Objectivism: "CLASS" considered harmful[1,2]

Jürgen F H Winkler
Institut für Informatik
Friedrich-Schiller-Universität Jena
http://www1.informatik.uni-jena.de

"Classes are difficult to arrange in predefined taxonomies." [CGN 90: 99]
"What constitutes the proper use of inheritance is a widely debated topic." [KM 90: 44]

Object, class and inheritance are the key concepts of object oriented programming (OOP), and in comparison to earlier languages inheritance is the crucial one of these three. Often, inheritance is motivated, introduced, and explained by referring to classification in taxonomic systems [BKM 86: 159:"taxonomy of types"; Boo 91: 132 f.; CY 91: 1; Weg 86], typically taxonomy of concepts [Bra 83: 30; MM 88: "classification of concepts"; Ped 89: 408: "Specialization and generalization are relationships between concepts."; KS 87: 14]. This is done especially in introductions to OOP and in texts dealing with the early phases of the software lifecycle (object oriented analysis (OOA) [CY 91] and object oriented design (OOD) [Boo 91]). Typical examples deal with the classification of biological species: the class of mammals and its subclasses as e.g. persons and elephants [Weg 89, 90] or different kinds of birds [EH 90: 149]. Often this motivation is combined with the idea that OO programs model objects of some section of the real world [Mey 88: 51; MS 88]. The orientation towards classification gave rise to the phrase "S IS-A C" [Mey 88: 230] to characterize the typical inheritance relationship between a subclass S and its superclass C: e.g. an elephant is a mammal (moreover: each elephant is also a mammal). We may call this view of inheritance the concept oriented view (COV) (Wegner uses the phrase "logical hierarchy" [Weg 90: 44]).

On the other hand, sources from the field of OO programming languages usually do not mention such motivations but introduce the concepts of OOP as technical concepts of software technology [DMN 70; ES 90; GR 85; Mey 86: 395: "The fundamental idea of inheritance is that new software elements may be defined as extensions of previously defined ones; ..."; Wir 88: "Type extensions"]. This extension mechanism defines a monotonic relation between superclass and subclass: the is-a subclass is a monotonic extension of its superclass. This property leads to the following substitutability of related objects: the behavior of any program remains unchanged if an object of a class C is replaced with an object of any subclass of C [WZ 88: 65]. One consequence of this principle is the polymorphism of pointer variables, which is very useful for practical programming. We may call this view of inheritance the program oriented view (POV) (Wegner uses the phrase "physical hierarchy" [Weg 90: 44]).

We can therefore state the fact that one key concept of OOP (is-a inheritance) is interpreted quite differently by different groups of the software community. This dichotomy has also been observed by other authors [Abr 91; LP 91; Tes 91; Weg 90]. Often these two views lead to rather different program structures, and especially, the COV may result in programs which are awkward, inefficient, and even incorrect. One example for this dichotomy is the set of classes representing different kinds of numbers from the class library of the Smalltalk system [GR 85: 14]: In this representation indentation indicates the subclass relation. Figure 1 shows the class hierarchy as implemented in the Smalltalk library, and Figure 2 shows the hierarchy of the corresponding abstract concepts, which is quite different.

```
Number
    Float
    Fraction
    Integer
```

Figure 1. POV

```
Complex
   Real
      Rational
         Integer
            Natural
```

Figure 2. COV

Another example are the classes Rectangle and Square [Win 91]. The typical abstract definition in Websters New Collegiate Dictionary:

"square: a rectangle with all four sides equal."

[1] A slightly different version has been published in: *Commun. ACM 35*, 8 (1992), 128-130.
[2] This version is dedicated to Prof. Horst Wettstein on the occasion of the 25th anniversary of his appointment.

leads to the class hierarchy depicted in Figure 3.

