Metaclasses

This module considers the implementation of the class structure within Smalltalk, and introduces the Metaclass concept. Classes such as ClassDescription and Behavior are explored. The concepts here are widely misunderstood, possibly because of the tongue–twisting terminology used; an attempt is made to clear away the confusion in this module. (It's also worth saying that some of the ideas presented here can be difficult to understand, and in practice, you need to know almost nothing about metaclasses to use the system effectively.)

1. Recap of terminology

- Every object is an *instance* of a *class*
- Every class (except Object) has a *superclass*

Example: 3@4 is an *instance* of Point. Point is a *subclass* of Object (indirectly, via classes ArithmeticValue and Magnitude). See Fig.1.

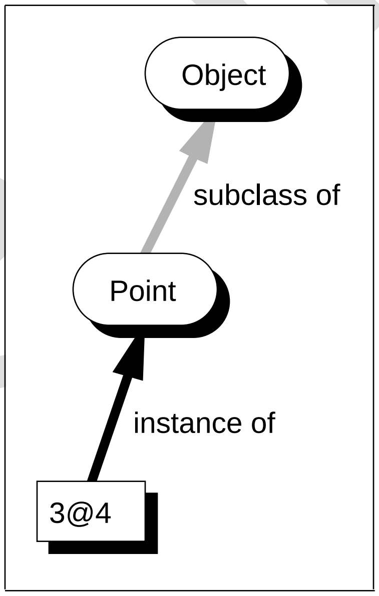

Figure 1: Instance vs. Subclass

To define things more precisely: an object is a class *if and only if* it can create instances of itself. There are two fundamental mechanisms which can create objects: the primitive methods new and new: defined in Behavior (basicNew and basicNew: are aliases for these). *Any* object that understands these messages (and eventually executes the definitions in Behavior) is a class.

Although you're not supposed to create instances of abstract superclasses, there's nothing to stop you doing so if you really want to.

Final note: the names of instance variables begin with a lowercase letter, while those of classes begin with an uppercase letter. Additionally, the first letter of a method selector is usually lowercase. Example: class is either an instance variable, or a selector, whereas Class is a class.

2. Why do we have classes and metaclasses?

In Smalltalk-72 classes were not objects. The users of Smalltalk-72 (at Xerox only) found this a problem: you couldn't send messages to classes, lack of orthogonality, etc.

In Smalltalk–76 classes became objects. Applying the rule that every object must be an instance of a class, this meant that classes had to be instances of something: Class. Class, being able to create classes, was itself a class, an instance of itself in fact — Fig.2. (This is the situation in Little Smalltalk.)

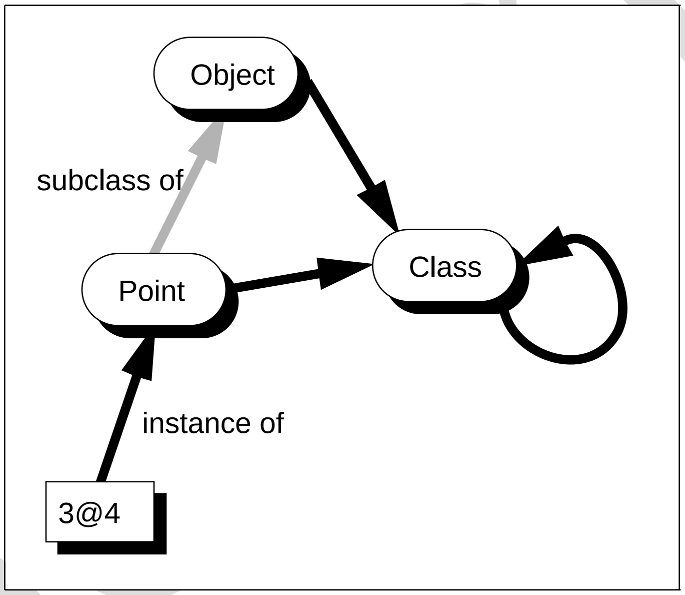

Figure 2: Smalltalk–76

3. So What's The Problem?

As the researchers at Xerox used the system they found that initializing objects was a little painful: If all classes are instances of Class, they all behave identically. This means that there is no class/instance distinction in the browser, and that the only way to create objects is by:

