Classifying Prototype-based Programming Languages

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

[1] LIRMM — Universit de Montpellier II
161, rue Ada. 34392 Montpellier Cedex 5, France.
E-mail: dony@lirmm.fr, bardou@lirmm.fr
[2] UR–VALORIA — Universit 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 similar sets 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 means of modeling concepts, current languages promote methodologies reintroducing abstract constructions to manage groups of similar objects efficiently. We have proposed two classifications of delegation-based languages in order to clarify these issues. The first classification considers the primitives of the virtual machine underlying each language, and classifies languages according to the semantics of these primitives. The second classification considers the group-oriented constructions provided in each language, and classifies languages according to the level of abstractness of these constructions. The two classifications complement each other and other existing classifications. They allow people to assess the relative merits of the different languages more precisely.

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 prototype notion.Numerous prototype-based languages have been designed and implemented, including SELF [46, 1, 2, 13, 37], KEVO [41, 42], AGORA [40], GARNET [33, 34], MOOSTRAP [31, 32], CECIL [12], OMEGA [5], and NEWTON-SCRIPT [38]. Other languages, such as OBJECT-LISP [3] or YAFOOL [19] which are not strictly prototype-based, nevertheless offer closely related mechanisms or use prototypes at the implementation level. Languages 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 object equipped with attributes and methods, three primitives to create objects: creation "ex nihilo" (from scratch), cloning and extension (differential copy), and

one control structure: message sending[1]. Beyond this general characterization, these languages exhibit slight differences in the definition of their fundamental mechanisms that turn out to have a profound impact on their programming models. Such differences somewhat measure the relative strength of the influence coming either from frame languages used in knowledge representation or from actor languages used in distributed artificial intelligence. Furthermore, some differences naturally arise because individual prototype-based languages have been developed for different application domains, and with different goals in mind. Among them, three main goals emerge:

- A first goal was to provide simpler descriptions of objects. People naturally grasp new concepts 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).
- A 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 from the traditional abstract data type model to move towards a less constrained model. For these applications, classes have been considered as a source of complexity because they play 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, 46, 34].
- A third 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.

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 are various interpretations of what is a prototype, a concrete object or an average representative of a concept, and these interpretations lead to different languages.
- There are various interpretations for the semantics of the basic mechanisms (cloning, differential copy, delegation). In particular, differential copy creates inter-dependent objects and its various interpretations implies differences in the precise nature of the concept of object it implements.
- Finally, practical programming experiments have shown that some of the capabilities offered by classes, for example the ability to express sharing at a conceptual level, appear so important for program organization that they have been reintroduced in different ways.

[1] Generally embedding a delegation mechanism.

So, understanding a prototype-based language, both in terms of their expressive power and their applicability to specific kinds of problems, is not always easy. The Treaty of Orlando [39] proposed a first comparison of class-based and prototype-based languages. In this chapter, we address the semantics of prototype-based languages in more details. We have proposed two complementary classifications to this end. In this chapter we 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 paper is organized as follows. Section 2 reviews the genesis of prototype-based programming and provides a first set of definitions. Section 3 proposes a first classification of prototype-based languages according to the vision their designer had for them and gives other definitions. 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

Understand the genesis of ideas is a first important step in order to classify their resulting concepts and mechanisms appropriately. This section reviews how frame-based and actor languages have influenced prototype-based languages.

2.1 Prototypes and frames systems

Both the prototype and differential description concepts can be found in frame theory [30] and in some systems inspired by frame theory, in particular, the frame-based languages KRL [6] or FRL [35]. Frames were designed to represent knowledge such as typical values, default values, or exceptions, which are difficult to describe in other formalisms [29]. Frame-based languages use the prototype theory and have influenced prototype-based languages.

Structure. A frame is a set of attributes. Each attribute represents one characteristic of a frame as an "attribute name – set of facets" pair. The most common facet, and the only one considered in our examples, is the attribute's value. Figure 1 shows a definition of a frame representing a `whale`.

Differential description and parents. 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 relationship between the new frame and its protorype (that will be called later on " parent " in prototype-based

[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].

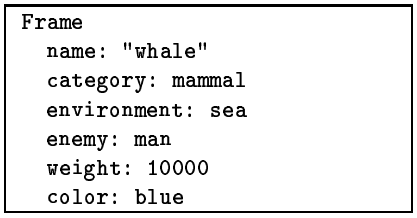

Fig. 1. Example of *Frame*

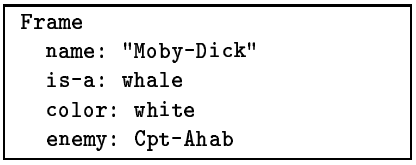

Fig. 2. Differential description

languages). This relationship is implemented by a link generally called "is-a"[3]. 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 relationship is an order relationship that defines frame hierarchies [9]. A frame can inherit a set of attributes from its parent. Frame-based systems have inheritance hierarchies made of representatives of concepts, which are conceptually very similar to those built later with prototype-based systems. Average representatives, such as `whale`, are found at the top of these inheritance hierarchies, and concrete representatives, such as `Moby-Dick`, are found at the bottom of these hierarchies. What is important about frame hierarchies is that what is shared are not descriptions, as with class inheritance, but rather concrete representations that is bindings of slots to values.

