Classifying Prototype-based Programming Languages

C. Dony[1], J. Malenfant[2] and D. Bardou[1]

[1] LIRMM — Université de Montpellier II
161, rue Ada. 34392 Montpellier Cedex 5, France.
E-mail: dony@lirmm.fr
[2] UR–VALORIA — Université de Bretagne sud
CGU de Vannes. 1, rue de la Loi. 56000 Vannes, France.
E-mail: Jacques.Malenfant@univ-ubs.fr

Abstract. The prototype-based programming model has always been difficult to characterize precisely. Prototype-based languages are all based on a similar set of basic principles, yet they all differ in their precise interpretation of these principles. Moreover, if the prototype-based model advocates concrete objects as the only mean to model concepts, current languages promote methodologies reintroducing abstract constructions to manage efficiently groups of similar objects. In the past, we have proposed two classifications of delegation-based languages in order to clarify these issues. In the present paper, we come back to these two classifications in the light of our recent work. The first classification looks at the primitives of the virtual machine underlying each language; it classifies languages according to the semantics of these primitives. The second considers the group-oriented constructions provided in each language; it classifies languages according to the level of abstractness of these constructions. The two classifications complements each others and also other existing classifications. They allow people to assess more precisely the relative merits of the different languages.

1 Introduction

A prototype is a typical member used to represent a family or a category of objects [15]. Prototype-based languages propose a vision of object-oriented programming based on the notion of prototype. Since the middle of the eightees, numerous prototype-based languages have been designed and implemented: SELF [45, 1, 2, 13, 37], KEVO [41, 42], AGORA [40], GARNET [33, 34], MOOSTRAP [31, 32], CECIL [12], OMEGA [5], NEWTON-SCRIPT [38]. Other languages, such as OBJECT-LISP [3] or YAFOOL [19] not qualified as prototype-based, nevertheless offer closely related mechanisms or use prototypes at the implementation level. Finally systems mixing prototypes and classes have also been proposed [22, 21].

A very general and informal characterization of prototype-based languages is rather simple: they propose a world in which there is one kind of objects equiped with attributes[1] and methods, three primitives to create objects: creation "ex

[1] We use that term to denote a structural characteristic of an object or "frame".

nihilo", cloning and extension (differential copy), and one control structure: message sending together with a delegation mechanism. Beyond this generalization, all these languages present slight yet profound and interesting differences.

They are different because they have been developed for different application domains. They are different because they have been inspired on the one hand by the earlier frame languages used in knowledge representation and on the other hand by actors languages used in distributed AI. They are also different because they have been developed with different goals.

- The first goal was to provide simpler descriptions of objects. People's natural way to grasp new concepts is generally to begin by creating concrete examples rather than abstract descriptions; class-based languages force people to work in the opposite direction, by creating abstractions (classes) prior concrete objects (instances).
- The second goal was to offer a simpler programming model with fewer concepts and primitives. Applications in the fields of user-interfaces [33] and virtual reality [7, 36] have tried to escape the traditional abstract data type model to move towards a less constrained one. For these applications, classes have been considered as a source of complexity because they are playing too many roles [8]. The alternative solution is often based on the concept of prototypes, more amenable to a form of programming-by-example and providing an alternative to class instantiation and class inheritance [8, 3, 45, 34].
- The last goal was to offer new capabilities to represent knowledge. Class-based languages constrain objects by disallowing, for example, distinctive behavior for individual objects among their instances and by forbidding inheritance between objects to share values of instance variables.

Thus, an exact and unique characterization of prototype-based programming raises a number of issues [17, 26, 4]. Current prototype-based languages differ in the semantics of object representation, object creation, object encapsulation, object activation and object inheritance.

- There exists various interpretations of what is a prototype, a concrete object or an average representative of a concept, which can lead to different languages.
- The semantics of the basic mechanisms (cloning, differential copy, delegation) is not unique and allows for different interpretations.
- Differential copy creates inter-dependent objects and authorizes various interpretations concerning the precise nature of the concept of object they implement.
- Finally, some of the capabilities offered by classes, for example the ability to express sharing at a conceptual level, have revealed to be so important for program organization that they have been reintroduced in different ways.

So, understanding each language, both in terms of their expressive power and their applicability to specific kinds of problems, is not always easy. The

```
Frame
  name: "whale"
  category: mammal
  environment: sea
  enemy: man
  weight: 10000
  color: blue
```

Fig. 1. Example of *Frame*

```
Frame
  name: "Moby-Dick"
  is-a: whale
  color: white
  enemy: Cpt-Haccab
```

Fig. 2. Differential description

Treaty of Orlando [39] proposes a first comparison of class-based and prototype-based languages; we go further by addressing more extensively the issues pending the alternative semantics associated to pure prototype-based languages and the level of abstration propsed by the different mecahnisms reintroducing class functionalities into prototyped-based languages. In the past, we have proposed two complementary classifications to this end. The goal of the present paper is to come back to these two classifications in order to present and explain their main classification criteria, but also to complement them in the light of our recent work.

The rest of the pqper is organized as follos. Section 2 recalls the genesis of prototype-based programming and thus proposes a first classification of languages. Section 3 proposes a first classification of prototype-based languages according to the vision their conceptor had on them. Section 4 presents the comparison criteria related to primitive operations and mechanisms that constitute a prototype-based language virtual machine. Section 5 presents the comparison criteria related to how programs written in prototype-based languages are organized. Section 6 proposes the two classifications of languages according to the previously defined criteria.

2 Genesis of the Prototype-Based Programming Model

To understand the genesis of ideas is a first important step in a classification process. This section recalls how frame-based and actor languages have influenced prototype-based ones.

2.1 Prototypes and knowledge representation

The concepts of prototype and differential description can be found in the frame theory [30] and in some systems inspired from this theory such as the frame-based languages KRL [6] or FRL [35]. Frames have been designed as an alternative way to represent knowledge such as typicality, default values or exceptions, which are difficult to describe in other formalisms [29]. Frame-based languages use the prototype's theory and they have influenced some prototype-based languages.

Structure. A frame is a set of attributes. Each attribute represents one characteristic of the frame and is made of a couple "attribute name - set of

facets". The most common facet, and the only one considered in our examples, is the attribute's value. The figure 1 shows a definition of a frame representing a `whale`.

Differential description. Differential description makes it possible to create a new frame by only expressing the differences from an existing one[2]. The differential description creates a relation between the new frame and the former (its prototype or parent). This relation is implemented by a link generally called "is-a"[3]. The figure 2 shows the definition of a frame representing `Moby-Dick`, which looks like the above `whale` but is white and has a specific enemy.

Frame hierarchies and inheritance. The is-a relation is an order relation that defines frame hierarchies [9]. A frame can inherit from its parent a set of attributes. Frame-based systems propose inheritance hierarchies made of representatives of concepts, which are conceptually very similar to those built later with prototype-based systems. One can generally find at the top of those hierarchies average representatives, such as `whale`, and at the bottom concrete representatives, such as `Moby-Dick`. What is tremendously important here is that what is shared are not descriptions, as with class inheritance, but rather concrete representations and so bindings of slots to values.

2.2 Actor languages

To represent entities with classless objects [46] has also been proposed in descriptions of ACT 1 [23] although the papers related to this language don't mention neither the prototype concept, nor its use in the earlier frame-based languages. ACT 1 provides objects and mechanisms conceptually close to those described above and part of the fundamental characteristics of today's prototype-based language.

From our point of view, the main difference between the frame-based language KRL quoted above and ACT 1 is that ACT 1 is a programming (and not a representation) language. Actors thus have attributes (acquointances) and a set of behavior. We will use the term "method" to denote the representation of a behavior and the term "property" to denote either an attribute or a method. Methods are invoked by sending messages to actors[4]. The figure 3 shows an actor named `point` with two attributes and one method. Actors, being classless objects, are created by cloning and extending existing ones with three primitives `create`, `extend` and `c-extend` [10].

Cloning. Although a copy primitive did exist in earlier languages such as SMALLTALK, ACT 1 has introduced cloning (shallow copying) as a primitive way

[2] "*The object being used as a basis for comparison (which we call the prototype) provides a perspective from which to view the object being described. (...) It is quite possible (and we believe natural) for an object to be represented in a knowledge system only through a set of such comparisons*" [6].

[3] This link may be accessible to the programmer via a related attribute also called `is-a`.

