

# Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation

Christophe Dony\*, Jacques Malenfant\*\* and Pierre Cointe\*\*\*

LITP (IBP) Université Paris-VI

## Abstract

Prototype-based languages are currently proposed as a substitute to class-based languages for a higher flexibility in manipulating objects. These languages are all based on a similar set of basic principles: object-centered representation, dynamic addition (deletion) of slots, cloning and message delegation. But they all differ in the precise interpretation of these principles and nobody has yet considered the semantic issues raised by their combination. In this paper, we propose a new taxonomy of prototype-based languages, enhancing the Treaty of Orlando by now discussing issues associated with the different semantics of the identified prototype-based languages. From this taxonomy, we extract a constructive proposal for the design of a new prototype-based language. This proposal is the chief result of this paper; it suggests one set of primitives which is regarded as the best to provide a clean, useful and coherent prototype-based computational model. We finally describe an implementation of most interesting language alternatives in the form of a Smalltalk-80 platform. This platform establishes an operational semantics for the basic primitives and – more interesting – validates our previous taxonomy by implementing it as a class hierarchy. Obviously, this platform has been used to relate in the same formalism the semantics of different languages with each others. For instance, the programming models of existing languages, such as Self, ObjectLisp and Actra's examplars, are faithfully derived as subclasses in this hierarchy.

## 1. Introduction

In object-oriented programming, a long road lead from class-based to prototype-based languages. Experimental work about knowledge representation in the AI community has shown the usefulness of frames for their capability to represent different types of knowledge (values, procedures, predicates, defaults) as a collection of slots. On the other hand, the quest for new computational models for distributed AI have led to the development of the actor model emphasizing the cloning and delegation mechanisms. In parallel with this evolution, many users of class-based languages developing applications in the fields of user-interfaces [MGDV90] and virtual reality systems [Born81, Smit86], have tried to escape the traditional abstract data type model to move towards a less constrained one. For such kind of applications, classes have also been considered as a source of complexity because they are playing too many roles [Born86].

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 [Born86, MACL89, UnSm87, MyGV92]. Prototype-based languages propose a new programming paradigm that is justified in two fundamental ways

---

The authors present addresses are the following:

\* LITP, 4 place Jussieu, 75252 Paris Cedex 05. Email: chd@rxf.ibp.fr

\*\* Université de Montréal, Département d'informatique et de recherche opérationnelle, C.P. 6128, Succursale A, Montréal, Québec, Canada, H3C 3J7. Email: malenfant@iro.umontreal.ca

\*\*\* École des Mines de Nantes, 3 rue Marcel Sembat, 44049 Nantes Cedex 04, France. Email: pc@rxf.ibp.fr.

compared to class-based languages. First, on the philosophical side, 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). Second, on the more pragmatic side, class-based languages seem to unnecessarily constrain objects, by disallowing distinctive behavior for individual objects among their instances and by forbidding inheritance between objects to share values of instance variables.

Recently, many languages using prototypes have been proposed. Borning derived a small prototype-based language to compare them to classes [Born86]. Lieberman [Lieb86] gave an informal description of the delegation system, from which he argued that prototypes are strictly more powerful than classes. Self [UnSm87] is a pure prototype-based language efficiently implemented [ChUL89, ChUn91]. Systems mixing prototypes and classes have also been proposed [LaTP86, Lalo89].

Nevertheless if we consider this recent evolution of prototypes, current prototype-based languages differ in the semantics of object representation, object creation, object encapsulation, object activation and object inheritance. So understanding the exact merits of each language is not always easy, both in terms of expressive power and applicability to specific kinds of problems. The Treaty of Orlando [StLU88] proposed a first comparison of class-based and prototype-based languages; we go further by addressing more extensively the problem of the alternative semantics associated to pure prototype-based languages.

More precisely, this paper has three goals:

- 1) to build a new taxonomy for prototype-based languages,
- 2) to use this taxonomy to propose our model of a prototype-based language and
- 3) to validate this taxonomy by implementing it as a (Smalltalk-80) class hierarchy.

To reach these goals, the methodology we used is essentially experimental. We have collected language proposals from several papers, identified

their primitives and classified their alternative semantics. To fully understand the semantics of primitive principles, we have implemented and experimented with them; this led to the implementation of ProtoTalk. From this experimentation, we collected observations that helped in expanding and building our taxonomy. So the structure of our argumentation is the result of a process of clarification of ideas coming from these observations. We have organized mechanisms in increasing order of complexity; this order is reflected in the construction of our taxonomy.

The paper is organized as follows. Section 2 recalls the basic principles of prototype-based languages and defines shortly their terminology. Section 3 studies the different semantics associated to prototype-based languages, introduces our taxonomy and proposes our model. Section 4 presents the platform Prototalk – Prototypes in Smalltalk – implementing the previous taxonomy as a Smalltalk class hierarchy. Then we conclude on future research work.

## 2. Terminology and Basic Issues

This section introduces the main principles of the prototype-based approach and points out its basic issues.

### 2.1 Prototypes

"If Clyde was the elephant most familiar to you, the prototypical elephant might be an image of Clyde himself. If I ask a question as "How many legs does an elephant have?", a way to answer the question is to assume that the answer is the same as how many legs Clyde has, unless there is a good reason to think otherwise." [Lieb86]

The most important idea of the prototype-based approach is that **concrete objects are the only mean to model applications**. There are no classes, as usual in object-oriented programming. Prototypes are not meant to be abstractions, as classes are, and they are not linked in any way to other objects that would describe them, as in the class-based approach. A prototype is a self-sufficient entity with its own state and behavior, and capable of answering messages. This means that the normal way to speak

about a concept in these languages is to provide a concrete example for this concept. This has tremendous importance as we shall see later; but note here that any reference to a concept must first be rephrased in terms of its concrete examples, as pointed out by Lieberman.

## 2.2 Delegation

Delegation was introduced [Lieb81] as a message forwarding mechanism. The basic idea of delegation is to forward messages that cannot be handled by an object to another object called its parent in Self or proxy in Act1 (we will use the term "parent" in the rest of the paper). Indeed, the key-point of delegation is that the pseudo-variable "self" still points to the original receiver of the message, even if the method used to answer the message is found in one of its parent [Lieb86]. Delegation is proposed as a mean for an object to retrieve and share knowledge provided by another object. Let's use Clyde and the elephant example again. Assume an elephant Fred; delegation means that you can construct an object representation of Fred that is related to Clyde in such a way that if I ask a question as "How many legs does Fred have?", then the way to answer this question is to assume that the answer is the same as how many legs Clyde has, unless we know Fred to have a different number of legs. Hence, Clyde acts as a prototypical instance, not only to answer questions about the concept of elephant but also to answer questions about specific elephants. This use of delegation is proposed as an alternative to class inheritance.