2.2 Actor languages

The actor language ACT 1 [23] also represents entities with classless objects [47]. ACT 1 provides objects and mechanisms conceptually close to those of frame systems described above. The main difference between the frame-based languages and ACT 1 is that ACT 1 is a programming (and not a representation) language. Actors thus have attributes (acquaintances) and behaviors. 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. 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 introduced cloning (shallow copying) as a primitive way to create new objects. The simple biological metaphor of cloning makes it very appealing, compared with the traditional way to create objects in class-based models, namely by instantiation. The basic idea is that, given an existing concrete object, it is easier to get a new object by copying the first one, rather than to instantiate a class correctly. Another primitive, called `create`, combines

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

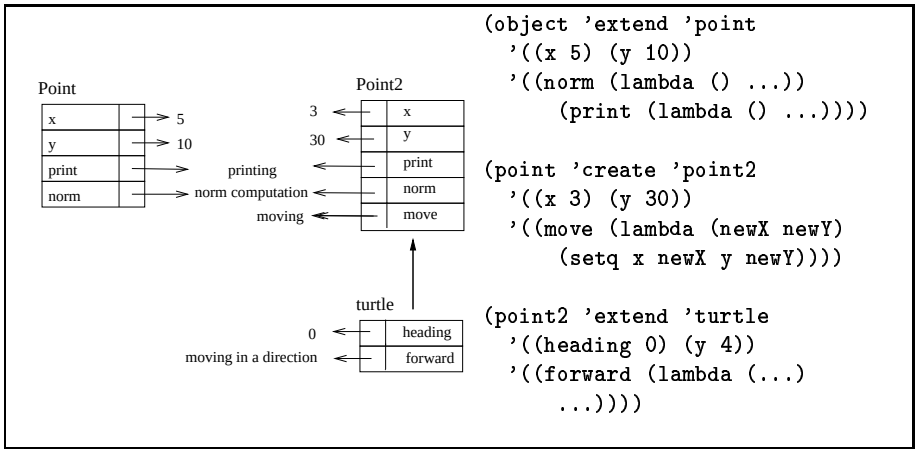

Fig. 3. Cloning and extension in ACT 1.

cloning and addition of new properties. For example, in Figure 3, `point2` is a clone of `point` with new attribute values and to which a new method has been added.

Extension. Differential description in ACT 1 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 parent (`proxy` in ACT 1 terminology). Figure 3 shows the definition of a Logo `turtle` actor as an extension of `point2`. A turtle is like a point but has a heading and a forward behavior to move in the direction defined by its heading. What we call extension is a new object linked to its should not be confused with the possibility to add new attributes to objects.

Inheritance. The relationship between an actor and its proxy is similar to the is-a relationship between frames. This relationship 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.

Delegation - act 1 version. Inheritance is used 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[4]. ACT 1 does not specify the context in which a method is executed after a delegation (the binding of self); this mechanism is however an early form of delegation found in later prototype-based languages.

[4] "*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

Frame systems and actor 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 were published in the middle of the 1980's. They brought to the fore several issues generally related to class-based programming and proposed various solutions based on the use of prototypes. This section reviews these early proposals and thus supports a classification based on the vision their designers had and the problems they wanted to solve.

3.1 Prototypical instances and cloning — A simpler programming model

Class-based program design is difficult [8, 22, 21, 39] because classes play a number of different roles: instance descriptors, method libraries, support for encapsulation sharing and reuse, support for the architecture of programs, etc. In reaction to this complexity, in a quest for a simpler programming model, Borning [8] proposed an informal description of a classless language in which new objects are produced by copying and modifying prototypes. Once the prototype is cloned, no relationship is maintained between the prototype and its clone. The resulting programming model is simple but quite poor in terms of sharing and in terms of program organization. The model has not been pursued by its author but has inspired some 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, Lieberman proposed to apply some ideas extracted from ACT 1 to object-oriented programming [24, 25]. He described an informal programming model based on prototypical instances, cloning and differential description. Many different languages have been inspired by his work including SELF, OBJECT-LISP, NEWTON-SCRIPT, MOOSTRAP, etc. SELF has attracted the largest development effort and it has been widely distributed.

A flavor of a prototype-based language : object-lisp. As an example of such languages, we can give the flavor of OBJECT-LISP [3], which based on LISP. Figure 4 shows an OBJECT-LISP version of the "point-turtle" example. Objects have attributes and methods; they can be cloned and extended. Extension objects have a parent and inherit their parent's properties. Computation is modelled as 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 not hold the required property.

