Studying a Minimal Object-Oriented Kernel

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.iam.unibe.ch/~ducasse/

Food for thoughts

"L'idée de l'expérience ne remplace pas l'expérience"
Alain

"Give a man a fish; you have fed him for today. Teach a man to fish; and you have fed him for a lifetime"

Therefore do not listen and do not do the exercises....

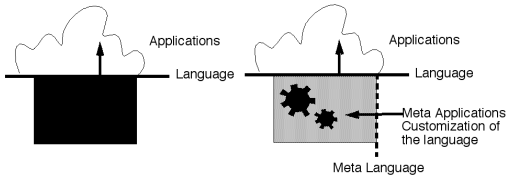
Goals

- Classes as objects
- Object and Class classes
- Semantics of inheritance
- Semantics of super and self
- Instantiation vs. Inheritance
- Allocation and Initialization
- Build your own language

Outline

- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping

Meta Programming Context



Classes 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***."
B. Meyer 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

At the Method Level

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 are one of the possible meta-entities (method, instance variables, method combination,...)
  allow the structural extension of the language
They may control
  Inheritance
  Internal representation of the objects (listes, vecteurs, hash-table, ...)
  Instance variable access
Separation of responsibilities
  Ordinary objects are used to model real world
  Metaobjects describe these ordinary objects
  Meta/Base level functionality is not mixed

Roadmap

- Classes as objects
- ***ObjVlisp in 5 postulates***
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping

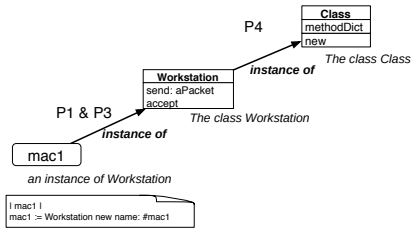
Why ObjVlisp?

- Minimal (only two classes)
- 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
- Simple: can be implemented with less than 300 lines of Scheme or 30 Smalltalk methods.
- Equivalent of Closette (Art of MOP example)

ObjVlisp Postulates (I)

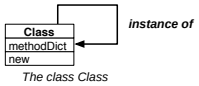
- P1: object = <data, behavior>
- P3: Every object belongs to a class that specifies its data (slots or instance variables) and its behavior. Objects are created dynamically from their class.
- P4: Following P3, a class is also an object therefore instance of another class its metaclass (that describes the behavior of a class).

ObjVLisp Postulates (II)



Infinite Recursion

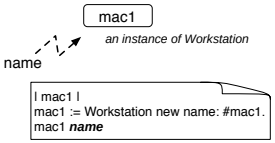
- A class is an object therefore instance of another class its metaclass that is an object too instance of a metametaclass that is an object too instance of another a metametametaclass.....
- To stop this potential infinite recursion
  - Class is the initial class and metaclass
  - Class is instance of itself and all other metaclasses are instances of Class



ObjVlisp 2nd Postulate

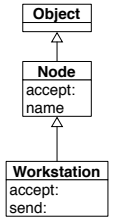
- P2: Message passing is the only means to activate an object

[object selector args]



ObjVlisp 5th Postulate

- P5: A class can be defined as a subclass of one or many other classes.



Unifying Class/Instance

- 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.

Metaclass

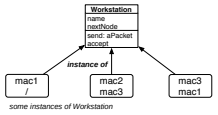
- Every object is instance of a class
- A class is an object instance of a metaclass (P4)
- But all the objects are not classes
- Only one kind of objects without distinction between classes and final instances.
- Sole difference is the ability to respond to the creation message: ***new***. Only a class knows how to deal with it.
- A ***metaclass*** is only a class that generates classes

RoadMap

- Classes as objects
- ObjVlisp in 5 postulates
- ***Instance Structure and Behavior***
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping

Instance Structure

- Instance variables
  - an ordered sequence of instance variables defined by a class
  - shared by all 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
  - 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***.

Methods

- Let's use a Smalltalk block
- name -> [:objself | objself unary: #name]
- no direct access to instance variables

RoadMap

- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- ***Class Structure***
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping

Class as an Object

- A class possesses the instance variable class inherited from Object that refers to its class (the metaclass that creates it).
  - Class value: an identifier of the class of the instance
- As an instance factory the metaclass Class possesses 4 instance variables that describe a class:
  - ***name*** the class name
  - ***superclass*** its superclass (we limit to single inheritance)
  - ***i-v*** the list of its instance variables
  - ***methodDict*** a method dictionary

Class Node as Object

The class Node

Class 'Node' Object 'name nextNode' methods...	*is instance of Class* *named Node* *inherits from Object* *has instance variables* *defines some methods*