## 2.3 The Meaning of Self

"To create an object that shares knowledge with a prototype, you construct an extension object, which has a list containing its prototypes, which may be shared by other objects, and personal behavior idiosyncratic to the object itself." [Lieb86]

More than message forwarding, delegation can also be interpreted as an **extension** mechanism. An object B that delegates to another object A, can be viewed as an **extension** of A. Hence, delegation defines objects having a shared part and a private part. This interpretation of delegation raises the important issue of the meaning of "self". The most

important point here is that **sharing is done at the level of concrete objects** and not at the level of concepts as with class inheritance; this means that structures, behavior and values are shared. This fact also has a tremendous importance in prototype-based programming models. With delegation, the notion of "self" or in other words, what means to be inside or outside an object, must be reconsidered in the light of this object extension view. What does it mean to be inside or outside of the extension object B? or to be inside or outside of the extended object A? How well do the different prototype models address these issues? These are crucial questions we will address in the next section.

## 3. Semantics of Prototype-Based Languages

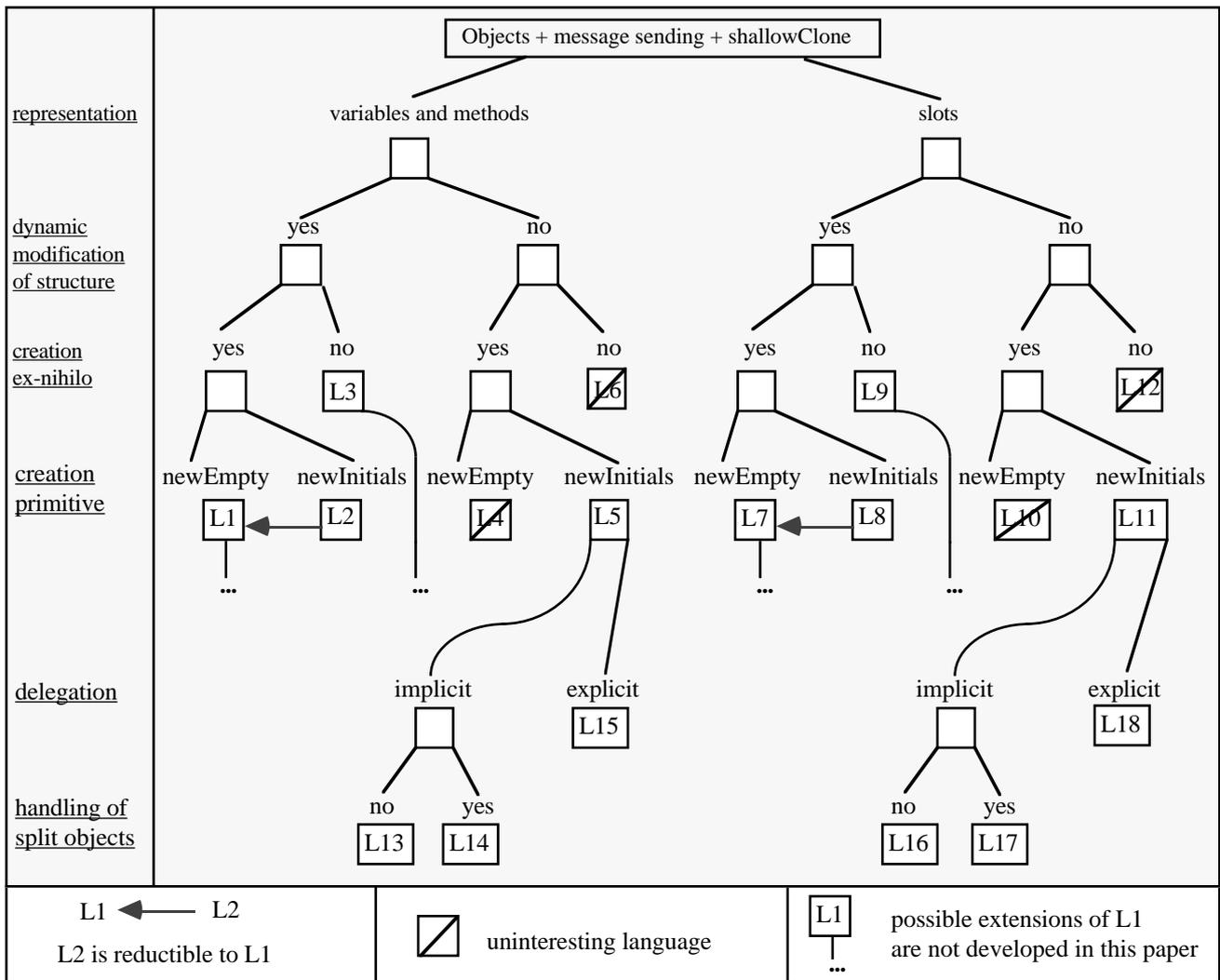
This section describes our taxonomy, uses it to provide a typology of existing languages and proposes our model of a prototype-based language.

### 3.1 Taxonomy

Most prototype-based languages provide a message passing mechanism to activate objects and a cloning primitive. But, they differ in the semantics of other basic mechanisms. We now classify these alternatives to get our taxonomy, which is shown in Figure 1. Each leaf of this taxonomic tree corresponds to one basic family of prototype-based languages built on a self-sufficient set of primitives and mechanisms.

#### 3.1.1 Object Representation

Two main solutions can be considered for the representation of objects in prototype-based systems: (1) to separate the concepts of object variables and object methods or (2) to amalgamate them using the concept of slot. The first alternative mimics objects in a traditional class-based system. In the second alternative, no distinction is made between variables and methods; instead an object gets a collection of slots where a variable is represented as a method that returns a constant value. The first level of our taxonomy distinguishes between these two alternatives in object representation. The advantages



**Figure 1. Our taxonomy of prototype-based programming languages (to ease the understanding of the figure, we hide part of the lowest levels).**

of slots are advocated by Self [UnSm87]. But in the variables&methods approach, encapsulation of variables is enforced by standard (Smalltalk-like) visibility rules, whereas new mechanisms must be provided to restrict the visibility of slots in the other approach.

### 3.1.2 Object Creation and Evolution

Two mechanisms are proposed to create new objects in current languages. An object can be created ex nihilo, using an appropriate primitive, or it can be created from an existing one by using the cloning primitive.

Let's first look at the creation of objects ex nihilo. Whether or not a primitive to create new objects ex nihilo is provided makes up the two alternatives of

the third level in the taxonomy. If a primitive to create new objects ex nihilo is provided, two new alternatives show up. We can create empty objects (let us call this primitive *newEmpty*), or objects with an initial structure (let us call this one *newInitials*:). These alternatives appear at the fourth level of the taxonomy.

