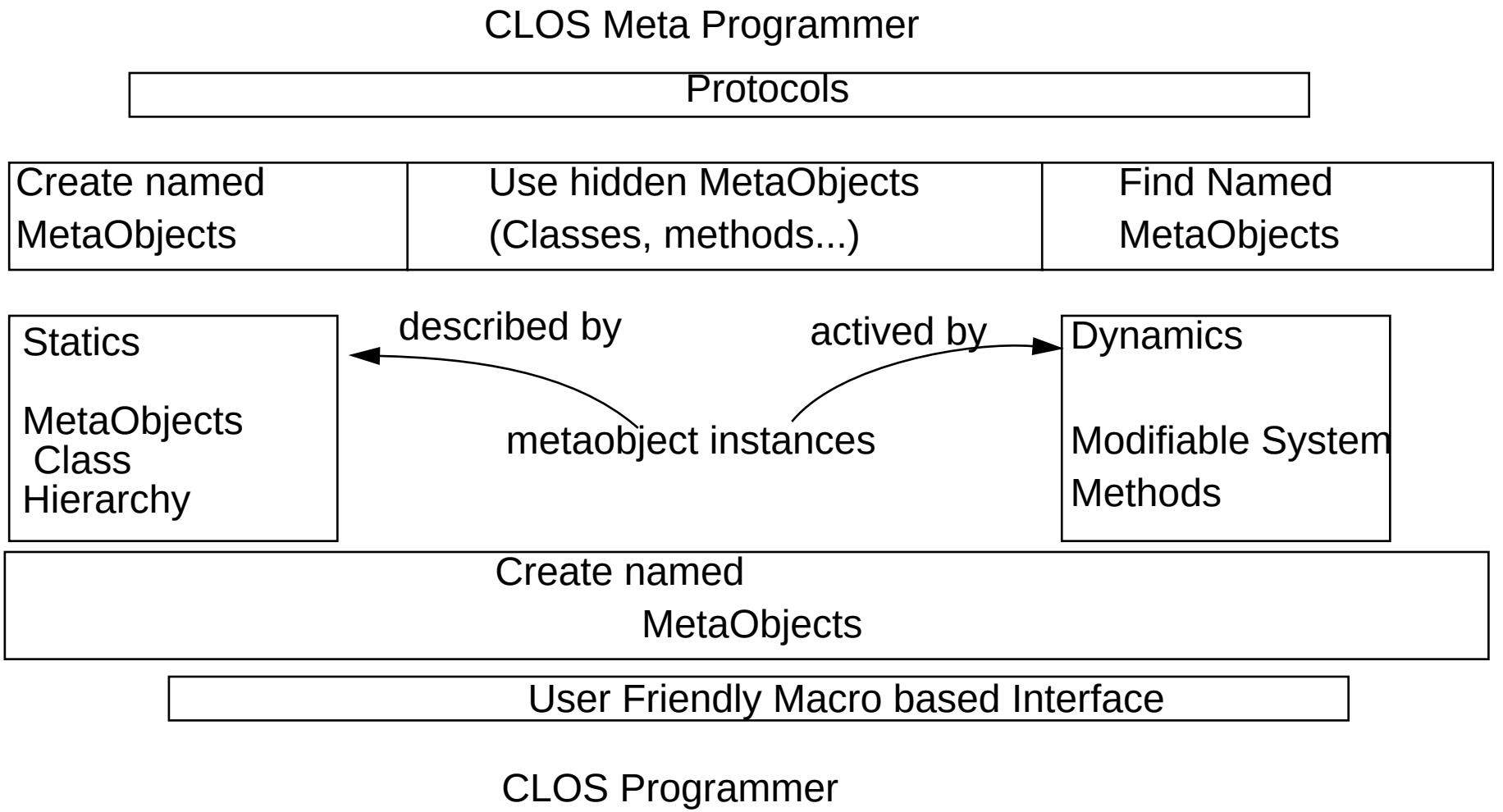
Meta Object Protocols

"Meta Object Protocols are interfaces to the language that give the users the ability to incrementally modify the language's behavior (semantics) and implementation, as well as the ability to write the programs with the language" [Paepcke'92]

- MOPs are composed by set of entry points whose specialization allows the introduction of new behavior.
- MOPs are based on meta-objects offering ways of specializing their behavior and representing specific aspects of the base level

Public MetaLevel Architecture (structure and static)	+	Public protocols (dynamic)	=	Implementation
Inspect	+	Modify	=	Open System

Meta Programming in CLOS



Infinite Tower vs Open Implementation

Infinite Tower vs Mop <=> Theory vs Practice

Open Implementations:

- are more efficient
- are specified declaratevly letting space of optimization
- define a region of possible changes
- dependencies between entry points
- allow more control over the possible extensions

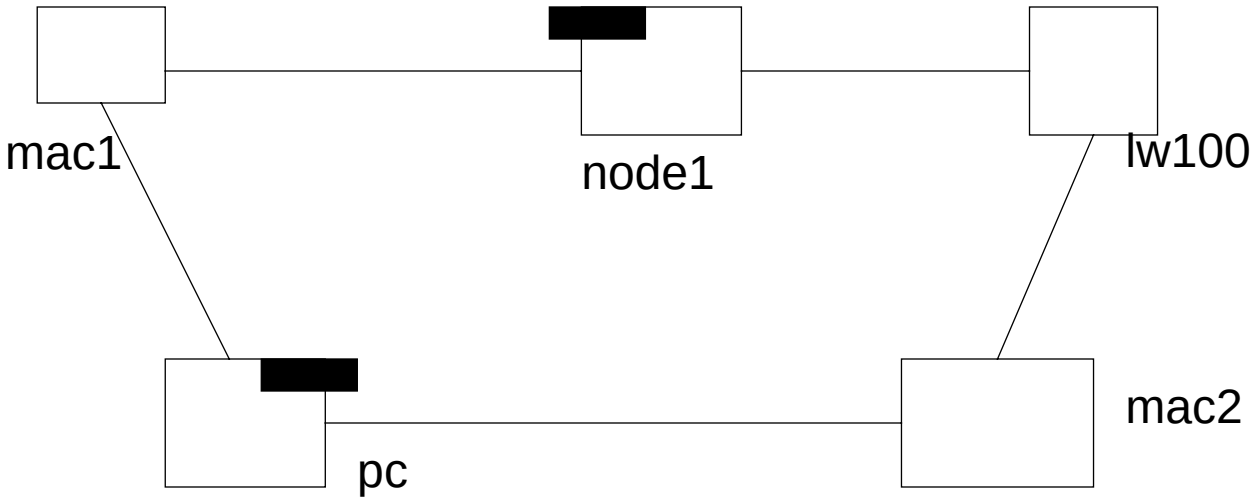
Infinite Tower:

- are more powerful
- slower
- less secure

A Simple Application as Example

A LAN Simulator:

- A LAN contains nodes, workstations, printers, file servers.
- Packets are sent in a LAN and the nodes treat them differently.



Problem: We want to know all the nodes of the system for analysis purpose

- ❑ We do not want to change the code of the node classes.
- ❑ We would like to ask to the class Node to gave us all its instances.

Programming in Explicit Metaclass Context

CLOS-like

```
(defclass Node ()
   ((name :initarg :name :default-value #lulu :reader name)
    (nextNode :default-value '() :accessor nextNode))
   (:metaclass Set))

(defmethod accept ((n Node) (p Packet))
   ....)
(defmethod send ((n Node) (p Packet))
   ...)

(setq n1 (make-instance Node :name "n1"))
(setq n2 (make-instance Node :name "n2" nextNode: n1))
(setq n3 (make-instance Node :name "n3" nextNode: n3))
((setf nextNode) n1 n3)

(allInstances Node)
-> (n1 n2 n3)
```

Reusing Meta Programs

Now imagine that we want to have a log of all the created packets

```
(defclass Packet ()
   ((addressee :initarg :addressee :accessor addressee)
    (contents :initarg :constents :accessor contents)
    (originator :initarg: originator :accessor originator)
   (:metaclass Set))

(defmethod isAddressedTo ((p Packet) (n Node))
   ....)
(defmethod isOriginatedFrom((p Packet) (n Node))
   ...)

(map Packet (lambda(x)
              (write outputstream
                     “packet addressed from: %s to %s”
                     (originator x) (addressee x))
```

MetaProgramming in OO Context

This simple functionality could have been implemented in C++ or Java defining static member and functions [Singleton Pattern]

But

- A Meta program is not mixed into objects
- Ordinary objects are used to model real world. Metaobjects describe these ordinary objects.
- MetaPrograms can be reused
- Some other properties cannot easily be implemented without meta programming traceMessage, finalClass, PrePostConditions, DynamicIVs, MessageCounting....

We may want to

- change the representation of the instance variables
(indexed for points, hashed for person,)
- change the way attributes are accessed (lazily via the net, stored in database)
- change the inheritance semantics
- change the invocation of method semantics (trace, proxies...)

MetaProgramming by Example

```
(defclass Set (class)
   ((instances :default-value '() :reader allInstances)))

(defmethod clear ((c Set))
   (setf-slot-value c 'instances '()))

(defmethod map ((c Set) fct)
   (map fct (allInstances c)))

(defmethod new ((c Set) initarg)
   (let ((newInstance (call-next-method))
      (cons newInstance (slot-value c 'instances))
      newInstance))
```

Costs of Reflective Programming

Design Cost

Reflective languages need more care and iteration

Use Cost

Concepts are more complex

Run-time Cost

“A key aspect of intercession is that reflective capability not impose an excessive performance burden simply to provide for the possibility of intercession. What is not used should not affect the cost of what is used; and the common case should retain the possibility of being optimised” [Bobrow, Gabriel and White in Paepke’92]

Clever implementations

- we only pay what we need, but we NEED it!
- Default behavior is optimized
- Do no rely on full runtime interpretation
  - Having entry point purely functional (same argument gives same result)
  - Optimization at compile-time
  - Memoization (decomposing static from dynamic entry point)

Designing Reflective Systems

- Which model
  - which kind of language?
  - which degre of reflection?
  - reflective language or open implementation?
- Which entry points?
  - Data, Entities, Control Structures, Interpreter, Environment
- Data Structure
  - simples, efficient, easliy modifiable
- Changing Level
  - Managing causal connection, reification and reflexion
- Uniformity between meta-level
  - Syntax, data structure, extensions

Meta-Problems

- Stability: Potentially an end-user can change the system
  - ☞ But not everybody should be meta-programming
- Several levels of complexity
  - ☞ Entity, meta entity, coherence and connection between levels
- Uniformity: same design conception problems than the original designer
  - ☞ Open implementations narrow the possibilities of change

Meta and Open are not Limited to Programming Languages

A reflective system is a system which incorporates structures representing (aspects of) itself.
Reflection is the processus of reasoning about and/or acting upon itself.

P.Maes (OOPSLA' 87)

- Network
- Workflow system
- Operating Systems (Apertos, Synthesis)
- Parallel Systems
- Library of

2. The Study of a Minimal Object-Oriented Reflective Kernel

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

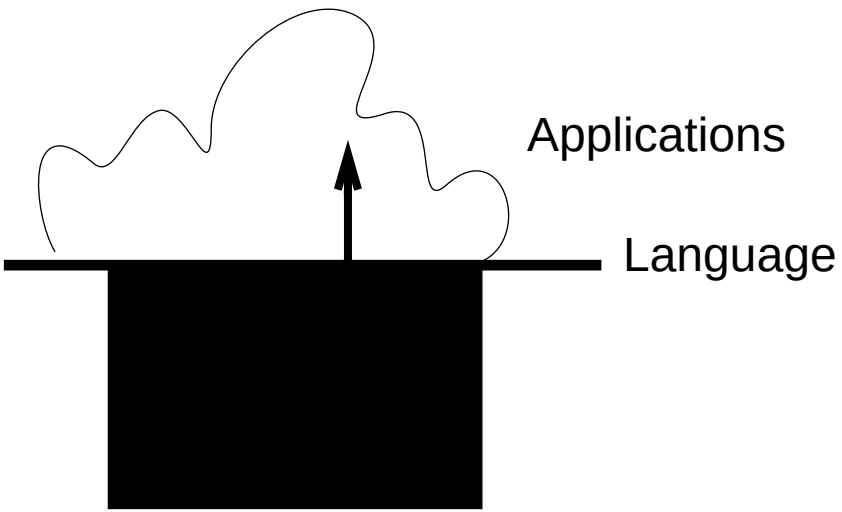
Goals of this Lecture

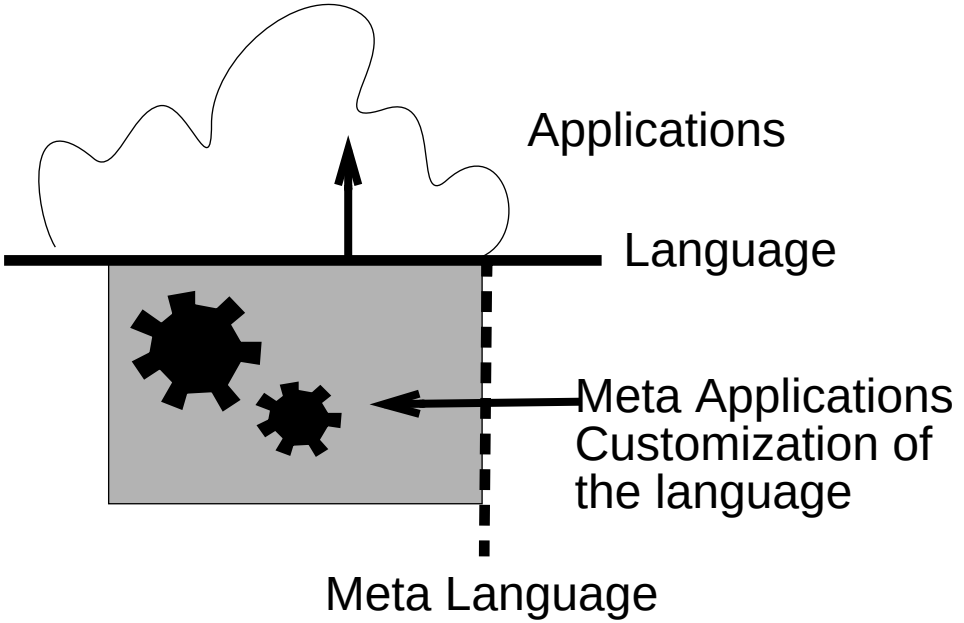
- Metaclass concept
- Reflective Architectures and Kernels (SOM, Smalltalk, CLOS)
- What are Object and Class classes?
- Semantics of inheritance, semantics of super
- Metaclass power

Outline

- ☞ Metaclasses?
- ❑ Examples of usefull metaclasses
- ❑ Towards a unified approach: Loops, Smalltalk
- ❑ ObjVlisp in 5 postulates
- ❑ Instance Structure and Behavior
- ❑ Class Structure
- ❑ Message Passing
- ❑ Object allocation & Initialization
- ❑ Class creation
- ❑ Inheritance Semantics
- ❑ Bootstrapping
- ❑ Examples: Playing with ObjVlisp

Recall: Meta Programming in Programming Language Context





Class as Objects

"The difference between classes and objects has been repeatedly emphasized. In the view presented here, these concepts belong to different worlds: the program text only contains classes; at run-time, only objects exist. This is not the only approach. **One of the subcultures of object-oriented programming, influenced by Lisp and exemplified by Smalltalk, views classes as object themselves, which still have an existence at run-time.**"

Bertrand Meyer in Object-Oriented Software Construction

Some Class Properties

- Abstract: a class cannot have any instance
- Set: a class that knows all its instances
- DynamicIVs: Lazy allocation of instance structure
- LazyAccess: only fetch the value if needed
- AutomaticAccessor: a class that defines automatically its accessors
- Released/Final: Class cannot be changed and subclassed
- Limited/Singleton: a class can only have a certain number of instances
- IndexedIVs: Instances have indexed instance variables
- InterfaceImplementor: class must implement some interfaces
- MultipleInheritance: a class can have multiple superclasses
- Trace: Logs attribute accesses, allocation frequencies
- ExternalIVs: Instance variables stored into database

Some Method based Properties

- Trace: Logs method calls
- PrePostConditions: methods with pre/post conditions
- MessageCounting: Counts the number of times a method is called
- BreakPoint: some methods are not run
- FinalMethods: Methods that cannot be specialized

Metaclass Responsibilities

"Metaclasses provide metatools to build open-ended architecture" [Cointe'87]

Metaclasses are one of the possible meta-entities (method, instance variables, method combination,...)

Metaclasses allow the structural extension of the language

They may control

- Inheritance
- Internal representation of the objects (listes, vecteurs, hash-table,...)
- Method access ("caches" possibility)
- Instance variable access

Separation of Concerns

- Ordinary objects are used to model real world
- Metaobjects describe these ordinary objects
- Meta/Base level functionality is not mixed

Outline

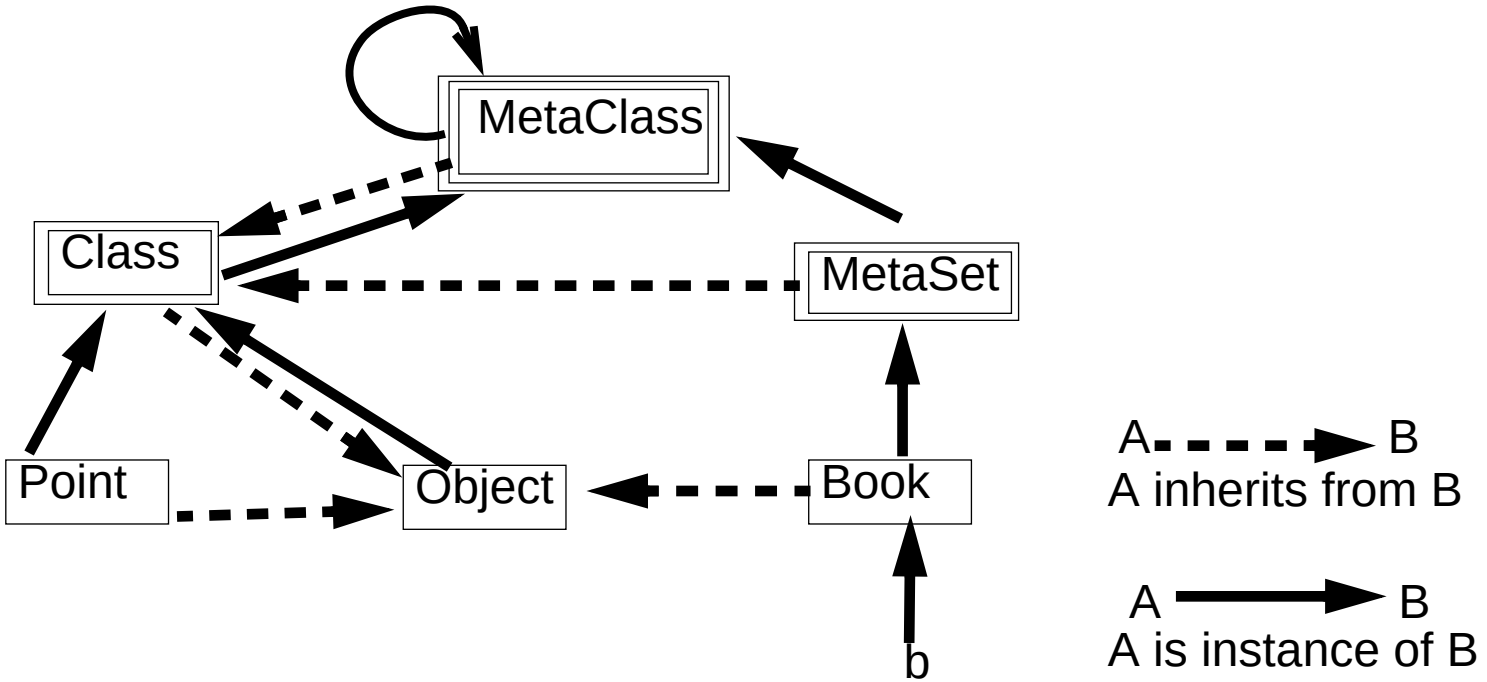
- Metaclasses?
- Examples of usefull metaclasses
  - ☞ Towards a unified approach: Loops, Smalltalk
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping
- Examples: Playing with ObjVlisp

Why ObjVlisp?

- Minimal (only two classes)
- Reflective: ObjVlisp self-described: definition of Object and Class
- Unified: Only one kind of object: a class is an object and a metaclass is a class that creates classes
- Open
- Simple: can be implemented with less than 300 lines of Scheme or 30 Smalltalk methods.
- Equivalent of Closette (Art of MOP example)
- Really good for understanding dynamic languages and reflective programming (D-SOM, CLOS, Smalltalk kernel)

The Loops Approach

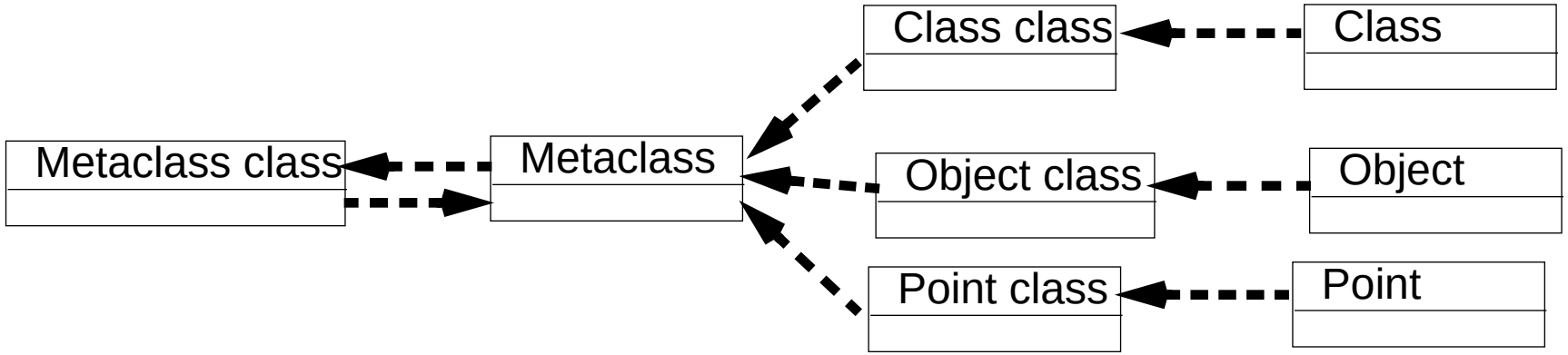
"For some special cases, the user may want ot have more control over the creation of instances. For example, Loops itself uses different Lisp data types to represent classes and instances. The new message for classes is fielded by their metaclass, usually the object MetaClass." [Bobrow83]



- Explict metaclass as a subclass of another but must be instance of MetaClass

The Smalltalk Pragmatical Approach

"The primary role of a metaclass in the Smalltalk-80 system is to provide protocol for initializing class variables and for creating initialized instances of the metaclasse'sole instance" [Goldberg84]



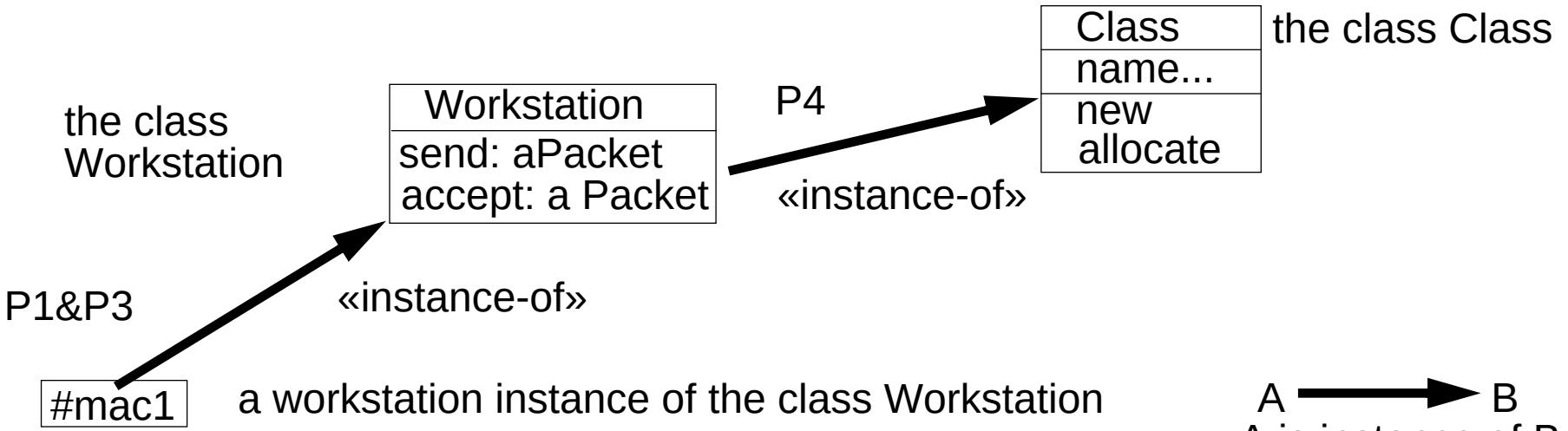
- A class is the sole instance of a metaclass
- Every metaclass is an instance of the `Metaclass` class
  - metaclasses are not true classes
  - number of metalevels is fixed
- Metaclass hierarchy inheritance is fixed: parallel to the class inheritance
  - dichotomy between classes defined by the user (instance of Class) and metaclasses defined by the system (instance of metaclasses)

ObjVlisp in 5 Postulates (i)

P1: object = <data, behavior>

P3: Every object belongs to a class that specifies its data (slots or instance variables) and its behavior. Objects are created dynamically from their class.

P4: Following P3, a class is also an object therefore instance of another class its metaclass (that describes the behavior of a class).



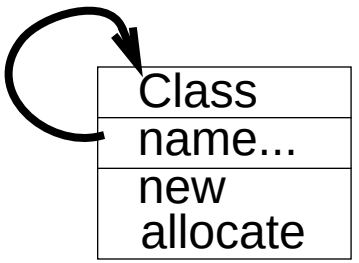

```
|mac1|
mac := Workstation new name: #mac1
```

How to Stop Infinite Recursion?

Aclass is an object therefore instance of another class its metaclass that is an object too instance of a metametaclass that is an object too instance of another a metametametaclass.....

To stop this potential infinite recursion

- `Class` is the initial class and metaclass
- `Class` is instance of itself and
- all other metaclasses are instances of `Class`.