Class Point as Object

The class Point

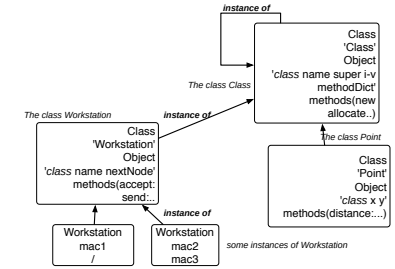
Class 'Point' Object 'x y' methods...	*is instance of Class* *named Point* *inherits from Object* *has instance variables* *defines some methods*

The class Class

The class Class

Class 'Class' Object 'name super i-v methodDict' methods...	*is instance of Class* *named Class* *inherits from Object* *has instance variables* *defines some methods*

Instances...



The class Class

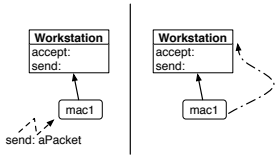
- Initial metaclass
- Defines the behavior of all the metaclasses
- Defines the behavior of all the classes

RoadMap

- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- ***Message Passing***
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Bootstrapping

Message Passing

- 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)

- Message send = 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)

RoadMap

- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- ***Object allocation & Initialization***
- Class creation
- Inheritance Semantics
- Bootstrapping

Object Creation

- 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: 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

Allocation Examples

[Point allocate]
-> #(Point nil nil) for x and y

[Workstation allocate]
->#(Workstation nil nil) for 'name' and 'nextNode'

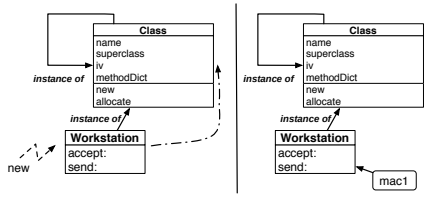
[Class allocate]
->#(Class nil nil nil....)

Object Initialization

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

Metaclass Role

Lookup method in the class of the receiver then we apply it to the receiver.

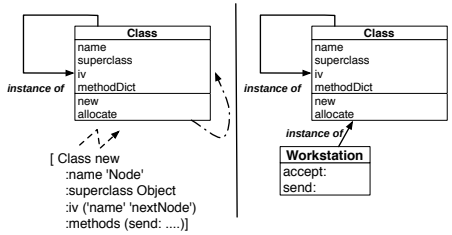


RoadMap

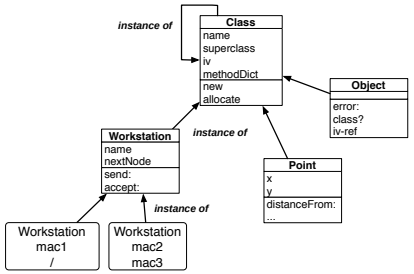
- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- ***Class creation***
- Inheritance Semantics
- Bootstrapping

Class Creation

Look in the *class of the receiver*



Instantiation Graph



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***

RoadMap

- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- ***Inheritance Semantics***
- Bootstrapping

Two kinds of inheritance

- Static for the state
  - subclasses get superclass state
  - At compilation time
- Dynamic for behavior
  - inheritance tree walked at run-time

Instance Variable 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))

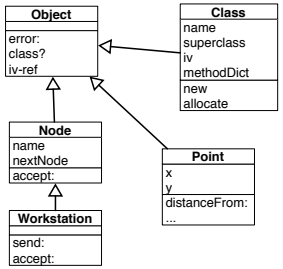
Method Inheritance

- Walks through the inheritance graph between classes using the super instance variable
  - lookup (selector class receiver):
    if the method 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.

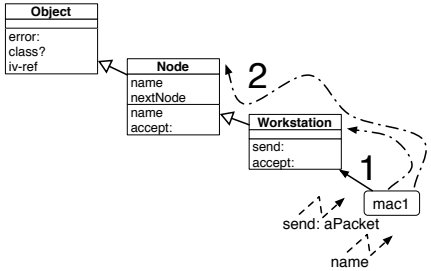
Inheritance Graph

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

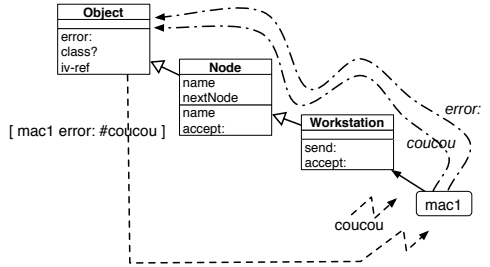
Inheritance Graph



Lookup (I)



Lookup (II)



Semantics of super

- As self, super is a pseudo-variable that refers to the ***receiver*** of the message.