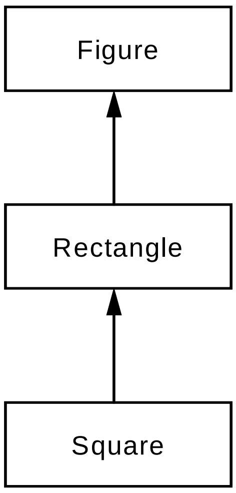

Figure 3. Hierarchy for COV

Figure 4 shows the partially completed ("flattened") specification part of the class `Square`, in which all components inherited directly from `Rectangle` are represented as comments. (The language used is Object CHILL [DW 92].)

```
Square: MODULE CLASS  IS_A Rectangle
   /* SetHeight:  PROC(Height Positive);        */
   /* SetWidth:   PROC(Width Positive);         */
   /* SetSides:   PROC(Height, Width);
                  PRE True;                     */
   /* GiveHeight: PROC()RETURNS (Positive);     */
   /* GiveWidth:  PROC()RETURNS (Positive);     */
   SetSideLength: PROC(SideLength Postive);
   GiveSideLength:PROC()RETURNS (Positive);
   SetSides:      PROC(Height, Width Positive);
                  PRE Height = Width;
   Square:        CONSTR(SideLength Positive);
/****** Begin of internal part ******/
   /* DCL Width, Height Positive;               */
END Square;
```

Figure 4. Partially flattened specification of the Class `Square`

This implementation of the classes `Rectangle` and `Square` has the following three problems:

1) Each `Square` object contains *two* data components for the sidelengths, where *one* such component is sufficient.From an engineering point of view this is not satisfactory.

2) All methods applicable to `Rectangle` objects are also applicable to `Square` objects, e.g.:

```
DCL MySquare  Square(5);
MySquare.SetSideLength(10);
   /* Height = 10 ∧ Width = 10 */
MySquare.SetHeight(20);
   /* Height = 20 ∧ Width = 10 */
   /* Does MySquare represent a square ?? */
```

As we can see, this leads to rather awkward states of `Square` objects.

The reason is that a subclass may not drop anything inherited from its superclass but may only define additional methods and attributes. To define the is-a relation this way is mainly motivated by the request to support polymorphism for references to objects. Polymorphism is very useful for practical programming because it allows the uniform manipulation of objects of different but related subclasses using methods of a common superclass. If the language also supports the subclass test (as e.g. Smalltalk and Object CHILL), the monotonic definition of is-a is even more useful.

3) In order to avoid a call of `SetSides` with different values for the two sides, this method has been redefined in the class `Square` by giving a new precondition (`PRE Height=Width`) for it. Unfortunately, this precondition is stronger than that of `SetSides` in the class `Rectangle` contradicting the substitutability mentioned earlier [Win 91]. With these preconditions the program is wrong.

If such a small and simple example shows that many problems, something must be wrong with the programming methodology (COV) used.

Figure 5 shows an alternative class hierarchy which avoids all three problems as can be seen in the class specification of `Square` in Figure 6. Each `Square` object now contains *one* data attribute for its sidelength and provides only those methods which are typical for squares.

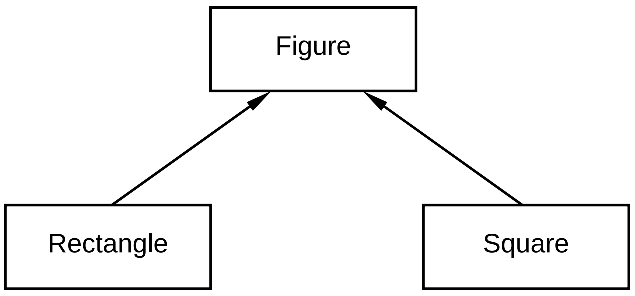

Figure 5. Hierarchy for POV

```
Square: MODULE CLASS  IS_A Figure
   SetSideLength: PROC(SideLength Postive);
   GiveSideLength:PROC()RETURNS (Positive);
   Square:        CONSTR(SideLength Positive);
/****** Begin of internal part ******/
   DCL SideLength Positive;
END Square;
```

Figure 6. Specification of the Class `Square`

When comparing Figure 3 and Figure 5 we observe the same effect as in Figure 1 and Figure 2: two (or more) classes, which are in an is-a relationship in the COV hierarchy, are siblings in the POV hierarchy. What is the reason for this?