[4] This simplified vision is sufficient here; in fact, methods are grouped in a script and invoked via pattern matching.

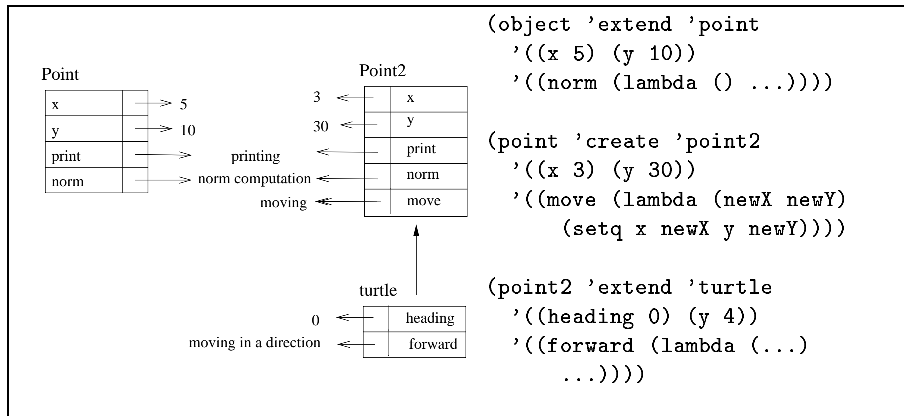

Fig. 3. Cloning and extension in ACT 1.

to create new objects. The simple biological metaphor of cloning makes it very appealing, compared to the traditional way to create objects in class-based models, namely by instantiation. The basic idea is that, given an existing and coherent object, it is easier to get a new object similar to the first one by simply copying it than to instantiate correctly a new one with coherent initial values for its instqnce variables. Another primitive, called `create`, combines cloning and addition of new properties. For example, in Figure 3, `point2` is a clone of `point` with new attribute's values and to which a new method has been added.

Extension. Differential description is conceptually similar to that of frames and is performed by a primitive called `extend`, which creates a new actor that we will call an "extension" of its model. The model is called the proxy and is the equivalent of the parent or prototype in the frame's terminology. The figure 3 shows the definition of a (Logo) `turtle` actor as an extension of `point2`. A turtle is like a point but as a heading and a forward behavior to move in the direction defined by its heading. What we call extension should not be confused with the possibility to add new attributes to objects.

Inheritance and delegation. The relation between an actor and its proxy is really similar to the is-a relation between frames. This relation is implemented by a link named `proxy`, that we will also call `parent` in the rest of the paper. An extension can inherit properties of its proxy. The inheritance is achieved whenever an actor does not know how to handle a message, in which case it (explicit delegation), or the system (implicit delegation), can ask its proxy to answer for it. The transfer of control to the proxy is called delegation[5]. Papers on ACT 1 do not precise the context in which a method is executed after a delegation (the binding of self); this mechanism is however a first form of what will become delegation in prototype-based languages.

[5] "*Whenever an actor receives a message he cannot answer immediately on the basis of his own local knowledge (...), he delegates the message to another actor, called his proxy*". [23]

3 Different visions of Prototype-Based Programming

The above systems contain the essence of what has been later called prototype-based programming. The first studies that led to the design of classless object-oriented programming languages have been published in the middle of the eighteees. They brought to the fore various issues generally related to class-based programming and proposed various solutions based on the use of prototypes. This section recalls these early proposals and thus supports classification based on the vision their conceptors had and the problems they wanted to solve.

3.1 Prototypical instances and cloning - A simpler programming model

The class-based model is complex [8, 22, 21, 39]; classes play different roles[6] that makes program design difficult. In reaction to this complexity, in a quest for a simpler programming model, Borning [8] has proposed an informal description of a classless language in which new objects are produced by copy and modification of prototypes. Once the copy is done, no relation is maintained between the copied prototype and its clone. The resulting programming model is simple but quite poor in term of sharing and in term of program organization. As far as we know, the model has not been pursued by its author but has inspired some complete cloning- and prototype-based languages such as KEVO [42], OMEGA [5] and OBLIQ [11].

3.2 Prototypical instances and differential description - A new way to conceive objects

At the same time, and partially for the same reasons, but also mainly to propose a new way to conceive and define objects, Lieberman has proposed to applied some ideas extracted from ACT 1 to object-oriented programming [24, 25]. He has described an informal programming model based on prototypical instances, cloning and differential description. Many different languages have been inspired by his work and share basically the same characteristics: SELF, OBJECT-LISP, NEWTON-SCRIPT, MOOSTRAP, etc. SELF has attracted the largest development effort and it has been widely distributed.

As a prototypical example of such languages, we can give a flavor of OBJECT-LISP [3], which has a well-known LISP syntax. The figure 4 shows an OBJECT-LISP version of the prototypical "point-turtle" example. Objects have attributes and methods; they can be cloned and extended. Extension objects have a parent and inherit their properties. Activation is based on message sending (function `ask`). There is no encapsulation, attributes can be accessed via message sending (`(ask turtle x)`) or through their name within methods. Delegation can occur either to retrieve the value of an attribute or to activate a method if the receiver does

[6] Let us quote: instance descriptors, method libraries, support for encapsulation sharing and reuse, support for the architecture of programs, etc.

```
(setq point (kindof))                 ;Creating an object ex nihilo.
(ask point (have 'x 3))               ;Creating an attribute for point.
(ask point (have 'y 10))
(defobfun (norm point) ()             ;A method norm for point.
  (sqrt (+ (* x x) (* y y))))         ;variables are the receiver's ones.
(defobfun (move point) (newx newy)    ;A method with parameters,
  (ask self (have 'x (+ x newx))      ;to add two points.
  (ask self (have 'y (+ y newy)))     ;Modifying values of attributes.
(defobfun (plus apoint) (p)           ;A method to add two points.
  (let ((newx (+ x (ask p x)))
        (newy (+ y (ask p y)))
        (newp (kindof apoint)))       ;Creating an extension of the object
    (ask newp (have 'x newx))         ;passed as parameter.
    (ask newp (have 'y newy))
    newp))

(setq point2 (kindof point))          ;point2 is an extension of point,
(ask point2 (have 'y 4)               ;with a new attribute y.
(setq turtle (kindof point2))         ;An extension of point2,
(ask turtle (have 'cap 90))           ;with a new attribute cap,
(defobfun (forward turtle) (dist)     ;and a method forward.
  (ask self (move (* dist (cos cap))
          (* dist (sin cap))))
```

OBJECT-LISP *is an extension of Lisp towards objects allowing both message passing and traditional function call. Message passing is implemented by the function* `ask` *; its first argument is the receiver and the second a function call where the name of a function is used as message selector (a method corresponding to that name must exist); the arguments of the function call are the arguments of the message. The following primitives are used in the example:*
– creating objects "ex nihilo": function `kindof` *without arguments,*
– creating extensions: functions `kindof` *(with one argument, which is the extended object),*
– method definition: function `defobfun`*,*
– attribute definition: method `have`*.*

Fig. 4. A prototype-based language example – OBJECT-LISP.

not hold the required property. Delegation can here be described informally in the following way: "an object that cannot answer a question can delegate it to its parent, if the parent can answer it, the answer will be performed in the context of the values of the original object". Thus, compared to ACT 1, the fundamental new points are explicit description of polymorphism and dynamic binding. In our example, the method `norm` is polymorph because it can be applied to a point or to a turtle. Dynamic binding ensures that a method executed after a delegation is applied in the context of the initial receiver (in our example, sending the message `norm` to `turtle` returns `5`).

3.3 Classless objects - knowledge representation

Another motivation to study classless languages was a quest for greater flexibility to represent knowledge and express sharing in object-oriented programs.

As shown by the frame experience, prototypes offer new capabilities to represent knowledge difficult to grasp in class-based languages such as [24, 39, 17]:

- different objects of the same family having different structures and behaviors,
- exceptional objects,
- objects with viewpoints and sharing at the object level,
- incomplete objects to be classified.

Such capabilities, although widely used, are more specifically the mark of a family of hybrid languages, combining declarative knowledge representation and programming. Representatives of such languages are GARNET [33, 34] (the object layer of which, named KR, being prototype-based), which is dedicated to graphical user interfaces, or YAFOOL [19], which is a knowledge representation language based on frames that can have methods in the sense of traditional object-oriented languages.

