Split Objects: a Disciplined Use of Delegation within Objects *

Daniel Bardou
bardou@lirmm.fr

Christophe Dony
dony@lirmm.fr

Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier
161, rue Ada – 34392 Montpellier Cedex 5 – FRANCE

Abstract

This paper's primary aim is to improve the understanding of the delegation mechanism as defined in [18]. We propose a new characterization of delegation based on the notions of name sharing, property sharing and value sharing. It allows us (1) to clearly differentiate delegation from class-inheritance in particular and more generally from other inheritance mechanisms and (2) to explain how a founded use of delegation relies on a correct semantics of variable property sharing between objects connected by a delegation link. We then describe a model of split objects which is proposed as an example of a disciplined and semantically founded use of delegation, where property sharing expresses viewpoints within objects.

1 Introduction

All kinds of inheritance mechanisms in object-oriented programming or representation languages, despite their diversities [5, 26], have at least the following common points [11]:

- They are based on a relation $\mathcal{I}$ between objects or between concepts (for example the subtype relation between abstract data types which is implemented in class-based languages by a "superclass" link between classes).
- The mechanism itself uses that relation to achieve inheritance (for example a message sending mechanism in class-based language which performs a *lookup* — whatever form it takes — along the "superclass" links to retrieve inherited properties).
- An interpretation or a semantics is given to the relation $\mathcal{I}$ to justify the above mechanism. For example in class-based languages the formal property that gives a meaning to the "superclass" link and underlies inheritance is inclusion polymorphism[1] [6], other interpretations, such as concept specialization or set inclusion, have been given in other contexts.
- Finally, a common characteristic of all inheritance mechanisms is that they are used to achieve some kind of sharing [26].

We deal in this paper with the "delegation" mechanism, as it is defined in [17, 18, 26], which does not exactly denotes a message redirection (as in actors systems) but actually a kind of inheritance mechanism (which can be understood in

*Partially supported by Marché CNRS/CNET 93 1B 142, Projet 5115 "Réseau futé".

[1] The reason why it is founded to "look up" for methods in a superclass is that any function applicable to objects of a type $\mathcal{T}$ (instances of a class $\mathcal{C}$) is intended to be also applicable to objects of type $\mathcal{T}'$ subtype of $\mathcal{T}$ (instances of $\mathcal{C}$'s subclasses).

a way as a message redirection). Delegation in this sense is thus a kind of inheritance mechanism which is generally associated to object-centered or prototype-based programming [18, 29, 30] in a "one kind of object-one kind of link" world. In fact, to be class-less and to provide a delegation mechanism are two orthogonal features of an object-oriented programming or representation language [19]. Efficient class-less languages with some kind of sharing but without delegation can be built [28] and it is possible to imagine some uses of delegation in many different worlds including worlds without classes [1] and worlds with classes [10, 15, 16]. The delegation mechanism is based on a link (generally called "parent" or "delegation" link) between objects[2], rather than between descriptions. In object-oriented programming, delegation is presented as a message forwarding mechanism [17, 18], informally described in the following way: "an object that cannot answer a question can delegate it to its parent, if the parent can answer it, the answer will be performed in the context of the values of the original object". An example of such objects can be found in the classical point-turtle example (see Fig. 3-a, page 8). Finally delegation is a kind of sharing mechanism [9, 26].

If this very general description is to be compared to the four points above, it appears that two very important things are missing that limit our understanding of delegation: (1) what is the kind of sharing actually achieved by this mechanism and (2) how to interpret the relation on which the mechanism is based:

1. The kind of sharing achieved by delegation is not yet well characterized, for example the difference between delegation and class-inheritance (let us give that name to the inheritance mechanism found in class-based systems) is still unclear. Contrasting with generally accepted ideas, we claim that delegation is not class-inheritance and that delegation is not to class-less objects what class-inheritance is to classes. Delegation is not class-inheritance because the former and the latter do not induce the same sharing relation between objects: an instance i_1 of a class α and an instance i_2 of a class β subclass of α do not share the same things than an object o_1 and an object o_2, o_1 having o_2 as parent.

2. Once the kind of sharing actually achieved by delegation has been characterized, the next question is: what does it mean for two entities to be connected by a delegation link? More precisely: what semantics or interpretation can be given to the relation based on that link?