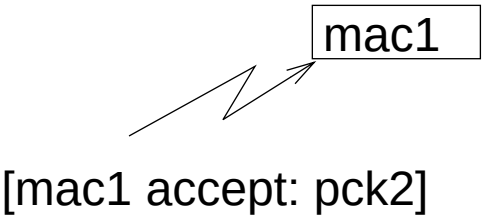
the class Class

A is instance of B

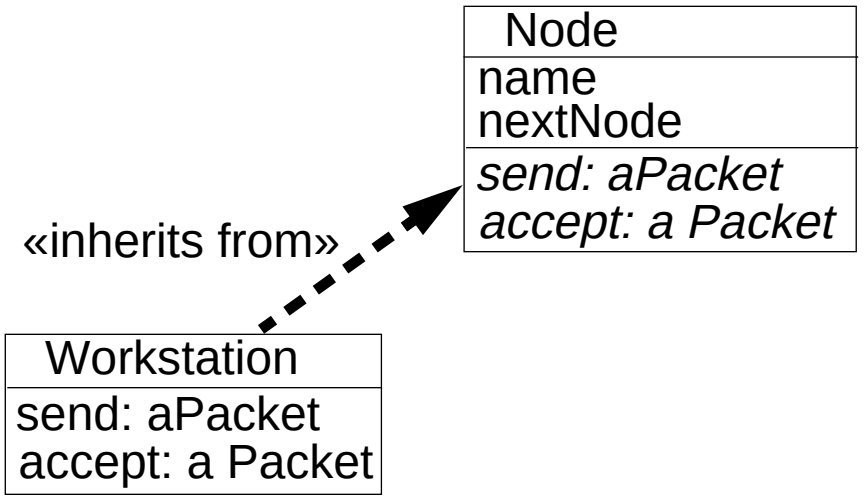
ObjVlisp in 5 Postulates (ii)

P2: Message passing is the only means to activate an object

[object selector args]



P5: A class can be defined as a subclass of one or many other classes. This mechanism is called inheritance. It allows the sharing of instance instance variable and methods. The class Object represents the behavior shared by all the objects.



Unification between Classes and Instances

"We claim that a class must be an object defined by a real class allowing a greater clarity and expressive power" [Cointe'87]

- Every object is instance of a class
- A class is an object instance of a metaclass (P4)
  - ☞ But all the objects are not classes

- Only one kind of objects without distinction between classes and final instances.
- Sole difference is the ability to respond to the creation message: new. Only a class knows how to deal with it.
- A metaclass is only a class that generates classes.

About the 6th ObjVlisp's Postulate

"Ordinary objects are used to model real world. Metaobjects describe these ordinary objects" [Rivard 96]

The ObjVlisp 6th postulate is:

class variable of anObject =instance variable of anObject's class

So class variables are shared by all the instances of a class.

We disagree with it.

- Semantically class variables are not instance variables of object'class!
- Instance variable of metaclass should represent class information not instance information.

Metaclass information should represent classes not domain objects

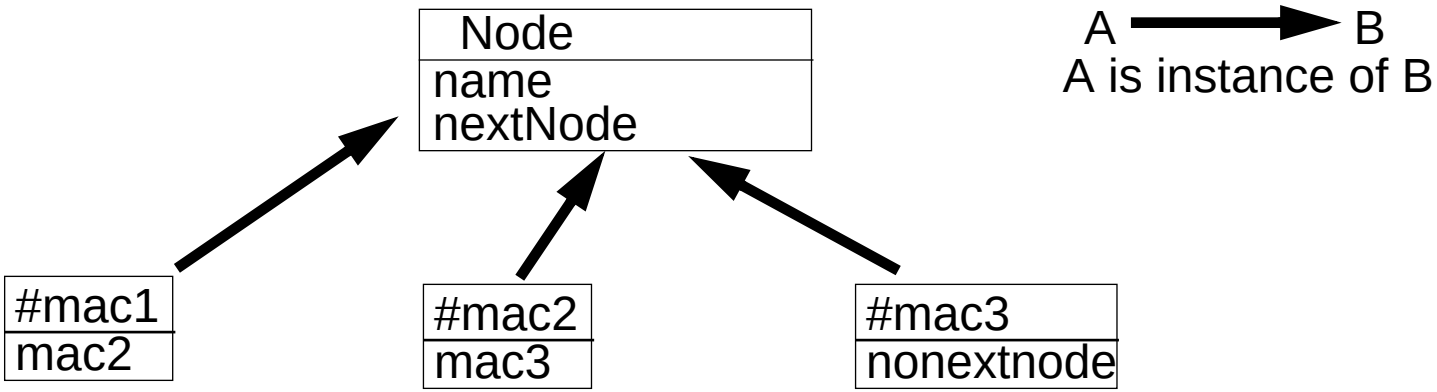
CLOS offers the :class instance variable qualifier class variables.

We could imagine that a class possesses an instance variable that stores structure that represents shared-variable and their values.

Instance Structure: Instance Variables

Instance variables:

- an ordered sequence of instance variables defined by a class
- shared by all its instances
- values specific to each instance



In particular, every object possesses an instance variable class (inherited from Object) that points to its class.

Instance Behavior: Methods

A method

- belongs to a class
- defines the behavior of all the instances of the class
- is stored into a dictionary that associates a key (the method selector) and the method body

To unify instances and classes, the method dictionary of a class is the value of the instance variable `methodDict` defined on the metaclass `Class`.

Class as an Object: Structure

- Considered as an object, a class possesses an instance variable class inherited from `Object` that refers to its class (here to the metaclass that creates it).
  - `class` an identifier of the class of the instance
- But as an instance factory the metaclass `Class` possesses 4 instance variables that describe a class:
  - `name` the class name
  - `super` its superclass (we limit to single inheritance)
  - `i-v` the list of its instance variables
  - `methodDict` a method dictionary

Example: class Node

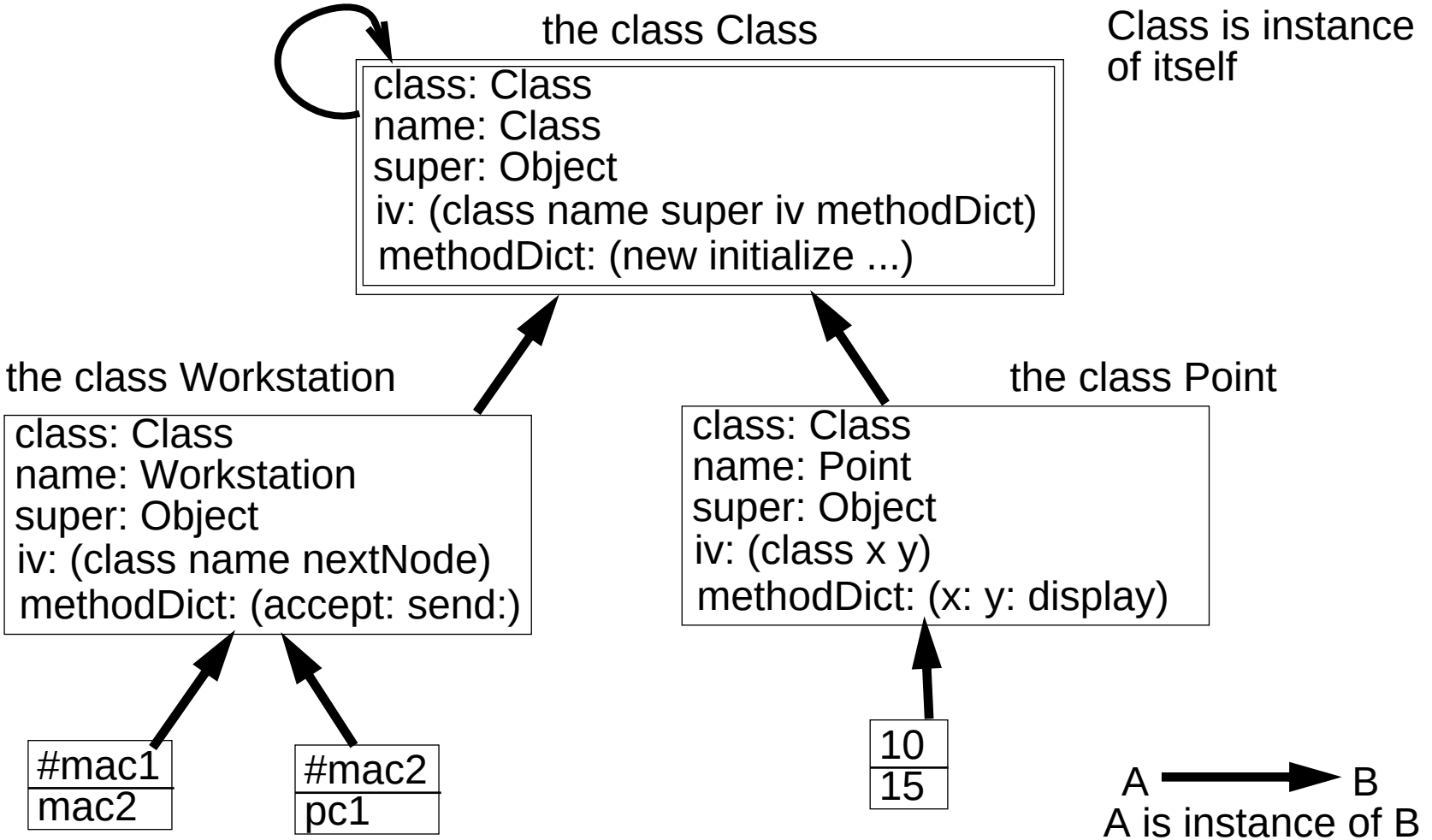
class:	Class	instance of Class
name:	Node	named Node
super:	Object	inherits from Object
i-v:	(name nextNode)	defines 2 instance variables
methods:	.....	defines methods

The class Class: a Reflective class

- Initial metaclass
- Defines the behavior of all the metaclasses
- Instance of itself to avoid an infinite regression

class:	Class	instance of Class
name:	Class	named Class
super:	Object	inherits from Object
i-v:	(name supers i-v methodDict)	describes any class
methods:	(new allocate initialize.....	behavior of a class

A Complete Example



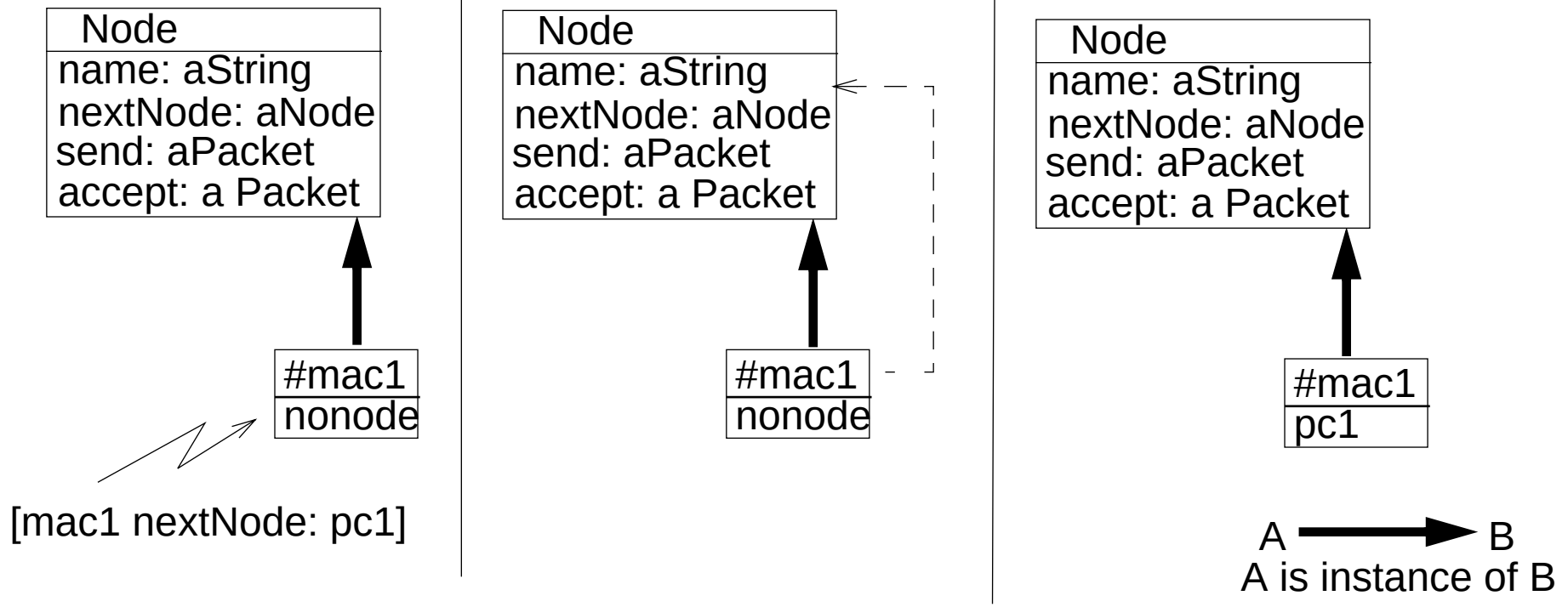
Outline

- Metaclasses?
- Examples of usefull metaclasses
- Towards a unified approach: Loops, Smalltalk
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
  - ☞ Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping
- Examples: Playing with ObjVlisp

Message Passing (i)

P2: Message passing is the only means to activate an object

P3: Every object belongs to a class that specifies its data and its behavior.



Message Passing (ii)

send message = apply O lookup

We **lookup** the method associated with the selector of the message **in the class** of the receiver then **we apply it to the receiver**.

[receiver selector args]
<=>
apply (found method starting from the class of the receiver)
on the receiver and the args
<=>
in functional style
(apply (lookup selector (class-of receiver) receiver)
receiver args)

Object Creation by Example

Creation of instances of the class Point

```
[Point new :x 24 :y 6]
[Point new]
[Point new :y 10 :y 15]
```

Creation of the class Point instance of Class

```
[Class new
        :name Point
        :super Object
        :i-v (x y)
        :methods (x ...
                display ...)
]
```

Object Creation: the Method new

Object Creation = initialisation O allocation

- Creating an instance is the composition of two actions:
  - memory allocation: `allocate` method
  - object intialisation: `initialize` method

(new aClass args) = (initialization (allocation aClass) args)

<=>

`[aClass new args]` = `[[aClass allocate] initialize args]`

- `new` creates an object: class or final instances
- `new` is a class method

Object Allocation

- Object allocation should return:
  - Object with empty instance variables
  - Object with an identifier to its class

- Done by the method `allocate` defined on the metaclass Class
- `allocate` method is a class method

example:

```
[Point allocate]  => #(Point nil nil)
   for x and y
[Workstation allocate] => #(Workstation nil nil)
   for name and nextNode
[Class allocate]  => #(Class nil nil nil....)
```

Object Initialization

- Initialization allows one to specify the value of the instance variables by means of keywords (:x ,:y) associated with the instances variables.

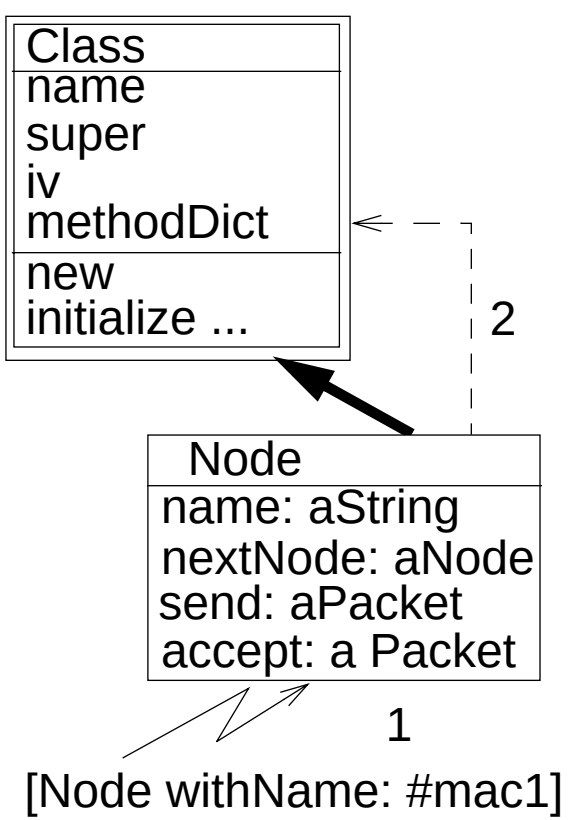
Example:

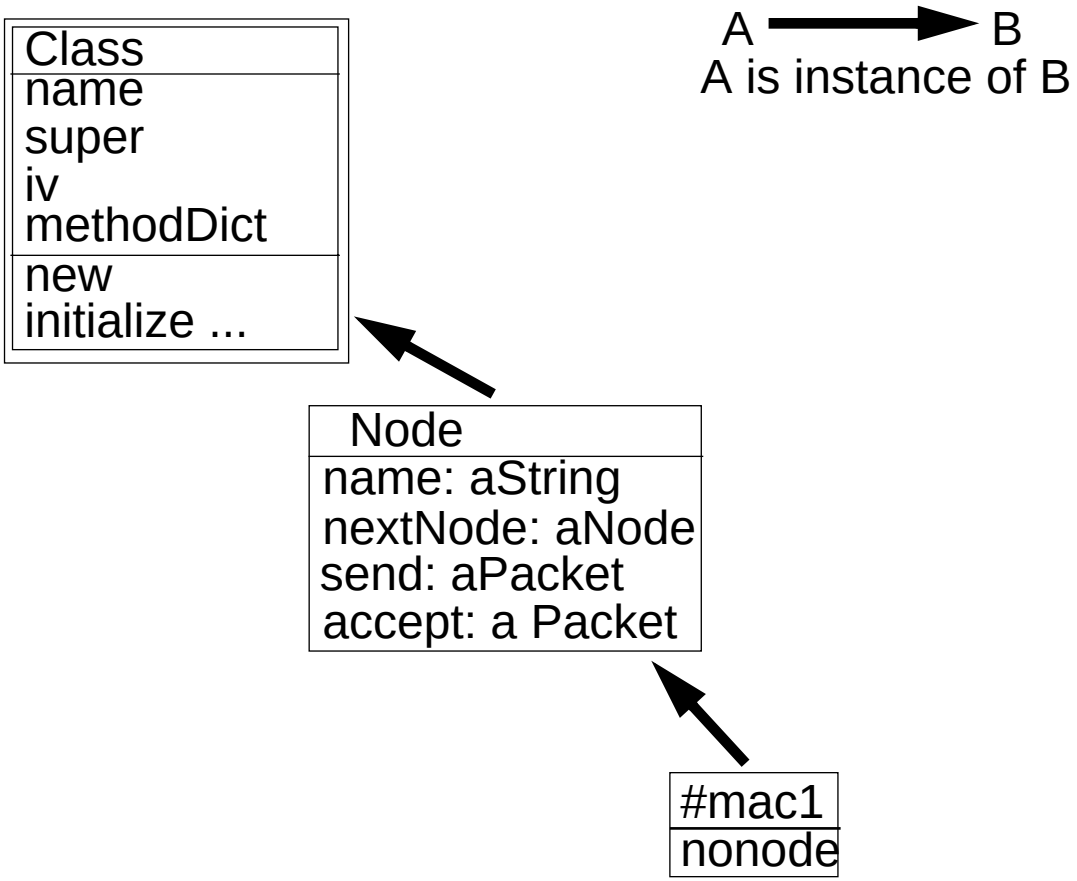
```
[ Point new :y 6  :x 24]
=> [ #(Point nil nil) initialize (:y 6 :x 24)]
==> #(Point 24 6)
```

- `initialize` : two steps
  - get the values specified during the creation. (y -> 6, x -> 24)
  - assign the values to the instance variables of the created object.

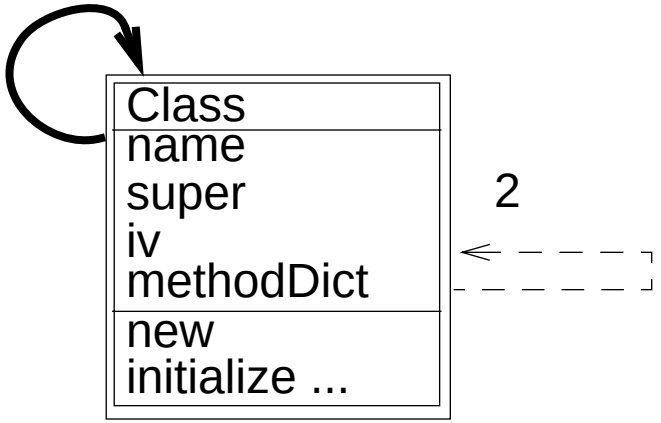
Object Creation: the Metaclass Role

We **lookup** the method associated with the selector of the message **in the class** of the receiver then **we apply it to the receiver**.

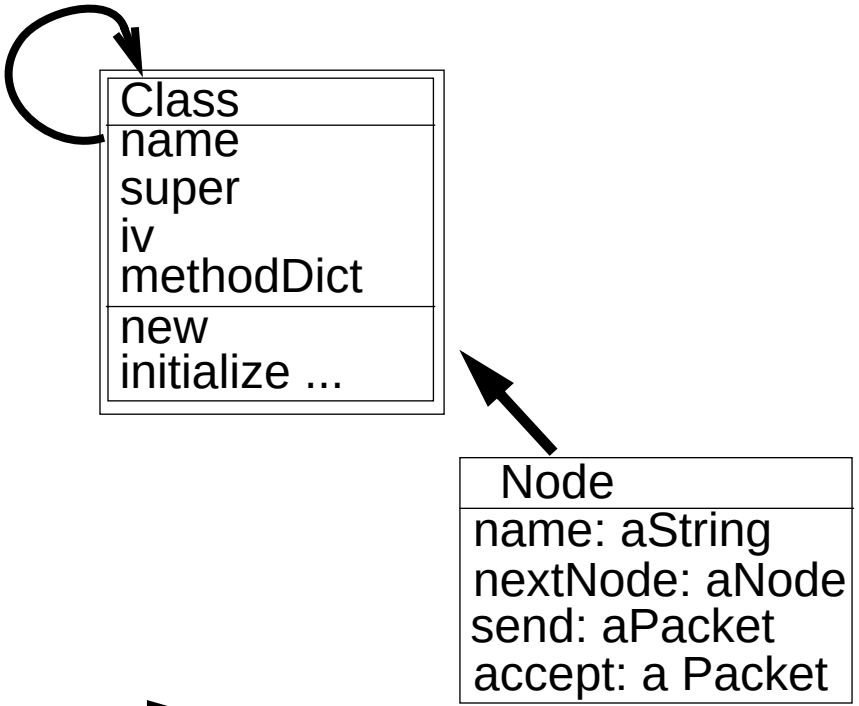




Class Creation

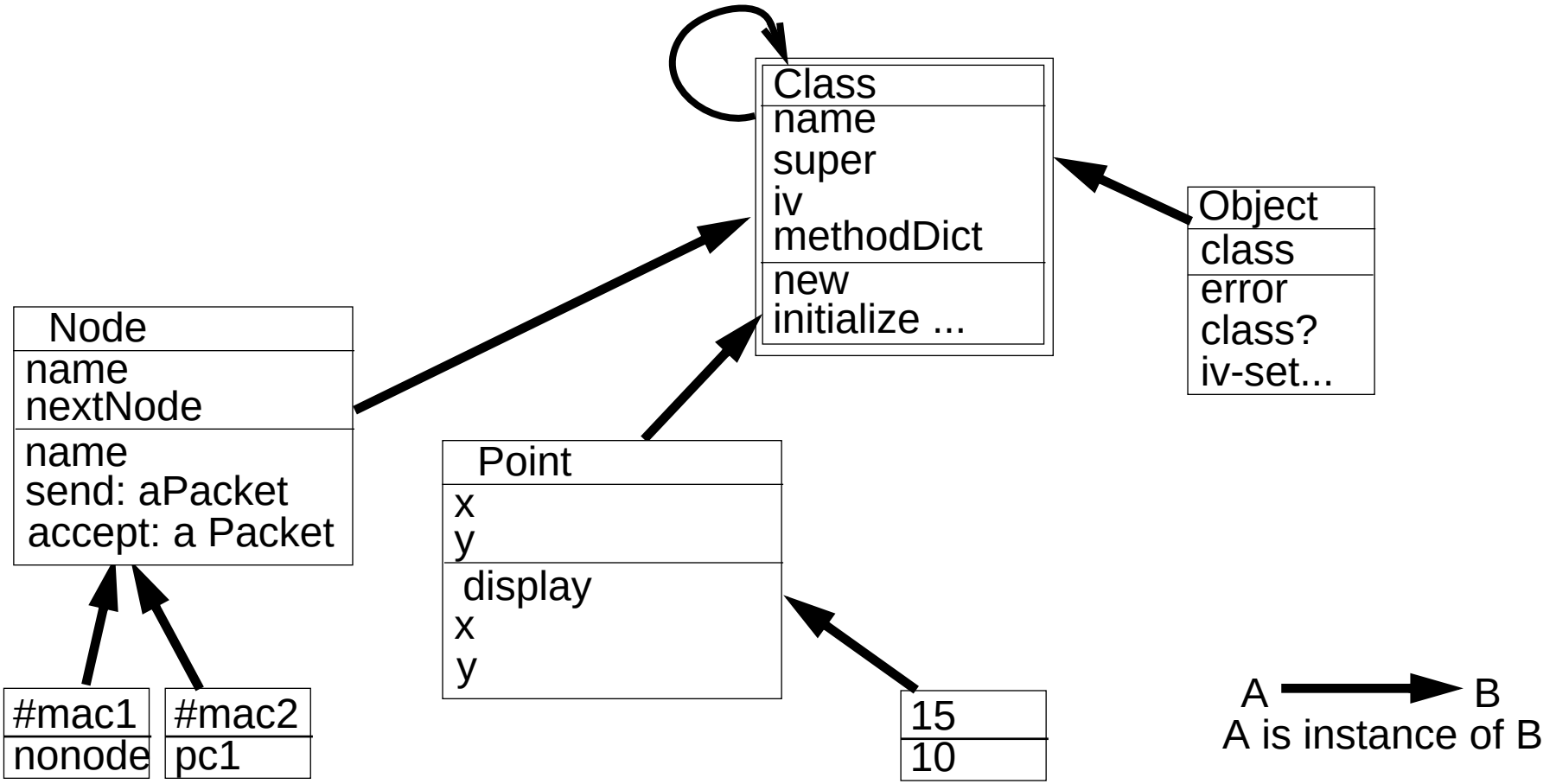



```
[Class new
        :name Node
        :supers Object
        :iv (name nextNode)
        :methods
        (send: ....))]
```



A → B
A is instance of B

A Simple Instantiation Graph



- `Class` is the root of instantiaton graph
- `Object` is a class that represents the minimal behavior of an object
- `Object` is a class so it is instance of `Class`

What is the minimal behavior shared by all the objects?

The class `Object` represents the common behavior shared by all the objects:

- ☞ classes
- ☞ final instances.

- ❑ every object knows its class: instance variable class (uses a primitive for accessing else that loops!)

