Classes vs. Prototypes
Some Philosophical and Historical Observations

Antero Taivalsaari
Nokia Research Center
P.O. Box 45, 00211 Helsinki
FINLAND
taivalsa@research.nokia.com

April 22, 1996

1. Introduction

> "Objects in the real world have only one thing in common
> -- they are all different."

In the recent years an alternative to the traditional class-based object-oriented language model has emerged. In this *prototype-based paradigm* [Bor86, Lie86, LTP86, Ste87, UnS87, SLU88, DMC92, Bla94] there are no classes. Rather, new kinds of objects are formed more directly by composing concrete, full-fledged objects, which are often referred to as *prototypes*. When compared to class-based languages, prototype-based languages are conceptually simpler, and have many other characteristics that make them suitable especially to the development of evolving, exploratory and distributed software systems.

The distinction between class-based and prototype-based systems reflects a long-lasting philosophical dispute concerning the representation of abstractions. Plato viewed forms -- stable, abstract, "ideal" descriptions of things -- as having an existence more real than instances of those things in the real world. Class-based languages such as Smalltalk, C++ or Simula are Platonic in their explicit use of classes to represent similarity among collections of objects. Prototype-based systems such as Self [UnS87], Omega [Bla91, Bla94], Kevo [Tai92, Tai93], GlyphicScript [Gly94] and NewtonScript [SLS94] represent another view of the world, in which one does not rely so much on advance categorization and classification, but rather tries to make the concepts in the problem domain as tangible and intuitive as possible. A typical argument in favor of prototypes is that people seem to be a lot better at dealing with specific examples first, then generalizing from them, than they are at absorbing general abstract principles first and later applying them in particular cases.

Prototypes give rise to a broad spectrum of interesting technical, conceptual and philosophical issues. In this paper we take a rather unusual, non-technical approach and investigate object-oriented programming and the prototype-based programming field from a purely philosophical viewpoint. Some historical facts and observations pertaining to objects and prototypes are presented, and conclusions based on those observations are derived.

2. Classes and classification -- early history

The central concepts behind object-oriented programming -- classes, instances and classification -- have been of interest to human beings for centuries. The earliest characterization of classes (types) versus instances was given by *Plato* over two thousand years ago [Plato]. Plato made a clear distinction between *forms* – i.e., stable, immutable, "ideal" descriptions of things – and particular *instances* of these forms. Plato regarded the world of ideas as much more important than the world of instances, and contended that forms always have an existence that is more real than the concrete entities and beings in the real world [Plato].

Research into classification (to be precise: biological classification) was continued by Plato's student *Aristotle* (384-322 BC) who had an endless interest in understanding and organizing the world to its smallest details. Whereas Plato was interested mainly in ideas and "eternal" concepts, Aristotle was the first philosopher interested especially in natural phenomena. In his works -- over 170 in total -- Aristotle aimed at providing a comprehensive, detailed taxonomy of all natural things – plants, animals, minerals, and so on [Aristo]. His classifications were based on the same idea that underlies object-oriented programming today. A group of objects belongs to the same category if the objects have the same properties. Thus, categories of objects are defined by *common properties* that a group of objects (the extension of the category) share. New categories can be defined in terms of other categories if the new categories have at least the same properties as the defining ("genus") categories. The general rule for classification can be presented as follows:

$$essence = genus + differentia$$

In other words, categories are defined in terms of their *defining* properties and *distinguishing* properties. This corresponds precisely to the idea behind traditional class-based object-oriented programming in which a class is defined in terms of its superclass (genus) and a set of additional variables and methods (differentia).

Work of Aristotle has lead to the common idea, at least in the West and in many other cultures, that there is a single correct taxonomy of natural things – animals, plants, minerals, and so on. Unfortunately, the level of categorization depends heavily who is doing the categorizing and on what basis. In practice, people have many ways of making sense of things – and taxonomies of all sorts abound. Yet the idea that there is a single universal taxonomy of natural things is remarkably persistent [Lak87].