3.4 Class and object hierarchies - Disconnecting subtyping and reuse

A last family is made of languages that integrates both prototypes and classes. If several propositions have been made in this direction and have influenced actual languages, this family has yet to produce a concrete representative.

Indeed, one of the first proposal embedding object hierarchies [22, 21] has suggested to mix in a single world classes and instances (called examplars) having some kind of autonomy. The main purpose of that proposition was related to object-oriented program organization; it was meant to experiment a separation between subtyping hierarchies between classes, and code inheritance hierarchies (or implementation hierarchies) between instances. In this proposal, object methods are stored in instances and type interfaces in classes; classes can have different instances with different implementations of the same method. For example, the instances `empty-list` and `non-empty-list` of the class `list` have a different version of the `append` method. New objects are created by cloning instances; for example, new lists are created by cloning either `empty-list` or `non-empty-list`. Besides, any instance can inherit from any other, some private methods necessary for the implementation of the methods that compose its interface. The delegation hierarchy between instances is not necessarily isomorphic to the inheritance hierarchy between classes[7].

Although such proposals have not been pursued, they have influenced actual languages and represent, in our opinion, a source of new ideas for the applications of the prototypes formalism [26, 4].

[7] By the way, the non isomorphic interface and class hierarchies of JAVA, each class implementing one interface, is a new form of that idea.

4 Classification criteria related to primitive mechanisms

The classification of the previous section is based on various visions of what is classless programming and on various application domains for prototype-based programming. We now focus on a classification based on the primitive operations of prototype-based languages. Beyond a set of similar basic principles (object-centered representation, dynamic addition and deletion of slots, cloning and delegation), prototype-based languages differ in the precise interpretation of the primitives that constitute their virtual machine and in the way programs are organized. To build some classifications reflecting those differences requires to present the comparison criteria.

4.1 Object Representation

Objects in prototype-based languages are defined by a set of properties. A property is basically the binding within the object of name to a value[8]. Properties are conceptually of two kinds: attributes or methods. Two main solutions have been considered for the representation of objects: (1) to separate the concepts of attributes and methods or (2) to amalgamate them using the concept of slot. The first alternative mimics objects of a traditional class-based approach, the value of each attribute can be accessed within methods by referencing its name and can be changed through assignment. In the second alternative, no distinction is made between variables and methods; instead, an object gets a collection of slots where a variable can be viewed as a method that simply returns a value.

Distinction between variables and methods : are objects represented with attributes and methods or with slots?

Indeed, both alternatives have advantages and disadvantages. The advantages of slots are advocated by SELF [45]: they are more flexible, they allow users to access in the same way attributes and methods, they permit to override an attribute with a method and conversely a method with an attribute. However, they force to implement an explicit encapsulation mechanism. With the variables&methods approach, encapsulation of variables can be simply enforced, by establishing standard (SMALLTALK-like) visibility rules (OBJECT-LISP— cf. Fig. 4 — does not provide such rules, variables can be accessed via message sending).

4.2 Criteria related to object creation and evolution

Two alternatives are possible to create new objects in current languages. An object can be created *ex nihilo* or it can be created from an existing one (by cloning or extending it). Whether or not a primitive to create new objects ex nihilo is provided makes up our second criteria.

[8] A property can also have a type, a domain, facets, ...

Creation *ex nihilo*: is it possible to create new objects *ex nihilo*?

If a primitive to create new objects ex nihilo is provided, two new alternatives show up. We can create empty objects, or objects with an initial structure. If we restrict ourselves to the creation of new empty objects alone, this raises the question of what to do with them? Some means must be provided to modify their structure in order to build the concrete objects in applications. Indeed, this introduce two other design alternatives, can the structure of objects be extended and/or shrunk dynamically, by adding or deleting properties?

Dynamic modification of object structure : can object's structure be modified?

Consider the creation of new objects from existing ones. The first solution is cloning. Most languages (except GARNET and AMULET) provide a primitive for cloning objects. Cloning (cf. Section 2.2) in itself offers two alternatives: shallow cloning or deep cloning. However, deep cloning is usually ruled out as an uninteresting alternative because it is time consuming and provide little interesting properties on its own. The second solution is extension and is discussed in the next section.

4.3 Extensions and other solutions for life-time sharing between objects

A new alternative appears with the question of extension objects (cf. Sections 2.1 and 2.2): are extensions necessary and what do they add to cloning? Some languages do not provide primitives to create extensions of other objects. For example KEVO only allows for cloning and adding new properties to clones. The object `point2` in figure 3 is a clone to which a new property has been added; it would have been possible to define `turtle` by cloning `point2` and by adding the new object the properties `heading` and `forward`.

The difference between an extension and an augmented clone lies in the way objects share properties [17]. From the sharing and reuse point of view, shallow cloning essentially means that immediately after cloning, the corresponding slots of an object and its clone will point to the same objects. For example, consider `point` and its clone `point2` in the same figure 3, they share the method `print` and their position by the virtue of pointing to the same value objects through their respective slots `print`, `x` and `y`. However, even if they get the same structure and the same values, they have independent slot bindings; if `point` is modified for example, the two objects cease sharing the values of the updated slots. If a new property is added to `point`, `point2` will not have it. Hence, shallow cloning enforce "creation-time sharing", characterized by an independent evolution of the clone and the copied object, which prevents objects to be unexpectedly modified through their clone. The independent evolution applies to slot individually; here `point` and `point2` continue to share the method `move`, even after `point2's y` slot has been modified.

With extensions, we face "life-time sharing": the extension inherits the properties of its parent and as long as the objects exist, the sharing will continue. In our example, `turtle` will continue to share the `y` slot of `point2` even if this slot's value is modified. Moreover, if a new property is added to `point2`, `turtle` will inherit it.

Cloning and extension mechanisms are thus different and have distinct applications.

Does the language provide both cloning and extension mechanisms?

When an extension mechanism is provided, some languages allow new objects to be extensions of more than one object. There is no conceptual difference, this possibility simply raises usual multiple-inheritance problems when accessing inherited properties. Some languages also allow an object to change its parent at run-time; this is called dynamic inheritance in SELF.

Multiple parents : is it possible to have multiple parents? Dynamic inheritance : is it possible for an object to change its parent.

When no extension mechanism is provided, achieving life-time sharing between objects requires other "group-wide mechanisms". For example, KEVO makes possible, via a mechanism that we will call *propagation*, to add, for example, a new method to all objects of the same clone family [43]. This requires of course that such families be handled automatically by the system.

If the language provides no extension mechanism, is there a propagation-like mechanism allowing to achieve some kind of life-time sharing.

4.4 Message sending

As in other object-oriented languages, the basic control structure of prototype-based languages is message sending. The alternatives with message sending lie in the way methods are searched. In the simplest case, methods are simply searched in the object that has received the message. However, most prototype-based languages provide a sharing mechanism named delegation. Delegation can be implicit or explicit.

Implicit delegation is used goes hand in hand with a corresponding extension mechanism. Delegation here is a kind of inheritance mechanism allowing to retrieve, access or apply properties of an object, which are in fact owned by its parents. Implicit delegation was first introduced by Lieberman [23] (see Section 2.2) as a message forwarding mechanism, who has accurately described it in [24] (see Section 3.2). With implicit delegation, when an object does not own the requested method or attribute, the interpreter automatically delegates the question to the object pointed by its parent link.

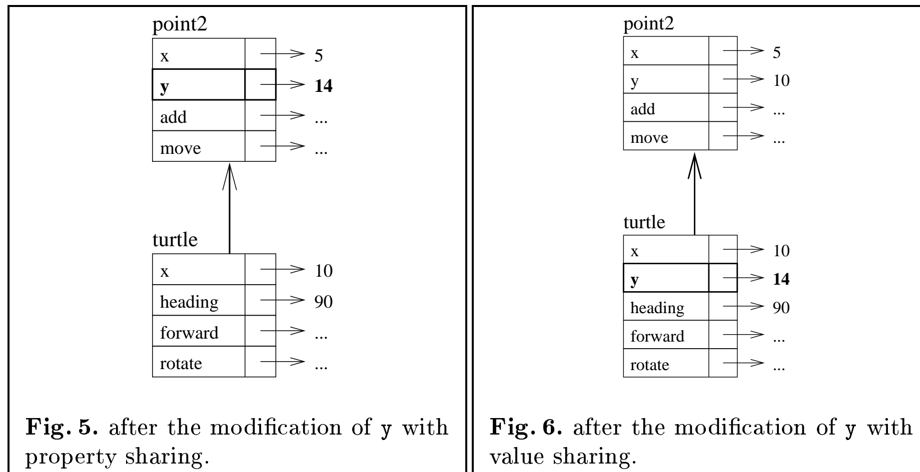

Fig. 5. after the modification of `y` with property sharing.