- Used to invoke overriden methods.

Dynamic vs. Static

- self is dynamic:
  - Using self the lookup of the method begins in the class of the receiver.
  - Bound at execution-time
- super is static:
  - Using super the lookup of the method begins in the superclass of the class of the method containing the super expression (not in the superclass of the receiver class).
  - Bound at compile-time

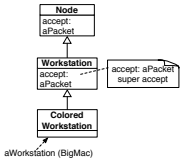
super is NOT the receiver class superclass

- Let us suppose the WRONG hypothesis: "The semantics of super is to start the lookup of a method in the superclass of the receiver class"
- agate accept: aPacket
- agate is an instance of DuplexWorkstation.
- accept: is looked up in the class DuplexWorkstation
- accept: is not defined in DuplexWorkstation, so the lookup continues in Workstation

Yes...Why?

accept: is defined in Workstation
lookup stops
method accept: is executed
Workstation>>accepts: does a super send
Our hypothesis: start in the super of the class of the receiver
=> superclass of class of a ColoredWorkstation is ...Workstation

Therefore we look in workstation again!!!

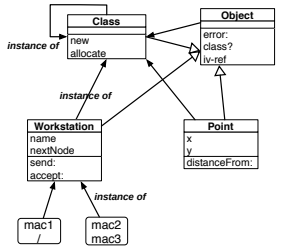


Minimal Shared Behavior

- The class Object represents the common behavior shared by all the objects:
  - classes
  - final instances.
- every object knows its class: instance variable class
- methods:
  - initialize (instance variable initialization)
  - error, class, metaclass?, class?

  Meta operations:
  - iv-set, iv-ref

A Simple Kernel



RoadMap

- Classes of objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- ***Some points***
- Bootstrapping

Class initialization

- 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***

Class initialization

[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: (...)]

About the 6th Postulate

- 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.

Why the 6th is wrong!

- Semantically class variables are not instance variables of object'class!
- Instance variable of metaclass should represent class information not instance information shared at the meta-level.
- Metaclass information should represent classes not domain objects
-

Solution

A class possesses an instance variable that stores structure that represents instance shared-variable and their values.

RoadMap

- Classes as objects
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Some points
- ***Recap***
- Bootstrapping

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
- Some 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

RoadMap

- Metaclasses?
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- Class Structure
- Message Passing
- Object allocation & Initialization
- Class creation
- Inheritance Semantics
- Some points
- Recap
- ***Bootstrapping***

Bootstrapping

- 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 Bootstrap

- Manual creation of the instance that represents the class ***Class*** with
  - inheritance simulation (class instance variable from ***Object*** class)
  - only the necessary methods for the creation of the classes (new and initialize)
- Creation of the class
  - ***Object*** [Class new :name 'Object'....]
  - definition of all the method of Object
- Redefinition of ***Class***
  - [***Class*** new :name 'Class' :super ***Object***.....] definition of all the methods of Class

Examples

- Metaclasses?
- ObjVlisp in 5 postulates
- Instance Structure and Behavior
- ...
- ...
- ...
- Examples

Abstract Classes

- The rule to define a new metaclass is to make it inherit from a previous one
- Prb. Abstract classes should not create instances
- Sol. Redefine the new method

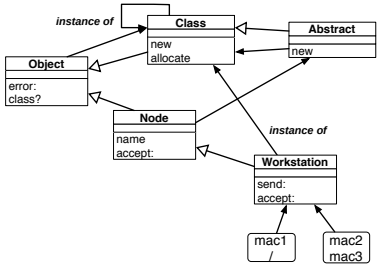
Metaclass Definition

- [Class new
  :name Abstract
  :super Class
  :methods
  (new (lambda (self initargs)
  (self error "Cannot create instance
  of class %s" self name))]
- Abstract is a class: It is instance of ***Class***
- Abstract define class behavior: It inherits from ***Class***

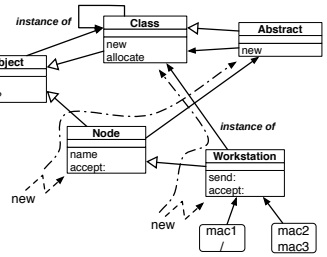
Metaclass Use

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

Complete Picture



Method Lookup



References

- [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.
- [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

Summary

Classes are objects too
Instantiation = initialize(allocate())
Class is the instantiation root
Object is the inheritance root
One single method lookup for classes and instances
  first go to the class
  then follow inheritance chain
super and self are referring to the message receiver but super changes the method lookup

License: CC-Attribution-ShareAlike 2.0