But if we restrict ourselves to the creation of new empty objects alone, this raises the question of what to do with these new empty objects? Some means must be provided to modify their structure in order to build the concrete objects in applications. Indeed, this introduces two other design alternatives which are whether or not the structure of objects can be extended and/or shrunk dynamically, by adding or deleting slots (let's call these primitives *addSlot*: and

*deleteSlot*;) or respectively variables and methods (let's call them *addVar*:, *deleteVar*:, *addMethod*: and *deleteMethod*:). For a better classification, we made these alternatives as the second level of our taxonomy.

Let's now consider cloning. 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 object, it is easy to get a new object similar to the first one by simply copying it. Cloning 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. Because most languages provide a primitive for shallow cloning that we call *shallowClone*, we introduced it in the root of the taxonomy.

### 3.1.3 Alternatives in the Semantics of Delegation

The two main alternatives of delegation are implicit or explicit delegation. In implicit delegation, when an object cannot answer a message, the interpreter automatically delegates it to another object; objects have a parent link to indicate to the interpreter to which objects messages should be delegated. In explicit delegation, on the other hand, the delegation of messages is done explicitly for each message to be delegated; the delegating object names the object to which the message has to be delegated. Ways to express that in the languages are discussed in Section 4, but explicitly delegating a message resembles externally to message passing, except that the the method invoked by the explicit delegation is executed in the context of the delegating object. These two alternatives appear at the fifth level in the taxonomy.

### 3.1.4 Alternatives in the handling of split objects

In Section 2.3, we have mentioned that the extension view of delegation raises very important issues like the notion of "self" for both extended and extension objects. In fact, we will argue in the rest of the paper that delegation introduces the concept of

split objects composed in this case of both the extended object and the extension object (we will defined this concept more precisely in Section 3.2.4).

Therefore, we introduce a distinction between languages that explicitly deal with split objects and languages that don't. These two alternatives make the sixth level in our taxonomy. Handling split objects in a language means that we can create them, refer to them, clone them and otherwise deal with them as with other objects. In other words, handling split objects means to treat them as first-class entities in the language.

### 3.1.5 Classification of Existing Languages

One goal of a taxonomy is to classify existing entities. At the moment, we have looked at five existing prototype-based languages: Self, ObjectLisp, Act1, Exemplars and Garnet.

Self and ObjectLisp are members of the language families (L8) and (L2) respectively, both extended with implicit delegation but without support for split objects. From the point of view of our taxonomy, they only differ in the representation of objects; Self uses slots while ObjectLisp uses variables and methods.

Exemplars 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 also providing classes, some of its characteristics are out of the scope of our taxonomy.

Act1 is an actor-based language and has very specific characteristics. Objects can have variables and one method named their 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 (here called the proxy), the script of which is in turn executed. These functionalities can be classified in an extension of the language (L5) for which the evaluator is also

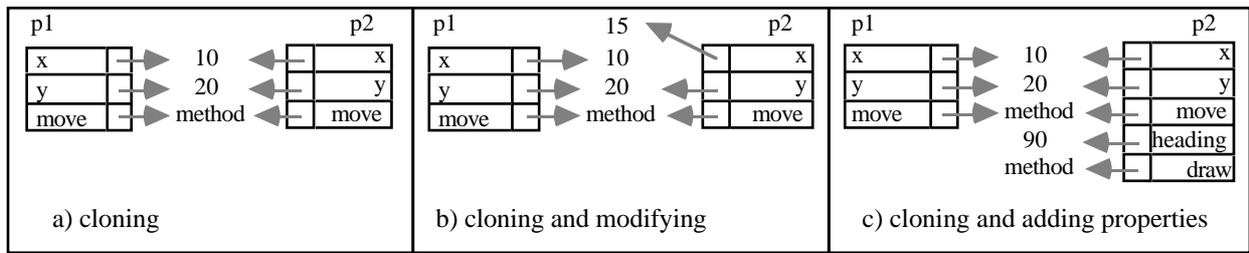


Figure 2: Cloning

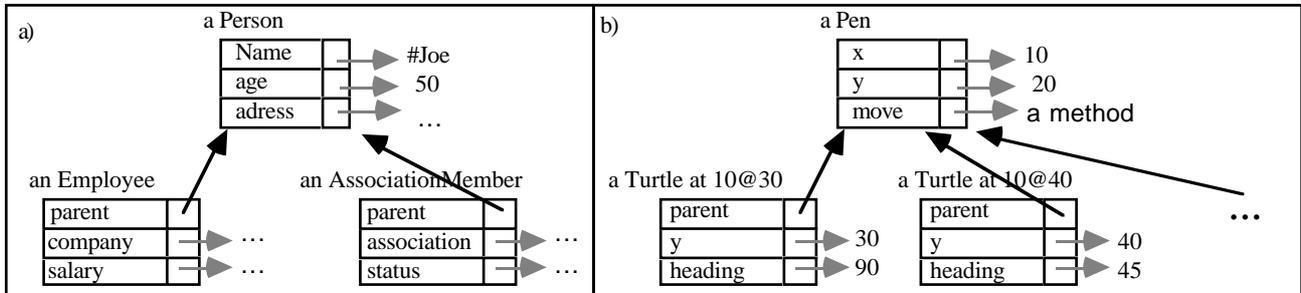


Figure 3: Two uses of delegation

able to deal with explicit delegation orders.

As far as we could figure out from [MyGV92], Garnet is a language in the family of (L8). However Garnet, like in [Born86], uses constraints propagation as primary computation mechanism rather than message passing and delegation, and this introduces some distortion when we classify it using our taxonomy.

### 3.2 Semantic Issues

Sharing plays a central role in object-oriented programming as the cornerstone of reusability. But what kind of sharing is provided by cloning and by delegation? Are they reducible to each other? What is the kind of sharing provided by prototypes and what is its impact on their semantics?

#### 3.2.1 The Kind of Sharing Achieved by Shallow Cloning

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 creating a new point  $p2$  as a clone of a point  $p1$  owning a method  $move$  to change its location (Fig. 2a). After shallow cloning,  $p1$  and  $p2$  share the method  $move$  and their position by the virtue of pointing to the same value objects through their

respective slots  $move$ ,  $x$  and  $y$ . Notice however that, even if  $p1$  and  $p2$  get the same structure and the same values, when  $p2$  is modified, for example, the two objects become independent (Fig. 2b) and cease sharing updated values. 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 by their clone. The independent evolution applies to slot individually; here  $p1$  and  $p2$  continue to share the method  $move$ , even after  $p2$ 's  $y$  slot has been modified.