- ❑ methods:
  - `initialize` (instance variable initialization)
  - `error`
  - `class`
  - `metaclass?`
  - `class?`

  Meta operations:
  - `iv-set`
  - `iv-ref`

Outline

- Metaclasses?
- Examples of usefull metaclasses
- Towards a unified approach: Loops, Smalltalk
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
  - ☞ Inheritance Semantics
- Bootstrapping
- Examples: Playing with ObjVlisp

Two Forms of Inheritance

- Static for the instances variables
  - ☞ Done once at the class creation
  - ☞ When C is created, its instances variables are the union of the instance variables of its superclass with the instance variables defined in C.

final-instance-variables (C) =
union (union (iv (super C)), local-instance-variables(C))

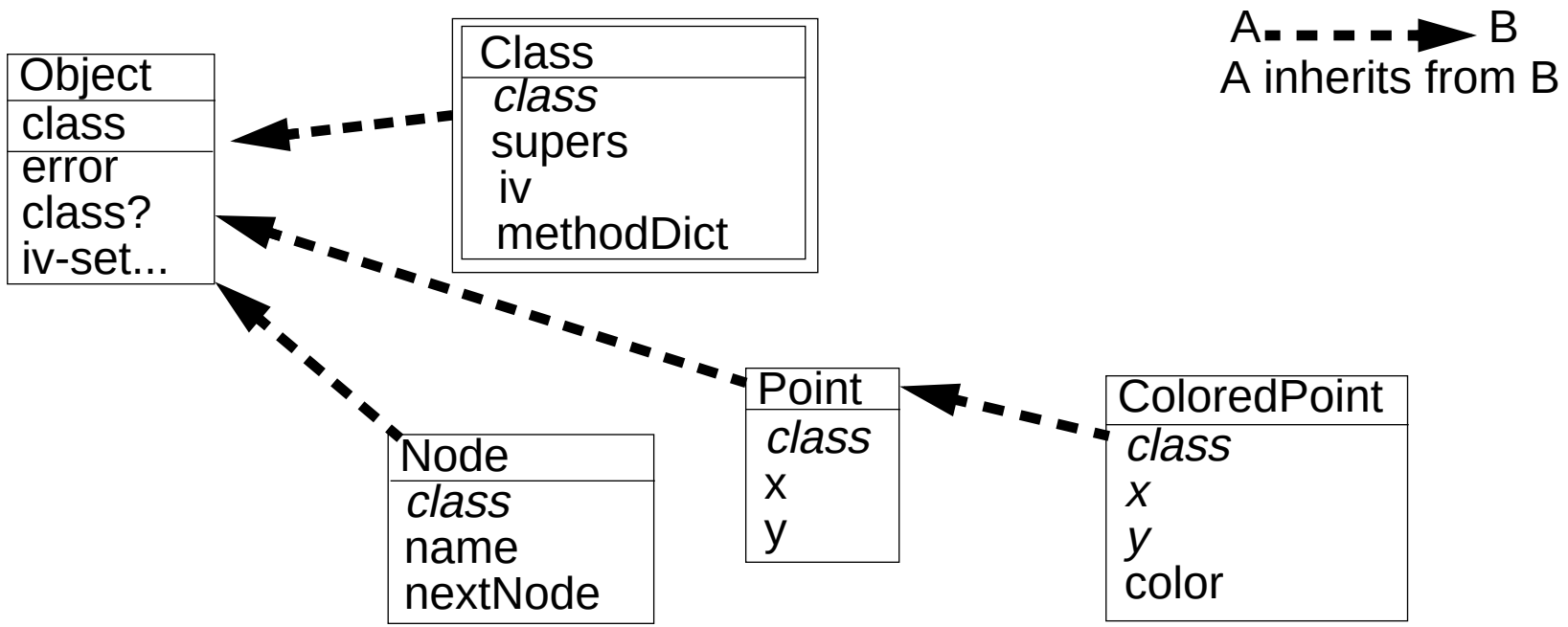
Dynamic Method Inheritance

- Walks through the inheritance graph between classes using the `super` instance variable

  lookup (selector class receiver):
  - if the method associated with the the selector is found
    then return it
    else
    - if receiver class == `Object`
      then [`receiver error selector`]
      else we lookup in the superclass of the class

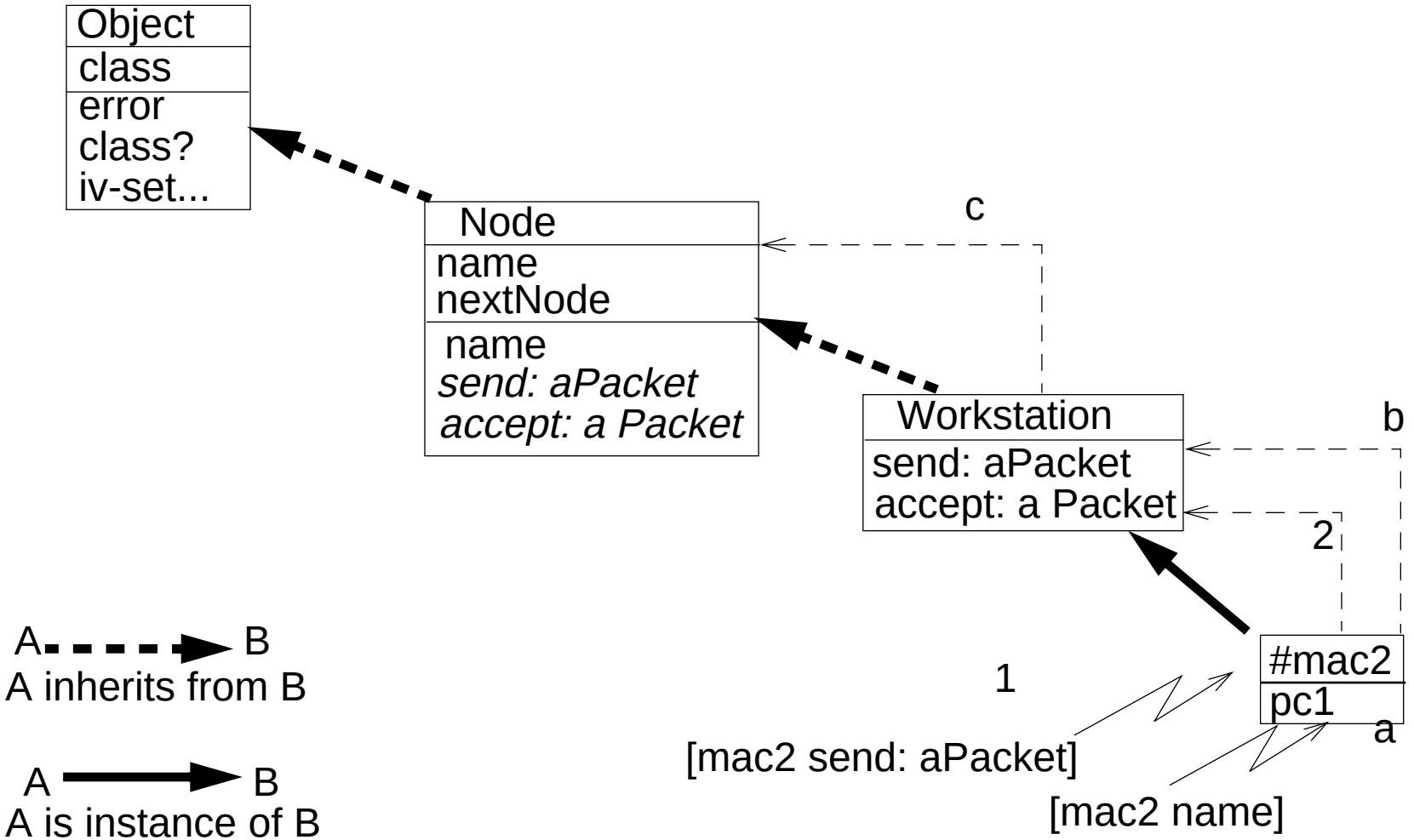
  ☞ the `error` method can be specialized to handle the error.

A Simple Inheritance Graph

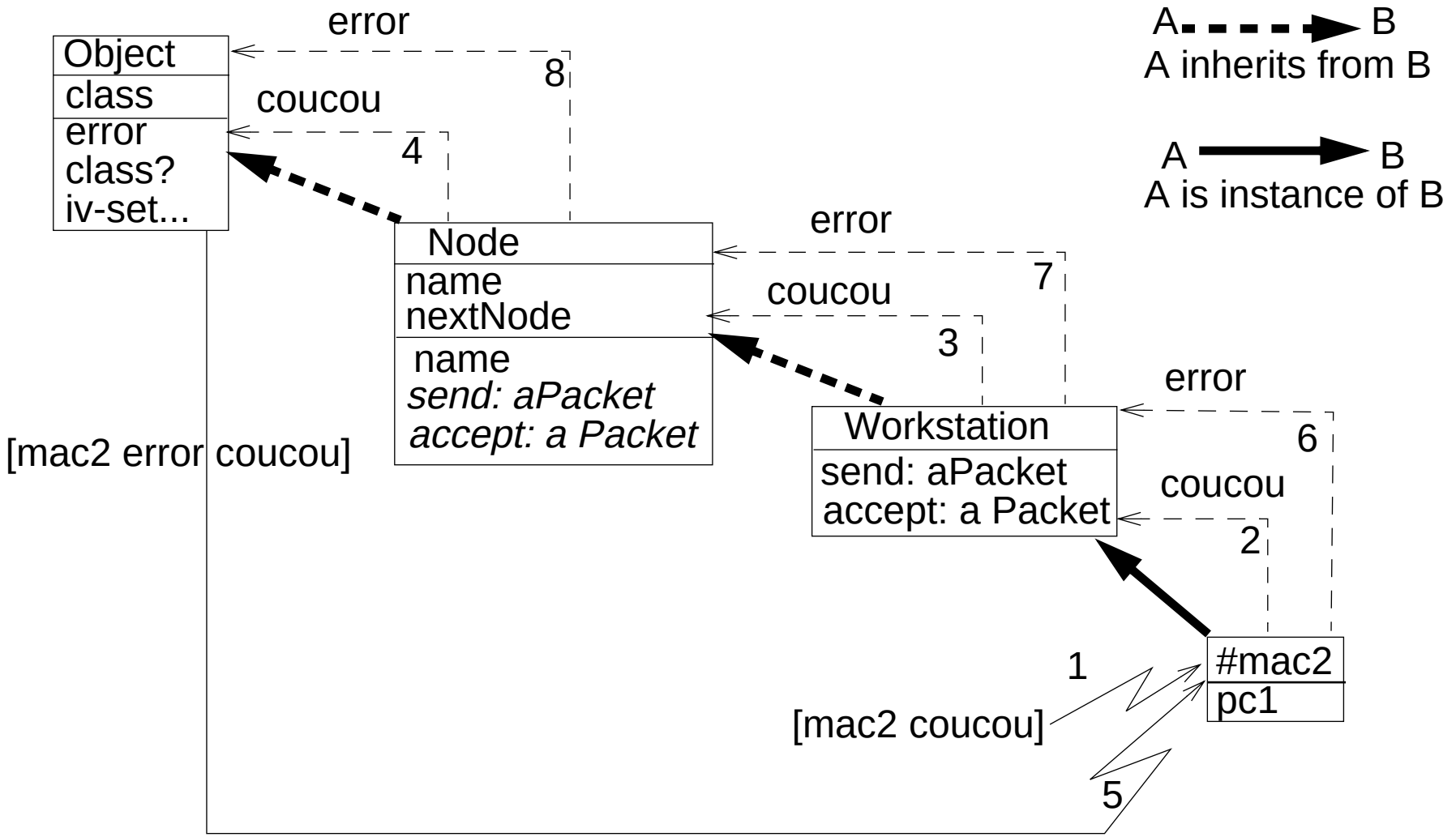


- `Object` class is the root of the hierarchy.
- a Workstation is an object (should at least understand the minimal behavior), so Workstation class inherits from `Object` class
- a class is an object so `Class` class inherits from Object class
- In particular, `class` instance variable is inherited from `Object` class.

Method Lookup Example (i)



Method Lookup Example (ii)



Semantics of super

- As `self`, `super` is a pseudo-variable that refers to the receiver of the message. Used to invoke overriden methods.
- Using `self` the lookup of the method begins in the **class of the receiver**.
- `self` is dynamic

- Using `super` the lookup of the method begins in the superclass of the class of the method containing the super expression and NOT in the superclass of the receiver class.
- `super` is static
- Other said: `super` causes the method lookup to begin searching in the superclass of the class of the method containing `super`

Let us be Absurb!

Let us suppose the WRONG hypothesis:

"IF super semantics = starting the lookup of method in the superclass of the receiver class"

What will happen for the following message: aC m1

m1 is not defined in C

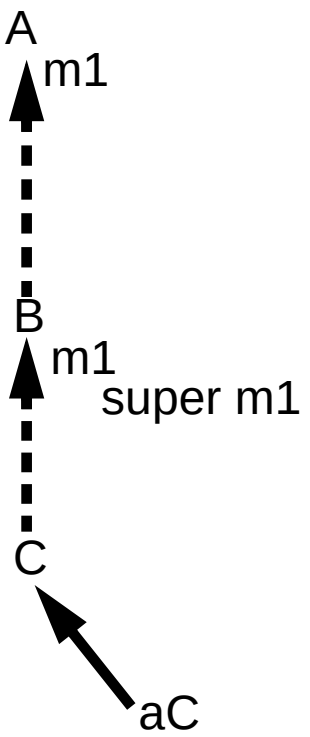
m1 is found in B

By Hypothesis: super = lookup in the superclass of the receiver class.

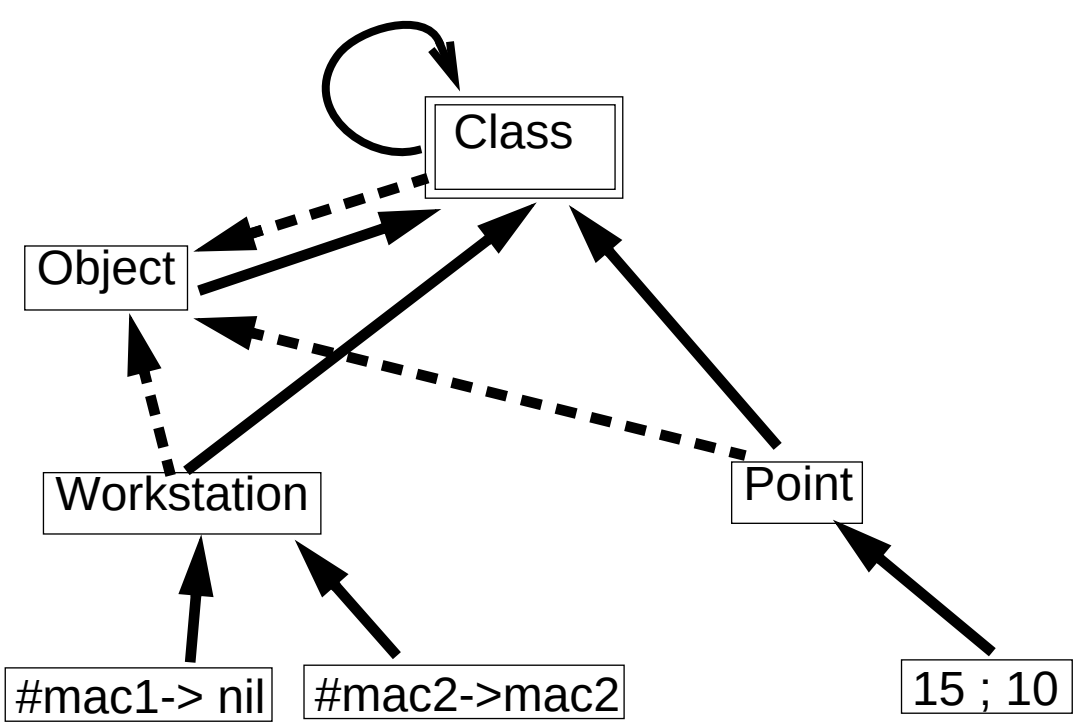
And we know that the superclass of the receiver class = B

=> That's loop

So Hypothesis is WRONG !!



A Simple Uniform Kernel



A - - - -> B
A inherits from B

A ——> B
A is instance of B

Class initialization: a Two Steps Process

`initialize` is defined on both classes `Class` and `Object`:

- on `Object`: values are extracted from initarg list and assigned to the allocated instance

```
[#(Point nil nil) initialize (:y 6 :x 24)]
=> #(Point 6 24)
```

`Initialize` is lookup in class of #(Point nil nil) : `Point`

Then in its superclass: `Object`

- on `Class`:

```
[Class new :name Point :super Object :i-v (x y)...]
[#(Class nil nil nil...) initialize (:name Point :super Object :i-v (x y)...]
```

☞ a class is an object

```
[#(Class Point Object (x y) nil #(x: (mkmethod...) y: (mkmethod ...)]
```

☞ a class is at minimum a class

inheritance of instance variables,

keyword definition,

method compilation

```
[#(Class Point Object (class x y) (:x :y) #(x: (...) y: (...)]
```

Recap: Class class

- Initial metaclass
- Reflective: its instance variable values describe instance variables of any classes in the system (itself too)
- Defines the behavior of all the classes
- Inherits from `Object` class
- Root of the instantiation graph
- Instance variables: `name, super, iv, methodDict`
- Methods
  - `new`
  - `allocate`
  - `initialize` (instance variable inheritance, keywords, method compilation)
  - `class?`
  - `subclass-of?`

Recap: Object class

- Defines the behavior shared by all the objects of the system
- Instance of `Class`
- Root of the inheritance tree: all the classes inherit directly or indirectly from `Object`
- Its instance variable: `class`
- Its methods:
  - `initialize` (initialisation les variables d'instance)
  - `error`
  - `class`
  - `metaclass?`
  - `class?`
  - `iv-set`
  - `iv-ref`

Outline

- Metaclasses?
- Examples of usefull metaclasses
- Towards a unified approach: Loops, Smalltalk
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
  - ☞ Bootstrapping
- Examples: Playing with ObjVlisp

Bootstrapping the Kernel

- Mandatory to have `Class` instance of itself
- Be lazy: Use as much as possible of the system to define itself
- Idea: Cheat the system so that it believes that `Class` already exists as instance of itself and inheriting from `Object`, then create `Object` and `Class` as normal classes

Three Steps:

1. manual creation of the instance that represents the class `Class` avec with
   - inheritance simulation (class instance variable from `Object` class)
   - only the necessary methods for the creation of the classes (new and initialize)
2. creation of the class `Object [Class new :name Object....]`
   - definition of all the method of `Object`
3. redefinition of `Class`

   `[Class new :name Class :super Object.....]`
   - definition of all the methods of `Class`

Abstract Classes

"The rule to define a new metaclass is to make it inherit from a previous one" [Cointe'87]

Prb. Abstract classes should not create instances
Sol. Redefine the new method

Metaclass Definition:

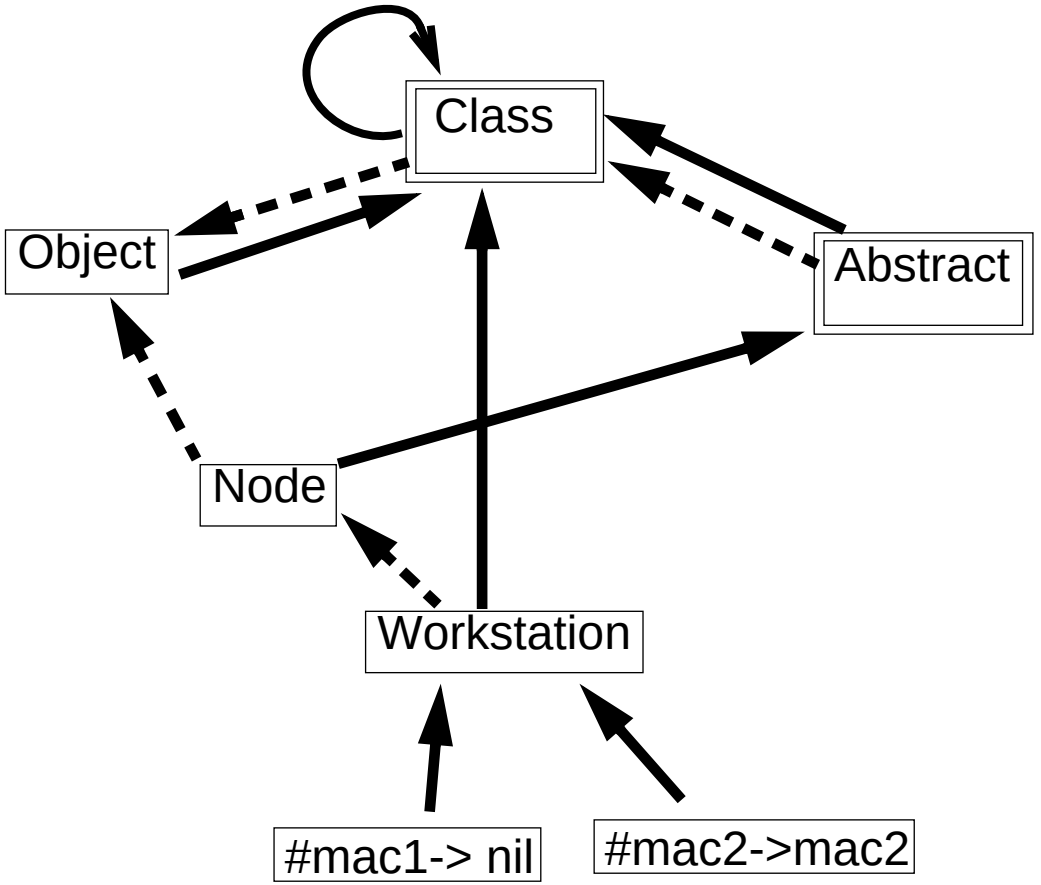
```
[Class new
   :name Abstract
   :super Class
   :methods (new (lambda (self initargs)
                    (self error "Cannot create instance of class %s" self name))]
```

Metaclass Use:

```
[ Abstract new :name Node :super Object ....]
[ Node new ]
-> Cannot create instance of class Node

[ Abstract new :name Abstract-Stack  :super Object ....]
```

Abstract

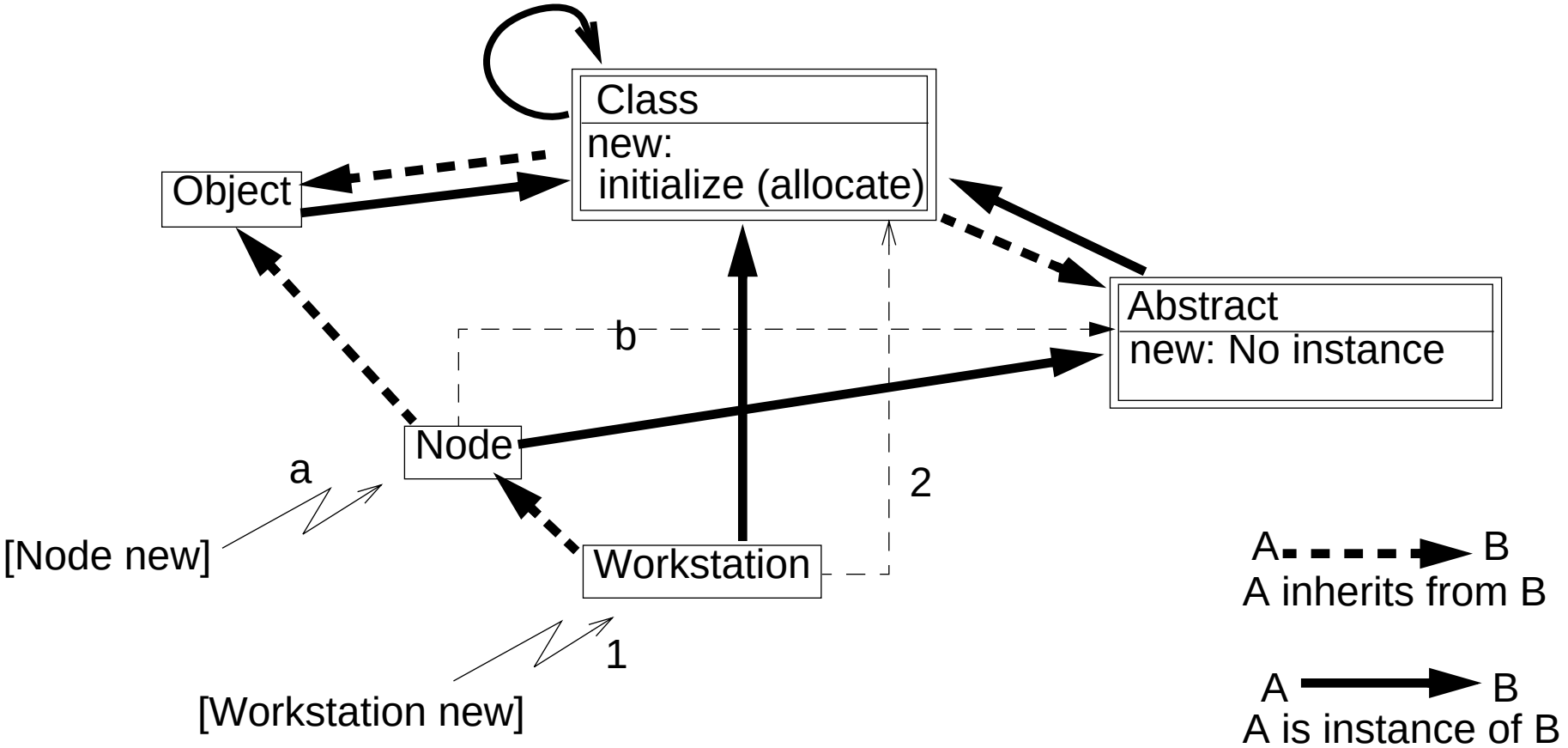
- Abstract is a class -> It is instance of `Class`
- Abstract define class behavior -> It inherits from `Class`



