Let's Agree on the Meaning of 'Class'

Anthony J H Simons

Contact Information

Dr Anthony J H Simons, Department of Computer Science, University of Sheffield,

Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.

Email: A.Simons@dcs.shef.ac.uk

Tel: (+44) 114 222 1838 (direct; +voicemail) (+44) 114 222 2000 (exchange)

Fax: (+44) 114 278 0972 (soon to change)

WWW: http://www.dcs.shef.ac.uk/~ajhs/

Abstract

It is a curious thing that after 30 years of object-oriented programming, we still have no consensus on the meaning of *class*. The OMG 1991 standard avoided using the term *class* altogether; and current design notations such as UML and OML still contain certain misconceptions regarding *class*: the debate on *type* versus *implementation* is old and focusses on the wrong argument - a class is not strictly either one of these things. This article unpacks 10 years of theoretical work on extensible interfaces and implementations to explain exactly how a *class* is different from a concrete type. This work impacts widely on current standards, definitions and notations.

Conference Stream

Research paper (presents old results in new light).

Subject Areas

Theoretical Foundations, Analysis and Design Methods, Software Engineering Practice

Let's Agree on the Meaning of 'Class'

Anthony J H Simons, Department of Computer Science, University of Sheffield,

Regent Court, 211 Portobello Street, Sheffield S1 4DP, United Kingdom.

Abstract

It is a curious thing that after 30 years of object-oriented programming, we still have no consensus on the meaning of *class*. The OMG 1991 standard avoided using the term *class* altogether; and current design notations such as UML and OML still contain certain misconceptions regarding *class*: the debate on *type* versus *implementation* is old and focusses on the wrong argument - a class is not strictly either one of these things. This article unpacks 10 years of theoretical work on extensible interfaces and implementations to explain exactly how a *class* is different from a concrete type. This work impacts widely on current standards, definitions and notations.

1. The Ambiguity of *Class*

What is classification? After thirty or so years, object-oriented languages have still not reached a consensus. This is somewhat alarming, since classification is arguably what makes a language distinctively *object-oriented* [Wegn87]. Perhaps the biggest obstacle to a common understanding of the notion of *class* comes from the multivalent use of the concept in existing languages. By default, one is left to assume that *class* replaces the usual notion of *type*, especially in languages without strong typing [GoRo83]. Class names are used everywhere as type identifiers [CoNo91, Stro91, Meye92]. Yet, a class is different from a standard type in that it is an incomplete description, open to extension through the mechanism of inheritance - but what do we mean by open-ended types? On the other hand, a class is like a package or module [IBHK79, Wirt82], the concrete implementation of a type. Yet, a class is also only a partial implementation, expecting further extensions - what kind of construct is an open-ended implementation? Or again, should the term *class* refer to the defining characteristics of a group of objects, or to the group so defined?

We can identify at least three main dimensions of ambiguity in the current usage of the term *class*:

- **Extension-Intension Dimension**

    1. class as *extension* - a set, referring to the family of objects which share some characteristic properties and behaviour; used when partitioning a domain into sub-groups;

    2. class as *intension* - a definition, describing the characteristic properties and behaviour of a family of objects; the admission criteria for group membership;

- **Abstract-Concrete Dimension**

    3. class as *specification* - providing the visible type, the interface (or *protocols*) for a family of objects, its instances;

    4. class as *implementation* - providing the instance creation template, a table of shared (or *static*) data and functions for a family of objects, its instances;

- **Monomorphic-Polymorphic Dimension**

    5. class as *monomorphic* - providing the exact concrete type of its own instances; used when creating objects;

    6. class as *polymorphic* - providing the upper bound on a heterogeneous family of objects that can be assigned to a given variable; used when attaching type information to variables.

We can resolve the first ambiguity immediately, by agreeing to distinguish a *class definition* (2) from its implicitly associated *class family* (1), where the need arises. The meaning (1) is that assumed in history, sociology and biological taxonomy, whereas (2) is that most often assumed in our field. These two definitions are of primary importance, since all our notions of classification derive from the ways in which we categorise not just software abstractions, but the world in general.

The abstract-concrete dimension has been considered previously [Snyd86, Lisk87, Sakk89, Amer90] and these arguments are by now fairly well understood. Languages like Eiffel [Meye92], Trellis [SCBK86], Sather [Omoh94] and C++ [Stro91] prefer the view that a class (3) describes a type with an associated implementation; whereas the more cavalier use of inheritance allowed by Smalltalk

[GoRo83] prompts the skeptical to consider that class (4) and type are entirely separate concerns [Snyd86, RaLe89, Amer90]. The more recent introduction of mechanisms to handle type compatibility independently from the class hierarchy, such as *protocols* in Objective C [NeXT93] and *interfaces* in Java [KaWa96], continues to force a wedge between type (3) and class (4). This view is now becoming quite the norm in emergent standards [UML97, OML97] which endorse phrases similar to: "a class realises, or implements a type", conveying the implicit demotion of the class (4) to a mere unit of implementation, quite at odds with (2), which is closer in spirit to (3).