#### 3.2.2 The Kind of Sharing Achieved by Delegation

Delegation achieves **life-time sharing**: as long as an object and its parent exist, the sharing will continue. Here is a good example using this. Consider an object representing the person *Joe* with slots *name*, *age* and *address*. Delegation allows users to create an object representing *Joe* as an employee as a new object with slots *company* and *salary*, extending the person *Joe* through a parent pointer (Fig. 3a). The structure of the object *Joe* is not modified and the object *Joe\_as\_employee* is split in two parts. *Joe\_as\_employee* shares the knowledge of *Joe*, an appropriate kind of sharing since both objects apply to the same real entity: *Joe*.

Creating an extension object instead of simply

adding the slots salary and company to *Joe* also leaves the door open for other extensions, e.g. *Joe* as a member of an association (Fig. 3a). Any modifications to *Joe* 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 be made at the person level and will be effective for all extensions of this person.

This example is also a good one of what delegation can do that class inheritance cannot. In class-based systems, it is possible to define the concept of employee by extension of the concept of person, but it is not possible to extend *Joe*, the instance of person, with an object like *Joe\_as\_employee*; if *Joe* already exists in the system, an instance of employee sharing the information stored in *Joe* cannot be created. Furthermore, it would be impossible to create an instance *anAssociationMember* which would share the information about the person.

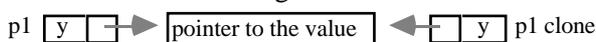
Another interesting use of life-time sharing is advocated in the *pen-turtle* example (Fig 3b). Sharing of values via delegation allows here a row of turtles to be derived from the same pen object by sharing the same x-coordinate.

### 3.2.3 Cloning and Delegation are Basic

Since cloning insures creation-time sharing and delegation life-time sharing, an interesting question is whether or not one of the two mechanisms can achieve both kind of sharing, therefore making the other one unnecessary as a primitive.

### Achieving Life-Time Sharing with Cloning

In cloning, the template and its clone both point to the same values after the cloning operation concluded. Using cloning to do life-time sharing would try to make this sharing through pointers last over the whole life of both objects. Such a life-time dependence could be enforced, but at the cost of a more complex addressing model in which an indirection would be added to access slot values, as illustrated in this drawing:

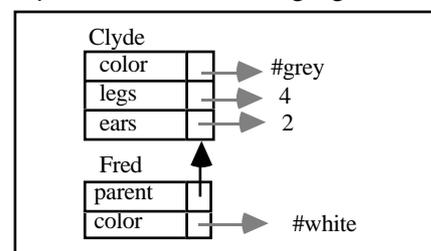


The assignment primitive could modify the value

but *p1* and *p1*'s clone would continue to share the same pointer to the value. However two villains would remain: firstly a new and distinct assignment primitive would be needed to modify the value of *y* of *p1*'s clone without modifying *p1*, and secondly, if modification of the structure of objects is allowed, a new method or variable created on *p1* would not be known of *p1* clone as it would with delegation. This last problem could be denoted as the incremental propagation of newly created slots to clones of an object requiring the memorization of every clone of each objects, as described by Borning [Born86]. Hence, simulating delegation with cloning is impossible without special mechanisms or if we want to keep both the addressing model and the clone operation simple.

### Achieving Creation-Time Sharing with Delegation

Conversely, trying to use delegation to enforce creation-time sharing can lead to design errors. Consider *Clyde*, the prototypical example of elephants [Lieb86]. The elephant *Clyde* is a grey, 4 legs, normal elephant. Then comes the white elephant *Fred*, since it is exactly like *Clyde* except for the colour we create an object with a colour slot assigned to the value *white* and with a parent link towards *Clyde* as in the following figure:



Now if *Clyde* has an accident and loses one leg, *Fred* also loses it! *Fred* should then be modified either to add it the slot *legs* assigned to its original value 4 or to change its parent. But there is no easy way for *Clyde* to tell *Fred*, and more generally for an object to tell all the objects that delegates to him, that he has changed, that he has moved from the status of prototypical object to the one of exceptional object. In such a case, cloning is more appropriate. This suggests first that prototypical instances should be immutable, but let us postpone the discussion of this issue to §3.2.5, and second, that in this case cloning

should have been used.

Our conclusion is that **both delegation and cloning are basic mechanisms**, providing different capabilities that sometimes overlap but all of which being necessary.

### 3.2.4 Delegation and the Meaning of Self: Split Objects

As the *person-employee-associationMember* example shows, delegation establishes a very strong link between an object and its parent. This has a very important effect on the notion of object itself. If we also consider the graphic turtle object [Lieb86] (Fig. 3b), how could we define the notion of individual object in a delegation-based programming model? Is *aPen* an object by itself when *aTurtle at 10@30* is created with a parent link towards it? If I reset the x-coordinate of *aPen* to 15, *aTurtle at 10@30* is also intimately changed and becomes *aTurtle at 15@30*. Conversely, if I reset the x-coordinate of *aTurtle at 10@30*, *aPen* is changed but also *aTurtle at 10@40*!

So, *aTurtle* is dependent of *aPen* first because it is incomplete and second because a change to *x* in *aPen* also changes *aTurtle*. Moreover, *aPen* is also dependent of *aTurtle* because a change to *x* in *aTurtle* is also modifying *aPen*; any later use of *aPen*, such as creating a new turtle reusing the same pen, that doesn't take that into account may lead to incoherences. Should we conclude that this use of delegation is not useful in practical examples and should be forbidden? No! On the contrary, both examples illustrate very interesting uses of this capability. They are also good examples of what delegation can do that class inheritance cannot, as we have said before.

However, as far as delegation allows objects to share slots, it raises the problem of the semantics of objects that delegate messages to each others. In the example, we can no more consider neither *Joe* nor *Joe\_as\_employee* nor *Joe\_as\_associationMember* as independent objects. In fact, all the information scattered among the different extension objects applies to the same real entity, here our fellow Joe. The same is true but in a slightly different way for the turtle example. Having many turtles derived from the same pen object makes the information scattered

among these elementary objects applying to the same entity, namely the row of turtles.

Hence, we conclude that a useful concept of **split object emerges from delegation and parent links**, and this concept must be treated as such in the language.

### How to manipulate split objects

Since split objects exist as a concept and are useful in practice, how can they be referenced and manipulated? Let us first define what is a split object:

a split object is an entire set of elementary objects linked together through parent links.

For example, the elementary objects *Joe*, *Joe\_as\_employee* and *Joe\_as\_associationMember* together make the split object *Joe\_as\_aWhole*. If we look at this split object, we can make some observations.

Handling such objects introduces several problems. The main one is how can I refer to it? If we consider first the elementary object *Joe* prior the creation of its children, there is no problem; I can refer to *Joe* simply through the standard object pointer. When I add it a child *Joe\_as\_employee*, I can refer to the elementary object *Joe\_as\_employee* through its standard object pointer but how can I refer to the split object created by the pair *Joe* and *Joe\_as\_employee*? One solution could be to promote the pointer to *Joe\_as\_employee* in order to make it designating the whole split object. For instance, we could make the *shallowClone* primitive aware of this promotion and, when the *shallowClone* message would be sent to *Joe\_as\_employee*, it could copy both *Joe\_as\_employee* and *Joe*. Moreover, if we promote the pointer to *Joe\_as\_employee* to designate the pair of objects, we loose the ability to refer to only the elementary object *Joe\_as\_employee*. This may not be what we want. Even worse, what if I add *Joe\_as\_associationMember*? How can I refer to *Joe\_as\_aWhole*?

