Programming with Explicit Metaclasses 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

Abstract

This paper discusses the introduction of explicit metaclasses à la ObjVlisp into the Smalltalk-80 language. The rigidity of Smalltalk metaclass architecture motivated this work. We decided to implement the ObjVlisp model into the standard Smalltalk-80 system. The resulting combination defines the Classtalk platform. This platform provides a full-size environment to experiment with class-oriented programming by combining implicit metaclasses à la Smalltalk and explicit metaclasses à la ObjVlisp. Obviously, these experiments are not limited to the Smalltalk world and will be useful to understand and practice the metaclass concept advocated by modern object-oriented languages such as ObjVlisp and CLOS.

1 Introduction

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

1.1 Metaclasses are Useful

It has already been argued that metaclasses are useful both at the user's and at the implementor's levels to describe and extend the class architecture.

For the implementor, metaclasses are the means to describe and parameterize the object system itself, for instance to tailor the implementation [Cointe&Graube88], describe and extend the language in a circular way [Bobrow&Kiczales88] [Attardi&al89], and control the execution process [Malenfant&al89]. In short to describe and control the implementation of objects at the user's level.

For the user, metaclasses define the *class methods*, which allow to send messages to classes, e.g. the messages to create new objects, and the *instance variables at the class level*, which enable the user to parameterize classes [Cointe87].

1.2 Metaclasses in Smalltalk

Historically, Smalltalk was the first language to introduce metaclasses. At the implementation level, they define the kernel of the architecture (in Smalltalk-80, the metaclasses of the Kernel-Classes category) in an object-oriented manner. But at the user's level, metaclasses have been hidden. When a class is defined, a new metaclass is automatically created by the system. This implicit metaclass is anonymous, unsharable and strongly coupled with its private instance.

This separation between the implementor's level and user's level results in an architecture which is not fully uniform. This choice was probably made in order to make things easier for the beginner, but complicated the general architecture of the class system in such a way that it became very difficult to understand it. Consequently people working in the field of learnability of object-oriented programming claim that the Smalltalk metaclasses complicate unnecessarily the model and that they should be removed or at least highlighted

[Borning&OShea87]. Nevertheless, taking the decision to remove metaclasses can lead to removing classes too, and to defining prototype-based Smalltalk languages [Ungar&Smith87].

1.3 Metaclasses in ObjVlisp & CLOS

On the contrary, many people have been looking for uniform and explicit metaclasses. Such systems are Loops, ObjVlisp, CLOS and others. We proposed the ObjVlisp model [Briot&Cointe87] which supports a simple, clean and minimal architecture for explicit metaclasses. The Common Lisp Object System (CLOS) [Bobrow&Kiczales88] has also been designed along such an architecture.

Meanwhile ObjVlisp has the drawback of its minimality. It does not have enough class libraries to allow realistic experiments with end-users. CLOS is a much richer language but there are currently few implementations and its programming environment is still under development.

1.4 Motivations

A previous study [Cointe88] convinced us that the Smalltalk language was extensible enough to support another metaclass system. Because we think Smalltalk-80 is currently the most complete and flexible object-oriented programming environment, we decided to introduce the uniform architecture of ObjVlisp metaclasses into it. This integration must be complete in order to experiment with (meta)class-oriented programming while still reusing standard Smalltalk-80 class libraries. The resulting system, named Classtalk, provides libraries of metaclasses which the programmer may combine as buliding blocks to design an unlimited number of metaclass levels.

1.5 Outline of the Paper

Section 2 discusses the limitations of the Smalltalk-80 metaclass architecture, namely the private class/metaclass "module" and the non-uniform protocol of instantiating objects. Section 3 reviews how the ObjVlisp and CLOS architectures fill these gaps. Section 4 discusses two alternatives to integrate the ObjVlisp architecture into Smalltalk-80, then describes in detail one implementation. Section 5 describes how we extend the standard Smalltalk-80 programming environment to provide a specific one suitable for Classtalk's explicit metaclasses. Section 6 introduces a basic library of metaclasses. Section 7 explains how we merge the Borning&Ingalls' multiple inheritance scheme into Classtalk. Section 8 gives an example of metaclass combination. In section 9 we present our implementation of uniform creation. Section 10 discusses the new issues raised by this work before concluding.

2 The Smalltalk-80 Arcanes

2.1 Kernel Metaclasses

Like ObjVlisp or CLOS, Smalltalk-80 uses a set of explicit metaclasses in order to describe classes. We call them *kernel classes*. Class describes *standard classes* (classes which are not metaclasses), and Metaclass describes metaclasses. To express the common properties of standard classes and metaclasses, they are both direct subclasses of ClassDescription, itself a subclass of Behavior. The inheritance hierarchy of the kernel classes is shown below. The instance variables are enclosed within ().

