Programming with ObjVlisp Metaclas in Smalltalk-80

Jean-Pierre Briot and Pierre Cointe

Equipe Mixte Rank Xerox France - LITP,
Université Pierre et Marie Curie,
4 place Jussieu, 75005 Paris, France
briot/pc@rxf.ibp.fr.uucp

Published in Proc. of OOPSLA'89,
ACM Sigplan Notices, Vol. 24, No 10, October 1989, pages 41

Abstract

This paper discusses the introduction of explicit metaclasses à l the Smalltalk-80 language. The rigidity of Smalltalk metaclass motivated this work. Consequently we decided to implement the into the standard Smalltalk-80 system. The resulting combinati Classtalk platform. This platform provides a full-size environment with class-oriented programming by composing implicit met Smalltalk and explicit metaclasses à la ObjVlisp. Obviously, these e not limited to the Smalltalk world and will be useful to understa the metaclass concept advocated by modern object-oriented lan ObjVlisp and CLOS.

1 Introduction

Uniformity is one of the main advantages of Object-Oriented [Goldberg&Robson83]. Therefore in the sub-field of class-oriented increasing number of people claim that classes must be considere objects" [Cointe87], i.e. described by true and appropriate metaclasses.

1.1 Metaclasses are Useful

It has already been argued that metaclasses are useful both at the implementor levels to describe and extend the class architect

For the implementor, metaclasses are the means to describe a the object system itself, for instance to tailor the im [Cointe&Graube88], describe and extend the language in a [Bobrow&Kiczales88] [Attardi&al89], and control the execut [Malenfant&al89]. In short to describe and control the implement at the user's level.

For the user, metaclasses define the class methods, which a messages to classes, e.g. the messages to create new objects, an variables at the class level, which allow to parametrize classes [Coi

1.2 Metaclasses in Smalltalk

Historically, Smalltalk was the first language to introduce meta implementation level, they define the kernel of the architecture the metaclasses of the Kernel-Classes category) in an object-oriented the user level, metaclasses have been voluntarily hidden from t the user defines a new class, a new metaclass is automatically system. This implicit metaclass is anonymous, unsharable and st with its private instance, the class which originated it.

This separation between the implementor-level and user-level architecture which is not fully uniform, complex, and difficult to extend. This choice eases the ergonomy of the programmer, but i constructs he may achieve, as we will see. Consequently people field of learnability of object-oriented programming claim that metaclasses complicate unnecessarily the model and that th removed or at least highlighted [Borning&OShea87]. Neverthele decision to remove metaclasses can lead to remove classes too, prototype-based Smalltalk languages [Ungar&Smith87].

1.3 Metaclasses in ObjVlisp & CLOS

Taking the opposite approach, many people have been looking f explicit metaclasses. Such systems are Loops, ObjVlisp, CLOS and proposed the ObjVlisp model [Briot&Cointe87] which supports a sin minimal architecture for explicit metaclasses. The Common Lisp (CLOS) [Bobrow&Kiczales88] has also been designed along such an

Meanwhile ObjVlisp has the drawbacks of its minimality. It d enough class libraries to allow real-size experiments with end-use much richer language but there are currently few implement programming environment is still under work.

1.4 Motivations & Objectives

A previous study [Cointe88] convinced us that the Smalltalk extensible enough to support another metaclass system. Because currently the most complete and flexible object-oriented environment, we decided to introduce the uniform architectu metaclasses into it. This integration must be complete in order with (meta)class-oriented programming while still reusing standa class libraries. The resulting system, named Classtalk, provide metaclasses which the programmer may combine as bricks to d metaclass levels.

1.5 Outline of the Paper

Section-2 discusses the limitations of the Smalltalk-80 architectu private class/metaclass module and the non uniform protocol objects. Section-3 shortly reviews how the ObjVlisp and CLOS arc these gaps. Section-4 discusses two options in order to integrat architecture into Smalltalk-80, and then details one implement describes how we extend the standard Smalltalk-80 programmi to provide a specific one suitable to Classtalk explicit metacla introduces a first library of metaclasses. Section-7 explains how Borning&Ingalls' multiple inheritance scheme into Classtalk. Secti example of metaclass combinations. Section-9 discusses the reoc class/metaclass module and how to definitely solve this c implementing uniform creation. Section-10 discusses the new is this work before concluding.

2 The Smalltalk-80 Metaclass Arcanes

2.1 Kernel (Meta)Classes

Like ObjVlisp or CLOS, Smalltalk-80 uses a kernel of primitive metaclasses in order to self-describe classes. Smalltalk-80 calls classes. Class describes standard classes (classes which are not m Metaclass describes metaclasses. To express the common properti classes and metaclasses, they are both direct-subclasses of ClassDe subclass of Behavior. The inheritance hierarchy of the kernel cl below. The instance variables, which are here defined at the c enclosed within ().

```
Object ()
  Behavior (superclass methodDict format subclasses)
    ClassDescription (instanceVariables organization)
      Metaclass (thisClass)
        Class (name classPool sharedPools)
```

We can check the structural difference between a class and a metaclass uses the backward pointer thisClass to memorize its pr while a class has name, classPool and sharedPools variables.

2.2 User Metaclasses

Besides this primitive kernel architecture, the Smalltalk designer the metaclass architecture from the user and to provide an automatic metalevel for standard classes.

When a new class is defined, e.g. class Actor, the system automat class/metaclass module. This means that the system first creates metaclass and then instantiates it in order to create the class w sole instance. Such an implicit metaclass is anonymous and is on sending the message class to the class it describes, e.g. Actor class connects the definitions of the class and its metaclass through th switch view of the browser.