Delegation and dynamic binding. Delegation can here be described informally in the following way: "an object that cannot answer a question can

```
(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 with objects, message passing and traditional function call. Message passing is implemented by the function* `ask`*; its first argument is the receiver and the second argument is 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.

delegate it to its parent, if the parent can answer it, the answer will be computed in the context of the original object". Compared to ACT 1, the fundamental new points are an explicit description of polymorphism and dynamic binding. In Figure 4, the method `norm` is polymorphic because it can be applied to a point or to a turtle. Dynamic binding ensures that a delegated method is applied in the context of the initial receiver (in Figure 4, 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 can have different structures and behaviors,
- exceptional objects,
- objects with viewpoints and sharing at the object level,
- incomplete objects can 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, is 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 consists of languages that integrates both prototypes and classes. Although 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 for embedding object hierarchies [22, 21] has combined both classes and instances (called examplars) having some kind of autonomy into a single world. This aimed to improve object-oriented program organization; it was meant to experiment with 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` can 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`. Any instance can inherit private methods from any other instance, to implement the methods composing its interface. The delegation hierarchy between instances is not necessarily isomorphic to the inheritance hierarchy between classes[5].

Although such proposals have not been pursued, they have influenced actual languages and represent a source of new ideas for the application of the prototype formalism [26, 4].

[5] 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

In this section we present a classification based on the primitive operations of prototype-based languages. Beyond a set of similar basic principles (object-based 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. We begin by presenting the criteria we will use to build the classification.

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 Properties are conceptually of two kinds: attributes or methods. There are two main ways to represent the properties of objects: (1) to separate attributes and methods or (2) to amalgamate them into slots. The first alternative mimics the class-based approach, as 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 is a collection of slots where a variable can be viewed as a pair of method that assign and return a value.

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

Both alternatives have advantages and disadvantages. The advantages of slots are advocated by SELF [46]: slots are more flexible because they allow users to access attributes and methods the same way, and they permit overriding of an attribute with a method and conversely a method with an attribute. Besides, slots force the language to implement an explicit encapsulation mechanism. With the variables and methods approach, encapsulation of variables can be simply enforced, by establishing SMALLTALK-like visibility rules.

4.2 Criteria related to object creation and evolution

Objects in prototype-based languages can be created in two ways: an object can be created *ex nihilo* or it can be created from an existing one (by cloning or extending it).

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

If a primitive to create new objects ex nihilo is provided, two further alternatives arise. Either the primitive can create empty objects, or it can create objects with an initial structure. If the primitive create empty objects, some means must be provided to modify their structure in order to build the concrete objects needed by applications.

Dynamic modification of object structure : can objects' 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.

Cloning : is a cloning primitive available in the language.

The second solution is extension and is discussed in the next section.

4.3 Inheritance and 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 [42] does not support extensions in our sense of the term, but does 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. In Figure 3, 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]. 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 objects through their respective slots `print`, `x` and `y`. Even if they get the same structure and the same values, they have independent slot bindings. For example, 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. Shallow cloning enforces what we call "creation-time sharing" and which is characterized by an independent evolution of the clone and the prototype. Properties values are shared between an object and its clone as long as values are not modified in one the objects. This prevents objects from being unexpectedly modified through their clones. The independent evolution applies to slot individually; for example `point` and `point2` will continue to share the method `move`, even after `point2's` `y` slot has been modified.