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

Note the structural difference between a class and a metaclass. A metaclass uses the backward pointer thisClass to memorize its private metaclass, while a class has name, classPool and sharedPools variables.

2.2 User's Metaclasses

Besides this primitive kernel architecture, the Smalltalk's designers chose to hide the metaclass architecture from the user and to provide an implicit and automatic metalevel for standard classes.

When a new class is defined, e.g. class Actor, the system automatically creates a *class/metaclass module*. This means that the system first creates a new implicit metaclass and then instantiates it in order to create the class which will be its sole instance. Such an implicit metaclass is anonymous and is only reachable by sending the message class to the class it describes, e.g. Actor class. The browser connects the definitions of the class and of

its metaclass through the instance/class switch view of the browser.

The user may define methods at the metaclass level. These methods describe messages which may be sent to the class itself, and are named *class methods*. In order to extend the structure of standard classes, the user may also define instance variables at the metaclass level. Nevertheless these variables have no specific names and are not part of the Smalltalk terminology. They must not be mistaken for *class* or *pool variables* which implement shared variables.

2.3 The Implicit Class/Metaclass Module

Being implicitly created by the system, the inheritance and instantiation of metaclasses should obey some implicit rules. To provide the same inheritance rule for class and instance methods, the inheritance hierarchy of metaclasses is parallel to the inheritance hierarchy of classes. In order to have the same structure and behavior for all implicit metaclasses, each of them is created as an instance of Metaclass. Smalltalk-80 connects the metaclass inheritance hierarchy to the class hierarchy by declaring the most general metaclass, Object class, a subclass of 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 ()
```

Nevertheless the implicit class/metaclass module provides too rigid a coupling between a class and its metaclass. This leads to limitations in the expressiveness of the language as illustrated by the following example.

2.4 The abstract Class Counter-Example

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

A simple example of *abstract class* appears when one tries to model complex numbers as objects. Two representations are useful for complex numbers, namely cartesian and polar coordinates. Therefore we define two classes, respectively Cartesian and Polar to implement them. The abstract class Complex factors the common behavior, for instance computing arithmetic. In the inheritance hierarchy figure, methods are enclosed within <>.

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

The problem is to model the general behavior of an abstract class, and more precisely, to ensure that such a class cannot create instances. The obvious way is to forbid instantiation by redefining the standard method for creation (in fact allocation) in order to raise an error. This standard method is named new and belongs to class Behavior. It should be redefined as a class method. Therefore we need to introduce a standard class, named Abstract. Its only purpose is to provide a metaclass.

```
!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 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 Complex class inherits the redefinition of the method new owned by Abstract class. Unfortunately, classes Cartesian and Polar both inherit from Complex. Consequently their corresponding metaclasses also inherit the forbidden instantiation. Thus, they become abstract classes too, and it will be impossible to

create any complex number. The rule for implicit inheritance of metaclasses does not match our intuition.

A pragmatic solution is to change explicitly the inheritance rule by updating the instance variable superclass, which specifies the inheritance link. Therefore we declare the most general metaclass, i.e. Object class, as the new superclass:

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

This is an ad hoc solution and which lacks modularity since we need to redefine inheritance for *every* subclass. The complete solution, given in section 3.2, uses explicit control of inheritance and instantiation of classes.

2.5 Non Uniform Creation

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

When an object is allocated, the values associated to its instance variables get the default initial value nil. In order to initialize these variables, no standard method is provided, and therefore one needs to define explicitly an initialization method. For instance, we define such a method which initializes Cartesian instances:

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

If we want to combine allocation and initialization into a single message for 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 in most cases specific to each class, because their selectors are built from the names of the instance variables. However, there is a method to create standard classes. All standard classes share the same structure (instance variables defined or inherited by Class) and are created by the method subclass:instanceVariableNames:...category:. But this assumption does not stand anymore when adding new instance variables at the class level (see section 6.4).