The monomorphic-polymorphic dimension has been considered in theoretical treatments [CaWe85, CCHO89a, CoPa89, CCHO89b, CHC90, Simo96] but the contribution offered by this valuable work to practical standards and definitions has largely been overlooked. Quite natural definitions of class (6) and type (5) fall out of this work, which have yet to be appreciated by the object-oriented mainstream. It is a common misperception that the state of the art still revolves around the old abstract-concrete debate. The popular belief is that by shifting the role of class (4) hierarchies to focus on implementation sharing alone, this frees up type (3) hierarchies to describe, independently, sets of subtyping relationships between abstract types. However, the article [CHC90] proved once and for all that the kind of type compatibility adopted in the majority of object-oriented languages is **not** subtyping. Against this, significant papers still appear [DGLM95] asserting that "most object-oriented languages have subtype polymorphism", which is largely incorrect.

In this paper, we want to bring to bear some of the theoretical results of the last decade on the present formation of standards in notations and definitions, especially those to be adopted by the OMG. We want to get the terminology and relationships in the notation correct. To do this, we present an intuitive model of objects, types and classes (avoiding difficult λ-calculus notation). Among the results we wish to emphasise are that:

- objects are instances of concrete types, rather than of classes - such a concrete type has a fixed implementation and fixed recursive type;
- classes are polymorphic definitions for heterogenous families of objects, instances of different concrete types - such a class has an extensible implementation and an extensible interface;

- the type compatibility relationship in object-oriented programming is (by and large) not subtyping, but polymorphic typing, a more subtle relationship;

- implementation and type extensions follow each other much more closely than the current wisdom would have you believe - in particular, implementation extensions which would violate subtyping are in fact well-typed in a polymorphic regime.

Relegating the *class* concept to the mundane level of an implementation unit constitutes a gross failure of nerve. In focussing too closely on the implementation strategies adopted by a generation of programmers, have we somehow lost sight of the sublime notions of *class* and *classification* (1, 2)? The false division of *class* and *type* along abstract/concrete lines is a red herring. Computational types were always considered from both the abstract and concrete points of view, so why not classes also? We show how the notion of class does indeed have a well-founded definition, which depends more on an appreciation of the monomorphic-polymorphic dimension than the abstract-concrete one.

2. Objects and Types

Our starting point is with a model *object-based* [Wegn87] language, similar to Modula-2 [Wirt82] or Ada (pre-95) [IBHK79], which offers data abstraction and encapsulated types, but not classification or inheritance. We want our understanding of *class* to harmonise with languages where composition [RaLe89] is the only strategy for building larger objects. However, in the model below, we assume that objects "own" the methods used to manipulate them; in other words, to invoke a method, it must be selected from the object which owns it. This contrasts with the Modula-2 and Ada style of applying procedures to objects, and is more in keeping with the "message passing" style.

```
point1.moveTo(3, 4);        -- message passing style
moveTo(point1, 3, 4);       -- procedural style
```

We need a language-independent notation to describe the following intuitive and, we hope, non-controversial concepts:

- object - a single software entity, an instance of a concrete type, that exists at run-time during the execution of a program;

- abstract type - the interface to an object, in terms of the type signatures (or *protocols*) and axioms (or *contracts*) of the public methods used to manipulate the object;

- concrete type - a complete definition for an object, in terms of data storage declarations and method definitions, that implements an abstract type;

Most objects and object types are recursively defined. This is because an object's methods will, in general, invoke other methods owned by the same object. To do this, an object must be able to refer to itself, to select the next method required. We shall therefore take the liberty of allowing recursive definitions in our formal notation[1]. Let us define a simple point object:

```
define object point1 =
{ attributes
        x = 0;  y = 0
  methods
        x() = { return x };  y() = { return y };
        equal(other) = { return point1.x() = other.x() and point1.y() = other.y() };
        moveTo(newX, newY) = { x := newX;  y := newY;  return point1 }
}
```

In this notation, *define object* creates an object, a single instance. The object's state is introduced by the *attributes* keyword and its behaviour by the *methods* keyword. Within these partitions, a semicolon separates declarations. Our formal model adopts the simplification whereby all objects are deemed to contain their own methods, rather than access them indirectly through a shared class table - this simplification has no bearing on the model's formal properties. All access to state[2] variables is handled through methods; these are distinguished as *x* and *x()* respectively. Notice how the definition of *point1* contains recursive references to *point1* in the body of the *equal()* method - this is

---

[1] Technically, recursive definitions are merely equations; we shall assume the result of the fixpoint theory to motivate the existence of recursive definitions - see [Simo96] for a recent theoretical treatment of a similar example. We assume that *define* introduces a generator [CoPa89] and then implicitly takes the fixpoint, for recursive definitions.

[2] Technically, this object is a functional closure, in the tradition of [CCHO89a, CCHO89b, CHC90]. Object state may be modelled as the variables within the static scope of the functional closure when it was defined.

to demonstrate how methods may in general refer to each other, remembering that all methods must be selected from the object (here, *point1*) which owns them[3]. The update method *moveTo()* also happens to return the current object, *point1*, a common practice in some languages.

Since all attributes are private and all methods public, the type of *point1* may be determined by considering the types of its methods:

```
define type Point =
{ protocols
        x() : Integer;  y() : Integer;  equal(Point) : Boolean;
                moveTo(Integer, Integer) : Point
}
```

In this notation, the keyword *protocols* introduces the type signatures[4] of the public methods of the *Point* type; for example, *x()* is a unary method returning an *Integer* result. Notice how this type is recursive: *equal()* accepts another *Point* type argument and *moveTo()* returns a result of type *Point*.