In contrast, extension enforces what we call "life-time sharing": the extension and its parents can continue to share properties values even if the properties of the parent are modified. In other words, the extension share the properties of its parent and the sharing will continue as long as the objects exist. In Figure 4, `turtle` will continue to share the `y` slot of `point2` even if this slot's value is modified. If a new property is added to `point2`, `turtle` will inherit it.

Cloning and extension are thus different and have distinct applications.

Extension : Does the language provide an extension mechanisms?

When no extension mechanism is provided, life-time sharing between objects requires other "group-wide mechanisms" to automatically handle clone families in the system. We call "clone family" of an object `o`, the transitive closure of the relation "is-a-clone-of(o)". Some languages such as KEVO provide a mechanism called *propagation* allowing to add a new method to all objects of the same clone family [43]. This makes it possible, for example, to add in a single operation a new method to all points in the system but also to all turtles, provided that the first turtle `t` has been created by cloning a point and by adding new properties to `t`.

Propagation : if the language provides no extension mechanism, is there a propagation mechanism supporting life-time sharing?

When object extension is provided, some languages allow new objects to be extensions of more than one object. There is no conceptual difficulty with multiple extension, it raises similar problems to multiple-inheritance 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 parent modification : is it possible for an object to change its parent?

4.4 Message sending and delegation

The basic control structure of prototype-based languages is message sending as in other object-oriented languages. 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. Most prototype-based languages provide a sharing mechanism named delegation. Delegation can be implicit or explicit.

Implicit delegation is used with a corresponding extension mechanism. Implicit delegation is a kind of inheritance which gives an object access to its parents properties. 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 hold the requested method or attribute, the interpreter automatically delegates the question to the object pointed to by its parent link.

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

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

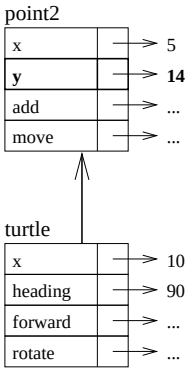

Fig. 5. after the modification of `y` with property sharing.

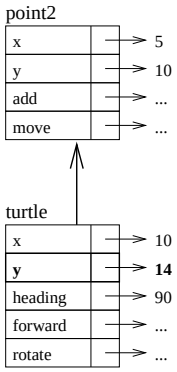

Fig. 6. after the modification of `y` with value sharing.

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, 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 representations of different entities In the first interpretation, that we will call *value sharing interpretation*, the notion of extension is used to express the sharing of attribute values and methods between two objects representing different domain entities. For example, consider again in 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 the reuse of the definition of `point2`'s properties. We do not want `point2`'s properties to be the actual properties of `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, and each of these properties are identified by the same name within the context of any of the two objects and has the same default value [24]. 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

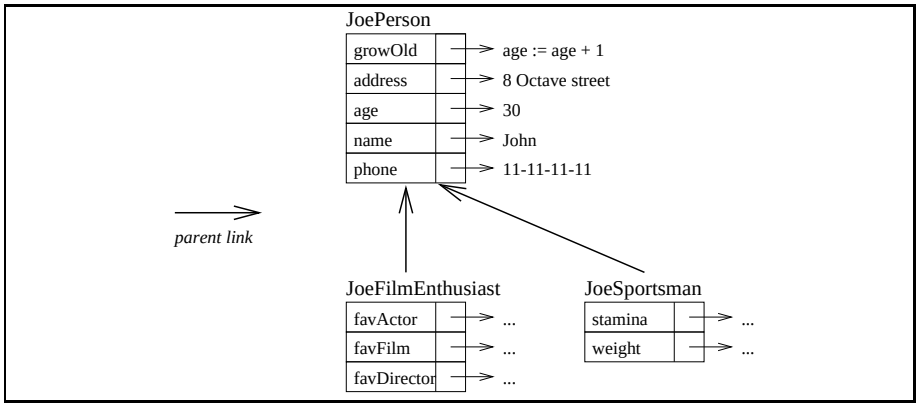

Fig. 7. A representation of a person with three objects.

`point2`'s `y` value being modified 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 cause 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 not write access to parent's properties. This makes the frontiers between objects clearer: in Figure 6 `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 restricting property sharing between an extension and its parent to value sharing. In the example, the object `turtle` does not share with `point2` the property named `y` but rather the value of that property as long as it is not redefined on `turtle`.

Delegation as a sharing mechanism between viewpoints on the same entity. In a second interpretation, that we will call "*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 Joe (his `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 facilitate subsequent extensions, e.g. Joe as a film enthusiast. Any modifications to `JoePerson` are automatically seen by its extensions. Also, changes to Joe can be made through its extension objects. For example, if Joe 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 the leaves of the hierarchy. In our example, person is a more general viewpoint on *Joe* than either sportsman or film enthusiast, so `JoePerson` is the top of the hierarchy.