The user may define methods at the metaclass level. These me messages which may be sent to the class itself, and are named cl order to extend the structure of standard classes, the user m instance variables at the metaclass level. Nevertheless these var specific names and are not part of the Smalltalk terminology. Th mistaken for class or pool variables which implement shared var

2.3 Rigidity of the Metaclass Architecture

Being implicitly created by the system, the inheritance and i metaclasses should obey to some implicit rules. To provide the s rule for class and instance methods, the inheritance hierarchy o parallel to the inheritance hierarchy of classes. In order to l structure and behavior for all implicit metaclasses, each of them instance of the kernel class Metaclass. Smalltalk-80 connects t inheritance hierarchy to the class hierarchy by declaring the metaclass, Object class, as a subclass of the kernel class Class.

```
Object ()
   Actor ()
  Behavior (superclass methodDict format subclasses)
     ClassDescription (instanceVariables organization)
        Metaclass (thisClass)
        Class (name classPool sharedPools)
           Object class ()
              Actor class ()
             Behavior class ()
                ClassDescription class ()
                   Metaclass class ()
                   Class class ()
```

We experienced that the implicit class/metaclass module provid rigid coupling between a class and its metaclass. This leads to lir expressiveness of the langage as illustrated by the next example.

2.4 The abstract Class Counter-Example

"Abstract class: a class that specifies protocol, but is not able to it; by convention, instances are not created of this kind [Goldberg&Robson83]

A simple example of abstract class appears when one tries to mo numbers as objects. Two models for representation are useful numbers, namely cartesian and polar coordinates. Therefore v standard classes (which are not abstract classes), respectively Carte implement them. The abstract class Complex factorizes the commo instance computing arithmetics. In the inheritance hierarchy f methods are enclosed within <>.

```
Complex () <+ - *  / conjugate modulus negated>
   Cartesian (x y) <x y rho theta x:y: printOn:>
  Polar (rho theta) <x y rho theta rho:theta: printOn:>
```

The problem is to modelize the general behavior of an abstract precisely to ensure the fact that such a class cannot create instan way is to forbid instantiation by redefining the standard cla creation (in fact allocation) in order to raise an error. This star named new and belongs to kernel class Behavior. It should be rede method, because it (re)defines the behavior of classes. Because

(and metaclasses) are implicit, we must define a standard class, na
own this method.

```
!Abstract class methodsFor: '(forbidden) allocation'!
new
  self error: 'no instance, I am an abstract class'! !
```

Then Complex is defined as a subclass of Abstract:

```
Object <...>
 Abstract <>
  Complex <...>
   Cartesian <...>
   Polar <...>
  Behavior<...  new ...>
   ClassDescription <...>
     Metaclass <...>
     Class <...>
       Object class <...>
         Abstract class <new>
          Complex class <x:y:  rho:theta:>
            Cartesian class <x:y:>
            Polar class <rho:theta:>
```

Because Complex is defined as a subclass of Abstract, its metaclass inherits the redefinition of method new owned by Abstract class. classes Cartesian and Polar both inherit from Complex. Conse corresponding metaclasses also inherit the forbidden instantiat become abstract classes too and it will be impossible to create number. Unfortunately the rule for implicit inheritance of meta match our intuition.

A pragmatic solution is to explicitly change the inheritance ru the instance variable superclass, which specifies the inheritance lin declare the most general metaclass, i.e. Object class, as the new supe

```
Cartesian class superclass: Object class.
Polar class superclass: Object class
```

This solution works but is ad hoc and not modular (we nee inheritance for every subclass). The complete solution, given in an explicit control on inheritance <u>and</u> instantiation of classes.

2.5 Non Uniform Creation

Smalltalk provides two primitive methods to allocate objects. T named new and new: are owned by the kernel class Behavior. Metho objects whose structure is defined by named instance variables (su whereas new: allocates objects whose structure is defined by in (such as Array). Every object in the system, except rockbottom c numbers, is created by calling one of these allocators. Consequen objects is (almost) uniform. However their initialization is not.

When an object is allocated, the values associated to its instanc the default initial value nil. In order to initialize these variable method is provided, and therefore one needs to define explicitly method. For instance we define such a method which initializes Cart

```
!Cartesian methodsFor: 'initializing'!
setX: xValue setY: yValue
  x _ xValue.
  y _ yValue! !
```

If we want to combine allocation and initialization into a singl creation, we have to define the following class method:

```
!Cartesian class methodsFor: 'creation'!
x: xValue y: yValue
  ^self new setX: xValue setY: yValue! !
```

Such initialization and creation methods are mostly specific because their selectors follow the names of the instance variables. creating classes, however, because all classes share the same instance variables defined or inherited by Class), there is a st method named subclass:instanceVariablesNames:...category: and owr provides the appropriate values to create (and initialize) standard

In summary there is no uniform way of creating objects in Sma allocation of objects is uniform. This is not too much burder Smalltalk-80 because all classes may be created through the method. But when we will start to parametrize classes by definin variables on metaclasses, we will need to define specific initi creation methods to deal with these new instance variables. This be touched upon after defining some explicit metaclasses of the C

3 The ObjVlisp & CLOS Alternative

The complete solution to the previous limitations has already b [Cointe87]. Classes must be explicitly and uniformly created as in other classes called metaclasses.

ObjVlisp and CLOS are two systems which propose such an ObjVlisp is also minimal in the sense of being self-defined by on which are the root of the instantiation tree (Class) and the root o tree (Object). Class, being an object, must be itself described by instance of) some class. The minimal solution proposed in [Br defines Class as instance of itself. This self-instantiation ensure uniformity and self-description (reflexivity) of the kernel.