Given the existence of this type, we can create a typed object (an instance with an associated type) by attaching the type information to an object definition:

```
define typed object point1 : Point =
{ attributes
        x : Integer = 0;  y : Integer = 0
  methods
        x() : Integer = { return x };  y() : Integer = { return y };
        equal(other : Point) : Boolean = { return point1.x() = other.x()
                                        and point1.y() = other.y() };
        moveTo(newX, newY : Integer) : Point = { x := newX;  y := newY;  return point1 }
}
```

---

[3] We have deliberately constructed *equal()* to invoke *x()* and *y()* to illustrate mutual method invocation; it would be technically possible for *equal()* to access the *x* and *y* attributes directly.

[4] We have deliberately adopted a programming-language style to express type signatures, rather than the product- and arrow notations normally found in theoretical treatments, for the sake of familiarity.

This nearly looks like a conventional encapsulated type definition - a concrete type with an implementation and an associated interface - however, please note that the *define typed object* expression merely creates a single typed object, *point1*, whose methods contain embedded references to itself, *point1*.

3. Object Derivation and Subtyping

In order to extend the model language to allow the derivation of objects with more structure, we introduce an operator *with* which has the two-fold effect of combining attribute declarations[5] and method definitions[6]. The operator performs field union with overriding, giving preference to fields on the right-hand side which have the same names as fields on the left-hand side. We may use it to create an extended point object, which has extra attributes and methods:

```
define object point2 = point1 with
{ attributes
        selected = false;
  methods
        selected() = { return selected };
        toggle() = { selected := not selected;  return point2 }
}
```

This object, *point2*, is a mouse-selectable "hot" point whose *toggle()* method is invoked to switch between the selected and deselected state. Now, the manner in which *point2* is defined reflects exactly what Stroustrup says in [Stro91, p183]:

> "An object of a derived class has an object of its base class as a subobject."

We can show this by internally evaluating the *with*-operator, in order to reveal the full definition of *point2* in which base and extra fields are combined:

---

[5] Formally, this is modelled by introducing new state variables in the scope of the functional closure representing the derived object. New methods are introduced in this scope.

[6] Formally, this is modelled by a simply-typed function override operator, ⊕, similar to Cook's operator in [CHC90].

```
point2 =
{ attributes
        [ x = 0;  y = 0 ];  selected = false
  methods
        [ x() = { return x };  y() = { return y };
          equal(other) = { return point1.x() = other.x() and point1.y() = other.y() };
          moveTo(newX, newY) = { x := newX;  y := newY;  return point1 } ];
        selected() = { return selected };
        toggle() = { selected := not selected;  return point2 }
}
```

Here, the *point1* base part[7] of *point2* is highlighted by enclosing the relevant sections in brackets [ ]. Please note how *point2* is somewhat schizophrenic in its self-reference - the "inherited" base method *moveTo()* returns the base object, *point1*, whereas the extra method *toggle()* returns *point2*. This outcome is absolutely normal when deriving one recursive object from another [CCHO89a].

We may create the corresponding *HotPoint* type in a similar fashion:

```
define type HotPoint = Point with
{ protocols
        selected() Boolean;  toggle() : HotPoint
}
```

and, expanding the "inherited" part of the type definition, we obtain:

```
HotPoint =
{ protocols
        x() : Integer;  y() : Integer;  equal(Point) : Boolean;
        moveTo(Integer, Integer) : Point;  selected() : Boolean;  toggle() : HotPoint }
```

The schizophrenia of *point2* is fairly reflected in its type signature, *HotPoint*. Whereas you might expect the methods of a *HotPoint* always to deal in *HotPoints*, it is clear that the "inherited" *equal()* and *moveTo()* methods only deal in *Points*.

---

[7] Technically, the *x* and *y* attributes are scoped only within the closure *point1*, hence the need to go through the public interface using *x()* and *y()* in the *point2* object. We redefine *equal()* later, to test the *selected* field also.

This model now captures the statically-bound aspects of C++. We can demonstrate that *HotPoint* is, in its current form, a proper subtype of *Point*. To recap on subtyping rules [CaWe85]:

- An object type B is a subtype of an object type A if and only if:
  - type B has the same number of method protocols, or strictly more protocols, than type A; and
  - any replacement methods defined in type B have the same protocols, or have protocols which are proper subtypes of their counterparts in A;
- A method protocol Q is a subtype of a method protocol P if and only if the methods have the same number of arguments and:
  - each argument of Q has the same type as, or has a proper **supertype** of, the corresponding argument in P (the contravariance rule); and
  - the result of Q has the same type as, or has a proper **subtype** of, the result of P (the covariance rule).

HotPoint is a proper subtype of *Point*, because it merely adds to the protocols of *Point* and does not replace any methods (the method protocol rules are not needed in this case).

Unfortunately, an object-oriented type system based on subtyping is rather less useful than popular opinion would have you believe. This is because subtyping provides *too weak* a type constraint for early (static) type checking. Consider the following expression:

```
result : Boolean;
result := point2.moveTo(4, 7).selected();       -- does not typecheck!
```

which does not typecheck, because *moveTo()* returns a *Point* object and *selected()* is not defined for this type. You may confirm this by observing that the implementation of *moveTo()* does in fact return *point1*, the embedded base object, rather than *point2*. This illustrates the phenomenon of *type loss* in experienced by programmers in C++, and is the reason why many adopt the unsafe practice [Meys92] of *type downcasting* to recover this lost type information:

```
result := HotPoint(point2.moveTo(4, 7)).selected();
```

You will no doubt agree that it is a poor state of affairs to have to keep reminding a programming language what kind of object it is really dealing with. Let us assume now that a *point3* has been defined exactly as *point2*, except for the fact that its *selected* attribute is initialised with the value *true*, instead of *false*. Consider the further expression:

```
result : Boolean;
result := point2.equal(point3);          -- result is unintendedly true!
```

which returns an unexpectedly *true* result, as a consequence of the "inherited" *equal()* method only considering the *x()* and *y()* parts of the two *HotPoint* objects, but not their *selected()* parts.