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 must change the attribute `age` of `JoePerson`.

In terms of sharing, this interpretation is an application of property sharing between objects. The `address` and `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 provides an extension mechanism, is it "property sharing" or "value sharing"?

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

The "property-sharing" interpretation raises some encapsulation issues when used in a "point-turtle" like example. Delegation establishes a so strong link between the objects `point2` and `turtle` that they can no longer be considered as representing distinct entities. Conversely, delegation 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 no way to handle split representations as first-class objects, entities for which we have coined the name *split objects*, in today's prototype-based languages. Making split objects first-class in a language would mean being able 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].

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

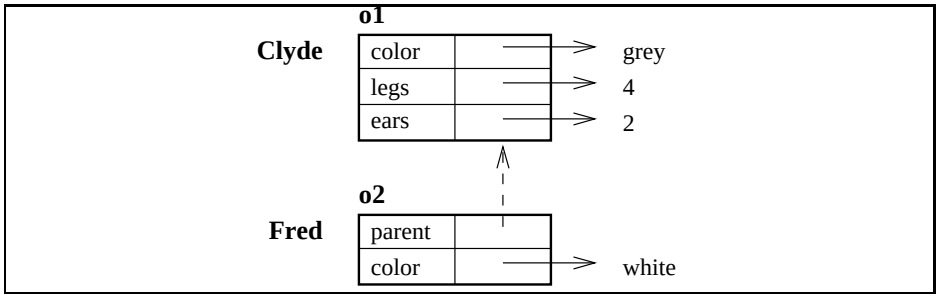

Fig. 8. Differential creation of the prototype `o2` representing Fred from the prototypical instance of elephants also representing Clyde.

5 Classification criteria based on program organization

Prototype-based programming advocates concrete objects as the only means to model application domain entities. But, current languages have mechanisms to organize programs into groups of similar objects, despite their basic principles. The two most common ones are prototypical instances and traits objects. This trend acknowledges an organizational gap in prototype-based programming. Other alternatives to tackle this problem have also been proposed, such as KEVO's clone families [42] and the related map mechanism of SELF.

Early in the foundation of prototype-based programming, Lieberman proposed using prototypical instances and delegation to share properties among similar objects. 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`.

In Lieberman's approach, the prototypical instance status of an object is not taken into account in the model, and this may lead to semantic inconsistensies. In our example, modifications to the individual object Clyde, such as losing a leg, will propagate to all its delegating objects, which still delegate to Clyde even though it is no longer a prototypical elephant. To overcome this kind of inconsistensies, a distinction must be made between the individual Clyde and the prototypical instance of elephant.

Traits and maps have been invented in the prototype-based language SELF [45]. A traits object is a repository for methods and shared variables applying to a group of similar objects. These objects all delegate to the traits object. This "trait-based" programming model [45] segregates object's slots in two parts: the representation part and the protocol part. Typically, the protocol part consists of slots holding method values while the representation part consists of slots

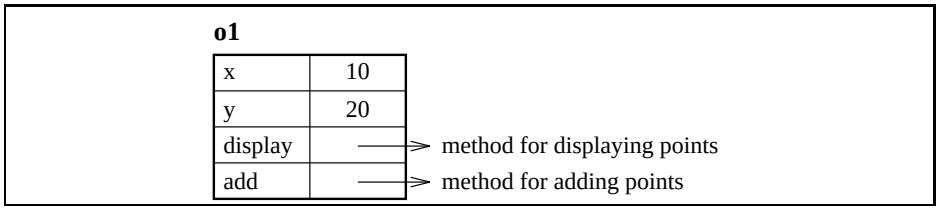

Fig. 9. A point example.

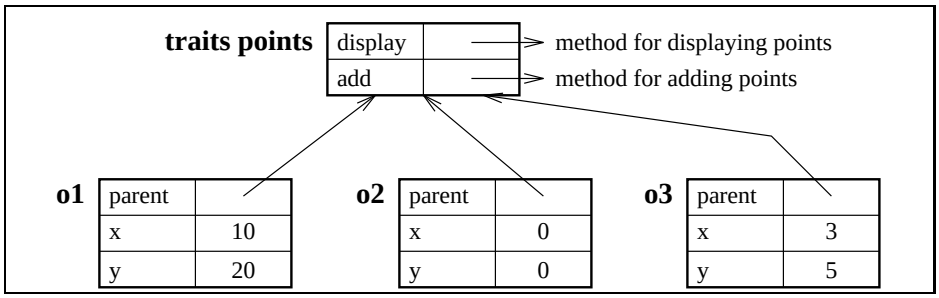

Fig. 10. A point example using a traits object.

holding object identifiers, but this need not to always be the case. In Figure 9, the representation part of a point object includes the slots `x` and `y`, while the protocol part includes 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 inheritance hierarchies in their highest levels are made of traits. At the leaves, we find concrete objects, 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` can be provided, which returns the first ancestor of an object that is a trait; testing whether two objects support the same protocol then reduces to testing 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. 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 this by storing the values of immutable slots into the map. 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 is returned; if a method is found, it is applied in the context of the receiver; if an index n is found, the

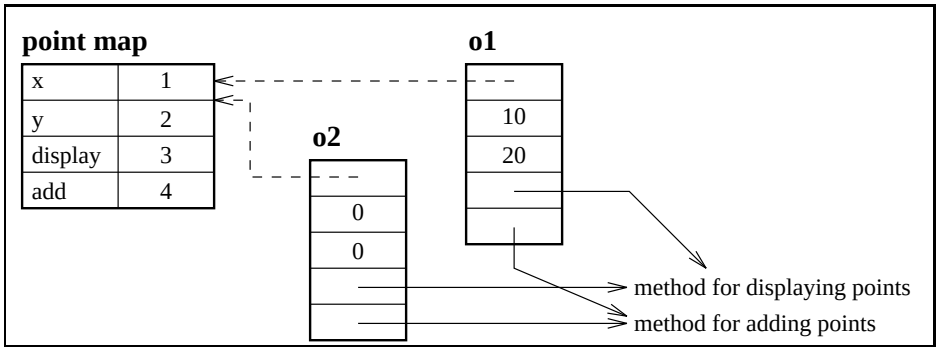

Fig. 11. The point example using maps.

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 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 objects in SELF, they only have an indirect influence on its programming methodology. By making cloning more efficient than creating objects from scratch, they emphasize program designs based on clone families. In order to provide a full-fledged programming methodology, we suggest they could be made first-class entities, with the advantage that the corresponding programming model would be able to express itself without resorting to external hidden mechanisms. KEVO already provides this kind of functionality through the concept of clone families and its related mechanisms.

5.1 Generalization