Fig. 6. after the modification of `y` with value sharing.

Another form of delegation is qualified as explicit, where any object can explicitely delegate, via a specific instruction, a request to another object. Explicit delegation is quoted in ACT 1 and is also used in OBLIQ (achieved through an alias mechanism).

> If the language provides a delegation mechanism, is delegation implicit or explicit?

4.5 Extensions, delegation and sharing

The next comparison criterion is related to the interpretation that is given to the notion of extension. Such an interpretation governs the exact semantics of the delegation mechanism. More precisely the interpretation given to the extension mechanisms governs what is inherited by an extension and what is shared between an extension and its parent. The delegation mechanism is responsible for correctly achieving inheritance and sharing. To introduce those different interpretations, let us recall that any object created in a program represents an entity being represented in the program, which is part of the real world; we call the set of such entities the "domain" of the program.

Delegation as a sharing mechanism between representation of different entities In the first interpretation, that we will call *value sharing interpretation*, the notion of extension is used to express attribute and method values sharing between two objects representing different entities of the domain. For example, consider again in the figure 3 the object `turtle` created as an extension of `point2`. The object `turtle` represents a Logo turtle of the domain and the object `point2` represents a point of the domain. The reason why the `turtle` object is made a child of the `point2` object is clearly the reuse of the definition of `point2`'s properties. We do not want `point2`'s properties to be the very properties of

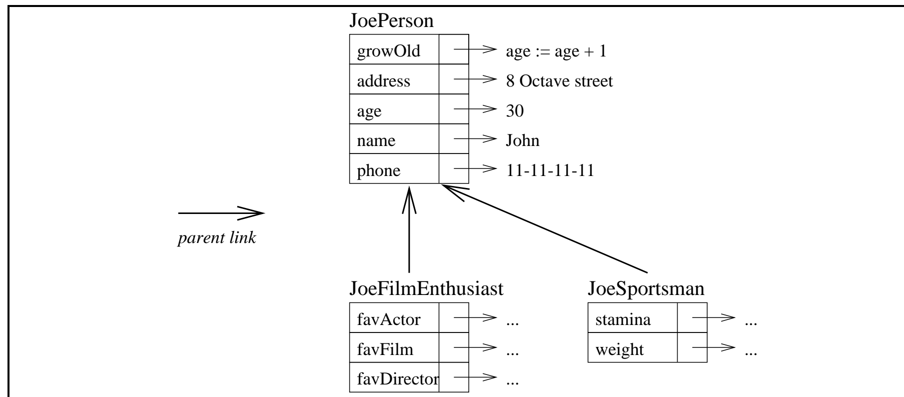

Fig. 7. A representation of a person with three objects.

`turtle`, but rather `point2` to provide default values for the `turtle`'s variables, which are not redefined. An object has at least as many properties as its parent, each of these properties is identified by the same name within the context of any of the two objects and has the same default value[9]. If the entities are distinct, the objects cannot share properties but only property values.

What about delegation? With this interpretation, an object should not be modified by sending an assignment message to one of its descendants. For example, an assignment message for the slot `y` sent to `turtle` should not result in `point2`'s `y` value modification as shown in Figure 5 but rather in the definition of the `y` variable in `turtle` as shown in Figure 6. In other words, assignment messages (or write access to attributes in methods bodies) should not be delegated but should eventually entail a local redefinition of the attribute. Parent (or delegation) links can in this case be intended as "is-like-a" links. Delegation then grants read access to variables but no more write access to the parent properties. The frontiers between objects are then clearer: in our example `point2` and `turtle` can be considered as two different objects because they can evolve almost on their own. The only way to modify the value of the `point2`'s `y` variable is to explicitly send a message to `point2`. Although this will also modify the `y` slot's value for `turtle`, this may be acceptable in a sense, since this value is intended to be a default one.

In terms of sharing, such an interpretation of variable assignment amounts to restrict, between an extension and its parent, property sharing to value sharing (the object `turtle` does not share with `point2` the property named `y` but the value of that property as long as it is not redefined on `turtle`).

Delegation as a sharing mechanism between representation of viewpoint on the same entity. In a second interpretation, that we will call :

[9] Lieberman has also suggested objects as default behavior and value repositories for their children in [24].

"*property sharing interpretation*", an extension object and its parent can be seen as different parts of the same domain entity. In such a case, properties of the parent can be seen as full properties of the extensions. To split a representation in several objects in a delegation hierarchy is a natural way of representing viewpoints.

For example, consider objects collectively representing a person — say "Joe" — in a delegation-based system as shown in Figure 7 [17, 27, 4]). The object `JoePerson` holds the basic information about the person (its `address`, `age`, `name` and `phone` and a method `growOld` while the object `JoeSportsman`, an extension of `JoePerson` holds information related to Joe as a sportsman. Creating `JoeSportsman` as an extension object instead of simply adding the slots `salary` and `company` to `JoePerson` also leaves the door open for other extensions, e.g. Joe as a film enthousiast. Any modifications to `JoePerson` are automatically seen by its extensions. Also, changes to the person Joe can be made through its extension objects. For example, if the employee changes its personal address, the change will naturally be made at the person level and will be effective for all extensions of this person. As in a description hierarchy, the most general viewpoints are those denoted by the objects near the top of the hierarchy whereas the most specific viewpoints are those denoted by the objects which are leaves of the hierarchy. In our example, person is a more general viewpoint on *Joe* than either sportsman or film enthusiast.

What about delegation? With this interpretation, it is coherent that an object may be modified by sending an assignment message to one of its descendants. For example, asking `JoeSportsman` to change its age can (and even has to) result in a modification of the attribute `age` of `JoePerson`.

In terms of sharing, this interpretation is an application of property sharing between objects. The `address` or `age` properties, not only their values, are shared by the objects `JoePerson`, `JoeSportsman` and `JoeFilmEnthusiast`. Property sharing is a characteristic of the extension mechanism and is achieved through the delegation mechanism.

> Interpretation of the extension mechanism : When the language proposes an extension mechanism, is it with a "property sharing" or "value sharing" interpretation?

A short analysis. Fully analysing these choices goes beyond the scope if this paper. A first partial analysis can be found in [17] and more recent ones in [27, 28, 16].

Let us just insist that the "property-sharing" interpretation raises some encapsulation issues when used in a "point-turtle" like example, indeed delegation establishes a so strong link between the objects `point2` and `turtle` that they cannot any more be considered as representing distinct entities. Conversely, it enables split representations. A split representation is a set of objects linked by delegation, representing a single entity of the domain such as the person Joe. There is, as far as we know, in today's prototype-based languages, no way

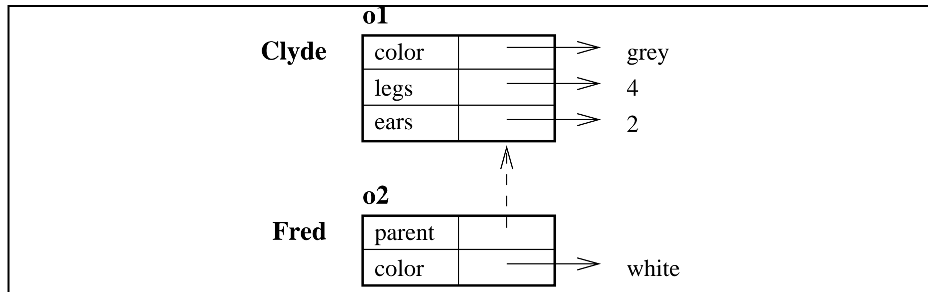

Fig. 8. Differential creation of the prototype `o2` representing Fred from the prototypical instance of elephants also representing Clyde.

to handle split representations as first-class objects, entities for which we have coined the name *split objects*. Handling split objects in a language would mean to create them, refer to them, clone them and otherwise deal with them as with other objects. This is an open issue on which we are currently working [4, 27].

Besides, the "value-sharing" interpretation partially eliminates encapsulation issues by making parents independent of their extensions but it also restrict the expressive power of the language by forbidding split representation with viewpoints.