(Aside: Aristotle realized himself that his model has problems and noted that many objects have "accidental" properties, i.e., properties that are characteristic of the object under examination but atypical for those kinds of objects in general. Thus, the actual substance of concepts was defined in terms of two aspects: the *essence* and the *accidents*. This dichotomy has later inspired many researchers, including Fred Brooks [Bro86].)

3. Later history: a criticism of classification

Aristotle's work on classification did not receive much criticism for a long time. Categories were regarded as well-understood and unproblematic. They were assumed to be abstract containers, with things either inside or outside the category. The idea that categories of things are defined by common properties is not only our everyday folk theory of what a category is, but it is also the principal technical theory – one that has been with us for more than two thousand years [Lak87]. Aristotle's ideas have later stimulated the work of many researchers, including, e.g., the famous Scandinavian natural scientist Carl von Linné.

The Aristotelian "classical" view was first challenged in the 19th century by the famous British philosophers *W. Whewell* and *W.S. Jevons*. They emphasized that there are no universal rules to determine what properties to use as a basis of classification of objects. Furthermore, they argued that classification is not a mechanical process but it requires creative invention and evaluation. Consequently, Whewell and Jevons argued that there are no objectively "right" classifications. In light of this view, the task of constructing general rules for classification seems rather complicated.

Criticism on classification was continued later in our century by *Ludwig Wittgenstein* [Wit53]. Wittgenstein observed that it is difficult to say in advance exactly what characteristics are essential for a concept. Wittgenstein gave several examples of seemingly simple concepts that are extremely difficult to define in terms of shared properties. A classical example is the concept of 'game' [Wit53 §66-67]. Some games involve mere amusement, like ring-around-the-rosy. In that game there is no competition -- no winning or losing -- though in other games there is. Some games involve luck, like board games where a throw of the dice or a draw from a card deck determines the next move. Others, like chess or water polo, involve skill. Still others, like poker or monopoly, involve both luck and skill to varying degrees. The number of players may also vary considerably from one, as in solitaire, to hundreds, thousands or even millions, as in lottery or horserace betting. (There are also games in which no players are needed at all, but many people do not regard those as "real" games.)

Another example of a concept that is hard to define in terms of shared properties is "work of art". Since no one can really define clear boundaries for what is art and what isn't, there is no general class "work of art" with shared, common properties. The definition is subjective and depends heavily on the situation or viewpoint. (Aside: the famous Russian writer Tolstoy once made it a criterion of value for a work of art that it should be intelligible to everybody: "The significance of an object lies in its universal intelligibility").

After presenting a criticism of the classical model, Wittgenstein then defined what can be seen as the origin of prototype-based programming: the notion of a "family resemblance". Games do not have any shared, common defining characteristics. Instead, games share a sort of *family resemblance*: Baseball is a game because it resembles the family of activities that people call games. Members of a family resemble each another in various ways: they may share the same build or the same

facial features, the same hair color, eye color, or temperament, or the like. But there need be no single collection of properties shared by everyone in a family [Lak87 p.16]. Except for technical terms in mathematics, Wittgenstein maintained that for most of the concepts, meaning is determined not by definition, but by family resemblances. Such terms can be defined only in terms of similarity and representative "prototypes".

4. Towards the prototype theory

Wittgenstein's results have sparked research into so called *prototype theory*. J.L. Austin [Aus61], L. Zadeh [Zad65] and F. Lounsbury, among many others, have studied the area. But it was *Eleanor Rosch* who introduced the prototype theory in the mid-1970s [RoM75, RMG76]. Rosch observed that studies by herself and others demonstrated that categories, in general, have *best examples* (called "prototypes") and that all of the specifically human senses play a role in categorization. Thanks to the pioneering work of Rosch, categorization has become a major field of study within cognitive psychology.