3 The ObjVlisp & CLOS Alternative

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

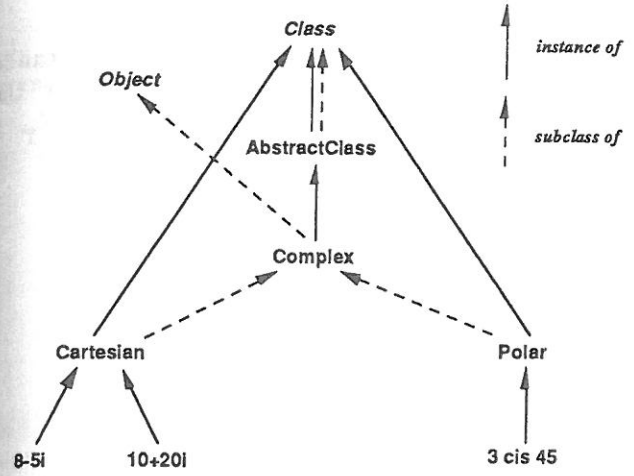
ObjVlisp and CLOS are two systems which propose such an architecture. ObjVlisp is also minimal in the sense of being self-defined by only two classes: the root of the instantiation tree (Class), and the root of the inheritance tree (Object). Class, being an object, must itself be described by (and must be an instance of) some class. The minimal solution proposed in [Briot&Cointe87] defines Class as an instance of itself. This self-instantiation ensures a complete uniformity and self-description (reflexivity) of the kernel.

3.1 Explicit Metaclasses

An ObjVlisp metaclass is a class which can have access to the standard allocation message by owning it or by inheriting it. Class, as the holder of the standard allocation method allocateInstance, is the first metaclass of the system. In order to inherit this standard allocator, a new metaclass is always created as a subclass of a previous one. As opposed to Smalltalk-80, there is no difference between classes and metaclasses. Consequently, the two metaclasses of Smalltalk-80 (Class and Metaclass) are merged into one (Class).

3.2 Abstract Class Revisited

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



There are three steps to this solution:

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

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

• create the two classes Cartesian and Polar as instances of Class and 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, the creation of objects is uniform. It is achieved by combination of an allocation and an initialization method:

creation = allocation + initialization

Class holds the standard allocation method, named allocateInstance, and the standard creation method, named makeInstance:. There are two standard initialization methods, both of them named initializeInstance:. The first one is owned by Object and defines standard initialization of objects. The second one, owned by Class, defines initialization of classes. Initializing classes is more complex and includes for instance compiling static inheritance of instance variables. Consequently this second initialization method specializes (and calls) the most general initialization method owned by Object. Here is the inheritance hierarchy of the ObjVlisp kernel:

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

Compared with Smalltalk, the ObjVlisp makeInstance: method includes parameters for object initialization, which it transmits to the initializeInstance: method, whereas the Smalltalk-80 method new is a simple allocator (equivalent to allocateInstance) and 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 automatic coupling between a class and its metaclass,

• introducing a unified method of creation which takes into account both the allocation and the initialization procedures.

Smalltalk-80 is extensible enough to propose a clean solution to the first problem. But its somewhat limited syntax makes it difficult to find a simple solution to the second problem. The result of our implementation, a subworld of explicit (meta)classes embedded into the standard Smalltalk-80 system, was named Classtalk, because the class concept is at its core.

4.1 Creating Classes Explicitly

In order to create a class as an explicit instance of a metaclass we introduce the new creation message:

newName:superclass:instanceVariableNames:category:. Its keywords are taken from ObjVlisp while retaining the Smalltalk-80 syntax and conventions. As advocated by ObjVlisp, class and pool variables are suppressed for the sake of simplicity. This new creation message is sent to the metaclass, i.e. the creator, and not to the superclass, as in standard Smalltalk-80. This follows the principle of creating every object as an instance of a class.

4.2 Implementation Alternative

We have to ask ourselves which metaclass should own this new creation method. More generally the question is: "How do we transpose the ObjVlisp kernel into the Smalltalk-80 architecture?". At the implementation level, two answers may be given:

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

4.2.1 Merging

Class already owns the standard method subclass:instanceVariableNames:...category: for creating standard Smalltalk classes. By identifying the ObjVlisp metaclass Class with the Smalltalk-80 class Class, the method newName:...category: becomes 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 indirect superclass of its metaclass Class class. This provides an implicit self-description of Class. But, as opposed to ObjVlisp, this self-description is partial, because Class class is not equal to Class.

4.2.2 Grafting

The grafting scheme makes it more difficult to express the self-instantiation of the first metaclass (Classtalk). Nevertheless we can change the implicit rule of Smalltalk metaclass inheritance to make Classtalk class a direct subclass of Classtalk:

```
Classtalk class superclass: Classtalk
```

We obtain two different inheritance trees: one for the structure and one for the behavior:

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

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

The grafting scheme allows a precise definition of Classtalk classes. Unused instance variables such as classPool and sharedPools are no longer defined. Nevertheless the instance variable name and some methods of Class need to be copied into Classtalk.