A - - - -> B
A inherits from B

A ——> B
A is instance of B

Abstract Class and Method Lookup



3. Study of an Object-Oriented Reflective Kernel

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

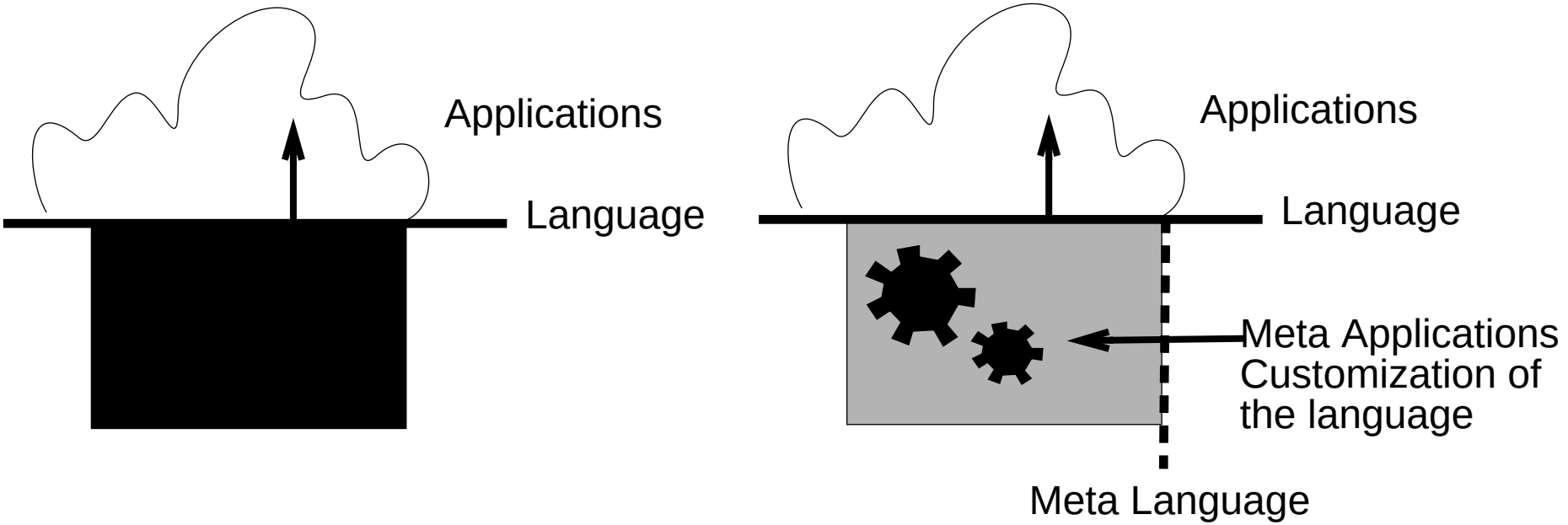
Goals of this Lecture

- Metaclass concept
- Reflective Architectures and Kernels (SOM, Smalltalk, CLOS)
- What are Object and Class classes?
- Semantics of inheritance, semantics of super
- Metaclass power
- Metaclass limits
- Metaclass composibility solution

Outline

- Examples of usefull metaclasses
- Examples of programming with metaclasses (client / metaprogrammer)
- Towards a unified approach: Loops, Smalltalk
- Building your own metaclass kernel: ObjVlisp
- Examples: Playing with ObjVlisp
- Metaclasses are powerful but
- Problems with composition
- Problems with property propagation
- Clos's solution
- Smalltalk's solution
- SOM's solution
- NeoClasstalk's solution
- Conclusion
- Bibliography

Recall: Meta Programming in Programming Language Context



Outline

- ☞ Examples of usefull metaclasses
- ❑ Examples of programming with metaclasses (client / metaprogrammer)
- ❑ Towards a unified approach: Loops, Smalltalk
- ❑ Building your own metaclass kernel: ObjVlisp
- ❑ Examples: Playing with ObjVlisp
- ❑ Metaclasses are powerful but
- ❑ Problems with composition
- ❑ Problems with property propagation
- ❑ Clos's solution
- ❑ Smalltalk's solution
- ❑ SOM's solution
- ❑ NeoClasstalk's solution
- ❑ Conclusion
- ❑ Bibliography

Class as Objects

"The difference between classes and objects has been repeatedly emphasized. In the view presented here, these concepts belong to different worlds: the program text only contains classes; at run-time, only objects exist. This is not the only approach. **One of the subcultures of object-oriented programming, influenced by Lisp and exemplified by Smalltalk, views classes as object themselves, which still have an existence at run-time.**"

Bertrand Meyer in Object-Oriented Software Construction

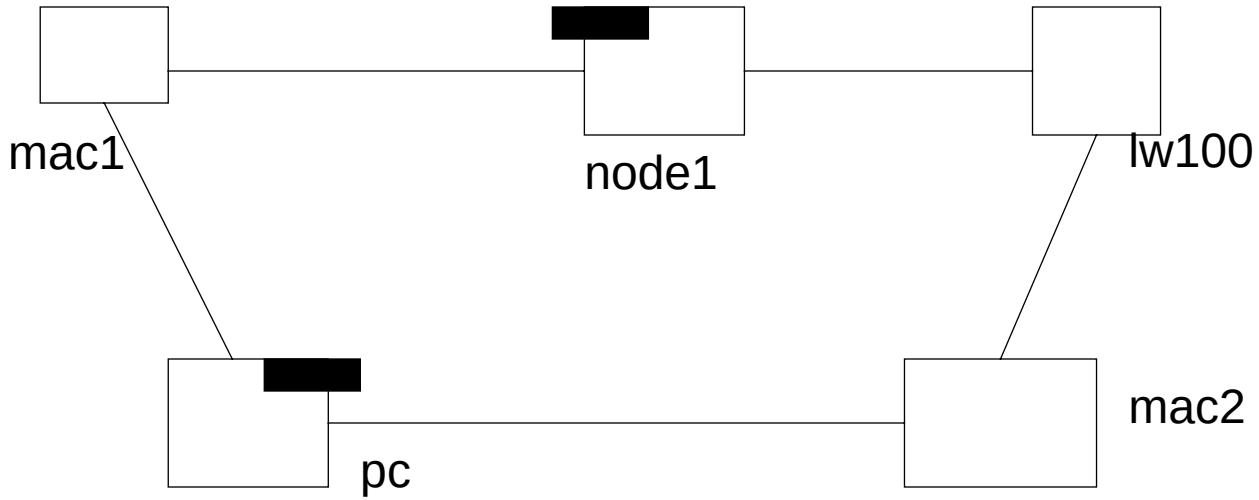
Some Class Properties

Abstract: a class cannot have any instance
Set: a class that knows all its instances
BreakPoint: some methods are not run and a debugger is opened
DynamicIVs: Lazy allocation of instance structure
LazyAccess: only fetch the value if needed
AutomaticAccessor: a class that defines automatically its accessors
Final: Class cannot be changed and subclassed
FinalMethods: Methods that cannot be specialized
Limited/Singleton: a class can only have a certain number of instances
IndexedIVs: Instances have indexed instance variables
InterfaceImplementor: class must implement some interfaces
MultipleInheritance: a class can have multiple superclasses
Released: a class that cannot changed anymore
Trace: Logs method calls, attribute accesses
PrePostConditions: methods with pre/post conditions
MessageCounting: Counts the number of times a method is called (missing method metaobject)

A Simple Application as Example

A LAN Simulator:

- A LAN contains nodes, workstations, printers, file servers.
- Packets are sent in a LAN and the nodes treat them differently.



Problem: We want to analysis all the messages sent

But:

- We do not want to change the code of the node classes.

@@stay??@@ Programming in Explicit Metaclass Context

CLOS-like

```
(defclass Node ()
   ((name :initarg :name :default-value #lulu :reader name)
    (nextNode :default-value '() :accessor nextNode)))

(defmethod accept ((n Node) (p Packet))
   ....)
(defmethod send ((n Node) (p Packet))
   ...)

@@Check counting Kiczales here@@
```

Reusing Meta Programs@@stay??@@

MetaProgramming in OO Context

- A MetaProgram is not mixed into objects
- Ordinary objects are used to model real world. Metaobjects describe these ordinary objects.
- MetaPrograms can be reused.
- Some other properties cannot easily be implemented without meta programming traceMessage, finalClass, PrePostConditions, DynamicIVs, MessageCounting....

We may want to:

- change the representation of the instance variables (indexed for points, hashed for person)
- change the way attributes are accessed (lazily via the net)
- change the inheritance semantics
- change the invocation of method (trace, proxies...)

Metaclass Responsibilities

"Metaclasses provide metatools to build open-ended architecture" [Cointe'87]

Metaclass are one of the possible meta-entities (method, method combination,...)
Metaclass allows the structural extension of the language
They may control

- Inheritance
- Internal representation of the objects (listes, vecteurs, hash-table,...)
- Method access ("caches" possibility)
- Instance variable access

Separation of Concerns

- Ordinary objects are used to model real world
- Metaobjects describe these ordinary objects
- Meta/Base level functionality is not mixed

On the Road Again

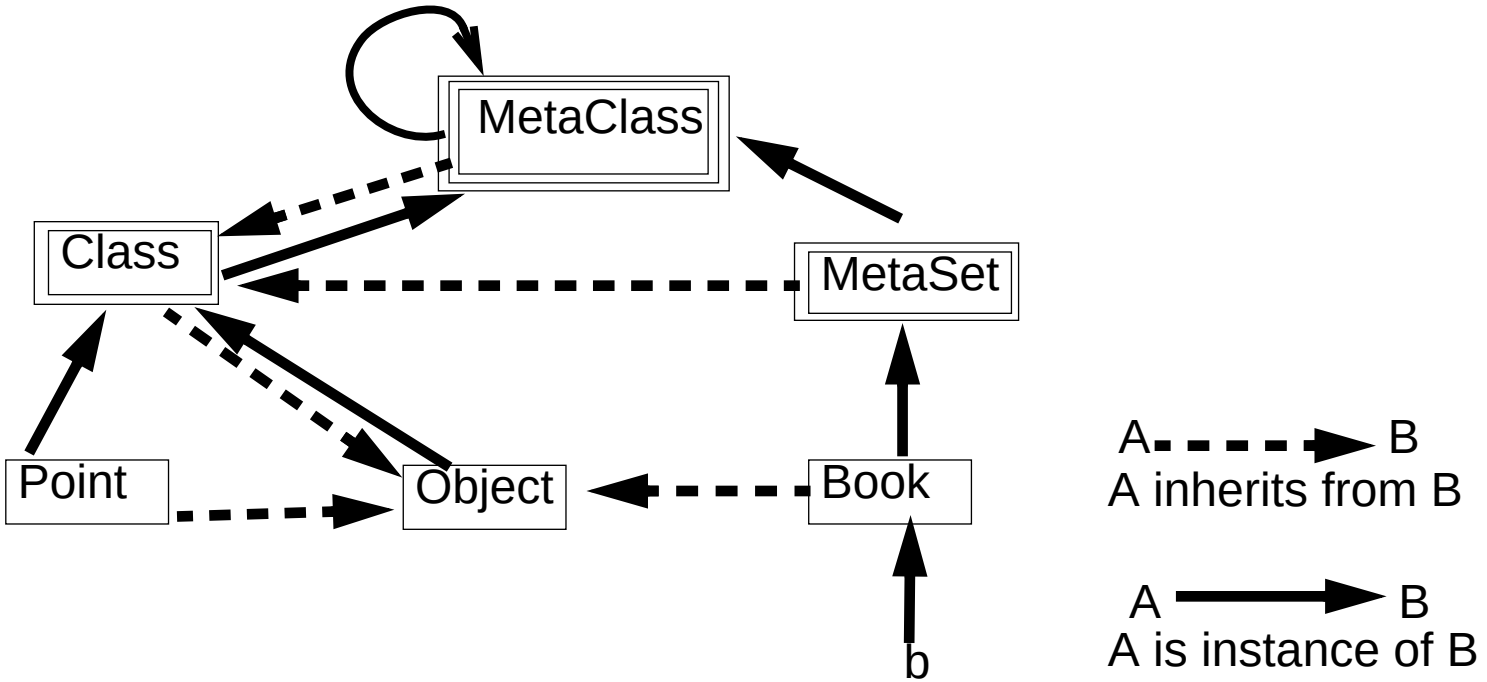
- Towards ObjVlisp
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Inheritance Semantics
- Bootstrapping

Why ObjVlisp?

- Minimal (only two classes)
- Reflective: ObjVlisp self-described: definition of Object and Class
- Unified: Only one kind of object: a class is an object and a metaclass is a class that creates classes
- Open
- Simple: can be implemented with less than 300 lines of Scheme or 30 Smalltalk methods.
- Equivalent of Closette
- Really good for understanding dynamic languages and reflective programming (D-SOM, CLOS, Smalltalk kernel)

The Loops Approach

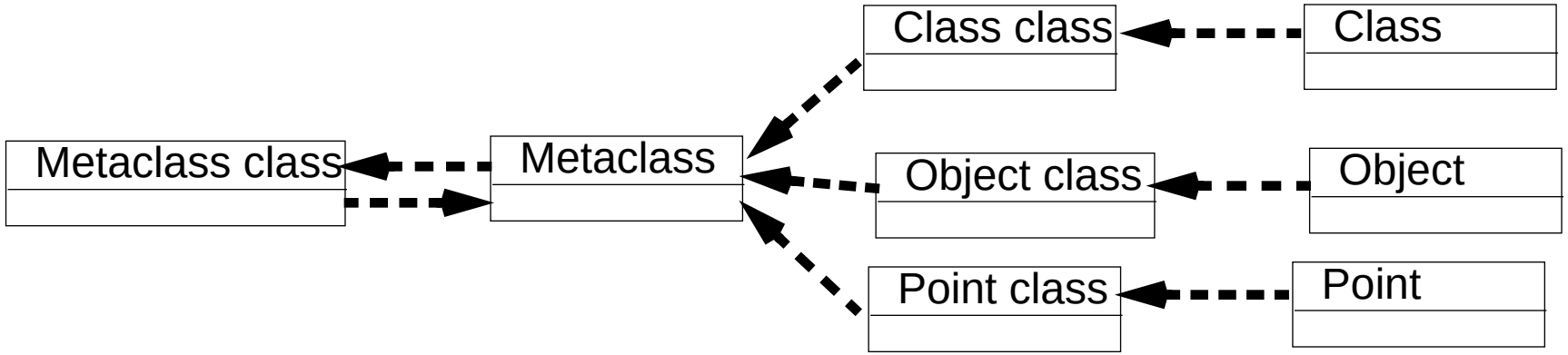
"For some special cases, the user may want ot have more control over the creation of instances. For example, Loops itself uses different Lisp data types to represent classes and instances. The New message for classes is fielded by their metaclass, usually the object MetaClass." [Bobrow83]



- Explict metaclass as a sublcass of another but must be instance of MetaClass

The Smalltalk Pragmatical Approach

"The primary role of a metaclass in the Smalltalk-80 system is to provide protocol for initializing class variables and for creating initialized instances of the metaclasse'sole instance" [Goldberg84]



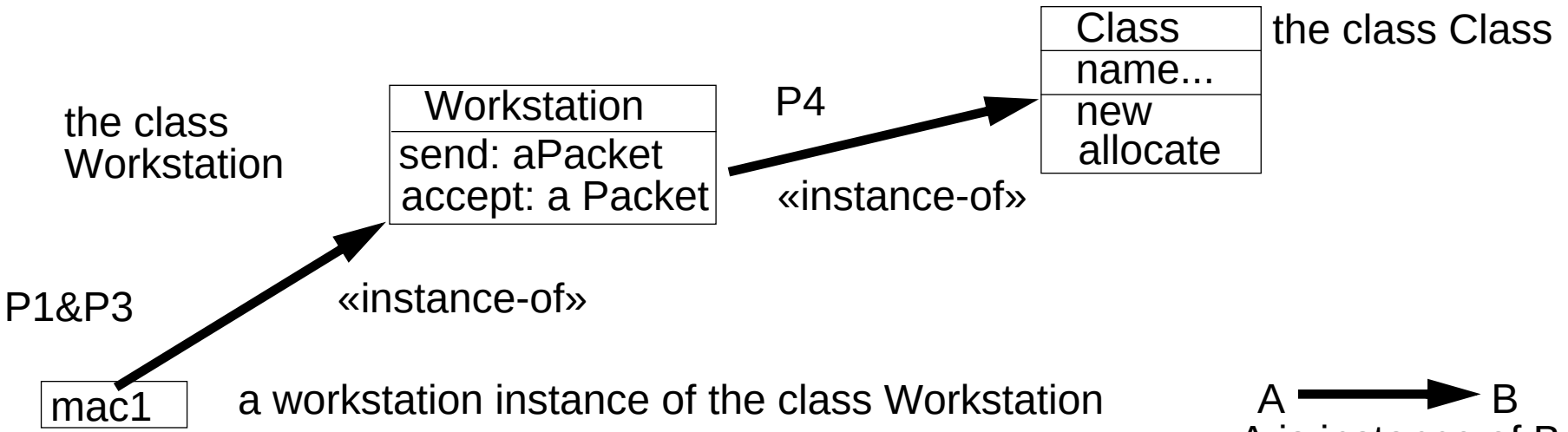
- A class is the sole instance of a metaclass
- Every metaclass is an instance of the `Metaclass` class
  - metaclasses are not true classes
  - number of metalevels is fixed
- Metaclass hierarchy inheritance is fixed: parallel to the class inheritance
  - dichotomy between classes defined by the user (instance of Class) and metaclasses defined by the system (instance of metaclasses)

ObjVlisp in 5 Postulates (i)

P1: object = <data, behavior>

P3: Every object belongs to a class that specifies its data (slots or instance variables) and its behavior. Objects are created dynamically from their class.

P4: Following P3, a class is also an object therefore instance of another class its metaclass (that describes the behavior of a class).

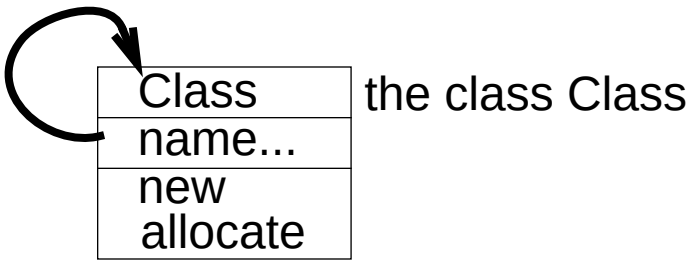


How to Stop Infinite Recursion?

Aclass is an object therefore instance of another class its metaclass that is an object too instance of a metametaclass that is an object too instance of another a metametametaclass.....

To stop this potential infinite recursion

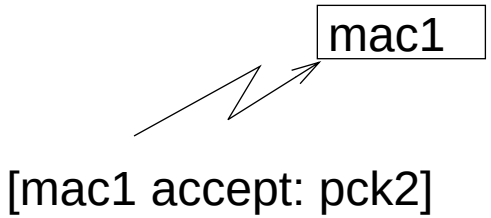
- `Class` is the initial class and metaclass
- `Class` is instance of itself and
- all other metaclasses are instances of `Class`.



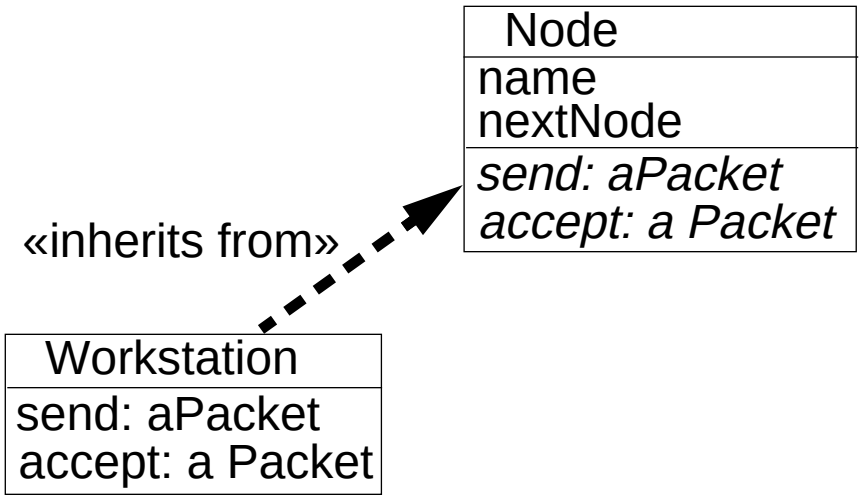
ObjVlisp in 5 Postulates (ii)

P2: Message passing is the only means to activate an object

[objet selecteur args]



P5: A class can be defined as a subclass of one or many other classes. This mechanism is called inheritance. It allows the sharing of instance instance variable and methods. The class Object represents the behavior shared by all the objects.



Unification between Classes and Instances

"We claim that a class must be an object defined by a real class allowing a greater clarity and expressive power" [Cointe'87]

- Every object is instance of a class
- A class is an object instance of a metaclass (P4)
  - ☞ But all the objects are not classes

- Only one kind of objects without distinction between classes and final instances.
- Sole difference is the ability to respond to the creation message: new. Only a class knows how to deal with it.
- A metaclass is only a class that generates classes.

About the 6th ObjVlisp's Postulate

"Ordinary objects are used to model real world. Metaobjects describe these ordinary objects" [Rivard 96]

ObjVlisp 6th postulate:

class variable (anObject) = instance variable (anObject's class)

So class variables are shared by all the instances of a class.

- But semantically class variables are not instance variables of object'class!
- Instance variable of metaclass should represent class information not instance information.

Metaclass information should represent classes not domain objects

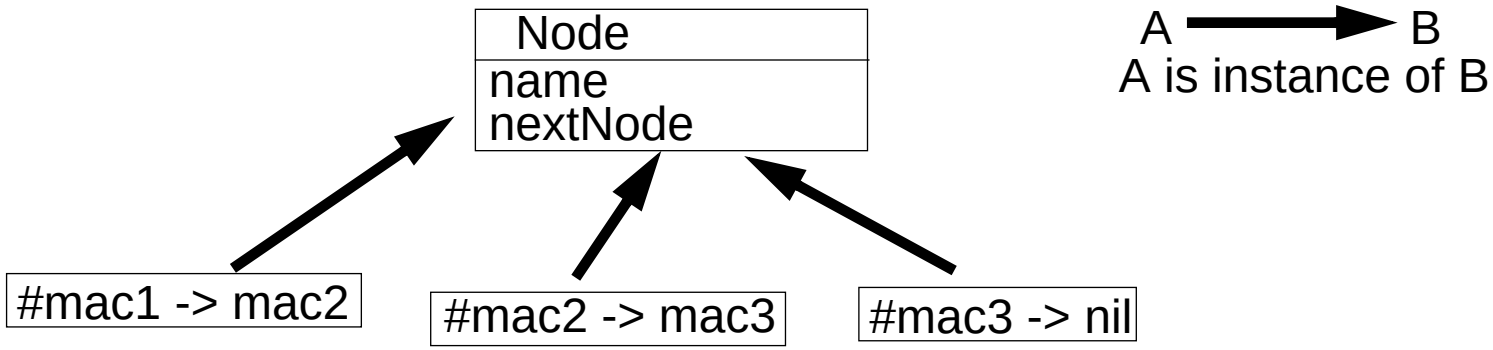
CLOS offers the :class instance variable qualifier class variables.

We could imagine that a class possesses an instance variable that stores structure that represents shared-variable and their values.

Instance Structure: Instance Variables

Instance variables:

- an ordered sequence of instance variables defined by a class
- shared by all its instances
- values specific to each instance



In particular, every object possesses an instance variable class (inherited from Object) that points to its class.

Instance Behavior: Methods

A method

- belongs to a class
- defines the behavior of all the instances of the class
- is stored into a dictionary that associates a key (the method selector) and the method body

To unify instances and classes, the method dictionary of a class is the value of the instance variable `methodDict` defined on the metaclass `Class`.

Class as an Object: Structure

- Considered as an object, a class possesses an instance variable class inherited from `Object` that refers to its class (here to the metaclass that creates it).
- But as an instance factory the metaclass `Class` possesses 4 instance variables that describe a class:

- `name`	the class name
- `supers`	the list of its superclasses
- `i-v`	the list of its instance variables
- `methodDict`	a method dictionary

Example: class Node

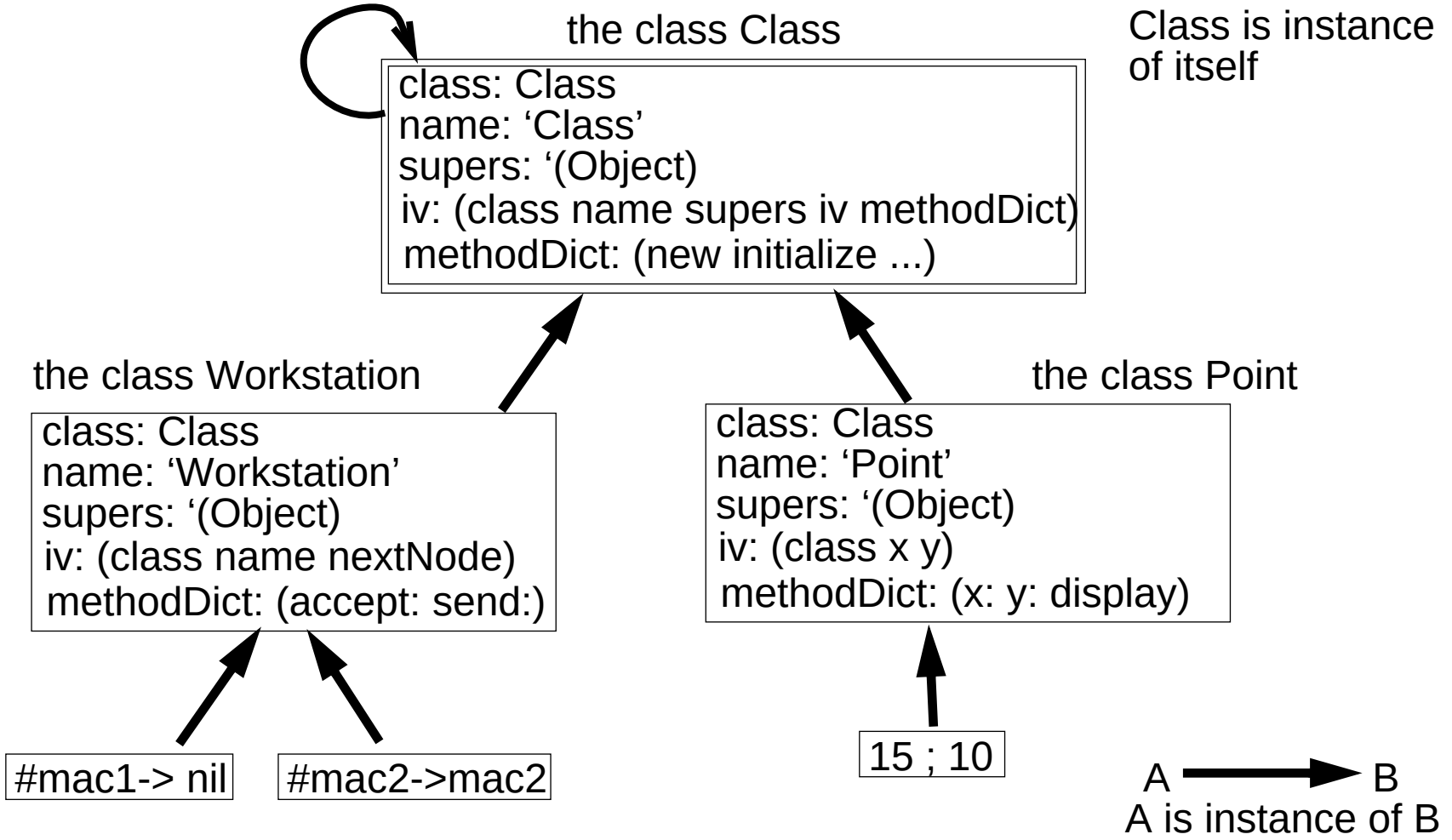
class:	Class	instance of Class
name:	"Node"	named Node
supers:	'(Object)	inherits from Object
i-v:	'(name nextNode)	defines 2 instance variables
methods:	.....	defines methods

The class Class: a Reflective class

- Initial metaclass
- Defines the behavior of all the metaclasses
- Instance of itself to avoid an infinite regression

class:	Class	instance of Class
name:	"Class"	named Class
supers:	'(Object)	inherits from Object
i-v:	'(name supers i-v methodDict)	
	describes the instance variables of any class	
methods:	‘(new allocate initialize.....	