In order to address these problems, the typical strategy adopted in many languages is to redefine the *moveTo()* and *equal()* methods in *point2* to behave in the desired manner:

```
define object point2 = point1 with
{ attributes
        selected = false;
  methods
        equal(other) = { return point2.x() = other.x() and point2.y() = other.y()
                        and point2.selected() = other.selected() };
        moveTo(newX, newY) = { x := newX;  y := newY;  return point2 };
        selected() = { return selected };
        toggle() = { selected := not selected;  return point2 }
}
```

The *with*-operator's preference for the fields of its right-hand operand will ensure that *point2* obtains the two redefined methods *moveTo()* and *equal()*, rather than the original implementations. All the problems of schizophrenic self-reference in *point2* have now been resolved; this is reflected in the fact that the revised *HotPoint* type definition deals exclusively in *HotPoints*:

```
HotPoint =              -- expansion of a revised 'define type' expression
{ protocols
        x() : Integer;  y() : Integer;  equal(HotPoint) : Boolean;
        moveTo(Integer, Integer) : HotPoint;  selected() : Boolean;  toggle() : HotPoint }
```

Unfortunately, this *HotPoint* type is not a subtype of *Point*, because it violates the contravariant rule on the *equal()* method protocol - it replaces the *Point* argument of *equal()* with a more specific

HotPoint argument, rather than a more general type as the rule requires. This prevents *HotPoint* from being considered a subtype of *Point*. It is not type-safe to pass a *HotPoint* instance to a *Point* variable; the consequences of doing this are severe in the context of dynamic binding, as observed in languages like Smalltalk, Objective C and Eiffel [Cook89]:

```
var : Point;
var := point2;          -- OK if HotPoint is a subtype of Point (but it isn't)
var.equal(point1);      -- statically type correct, but dynamically fatal
```

because the *equal()* method selected dynamically (*HotPoint's* version), while appropriate for *point2*, is inappropriate for *point1*, since it asks for its non-existent *selected()* field. (C++ would still bind this call statically, according to its overloading rules, treating *point2* as though it were a *Point*).

Now, the focus of this argument is not to insist that all object-oriented languages should obey subtyping rules. As we saw above, this severely limits the expressive power of the type system. Rather, the argument demonstrates that the kinds of object used *routinely* in object-oriented systems have types which *routinely* do not obey simple subtyping. The polymorphic type-compatibility expressed in the class-subclass relationship is not a type-subtype relationship, but something more subtle. The mathematics of simple subtyping is inappropriate for this type-checking task.

4. Class Interfaces and Implementations

The crux of the matter is that both *Point* and (the revised) *HotPoint* have methods which are closed over their own types: we want *point1* to own methods that deal in *Points*, and *point2* to own similar methods that deal in *HotPoints*. This is not just a peculiarity of these examples, but a general phenomenon. Consider that many object-oriented languages have an abstract *Number* type, which defines the protocols for all the arithmetical operations (usually, deferred), which are closed over *Number* arguments and results. Then, "compatible" subclass types are defined, which implement different versions of *plus(), minus(), ...* which are closed over arguments having *Integer, Real, Complex, ...* types. These protocols will always violate contravariance, such that *Integer, Real, Complex, ...* can never be subtypes of *Number*. In general, it is impossible to create a closed, recursive type that is a proper subtype of another recursive type.

Intuitively, we would like to think of *Integer* as a "subclass" of the *Number* "class" (pending a proper formalisation of these notions), because it offers the same kind of interface - the operations are the same, even though their protocols are not subtypes. We can capture the notion of an *interface* by a type function, which differs from a recursive type in that the self-type is not pre-determined, but supplied as a formal argument, *MyType*. An interface for a *Point* may be defined as:

```
define interface PointInterface (MyType) =
{ protocols
        x() : Integer;  y() : Integer;  equal(MyType) : Boolean;
                moveTo(Integer, Integer) : MyType
}
```

This kind of interface can be adapted to different types; for example, if the interface function is applied to the *HotPoint* type, the parameter *MyType* is replaced throughout by *HotPoint*:

```
PointInterface(HotPoint) =              -- supply type argument HotPoint
{ protocols
        x () : Integer;  y () : Integer;  equal(HotPoint) : Boolean;
                moveTo(Integer, Integer) : HotPoint
}
```

generating a retyped version of the *Point* interface for a *HotPoint* object. This suggests that the notion of *class* has something to do with parameterised type interfaces, rather than with simple types *per se* [CCHO89b]. Furthermore, this mathematical notion of *interface* is exactly what is required to support the OMG's use of the term in its 1991 definition [OMG91].