In her criticism of the classical approach, Rosch focused on two implications of the classical theory [Lak87]:

- First, if categories are defined only by properties that all members share, then no members should be better examples of the category than any other members.

- Second, if categories are defined only by properties inherent in the members, then categories should be independent of the peculiarities of any beings doing the categorizing; that is, they should not involve such matters as human neurophysiology, human body movement, and specific human capacities to perceive, to form mental images, to learn and remember, to organize the things learned, and to communicate efficiently.

It can be shown relatively easily that the above mentioned implications are not typically true when people do classification. For instance, the fact that some instances are "better" representatives of categories than others can be confirmed simply by asking people to give examples of 'numbers'. Typically people respond with relatively simple integers like one, two, five or forty-two rather than -127.798432, 0x12ABFF4C or 12.5-5i, although in principle real, complex, hexadecimal, or transfinite numbers would be equally good examples of numbers. Thus, integers (and small integers in particular) are, in a sense, better examples than other kinds of numbers.

Also, it can be proven rather easily than our background, mental capabilities and experience play a significant role in the classification process. For instance, some people living near the equator are claimed to be unable to distinguish between snow and ice, whereas the Eskimos have numerous words for describing different types of snow. Dani people of New Guinea have only two basic color terms: mili (dark-cool)

and mola (light-warm) that cover the entire spectrum, and have great difficulties in differentiating between colors in more detail [Lak87]. A professional limnologist might be able to identify several hundreds or even thousands of different animals living in the water, whereas a layman might recognize only a few dozen. Also, classifications by persons who have substantial expertise in a certain area are typically much more refined than those created by non-experienced people (conversely, people with little expertise easily make mistakes such as classifying whales and dolphins as fish, and so on).

In general, cognitive observations such as those above revealed some inherent flaws in the traditional classical model, and formed the basis for research leading into the prototype theory presented by Rosch and others. The essential results of prototype theory leading up to the cognitive models approach can be summarized as follows [Lak87 p.56]:

- Some categories, like tall man or red, are graded; that is, they have inherent degrees of membership, fuzzy boundaries, and central members whose degree of membership (on a scale from zero to one) is one.

- Other categories, like bird, have clear boundaries; but within those boundaries there are graded prototype effects – some category members are better examples of the category than others.

- Categories are not organized just in terms of simple taxonomic hierarchies. Instead, categories "in the middle" of a hierarchy are the most basic, relative to a variety of psychological criteria. Most knowledge is organized at this level.

- The basic level depends upon perceived part-whole structure and corresponding knowledge about how the parts function relative to the whole.

- Categories are organized into systems with contrasting elements.

- Human categories are not objectively "in the world", external to human beings. Many categories are embodied, and defined jointly by the external physical world, human biology, the human mind, plus cultural considerations.

It has also been shown that in many situations people perform classification on an almost totally ad hoc basis, creating unconventional and previously non-existing structures on the fly for some immediate purpose. Examples of such categories include

- what to get for a birthday present,
- what to do for entertainment on a weekend, and
- things to be taken from one's home during a fire.

For a detailed discussion on the cognitive and other observations and experiments that have lead to the development of the prototype theory, the reader is referred to the excellent book "Women, fire, and dangerous things" by George Lakoff [Lak87].

5. Implications on programming languages and methods

Albeit rather philosophical, the discussion above has some important implications on the world of programming. In this section we present some thoughts and consequences that the theories of classification have on programming languages and software development methods. Note that classification has been studied rather actively in the field of artificial intelligence (see, e.g. [Bra83] and [Tou86]), but surprisingly, many object-oriented software designers seem to be almost completely unaware of the conceptual and philosophical background that underlies object-oriented programming.