Notice that we can decompose *Joe\_as\_aWhole* into four interesting sub-objects that we call constituents. The first constituent is the elementary object *Joe*. The second constituent is the pair

*Joe\_as\_employee* and *Joe* that describes completely the employee Joe. A third constituent is the pair *Joe\_as\_associationMember* and *Joe* that describes the member of an association Joe. The last constituent is the split object *Joe\_as\_aWhole* itself. Each of these constituents is an interesting entity that we would like to be able to refer to in order to manipulate them normally.

The whole treatment of split objects as first-class entities in the language resides in this designation problem. For example, should we introduce in our example new inverse links from the elementary object *Joe* to all its children and use the pointer to *Joe* to designate the split object? Should we introduce new elementary objects to play the role of the split objects by having pointers to all its elementary objects? or to all its constituents?

At present time, we have no satisfactory solution to this problem. It is an open research problem on which we will come back in the conclusion of the paper.

### 3.2.5 Class-like Sharing in Prototype-Based Languages

Cloning and delegation achieve different but useful kinds of sharing. Are they providing the kind of sharing provided by classes in traditional object-oriented languages?

#### Prototypical Instances

Reuse from prototypical instances using both cloning and delegation cannot serve as a mechanism to provide class-like sharing because it causes semantics problems, especially when modifications are made to the prototypical instance.

The problem with cloning is that when modifying the prototypical instance, there is no way to insure that the family of objects cloned from this prototypical instance will be homogeneous; objects cloned before the modification will not be affected, thus they will diverge from the new concept implemented by the updated prototypical instance.

With delegation, the problem is seen the other way around. When the prototypical instance is modified, all objects delegating to it are affected by the

modification; if the prototypical instance loses its status of being prototypical and instead becomes an exceptional instance of the concept, the original concept is lost.

In both cases, a solution may be to make prototypical instances immutable, but this solution is contrived in two ways. First, it introduces a distinct kind of objects in the prototype-based model, which then loses part of its elegance and simplicity. Second, by making prototypical instances immutable, we lose the capability of normally speaking about these objects. For example, I can't use Clyde, and say that it has lost a leg, because of its status of prototypical instance. We conclude that prototypical instances are not the right device to implement class-like reuse in prototype-based languages.

#### Factoring Common Properties

Given the capability of delegating messages to other objects, an alternative is to factor common knowledge of a group of similar objects into one shared repository. For example, let's use the point example. A prototypical instance of point may contain the  $x$  and  $y$  coordinates of the point, the methods *move*, to change its coordinates, and *print* to print points on the terminal. Delegation offers the possibility to create first a shared repository  $S$  containing only the common properties of points, here the methods *move* and *print*, and then to create individual points containing only  $x$  and  $y$  coordinates, but having  $S$  as parent. As far as we know, Self is the only language that explicitly raised this issue, and it calls shared repositories "traits objects" [CUCH91]. The comments made here apply to traits objects in Self, but would also apply to any language using similar traits objects.

Shared repositories raise some important questions of semantics. First, their status as objects receiving messages is intriguing. Consider again the shared repository  $S$ ; since the *move* and *print* methods it defines and owns presumably access the  $x$  and  $y$  coordinates of points, it is impossible to send it *move* and *print* messages without raising errors. Also, we lose in some sense the capability to speak about  $S$  as a standard object in the system. For example, how could we have a *print* method for  $S$  itself? As soon

as *S* redefines the *print* method to specify the behavior of its eventual sub-objects, it masks any other *print* method applying to itself, therefore rendering impossible the application of a message *print* to itself. Although ad hoc solutions to this problem exist, they do not change the fundamental problem.

In fact, shared repositories are a form of abstract objects and creating them prior concrete objects seems to go against one of the *raison d'être* of prototypes. The problem of printing is just a sign that the abstract shared repository in fact declares methods for its sub-objects and that the link between an object and the shared repository is not simply an inheritance one. Hence, shared repositories are not a solution to achieve class-like reuse in prototype-based languages.

Now, if such abstract objects must be banished from prototype-based applications (at least at the programming level), the programmer could be prevented from creating them by testing at method creation time that nowhere a method tries to access a variable or another method within the object which is not implemented by the object itself or by one of its parent. Indeed, this does not banish references to self and late binding still useful to create extension objects redefining variables or methods.

### How to Provide Class-Like Sharing?

A negative result seems now to emerge: there is no way within the prototype-based model to achieve class-like sharing. Prototypical instances with both delegation and cloning have been shown not to work safely in full generality. Shared repositories are ruled out because they impair the intended semantics of prototypes. No doubts however that this kind of sharing is mandatory to obtain space-efficient implementation of prototype-based languages. We cannot afford to copy exactly the same information from objects to objects, for an entire family of similar objects.

In fact, we have now shown with conclusive arguments that Borning was right when he said that this kind of sharing should be regarded as an implementation technique [Born86]. Trying to achieve class-like sharing within the prototype-based programming model leads to incoherences. The

positive result is that maybe we do not have to bother about this kind of sharing at the programming model level. In the implementation of Self [ChUL89], maps provide exactly the kind of sharing we need, and maps are generated and managed automatically at the implementation level. Actually, maps are shared repositories but objects are linked to their map through a map-of-link. This link is different from the parent link but this does not hurt the prototype-based model at all, simply because this new link is hidden within the implementation. Hence, a proper way to achieve sharing and reuse among a set of similar objects is to do it at the implementation level using devices such as maps.

## 3.3 Our Model

Our taxonomic tree leads to a number of different languages. Let's now look at them and propose one model as the best in the light of the above discussion.

### 3.3.1 Feasible Languages in the Taxonomy

Families of prototype-based language are represented by the leaves of our taxonomy (Fig. 1) each of which having different impacts on sharing and encapsulation.

Languages (L1) to (L12) are the most basic since they don't provide delegation. Let us cope first with those that are uninteresting for the sake of this discussion.

- Languages (L4) and (L10) only allow to create empty objects, the structure of which cannot be modified.
- Languages (L2) and (L8) are reducible to languages (L1) and (L7) respectively; both provide primitives to create non-empty objects the structure of which can be dynamically modified. Hence, similar results can be achieved by first creating a new empty object and by modifying its structure. Creation of non-empty objects in this case is just syntactic sugar.
- Even if they are supplemented with delegation lower in the taxonomy, languages (L6) and (L12) are useless because no objects can be created ex-nihilo and no new properties can be added to objects.