The same principle may be used to define functions representing flexible object *implementations*, in which self-reference does not yet refer to any particular object:

```
define implementation PointImplementation (self) =
{ attributes
        x = 0;  y = 0
  methods
        x() = { return x };  y() = { return y };
        equal(other) = { return self.x() = other.x() and self.y() = other.y() };
        moveTo(newX, newY) = { x := newX;  y := newY;  return self }
}
```

This *PointImplementation* function is the pattern for all objects with state and methods like *point1*, except that recursive references in the methods are parameterised by the formal argument, *self*. By applying *PointImplementation* to any particular object, we may redirect all self-reference in these method definitions to refer to that object (see below; also section 5).

By combining the two notions of the flexible *interface* and flexible *implementation*, we create exactly the notion of a *class definition* (2). This is a typed function in which both *self* and its associated type, *MyType*, are parameters[8], formal arguments to the function:

```
define class PointClass (self : MyType) =
{ attributes
        x : Integer = 0;  y : Integer = 0
  methods
        x() : Integer = { return x };  y() : Integer = { return y };
        equal(other : MyType) : Boolean = { return self.x() = other.x()
                                        and self.y() = other.y() };
        moveTo(newX, newY : Integer) : MyType = { x := newX;  y := newY;  return self }
}
```

This class definition represents the minimum interface and implementation requirements for any object, and its associated type, belonging to the *Point* class. It should be clear by now that interface and implementation are closely linked: this is as true with classes as it is with concrete types. Naturally, it is possible to provide alternative flexible *implementations* that exhibit the same flexible *interface*, but this is not especially what characterises a class apart from a concrete type, which also may have multiple implementations. The only salient difference between this class definition and our earlier definition of a recursive typed object is that *self* and *MyType* are not bound in a class, whereas they are bound in a typed object, to refer back to the object and its type.

The difference between a (concrete) class and a (concrete) type is illustrated in figure 1. A class encompasses a family of different recursive types which, although they are not related by subtyping,

---

[8] In the second-order typed λ-calculus, it is normal practice to define the type function first, followed by the typed object function in which *MyType* is introduced before *self* [CHC90].

have *at least* the interface and implementation described by the class. In the figure, a class is denoted by a cone describing a bounded closed volume, whereas a type is the point at the apex of a cone. This illustrates, for example, that the *Point* type is the least elaborate type which matches the interface of the *Point* class (although, as we shall see in section 5 below, the more elaborate *HotPoint* type also matches the *Point* class interface).

We may relate an *interface* to a *type* in our model language by defining *Point* as the type which fixes[9] the Point class interface, such that all occurrences of *MyType* now refer to *Point*:

define type Point = PointInterface(Point)

The application of the interface function: *PointInterface(Point)* propagates the type argument *Point* into the body of the interface function, yielding exactly the body of a recursive *Point* type. (We take the liberty of allowing this form of recursive definition, which is really no different from how we defined *Point* in the first place). We may also relate an *implementation* to an *object*, by fixing the implementation such that all occurrences of *self* now refer to the desired object:

define object point1 = PointImplementation(point1)

So, in the same way that a *type* fixes an *interface*, an *object* fixes an *implementation*. An *object* has a *type*, which is fixed, but a *class implementation* has an *interface*, both of which are flexible.

5. Class Inheritance and Subclassing

The flexibility with *self* and *MyType* allows us to extend classes without the normal schizophrenia associated with extending concrete recursive types. In Smalltalk, Eiffel, Objective C and dynamically bound aspects of C++, authentic object-oriented inheritance involves not just embedding a base object in an derived object, but redirecting self-reference in the base object's methods to refer to the derived object. Formally, we model this by adapting *interfaces* and *implementations* [CHC90]. A subclass has an extended implementation:

---

[9] In λ-calculus, this is known as taking the *least fixed point* of a generator, using the fixpoint combinator **Y**.

```
define implementation HotPointImplementation (newSelf)
        = PointImplementation (newSelf) with
{ attributes
        selected = false;
  methods
        equal(other) = { return newSelf.x() = other.x() and newSelf.y() = other.y()
                                and newSelf.selected() = other.selected() };
        selected() = { return selected };
        toggle() = { selected := not selected;  return newSelf }
}
```

in which the base part is created by the evaluation of: *PointImplementation (newSelf).* Examine this to see how the argument variable *newSelf* is propagated into the body of *PointImplementation,* replacing all occurrences of *self* by *newSelf.* We therefore no longer need to redefine *moveTo(),* although we still redefine *equal()* for the sake of the extra field-comparison. After the internal evaluation of the *with*-operator, self-reference in all of the methods of *HotPointImplementation* is in terms of *newSelf,* demonstrating that this model of inheritance eliminates schizophrenia.

A subclass also has an extended interface:

```
define interface HotPointInterface (NewType) =
        PointInterface (NewType) with
{ protocols
        equal (NewType) : Boolean;  selected() : Boolean;  toggle() : NewType
}
```

in which: *PointInterface (NewType)* adapts the types of the inherited method protocols, such that all occurrences of *MyType* in the body of *PointInterface* are now replaced by *NewType*. This ability to adapt the self-type captures the: *like Current* type-anchoring mechanism in Eiffel [Meye92].

Determining the type-compatibility of a class interface with any given superclass is a more subtle matter than simple subtyping, because the self-type is not fixed, rather it is a parameter. The subclassing rule must therefore ensure that, *whatever* the self-type, the interface generated by a subclass is larger than that of the corresponding superclass [CCHO89b, CHC90]:

- A class interface B is a subclass of a class interface A if and only if, for **any self-type** T, B(T) yields the same protocols, or a larger set of protocols, than A(T) .