Recognizing the limited modeling capabilities of OO. As already mentioned earlier in the paper, the programming models used in most object-oriented programming languages today are surprisingly similar to the Aristotelian classical model of the world. For instance, object-oriented languages typically assume that new classes are defined in terms of shared properties, and that instances of a class always have an identical set of properties. Furthermore, the class inheritance model used in most object-oriented languages closely resembles the Aristotelian way of defining new classes (categories) in terms of existing genealogical parents. The inheritance hierarchies characteristic of object-oriented programs also bear an intimate resemblance to the Aristotelian idea of "single correct taxonomy of all natural things".

In philosophy it has already been shown that the Aristotelian classical model has severe limitations when it comes to the modeling of real world phenomena. Taking into account the conceptual similarity of the classical model and the current object-oriented paradigm, it is therefore fairly obvious that object-oriented languages of today have pretty much the same shortcomings when it comes to modeling the real world. This is exemplified by the fact that there are many concepts and domains that are of interest to us but that cannot naturally be modeled in terms of shared properties. Examples of such "objects" include things like traffic jam, photon, water, ozone hole or greenhouse effect. If we want to use the current object-oriented paradigm to model concepts like these, we will have to explicitly resort to discrete, stochastic or probabilistic simulation models in which the actual problem domain is first converted into a form in which objects with shared properties exist. But the actual concepts themselves simply cannot be defined in terms of shared properties.

In most cases the limited modeling capabilities of the current object-oriented paradigm is not really a problem, since it usually suffices to have "good enough" models that describe the problem domain in a sufficiently rich level of detail. Also, the majority of business applications are such that they can fairly easily be represented with concepts that are defined in terms of shared properties. Furthermore, despite the inherent limitations, the modeling capabilities of the object-oriented paradigm are still much better than those of the other well-known programming paradigms. Consequently, the realization that object-oriented programming is not

really capable of modeling the real world is more an observation than a real problem. But what is important is that we should steer clear of claims such as "object-oriented programs directly reflect the real world" that have been surprisingly common in the past, at least in the OO marketing material.

No optimum class hierarchies. Another implication that arises from the Aristotelian classical model and its adoption in current object-oriented programming languages is the fact that there are no "optimum" class hierarchies. This is easily seen in everyday design and implementation of object-oriented systems. In many situations a class hierarchy that is very natural and intuitive from the conceptual point of view is not the most reusable, extensible, or time- or space-efficient one; the most reusable or extensible libraries are not necessarily efficient or conceptually elegant; or, the most efficient libraries may lack both the conceptual elegance and extensibility. In general, the design of a good class hierarchy typically involves trade-offs in various regards. Cook's paper on the redesign of the Smalltalk-80 class library serves as an interesting example of this phenomenon [Coo92]. Also, the mixin style of programming [BrC90] often leads to highly reusable libraries at the cost of reduced conceptual clarity.

An implication of the fact that there are no optimum class hierarchies is that the designers of object-oriented software should always be prepared for change. No matter how well designed the class library is, requirements may change in such a manner that substantial changes in the library are needed. Consequently, there is a clear need for methods and tools that allow class libraries to be easily transformed from one form to another. Such methods and tools have been investigated by several researchers, including Bergstein [Ber91], Casais [Cas92] and Opdyke [Opd92, OpJ93].

Consensus-driven design and "good enough" models. The fact that there are no optimum classifications and therefore no optimum class hierarchies either leads us to the observation that there is no such thing as a "perfect" design. In other words, when designing object-oriented software, we should not spend too much time on trying to come up with a solution that would meet all the desired requirements and criteria. Rather, the design phase should be more like a *consensus-oriented or consensus-driven* process in which a group of designers aims at reaching a sufficient, or "good enough" model of the problem domain. A central goal of this process is to come up with a common vocabulary that will assist the designers in subsequently communicating about the problem domain and discussing about their designs more efficiently. This is far more important than the perfectness of the design. Also, it should be kept in mind that the requirements are likely to change and that iteration is typically needed (discussed below). Thus, "good enough" is usually enough, and spending additional time on design would just lead to work that could potentially be wasted. (Of course, deciding what is "good enough" is often very hard; Ed Yourdon has provided interesting insights into the topic in his recent paper on "good enough software" [You96]).