Prototypical instances, traits and maps are specific mechanisms used in existing languages to structure programs, but they appear to detract from the original objectives of prototype-based programming [26]. But from them emerges a notion of *delegation-based programming* much richer than pure prototype-based programming and where the organizational gap of pure prototypes can be overcome without going back to the standard class-based model. Delegation-based languages exhibit much more diversity than may appear, because these new mechanisms support slightly different programming models, which can hardly be put under the same 'prototype-based" hat. We take the point of view that concrete objects with delegation are the salient features of delegation-based programming languages, and therefore we propose a more general notion of object-centered programming to be contrasted with traditional class-centered (resp. abstraction-centered) programming:

object-centered programming: a programming paradigm where the main activity during program design revolves around the creation of concrete objects.

class-centered (resp. abstraction-centered) programming: a programming paradigm where the main activity during program design revolves around the creation of classes (resp. abstractions).

The important point here is that designing an object-centered program really implies the creation of concrete objects as the major activity. In this sense, abstractions like traits and maps are to object-centered programming what assignments are to impure functional programing languages like Scheme and ML: an orthogonal feature which nevertheless leaves the programming model dominated by the major paradigm.

Considering prototypical instances, traits and maps as explicit organisation mechanisms, allows us to complement existing classifications, such as the one presented in the previous section, but also those of Wegner [47] and the Treaty of Orlando [39]. Wegner identifies our object-centered programming languages as classless object-based languages, within which he singles out delegation-based languages, i.e. "*classless objects with delegation*".In our point of view, Wegner's classification underestimates the diversity of classless object-based languages, because it attempts to characterize the whole design space of object-oriented language. Moreover, at the time of his writing, delegation-based languages were under-investigated. In this sense, our approach allows us to complement his classification, being more precise in one of his original class.

We classify delegation-based programming languages according to the number of different kinds of objects and the number of different kinds of links they manipulate (except the basic "value-of" of "points-to" link). For example, delegation's parent link is manipulated in all delegation-based programming languages. A trait-based programming language manipulates two kinds of objects: concrete ones and traits. The kinds of objects and the kinds of links manipulated in a language bears important insights into the nature of its programming model. We explore below four main classes of delegation-based languages: languages with one kind of object and one kind of link, those with two kinds of objects and one kind of link, those with two kinds of objects and two kinds of links and finally those 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].

The general criticism opposed to prototype-based programming languages is their lack of manifest organization in programs. Our new classification highlights the existence of delegation-based languages with progressively more structured programming models, forming a continuum between pure prototype-based languages and class-based ones. This shows not only that the organization of a program can be made more explicit without sacrificing an object-centered programming style, it also suggests a step by step (possibly automatic) transformation of prototype-based programs into class-based ones [47, 39].

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

We classify typical delegation-based languages as languages with one kind of object and one kind of link, namely the parent-of link. In these languages, the space of objects is completely homogeneous and delegation is used for sharing. 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.

Two kinds of objects, one kind of link

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. Some objects in a program can be distinguished from standard ones because they play a specific role in a particular programming methodology that must be explicitely supported at the language level in order to provide its full benefits. They may also be distinguished when their "first-classness" is severely restricted, such as being immutable or abstract (in the sense that they cannot answer messages). These distinct objects may not be cloned or used as parents for example. We illustrate this category of languages with a trait-based programming language.

The trait-based programming model is a good example of this class of languages, having two kinds of objects, prototypes and traits, and one kind of link, delegation. In prototype-based programming all objects are assumed to be concrete, and to be able to respond to messages, whereas in trait-based programming, messages sent to a traits object will fail because they cannot access slots which are stored in descendants. For trait-based programming to work properly, most traits objects must never receive messages. This restriction can be enforced in a language with two kinds of objects.