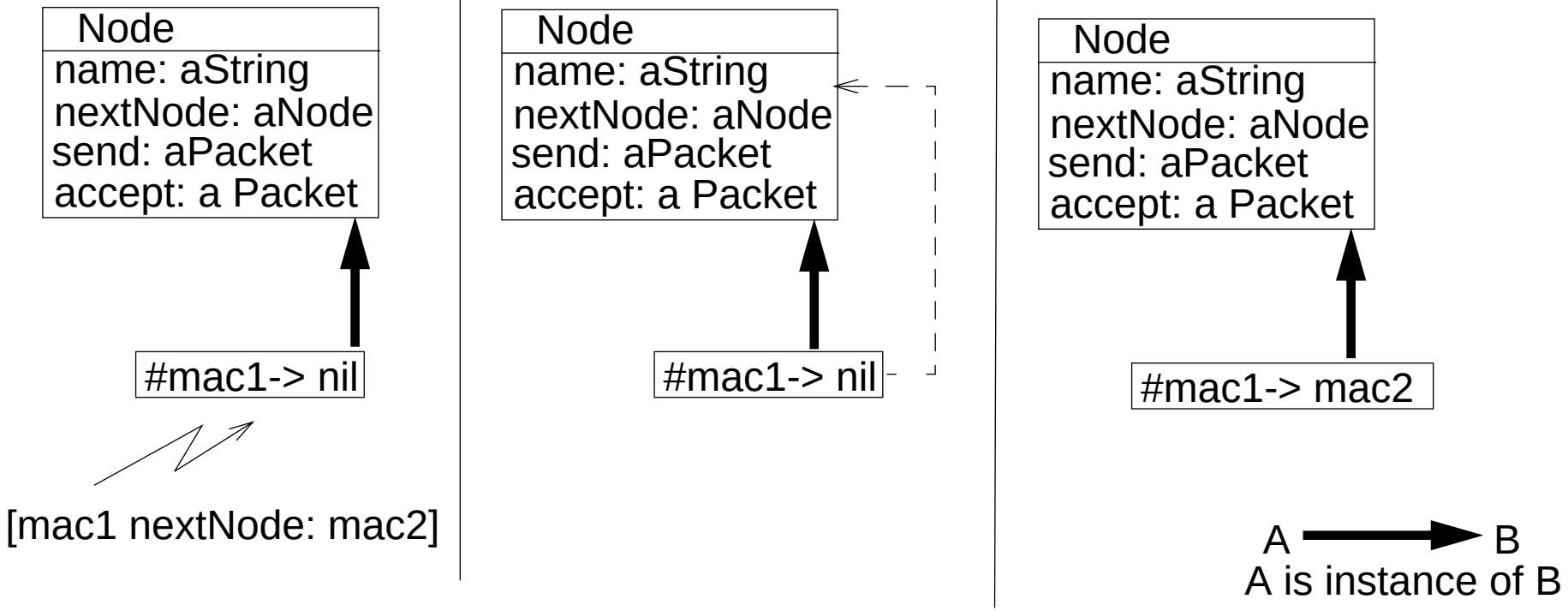
A Complete Example



Message Passing (i)

P2: Message passing is the only means to activate an object

P3: Every object belongs to a class that specifies its data and its behavior.



Message Passing (ii)

We **lookup** the method associated with the selector of the message **in the class** of the receiver then **we apply it to the receiver**.

[receiver selector args]
<=>
apply (found method starting from the class of the receiver)
on the receiver and the args
<=>
in Scheme
(apply (lookup selecteur (class-of receiver) receiver)
receiver args)

Object Creation by Example

Creation of instances of the class Point

```
[Point new :x 24 :y 6]
[Point new]
[Point new :y 10 :y 15]
```

Creation of the class Point instance of Class

```
[send Class 'new
        :name Point
        :supers '(Object)
        :i-v '(x y)
        :methods '(x (lambda (self)...)
                  display (lambda (self)...))
]
```

Object Creation: the Method new

- new creates an object: class or final instances
- new is a class method
- Creating an instance is the composition of two actions:
  - memory allocation: allocate method
  - object intialisation: initialize method

(new aClass args) = (initialization (allocation aClass) args)

<=>

[aClass new args] = [[send aClass allocate] initialize args]

Object Allocation

- Object allocation should return:
  - Object with empty instance variables
  - Object with an identifier to its class

- Done by the method allocate defined on the metaclass Class
- Allocate method is a class method

example:

```
[Point  'allocate]  => #(Point nil nil)
   for x and y
[Workstation ‘allocate] => #(Workstation nil nil)
   for name and nextNode
[Class 'allocate]  => #(Class nil nil nil....)
```

Object Initialization

- Initialization allows one to specify the value of the instance variables by means of keywords (:x ,:y) associated with the instances variables.

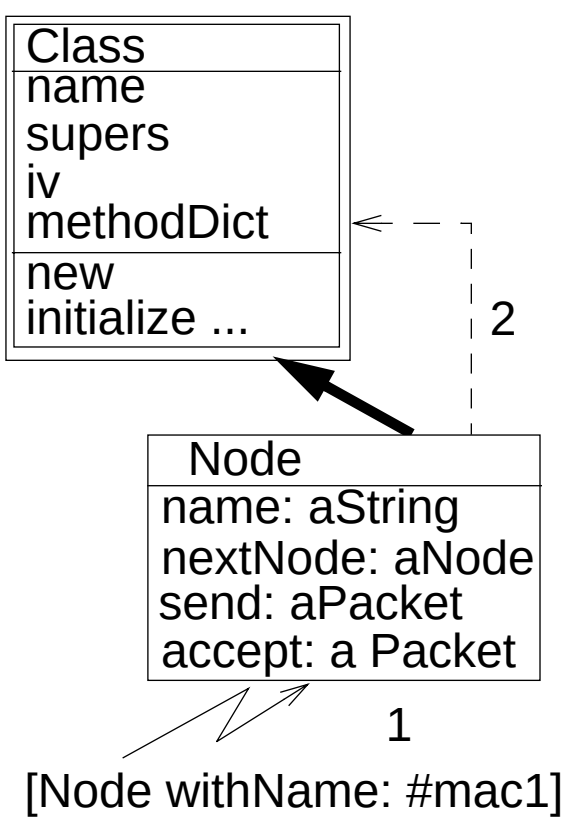
Example:

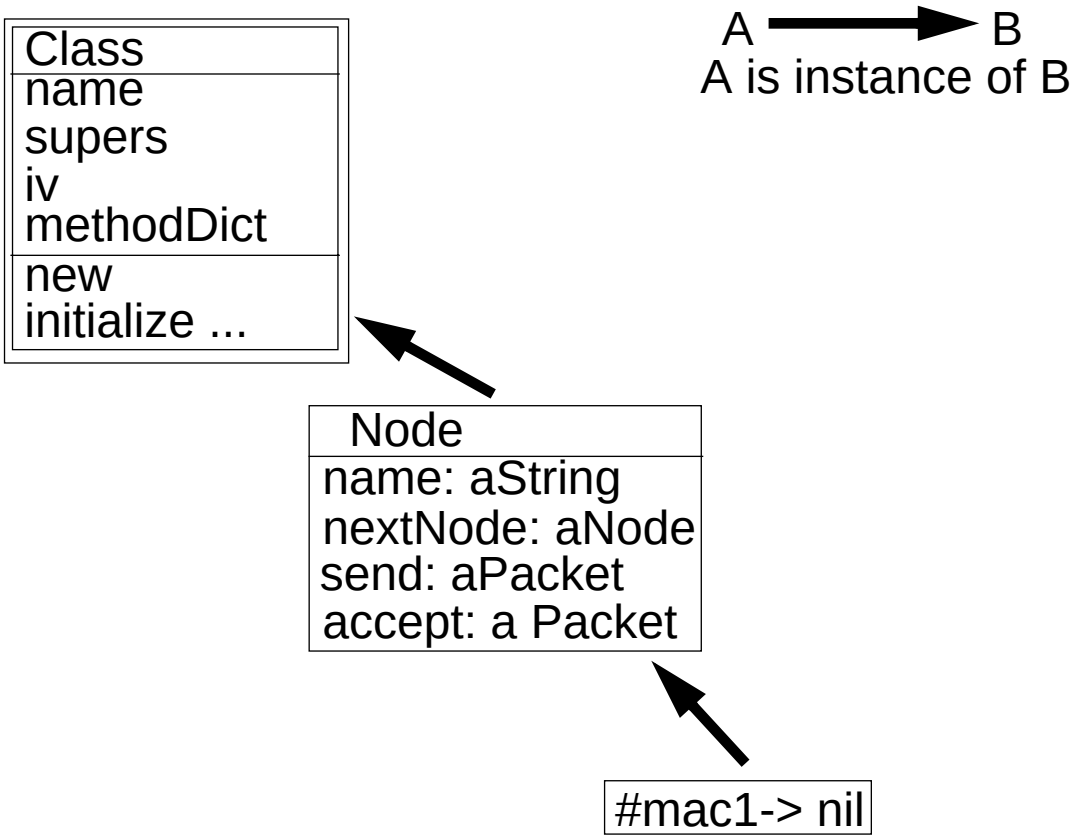
```
[ Point 'new :y 6  :x 24]  =>
[ #(Point nil nil) initialize '(:y 6 :x 24)] =>
#(Point 24 6)
```

- initialize two steps
  - get the values specified during the creation. (y -> 6, x -> 24)
  - assign the values to the instance variables of the created object.

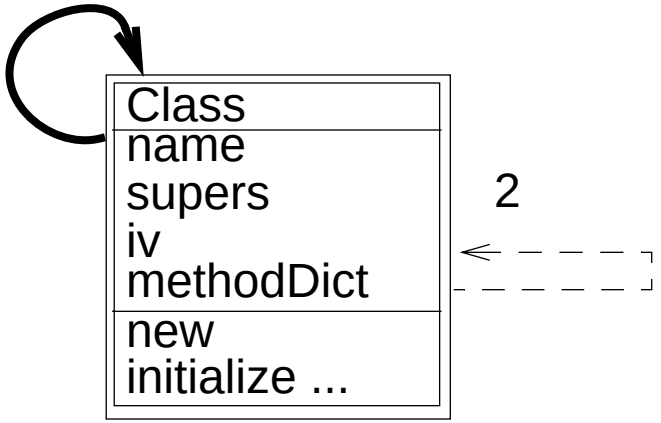
Object Creation: the Metaclass Role

We **lookup** the method associated with the selector of the message **in the class** of the receiver then **we apply it to the receiver**.

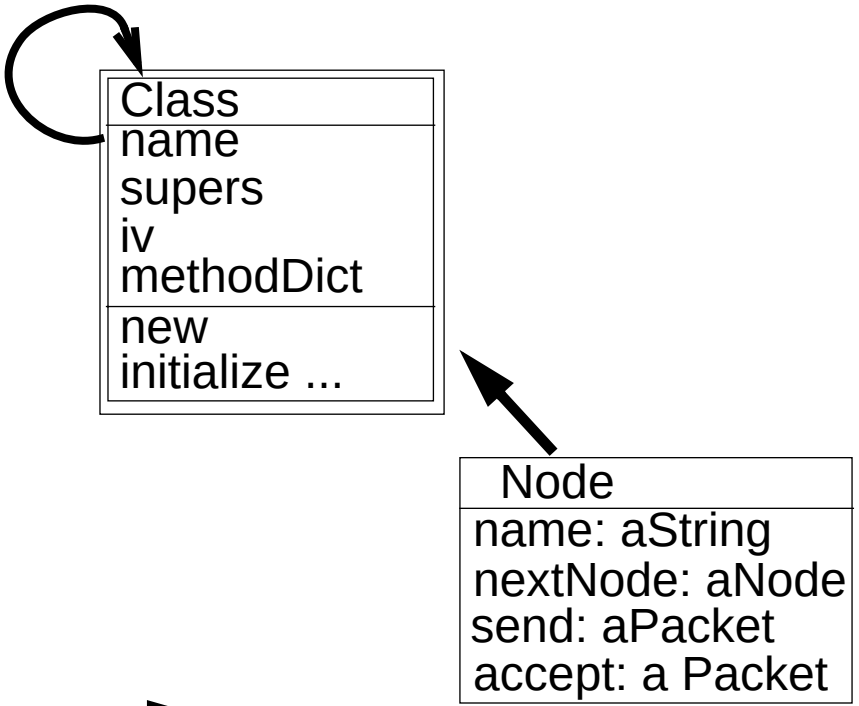




Class Creation

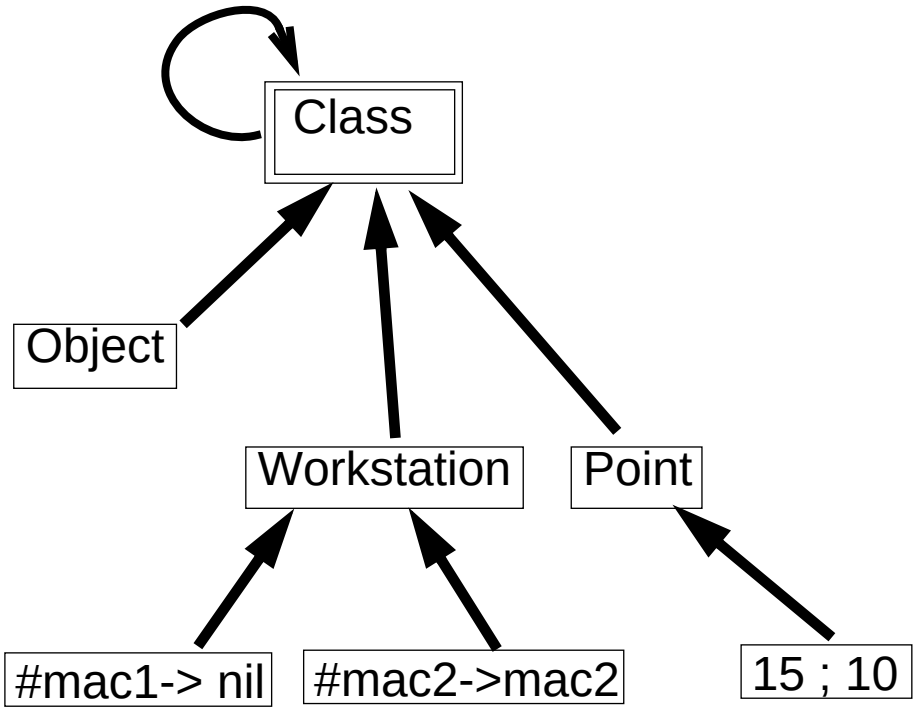



```
[Class 'new
        :name Node
        :supers '(Object)
        :iv '(name nextNode)
        :methods
        '(send: (lambda(self aPack)....))]
```



A → B
A is instance of B

A Simple Instantiation Graph



- Object is a class that represents the minimal behavior of an object.
- Object is a class so it is instance of class

What is the minimal behavior shared by all the objects?

The class Object represents the common behavior shared by all the objects:

- ☞ classes
- ☞ final instances.

- ❑ every object knows its class: instance variable class (uses a primitive for accessing else that loops!)

- ❑ methods:
  - initialize (instance variable initialization)
  - error
  - class
  - metaclass?
  - class?
  - iv-set
  - iv-ref

Two Forms of Inheritance

- Static for the instances variables
  - ☞ Done once at the class creation
  - ☞ When C is created, its instances variables are the union of the instance variables of its superclass with the instance variables defined in C.

i-v(C) = union (union (iv (supers C)), :i-v(C))

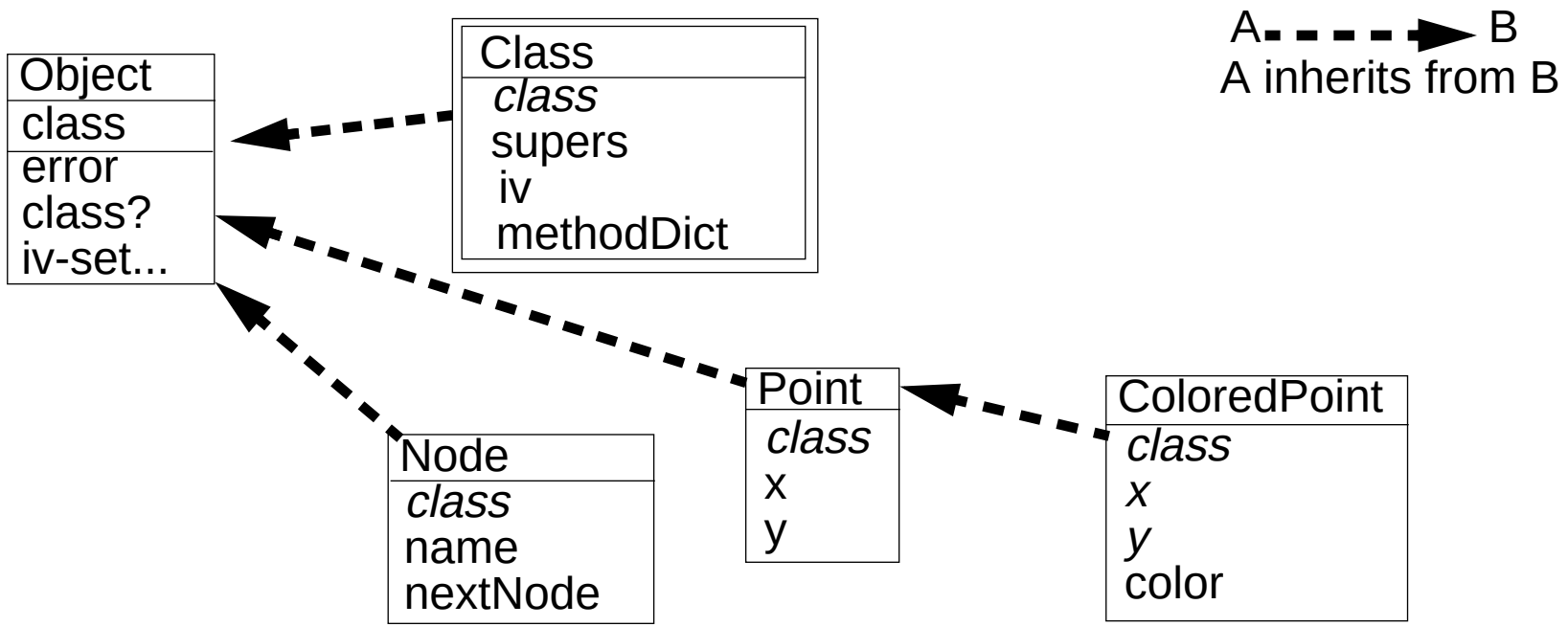
Dynamic Method Inheritance

- Walks through the inheritance graph between classes using the super instance variable

lookup (selector class receiver):
    if the method associated with the the selector is found
    then return it
    else
        if receiver class == Object
        then [receiver 'error selector]
        else we lookup in the superclass of the class

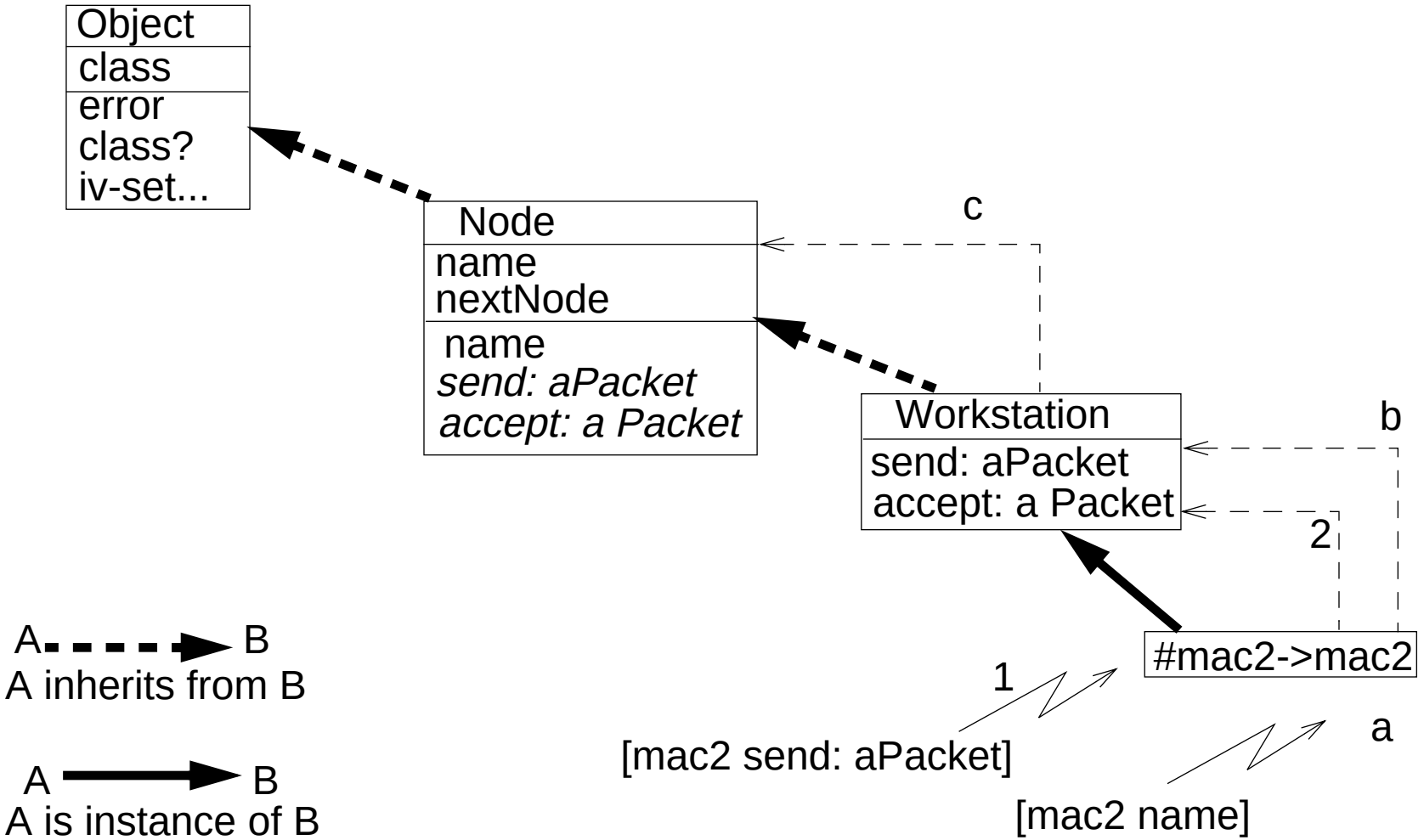
the error method can be specialized to handle the error.

A Simple Inheritance Graph

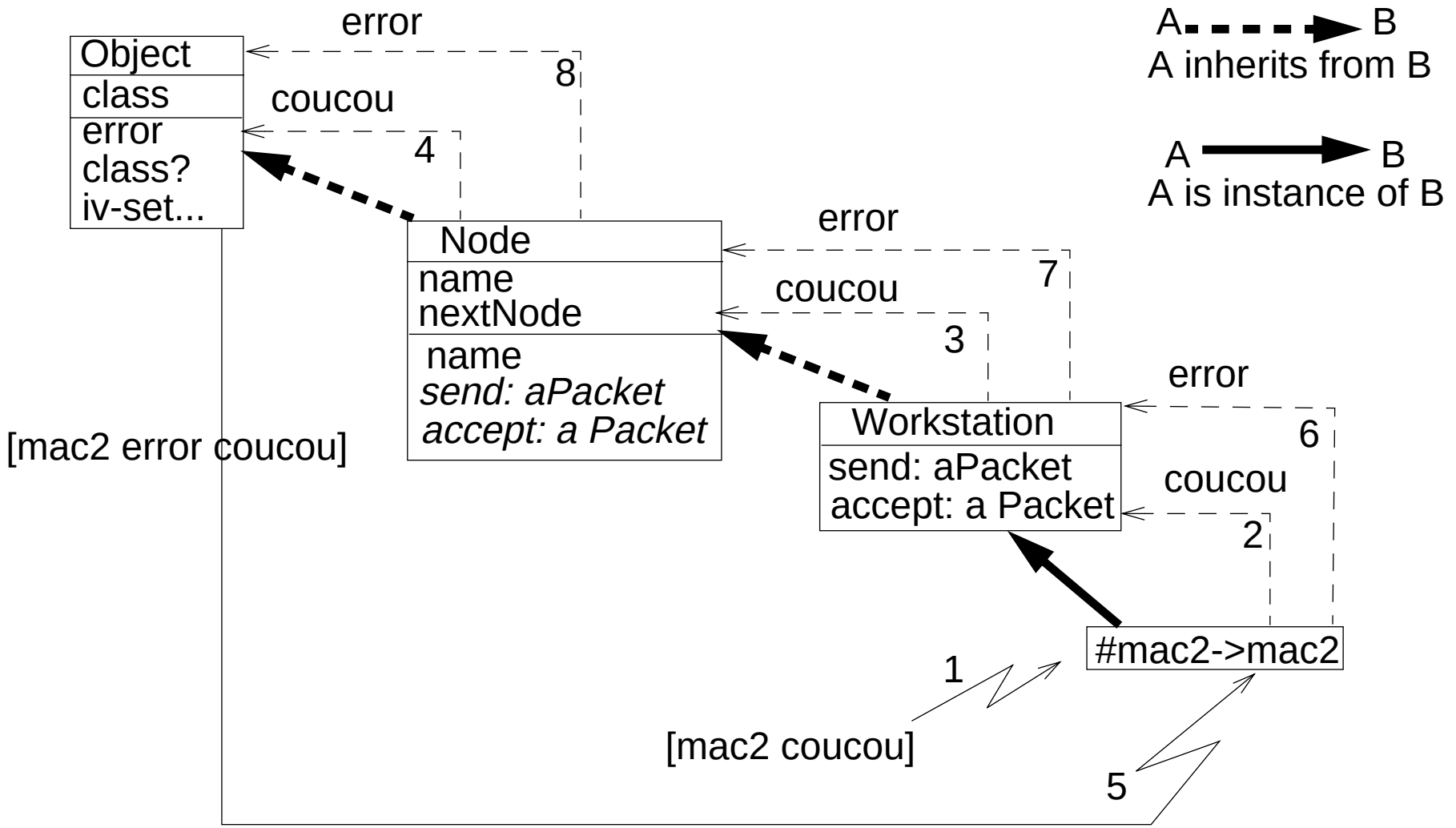


- `Object` class is the root of the hierarchy
- a Workstation is an object (should at least understand the minimal behavior), so Workstation class inherits from `Object` class
- a class is an object so `Class` class inherits from Object class
- In particular, class instance variable is inherited from `Object` class.