Basic classes and the need for iteration. One of the central results of the prototype theory presented by Rosch and others [RoM75, RMG76] is the observation that not all concepts and categories are equal. Rather, there are categories that are more "basic" than others and objects that are "better" representatives of categories than

other objects. These "basic" categories (classes) and "best" representative objects are those that are usually found first, whereas the more general and/or specific classes can only be deduced later when more experience from the problem domain has been gathered.

An interesting observation is that when categories are organized into taxonomic hierarchies, such as class hierarchies in object-oriented programming, the basic classes typically end up *in the middle of the class hierarchy*. In contrast, those classes that are at the top (root) or at the bottom (leaves) of the hierarchies are typically of less interest, either because they are overly generic or overly specific for the purpose of examination.

However, the implementation of an object-oriented class hierarchy always proceeds (technically) from top to bottom, i.e., superclasses must exist before their subclasses. Therefore, there is an inherent conflict between the classification process and the implementation of an object-oriented class hierarchy: the generic, more abstract classes can only be found when a substantial amount of expertise on the problem domain has been gathered. If the implementation of a class hierarchy is started *a priori*, i.e., before a sufficient level of expertise has been reached, substantial iteration in the implementation of the library is inevitable, since later experience is bound to reveal generalizations and new abstractions that will necessitate changes in the superclasses. Alternatively, we could try to postpone the implementation until the "final" classification of the problem domain has been reached, but since we already know that perfect classification is rarely possible, this will not solve the problems in the long run.

In general, by utilizing the Aristotelian philosophy and prototype theory, we can prove that the construction of object-oriented class libraries is an *inherently iterative process*. This fact has been presented informally by several researchers and practitioners in the area of object-oriented programming. For instance, Johnson and Foote [JoF88] have argued that abstractions are usually discovered by generalizing from a number of concrete examples and that the abstraction process is likely to succeed much better if we have a lot of experience with the problem domain. That is the reason why it is typically much easier to build good class hierarchies for graphical windowing systems and parsers than, e.g., nuclear plant or solar system simulation. In general, useful abstractions are usually designed from the bottom up, i.e., they are discovered rather than invented, and will usually undergo a number of iterations until they will become conceptually and technically satisfactory. The less experience we have with the domain, the more iteration is needed.

6. Prototype theory and prototype-based object-oriented programming

Not all object-oriented programming languages are class-based. There is an interesting category of object-oriented languages in which there are no classes at all. In this *prototype-based object-oriented programming* model, all programming is done in terms of concrete, directly manipulatable objects that are often referred to as *prototypes*. These prototypical objects resemble the instances in class-based languages, except that prototypical objects are more flexible in several regards. For

instance, unlike in class-based languages in which the structure of an instance is dictated by its class, in prototype-based languages it is usually possible to add or remove methods and variables at the level of individual objects. Other differences include that in prototype-based languages object creation usually takes place by copying, and that inheritance is replaced by some other, less class-centered mechanism. *Self* [UnS87], for instance, uses a mechanism called *delegation* [Lie86] which allows objects to forward messages to other objects in case the current object does not know how to respond to the given message, thereby supporting the essence of inheritance: *incremental modification* [Coo89]. *Kevo* [Tai92, Tai93] uses a mechanism called *concatenation* to reach the same goal.

Prototype-based languages are conceptually elegant and possess many other characteristics that make them appealing. These languages are also seemingly closer to the prototype theory presented by cognitive psychologists and philosophers. For instance, the ability to modify and evolve objects at the level of individual objects reduces the need for a priori classification and encourages a more iterative programming and design style. In general, when working with prototypes, one typically chooses not to categorize, but to exploit alikeness. Rather than dealing with abstract descriptions of concepts (intensions), the designer is faced with concrete realizations of those concepts. Consequently, design is driven by evaluation in the context of examples: designers run their solutions to evaluate them in the context of some input to the program.