1. Sending new or new: to the class, then
2. Sending an initialization message to the created instance.

For example, creating and initializing a Point was done by:

```
Point new x: 4 y: 5
```

A common source of bugs was to forget to initialize the newly created object, or initialize it incorrectly. Clearly, it is desirable that the instance creation messages either:

1. Perform the correct initialization automatically, or
2. Insist that appropriate parameters be supplied.

For example, browsing the *instance creation* category of messages that can be sent to Point, you see only x:y: and not new. This is as it should be.

4. The Solution: Metaclasses

If Point is to be able to respond to x:y:, while Object is not, then Point and Object cannot be instances of the same class. The solution adopted in Smalltalk–80 was to have every user-defined class the sole instance of another class, termed its *metaclass*. Every metaclass is of course an instance of a class, but as all metaclasses behave identically they are instances of the same class, called Metaclass. (Pay attention to the case of the first letter, and to the font used!)

Note: all objects can be sent the message class to find out what their class is. So just as (3@4) class responds with Point, Point class responds with its metaclass. However, since metaclasses are system-generated, and in one-to-one correspondence with the user-defined classes, they are not named directly, but always via their related class. So when you print Point class, the answer is Point class!

Following the argument further, Metaclass must also be an instance of a class. Noting that Metaclass is a class with multiple instances, it follows that Metaclass is an instance of its metaclass, and that its metaclass, like all the other metaclasses, is an instance of Metaclass itself. By now you're probably totally confused, so it's time for a diagram! (See Fig.3.)

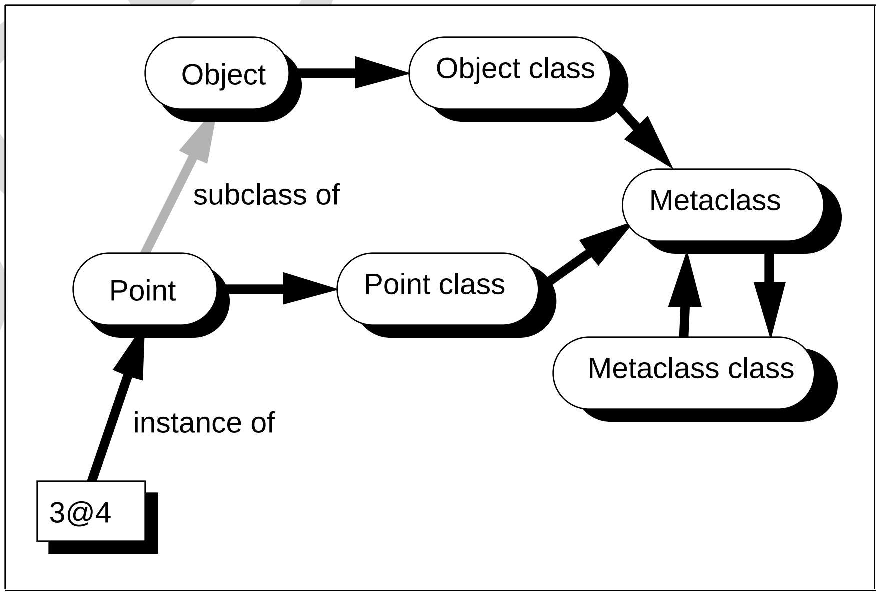

Figure 3: Metaclasses

In summary, we have

```
(3@4) class == Point "Point class is the metaclass of Point"
Point class class == Metaclass
Metaclass class class == Metaclass
```

5. So what do I care?

You may well ask. As far as most development is concerned, you needn't care. The creation of metaclasses is hidden so well by the programming environment that you needn't be aware of it. The only thing you need be aware of is that just as a class specifies the behavior of its instances in its instance methods, the behavior of the class itself (the class methods) is specified by its metaclass.