The aim of this paper is to address these two points. Concerning sharing, it is already well known that delegation induces a sharing of variable values but a formal characterization of delegation is needed to express more precisely and in a simple way the differences with other inheritance-based sharing mechanisms. For what concerns the interpretation of the link, the problem is to correctly use variable value sharing which, when used without care, raises the problem of object-identity [9, 19, 27]. What does it mean for an entity to share variables values with another? A first answer can be found in some existing systems in which delegation is used to perform some life-time default value sharing between concepts and objects. We will introduce and develop a second answer [3, 20] by defining what we call "split objects". In split objects, delegation is used **inside** objects to express property sharing between different perspectives or viewpoints.

The paper is organized as follows. Section 2 formally defines name sharing, value sharing and property sharing. Section 3 proposes a characterization, in term of these notions, of both sharing in class-based systems and sharing in delegation class-less systems. It shows how delegation induces property sharing for both variables and methods. Section 4 recall the possible bad consequences of sharing variables on object modularity. Section 5

[2] We more exactly mean between entities holding variable values and used to perform computations, these can be classes but seen as objects.

recall a well-known use and interpretation of variable sharing with delegation and its semantics as it can be found in existing systems. Section 6 then discuss a second founded use of delegation which express viewpoints. Section 7 describes what could be a model in which the latter use of delegation is disciplined within what we call split objects. Finally section 8 briefly compares split objects with some existing object systems allowing to express viewpoints.

2 Name sharing, value sharing, and property sharing

We define in this section the notions of name sharing, property sharing and value sharing[3] between objects integrated in a single[4] inheritance world. These notions will be used in the next section to formally characterize the kind of sharing achieved by various inheritance mechanisms.

2.1 Properties

What defines objects in object-oriented systems are, generally speaking, properties. We use the term property to designate either a "slot", an "attribute", a "field", an "instance variable", or a "method". We do not use the term property to designate generic properties[5] (it is for example possible to talk about the global entity *printOn:* in Smalltalk), but rather the properties actually defined on classes or objects (for example the method *printOn:* of the class *String* in Smalltalk). Properties are declared and defined for objects (wherever the declaration and definition take place).

Property names. We consider that properties can be identified by a name in the context of an object[6]. Several properties can of course be given the same name for different objects.

Property values. Properties may be complex entities but they have at least a value (they may also have a type, a signature, a cardinality, etc.). Each property has only one value (of course, a value can be the value of several properties).

Formally, let us consider an object system characterized by a set $\mathcal{O}$ of objects, a set $\mathcal{P}$ of properties, a set $\mathcal{N}$ of property names and a set $\mathcal{V}$ of values, we define the following functions:

$$\begin{array}{lcccc} NameOf & : & \mathcal{P} & \longrightarrow & \mathcal{N} \\ RefProp & : & \mathcal{N} & \longrightarrow & 2^{\mathcal{P}} \\ Prop & : & \mathcal{O} \times \mathcal{N} & \longrightarrow & \mathcal{P} \\ Val & : & \mathcal{P} & \longrightarrow & \mathcal{V} \end{array}$$

Given an object $o \in \mathcal{O}$, a property $p \in \mathcal{P}$ and a property name $n \in \mathcal{N}$: $NameOf(p)$ is the name of p, $RefProp(n)$ is the set of the properties named n, $Prop(o, n)$ is the property of o which is named n (the property identified by n within the context of o), and $Val(p)$ is the value of p. We also note $Value(o, n) = Val(Prop(o, n))$ the value of the property of o which is named n.

Various kind of sharing will then be defined in terms of two sets: $\mathcal{N}_o$ and $\mathcal{N}^o$. $\mathcal{N}_o$ is the set of property names which identify a property within the context of o, in other words $\mathcal{N}_o$ is the set of properties declared for o. The value of a property for an object o can be either defined at the object level[7] or inherited. For an object o, $\mathcal{N}^o$ is the set of the names of properties the value of which is defined at the object level. We separate o's property names in two sets: $\mathcal{N}^o$ and $\mathcal{N}_o - \mathcal{N}^o$, the set of the names of properties the value of which is inherited.