We do not need to appeal to subtyping among individual method protocols in this simple rule, since the adapted protocols either match exactly, or do not. We can illustrate this with:

HotPointInterface (T) = { **protocols** x() : Integer; y() : Integer; equal(T) : Boolean;
moveTo(Integer, Integer) : T; selected() : Boolean; toggle() : T }

PointInterface (T) = { **protocols** x() : Integer; y() : Integer; equal(T) : Boolean;
moveTo(Integer, Integer) : T }

demonstrating that the *HotPointInterface* describes a subclass of *PointInterface*, since the former substitution with an arbitrary self-type T yields a larger set of protocols than the latter substitution (and all the common protocols are pairwise identical).

In section 4 above, we asserted that objects of many different recursive types could belong to the same class. The simple rule governing class membership is subtle, and allows this.

- An object having (self-) type T belongs to a class with interface A if the protocols T are the same, or a larger set than A(T).

So, the object *point2* which has the type *HotPoint*, may nonetheless be considered a member of the *Point* class, because *HotPoint* has a larger set of protocols than *PointInterface (HotPoint):*

HotPoint = { **protocols** x() : Integer; y() : Integer; equal(HotPoint) : Boolean;
moveTo(Integer, Integer) : HotPoint; selected() : Boolean; toggle() : HotPoint }

PointInterface (HotPoint) = { **protocols** x() : Integer; y() : Integer;
equal(HotPoint) : Boolean; moveTo(Integer, Integer) : HotPoint }

Similar principles apply when considering implementations. A *HotPointImplementation* must provide at least all the methods of a *PointImplementation*, in which occurrences of *self* are replaced by *newSelf*; however, methods may also be replaced. A *point2* object satisfies the requirements of a *PointImplementation* if its own attributes and methods are a larger set than the result of applying the function: *PointImplementation(point2)* [Simo95].

6. Implications for Future Standards

We have situated the four notions of *object, type, class interface,* and *class implementation* with respect to each other. The primary notion of a *class definition* (2) was developed quite naturally and encompasses both specification and implementation aspects - the two are inextricably linked. The specification aspect of a class is commonly known as an *interface* [OMG91], formally a type function of *MyType*, the self-type recursion variable. The related *implementation* aspect of a class is a function of *self*, the object recursion variable. The primary notion of a *class family* (1) was also implicitly defined, through the development of *subclassing* and *class membership* rules. We emphasise the fact that a *class family* (1) may contain objects that are instances of different concrete, recursive types, providing that each type satisfies the class interface and each object satisfies the chosen class implementation. By the reverse argument, an object, though it has exactly one type upon creation, may be considered a member of more than one class (1), specifically of all those superclasses which include its fixed concrete type within their bounds (cf figure 1).

Part of our aim has been to correct popular misconceptions regarding *class*. It is, in general, not correct to assert that a class identifier has the status of a type, except perhaps in the languages Trellis [SCBK86], Modula-3 [CDJG89] and the statically bound parts of C++, which have modest type systems based on subtyping. In a majority of object-oriented languages and contexts, a class is not a type and subclassing is not subtyping, not simply because the language's type rules may happen to violate subtyping, but more fundamentally because self-reference is *routinely redirected* during inheritance. Neither is it correct to assert that a class merely realises, or implements a type [UML97, OML97]. The observation that a derived class implementation does not have a corresponding subtype signature should not compel one to think that a class only serves an implementation purpose. A subclass typically extends a recursive structure, and there are *in any case* no proper subtypes of closed recursive types. Instead, a class is an inherently more flexible formal construct.

Our chief aim has been to establish a solid foundation for the popular, historic notion of *class*, and for the relationships between *class, type* and *object*, with a view to informing present standardisation efforts. We have defined precisely the difference between *derivation*, an operation on simply-typed objects, and *subclassing*, an operation on polymorphically-typed classes. The relationship between

an object and an adaptable implementation is similar to that between a type and an adaptable interface, in that they both fix a flexible structure at a given level of specialisation, by binding recursion variables. Figure 2 illustrates some of the relationships which should form part of a future standard. In particular, we want to emphasise that:

- an *object* has an associated *type*;
- a class *implementation* has an associated *interface*;
- an *object* conforms to many, and fixes one, class *implementation*;
- a *type* conforms to many, and fixes one, class *interface*;
- a *class definition* encompasses an *interface* and an *implementation*;
- a *class family* contains objects having different, structurally related types, but which are unrelated by subtyping;
- *derivation* produces objects having subtypes and schizophrenic self-reference;
- *inheritance* produces classes with subclass interfaces and uniform self-reference.

The difference between a class and a typed object is explained formally in terms of generator functions and fixpoints in the λ-calculus. We have tried to present these notions without appealing self-consciously to the λ-calculus foundations [CCHO89a, CoPa89, CCHO89b, CHC90, Simo95, Simo96], for the sake of the average object technologist. To that end, we developed a user-friendly formal language for reasoning about objects, types and classes, which bears more resemblance, in its syntax, to a programming language than to pure mathematics. With all this clear evidence before us, can we please, finally, agree on the meaning of *class*?