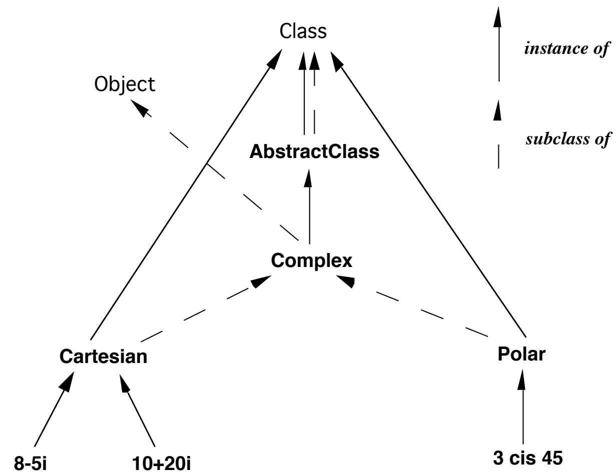
3.1 Explicit Metaclasses

An ObjVlisp metaclass is a class which can have access to allocation message by owning it or by inheriting it. Class, as the standard allocation method allocateInstance, is the first metaclass order to inherit this standard allocator, a new metaclass is alwa

subclass of a previous one. As opposed to Smalltalk-80, there is
between classes and metaclasses. Consequently the two metaclasse
80 (Class and Metaclass) are merged into one only (Class).

3.2 Abstract Class Revisited

In ObjVlisp, as opposed to Smalltalk-80, there is no implicit li
class and its private metaclass. Consequently a same metaclass
(shared) to describe different classes. The ObjVlisp solution to th
problem is summarized by the following architecture:



There are three steps to this solution. Below are the correspon
in Classtalk:

• create the new metaclass describing all abstract classes. Abst
instance and a subclass of the first metaclass Class. AbstractClass
allocation methods new (and new:) in order to raise an error,

```
Class newName: #AbstractClass
  superclass: Class
  instanceVariableNames: ''
  category: 'Metaclass-Library'!

!AbstractClass methodsFor: '(forbidden) allocation'!
new
  self error: 'no instance, I am an abstract class'!

new: n
  self error: 'no instance, I am an abstract class'! !
```

• create a new abstract class Complex, instance of AbstractClass a
Object,

```
AbstractClass newName: #Complex
  superclass: Object
  instanceVariableNames: ''
  category: 'Numeric-Complex
```

• create the two standard classes Cartesian and Polar as instanc subclasses of Complex,

```
Class newName: #Cartesian
  superclass: Complex
  instanceVariableNames: 'x y '
  category: 'Numeric-Complex'!

Class newName: #Polar
  superclass: Complex
  instanceVariableNames: 'rho theta '
  category: 'Numeric-Complex'!
```

3.3 Uniform Creation

In ObjVlisp and CLOS, creation of objects is uniform. This is combination of an allocation and an initialization method:

creation = initialization o allocation

Class holds the standard allocation method, named allocateIns standard creation method, named makeInstance:. There are initialization methods, both of them named initializeInstance:. The fi by Object and defines standard initialization of objects. The seconc Class, defines initialization of classes. Initializing classes is more includes for instance compiling static inheritance of instan Consequently this second initialization method extends (and c general initialization method owned by Object. Following is tl hierarchy of the ObjVlisp kernel:

```
Object <initializeInstance:>
  Class <allocateInstance initializeInstance: makeInstance:>
```

Compared with Smalltalk, the ObjVlisp makeInstance: method inclu for object initialization, which it transmits to the initializeInstance: the Smalltalk-80 method new is a simple allocator (equivalent to allo not a complete creation method.

4. Classtalk: ObjVlisp in Smalltalk-80

Implementing ObjVlisp in Smalltalk-80 raises two problems:

• introducing an explicit class architecture not limited to an au between a class and its metaclass,

• introducing a unified method of creation which takes into a allocation and the intitialization procedures.

Smalltalk-80 is extensible enough to propose a simple and clean first problem. But its somewhat limited syntax makes it difficult satisfactory solution to the second problem (this will be discusse

The resulting system, named Classtalk, provides explicit classes a in Smalltalk-80 and a solution to uniformity of creation. Classtal platform to experiment with (meta)class-oriented programming.

4.1 Creating Classes Explicitly

In order to create a class as an explicit instance of a metaclass new creation message, with selector newName:superclass:instanceVariableN parameters come from ObjVlisp, but its keywords retain the Sma and conventions. Thus we chose as its selector. Note that, as ObjVlisp, class and pool variables are suppressed for the sake of new creation message is sent to the metaclass, i.e. the creator, superclass as in standard Smalltalk-80. This follows the principle object as an instance of a class.

4.2 Implementation Alternative

The question which remains opened is: "Which metaclass should method for creating explicit classes?". Because in ObjVlisp the method is the root of the kernel, i.e. metaclass Class, this questi into: "How do we transpose the ObjVlisp kernel into the architecture?". At the implementation level, two answers may be

- identifying (merging) the ObjVlisp kernel, classes Class and O two corresponding Smalltalk-80 classes,
- grafting ObjVlisp by adding to the Smalltalk-80 kernel a n named Classtalk, defined as a subclass of ClassDescription.

4.2.1 Merging

Class already owns the standard method subclass:instanceVariableNa creating standard Smalltalk classes. By identifying the ObjVlisp with the Smalltalk-80 kernel class Class, the method newName:...cat also a method of Class.

Class <subclass:...category: ... newName:...category:>
 Object class <...>
  Behavior class <...>
   ClassDescription class <>
    Class class <...>

Class is both the instance and an undirect superclass of its meta This provides an implicit self-description of Class, although, a ObjVlisp, Class class is <u>not equ</u>al to Class. This solution is minimal.