The basic reason for the difference between these two class hierarchies is the following: the classes in an implementation are descriptions of particular rectangles or squares with further attributes as e.g. data components and operations, whereas the conceptual is-a relation mentioned above does refer to the abstract properties "rectangleness" (i.e. the property to be a rectangle) and "squareness". For the particular instances the is-a relation does not hold: NOT(square(10) IS-A rectangle(5, 20)), i.e. a square with a sidelength of 10 is not a specialization of a rectangle with height 5 and width 20. Therefore, the difference between COV and POV can, for this example, be characterized by:

Squareness IS-A Rectangleness AND NOT (particular square IS-A particular rectangle).

The dichotomy between the hierarchies for the abstract properties (concepts) and the particular objects having these properties currently leads to difficulties when the OO method is applied in the software lifecycle. In many cases the class hierarchy developed in the early phases will differ significantly from that developed in the implementation phase (i.e. somewhere in the middle of the lifecycle, the program structure must be changed dramatically). This is very undesirable and error prone. Therefore, introductory texts and texts dealing with the early phases should put less stress on conceptual classification and should explain the elements of OOP in a more technical manner.

Since the classes in OO software systems typically describe particular objects rather than abstract concepts, and since the field is labeled OBJECT-oriented programming, it would be a good idea to avoid the term "CLASS" in this field and use e.g. "OBJECT type" instead, because "CLASS" has too strong a connection with the area of taxonomy of concepts. This way has been followed first by Object Pascal [Tes 85: 14] and later by other OO Pascal dialects [Bor 90]. Wirth went even one step further and generalized the record type into an object type [Wir 88] . Therefore, instead of POV we should better speak of OOV (=OBJECT Oriented View). The following table shows the terminologies used by different OO languages, where the terminology of the Pascal dialects occurs in the right-most column:

Coad / Yourdon	Simula	Smalltalk Object CHILL	C++	Object Pascal Turbo Pascal
object	object	object	object	object
class	class (object class)	class	class	object type
specialization class	subclass	subclass	derived class	descendant
generalization class	prefix	superclass	base class	ancestor
inheritance	concatenation	inheritance	inheritance	inheritance

CLASS	OBJECT

Even the terminology of Object Pascal does not seem to capture the essence of the problem in the most lucid way. Therefore I propose the following terminology for the elements of object oriented programming :

object
object type
extension type
base type
inheritance

The whole discussion can be summed up by the following equation:

$$\text{inheritance} = \text{extension} \quad \text{AND} \quad \text{inheritance} \neq \text{specialization}$$

Acknowledgment

I am very grateful to G.Dießl, W.Günther, H.-J. Hoffmann, G. Kappel, K. Lawler, P. Sommerlad, and W.Weber for very helpful remarks on an earlier draft of the paper.

References

Abr 91 Abrahams, P. W.: Subject: Objectivism. Letter to ACM Forum. *Commun. ACM* 34,1 (1991) 15-16.

BKM 86 Mittal, S.; Bobrow, D. G.; Kahn, K. M.: Virtual Copies - At the Boundary Between Classes and Instances. OOPSLA'86. *SIGPLAN Not*. 21, 11(1986), 159-166.

Boo 91 Booch, G.: *Object Oriented Design With Applications*. Benjamin/Cummings, Redwood City, 1991.

Bor 90 Borland International: Turbo Pascal. Version 6.0. Programmer's Guide. ScottsValley, 1990.

Bra 83 Brachman, R. J.: What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *Comput*. 16, 10 (1983) 30-36.