Method Lookup Example (i)



Method Lookup Example (ii)



Semantics of super

- As `self`, `super` is a pseudo-variable that refers to the receiver of the message. Used to invoke overriden methods.
- Using `self` the lookup of the method begins in the **class of the receiver**.
- `self` is dynamic

- Using `super` the lookup of the method begins in the superclass of the class of the method containing the super expression and NOT in the superclass of the receiver class.
- `super` is static
- Other said: `super` causes the method lookup to begin searching in the superclass of the class of the method containing `super`

Let us be Absurb!

Let us suppose the WRONG hypothesis:

"IF super semantics = starting the lookup of method in the superclass of the receiver class"

What will happen for the following message: aC m1

m1 is not defined in C

m1 is found in B

By Hypothesis: super = lookup in the superclass of the receiver class.

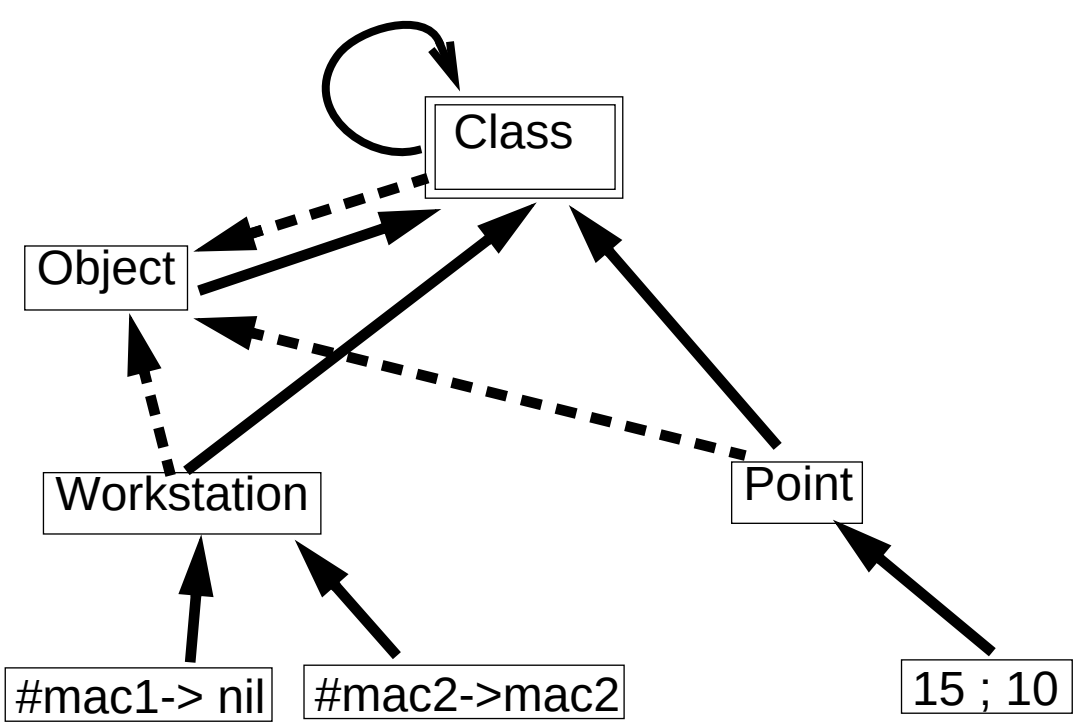
And we know that the superclass of the receiver class = B

=> That's loop

So Hypothesis is WRONG !!

@@Stef redo the pictuire with the right arrow@@

A Simple Uniform Kernel



A - - - -> B
A inherits from B

A ——> B
A is instance of B

Class initialization: a Two Steps Process

`initialize` is defined on both classes `Class` and `Object`:

- on `Object`: values are extracted from initarg list and assigned to the allocated instance

```
[#(Point nil nil) initialize '(:y 6 :x 24)]
=> #(Point 6 24)
```

`Initialize` is lookup in class of #(Point nil nil) : `Point`

Then in its superclass: `Object`

- on `Class`:

```
[send Class 'new :name "Point" :supers '(Object) :i-v '(x y)...]
[#(Class nil nil nil...) initialize '(:name Point :supers '(Object) :i-v '(x y)...]
```

☞ a class is an object

```
[#(Class "Point" '(Object) '(x y) nil #(x: (mkmethod...) y: (mkmethod ...)]
```

☞ a class is at minimum a class

inheritance of instance variables,

keyword definition,

method compilation

```
[#(Class "Point" '(Object) '(class x y) (:x :y) #(x: (lambda...) y: (lambda ...)]
```

Recap: Class class

- Initial metaclass
- Reflective: its instance variable values describe instance variables of any classes in the system (itself too)
- Defines the behavior of all the classes
- Inherits from `Object` class
- Root of the instantiation graph
- Instance variables: `name, supers, iv, methodDict`
- Methods
  - new
  - allocate
  - initialize (instance variable inheritance, keywords, method compilation)
  - class?
  - subclass-of?

Recap: Object class

- Defines the behavior shared by all the objects of the system
- Instance of `Class`
- Root of the inheritance tree: all the classes inherit directly or indirectly from `Object`
- Its instance variable: `class`
- Its methods:
  - initialize (initialisation les variables d'instance)
  - error
  - class
  - metaclass?
  - class?
  - iv-set
  - iv-ref

Bootstrapping the Kernel

- Mandatory to have `Class` instance of itself
- Be lazy: Use as much as possible of the system to define itself
- Idea: Cheat the system so that it believes that `Class` already exists as instance of itself and inheriting from `Object`, then create `Object` and `Class` as normal classes

Three Steps:

1. manual creation of the instance that represents the class `Class` avec with
   - inheritance simulation (class instance variable from `Object` class)
   - only the necessary methods for the creation of the classes (new and initialize)
2. creation of the class `Object [Class new :name Object....]`
   - definition of all the method of `Object`
3. redefinition of `Class`

   `[Class new :name Class :super '(Object).....]`
   - definition of all the methods of `Class`

On The Road

- Context
- Examples of metaclasses
- Examples of programming with metaclasses
- Previous Approaches: Loops, Smalltalk
- Building your own metaclass kernel: ObjVlisp
- Examples: Playing with ObjVlisp

- Metaclasses are powerful but
- Problems with composition
- Problems with property propagation
- Clos's solution
- SOM's solution
- Smalltalk's solution
- NeoClasstalk's solution
- Conclusion
- Bibliography

Abstract Classes

“The rule to define a new metaclass is to make it inherit from a previous one” [Cointe’87]

Prb. Abstract classes should not create instances
Sol. Redefine the new method

Metaclass Definition:

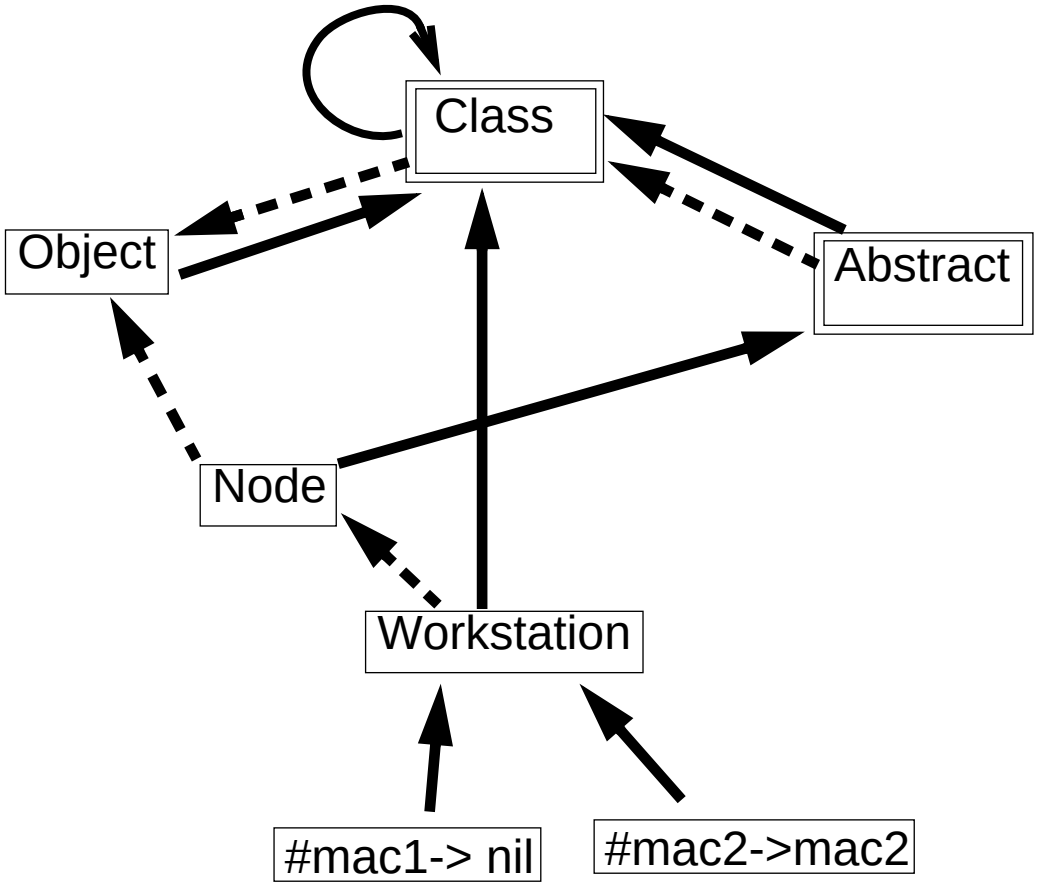
```
[Class new
   :name Abstract
   :supers '(Class)
   :methods '(new (lambda (self initargs)
                    (self error "Cannot create instance of class %s“ self name))]
```

Metaclass Use:

```
[ Abstract new :name Node :supers '(Object) ....]
[ Node new ]
-> Cannot create instance of class Node

[ Abstract new :name Abstract-Stack  :supers '(Object) ....]
```

Abstract

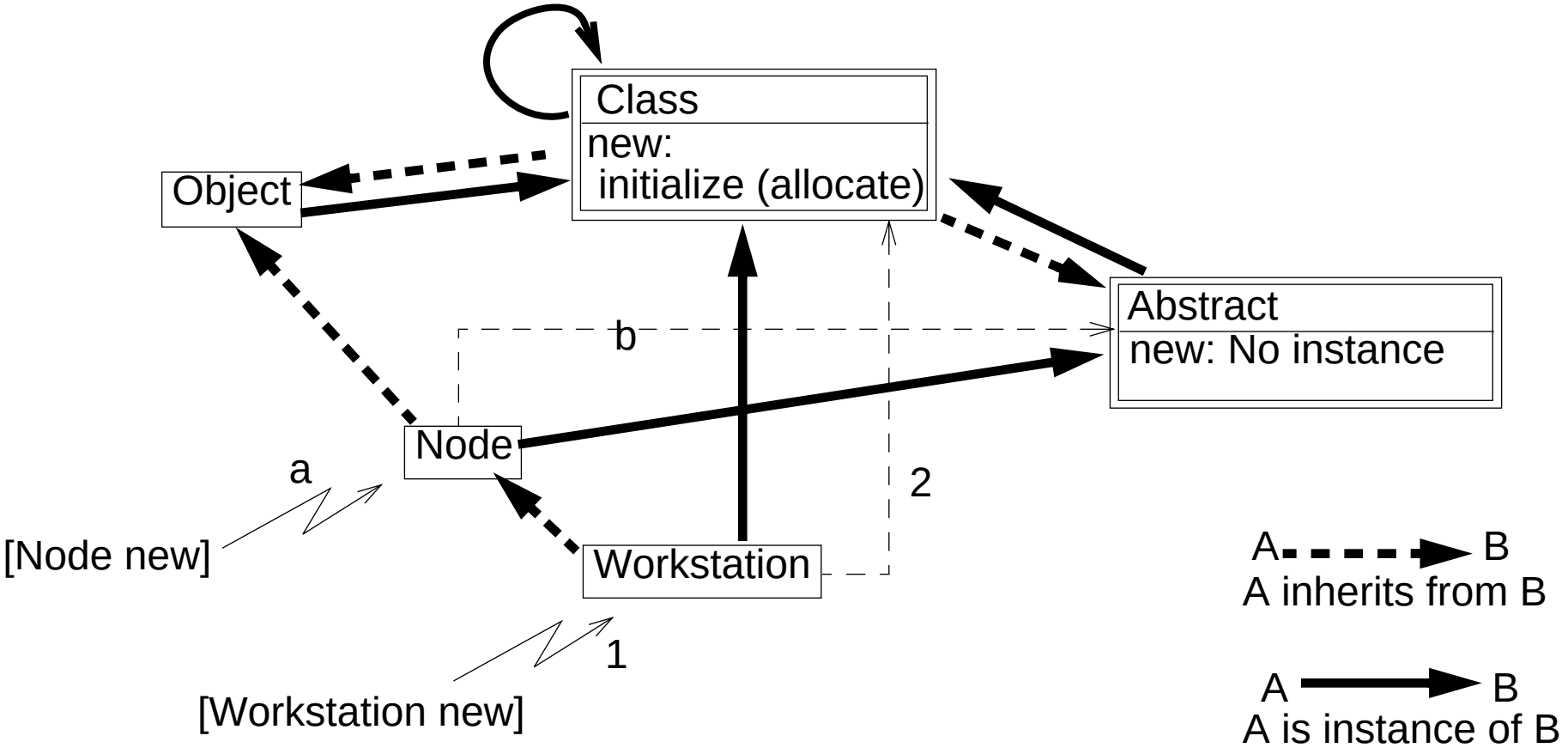
- Abstract is a class -> It is instance of `Class`
- Abstract define class behavior -> It inherits from `Class`



A - - - -> B
A inherits from B

A ——> B
A is instance of B

Abstract Class and Method Lookup



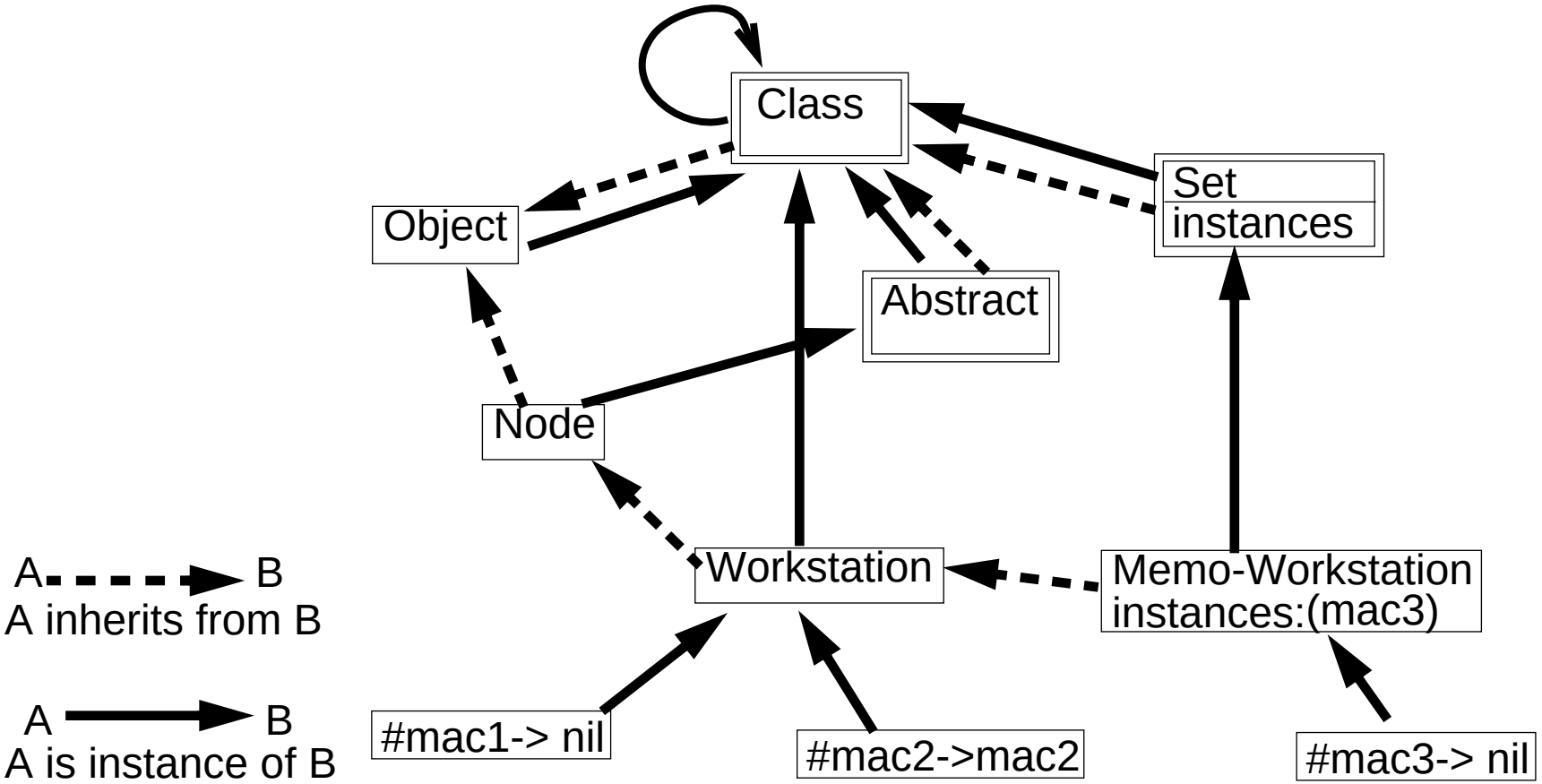
The Metaclass Set

Prb. How to access to all the instances of a certain class

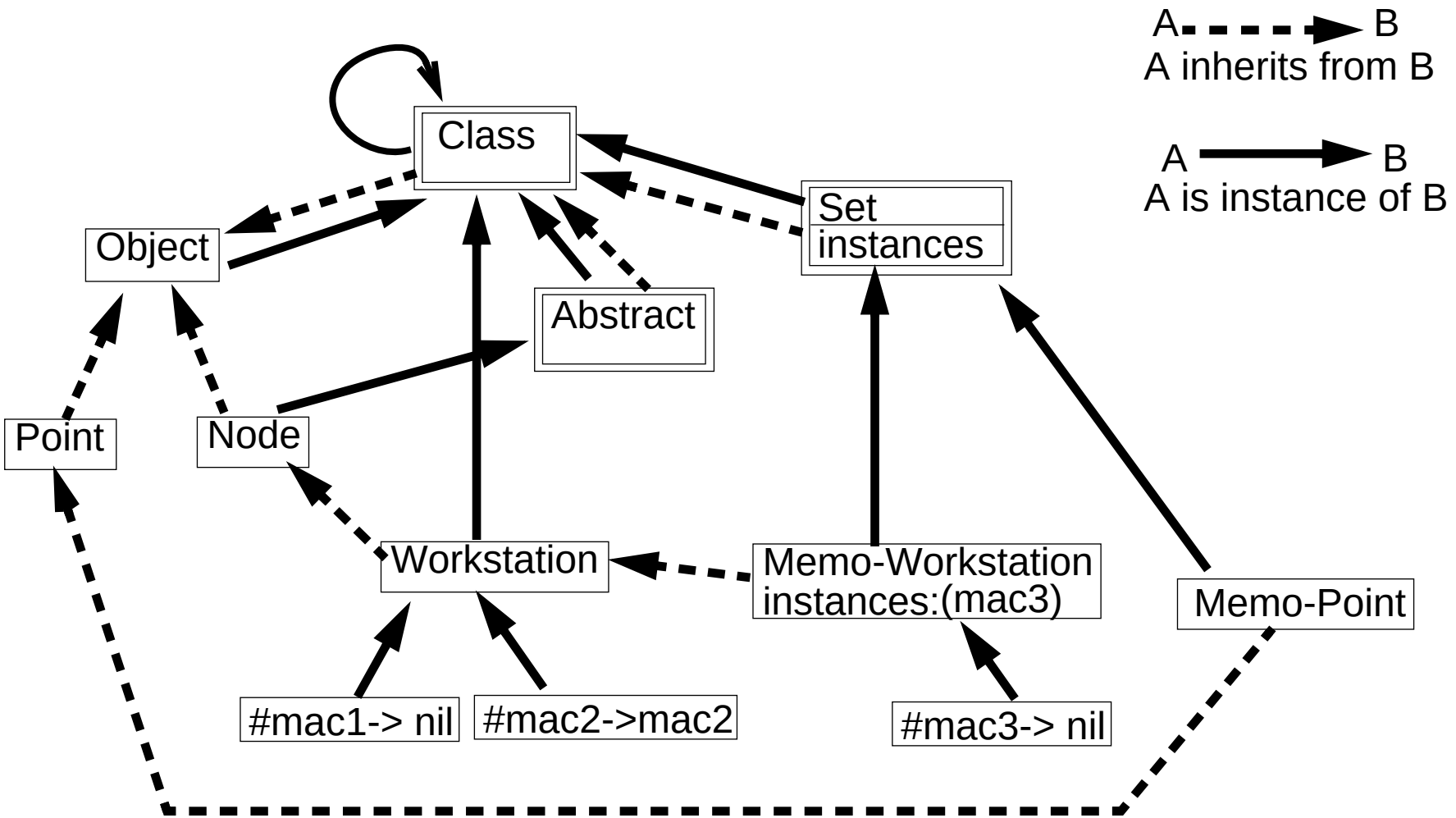
Sol. Store the instances when there are created.

```
[Class new
   :name “Set”
   :supers '(Class)
   :iv '(instances)
   :methods ‘(
      instances (lambda (self) (self iv-ref ‘instances))
      instances! (lambda (self newInstances)
            (self iv-set! ‘instances newInstances))
      initialize (lambda (self initargs)
            (super initialize initargs)
            (self instances! ()))
      new (lambda (self initargs)
            (let ((n-i (super new)))]
                (self instances! (cons n-i (self instances ))))]
```

Sets


Class
Object
Set
instances
Abstract
Node
Workstation
Memo-Workstation
instances:(mac3)
#mac1-> nil
#mac2->mac2
#mac3-> nil
A B
A inherits from B
A B
A is instance of B

Sharing Metaclasses



Zooming in: Creation of Memo-Point (i)

Remember: (apply (lookup selecteur (class-of receveur) receveur)
receveur args)

[Set new :name Memory-Point :supers '(Point)]
(apply (lookup 'new (class-of Set) Set) Set '(:name Memo-Point :supers '(Point)))
(apply (lookup 'new Class Set) Set '(:name Memo-Point :supers '(Point)))

New : [[Set allocate] initialize '(:name Memo-Point :supers '(Point))]

[Set allocate]
(apply (lookup 'allocate (class-of Set) Set) Set)
(apply (lookup 'allocate Class Set) Set)
Allocate -> #(Set nil nil nil nil nil nil)

Zooming in: Creation of Memo-Point (ii)

[#(Set ()...()) initialize '(:name Memo-Point :supers '(Point))]
(apply (lookup 'initialize (class-of #(Set nil...nil) #(Set nil...nil)
#(Set nil...nil) '(:name Memory-Point :supers '(Point)))
.... (lookup 'initialize Set #(Set nil...nil) ....

initialize method is not found in the class Set => we search in supers Set : Class

.... (lookup 'initialize Class #(Set nil...nil) ....