4.2.2 Grafting

The grafting scheme gives more control on the design of the C but complicates its self-description. Classtalk class is a subclass of Classtalk. Nevertheless we can change the implicit rule of Small inheritance to make Classtalk class a direct subclass of Classtalk:

Classtalk class superclass: Classtalk

This splits inheritances of structure and behavior into two diffe

```
Object ()
  Behavior (superclass methodDict format subclasses)
    ClassDescription (instanceVariables organization)
      Metaclass (thisClass)
      Class (name classPool sharedPools)
        ObjectClass ()
          Behavior class ()
            ClassDescription class ()
              Classtalk  class ()
      Classtalk (name  category)

Object <...>
  Behavior <... new ...>
    ClassDescription <...>
      Metaclass <>
     Class <... subclass:...category: ...>
     Classtalk <newName:...category:>
       Classtalk  class <>
```

This solution also allows a more precise definition of Classtalk c instance variables such as classPool and sharedPools are no longer c may define new ones, e.g. category which will be useful in section 9 the instance variable name and some methods of Class need to b Classtalk.

In the following of the paper, although both solutions are mostly choose the first one, i.e. class Classtalk, in order to emphasize the standard Smalltalk-80.

4.3 Explicit Creation of Classes

The implementation of the method newName:...category: to create follows the standard implementation of class creation. It inclu along the type of the superclass (with named or indexed vari standard Smalltalk-80, the "auxiliary method" newName:environment:. common implementation between classes with named or ind variables.

To focus on the semantic of these two methods, we give the without the type dispatcher and without the pieces of code management of the programming environment (syntax ch management...) which are replaced by comments.

```
!Classtalk methodsFor: 'Classtalk - class creation'!
newName: n superclass: s instanceVariableNames: i category: c
   "Dispatch along classes with indexed variables."
   ^self
       newName: n
       environment: Smalltalk
       superclass: s
       otherSupers: nil
       instanceVariableNames: i
       variable: false
       words: true
       pointers: true
       category: c!

newName: n environment: e superclass: s otherSupers: o instanceVariableNam
   variable: v words: w pointers: p category: c
   | newClass "..." |
   "Syntax checking and redefinition management."
   "(1) Allocation of the new class."
   newClass _ self new.
   "(2) Initialization of the new class - 1."
   newClass
      superclass: s
      methodDict: MethodDictionary new
      format: -8192
      name: n
      organization: ClassOrganizer new
      instVarNames: (Scanner new scanFieldNames: i)
      classPool: nil
      sharedPools: nil.
   "(3) Specification of remaining superclasses."
   os isNil ifFalse: [newClass otherSupers: os].
   "(4) Initialization of the new class - 2."
   newClass
      format: newClass allInstVarNames size
      variable: v
      words: w
      pointers: p.
   "Environment management."
   ObjVlispOrganization classify: newClass name
                               under: categoryString asSymbol.
   "Hierarchy updating and change management."
   "(5) Compilation of multiple inheritance."
   o isNil  ifFalse: [newClass copyMethods].
   ^newClass! !
```

• as suggested by ObjVlisp a class creation is realized in two st (1) and initialization (2 & 4). The new class created (temporary va is defined explicitly as an instance of a previous metaclass: self standard Smalltalk-80, the initialization process takes place in steps: (2) and (4).

• to organize Classtalk classes in a specialized browser we int organizer, the global variable ObjVlispOrganization which is cou Classtalk browser.

• the method newName:environment:..category: introduces a param
the keyword otherSupers:. It specifies an unused array of superclass
is nil). Meanwhile, this allows this method to be reused when intro
inheritance (see section-7).

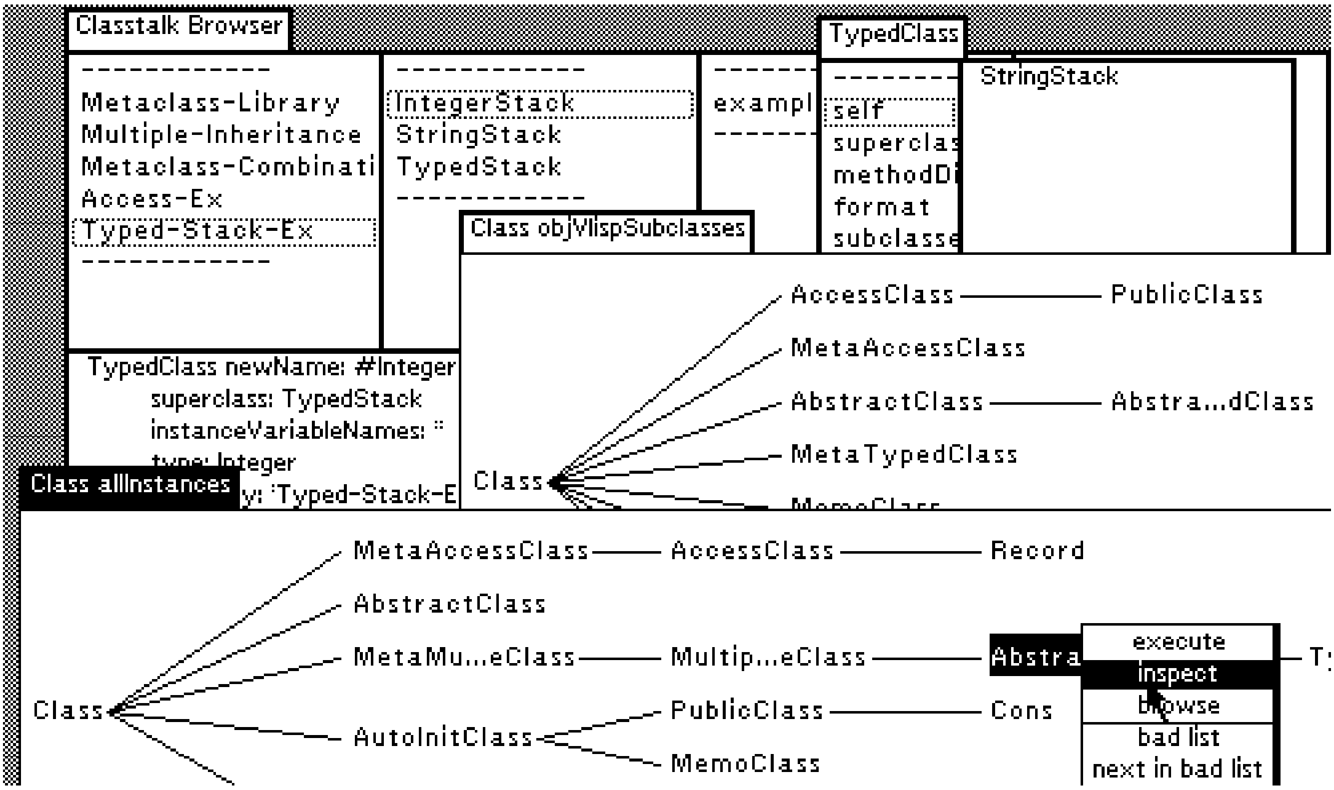
• expressions (3) and (5) are evaluated in the case of multiple
(3) assigns the array of remaining superclasses. (5) calls the
multiple inheritance provided by the standard extension of
[IngallsBorning82]. This will recompile the methods or genera
methods when needed.