5 Classification criteria based on program organization

Despite the basic principles of prototype-based programming, which advocates concrete objects as the only mean to model concepts, current languages have introduced mechanisms to deal with groups of similar objects. The three most common ones are: prototypical instances, traits objects, and maps.

Early in the foundation of prototype-based programming, Lieberman has proposed the mechanism of prototypical instances to share properties among objects related to a similar concept. The essential idea is to use the first concrete example of a concept as the representation of the concept itself. The well-known elephant example illustrates this (Fig. 8). Here, the object `o1` represents the elephant Clyde, which is the first elephant we have encountered. Clyde is therefore chosen as the representation of the concept of elephant. It is grey, has four legs and two ears. When the white elephant Fred appears, delegation allows us to differentially create Fred as an object `o2`, having `o1` as parent and simply redefining the slot `color` to be `white`.

Traits and maps have been invented in the prototype-based language SELF [44]. A traits object is a repository for methods (and "semi-global" variables) applying to the whole group of its delegating objects. The trait-based programming model [44] proposes to segregate the slots of a prototypical object in two parts: the representation part and the protocol part. Typically, the protocol part consists of slots holding method values while the representation part will consist

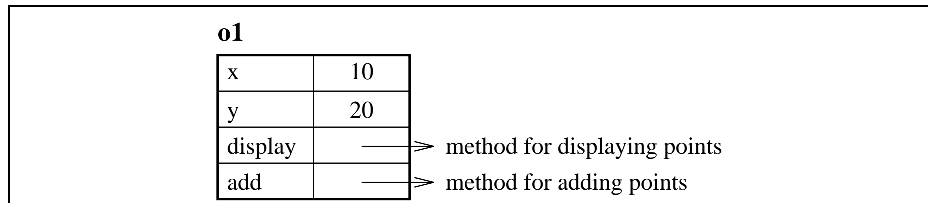

Fig. 9. A point example.

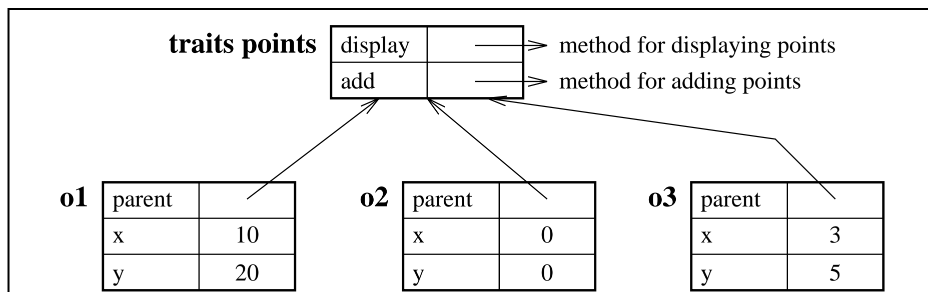

Fig. 10. A point example using a traits object.

of slots holding either object identifiers or constant values, but this need not to always be the case. In our first example (Fig. 9), the representation part of a point object would include the slots `x` and `y`, while the protocol part would include the slots `display` and `add`. A traits object is an object holding the protocol part, the idea being that this object can be shared among several copies of the representation part (themselves represented as objects, see Fig. 10).

Traits encourage the creation of delegation hierarchies that look like an inheritance one in its higher levels made of traits. At the leaves, we find concrete objects, somewhat like the instances in class-based programming. Trait-based programming leaves more flexibility in the creation of objects, and traits are not classes: they are less flexible and less abstract. An operation `traitof` returning the first parent traits of an object can be provided; testing whether two objects support the same protocol then boils down to test if the two objects' traits are the same or have a common parent. In this sense, trait-based programming emphasizes the notion of similar behavior among objects.

A map factors structural information out of objects in a clone family, i.e. a group of structurally identical objects obtained by cloning one another. Maps, illustrated in Figure 11, are similar to standard prototypes, except that the slot values are replaced by indexes giving the relative position of the slot values in the object, now represented merely as a vector of slot values. In fact, SELF goes beyond that by putting in the map the values of immutable slots, an existing concept in this language. When an object receives a message, the selector is used to look up the map to find a slot index, a value or a method. If a value is found, it

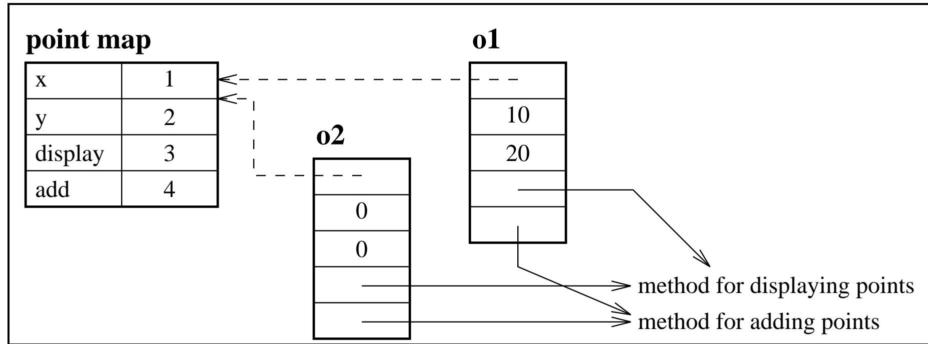

Fig. 11. The point example using maps.

is returned; if a method is found, it is applied in the context of the receiver; if an index n is found, the content of receiver's *nth* slot is retrieved and either returned if it is a value or applied if it is a method. In SELF, maps are created automatically behind the scene and if an object is cloned, the two clones will share the same map. They give SELF the same space efficiency in the representation of objects as the one of class-based languages. Because maps are not object in SELF, they have no direct influence on its programming methodology.

5.1 Generalization

Prototypical instances, traits and maps are specific mechanisms used in existing languages to structure programs. Compared to the original objectives of the prototype-based model, we have argued [26] that they are going against the very notion of prototype-based programming as it has been originally defined. But from them emerge a much richer notion of *delegation-based programming.* In fact, delegation-based languages exhibit much more diversity than it first appeared, because these new mechanisms impose slightly different programming models, which can hardly be put under the same "prototype-based" hat.

This approach allows us to complement existing classifications, such as the one presented in the previous section, but also the ones of Wegner [46] and the Treaty of Orlando [39]. Wegner identifies the class of object-centered programming languages as classless object-based languages, within which he singles out delegation-based languages, i.e. "*classless objects with delegation*"[10]. In our point of view, Wegner's classification underestimates the diversity of classless object-based languages, indeed because it is an attempt to characterize the whole design space of object-oriented language. Moreover, at the time of his writing, delegation-based languages were still underinvestigated. It is in this sense that we complete his classification, being more precise in one of his original class.

[10] A little confusion appears in [26], where object-based languages are understood as always classless.

We propose to classify delegation-based programming languages according to the number of different kinds of objects and the number of different kinds of links they manipulate. For example, the parent-of link of delegation is one link manipulated in all delegation-based programming languages. Similarly, a trait-based programming language manipulates two kinds of objects: concrete ones and traits. We argue that the number of kinds of links and objects manipulated in a language bears important insights into the nature of its programming model. We explore four classes of delegation-based languages: languages with one kind of object and one kind of link, ones with two kinds of objects and one kind of link, ones with two kinds of objects and two kinds of links and finally ones with one kind of object and two kinds of links. This classification is founded on four illustrating languages, the formal semantics of which have been developed and thoroughly examined [26].

Prototype-based languages have always been criticized for their lack of manifest organization in programs. Our new classification brings to the fore the existence of delegation-based languages with a more and more structured programming model, forming a continuum between pure prototype-based languages and class-based ones. Not only this shows that the organization of a program can be made more explicit without sacrificing an object-centered programming model, it also suggests a step by step (possibly automatic) transformation of prototype-based programs into class-based ones, a programming methodology advocated before [46, 39].

5.2 Languages based on one kind of object and one kind of link

Delegation-based languages where the space of objects is completely homogeneous and where delegation is used for sharing, correspond to the usual notion of prototype-based languages. All objects are equally first-class entities: they can be created dynamically, they can be sent a message, they are all mutable, they can be passed as parameters and returned as results. All of them can be used as parent and cloned. We categorize these languages as having one kind of object and one kind of link, namely the parent-of link.