Both solutions are almost equivalent. In this paper we chose the grafting scheme, in order to easily distinguish between Smalltalk and Classtalk classes.

4.3 Explicit Creation of Classes

The implementation of the method newName:...category: to create Classtalk classes follows the standard implementation of class creation. It includes a dispatch by the type of the superclass (with named or indexed variables). As in standard Smalltalk-80, the "auxiliary method" newName:environment:...category: shares a common implementation between classes with named or indexed instance variables.

To focus on the semantics of these two methods, we give their definitions without the type dispatcher and without the pieces of code related to the management of the programming environment (syntax check, changes 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 instanceVariableNames: i
    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."
    o isNil ifFalse: [newClass otherSupers: o].
    "(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 stages: allocation (1) and initialization (2 & 4). The new class allocated (temporary variable `newClass`) is defined explicitly as an instance of a previous metaclass: `self new` (1). As in standard Smalltalk-80, the initialization process takes place in two successive steps: (2) and (4).

• to organize Classtalk classes in a specialized browser we introduce a new organizer, the global variable `ObjVlispOrganization` which is coupled with the Classtalk browser.

• the method `newName:environment:..category:` introduces a parameter prefixed by the keyword `otherSupers:`. It specifies an unused array of superclasses (calling value is `nil`). Meanwhile, this allows this method to be reused when introducing multiple inheritance (see section 7).

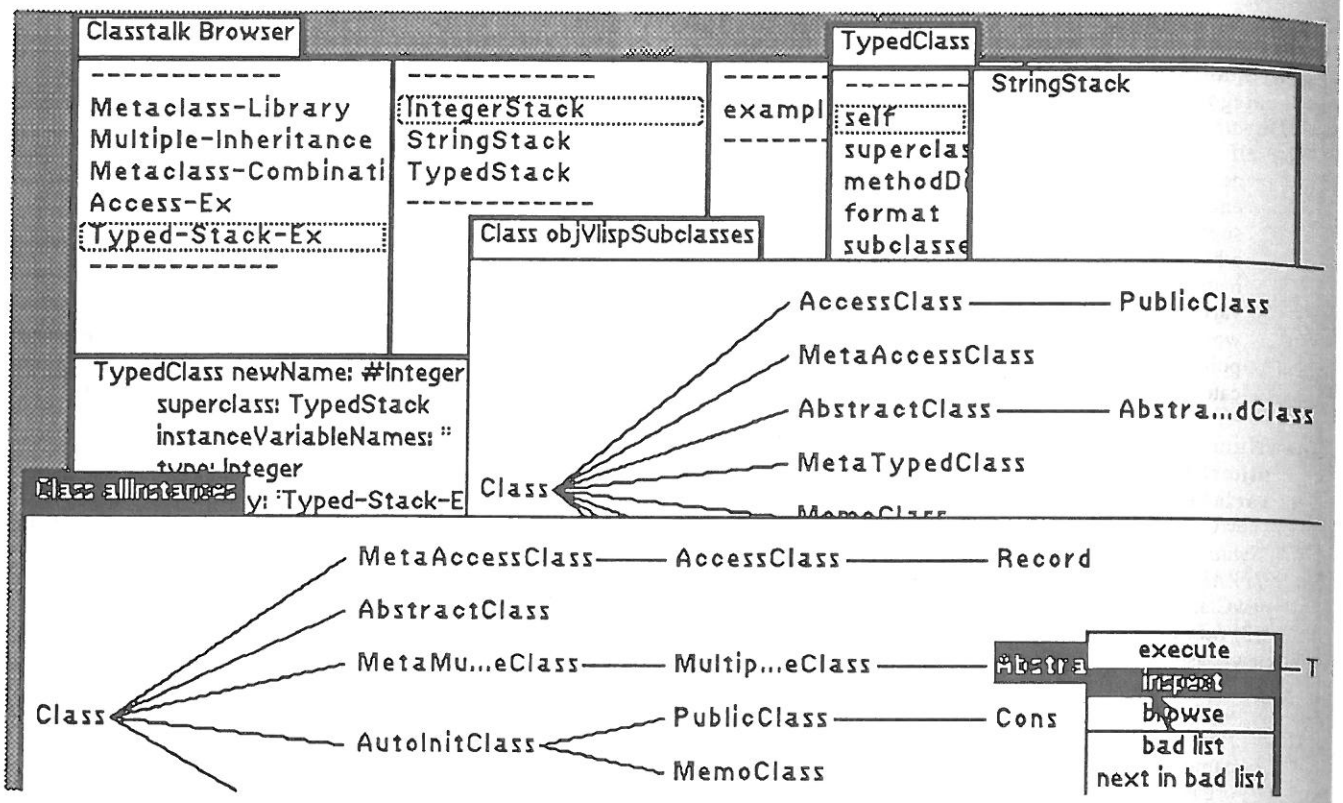
• expressions (3) and (5) are evaluated only in the case of multiple inheritance. (3) assigns the array of remaining superclasses. (5) calls the management of multiple inheritance provided by the standard extension of Smalltalk-80 [IngallsBorning82]. This will recompile the methods or generate conflicting methods when needed.