Two kinds of objects, two kinds of links

Languages with one kind of link will have a delegation link, which implements sharing. Adding a second kind of link suggests introducing a structural description link similar to the class-of link. In prototype-based languages each object, being one-of-a-kind, must store its slot names and slot values. When a large number of structurally identical objects are created, the structural information can be shared, rather than replicated, as in SELF *maps* [14].

A map-based language is constructed by making maps first-class objects. Because their slots contain indices to be reinterpreted in the context of the receiver, first-class maps cannot answer messages. Like traits, maps should be abstract objects unable to answer messages. With first-class maps, 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

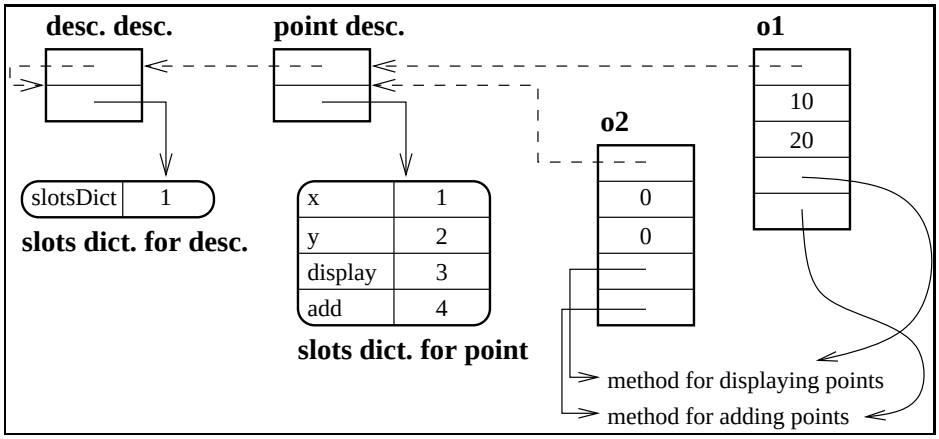

Fig. 12. The point example using descriptors.

speak about a group of structurally identical objects. Map-based operations to add or delete a slot to all objects in a clone family should be implemented to give a better idea of the supported programming methodology.

One kind of object, two kinds of links

Typically, a language with one kind of object and two kinds of links can be obtained by some rationalization of a language with two kinds of objects (and two kinds of links). For example, consider our map-based language. Maps can be transformed in order to make them standard objects capable of answering messages without impairing the programming model. Figure 12 shows how maps can be replaced by descriptor objects with one slot called `slotsDict`, which points to an object implementing a slot dictionary. The slot dictionary is not an ordinary prototype, but rather an object similar to SMALLTALK's method dictionary. A descriptor is an object answering `at:`⟨*some selector*⟩ returning its associated value and `at:`⟨*some selector*⟩ `put:`⟨*some value*⟩ messages like a SMALLTALK dictionary. We draw it as a box with round corners in Figure 12.

The advantage of this representation is that we can send legitimate messages to descriptors as well as to slot dictionaries. In fact, our descriptors look pretty much like SMALLTALK classes. A descriptor-based language similar to the previous map-based language can be designed very simply. In order to preserve an object-centered programming model, descriptors should be created automatically, like maps. Nonetheless, descriptors are so much like classes that descriptor-based languages are at the frontier between abstraction-centered and object-centered programming. Our descriptor-based language still promote an object-centered programming model because it lacks 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 encourage program-

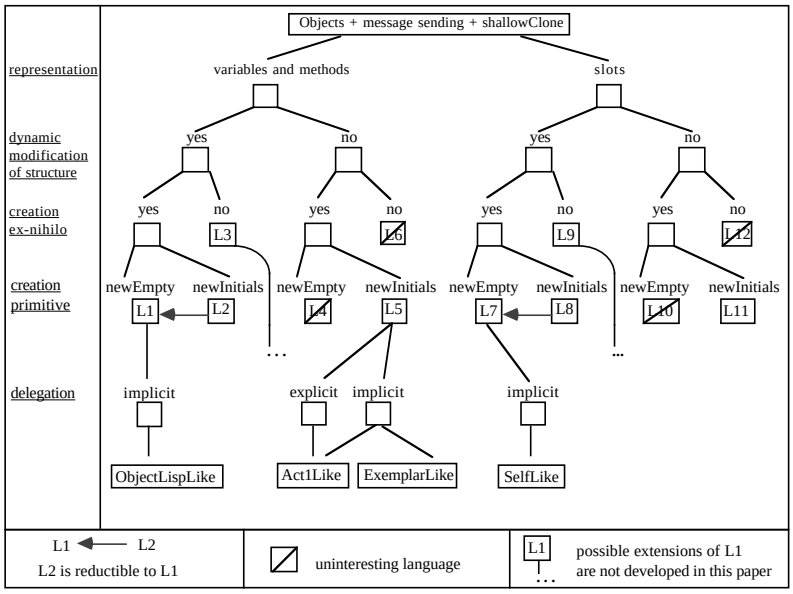

Fig. 13. A historical taxonomy [17].

mers to design their applications in an object-centered style (see [28] for more information about descriptors).

5.2 But what's in a link?

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 the 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, because Amulet's four modes are essentially delegation links.

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

6 Classifying Existing Languages

6.1 First classification based on various primitives mechanisms

The first taxonomy shown in Figure 13 is extracted from [17] and is only shown here to give a historical perspective that reflects our understanding of prototype-

based languages at that time[6]. 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 [22].

SELF and OBJECT-LISP are members of the language families (L8) and (L2) respectively, both extended with implicit delegation. From the point of view of this first taxonomy, SELF and OBJECT-LISP 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. EXAMPLARS is a hybrid language also providing classes, so some of its characteristics are out of the scope of our taxonomy. ACT 1, described in Section 2.2, has 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, and the parent's script is then executed. ACT 1 can be classified as a version 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 primitive mechanisms

This section now proposes a second classification taking into account the whole set of criteria presented in Section 4. This classification is summarized in a comparative table (cf. Fig. 14). Since 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 support both by allowing the creation of the two kinds of extensions implemented by two links named `_proto` and `_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. Mixed solutions are also possible, but they raise the issue of managing both kind of delegation links. For what concerns specific mechanisms, let us quote what we have called *multiple-cloning*

[6] This classification is also an operational one that was 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 people to rapidly implement a simulation of any prototype-based language [18].

	Self	Object Lisp	Garnet (KR)	Amulet (ORE)	Agora	Moostrap