[Figures and references attached on following supplementary pages]

Figures

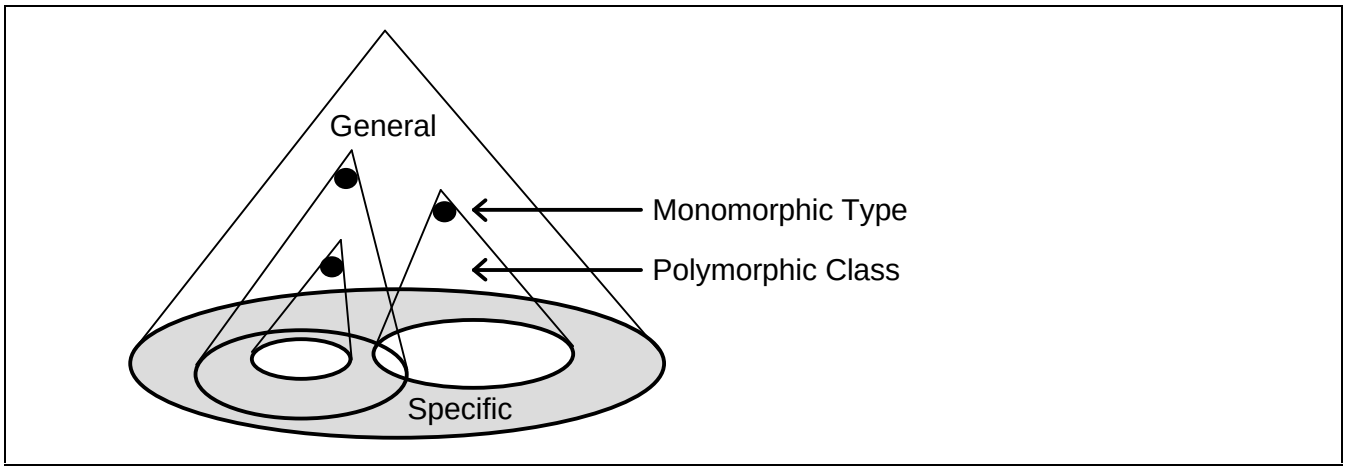

Figure 1: Relationship between Classes and Types

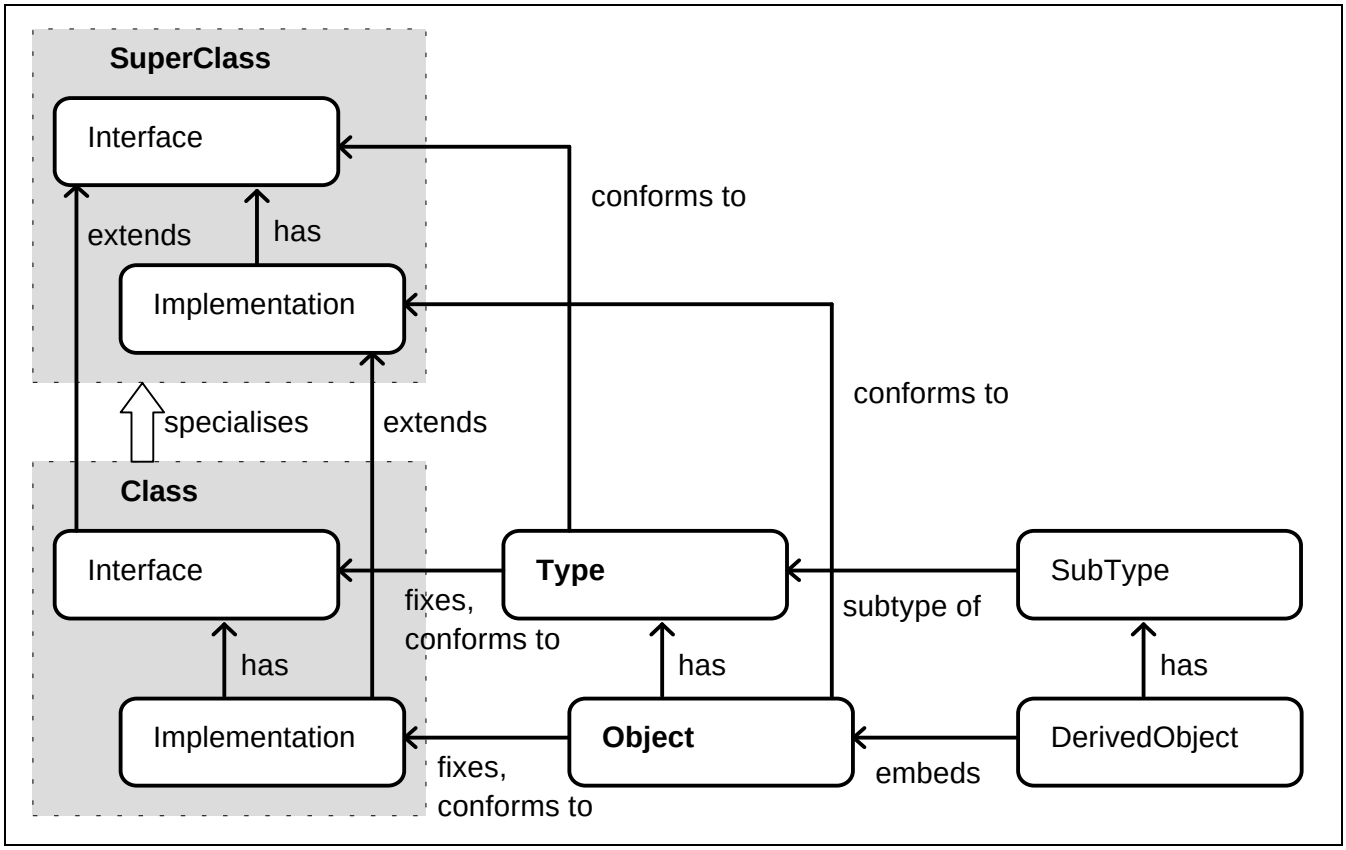

Figure 2: Standard Modeling Concepts and Relationships

References

[Amer90] P America (1990), 'Designing an object-oriented language with behavioural subtyping', *Proc. Conf. Foundations of Object-Oriented Lang.*, 60-90.