Initialize:

[super initialize ...] 2

Memory-Point class is an object. super looks in the superclas of Class (Class in whihc we found it) and not in Set

(inherit-iv ...) 3

Memory-Point is a class

2 (apply (lookup 'initialize Object #(Set nil...nil))
#(Set nil...nil) '(:name Memory-Point :supers '(Point)))
-> #(Set Memory-Point '(Point) nil nil nil)
3 #(Set Memory-Point '(Point) (class x y) nil nil)

On The Road

- Context (differences between compiled languages..definitions)
- Examples of usefull metaclasses (final, abstract....)
- Examples of programming with metaclasses (client point of view)
- Previous Approaches: Loops, Smalltalk
- Building your own metaclass kernel: ObjVlisp
- Examples
- Metaclasses are powerful but
- Problems with composition

- Problems with property propagation
- Clos's solution
- SOM's solution
- Smalltalk's solution
- NeoClasstalk's solution
- Conclusion
- Bibliography

4. Open Implementation: the CLOS MOP Example

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

Goals of this Lecture

- CLOS in a Nutshell
- CLOS MOP overview and example
- Difference between a reflective language and an open language
- Lessons learnt in the MOP Design
- Open Implementation Design Guidelines

CLOS

- Integration of object-orientation and functional style
  - Generic function, multiple discrimination and not receiver and message based, types and classes
- Take into account other Lisp OO like languages (Flavors, Loops)
  - migration path
- Small (they failed a bit) but extensible
  - CLOS MOP: essential language entry points are externalised

CLOS in a nutshell

Essential

- Class based
- Multiple Inheritance (with graph linerization)
- Multiple argument discrimination for method selection
- Methods associated with multiple classes
- Methods combined to be executed
- Generic function: group of method having the same “name”

Too much details:

- specializers (eql instance based method selection)
- argument-precedence-order (changing the weight of classes for method selection)
- default-initargs (default values for instance variable redefinable via inheritance)
- auxillary methods (around, before, after methods)
- method combination (how to compose the results of the methods selected for a given set of arguments)
- automatic accessors and initialization per instance variables

Class Definition

- In its simplest form:

```
(defclass rectangle ()
     ((height :initarg :start-height
              :initform 5
              :accessor height)
      (width  :initform 8
              :writer width)))
```

- Other possibilities

```
:allocation (per instance, shared among all instances)
specification of class defautl values inherited
```

Instance Creation

```
(setq r1 (make-instance ‘rectangle
                :start-height 25))
(height r1)
   -> 25
(width r1)
   -> 8
```

Encapsulation and Attribute Accesses

- Accessors can be created automatically
  - ☞ :accessor

```
(height r1)
(setf (height r1 75)
```

- Attributes can always be accessed using slot-value

```
(slot-value r1 'height)
(setf (slot-value r1 'height) 75)
```

- Accessors are defined in terms of slot-value
- Accessors are preferred style

Inheritance

- Simple

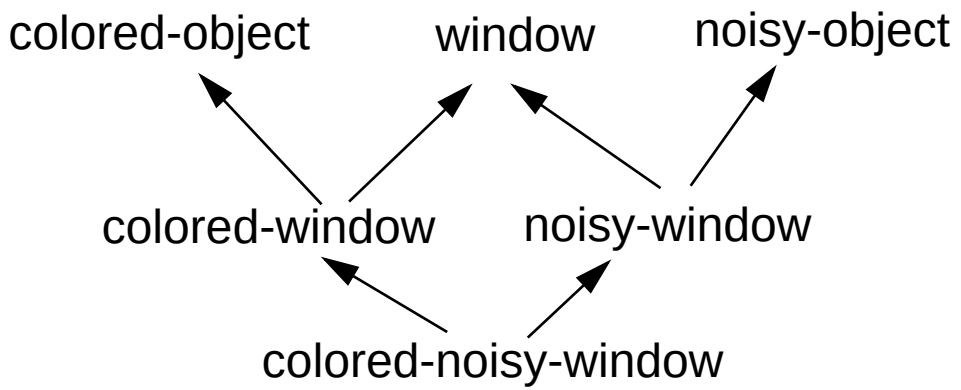
```
(defclass color-rectangle (rectangle)
   ((color  :initform ‘red
            :initarg :color
            :accessor color)
    (clearp :initform nil
            :initarg :clearp
            :reader   clearp)
    (height :initform 100)))
```

- Multiple

```
(defclass color-mixin ()
   ((color :initform ‘red :initarg :color :accessor color)))

(defclass color-rectangle (color-mixin rectangle)
   (clearp :initform nil
            :initarg :clearp
            :accessor clearp)
    (height :initform 100)))
```

Multiple Inheritance Conflict Resolution

- Which methods of the superclasses should be invoked using call-next-method (super equivelant)
- How multiple instance variables over the inheritance graph are accessed? (if window has an instance variable, colored-noisy-window still only has one)
  - ☞ graph linearization




```
(class-precedence-list (find-class 'colored-noisy-window))
   -> (colored-noisy-window colored-window noisy-window window noisy-object colored-
object standard-object t)
```

Generic Function

A generic function describing all the methods named paint taking two arguments

```
(defgeneric paint (shape medium))
```

- Holding bag of methods having the same name, number of argument but different types and different qualifier (instance based, before, after around, normal method)
- Not strongly defined in classes because of multiple discrimination

Method Definition (i)

```
1 (defmethod paint ((shape rectangle) medium)
   (vertical-stroke (height shape) (width shape) medium))

2 (defmethod paint ((shape circle) medium)
   (draw-circle (radius shape) medium))

(paint r1 *standard-display*)     -> 1
```

- ☞ Discriminating only on one single argument ->Java, Smalltalk like

```
3 (defmethod paint ((shape color-rectangle) medium)
     (if (not (clearp shape))
          (call-next-method))
```

- ☞ invoking an overriden method

(Method) Generic Function Application

```
4 (defmethod paint ((shape rectangle) (medium vector-display))
   ...)
5 (defmethod paint ((shape rectangle) (medium bitmap-display))
   ...)
6 (defmethod paint ((shape rectangle) (medium optimized-bitmap-stream))
   ...)
7 (defmethod paint ((shape circle) (medium ps-stream))
   ...)
8 (defmethod paint :after ((shape rectangle) medium)
      (log paint rectangle))
```

- ☞ 1,2,3,4,5,6,7 are primary methods
- ☞ 8 is an auxiliary method

Applying a generic function:

From all the methods, an effective method is created:

- ❑ Selecting the applicable methods to a given set of arguments
- ❑ Ordering them
- ❑ Applying them

Method Selection

- The methods are sorted according to the type of their first argument, then they ordered according to the second argument....

```
(paint r1 *bitmap*)
   -> selction of 5 1
(paint r1 *optimized-bitmap*)
   -> selection 6 5 1
```

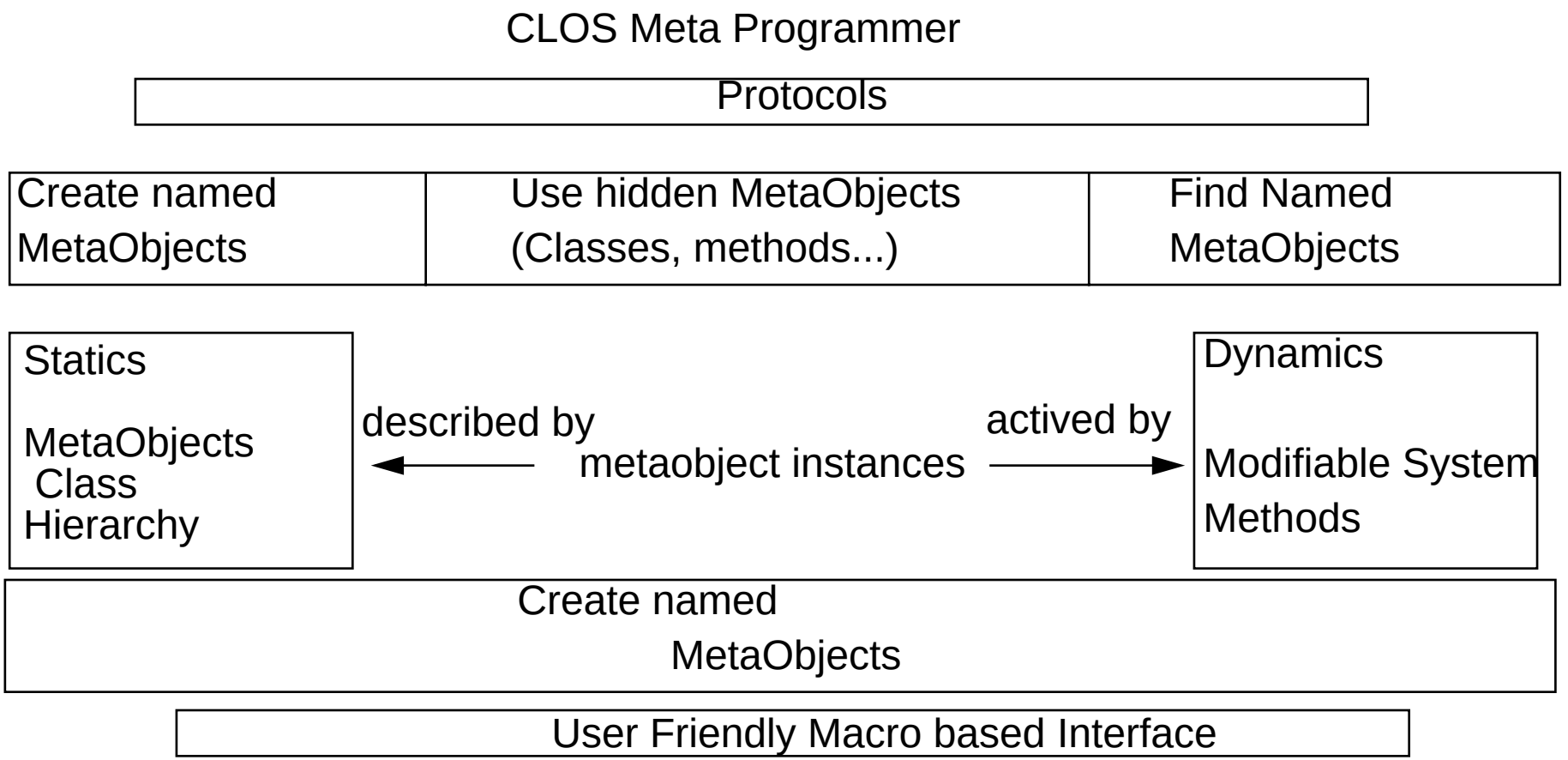
Effective method application leads to execute:

- All the before methods are invoked in decreasing order
- Most specific primary method (6 in the second call), other if call-next-method is used
- All the after methods are invoked in increasing order

Why CLOS MOP?

“Traditionally, languages have been designed to be viewed as black box abstractions; end programmers have no control over the semantics or implementationof these abstractions. The CLOS Mopon the other hand, “opens up” the CLOS abstraction, and its implementation to the programmer. The programmer can, for example, adjust aspects of the implementation strategy such as instance representation, or aspects of the language semantics such as multiple inheritance behavior. The design of the CLOS MOP is such that this opening up does not expose the programmer to arbitrary details of the implementation, nor does it tie the implementor’s hand unnecessarily-- only the essential structureof the implementation is exposed” [Kiczales’92a]

Meta Programming in CLOS



CLOS was too big!

Lot of could have been dropped and reintroduced if wanted using the CLOS MOP

- Instance based methods (eql) , auxiliary
- Method combination,
- argument-precedence-order option,

.

.

- slot-filing initargs, default initargs

.

.

.

.

- multiple inheritance, multi methods

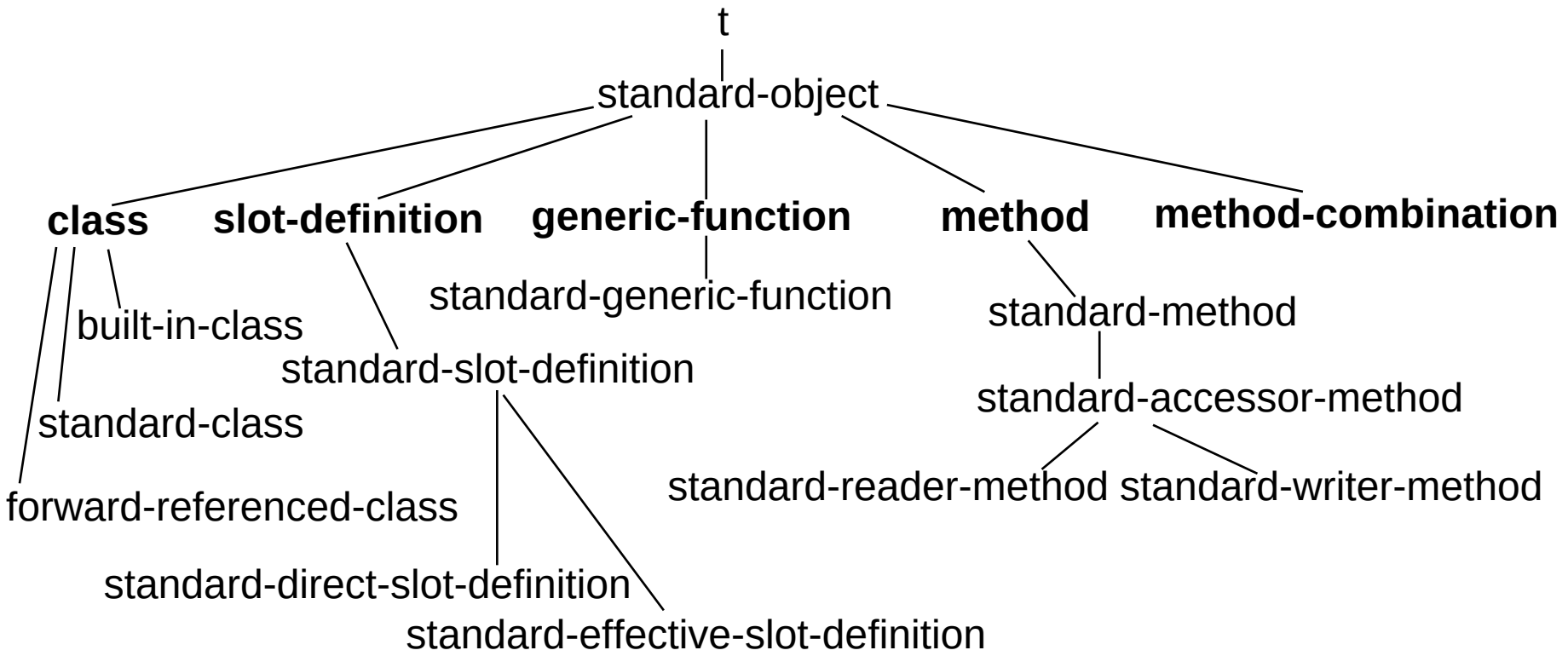
5 MetaObjects

- Classes
  - instance creation: make-instance
  - instance allocation: allocate-instance
  - class initialization: initialize-instance
  - instance variables storage and accesses: slot-value-using-class, (setf slot-value-using-class)
  - finalize-inheritance
- Methods
  - apply method
  - extra-method-bindings
- Generic Functions
  - apply-generic-function
- Slots
  - slot-boundp
- Method combinations

Static Elements

5 Metaobjects:

- Class, Method Combination (Semantics of method calls regarding inheritance)
- Method and Generic Function
- Slot (attribute)



Structure Protocols (i)

- global queries not attached to any meta-entities

  `find-class, find-generic-function, find-method`

  `ensure-generic-function, ensure-class, ensure-method,`

- User interfaces

  `defclass, defgeneric, defmethod`

Structural queries associated with meta-entities

- Object

  `class-of, print-object, reinitialize-instance, slot-makeunbound`

- Class

  `class-name, class-slots,`

  `class-direct-subclasses, class-direct-superclasses`

  `class-direct-slots, class-direct-methods,`

  `compute-class-precedence-list, compute-slots,`

  `compute-effective-slot-definition`

  `class-finalized-p,`

Structure Protocols (ii)

- Generic Function

```
add-method, add-reader-method, generic-function-methods, generic-function-name,
```

- Method

```
method-body,method-environment,method-generic-function,method-more-specific-p,
method-qualifiers, method-specializers,
```

- Slot

```
slot-definition-initfunction, slot-definition-initargs, slot-definition-initform,
slot-definition-name, slot-definition-readers, slot-definition-writers
slot-boundp, slot-boundp-using-class,
slot-exists-p,
```

Extension Example

```
(defclass hash-table-representation-class (standard-class)
   ()) ; no extra instance variables

(defmethod allocate-instance ((c hash-table-representation-class))
   ...allocate a small hash-table to store the slot)

(defmethodslot-value-using-class((chash-table-representation-class)instanceslot-
name))
   ...)

(defmethodsetfslot-value-using-class((chash-table-representation-class)instance
slot-name newvalue))
   ...)

(defclass person ()
   (name age address...)
   (:metaclass hash-table-representation-class))
```

Dynamic Elements

- instance initialization and creation,
- class-change, instance updating
- finalization (inheritance)
- method selection, method invocation,
- slot access
  - ☞ are controlled by metaobjects and their protocols

Class Definition: Defclass

1 Syntax error checking
2 Canonicalize information
3 Obtain class metaobjects
(ensure-class, ensure-class-using-class)
3.1 Find or make instance of proper class metaobject class
`(make-instance, the :metaclass option)`
3.2 (Re)initialize the class metaobject (`(re)initialize`)
3.2.1 Default unsupilied keyword arguments/error checking
3.2.2 Check compatibility with superclass (`validate-superclass`)
3.2.3 Associate superclasses with this new class metaobject
3.2.4 Determine proper slot-definition metaobject class
`(direct-slot-definition-class)`
3.2.5 Create and initialize the slot-definition metaobjects
(`make-instance`, `initialize-instance`)
3.2.6 Maintain subclass lists of superclasses
(`add-direct-subclass, remove-direct-subclass`)
3.2.7 Check `default-initargs`
3.2.8 Initiate inheritance finalization (if appropriate)
(`finalize-inheritance`)
3.2.9 Create reader/writer methods
3.2.10 Associate them with the new class metaobjects

Instance creation

- Class responsibility:
  make-instance, allocate-instance, initialize-instance (for class creation)

```
(make-instance class)
   => (initialize (allocate-instance class))
```

- Object responsibility

```
(initialize-instance anObject)
```

- Changing class and updating instance

```
change-class
update-instance-for-different-class
```

Method Creation: Defmethod (i)

1. Syntax error checking
2. Obtain target generic function metaobject (`ensure-generic-function`, `ensure-generic-function-using-class`)
    - 2.1. Find or make instance of proper generic-function metaobject (`make-instance,:generic-function-class`)
    - 2.2 (Re)initialize the generic function metaobject (`(re)initialize-instance`)
        - 2.2.1 Default unsupplied keyword arguments/error checking
        - 2.2.2 Check lambda list congruence with existing methods
        - 2.2.3 Check argument precedence order spec against lambda list
        - 2.2.4 (Re)define any old ‘initial methods’
        - 2.2.5 Recompute the generic function’s discriminating function (`compute-discriminating-function`)
3. Build method function (`make-method-lambda`)

Defmethod (ii)

4 Obtain method metaobject

- 4.1 Make instance of proper method metaobject class
  - (`make-instance, generic-function-method-class`)
- 4.2 Initialize the method metaobject
  - (`initialize-instance`)
  - 4.2.1 Default unsupplied keyword argument/error checking

5 Add the method to the generic function

(`add-method`)

- 5.1. Add method to the generic function's method set
- 5.2. Recompute the generic function's discriminating function
  - (`compute-discriminating-function`)
- 5.3. Update discriminating function
- 5.4. Maintain mapping from specializers (classes) to methods
  - (`add-direct-method`)

Method lookup and apply protocol

generic function call

(`apply-generic-function`)

1 invoke the generic function's discriminating function

1.1 Find out which methods are applicable for the given arguments

(`compute-applicable-using-classes`,
`compute-applicable-methods`,
`methods-more-specific-p`)

1.2 Combine the methods into one piece of code

(`compute-effective-method`)

1.3 Execute the combined method

(`method-function-applier, apply-methods,`
`apply-method, extra-function-bindings`)

Apply Protocol Example

- Counting the calls of a method
  - ☞ Define a new class of method and specialise apply-method

```
(defclass counting-method (standard-method)
   ((numberOfCalls :initform 0 :accessor numberOfCalls)))
(defmethod apply-method :before ((method counting-method) args next-methods)
      (incf (numberOfCalls method)))
```

  - Define new method of the right class or (depending on the implementation) change the class of certain methods

```
(defgeneric ack (x)
   (:method-class counting-method)))
(defmethod ack (x)
   t)
(defmethod ack ((i integer))
   1)
(ack 1) -> 1
(ack anObject) -> t
(numberOfCalls (find-generic-function #'ack)) -> 2
(numberOfCalls (find-method (find-generic-function #'ack) ((integer)) ()) -> 1
```

Apply Protocol Remark

- The generic function has the responsibility of class methods specification
- We cannot specify the class of a method at the method level
- Dynamically changing the class of a generic function was not allowed (at leats in the MOP description)

```
:generic-function-class
:method-class
```

are only associated with defgeneric

Slot Access Protocol

The class has the control over its attributes

- How to store and access them

```
(slot-value object slotname)
```

calls or has semantics defined by

```
(slot-value-using-class class instance slotname)
```

```
((setf slot-value) value object slotname)
```

calls or has semantics defined by

```
((setf slot-value-using-class) value class instance slotname)
```

1. Check for existence of slot

   `slot-exists-p, slot-missing`

2. Check for slot being unbound

   `slot-unboundp, slot-boundp-using-class`

3. Making a slot unbound

   `slot-makunbound, slot-makeunbound-using-class`

Finalize Inheritance

1 Compute the class precedence list
    (`compute-class-precedence-list`)
2 Resolve conflicts among inherited slots with the same name
    2.1Determine proper effective slot definition metaobject class
        (`effective-slot-definition-class`)
    2.2Create the effective slot definition metaobjects
        (`make-instance`)
    2.3 Initialize the effective slot definitions
        (`initialize-instance`,
        `compute-effective-slot-definition`)
    2.4 Associate them with the class metaobject
3Enable/Disable slot access optimizations
    (`slot-definition-elide-access-method-p`)

Open Implementation and Reflective Languages

Smalltalk is reflective but

- does not have a MOP
  - Programming and meta-programming are mixed
  - e.g., knowing that methods are stored into a method dictionary is not necessary for programming. This is a meta-information.
  - Stripping image is difficult.
  - Implementor of VM cannot optimize completely.
  - Implementors could provide several optimized environments
  - Firewall 93 was a declarative Smalltalk where hello world took 10 k

5. Open Implementation Design Issues

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

Goals of this Lecture

- Lessons learnt in the MOP Design
- Open Implementation Design Guidelines

Locality in MOP Design

- Feature Locality
  - MOP should provide access to individual features of the base language
- Textual Locality
  - The programmer should be able to indicate, using convenient reference to their base program, what behavior they would like to be different
- Object Locality
  - The programmer should be able to affect the implementation on a per-object basis.
- Strategy Locality
  - The programmer should be able to affect individual parts of the implementation strategy.
- Implementation Locality
  - Extension of an implementation ought ot take code proprotional to the size of the change. A resonably good default implementation must be provided and the programmer should be able to describe their extension as an incremental deviation from that default.

Open Implementation Design Guidelines

Stepping back from CLOS and its MOP and generalization

Black-box abstraction:

A module should expose its functionality but hide its implementation

Pros

- Localization of changes
- Level of abstraction
- Modularization easier
- Reuse easier

Cons

- Performance problems
- Needs to code around

Whereas black-box modules hide all aspects of their implementation, open implmentation modules allow clients some control over selection of their implementation strategy while still hiding many true details of their implementation. [Kiczales 97]

Quality in interface designs

from [Hoffman 90]

- consistent (e.g., same parameter passed always at the same place)
- essential (e.g., each service is offered in only one way)
- general
- minimal (e.g, each function provides one operation)
- opaque (e.g., the interface hides the way the module has been implemented)

Set Module: Design A

```
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)
```

- Simple, Consistent, Essential, General, Minimal, Opaque

But is the implementation performing well for?

- few/many elements
- frequent/unfrequent removal
- frequent/unfrequent addition

Set Module: Design B

```
makeSet(usage)
makeSet()
insert(item, set)
delete(item, set)
isIn(item, set)
map(function, state, set)
```

Use

```
makeSet (“n=10000,insert=lo,delete=lo,isIn=hi”)
makeSet (“n=5,insert=hi,delete=hi”)
```

- Same property than design A and still hidding implementation
- Only a small change in the interface
- New functionality optional
- Well-bounded effect (only the set created by the call affected)
- Use of the new functionality orthogonal to previous one: distinction between client use and implementation strategy

First Guideline

Separation of Use from Implementation Strategy Control

Open Implementation module interfaces should support a clear separation between client code that uses the module's functionality (use code) and client code that controls the module's implementation strategy (ISC code)

Second Guideline

Open implementation module interfaces should be designed to make the ISC code optional, make the ISC code easy to disable, and support alternative ISC codes for one piece of use code.

Example: High Performance Fortran (for efficient parallel processing)

```
Real A(1000,1000) B (998,998)
!HPF$ ALIGN B(I,J) WITH A(I+1,J+1)
```

ISC coded into comments

- use/ISC code has clear separation
- ISC code is optional
- ISC code easy to disable
- HPF doesnot support multiple ISC for the same piece of code but easy to implement

Third Guideline

Scope control

Open implementation module interfaces should be designed to allow the scope of influence of ISC code to be controlled in a way that is both natural and sufficiently fine-grained

```
s1 = makeSte(“n=1000“)
for i = 1 to 700 do
   insert(s1 , i +1)

s2 = makeSet(“n=5“)
   insert(s2, 5)
   insert(s2,6)
```

Subject Matter

Design B has some weaknesses

- ❑ client programmer can mis-describes and get a solution worse than the default
- ❑ no garantee that they will get an optimal implementation strategy

Design C

```
makeSet(strategy)
```

Use

```
makeSet("LinkedList"), makeSet("BTree")
```

ICS can be about different subject matter

- – the client program's behavior (design B),
- – module implementation strategy (design C), or
- – performance requirements

No automatic solution

☞ Analysis steps in the process of selecting implementation strategy

client use code ---> client usage profile --->
client performance requirements ---> module implementation strategy

Fourth Guideline

Implementation Details must be hidden

Open Implementation module interfaces should be designed to pass only essential implementation strategy information

Design D

- Design C is limited to the implementation strategies provided by the module
- Might be not flexible enough

```
class mySet (Set) {
   method insert...
   method delete...
   method isIn...}
```

Use

```
makeSet("mySet")
```

- Programmatic interfaces tend to be less robust
  - locality is extremlly important
  - Layered interface

Last Guideline: Layered Interfaces

Client

- No ISC code -> get default one
- Select from built-in ones
- Provide a new strategy

When there is a simple interface that can describe strategies that will satisfy a significant fraction of clients, but it is impractical to accomodate all important strategies in that interface, then the interfaces should be layered

90%/10% Rule

90% of the clients use the default strategy

10% write new ISC code

90% of 10% select in the built-in strategies

1% should provide a new strategy

But this is a really needed one!!!!

6. Comparing Reflection in CLOS, Smalltalk and Java

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Winter Semester 2000-2001

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

Sorry but this is your work!

Material you can use

- Java: Reflection API, OpenJava
- Smalltalk: Smalltalk a Reflective Language, Smalltalk 80 the Language, VisualWorks
- CLOS: The Art of the MetaObject Protocols, Paepcke Paper,
  - ☞ www.franz.com download a trial version.
- Other documents available for you in my office

Some Criterias

- Which entities?
- Introspection and/or Intercessory?
- Which aspects?
- Is the causal link respected? Only representation of data or can we affect them?
- Level of power,
  - for example try to invoke method m of class A on an instance of the class B subclass of A in Java => m defined on B is called
  - Use valueWithReceiver... in VW

7. Implementing Message Passing Control in Smalltalk: an Analysis

Dr. Stéphane Ducasse
Software Composition Group
University of Bern
Switzerland

Email: ducasse@iam.unibe.ch
Url: http://www.iam.unibe.ch/~ducasse/

Outline

- Limited Survey
- Method Wrappers in Use
- Opening the Box
- DoesNotUnderstand
- Method Wrapper
- Instance based Behavior

Why Controling Message?

- Application Analysis and introspection
  - Do not require program instrumentation (imagine in C++!!!)
  - Dynamic traces, analysis of collaborations, hints for distribution
- Language Extension
  - Distribution
  - Security
  - Atomic Data Types
  - Multiple inheritance
  - Instance based programming
  - Object connections
- New objects models
  - Active object model
  - Concurrent Smalltalk
  - Composition Filters
  - New Meta Models (codA)

Controling What Exactly!

Which objects are controlled?

- Instance based: One instance
- Group based: A group of objects
- Class-based All instances of a class

What methods are controlled?

- All methods
- Unknown methods
- Selected methods

Technical quality of the control?

- Existing Smalltalk systems and tools
- Not another interpreter with an explicit send message!
- Not only pre and post methods
- Changing arguments (marshalling...)

Who does the control?

- The receiver
- Another object

A Limited Survey

- CLOS Mop: clean, integrated into the MOP
- Smalltalk: everythign is there but not polished
  - do it yourself syndrome!
  - MethodWrappers (http://st-www.cs.uiuc.edu/~brant/)
  - Some well-known techniques
- Open C++ (first version, runtime, second version precompiler based)
- OpenJava (class loader annotations) @@Find paper @@

CLOS Example (i)

- Counting the calls of a generic function
  - ☞ Define a new class of generic function and specialise apply-generic-function

```
(defclass counting-gf (standard-generic-function)
   ((numberOfCalls :initform 0 :accessor numberOfCalls)))

(defmethod apply-generic-function :before ((gf counting-gf) args)
      (incf (numberOfCalls gf)))
```

- Counting the calls of a method
  - ☞ Define a new class of method and specialise apply-method

```
(defclass counting-method (standard-method)
   ((numberOfCalls :initform 0 :accessor numberOfCalls)))

(defmethod apply-method :before ((method counting-method) args next-methods)
      (incf (numberOfCalls method)))
```

CLOS Example (ii)

- Define new method of the right class or (depending on the implementation) change the class of certain methods

```
(defgeneric ack (x)
   (:generic-function-class counting-gf)
   (:method-class counting-method)))
(defmethod ack (x)
   t)
(defmethod ack ((i integer))
   1)
(ack 1)
   -> 1
(ack anObject)
   -> t
(numberOfCalls #'ack)
   -> 2
```

CLOS Example (iii)

- Separation between programmer and meta programmer job
- MOP entry points

```
apply-generic-function
   compute-applicable-methods-using-classes
      method-more-specific-p
   apply-methods
      apply-method
         extra-function-bindings
```

- Optimized the following way: separate parts that change from part that don't

```
(apply-methods gf args methods)
   <=>
(funcall (compute-effective-method-function gf methods) args)

(apply-method method args next-methods)
   <=>
(funcall (compute-method-function methods) args next-methods)
```

A Coverage Tool in Smalltalk

@@MW or Michel tools@@

Smalltalk: Do It Yourself Syndrome

- Reflective sure !!
- But not a well defined MOP
- Full implementation details on the shoulder of the programmer

Extra Criteria

- Reproductible easily
- Cost of implementation
  - ☞ at the normal level of programming or fighting with bits
- Cost of activation
  - ☞ (recompile or not)
- Run-time cost
- Integration into the programming environment
  - ☞ is control visible for the programmer?