All the other languages of this first set can be

extended with the different alternatives in cloning and delegation. (L1), (L3), (L5), (L7), (L9) and (L11) are usable and interesting. To our knowledge and except in our platform (see section 4), these have not been implemented yet. This is certainly due to their impoverished capabilities, but they could be used as assembler languages to implement other more powerful prototype-based languages.

To extend these six languages with delegation and eventually with a support for split objects yield a second set of higher-level languages. In this paper we will focus on those ((L13) to (L18)) derived from (L5) and (L11) because, as we will explain in the next section, they are, from our point of view, the most interesting. Languages (L13) to (L18) are workable; instead of assessing them one by one, we prefer to look at the best choice at each level in the taxonomic tree in order to derive the proposals for new prototype-based languages.

### 3.3.2 The Proposal

Let us take the taxonomic tree and identify the best path among all these alternatives. Both alternatives in the representation of prototypes have advantages and disadvantages. Slots are more flexible but they force to implement an encapsulation mechanism, which comes for free to some extent in the variables&methods approach. At the moment, we still need to experiment before inclining to favour one or the other, so we see **both as justifiable** choices.

Dynamic modification of the structure of objects can be very dangerous. No encapsulation mechanism, either by making variables inaccessible from the outside of an object or by marking some slots as private, can be implemented (properly) in languages with dynamic modification of structures. If the user of an object can add it a new method, he can also break the encapsulation by adding reading or writing accessor methods for any slot. Therefore, **languages without dynamic modification of structure** enforce a robust and more disciplined view of prototype-based programming while other languages provide some flexibility which can hurt more than it may help.

Given our choice of languages without dynamic

modification of structure, we must choose to have primitive to **create objects ex nihilo**. The other alternative leads to use cloning to create new objects and we have seen above that languages using cloning without dynamic modification of structure are impossible to use. For the very same reason, we choose to create objects ex nihilo using the primitive *newInitials*: instead of *newEmpty* because the latter would need dynamic modification of structure in order to create useful objects.

Delegation gives us the choice between implicit and explicit delegation. Obviously, explicit delegation is more tedious to use than implicit delegation. But it is more flexible since it gives the possibility to choose the object where to delegate on a message per message basis. In implicit delegation, an object identifies its parents through parent links and messages are delegated to these objects only. The reason why we favour implicit delegation comes from the fact that delegation introduces split objects. The chief difference between implicit and explicit delegation is that when delegation is made implicit in the language, the link between an object and its parent must be explicit but when delegation is explicit, this link is blurred into the code of methods where the explicit delegation messages appear. Because the link between an object and its parent appears to be so important in the structure of objects, we also conclude that **implicit delegation is highly preferable** since this link is made explicit.

Because of our previous discussion on the emergence of split objects, we prefer languages which make them first-class entities. Hence, (L14) and (L17) provide the kind a prototype-based programming models we advocate.

To summarize, our proposal is the model characterized by the following statements:

- Prototypes should be represented either with variables and methods or with slots, the most important point being to implement an encapsulation mechanism. The structure of prototypes should be immutable, to enforce encapsulation by preventing malicious users to dynamically add public accessors to private information.
- The first primitive to create new objects should be

*newInitials:*, a primitive which create an object with an initial set of slots and slot values. This avoids the need of dynamic modification of the structure of objects.

- Delegation should be implicit rather than explicit, using a parent link to implement the delegation. Delegation should be used in the language to extend existing objects thus creating split objects. The chief advantage of implicit delegation is to make the relationship between objects explicit to ease the understanding of their behavior.
- The primitive *newInitials:* should prevent users from creating abstract objects by forcing each message send to self in all objects to have a binding, either by a slot in the object itself or by a slot in one of its parent; this avoids the semantic problems associated to abstract objects, but still preserves the ability to redefine slots.
- Split objects should be treated as first class entities and, in particular, it must be possible to create them, refer to them, shallow clone them, and otherwise deal with them as with any object in the language.
- Neither delegation nor cloning should be used to achieve class-like sharing among a family of similar objects. We have shown that this kind of sharing causes semantic incoherences in the prototype-based model. Since this kind of sharing is mandatory to get space-efficient implementations of prototype-based languages, implementation devices such as maps proposed by Self should be used to achieve it at the implementation level.

## 4. Prototalk

We now discuss some implementation issues through an overview of the Prototalk platform.<sup>1</sup> The snapshot in Appendix A illustrates the use of this platform to experiment with various prototype-based languages.

---

<sup>1</sup>Note that we do not implement prototype-based languages using Smalltalk-80 constructs. Lieberman [Lieb86] has shown that delegation cannot be implemented in a class-inheritance oriented language using message passing unless the specific properties of Smalltalk class variables are used [Steil87]. We choose Smalltalk as the implementation language mainly for its programming environment and its reflective capabilities.

## 4.1 The architecture of the platform

The platform is built on the following principles: each "interesting" language (L) in our taxonomy is represented by a class (CL). The classes are organized into an inheritance hierarchy reflecting the taxonomy. More precisely:

- The object of the language L are implemented by instances of the class CL.
- The internal representation chosen for the objects of the language L is implemented by the instance variables of the class CL. For example the class *VMProto* (prototypes having variables and methods) has an instance variable named "methods"; the method dictionary of each object of the languages implemented by subclasses of *VMProto* will be stored in the related field.
- The primitives of the language L are represented by methods defined on the class CL, for example the primitive "clone" of the language (L13) is implemented by a method *clone* defined on the class L13.
- The mechanisms associated to the language L are implemented in the evaluator (EL) associated to the class CL, i.e. the evaluator for the language L. For example the delegation mechanism is implemented in the evaluator associated to the class *ImplicitDelegation*. To each class implementing a prototype-based language are associated a set of program nodes classes representing the various instructions of L. In the tradition of Lisp interpreter, the evaluator for L is made of a set of methods *eval:* and *apply:* defined on the various program node classes [Lieb87], each one being tailored to the interpretation of a particular instruction of the language. For example when the parser for a language providing implicit delegation encounters a message sending instruction, it generates in the parse tree an instance of our new class *ImplicitDelegationMessageNode*. This class has three instance variables: *receiver* which is a program node, *selector* which is a symbol and *arguments* which is an array of program nodes.
- Finally, a workspace is associated to each class CL in the hierarchy, where character strings typed in by the user are considered as expressions of the corresponding language L.

```

eval: context  “defined on the class ImplicitDelegationMessageNode”
“The argument “context” is a dictionary containing the current values of “self” and of the arguments and temporaries of the
method in which the message sending is performed. The variables “receiver”, “selector” and “arguments” are instance
variables of the receiver of the “eval:” message.”
| client server args newContext |
1  client := receiver eval: context.          “receiver is evaluated”
2  server := rec serverFor: selector.        “I search the method in the receiver”
3  if (server equals nil)
4      then the exception doesNotUnderstand is raised
5  else  args := arguments evlis: context.    “the arguments are evaluated”
6        newContext := Dictionary new.      “I create a new context”
7        newContext at: #self put: client.  “In which I bind self to the new receiver”
8        newContext at: #super put: (server parent)
9  I return: ((server methodName: selector) applyWith: args in: newContext)

```

**Figure 4: Evaluation of a message sending instruction for a language with implicit delegation.**

```

7  newContext at: #self put: (context at: #self)

```

**Figure 5 : Modification of fig. 4 to implement eval: on MessageToSuperMessageNode**

An interesting property of the platform lies in its use of the class hierarchy to inherit evaluators of the different languages from one another. Each new evaluator inherits its implementation from a previous one, except for those constructions that are given a different semantics in the new language.