5 The Classtalk Environment

The Smalltalk-80 standard browser may confuse the programmer when browsing on Classtalk classes. When the instance/class switch is set to class, the browser shows the explicit metaclass, and not an implicit one as in standard Smalltalk-80. Moreover the template and the definition printed in the browser do not reflect the Classtalk definition.

Therefore we designed a browser specifically dedicated to Classtalk classes. The differences lie in the removal of the instance/class switch and the adjustment of templates and definitions in order to make clear the Classtalk way of creating classes.

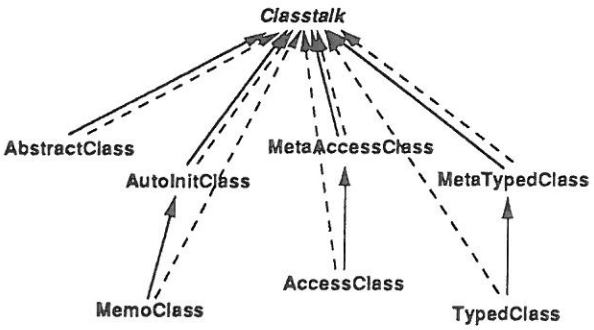
This browser is also interfaced with a generic tree editor [Wolinski89] in order to browse both the instantiation and the inheritance graphs.



6 Library of Metaclasses

This new browser was helpful to develop a library of Classtalk metaclasses. Our idea is to reuse them as building blocks to define more complex metaclasses by combining them with both the instantiation and inheritance mechanisms.

In this section we propose to introduce and explain some of them. Let us recall the creation rule for these metaclasses: each Classtalk metaclass is a subclass of another explicit metaclass. Our naming convention is that they end up with Class.



6.1 AbstractClass

This metaclass models abstract classes, i.e. non-instantiable classes, as defined and used in section 3.2.

6.2 AutoInitClass

This metaclass models classes which provide their instances with automatic initialization.

In order to get automatic initialization of objects, every Smalltalk-80 programmer has at least once redefined the class method new. To avoid code duplication, we model this behavior in the specific metaclass AutoInitClass. A class instance of AutoInitClass has the following behavior: after being created a new object will automatically receive the message init:

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

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

6.3 MemoClass

This metaclass models classes which memorize the collection of all their instances by using an explicit backpointer.

This backpointer is implemented by a new instance variable `instances` added at the metaclass level. Its value is an ordered collection remembering all the instances which are created.

This variable needs to be initialized to an empty collection before starting to create instances. In order to provide automatic initialization, we define `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 models classes which are parameterized by a type [Cointe87].

`TypedClass` introduces the new instance variable `type` and two associated accessor methods. In order to provide an explicit initialization of this variable, we need to extend and specialize the standard Classtalk message for creating classes. The new creation method `newName:...type:category:` combines the standard `newName:...category:` with the assignment of the type. Meanwhile, the definition of this new method led us to introduce the new metaclass `MetaTypedClass` whose only goal is to hold this extended creation method. A non-uniform initialization forces us to reintroduce the class/metaclass module:

```
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 models classes which may provide automatic (read-write) accessors to their instance variables.

Another repetitive programming problem lies in the definition of accessor methods. Their selectors are usually associated with the instance variables to which they give access. In order to relieve the programmer from this routine, we propose the metaclass `AccessClass` which describes how to generate automatically such accessors. The programmer can specify which instance variables will be public (i.e. with accessors) by using the declaration `public:`.

The following example is the Classtalk solution to the example described in [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 to introduce a new metaclass, named `MetaAccessClass`, to define the extended creation method.

This method, named `newName:...public:category:`, will compose the standard `newName:...category:` method with the call of the method to generate accessors. This method, named `makeIvAccessOn:`, is owned by `AccessClass`. A *scanner* parses the string specifying public variables into an array which becomes the parameter of the 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)! !
```