The change of focus in the design phase raises an interesting question: do prototype-based object-oriented languages help overcome the limitations of the Aristotelian tradition that constrains the modeling capabilities of the current class-based object-oriented languages? Unfortunately, this is not really the case. Most prototype-based languages of today are motivated by relatively technical matters. For instance, prototypes are commonly used for reaching better reusability through increased sharing of properties and more dynamic bindings of objects, or for providing better support for experimental programming [UnS87]. In contrast, they do not usually take into account the conceptual modeling side, let alone pay any attention to the philosophical basis that underlies object-oriented programming. In a way, thus far the developers of prototype-based object-oriented programming languages seem to have been even more ignorant to these underlying conceptual and philosophical issues than, e.g., the Scandinavian inventors of the class-based object-oriented paradigm (see, e.g. [BDM73, Knu88]).

Kevo -- a language motivated by family resemblances. Perhaps the only object-oriented language that comes closer to the family resemblance model presented by philosophers and cognitive psychologists is *Kevo* [Tai92, SLS94]. Kevo differs from most other prototype-based object-oriented languages in the respect that it does not support inheritance or delegation in the traditional way. Instead of these and other mechanisms that put a heavy emphasis on sharing and shared properties, Kevo objects are logically stand-alone and typically have no shared properties with each other. (Note that at the implementation level Kevo uses sharing extensively in order to conserve memory, but this is fully transparent to the programmer.) New objects are created by copying, and the essence of inheritance, incremental modification, is captured by providing a set of *module operations* that allow the objects to be

manipulated flexibly. Late binding is used to ensure that methods defined earlier can be overridden to extend existing behavior in an object-oriented manner.

In order to make it possible to perform modifications to objects not only at the level of individual objects, but also per group, Kevo uses a notion of *object (clone) family*. An object family is a system-maintained group of objects who are considered to be similar. When objects are modified, the system implicitly moves objects from one family to another, or creates new families as necessary. For instance, when adding new properties to a window object, a new family of objects is created, unless another object with identical properties already exist. Conversely, if the added properties are later removed from the window object, the object will again return to its earlier family (provided that the family still exists). As the criterion of similarity, object interface compatibility is used, meaning that objects are considered to be similar if they have the same external interface/signature. In an ideal situation, object comparison should be based on *behavioral compatibility*, i.e., ensuring that objects react to external stimuli identically, but in practice coming up with an algorithm that could determine 100% surely and efficiently whether two objects are behaviorally compatible is impossible.

Object families in Kevo have a conceptual relation to family resemblances presented by philosophers. When combined with the notion of stand-alone objects and the reduced focus on shared properties, this naturally leads to a design and programming style in which advance classification and categorization have a lesser role. Yet even Kevo is still far away from the model presented by prototype theorists who argue that in modeling and classification subjective perceptions have a central role. There is some recent work in the area of *subject-oriented programming* that aims at taking into account the subjective factors in object-oriented design, but at this point this work is still mostly preliminary (see, e.g., [HaO93]).

A Macintosh implementation of Kevo is available freely from address "ftp://cs.uta.fi/pub/kevo". Detailed information on Kevo is provided in the author's doctoral thesis [Tai93] that is also available electronically from address "ftp://cs.uta.fi/pub/atps".

6. Conclusion

In this paper we have given a short overview of the historical and philosophical background of object-oriented programming, and examined the implications of these background issues on current object-oriented programming languages and methods. We recognized Aristotle as the "conceptual father" of class-based object-oriented programming, whereas the work of Wittgenstein has served as an inspiration for the alternative prototype-based approach. It was noted that in philosophy and cognitive psychology, the Aristotelian "classical" model has been abandoned a long time ago, whereas in object-oriented programming that model is still the prevalent one. This is not necessarily a problem, however, since most of the typical business applications can be modeled fairly well even with the limited classical model, especially if the designers are aware of the limitations of the classical model. It was also pointed out that current prototype-based object-oriented languages are poorly developed when it

comes to taking into account the conceptual and philosophical benefits of the prototype-based approach, and that a lot of possibilities for future research in this area remain.