CGN 90 Gibbs, S.; Tsischritzis, D.; Casais, E.; Nierstrasz, O.; Pintado, X.: Class Management for Software Communities. *Commun. ACM 33*,9 (1990) 90-103.

CY 91 Coad, P. and Yourdon, E.: *Object-Oriented Analysis*. Prentice Hall, Englewood Cliffs, N.J., 1991.

DMN 70 Dahl, O.; Myhrhaug, B.; Nygaard, K.: *Common Base Language*. Norwegian Computing Center, 1970.

DW 92 Winkler, J. F.H.; Dießl, G.: Object CHILL - An Object Oriented Language for Systems Implementation. ACM Computer Science Conference 1992,.139-147.

EH 90 Henderson-Sellers, B.; Edwards, J. M.: The Object-Oriented Systems Life Cycle. *Commun. ACM* 33,9 (1990) 142-159.

ES 90 Ellis, M. A.; Stroustrup, B.: *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, Mass., 1990.

GR 85 Goldberg, A.; Robson, D.: *Smalltalk-80*. Addison Wesley, Reading, Mass., 1985.

KM 90 Korson, T.; McGregor, J. D.: Understanding Object-Oriented: A Unifying Paradigm. *Commun. ACM* 33,9 (1990) 40-60.

KS 87 Kempf, R.; Stelzner, M.: Teaching Object-Oriented Programming with the KEE System. OOPSLA'87. *SIGPLAN Not.* 22, 12 (1987), 11-25.

LP 91 LaLonde, W.; Pugh, J.: Subclassing ≠ subtyping ≠ Is-a. *J. Obj. Orien. Prog.* 3,5 (1991), 57-62.

Mey 86 Meyer, B.: Genericity versus Inheritance. OOPSLA'86, *SIGPLAN Not*. 21, 11 (1986), 391-405.

Mey 88 Meyer, Bertrand: *Object-oriented Software Construction*. Prentice Hall, Englewood Cliffs, N.J., 1988.

MM 88 Madsen, O.; Møller-Pedersen, B.: What object-oriented programming may be - and what it does not have to be. In: S. Gjessing, K. Nygaard, Eds.: ECOOP '88, Springer, Berlin, 1988, 1-20.

MS 88 Shlaer, S.; Mellor, S.J.: *Object-Oriented Systems Analysis*. Yourdon Press, Englewood Cliffs, N.J., 1988.

Ped 89 Pedersen, C. H.: Extending Ordinary Inheritance Schemes to Include Generalization. OOPSLA'89 *SIG-PLAN Not.* 24, 10 (1989), 407-417.

Tes 85 Tesler, L.: OBJECT PASCAL - Preliminary Documentation. Structured Language World 9,3 (1985), 10-17.

Tes 91 Tesler, L. G.: Object-Oriented Approach. *Commun. ACM* 34,8 (1991), 13-14.

Weg 86 Wegner, P.: Classification in Object-Oriented Systems. P. Wegner, B. Shriver, Eds.: Object-Oriented Programming Workshop *SIGPLAN Not. 21*,10 (1986), 173-182.

Weg 89 Wegner, P.: Learning the Language. *BYTE* March 1989, 245-253.

Weg 90 Wegner, P.: Concepts and Paradigms of Object-Oriented Programming. *OOPS Mess. 1*,1 (1990), 7-87.

Win 91 Winkler, J. F. H.: Square IS-A Rectangle ? or IS-A Inheritance in OOP and OOA/D. Internal paper. Siemens, Corporate Research and Development, 1991.

Wir 88 Wirth, N.: Type Extensions. *ACM TOPLAS, 10*, 2 (1988), 204-214.

WZ 88 Wegner, P.; Zdonik, B.: Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like. S. Gjessing, K. Nygaard, Eds.: ECOOP '88, Springer, Berlin, 1988, 55-77.