[CaWe85] L Cardelli and P Wegner (1985), 'On understanding types, data abstraction and polymorphism', *ACM Computing Surveys, 17(4)*, 471-521.

[CCHO89a] P Canning, W Cook, W Hill, W Olthoff and J Mitchell (1989), 'F-bounded polymorphism for object-oriented programming', *Proc. 4th Int. Conf. Func. Prog. Lang. and Arch.*, Imperial, London, September, 273-280.

[CCHO89b] P Canning, W Cook, W Hill and W Olthoff (1989), 'Interfaces for strongly-typed object-oriented programming', *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang and Appl.*, 457-467.

[CDJG89] L Cardelli, J Donahue, L Glassman, M Jordan, B Kalsow and G Nelson (1989), 'Modula-3 report (revised)', *Tech. Rep. 52*, Digital Equipment Corporation Systems Research Centre.

[CHC90] W Cook, W Hill and P Canning (1990), 'Inheritance is not subtyping', *Proc. 17th ACM Symp. Principles of Prog. Lang.*, 125-135.

[CoNo91] B J Cox and A J Novobilski (1991), *Object-Oriented Programming: an Evolutionary Approach, 2nd Ed.*, Addison-Wesley.

[CoPa89] W Cook and J Palsberg (1989), 'A denotational semantics of inheritance and its correctness', *Proc. 4th ACM Conf. Object-Oriented Prog. Sys., Lang and Appl.*, 433-443.

[Cook89] W Cook (1989), 'A proposal for making Eiffel type-safe', *Proc. 3rd Eur. Conf. Object-Oriented Prog.*, 57-70.

[DGLM95] M Day, R Gruber, B Liskov and A Meyers (1995), 'Subtypes vs. where clauses: constraining parametric polymorphism', *Proc. 10th ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, pub ACM Sigplan Notices 30(10), 156-168.

[GoRo83] A Goldberg and D Robson (1983), *Smalltalk-80: the Language and its Implementation*, Addison Wesley.

[IBHK79] J Ichbiah et al. (1979), 'Rationale and design of the programming language Ada', *ACM Sigplan Notices, 14(8).*

[KaWa96] M Campione and K Walrath (1996), *The Java Tutorial: Object-Oriented Programming for the Internet*, Addison Wesley.

[Lisk87] B Liskov (1987), 'Data abstraction and hierarchy', *Addendum to Proc. 2nd ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl.*, 17-34.

[Meye92] B Meyer (1992), *Eiffel: the Language*, Prentice Hall.

[Meys92] S Meyers (1992), *Effective C++: 50 Ways to Improve your Programs and Designs*, Addison-Wesley.

[NeXT93] NeXT Computer Inc. (1993), *NeXTStep Object-Oriented Programming and the Objective C Language*, Addison Wesley.

[OMG91] The Object Management Group (1991), *The Common Object Request Broker: Architecture and Specification, Revision 1.1*, OMG.

[OML97] The OPEN Consortium (1997), *The OPEN Modelling Language, Revision 1.0*, http://www.csse.swin.edu.au/cotar/OPEN/OML.html, OPEN.

[Omoh94] S Omohundro (1994), *The Sather 1.0 Specification*, ICSI Berkley.

[RaLe89] R K Raj and H M Levy (1989), 'A compositional model for software reuse', *Proc. 3rd Eur. Conf. Object-Oriented Prog.*, 3-24.

[Sakk89] M Sakkinen (1989), 'Disciplined inheritance', *Proc. 3rd Eur. Conf. Object-Oriented Prog.*, 39-56.

[SCBK86] C Shaffert, T Cooper, B Bullis, M Killian and C Wilpolt (1986), 'An introduction to Trellis/Owl', *Proc. 1st ACM Conf. Object-Oriented Prog. Sys., Lang. and Appl*, 9-16.

[Simo95] A J H Simons (1995), *A Language with Class*, PhD Thesis, University of Sheffield.

[Simo96] A J H Simons (1996), 'A theory of class', *Proc. 3rd Int. Conf. Obj.-Oriented Info. Sys.*, eds. D Patel, Y Sun and S Patel, Springer Verlag, 44-56.

[Snyd86] A Snyder (1986), 'Encapsulation and inheritance in object-oriented programming languages', *Proc. 1st ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl.*, pub. *ACM Sigplan Notices, 21(11)*, 38-45.

[Stro91] B Stroustrup (1991), *The C++ Programming Language, 2nd Edn.*, Addison-Wesley.

[UML97] Rational Corp. (1997), *Unified Modeling Language, Version 1.0*, http://www.rational.com/ot/uml/1.0/index.html, Rational.

[Wegn87] P Wegner (1987), 'Dimensions of object-based language design', *Proc. 2nd ACM Conf. Object-Oriented Prog., Sys., Lang. and Appl.*, pub *ACM Sigplan Notices, 22(12)*, 168-182.

[Wirt82] N Wirth (1982), *Programming in Modula-2*, Springer Verlag.