5 The Classtalk Environment

The Smalltalk-80 standard browser may confuse the progr
browsing on Classtalk classes. When the instance/class switch is s
browser shows the explicit metaclass, and not an implicit one
Smalltalk-80. Moreover the template and the definition printed
do not reflect the Classtalk definition. If evaluated, the definitio
already created won't lead to the same result. It will redefine t
standard Smalltalk-80 way (with an implicit metaclass) while fo
the previous explicit metaclass.

Therefore we designed a specific browser dedicated to Classtal
differences lie in the removal of the instance/class switch and th
templates and definitions in order to make clear the Classtalk v
classes.

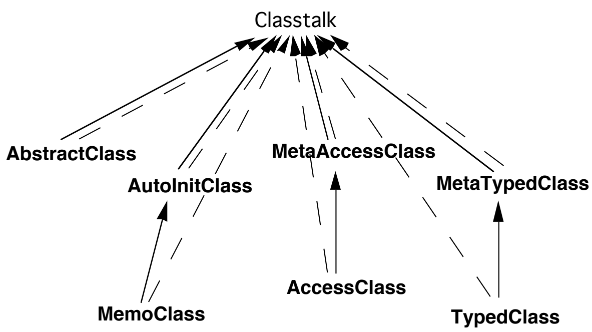
This browser is also interfaced with a generic tree editor [Woli
to browse onto both the instantiation and the inheritance graphs



6 Library of Metaclasses

This new browser was helpful to develop a library of primi metaclasses. Our idea was to reuse them as bricks to define metaclasses by combining with both the instantiation anc mechanisms.

In this section we propose to introduce and comment some of recall that the creation rule for these metaclasses is the followir direct-subclass of Classtalk or a direct-subclass of another explicit naming convention is that each metaclass' name ends up with Class



6.1 AbstractClass

This metaclass modelizes abstract classes, i.e. non-instantiabl defined and used in section-2.4.

6.2 AutoInitClass

This metaclass modelizes classes which provide their instances
initialization.

In order to get an automatic initialization of objects, every
programmer has at least once redefined the class method new. T
common and to avoid code duplication, we modelize this behavio
metaclass AutoInitClass. A class instance of AutoInitClass has the follov
sends the message init to an instance being created.

```
Classtalk newName: #AutoInitClass
  superclass: Classtalk
  instanceVariableNames: ''
  category: 'Metaclass-Library'!

!AutoInitClass methodsFor: 'allocation'!
new
  ^super new init! !
```

6.3 MemoClass

This metaclass modelizes classes which memorize the collectio
instances by using an explicit backpointer.

This backpointer is implemented by a new instance variable ins
the metaclass level. Its value is an ordered collection memo
instances which are created.

This variable needs to be initialized to an empty collection be
create instances. In order to provide an automatic initializati
MemoClass as an instance of AutoInitClass.

```
AutoInitClass newName: #MemoClass
  superclass: Classtalk
  instanceVariableNames: 'instances '
  category: 'Metaclass-Library'!

!MemoClass methodsFor: 'init'!
init
  instances _ OrderedCollection new! !

!MemoClass methodsFor: 'allocation'!
new
  "Method add: returns the object added."
  ^instances add: super new! !

!MemoClass methodsFor: 'accessing'!
instances
  ^instances! !
```

6.4 TypedClass

This metaclass modelizes classes which are parametrized by a ty

TypedClass introduces the new instance variable type and two ass
methods. In order to provide an explicit initialization of this vari
extend and specialize the standard Classtalk message for creati
new creation method, named newName:...type:category: combine

newName:...category: creation method with the assignment of the t
the definition of this new method led us to introduce the
MetaTypedClass whose only goal is to hold this extended creation
uniform initialization constrains us to reintroduce the class/meta

```
Classtalk newName: #MetaTypedClass
  superclass: Classtalk
  instanceVariableNames: ''
  category: 'Metaclass-Library'!

MetaTypedClass newName: #TypedClass
  superclass: Classtalk
  instanceVariableNames: 'type '
  category: 'Metaclass-Library'!

!TypedClass methodsFor: 'accessing'!
type
  ^type!

type: aClass
  type _ aClass! !

!MetaTypedClass methodsFor: 'creation'!
newName: n superclass: s instanceVariableNames: i type: aClass category: c
  ^(self newName: n superclass: s instanceVariableNames: i category: c) type: aClass
```

6.5 AccessClass

This metaclass modelizes classes which may provide automati
accessors to their instance variables.