References

Aristo   Barnes, J. (ed), The complete works of Aristotle volume 1 (the revised Oxford translation), Princeton University Press, 1984.

Aus61   Austin, J.L., Philosophical papers. Oxford University Press, 1961

BDM73   Birtwistle, G.M., Dahl, O-J., Myhrhaug, B., Nygaard, K., Simula begin. Studentlitteratur, Lund, Sweden, 1973.

Ber91   Bergstein, P., Object-preserving class transformations. In Paepcke, A. (ed), OOPSLA'91 Conference Proceedings (Phoenix, Arizona, October 6-11), ACM SIGPLAN Notices vol 26, nr 11 (Nov) 1991, pp.299-313.

Bla91   Blaschek, G., Type-safe OOP with prototypes: the concepts of Omega. Structured Programming vol 12, nr 12 (Dec) 1991, pp.1-9.

Bla94   Blaschek, G., Object-oriented programming with prototypes. Springer-Verlag, 1994.

Bor86   Borning, A.H., Classes versus prototypes in object-oriented languages. In Proceedings of ACM/IEEE Fall Joint Computer Conference, November 1986, pp.36-40.

BrC90   Bracha, G., Cook, W., Mixin-based inheritance. In Meyrowitz, N. (ed), OOPSLA/ECOOP'90 Conference Proceedings (Ottawa, Canada, October 21-25), ACM SIGPLAN Notices vol 25, nr 10 (Oct) 1990, pp.303-311.

Bra83   Brachman, R., What Is-a is and isn't? IEEE Computer vol 16, nr 10 (Oct) 1983, pp.30-36.

Bro86   Brooks, F.P.Jr., No silver bullet – essence and accidents of software engineering. In Kugler, H.J. (ed): Information Processing 86, Elsevier Science Publishers (North-Holland), 1986, pp.1069-1076. Also in IEEE Computer vol 20, nr 4 (Apr) 1987, pp.10-19.

Cas92   Casais, E., An incremental class reorganization approach. In Madsen, O.L. (ed), ECOOP'92: European Conference on Object-Oriented Programming (Utrecht, The Netherlands, June 29–July 3), Lecture Notes in Computer Science 615, Springer-Verlag, 1992, pp.114-132.

Coo89   Cook, W.R., A denotational semantics of inheritance. Ph.D. thesis, Brown University, Technical report CS-89-33, May 1989.

Coo92   Cook, W.R., Interfaces and specifications for the Smalltalk-80 collection classes. In Paepcke, A. (ed), OOPSLA'92 Conference Proceedings (Vancouver, Canada, October 18-22), ACM SIGPLAN Notices vol 27, nr 10 (Oct) 1992, pp.1-15.

DMC92   Dony, C., Malenfant, J., Cointe, P., Prototype-based languages: from a new taxonomy to constructive proposals and their validation. In Paepcke, A. (ed), OOPSLA'92 Conference Proceedings (Vancouver, Canada, October 18-22), ACM SIGPLAN Notices vol 27, nr 10 (Oct) 1992, pp.201-217.

Gly94   Schwartz, B., Lentczner, M., Glyphic Codeworks(tm) scripting. Unpublished manual, Glyphic Technology, 1994

HaO93   Harrison, W., Ossher, H., Subject-oriented programming (A critique of pure objects). In OOPSLA'93 Conference Proceedings (Washington, D.C.), pp.411-428.

JoF88   Johnson, R.E., Foote, B., Designing reusable classes. Journal of Object-Oriented Programming vol 1, nr 2 (Jun/Jul) 1988, pp.22-35.