6. And for my next trick…

So far we have only been talking about *instance* relationships: 3@4 is an instance of Point, Point is an instance of its metaclass, its metaclass is an instance of Metaclass, etc. But earlier we said that all classes except Object have a superclass. What is the superclass of Point, its metaclass, Metaclass and its metaclass? (Hint: you can find out the superclass of a class by sending it the message superclass; or take a look at Fig.4)

In general if X is a subclass of Y, then X class is a subclass of Y class. While it would have been perfectly feasible to build the system in such a way that every metaclass was a direct subclass of Object, building it this way means that class methods are inherited using a parallel hierarchy.

7. Can't see the wood for the trees?

This means that there are two, parallel hierarchies: one for the (user-defined) classes, and one for their metaclasses. But the *only* class that has no superclass is Object, so what is the superclass of Object class?

Because all the instances of the metaclasses are classes, it makes sense to concentrate classness into one class, Class, and make that the superclass of Object class. (See Fig.4.) Furthermore, because classes and metaclasses are very similar, Class and Metaclass are both subclasses of an abstract class, Behavior. Behavior provides all the protocol for making new instances, while an intermediate class, ClassDescription provides extra protocol used by the programming environment: class comments, for example; see Fig.5.

8. Some interesting properties

1. All objects, including classes and metaclasses, are instances of classes that have Object as their ultimate superclass. This means that *all* objects respond to the messages defined in Object.

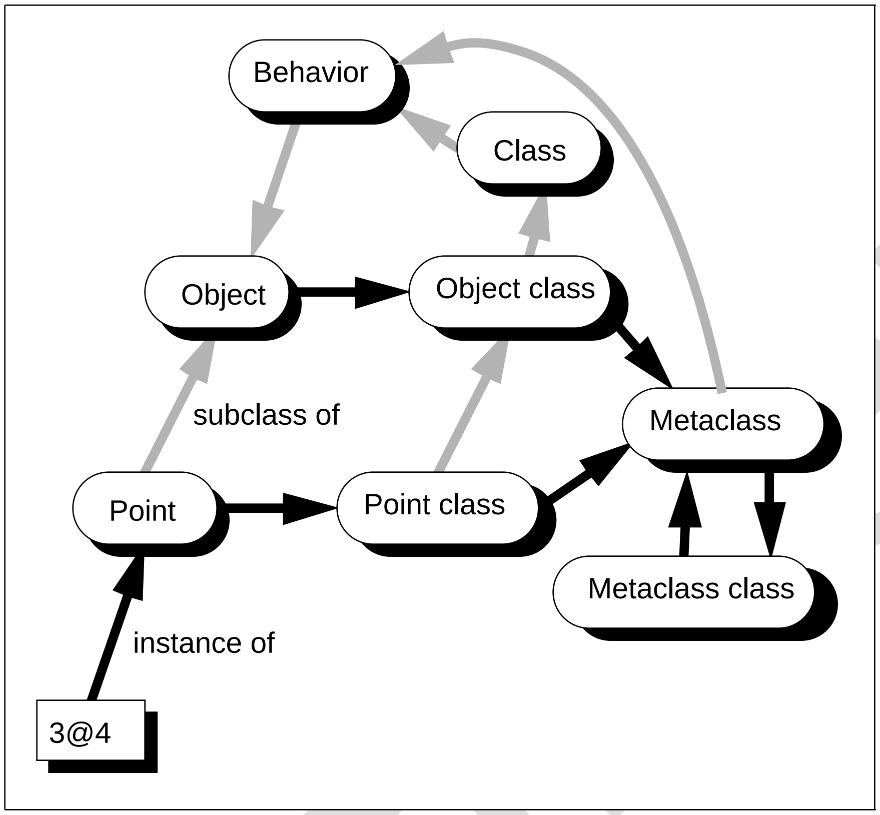

Figure 4: The superclass of Object class?