Smalltalk Basic Reflective Tools

Reflective but the VM has the control

- the way the objects are represented in memory
- how messages are handled.

Programmer possibilities

- Instance variable access (instVarAt:)
- Compiling class on the fly (subclass:instanceVariable...)
- Compiling method on the fly (compile:notifying:)
- Changing inheritance chain (superclass:)
- Changing reference between objects (become:, becomeOneWay:)
- Changing class (changeClassToThatOf:)
- Message reification (only for error handling)
- Stack Reification (sender, receiver...)
- Methods are objects (mclass, sourceCode, bytes)
- Object methods can be invoked (valueWithReceiver:arguments:)
- Lookup can be called (perform:with:)

6 Techniques

- Source code modification

```
setX: t1 setY: t2
    ...
    Original Code
    ...
```

  - reparsed, recompiled for installation and desintallation
  - not applicable to stripped image
- Byte code extension
  (add a new byte code in the VM)
  - dialect specific
- Byte code modification
  (insert a new byte code directly in the code of the method)
  - dialect specific

Deeply evaluated

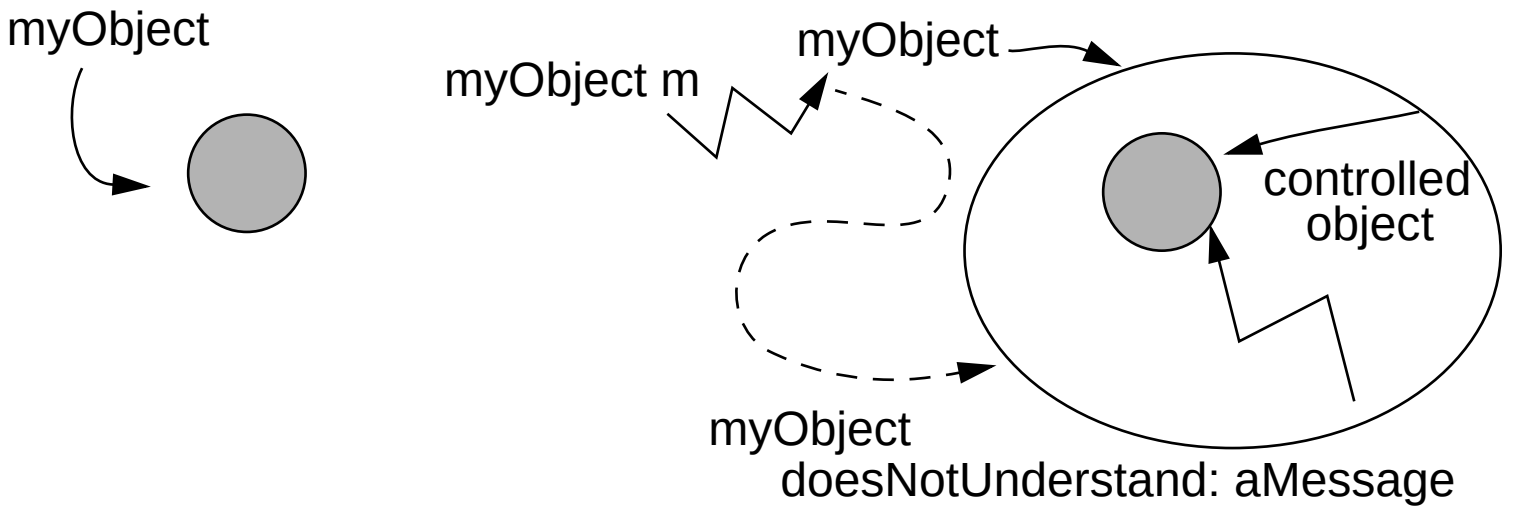
- Error handling specialisation
- Anonymous classes
- Method Wrappers

Unknown Messages

Context: When an object does not understand a message, it sends doesNotUnderstand: with a reification of the message

Solution:

- define a minimal object that raises doesNotUnderstand: for every message
- wrap an object in a minimal object
- specify control semantics by specializing the doesNotUnderstand: method



Creating a MinimalObject

- Object that does not inherit from Object `nil subclass: MinimalObject`
  - ☞ does really not work because we cannot debug, print....

The trick: (1) creating a normal class

```
Object subclass: MinimalObject
      instanceVariableNames: 'controlledObject'
```

(2) setting the inheritance to `nil`,

(3) copying some minimal behavior from Object.

```
MinimalObject class>>initialize
   superclass := nil.
   #(doesNotUnderstand: error: ~~ isNil = == printString printOn: class inspect ba-
sicInspect basicSize instVarAt: instVarAt:put:)
         do: [:sel | self recompile: selector from: Object]
```

- Example of possible control

```
MinimalObject>>doesNotUnderstand: aMessage
   ...
   controlledObject perform: aMessage selector
                    withArguments: aMessage arguments
   ...
```

Wrapping anObject

Wrapping

```
MinimalObject class>>newOn: anObject
    | x e |
    x := anObject.
    e := self new.
    x become: e.
    x object: e.
    ^x
```

Unwrapping

```
MinimalObject>>uninstall
    | x |
    x := controlledObject.
    controlledObject := nil.
    x become: self
```

Evaluation

- Instance based control controlling all methods (even not known a priori)
- Simple
- Slowest solution
  - Message reified + Exception Handling
  - even if doesNotUnderstand: is cached in certain VM
- Installation: no recompilation

Known Problems

- Messages sent to self by the object itself are not controllable
- Messages sent to the object via reference to self
- Class control is impossible, cannot swap a class by an object
- Interpretation of minimal set of messages by the minimalObject and not the controlled object.

```
anObject inspect => anObject controlledObject inspect
```

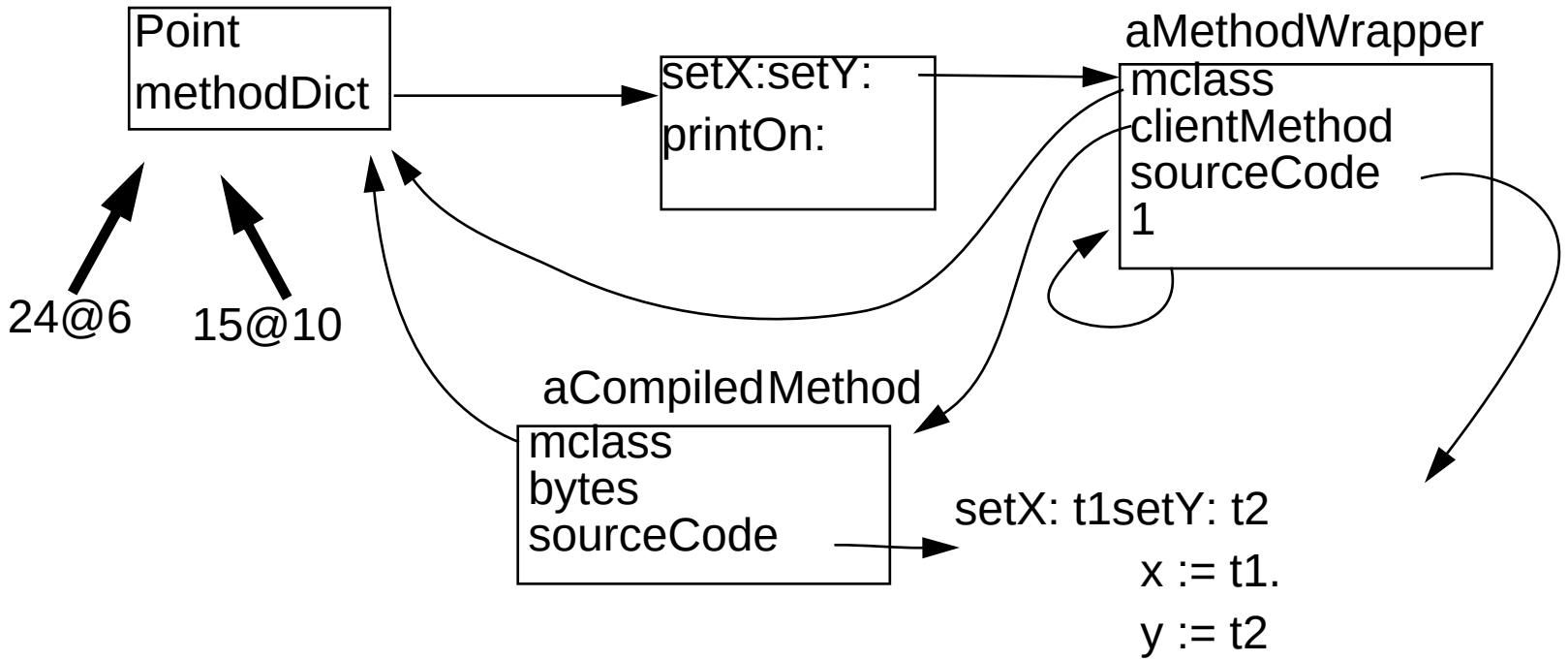
Method Wrappers: an Example

```
MethodWrapper variableSubclass: #CountMethodWrapper
   instanceVariableNames: 'count '

CountMethodWrapper>>class: aClass selector: aSymbol
   count := 0.
   ^super class: aClass selector: aSymbol

CountMethodWrapper>>valueWithReceiver: anObject arguments: anArrayOfObjects
   count := count + 1.
   ^clientMethod valueWithReceiver: anObject arguments: anArrayOfObjects

CountMethodWrapper>>count
   ^ count
```

Method Wrappers

The idea:

- substitute a method by a wrapper that has a reference to the original method
- wrapper has as source code the code of the original method
  - ☞ transparent for the programmer



Control

```
MethodWrapper>>valueWithReceiver: object arguments: args
    "This is the general case where you want both a before and after method, but if you want just a
    before method, you might want to override this method for optimization."
    self beforeMethod.
    ^[clientMethod valueWithReceiver: object arguments: args]
        valueNowOrOnUnwindDo: [self afterMethod]
```

To control the method originalSelector: on aClass the following code is automatically generated

```
aClass>>originalSelector: t1

    |t2|
    (t2:=Array new: 1) at: 1 put: t1.
    ^#() valueWithReceiver: self arguments: t2.
```

To have a way to refer to the method object itself and not the receiver of the message #() reserves some place byte code that is then latter filled with the method wrapper being installed.

MethodWrapper Optimization

Create method skeletons depending on number of parameters and then copy them

☞ no compilation needed

```
MethodWrapper class>>on: selector inClass: class
    | wrapper |
    (self canWrap: selector inClass: class) ifFalse: [^nil].
    wrapper := (self methods at: selector numArgs
                          ifAbsentPut: [self createMethodFor: selector numArgs]) copy.
    wrapper class: class selector: selector.
    ^wrapper

MethodWrapperclass>>createMethodFor: numArgs
    ^((MethodWrapperCompiler new) methodClass: self;
        compile: (self codeStringFor: numArgs)
        in: self
        notifying: nil
        ifFail: []) generate
```

MW method body

```
'valuevalue: t1 value: t2
    | t |
    (t := #Array new: 2)  at: 1 put: t1; at: 2 put: t2.
    ^#''The method wrapper should be inserted in this position'' valueWithReceiver: self arguments: t'
MethodWrapper class>>codeStringFor: numArgs
    "self codeStringFor: 2"
    | nameString tempsString |
    nameString := 'value'.
    tempsString := numArgs == 0
                    ifTrue: ['t := #()']
                    ifFalse: ['(t := #Array new: ' , numArgs printString , ') '].
    1 to: numArgs do: [:i |
                        nameString := nameString , 'value: t' , i printString , ' '.
                        tempsString := tempsString , (i == 1 ifTrue: [''] ifFalse: [';']) , ' at: '
                        , i printString , ' put: t' , i printString].
    ^nameString , '
    | t |
    ' , tempsString , '.
    ^'
        , self methodWrapperSymbol printString
            , ' valueWithReceiver: self arguments: t'
```

Installation

(MethodWrapper on: #blop inClass: Test) install

```
MethodWrapper>>class: aClass selector: sel
   | position |
   self at: self methodPosition put: self.
   position := self arrayPosition.
   position == 0 ifFalse: [self at: position put: Array].
   mclass := aClass.
   selector := sel

MethodWrapper>>install
   | definingClass method |
   definingClass := mclass whichClassIncludesSelector: selector.
   definingClass isNil ifTrue: [^self].
   method := definingClass compiledMethodAt: selector.
   method == self ifTrue: [^self].
   clientMethod := method.
   sourceCode := clientMethod sourcePointer.
   mclass addSelector: selector withMethod: self
```

MW Evaluation

- Transparent fro the programmer
- Class-based (all instance of a class are controlled)
- Selective (only certain methods are controlled)
- Run-Time Cost: less than doesNotUnderstand:
- Coding cost: Tricky so this is better to reuse the library

Exploiting VM Lookup Algorithm

The idea:

- Interposing between the object and its class a class that specializes certain methods to introduce the control.

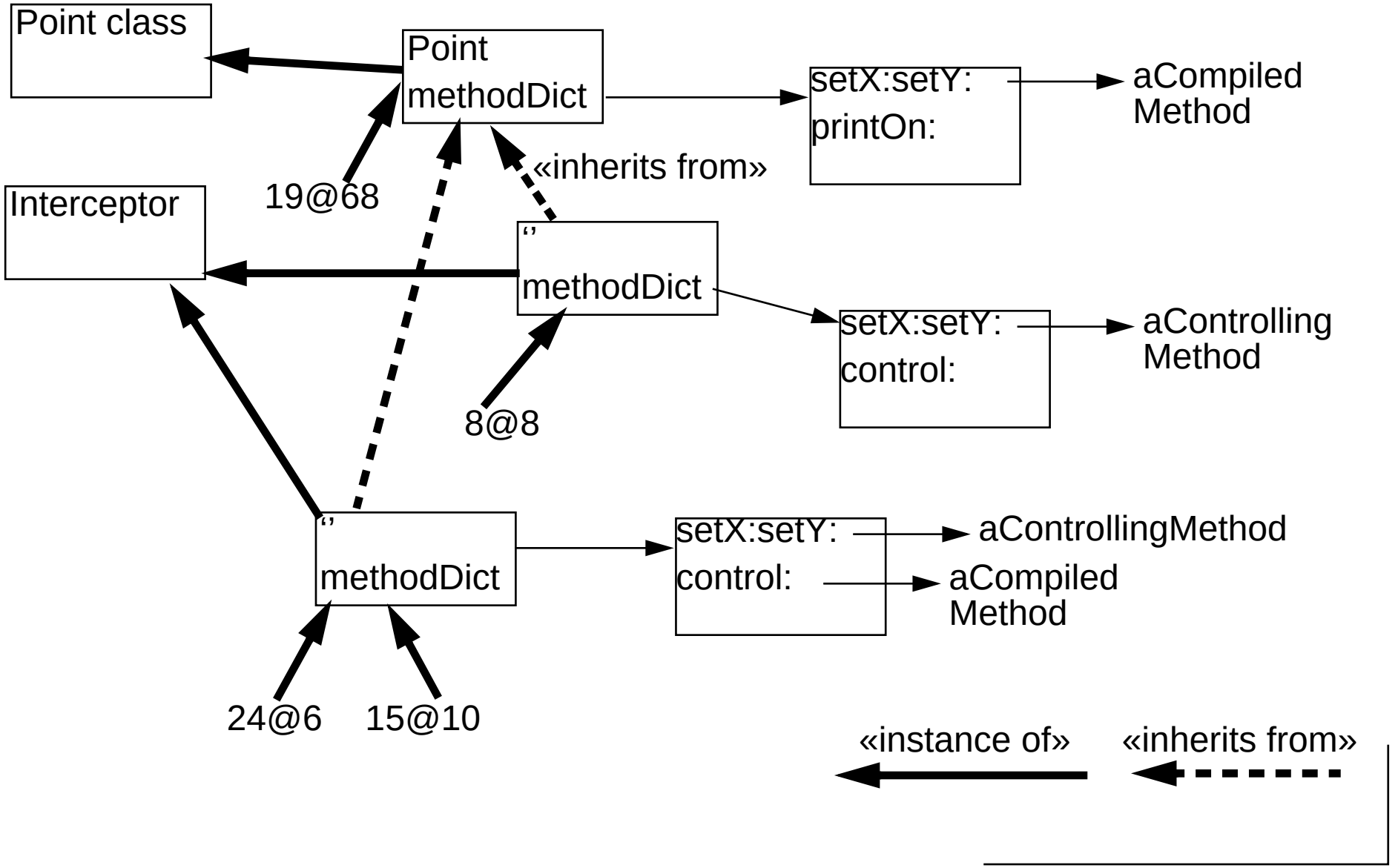
Solution 1

- Explicit subclassing + change the class of the controlled instance
  - Instance, group or class based, Selective method control
  - Without optimization: compile methods and classes
  - Polution of the class namespace for controling classes the programmer is aware of the control

Solution 2

- Implicit subclassing: creation of an anonymous class

Let's view it


Point class
Point
methodDict
setX:setY:
printOn:
aCompiled
Method
19@68
«inherits from»
Interceptor
"
methodDict
setX:setY:
control:
aControlling
Method
8@8
"
methodDict
setX:setY:
control:
aControllingMethod
aCompiled
Method
24@6
15@10
«instance of»
«inherits from»

Interceptor: Anonymous Classes

- Create an interceptor (a class): instance of Behavior
- Copy class description of the original class in the interceptor
- interceptor inherits from original class
- Compile in interceptor class class the methods needing control

```
InterceptorClass class>>takeControlOf: anObject
   | interceptor |
   (anObject isControlled)
      ifFalse: [interceptor := self new.
                interceptor conformsToThatClass: anObject class.
                interceptor installEssentialMethods.
                anObject changeClassToThatOf: interceptor new.].
   ^anObject

InterceptorClass>>conformsToThatClass: aClass
   "Return an instance of an anonymous class that is conforms to the class <aClass>"
   self setInstanceFormat:  (aClass format) ;
         superclass: aClass ;
         methodDictionary: (MethodDictionary new).
```

Let us think a bit

From the implementor point of view

- How to access the original class?

```
anObject class superclass
```

- How to access the anonymous class?

```
anObject class
```

But how can we access them in a conceptual manner?

- original class?

```
anObject class
```

```
anInterceptor>>class
   ^super class superclass
```

- interceptor?

```
anObject interceptor
```

```
anInterceptor interceptor
      ^ super class
```

Essential Methods

```
InterceptorClass>>installEssentialMethods
   "all the necessary methods to ensure right behavior of an interceptor class.
   It should be always invoked: Can be specialize but not overriden"
   self basicCompile: 'class  ^super class superclass'.
   self basicCompile: 'isControlled  ^ true'.
   self basicCompile: 'interceptor ^ super class'.
   self basicCompile: 'addSpecificMethod: aString
                          self interceptor compile: aString notifying: nil'.
   self  basicCompile: 'removeSpecificMethodWith: aSymbol
                          self removeSelector: aSymbol'

InterceptorClass>>compile: code notifying: requestor ifFail: failBlock
   "we redefine this method to ensure that essential methods such as #class, #interceptor,
   #isControlled will be never recompile on an interceptor class instance"
   |selector|
   selector :=  Parser new parseSelector: code.
   (self isEssentialMethod: selector)
      ifFalse: [ self basicCompile: code]

InterceptorClass>>basicCompile: code
   super compile: code notifying: requestor ifFail: failBlock
```

Naive Control Implementation (i)

Naive because we compile all the times (see optimization).

We want to generate the following code on the interceptor for a given method

```
'setX: t1 setY:  t2
   ^self interceptor control: self
                receiving: #setX:setY:
                withArgs: (Array  with: t1 with:  t2 )
                originalCall: [super setX: t1 setY:  t2]'

Interceptor>>installControlledMethod: aSymbol
   "control the method with selector <aSymbol>"
   self compile: (self generateSourceOfControlledMethod: aSymbol) contents
         notifying: nil ifFail: []
Interceptor>>generateSourceOfControlledMethod: aSymbol
   "generate the source of a controlled method"
   |methodCode signature|
   methodCode := WriteStream on: (String new: 32).
   signature := self generateSignature: aSymbol on: methodCode.
   methodCode cr ; tab.
   self generateBody: aSymbol withSignature: signature on: methodCode.
   ^methodCode
```

Naive Control Implementation (ii)

```
Interceptor>>generateSignature: aSelector on: methodCode
   "Return anArray containing at:1 signature and at:2 a string representing formal parameters"
   "self new generateSignature: #setX:setY: on: (WriteStream on: (String new: 32))
            -> #('setX: t1 setY:  t2 ' ' with: t1 with:  t2 ')"
   |  numArgs keywords parameters|
   parameters := WriteStream on: (String new: 10).
   keywords := aSelector keywords.
   methodCode nextPutAll: (keywords at: 1).
   (numArgs := aSelector numArgs) >= 1
      ifTrue:[parameters nextPutAll: ' with: t1'.
            methodCode  nextPutAll: ' t1 '.
            2 to: numArgs do:
               [:i | parameters nextPutAll: ' with:  t'; nextPutAll: (i printString) ; space.
               methodCode nextPutAll: (keywords at: i) ;
               nextPutAll: '  t'; nextPutAll: (i printString) ; space]].
   ^ Array with: (methodCode contents) with: (parameters contents).
```

Naive Control Implementation (iii)

```
Interceptor new generateBody: #setX:setY: withSignature: #'with: t1 with: t2'
      ->
^self interceptor control: self receiving: #setX:setY:
      withArgs: (Array  with: t1 with: t2) originalCall: [super setX: t1 setY: t2 ]
```

```
Interceptor>>generateBody: aSelector withSignature: aSignature  on: methodCode
   methodCode cr; tab;
            nextPutAll:  '^self interceptor control: self receiving: ';
            nextPut: $# ; nextPutAll: (aSelector asString) ; cr ;
            tab; tab; nextPutAll:'withArgs: (Array '; nextPutAll: (aSignature at: 2) ; nextPut: $) ;
            tab; tab ; nextPutAll:  'originalCall: [super ' ; nextPutAll: (aSignature at: 1) ;
                     nextPutAll: ' ] '; cr .
   ^ methodCode contents
```

- The original call could be called via super but it may happen that another object than the interceptor defines the control.
- [super setX...] is costly

Possible Optimization

Like Method Wrapper implementation

- To avoid compilation when installing the control
  - for each number of parameters skeletons of methods containing a call to the control can be created once, then copied and adjusted (change selector) in the instantiated interceptor class.
  - copy essential method instead of recompiling them

Evaluation

- Instance, group and class based control selective methods
- Simple but bugs during implementation may crash the system
- Efficient solution
- Installation: compilation but optimization is possible
- Good integration in the system (class is still the class)

Why A Mop for Smalltalk is Needed?

- Free the developer from doing everything himself
- Free the VM or meta-programmer to optimize the code
- ANSI Normalization -> declarative Smalltalk but no MOP
- MOP
  - instance variable representation
  - instance variable access
  - method control

Pratice

- Lab session: Implement Actalk [Briot89]
- Play with the MethodWrappers
  - Look at the coverage tools
- Play with anonymous class
  - Implement an instance based language

Selected Bibliography

Metaclasses

- [Bobrow'83] D.Bobrow and M. Stefik: "The LOOPS Manual, Xerox Parc, 1983.
- [Goldberg'83] A. Goldberg and D. Robson: "Smalltalk-80: The Language", Addison-Welsey, 1983.
- [Cointe'87] P. Cointe: "Metaclasses are First Class: the ObjVlisp Model", OOPSLA'87.
- [Graube'89] N. Graube: "Metaclass compatibility",OOPSLA'89, 1989.
- [Briot'89]J.-P. Briot and P. Cointe, "Programming with Explicit Metaclasses inSmalltalk-80", OOPSLA'89.
- [Danforth'94] S. Danforth and I. Forman: "Reflection on Metaclass Programming in SOM", OOPSLA'94.
- [Ledoux'96] T. Ledoux and P. Cointe, "Explicit Metaclasses as a Tool for Improving the Design of Class Libraries", ISOTAS'96, LNCS 1049,1996
- [Rivard'96] F. Rivard, "A New Smalltalk Kernel Allowing Both Explicit and Implicit Metclass Programming" OOPSLA'96 Workshop Extending the Smalltalk Language, 1996
- [Bouraqadi'98] M.N. Bouraqadi-Saadani, T. Ledoux and F. Rivard: "Safe Metaclass Programming", OOPSLA'98

Open Implementations

- [Kiczales'91] G. Kiczales, J. des Rivieres and D. Bobrow : "The Art of the Metaobject Protocol", MIT Press, 1991
- [Paepke'92] Object-Oriented Programming: The CLOS Perspective, MIT Press, 1992
- [Kiczales'92a] G. Kiczales: "Metaobject protocols - why we want them and what else they can do", in Object oriented Programming: the CLOS Perspective, MIT Press, 1992
- [Kiczales'92b] G. Kiczales: "Towards a New Model of Abstraction in the Engineering of Software", Proc. of IMSA'92 Workshop on Reflection and Meta-Level Architecture,1992
- [Kiczale'97] G. Kiczales, J. Lamping, C. Videira Lopes, A. Mendhekar and G. Murphy, Open Implementation Design Guidelines
- [Hoffman 90] D. Hoffman, On Criteria for Module Interfaces, IEEE Transactions on Software Engineering and Methodology, 1990, 16(5), 537--542

Other Related

- Maes OOPSLA'87
- [Mulet'94] P. Mulet, J. Malenfant P. Cointe, "Towards a Methodology for Explicit Composition of MetaObjects", Proc. of OOPSLA'95, 1995

Intercessory

- [Ducasse'99] S. Ducasse, "Message Passing Control Techniques in Smalltalk", JOOP, 1999
- [Rivard'96] F. Rivard, Smalltalk : a Reflective Language, REFLECTION'96,1996
- []Wrappers To The Rescue, ECOOP'98, 1998

Use of Message-Passing Control

- [] CodA, Ecoop'95
- Actalk, OOPSLA'89
- Proxies
- OpenC++
- Jassist
- Fabre paper

Web pages

CLOS:

http://www.franz.com/

Open Implementation:

http://

Languages:

- NeoClasstalk: http://www.emn.fr/cs/neoclasstalk/
- VisualWorks: http://www.objectshare.com/VWNC/
- Smalltalk Archive: http://www-st.cs.uiuc.edu/
- Squeak: The Smalltalk Open Source http://www.squeak.org/
- OpenC++
- JavaAssist