5.3 Two kinds of objects, one kind of link

Typically, a language with one kind of object and one kind of link will evolve towards one with two kinds of objects when some objects become exceptional compared to others. Objects can become exceptional because of a particular programming methodology that must be supported at the language level in order to have all its strength. They also become exceptional when their "first-classness" is severely restricted, such as being immutable or abstract (in the sense that they cannot answer messages). They may not be cloned or used as parents. We illustrate this category of languages with a trait-based programming language.

The trait-based programming model is a good example for this class of languages, having two kinds of objects, prototypes and traits, and one kind of link,

delegation. In prototype-based programming, while assumed to be a concrete object, the traits object will fail if sent messages since presumably the corresponding methods will try to send the receiver (self) messages accessing the representation part, which will fail. In fact, for trait-based programming to work properly, most traits objects must never receive messages. And we propose that the language must make sure that it happens like that.

5.4 Two kinds of objects, two kinds of links

Languages with one kind of link will typically have a delegation, or parent-of link, which implements sharing akin to inheritance. Adding a second kind of link suggests introducing a structural description link similar to the class-of link. In traditional prototype-based languages, each object being one-of-a-kind, it gets both its slot names and slot values. When a large number of structurally identical objects are created, it becomes tempting to share the structural information, a line of reasoning that pushed the SELF team to invent *maps* [14].

A map-based language is constructed by making maps true objects. Because their slots contain indices to be reinterpreted in the context of the receiver, maps, as objects, cannot answer messages. So we make them abstract objects unable to answer messages. With maps represented as objects, it becomes possible to take advantage of them in the programming methodology. A map-based language encourages a programming methodology where the notion of structurally similar objects becomes very important. In such a language, we can speak about a group of structurally identical objects. Map-based operations to add or delete a slot to all objects in a clone family, etc. should be implemented.

5.5 Two kinds of links, one kind of object

Typically, a language with two kinds of links and one kind of object can be obtained by some rationalization of a language with two kinds of objects (and two kinds of links). For example, consider the map-based language proposed earlier. Can we transform maps in order to make them standard objects capable of answering messages without impairing the programming model? The answer is yes. The problem with maps begins when we send them messages whose result needs a reinterpretation in the context of a concrete object in order to make sense. But maps don't need to be implemented like standard prototypes. A simple idea, illustrated in Figure 12, is to replace the map by a descriptor object with one slot called `slotsDict`, which points to an object implementing a slot dictionary. The slot dictionary is not an ordinary prototype (despite a similar yet not identical aspect in Figure 12), but rather an object similar to SMALLTALK's method dictionary. We draw it as a box with round corners and it is actually an object answering `at:`⟨*some selector*⟩ returning its associated value and `at:`⟨*some selector*⟩ `put:`⟨*some value*⟩ messages like a SMALLTALK dictionary.

The advantage of this representation is that now we can send legitimate messages to descriptors, namely `slotsDict`, as well as to slot dictionaries. In

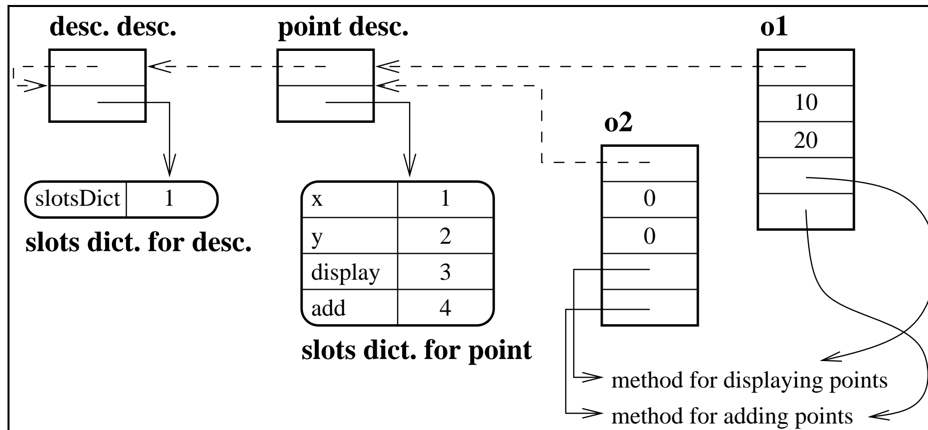

Fig. 12. The point example using descriptors.

fact, our descriptors look pretty much like SMALLTALK classes. A descriptor-based language similar to the previous map-based one can be designed very simply. In order to preserve an object-centered programming model, descriptors should be created automatically, like maps were. Nonetheless, descriptors are so much like classes that we are at the frontier between abstraction-centered and object-centered programming here. An important missing feature that makes the language still promoting an object-centered programming model is the lack of a sharing mechanism between descriptors that would have a semantics similar to inheritance. By taking care not to introduce such a link between descriptors, we don't encourage programmers to design their applications mainly around descriptors (see [28] for more information about descriptors).

5.6 But what's in a link?

The characterization by the number of links is an important measure of the complexity of a language, but since this classification was first published, examples have proved that the nature of these links is also important. Amulet, for instance, defines four different delegation semantics called modes. Clearly, classifying Amulet has having four kinds of links would bring little insight into the nature of the language.

This suggests another level of classification: classifying the links themselves. We have identified two main family of links: comparative links (*like*, delegation, inheritance, etc.) and descriptive links (*is-a*, *class-of*, *map-of*, *descriptor-of*, etc.). Other family of links could also be added, especially reflective links such as a *behavior-of* found in reflective languages [31, 40]. Kinds of links ars therefore better understood in our classification as kinds of families of links.

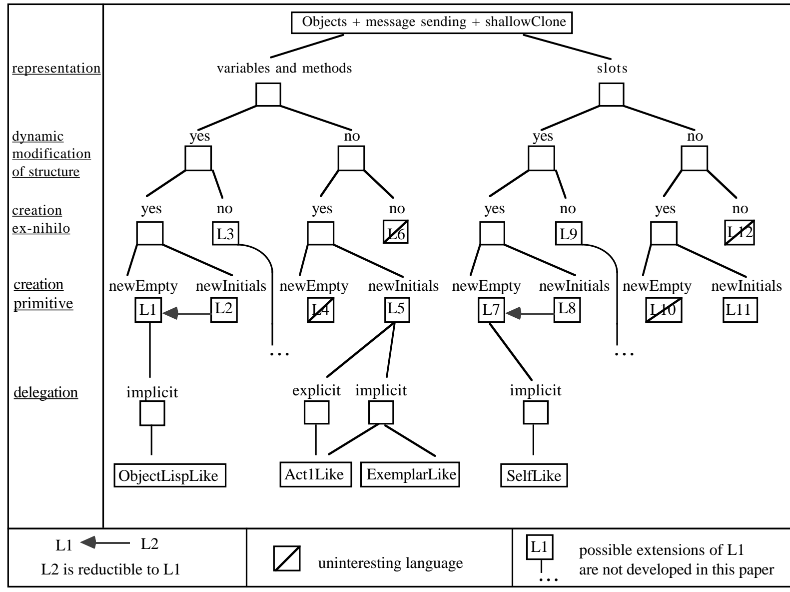

Fig. 13. An historical taxonomy [17].

6 Classifying Existing Languages

6.1 First classification based on primitives mechanisms

The taxonomy shown in Figure 13 is extracted from [17] and is only shown here to give an historical perspective that reflects our understanding of prototype-based languages at that time[11]. It takes into account how objects are represented, how they can be created and modified and the form of delegation (implicit or explicit) proposed by the language. Four existing prototype-based languages were classified in this taxonomy: SELF, OBJECT-LISP, ACT 1 and EXAMPLARS.

SELF and ObjectLisp are members of the language families (L8) and (L2) respectively, both extended with implicit delegation. From the point of view of that taxonomy, they only differ in the representation of objects; SELF uses slots while OBJECT-LISP uses variables and methods. EXAMPLARS is best characterized by the family illustrated by (L13): prototypes have variables and methods, the structure of prototypes cannot be modified dynamically, new objects are created ex- nihilo or by copying existing ones, the parent link is named "superExemplar" and delegation is implicit along this link. However since it is an hybrid language

[11] This classification is also operational one implemented as a SMALLTALK class hierarchy. The hierarchy can be used to interpret expressions of simulations of the classified languages. This SMALLTALK program, named Prototalk, is in fact a framework allowing to rapidly implement a simulation of any prototype-based language [18].

also providing classes, some of its characteristics are out of the scope of our taxonomy. ACT 1, described in Section 2.2, can be attached specific characteristics. Objects can have variables and a script. Messages to objects are examined by the script in which various actions can be performed, including explicit delegation to other objects in the system. When the script rejects a message, there is an implicit delegation to the parent of the receiver, the script of which being in turn executed. These functionalities can be classified in an extension of the language (L5) for which the evaluator is also able to deal with explicit delegation orders.