Another repetitive programming problem lies in the definitio
methods. Their selectors are usually associated to the instance va
they give access. In order to relieve the programmer from th
propose the metaclass AccessClass which describes how to generat
such accessors. The programmer can specify which instance va
public (i.e. with accessors) by using the declaration public:.

The following example is the Classtalk solution to the example
[Goldberg&Robson83], pages 289-290:

```
AccessClass newName: #Record
  superclass: Object
  instanceVariableNames: 'name address '
   public: 'name '
  category: 'Access-Example'!
```

Like TypedClass, the specialization of the creation message leads
new metaclass, named MetaAccessClass, to own the extended creatio
This method, named newName:...public:category:, will compose
newName:...category: method with the call of the method to generat
method, named makeIvAccessOn:, is owned by AccessClass. A scanner
specifying public variables into an array which becomes the pa
message:

```
Classtalk newName: #MetaAccessClass
   superclass: Classtalk
   instanceVariableNames: ''
   category: 'Metaclass-Library'!

!MetaAccessClass methodsFor: 'creation'!
newName: n superclass: s instanceVariableNames: i public: p category: c
   ^(self newName: n superclass: s instanceVariableNames: i category: c)
        makeIvAccessOn: (Scanner new scanFieldNames: p)! !

MetaAccessClass newName: #AccessClass
   superclass: Classtalk
   instanceVariableNames: ''
   category: 'Metaclass-Library'!

!AccessClass methodsFor: 'access generation'!
makeIvAccessOn: ivNameArray
   ivNameArray isNil ifFalse:
      [ivNameArray do: [:ivString |
          self compile: ivString , '^' withCRs , ivString
               classified: #accessing;
               compile: ivString , ': aValue' withCRs, ivString , ' _ aValue'
               classified: #accessing]]! !
```

7 MultipleInheritance

We described samples of the library of metaclasses. The progra them and start to combine them by using instantiation and inh trivial cases, simple inheritance may be not enough, for instan classes which memorize and initialize their instances, by combi from both) MemoClass and AutoInitClass. Therefore we need multiple

We introduce such an extension in Classtalk, while reusing most Smalltalk-80 extension for multiple inheritance. We will at first the Smalltalk-80 extension and then describe how we interface it

7.1 Multiple Inheritance In Smalltalk-80

The strategy proposed in [Ingalls&Borning82] is to keep the sin scheme working. In case of multiple inheritance the first supercla be the standard superclass while the other ones are stored in the class. These remaining superclasses are referenced by the new i otherSupers, which is introduced by the kernel class MetaclassForMultiple

When creating a class with multiple superclasses, the methods w reached by the standard single inheritance lookup are recom method dictionary of the new class. If several methods with a sa be reached, conflicting inherited methods are automatically gen the conflicts their bodies need to be explicited by the programme

7.2 Multiple Inheritance In Classtalk

When modelizing multiple inheritance in Classtalk we define variable otherSupers directly at the class level (and no more at th Consequently we introduce the metaclass MIClass to define this

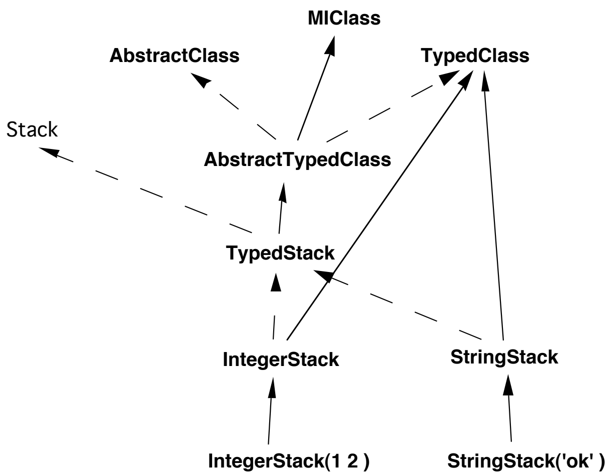
variable. As with metaclasses TypedClass and AccessClass, to exten
method we have to introduce a metaclass, named MetaMIClass.

The method to create classes with multiple superclasse newName:superclasses:instanceVariableNames:category:. Its syntax and in analog to those of the standard Classtalk newName:superclass:instanceVariableNames:category: and we do not sho paper.

8. Example of Metaclass Combination

To emphasize the Classtalk methodology we now develop the stacks example. Our goal is to define stacks whose parameter of th is typechecked. To make the demonstration easier, and to show reuse standard libraries, we suppose that a class Stack has be defined, e.g. as a subclass of primitive class Array extended with an Stack can be either a Classtalk class either a Smalltalk-80 class.

The class architecture we want to discuss is summarized by the and steps:



• to express the different types of stacks (IntegerStack, StringStack. stack is defined as a parametrized class (i.e. an instance of TypedCla

• to express the common behavior (and structure) of type introduce the abstract class TypedStack,

• to keep consistency between TypedStack and its subclasses StringStack...), TypedStack must be also parametrized,

• TypedStack having to be both abstract and parametrized, we metaclass AbstractTypedClass, subclass of both AbstractClass and T therefore instance of MIClass. Conflicting methods, namely new (an be redirected onto AbstractClass.