Knu88   Knudsen, J.L., Name collision in multiple classification hierarchies. In Gjessing, S., Nygaard, K. (eds), ECOOP'88: European Conference on Object-Oriented Programming (Oslo, Norway, August 15-17), Lecture Notes in Computer Science 276, Springer-Verlag, 1988, pp.93-109.

Lak87   Lakoff, G., Women, fire, and dangerous things: what categories reveal about the mind. University of Chicago Press, 1987.

Lie86   Lieberman, H., Using prototypical objects to implement shared behavior in object-oriented systems. In Meyrowitz, N. (ed), OOPSLA'86 Conference Proceedings (Portland, Oregon, September 26–October 2), ACM SIGPLAN Notices vol 21, nr 11 (Nov) 1986, pp.214-223.

LTP86   LaLonde, W.R., Thomas, D.A., Pugh, J.R., An exemplar based Smalltalk. In Meyrowitz, N. (ed), OOPSLA'86 Conference Proceedings (Portland, Oregon, September 26–October 2), ACM SIGPLAN Notices vol 21, nr 11 (Nov) 1986, pp.322-330.

Opd92   Opdyke, W.F., Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, Technical report UIUC-DCS-R-92-1759, 1992.

OpJ93   Opdyke, W.F., Johnson, R.E., Creating abstract superclasses by refactoring. In Kwasny, S.C., Buck, J.F. (eds): 1993 ACM Computer Science Conference (Indianapolis, Indiana, February, 16-18), ACM Press, 1993, pp.66-73.

Plato   Plato, The republic. Kustannusosakeyhtiö Otava, Keuruu, Finland, 1981 (Finnish translation).

RMG76   Rosch, E., Mervis, C., Gray, W., Johnson, D., Poyes-Brahem, P., Basic objects in natural categories. Cognitive Psychology 8, 1976, pp.382-439.

RoM75   Rosch, E., Mervis, C., Family resemblances: studies in the internal structure of categories. Cognitive Psychology 7, 1975, pp.573-605.

SLU88   Stein, L.A., Lieberman, H., Ungar, D., A shared view of sharing: the treaty of Orlando. In Kim, W., Lochowsky, F. (ed), Object-oriented concepts, applications and databases, Addison-Wesley, 1988, pp.31-48.

Smi94   Smith, W.R., The Newton application architecture. In Proceedings of the 39th IEEE Computer Society International Conference (San Francisco), 1994, pp.156-161

SLS94   Smith, R.B., Lentczner, M., Smith, W.R., Taivalsaari, A., Ungar, D., Prototype-based languages: object lessons from class-free programming (panel). In OOPSLA'94 Conference Proceedings (Portland, Oregon, October 23-27), ACM SIGPLAN Notices vol 29, nr 10 (Oct) 1994, pp.102-112

Ste87   Stein, L.A., Delegation is inheritance. In Meyrowitz, N. (ed), OOPSLA'87 Conference Proceedings (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987, pp.138-146.

Tai92   Taivalsaari, A., Kevo – a prototype-based object-oriented language based on concatenation and module operations. Technical report DCS-197-1R, University of Victoria, B.C., Canada, June 1992.

Tai93   Taivalsaari, A., A critical view of inheritance and reusability in object-oriented programming. Ph.D. thesis, University of Jyväskylä, Finland, November 1993. ISBN 951-34-0161-8.

Tou86   Touretzky, D.S., The mathematics of inheritance systems. Research notes in artificial intelligence, Morgan Kaufmann Publishers, 1986.

UnS87   Ungar, D., Smith, R.B., Self: the power of simplicity. In Meyrowitz, N. (ed), OOPSLA'87 Conference Proceedings (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987, pp.227-241.

Wit53   Wittgenstein, L., Philosophical investigations. Macmillan, New York, 1953.

You96   Yourdon, E., Good enough software. Application Development Strategies vol 8, nr 1 (Jan) 1996, pp.1-13.

Zad65   Zadeh, L., Fuzzy sets. Information and Control vol 8, 1965, pp.338-353.