2. All classes and metaclasses are instances of classes that have Behavior and ClassDescription in their superclass chain. This means that new is only defined in one place, and that all classes and metaclasses have protocol to support the programming environment.

3. If an instance method is defined in the "connecting bridge" (Behavior, ClassDescription), then it is also available as a class method.

Thus the problems seem to have been solved. The solution has a certain amount of elegance to it, but has been criticized for being complex and confusing. This is probably due to the tongue-twisting terminology used, and the lack of suitable explanatory material in the textbooks (it's all there, but it's complicated).

Various authors have proposed going back to the situation in Smalltalk-76 (i.e., all classes are instances of Class), and other alternatives have also been suggested. It is doubtful that any will be adopted, but it remains to be seen whether anyone can come up with a better solution.

9. Some useful protocol in Behavior

There are many methods defined in class Behavior, but the most useful ones are to be found in the following protocols: 'accessingclasshierarchy', 'accessingmethoddictionary',

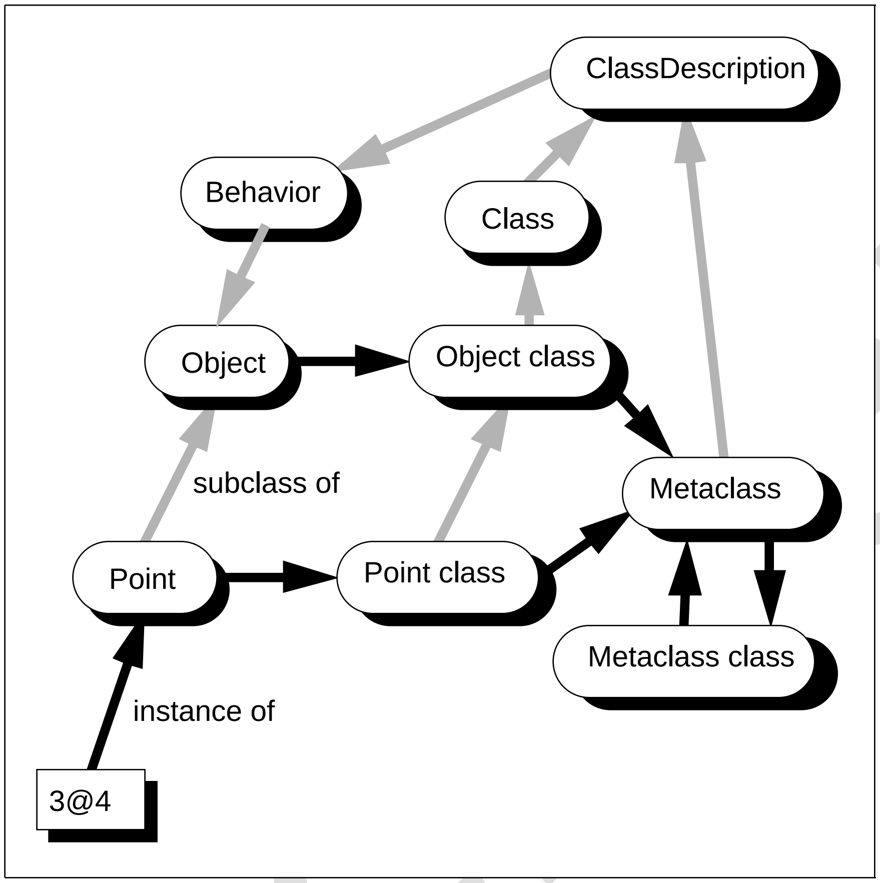

Figure 5: The Actual Hierarchy

'accessinginstancesandvariables', 'testing class hierarchy', and 'testingmethoddictionary'. These include:

subclasses	returns an instance of class Set containing all the *direct* subclasses of the receiver.
allSubclasses	returns an OrderedCollection of the direct subclasses, and their subclasses, and their subclasses, and so on.
allSelectors	returns a Set of all the selectors to which instances of the receiver will respond.
allInstances	returns a Set of all the instances of the receiver in the image.
instanceCount	returns the number of instances of the receiver in the image.