The definition of these classes is following:

```
MIClass newName: #AbstractTypedClass
   superclasses: 'AbstractClass TypedClass '
   instanceVariableNames: ''
   category: 'Metaclass-Combination'!

!AbstractTypedClass methodsFor: 'conflicting methods'!
new
   ^self AbstractClass.new! !

AbstractTypedClass newName: #TypedStack
   superclass: Stack
   instanceVariableNames: ''
   category: 'Stack-Collection'!

!TypedStack methodsFor: 'operations'!
push: x
   (x isKindOf: self class type)
      ifTrue:  [super push: x]
      ifFalse: [self error: 'wrong type']! !

TypedClass newName: #IntegerStack
   superclass: TypedStack
   instanceVariableNames: ''
   type: Integer
   category: 'Stack-Collection'!
```

9 Class/Metaclass Module vs Uniform Creation

9.1 Classtalk Library Revisited

The Smalltalk-80 class/metaclass module is split by Classtalk int components. On the one hand, this allows an unlimited level of provides the user with more freedom and possibilities as demo metaclasses library. But on the other hand, we need to define (r to own extended class creation methods each time we add som variable, e.g. metaclasses MetaTypedClass and MetaAccessClass.

The class/metaclass module remains necessary when defin creation messages, as in standard Smalltalk-80. But Smalltalk-80 w of implicitly creating a metaclass to support the class metho Classtalk the programmer gets the burden to explicitly define it.

Consequently in order to avoid defining metaclasses in such ca make creation to become uniform.

9.2 Initialization Synchronization

Another limitation of the standard Smalltalk-80 non-uniform c will be illustrated by the following example. Suppose that we wa classes whose all instance variables are public. Therefore we define an instance of AutoInitClass and a subclass of AccessClass, and who generates accessors on instance variables defined by the class:

```
AutoInitClass newName: #PublicClass
  superclass: AccessClass
  instanceVariableNames: ''
  category: 'Metaclass-Library'!

!PublicClass methodsFor: 'init'!
init
  self makeIvAccessOn: instanceVariables! !
```

Unfortunately, this intuitive formulation won't stand when c
classes. The method init is called during the process of allocatio
redefined in AutoInitClass) and before creation and initializatio
(method newName:...category:). Consequently instanceVariable will sti
(value nil) and none accessing method won't be generated.

We may redefine method newName:...category: to call the init meth
call the init method twice (once at allocation time ar
creation/initialization time). The modular solution would be to
the standard and uniform initialization method if it would exist.

9.3 Uniform Creation

ObjVlisp provides uniform initialization, and consequently creati
method initializeInstance: accepts a variable number of argument
initial values of the instance variables. Unfortunately Smalltalk-8
not allow selectors with variable arity. Therefore we need t
arguments into a single one, some data structure, for instance
creation of an instance of a cartesian complex would look like:

```
Cartesian create: #(y 2 x 1)
```

This reproduces the strategy of CommonLisp-like keywords w
reordered at wish, as opposed to explicit and ordered keywords i

9.4 Architecture

The uniform creation method is simply defined as the comp
standard allocation method (basicNew) and the uniform initiali
named initialize:. This method is defined in kernel class Behavior
useable by all classes, standard Smalltalk-80 or Classtalk ones:

```
!Behavior methodsFor: 'creation'!
create: initArray
  ^self basicNew initialize: initArray! !
```

Initialization of objects is made generic by defining two meth
initializing classes and owned by class Classtalk, and the other
terminal objects, and owned by class Object. The initialization c
specialization of initialization of general objects (use of pseudo-vari

```
!Classtalk methodsFor: 'initialization'!
initialize: initArray
   super initialize initArray.
  self environment: Smalltalk
        variable: false
        words: true
        pointers: true
        category: category! !
```

The method environment:...category: is defined as equivaler newName:environment:...category:, and manages initialization of the cl that category is defined as instance variable of class Classtalk, in ord value through the initialization process.

9.5 Implementation

There are several alternatives to implement the general initial owned by Object. The main problem is to evaluate the argument instance variables.

One solution is to extend Smalltalk-80 syntax in order to su creation of arrays, by using some macro-method or macro-chara Lisp's backquote (`).

We propose here another solution by evaluating the arguments calls to the compiler. For each instance variable, the standard meth realizes the binding, by using method indexOf:ifAbsent: to find t variable, and explicitly calling the compiler to compute the initial

```
!Object methodsFor: 'initialize-release'!

initialize: initArray
  | i max ivNames context |
  initArray isNil ifFalse:
      [i _ 1.
      max _ initArray size.
      ivNames _ self class allInstVarNames.
      context _ thisContext sender sender.
      [i < max] whileTrue:
         [self instVarAt:  (ivNames indexOf: (initArray at: i) ifAbsent:
             [self error: 'unknown instance variable: ' , (initArray at: i) printString])
                put: (Compiler new
                         evaluate: (initArray at: i+1) printString
                         in: context
                         to: context receiver
                         notifying: self
                        ifFail: [self error: 'compilation of initialize failed']).
         i _ i+2]]! !
```

9.6 Examples

We will now redefine the metaclass TypedClass and its instance show this simplification. Note that defining MetaTypedClass is necessary, because no specific creation method needs to be define

```
Classtalk create: #(
  name        #TypedClass
  superclass  Classtalk
  instanceVariables 'type'
  category    'Metaclass-Library')!

TypedClass create: #(
  name        #IntegerStack
  superclass  TypedStack
  instanceVariables ''
  type        Integer
  category    'Stack-Collection)!
```

The good definition of PublicClass is by redefining the initialize: m initialization is now uniform, the metaclass AutoInitClass is no mor may be removed from the library.

```
Classtalk create: #(
  name        #PublicClass
  superclass  AccessClass
  instanceVariables ''
  category    'Metaclass-Library')!

!PublicClass methodsFor: 'init'!
initialize:  initArray
  super initialize: initArray.
  self makeIvAccessOn: instanceVariables! !
```

10 Future Work & Conclusion

Before concluding, we now review the current limitations or wea Classtalk platform and methodology.