Distinction between variables and methods	no	yes	no	no	yes	no (slots)
Creation ex nihilo	yes	no	yes	no	no	yes
Dynamic modification of object structure	yes	yes	yes	yes	no (possible at meta level)	yes
Cloning	yes	yes	no	no	yes	yes
Extension mechanism	yes	yes	yes	yes	yes	optional
Propagation mechanism	no	no	no	no	no	no
Single/multiple parents	multiple	single	multiple	simple	simple (mixins)	simple or multiple
Dynamic parent modification	yes		yes		no	yes
Interpretation of extension mechanism	property sharing	property sharing	value sharing	4 per slot different kinds of sharing	encapsulated inheritance	property sharing
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
Distinction between variables and methods	no	yes	yes	no	no
Creation ex nihilo	yes	no	no	yes	yes
Dynamic modification of object structure	yes	yes	yes for prototypes no for others	no	yes
Cloning	yes	yes	yes	yes (multiple)	yes
Extension mechanism	yes	no	no	no	yes
Propagation mechanism	no	yes	yes	no	no
Single/multiple parents	double	-	-	-	multiple
Dynamic parent modification	yes	-	-	-	yes
Interpretation of extension mechanism	property sharing and value sharing		type-like sharing		value sharing
Other language characteristics	inheritance hierarchy and ROM-defined prototypes	composition hierarchy and clone families	type hierarchy	multiple cloning, aliases and distributed programming	Models and instances ...

Fig. 14. Language comparison.

6.3 Abstraction-based Classification

Figure 15 summarizes a third 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 view, 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

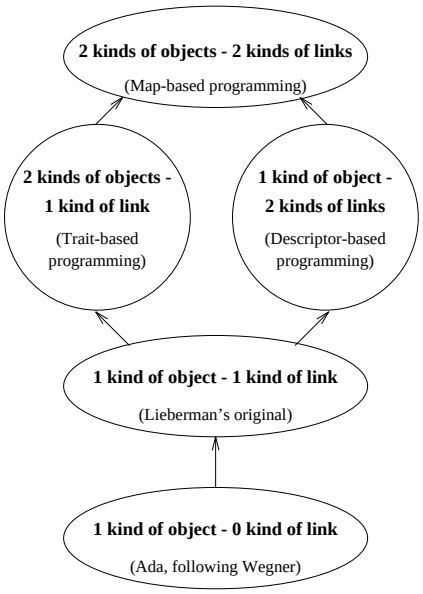

Fig. 15. The second classification with example languages.

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 this kind of language 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 be restricted to object-centered languages. We have 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 would produce a language with one kind of object but three kinds of links (parent-of, descriptor-of and descriptor inheritance) which cannot be characterized as object-centered. Moreover, consider a language where meta-classes are first-class objects as Cointe's ObjVlisp [Coi87]. Such a language has two kinds of objects (instances and classes, since meta-classes are simply classes whose instances are classes) and two kinds of links (instantiation and inheritance), but it is certainly not object-centered. We have used our classification to characterize object-centered programming languages, but it could also be used to classify other kinds of languages.

It is interesting to classify existing languages in the light using this scheme. 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 as presented here, the design of SELF can be positively enhanced. Our preferred path to upgrade it 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, where the traits-based programming methodology could be safely used.

7 Perspectives

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

Our work has also opened some new research perspectives. The explicit definition of the kinds of sharing, life-time and creation-time sharing, achieved by cloning and delegation respectively has shown that the two mechanisms are not reducible to each other. It has also led us to identify the self problem, which occurs when frontiers between objects are blurred by the sharing of slot bindings between them. In turn, the self problem has led us to identify the major use of delegation, which is to create split representations of domain entity using first-class split objects to preserve strict object identity and encapsulation. 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 to represent viewpoints or roles and in object-oriented databases or other persistent applications.

A second perspective is opened by our second classification. We have introduced a general notion of object-centered programming, which admits some kinds of abstract objects, 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 capable of structuring programs but different from classes, we have reconciled prototypes and abstractions. Consequently, we have brought to the fore the existence of progressively more structured delegation-based languages forming a continuum between pure prototype-based languages and class-based ones.

Proponents of the prototype-based approach have suggested software development methodologies evolving from liberal designs, using prototypes, towards 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 conviction 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.

[Coi87] P. Cointe. Meta-classes are First Class: the ObjVlisp Model. In *Proceedings of OOPSLA'87*, pages 156–167, Orlando, Florida, December 1987. ACM Sigplan Notices.

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. A. Taivalsaari. On the Notion of Inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.
45. D. Ungar, C. Chambers, B.W. Chang, and U. Hølzle. Organizing Programs Without Classes. *Lisp and Symbolic Computation*, 4(3):223–242, 1991.
46. 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.
47. 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.