Table 1: Useful Behavior protocol

someInstance	returns an arbitrary instance of the receiver, or nil if there are no instances in the image.
inheritsFrom: aClass	returns true if the receiver is a subclass (direct or indirect) of aClass.
canUnderstand: aSelector	returns true if instances of the receiver respond to aSelector.
whichClassIncludesSelector: aSelector	returns the class in which the response to the message aSelector is found.

Table 1: Useful Behavior protocol (Continued)

Ex 1. Browse classes Class, Metaclass, ClassDescription and Behavior. You might also like to try inspecting an instance of a metaclass (i.e. a class), by sending the inspect message to a class of your choice. For example:

```
Point inspect
Metaclass inspect
```

Ex 2. You might also like to try exploring the class and metaclass hierarchies using the class and superclass messages. For example, what is the result of the following expressions?

```
42 class superclass class class superclass
Metaclass superclass class superclass superclass class
```

You should be able to work out the expected result from Fig.5.

Ex 3. Try some of the methods defined in class Behavior.

Ex 4. Metaclass and Metaclass class are instances of each other. Can you think of a way to create a pair of objects that are instances of each other? What about making an object that is an instance of itself? Can you envisage any uses for such objects?

10. Class instance variables

Any class may have class variables. A class variable is accessible to the class methods, the instance methods (i.e., by the class's instances), and also by any subclasses and their instances. All of these classes and instances access the *same* variable.

However, it is occasionally desirable to have a class variable which is not shared by subclasses; each subclass has a different variable. This is possible in VisualWorks by noting that each class is the sole instance of its metaclass. Thus, by adding extra instance variables into the specification of the metaclass, the class can have its own private variables. Class instance variables are *only* accessible in the class methods of the class;

they are not accessible in the instance methods. If you browse references to the variables (using the “instance variables” menu item when the class methods have been selected) you will see that they cannot be modified outside the class.

The use of class instance variables is little understood, mainly because there are few examples of their use in VisualWorks. As its name suggests, a class instance variable is an instance variable for a class. At first this may seem a little confusing, but you should remember that every object is an instance of some class, thus every class is also an instance of some class. We have seen earlier that instance variables are inherited by classes, similarly, class instance variables are inherited.

It’s important to distinguish between the use of a class variable and a class instance variable. For example, suppose class Persian inherits from class Cat. If Cat has a class variable, then Persian has the exact same class variable and value, i.e. if an instance of Persian modifies it, then instances of all subclasses of Cat will refer to that new value.

On the other hand, if Cat has a class instance variable, then all subclasses of Cat (including Persian) have their own copy of that variable and therefore can have their own private values.

Although there are not many examples of the use of class instance variables in VisualWorks, there is one which is a good example: class UILookPolicy. This class is an abstract superclass for classes that emulate the “look–and–feel” of various window managers; its subclasses provide specific emulation for Macintosh, Windows, Motif, or MS–Windows. It introduces three class instance variables: systemAttributes, systemWidgetAttributes and useSystemFontOnWidgets. Each of its subclasses initialize these variables in their respective class initialize methods to provide class–specific values. It is important to note that the class instance variables can only be accessed by class methods.

Ex 5. Open a Hierarchy Browser on class UILookPolicy. Browse references to the class instance variables mentioned above. Where are they initialized?

Ex 6. Open a Browser on all those classes that contain class instance variables. Hint: The following code returns true if the receiver has a class instance variable.

```
aClass class instVarNames isEmpty not
```