Methodology: when designing non trivial constructions, the needs to define new specific metaclasses which will model specific contribute to ultimately create the bottom-end objects that he wi Because of the class abstraction metaphor, the programmer ne classes before creating its concrete instances. When using metaclasses on multiple levels, this leads the programmer to go u to define the root of his application architecture before instantiating down to bottom-end instances. This unfortunately little bit the interactive and incremental programming philosoph 80.

Some practical consequence is also about example methods. Smalltalk-80 is to define them as class methods. In Classtalk a me shared by several classes, consequently if defining specific (examples, the programmer needs to define some specific metaclas

In order to alleviate these problems, we are currently stud interactive programming environment could help the program design of multi-levels constructions.

Uniformity of Creation: We implemented a prototype of uni and validated it by redefining all Classtalk library metaclasses

However we find the explicit calls to the compiler too much heav will now evaluate some minimal syntactic extension to support cr with evaluation of their elements.

Class/Metaclass Compatibility: Defining explicit metaclasses issue of compatibility between a class and its metaclass, i.e. hypotheses about the methods they define [Graube89]. This ma transparent problems when reusing standard Smalltalk-80 classes suppose that we define the class Stack as a subclass of class Ordered(than class Array. OrderedCollection defines some private initializatior setIndices, to properly initialize the indices. The allocation method of class is redefined in order to automatically ensure the initializatior

```
!OrderedCollection class methodsFor: 'instance creation'!
new: anInteger
  ^(super new: anInteger) setIndices! !
```

If the metaclass of typed stacks, i.e. metaclass AbstractTypedClass (does not provide such redefinition, stacks won't be properly cannot be used.

Smalltalk-80 ensures automatic compatibility between a cl metaclass, thanks to the rule for parallel inheritance hierarchies. Smalltalk-80 implicit class/metaclass module, we leave this respon programmer. In order to help him when defining classes, we w Classtalk programming environment in order to automatically incompatibilities by browsing through the hierarchies.

(No) Method Combination: The example of typed stacks ma extended and complexified by adding the memorization ability t By inheriting both from AutoInitClass and MemoClass, we enc combination problem. Choosing the right method new to solve th enough, we need a real combination of the two inherite Unfortunately such ability for method combination is not av standard Smalltalk-80 extension for multiple inheritance that w Therefore we plan to design a much richer extension for multipl provides real combination of methods in the spirit of CLOS.

Further Work: Besides solving limitations of the current system, use the Classtalk platform for intensive experiment with the prog metaclasses methodology, and to apply it to simulate part-wh parametrization..., by extending the current library of metacla elements and new combinations.

Conclusion: Although a starting project, this realization alread once more that Smalltalk-80 is an extensible system, and that architecture is general enough to be applied to Smalltalk.

Acknowledgements: We thank Nicolas Graube, François Pach François Perrot for discussions about the project, and Francis providing his generic tree editor which is used in the Classtalk en

11 Bibliography

[Attardi&al89] G. Attardi, C. Bonini, M. Boscotrecase, T. Flagella ar Metalevel Programming in CLOS, ECOOP'89, July 1989.

[Bobrow&Kiczales88] D.G. Bobrow and G. Kiczales, The Common Lisp Metaobject Kernel - A Status Report, ACM Conference on Lisp a Programming (LFP'88), pages 309-315, July 1988.

[Borning&OShea87] A. Borning and T. O'Shea, Deltatalk: An Em Aesthetically Motivated Simplification of the Smalltalk-80 Langua LNCS, No 276, pages 1-10, Springer-Verlag, June 1987.

[Briot&Cointe87] J.-P. Briot and P. Cointe, A Uniform Model for C Languages Using The Class Abstraction, IJCAI'87, Vol. 1, pages 1987.

[Cointe87] P. Cointe, Metaclasses are First Class: the ObjVlisp Mode pages 156-167.

[Cointe&Graube88] P. Cointe and N. Graube, Programming with CLOS, First CLOS Users and Implementors Workshop, Xerox Parc, USA, October 1988.

[Cointe88] P. Cointe, A Tutorial Introduction to Metaclass Architect by Class Oriented Languages, International Conference on Fi Computer Systems (FGCS'88), Vol. 2, pages 592-608, November-I

[Goldberg&Robson83] A. Goldberg and D. Robson, Smalltalk-80: the its Implementation, Series in Computer Science, Addison Wesley,

[Graube89] N. Graube, Metaclass Compatibility, same volume.

[Ingalls&Borning82] D.H.H. Ingalls and A.H. Borning, Multiple I Smalltalk-80, Proceedings of the National Conference on Artific pages 234-237, August 1982.

[Malenfant&al89] Malenfant, G. Lapalme and J. Vaucher, ObjVProlo in Logic, ECOOP'89, July 1989.

[Ungar&Smith87] D. Ungar and R.B. Smith, Self: The Power of OOPSLA'87, pages 227-242.

[Wolinski89] F. Wolinski, Le Système MV^2C: Modélisation et d'Interfaces Homme-Machine, Report 89/38, Laforia, Université Curie, Paris, April 1989.