6.2 Second classification based on the semantics of primitives

This section now proposes a comparative table taking into account the whole set of criteria presented in Section 4. Most of the entries have been explained, let us simply quote a few particularities.

For what concerns the interpretation of the extension mechanism, languages such as GARNET or YAFOOL only propose the "value sharing interpretation" while others such as SELF or OBJECT-LISP only propose the "property sharing" one. Others, such as NEWTON-SCRIPT propose both by allowing to create the two kind of extensions implemented by two links named `_proto` or `_parent`. Another approach (encapsulated inheritance) is proposed by AGORA which permits each object to control the creation of its future extensions and the read/write access they will have on the attributes of the extended object [40]. We have quoted the advantages and drawbacks of both interpretations; besides, mixed solutions raise the issue of managing both kind of delegation links.

6.3 Abstraction-based Classification

The figure 15 summarizes our second classification. We have represented five categories of languages: four described here with one example language in each, and one corresponding to Wegner's classless object-based languages, in which he classifies Ada [20]. The list of possible languages in each class is by no mean closed. For example, another path towards a language with two kinds of objects but one kind of link appears when considering the status of prototypical objects in Lieberman's first proposal. In Lieberman's mind, the standard way to represent a concept is to provide a prototypical instance of this concept (Clyde is the prototypical elephant) to which other objects of the same concept can delegate for default properties. Because modifying a prototypical object has an important, and often undesirable, effect on all other objects delegating to it, we can make them immutable objects, hence stressing their particular role. In the Clyde–Fred example, Clyde would no longer be mutable, therefore making it impossible to indirectly modify Fred by mutating Clyde. A safe version of such a language designed in this line provides another example of a language with two kinds of objects and one kind of link.

It is also worth noting that the classification using the number of kinds of objects and links needs not to be restricted to object-centered languages. We have

	SELF	OBJECT LISP	GARNET (KR)	AMULET (ORE)	AGORA	MOOSTRAP
Creation ex nihilo	yes	no	yes	no	no	yes
Cloning	yes	yes	no	no	yes	yes
Extension mechanism	yes	yes	yes	yes	yes	optional
Dynamic modification of object structure	yes	yes	yes	yes	no (possible at meta level)	yes
Distinction between variables et methods	no	yes	no	no	yes	no (slots)
Interpretation of extension mechanism	property sharing	property sharing	value sharing	4 slot per slot different kinds of sharing	encapsulated inheritance	property sharing
Sharing mechanism	delegation	delegation	delegation	delegation	delegation (mixin-methods based)	delegation
Single/multiple parents	multiple	single	multiple	simple	simple (mixins)	simple or multiple
Dynamic inheritance	yes		yes		no	yes
Other language characteristics	*traits* and `lobby` based organization		inheritance hierarchy	inheritance hierarchy and composition hierarchy	inheritance hierarchy	reflexive kernel (`hall` and *traits*)

	NEWTONSCRIPT	KEVO	OMEGA	OBLIQ	YAFOOL
Creation ex nihilo	yes	no	no	yes	yes
Cloning	yes	yes	yes	yes (multiple)	yes
Extension mechanism	yes	no	no	no	yes
Dynamic modification of object structure	yes	yes	yes for prototypes no for others	no	yes
Distinction between variables et methods	no	yes	yes	no	no
Interpretation of extension mechanism	property sharing and value sharing		type-like sharing		value sharing
Sharing mechanism	delegation	propagation	propagation	concatenation	delegation
Simple/multiple parents	double	simple	simple	multiple	multiple
Dynamic inheritance	yes		no		?
Other language characteristics	inheritance hierarchy and ROM-defined prototypes	composition hierarchy and clone families	type hierarchy	aliases and distributed programming	Models and instances ...

Fig. 14. Languages comparison.

already noted that our descriptor-based language is at the frontier of abstraction-centered programming. If we take this language and add an inheritance link between descriptors, we end up having a language with one kind of objects but three kinds of links (parent-of, descriptor-of and descriptor inheritance) which cannot be characterized as object-centered. Moreover, consider a language where metaclasses are first-class objects as Cointe's ObjVlisp. Such a language has two kinds of objects (instances and classes, since metaclasses are simply classes whose instances are classes) and two kinds of links: instantiation and inheritance. However, it is certainly not object-centered. We have used our classification to

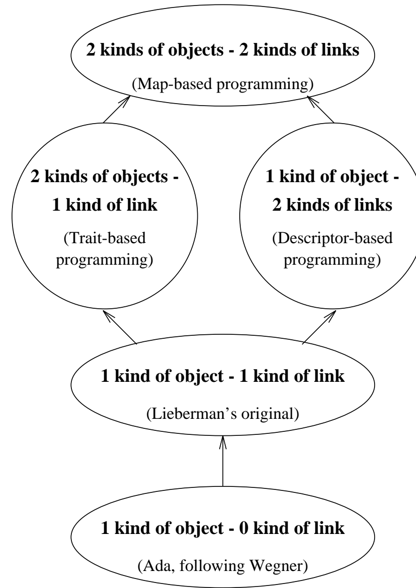

Fig. 15. The second classification with example languages.

characterize object-centered programming languages, but it is not restricted to this class of languages.

An interesting work now is to assess existing languages in the light of our classification. For example, in our view SELF is not prototype-based but it is certainly object-centered. It is our opinion that using the notion of object-centered programming with abstract objects presented here, the design of SELF can be positively enhanced. Our preferred path of upgrade would be to introduce traits as abstract objects like in Section 5 but to make maps first-class objects in the line of our descriptors (Section 5 and [26]). This would make SELF a delegation-based programming language with two kinds of objects and two kinds of links.

7 Perspectives

A domain, in order to be considered mature, must provide a comprehensive and intelligible description of its basic principles, its roots, its foundations, its alternative designs and its concrete realizations. It is our hope that our work since the beginning of the nineties has help to convey people in the field of prototype-based programming and their potential users some of this deep understanding. To this end, we have used traditional scientific processes: observation, comparison and classification. The two major contributions are the two classifications, which have clarify the design alternatives behind prototype-based languages, but also the richness of their different programming models as well as their position in the larger domain of object-oriented programming. These contributions

complement existing work, and especially the Treaty of Orlando and Wegner's classification.

Besides this classification effort, our work has also open some new research perspectives. The explicit definition of the kinds of sharing, life-time and creation-time sharing, achieved by cloning and delegation has shown that the two mechanisms are irreducible to each other. It has also led us to identify the problem of self, which occurs when frontiers between objects are blurred by teh sharing of slot bindings between them. In turn, the self problem has led us to identify on major use of delegation, which is to create split representations of domain entity using first-class split objects. Split objects consider as a whole a set of individual objects delegating to each other that represent one single domain entity. The kind of sharing allowed by delegation gives unique properties to split objects that make them especially useful in object-oriented databases and other persistent applications.

A second perspective is open by our second classification. We have introduced a general notion of object-centered programming, which admits some kinds of abstract devices, such as traits and maps, provided that the programming model is still dominated by the object-centered subset of the language, i.e. the major application design activity revolves around the creation of concrete objects. By recognizing the potential for abstract objects different from classes yet capable of structuring object-centered programs, we have reconcile prototypes and abstractions. Corollarily, we have brought to the fore the existence of more and more structured delegation-based languages forming a continuum between pure prototype-based languages and class-based ones.

For a long time, proponents of the prototype-based approach have suggested software development methodologies evolving from a liberal designs, using prototypes, towards a more a more structured one to end up with a completely structured applications using classes. Our work not only highlights the potential for several object-centered programming models, but also suggest a concrete path of evolution where prototype-based programs could be turned into class-based ones by successive transformations from less structured to more structured object-centered programs.

If prototype-based programming has still to find its place in the realm of software development, it is our convictions that split objects and object-centered programming have an essential role to play in its future.