## 4.2 The organization of classes.

The classes of the platform reflect the taxonomy described in Section 3. *AbstractProto* is an abstract class owning common behavior. *VMProto* stands for prototypes having variables and methods, it owns two instance variables to store them. *Sproto* stands for prototypes having slots. *VMModifiable* stands for prototypes having variables and methods (it derives from *VMProto*) in which the structure of objects can be modified after their creation. It owns two methods: *addMethod:* to add methods and *addVar:value:* to add variables to objects. It is also an abstract class since it does not provide methods to create objects. The class *L1* implements the language (L1) of the taxonomy, it owns the method *newEmpty* to create empty objects. The classes implementing the other languages of the taxonomy are integrated in the inheritance hierarchy by following the same principles.

## 4.3 Implementation of delegation.

To describe our implementation of delegation will give an idea of how the evaluators work, especially for explicit delegation since no precise description of

its implementation is given in the literature. Delegation requires that when applying a method, the pseudo-variable “self” be bound to the receiver of the message (the client) rather than to the object in which the method was found (the server).

### 4.3.1 Implicit delegation.

The class *L7AndImplicitDelegation* extends the class *L7* with an internal representation for prototypes having parents, with methods to create such objects and with an evaluator knowing how to delegate messages

A choice has to be made when representing the parent link: are parents accessible in the language? In other words, should the parent slot be stored in the objects or at the implementation level? Parents are not a priori supposed to be visible within objects since they are only used by the evaluator to achieve delegation. We thus have defined “parent” as an instance variable, and “son” as a method creating new empty objects (e.g. *o1* and *o2* in Appendix A, *L7AndImplicitDelegation* workspace), having the receiver as parent. Future extensions of the class *L7AndImplicitDelegation* will be free to provide reading or writing accesses to object’s parents. The internal representation of these prototypes thus have two fields, one pointing to the parent and one, inherited from *SProto*, pointing to the slots.

The method *eval:* defined on the class *ImplicitDelegationMessageNode* evaluates message sending instruction in the language

*L7AndImplicitDelegation*. This amounts to evaluate the receiver (line 1), to search a server for the selector M in the receiver (line 2), if a server is found (line 5) to evaluate the arguments, to create a new context (line 6), to bind in this context the pseudo-variable “self” to the receiver (line 7) and to apply the method M to its arguments in the new context (line 9). The method *serverFor:*, defined on the class *L7AndImplicitDelegation* finds which object owns the method to be applied, if the method is not owned by the receiver, it performs a lookup through the parent hierarchy.

The resend of messages via a pseudo-variable “super” (as in Smalltalk) is a fundamental mechanism in an implicit delegation language whereas it is not needed with explicit delegation. For example, considering the implementation of Lieberman's dashed turtle example [Lieb86] using implicit delegation shows that resend of messages is mandatory.

The line 8 in Figure 4 partly implements the resend of messages: if “super” is used in the method to be applied, the search will start at the right place. But this is not enough: indeed, while interpreting “super m”, the original client will be lost; “super m” does not mean “send the message m to the object which is the value of the variable super” but “delegates the message m to the appropriate server while preserving the current client”. We thus have modified our parser so that it creates a different kind of message node (instance of the class *MessageToSuperMessageNode*) when encountering a message to “super” in a method. The *eval:* method defined on that new class is similar to the above one (Fig 4) except for the line 7 (Fig. 5).

We will see that the same solution will be used for the implementation of explicit delegation.

### 4.3.2 Explicit delegation.

Explicit delegation requires (1) the ability for an object to intercept the messages he wants to delegate and (2) a way to express delegation without confusing it with message sending.

Before delegating a message, an object first has to intercept it. The obvious and simplest solution is to

redefine, for each message to be delegated, a method on the delegating object, the body of which simply delegating the message. Putting simplicity aside, this solution has several drawbacks [Lieb86]: expressing delegation becomes tedious, grouping delegation orders is impossible, clients have to redefine methods each time new ones are added in their server, etc. More sophisticated solutions to this problem exist (see e.g. the objects scripts of Act1). Currently, only the first solution is implemented: if an object A wants to delegates a message M to B, he has to define a method named M in which it delegates the message using the appropriate construct. (cf. fig 6).

Here is an example of code in the language associated to the class *L7AndExplicitDelegation* (Fig. 6).

The expression “*o1 delegates: #m1 withArgs: nil*” in Figure 6 stands for “delegate the message *m1* to *o1* and apply the related method in my environment”. The question now is: how to interpret the expression “*o1 delegates: #m1 withArgs: nil*” located in the method *m1* of *o2*?

```

o1 := Root newEmpty.
o1 addSlot: 'x 10'.
o1 addSlot: 'm1    self x + self x'.
o2 := Root newEmpty.
o2 addSlot: 'x 20'.
o2 addSlot: 'm1    o1 delegates: #m1 withArgs: nil'.
o2 m1 --> 40.

```

**Figure 6 : L7AndExplicitDelegation workspace (see also appendix A)**

Should that expression be a standard message sending, “self” would change [Lieb86] and the reference to the actual client would be lost. A solution could be, for the client, to explicitly pass itself as argument when delegating a message, for example by writing: “server delegates: aMessage withArgs: arguments forClient: self”. Unfortunately this would also allow programmers to write things like “server delegates: aMessage withArgs: arguments for: any\_object\_out\_there”, leading to interesting semantic questions.

In our solution, the expression “*o1 delegates: #m1 withArgs: nil*”, although having the syntax of a message sending, will not be interpreted as such by the system. The selector *delegates:WithArgs:* is tracked at parsing time to generate an instance of the

class : *ExplicitDelegationNode*. For example, parsing the expression “*o1 delegates: #m1 withArgs: nil*” creates an instance of *ExplicitDelegationNode* with receiver *o1* and selector *#m1*. Then, the new method *eval:* defined on *ExplicitDelegationNode* is exactly the same that *eval:* for *MessageToSuperMessageNode* (Fig. 5).

#### 4.4 Integration of existing Systems

The purpose of the platform is to easily implement simulations of various prototype-based languages and to write comparative programs. Up to now, we have implemented in the platform all the languages described in Figure 1 except (L14) and (L17) on which we are still working and we have integrated simulations of Self, Exemplars, Object-Lisp and Act1 (cf. Appendix A). The position in the hierarchy of a class has been discussed in §3.1.5, it indicates the kind of instructions that can be written, the kind of mechanisms that are available in the language and the operational semantics of these mechanisms. The platform faithfully simulates these languages, but of course it gives no information on the real internal representation of objects nor on how the mechanisms are really implemented in the actual language.

### 5. Conclusion

Prototypes provide a serious alternative to the class-based approach but current languages have been designed using very different semantics. In this paper, we have proposed a new taxonomy of prototype-based languages validated by its implementation as a Smalltalk-80 class hierarchy. From this taxonomy, we have extracted a set of design choices providing, in our view, the most elegant and the most useful prototype-based programming model. This model is characterized by the following statements:

- Prototypes should be represented either with methods and variables or with slots, the most important point being to implement an encapsulation mechanism.
- The first primitive to create new objects should be *newInitials:*, a primitive which creates an object with an initial structure.
- Delegation should be implicit, using a parent link

to implement the delegation. Delegation should be used in the language to extend existing objects thus creating split objects.

- The primitive *newInitials:* must prevent the users from creating abstract objects i.e objects having methods in which are referenced other slots not defined in the object or in one of its parents.
- Split objects should be treated as first class entities.
- Neither delegation nor cloning should be used to achieve the class-like sharing among a family of similar objects, which should instead be achieved at the implementation level.

Many avenues are now awaiting a more complete exploration. First and most important, we have shed the light on the central concept of split objects. A deeper understanding of this concept is mandatory for the correct use of delegation but it raises several crucial issues. How can we treat split objects as full first-class entities in the language, i.e. create them, refer to them, clone them and otherwise deal with them as with any other object? Is the parent link enough to deal with them or will another link be necessary (e.g. the inverse of the parent link)? How can we refer to the constituents of a split object but still enforce the encapsulation of the split object? Can we define more precisely what is a split object? Are there many different kinds of split objects, as the *person-employee-associationMember* example compared to the *pen-turtles* example seems to suggest? Is it possible to design a programming methodology for prototypes which would insure that only such legitimate split objects are created?

As a second avenue, new programming models are challenging our taxonomy. For example, some prototype-based languages use a constraint propagation computing model which subsumes message passing and achieve similar effects on objects as the sharing provided by delegation do in more traditional languages. The taxonomy should be adapted to take these new trends into account.

Finally, as a third avenue, some interesting but less basic issues have still to be addressed: dynamic modification of parents links, multiple delegation (the prototype's interpretation of multiple inheritance), the questions of whether or not there should be an initial

object in the system and what it should look like, and also the questions of how to access primitives in the language and how they could be redefined by prototypes.

Our future work is to look more attentively at these issues and to experiment the design of representative applications using our language proposal.

## Acknowledgements

The authors wish to thank Jean-François Perrot and Jean Vaucher for their useful comments on the preliminary drafts of this paper.

## References

- [Born81] A.H. Borning. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. *ACM Transaction on Programming Languages and Systems*, 3(4):353-387, October 1981.
- [Born86] A.H. Borning. Classes versus Prototypes in Object-Oriented Languages. In *Proceedings of the IEEE/ACM Fall Joint Conference*, pages 36-40, 1986.
- [ChUL89] Craig Chambers, David Ungar, and Elgin Lee. An Efficient Implementation of Self, a Dynamically-typed Object-Oriented Language Based on Prototypes. *Proceedings of OOPSLA'89*, ACM Sigplan Notices, 24(10):49-70, October 1989.
- [CUCH91] 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):207-222, 1991.
- [ChUn91] C. Chambers and D. Ungar. Making Pure Object-Oriented Languages Practical. *Proceedings of OOPSLA'91*, ACM Sigplan Notices, 26(11):1-15, November 1991.
- [LaLo89] W.R. LaLonde. Designing Families of Data Types Using Exemplars. *ACM Trans. on Prog. Languages and Systems*, 11(2):212-248, April 1989.
- [Lieb81] H.Lieberman. A preview of Act1.AI memo No 625, MIT, June 1981.
- [Lieb86] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. *Proceedings of OOPSLA'86*, ACM Sigplan Notices, 21(11):214-223, November 1986.
- [Lieb87] H.Lieberman. Reversible Object-Oriented Interpreters. in *Proceedings of European Conference on Object-Oriented Programming (ECOOP'87)*, special issue if BIGRE No 54, pp 13-22, June 1987, Paris.
- [LaTP86] W.R. LaLonde, D. Thomas, and J.R. Pugh. An Exemplar Based Smalltalk. *Proceedings of OOPSLA'86*, ACM Sigplan Notices, 21(11):322-330, November 1986.
- [MACL89] Macintosh Allegro Common-Lisp Reference Manual, version 1.3.
- [MGDV90] B.A. Myers, et al.. Garnet, Comprehensive Support for Graphical, Highly Interactive User Interfaces. *IEEE Computer*, 23(11):71-85, November 1990.
- [MyGV92] B.A.Myers, D.A.Giuse, B.Vander-Zanden. Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. To appear in the *Proceedings of OOPSLA'92*.
- [Smit86] R.Smith. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. *Proc. of the 1986 IEEE Workshop on Visual Languages*, Dallas, Texas, pages 99-106, June 1986.
- [StLU88] L.A. Stein, H. Lieberman, and D. Ungar. A Shared View of Sharing: The Treaty of Orlando. In W. Kim and F. Lochovsky eds., *Object-Oriented Concepts, Applications and Databases*. Addison-Wesley, 1988.
- [Steir87] L.A. Stein. Delegation is Inheritance. *Proceedings of OOPSLA'87*, ACM Sigplan Notices, 22(12):138-146, December 1987.
- [UCCH91] D. Ungar, C. Chambers, B.-W. Chang, and U. Hölzle. Organizing Programs without Classes. *Lisp and Symbolic Computation*, (4):223-242, 1991.
- [UnSm87] D. Ungar and R. Smith. Self: The Power of Simplicity. *Proc. of OOPSLA'87*, ACM Sigplan Notices, 22(12):227-242, December 1987

## Appendix A

### Overview of the Smalltalk-80<sup>2</sup> platform for prototype-based languages simulation.

---

<sup>2</sup>The platform has been programmed with Objectworks\Smalltalk, releases 2.5 and 4.0; Objectworks\Smalltalk is a trademark of ParcPlace System, Inc.