References

1. O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Hølzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. *The SELF 3.0 Programmer's Reference Manual.* Sun Microsystems Inc. and Stanford University, 1993.
2. O. Agesen, L. Bak, C. Chambers, B.W. Chang, U. Hølzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. *The SELF 4.0 Programmer's Reference Manual.* Sun Microsystems Inc. and Stanford University, 1995.
3. Apple Computer, Inc. *Macintosh Allegro Common Lisp Reference Manual, Version 1.3*, 1989.

4. D. Bardou and C. Dony. Split Objects: A Disciplined Use of Delegation Within Objects. In *Proceedings of OOPSLA'96, Sans Jose, California. Special Issue of ACM SIGPLAN Notices (31)10*, pages 122–137, 1996.
5. G. Blaschek. *Object-Oriented Programming With Prototypes.* Springer-Verlag, Berlin, 1994.
6. D.G. Bobrow and T. Winograd. An Overview of KRL, a Knowledge Representation Language. *Cognitive Science*, 1(1):3–46, 1977.
7. A.H. Borning. The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory. *ACM Transactions on Programming Languages and Systems*, 3(4):353–387, 1981.
8. A.H. Borning. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM-IEEE Fall Joint Computer Conference, Montvale, New Jersey*, pages 36–39, 1986.
9. R.J. Brachman. What IS-A Is and Isn't: An Analysis of Taxonomic Links in Semantic Networks. *Computer*, 16(10):30–37, 1983.
10. J.-P. Briot. *Instanciation et héritage dans les langages objets.* Thèse de 3ième cycle, Université de Paris 6, 1984. Rapport LITP 85-21.
11. L. Cardelli. A Language With Distributed Scope. *Computing Systems*, 8(1):27–59, 1995.
12. C. Chambers. Predicate Classes. In O. Nierstrasz, editor, *Proceedings of Ecoop'93, Kaiserslautern, Germany. Lecture Notes in Computer Science 707*, pages 268–296, Berlin, 1993. Springer-Verlag.
13. C. Chambers, D. Ungar, B.W. Chang, and U. Hölzle. Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF. *LISP and Symbolic Computation*, 4(3):207–222, 1991.
14. Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *SIGPLAN Notices*, 24(10), October 1989.
15. B. Cohen and G.L. Murphy. Models of Concepts. *Cognitive Science*, 8(1):27–58, 1984.
16. C. Dony, J. Malenfant, and D. Bardou. Les langages à prototypes. In R. Ducournau, J. Euzenat, and A. Napoli, editors, *Langages et Modèles d'Objets.* INRIA - Collection Didactique, 1998. A paraître.
17. C. Dony, J. Malenfant, and P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In A. Paepcke, editor, *Proceedings of OOPSLA'92, Vancouver, Canada. Special Issue of ACM SIGPLAN Notices (27)10*, pages 201–217, 1992.
18. Christophe Dony. Prototalk: A framework for the design and the operational evaluation of prototype-based languages. Technical Report 97254, LIRMM, 1997. SOUMIS PUBLICATION.
19. R. Ducournau. *Y3 : YAFOOL, le langage à objets, et YAFEN, l'interface graphique*. SEMA GROUP, Montrouge, 1991.
20. J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, and B.A. Wichman. Ada Reference Manual and Rationale for the Design of the Ada Programming Language. *ACM SIGPLAN Notices*, 14(6):159–198, 1979.
21. W.R. Lalonde. Designing Families of Data Types Using Examplars. *ACM Transactions on Programming Languages and Systems*, 11(2):212–248, 1989.
22. W.R. LaLonde, D.A. Thomas, and J.R. Pugh. An Exemplar Based Smalltalk. In N.K. Meyrowitz, editor, *Proceedings of OOPSLA'86, Portland, Oregon. Special Issue of ACM SIGPLAN Notices 21(11)*, pages 322–330, 1986.

23. H. Lieberman. A Preview of Act 1. AI Memo 625, Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1981.
24. H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of OOPSLA'86, Portland, Oregon. Special Issue of ACM SIGPLAN Notices 21(11)*, pages 214–223, 1986.
25. H. Lieberman. Habilitation à diriger des recherches. Mémoire de synthèse. Université Pierre et Marie Curie, Paris 6 / LITP, Institut Blaise Pascal, 1990.
26. J. Malenfant. On the Semantic Diversity of Delegation-Based Programming Languages. In *Proceedings of OOPSLA'95, Austin, Texas. Special Issue of ACM SIGPLAN Notices (30)10*, pages 215–230, 1995.
27. J. Malenfant. Abstraction et encapsulation en programmation par prototypes. *Technique et science informatiques*, 15(6):709–734, 1996.
28. J. Malenfant. *Abstraction, encapsulation et réflexion dans les langages à prototypes.* Habilitation a diriger des recherches, Nantes University, 1997. Technical Report 97-4-INFO, École des mines de Nantes.
29. G. Masini, A. Napoli, D. Colnet, D. Léonard, and K. Tombre. *Les langages à objets.* InterEditions, Paris, 1989.
30. M. Minsky. A Framework for Representing Knowledge. In P. Winston, editor, *The Psychology of Computer Vision*, pages 211–281. McGraw-Hill, New York, 1975.
31. P. Mulet and P. Cointe. Definition of a Reflective Kernel for a Prototype-Based Language. In *Proceedings of the 1st JSSST International Symposium on Object Technologies for Advanced Software, ??. Lecture Notes in Computer Science 742*, pages 128–144, 1993.
32. Philippe Mulet. *Réflexion & langages à prototypes.* Thèse d'université, Université de Nantes, 1995.
33. B.A. Myers, D.A. Giuse, R.B. Dannenberg, B. Van der Zanden, D. Kosbie, E. Previn, A. Mickish, and P. Marchal. Garnet: Comprehensive Support for Graphical Highly-Interactive User Interfaces. *IEEE Computer*, 23(11):71–85, 1990.
34. B.A. Myers, D.A. Giuse, and B. Van der Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. In A. Paepcke, editor, *Proceedings of OOPSLA'92, Vancouver, Canada. Special Issue of ACM SIGPLAN Notices (27)10*, pages 184–200, 1992.
35. R.B. Roberts and I.P. Goldstein. The FRL Manual. AI Memo 409, Artificial Intelligence Laboratory, MIT, Cambridge, Massachusetts, 1977.
36. R. Smith. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. *Proc. of the 1986 IEEE Computer Society Workshop on Visual Languages, Dallas, Texas*, pages 99–106, June 1986.
37. R.B. Smith and D. Ungar. Programming as an Experience: The Inspiration for Self. In Walter Olthoff, editor, *Proceedings of ECOOP'95, Aarhus, Denmark. Lecture Notes in Computer Science 952*, pages 303–330, Berlin, 1995. Springer-Verlag.
38. W.R. Smith. The Newton Application Architecture. In *Proceedings of the 39th IEEE Computer Society International Conference, San Francisco, California*, pages 156–161, 1994.
39. L.A. Stein, H. Lieberman, and D. Ungar. A Shared View of Sharing: The Treaty of Orlando. In W. Kim and F.H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pages 31–48. ACM Press/Addison-Wesley, Reading, Massachusetts, 1989.
40. P. Steyaert. *Open Design of Object-Oriented Languages. A Foundation for Specialisable Reflective Language Frameworks.* PhD thesis, Vrije Universiteit Brussel, Belgium, 1994.

41. A. Taivalsaari. Cloning Is Inheritance. Computer Science Report WP-18, University of Jyväskylä, Finland, 1991.
42. A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. PhD thesis, University of Jyväskylä, Finland, 1993.
43. A. Taivalsaari. Delegation Versus Delegation (or Cloning Is Inheritance Too). *ACM OOPS Messenger*, 6(3):20–49, 1995.
44. D. Ungar, C. Chambers, B.W. Chang, and U. Hølzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, 1991.
45. D. Ungar and R.B. Smith. Self: The Power of Simplicity. In N.K. Meyrowitz, editor, *Proceedings of OOPSLA'87, Orlando, Florida. Special Issue of ACM SIGPLAN Notices 22(12)*, pages 227–242, 1987. Also published in Lisp and Symbolic Computation 4(3), Kluwer Academic Publishers, pages 187–205, 1991.
46. P. Wegner. Dimensions of Object-Based Language Design. In *Proceedings of OOPSLA'87, Orlando, Florida. Special Issue of ACM SIGPLAN Notices 22(12)*, pages 168–182, 1987.