[3] The formalism we define has been inspired from the one appearing in [11] in which the notions of name inheritance and value inheritance are defined. The two formalisms are however different and there is no direct correspondence between name inheritance and name sharing, or between value inheritance and value sharing. Indeed, generic properties are considered in [11] whereas they aren't here.

[4] These notions are basically not different in presence of multiple inheritance but are more difficult to express formally.

[5] Such entities are reified in Clos (generic functions) and in Lore.

[6] We use the term "name" to designate what is sufficient to identify a property, this term can thus encompass name and signature for example.

[7] We mean either in the object or in its class or anywhere in a place directly accessible independently of the inheritance mechanism.

2.2 Sharing

To say that an object o has a property n with value v means three things: (1) o has **a** property named n (i.e. $n \in \mathcal{N}_o$), (2) this property is identified in o by n as a certain property p (i.e. $Prop(o, n) = p \in \mathcal{P}$), and (3) the value of **the** property of o which is named n is v (i.e. $Value(o, n) = Val(p) = v \in \mathcal{V}$).

We define sharing as a relation[8] $\mathcal{S}$ from $\mathcal{O}$ to $\mathcal{O}$, and we note $<_\mathcal{S}$ (respectively $\leq_\mathcal{S}$) the transitive (respectively transitive and reflexive) closure of $\mathcal{S}$. Sharing thus applies at those three levels: what can actually be shared are property names, properties themselves and property values.

1. **Name sharing**

   What is shared in *name sharing* is the fact of having a property of a given name, or in other words, the declaration of a property or the existence of a property.

   More formally, a sharing relation $\mathcal{S}$ is characterized as being a *name sharing* relation when:

   $$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 <_\mathcal{S} o_2 \Rightarrow \mathcal{N}_{o_1} \supseteq \mathcal{N}_{o_2}$$

   or: if $o_1 <_\mathcal{S} o_2$ then if o_2 has a property named n, then o_1 also has one. For a concrete example, consider a class-based world with two instances i_1 and i_2 of a class α (or i_1 an instance of α and i_2 an instance of a subclass of α): if an instance variable name n is declared on α then i_1 and i_2 share the fact that they have a property named n, and that it will be possible to ask any of i_1 and i_2 the value of its property named n.

2. **Property sharing**

   Property sharing implies name sharing. What is shared in *property sharing* are properties themselves. It occurs when a given name n identifies exactly the same property in two or more different objects). We characterize as being a *property sharing* relation any *name sharing* relation $\mathcal{S}$ such that:

   $$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S} o_2 \Rightarrow$$
   $$(\forall n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1},\ Prop(o_1, n) = Prop(o_2, n))$$

   or: if $o_1 \mathcal{S} o_2$ then if o_2 has a property named n and if o_1 has no local definition for that property ($n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1}$) then the property of o_1 which is named n **is** also the property of o_2 which is named n. The important consequence (discussed in § 4) is that the value of the property of o_1 which is named n will be the same than the value of the property o_2 which is named n (at least as long as $n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1}$). For an example, consider two instances i_1 and i_2 of the same class α and a method named m which value is defined on α, i_1 and i_2 share that method.

3. **Value sharing**

   What is shared in *value sharing* are property values. A sharing relation $\mathcal{S}$ is said to be a *value sharing* relation if:

   $$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S} o_2 \Rightarrow$$
   $$(\forall n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1},$$
   $$Value(o_1, n) = Value(o_2, n))$$

   or: if $o_1 \mathcal{S} o_2$ then if o_2 has a property named n then the value of the property named n of o_1 is the same as the value of o_2's property. *Value sharing* implies *name sharing*. *Property sharing* implies *value sharing* but the opposite is not necessarily true. *Value sharing* is further discussed in § 5.

3 Sharing in class-based systems and sharing achieved by delegation

We give in this section a characterization of both kinds of sharing found in class-based systems and

[8] We do not expect a sharing relation to be symmetric. This can be misleading as far as to say that something is shared between an object o_1 and another one o_2 is equivalent to say that the same thing is shared between o_2 and o_1. We define however sharing as a directed relation in order to be able to identify what is shared between two objects.

delegation-based systems, using the previously defined notions of *name sharing* and *property sharing*. This characterization will reveal how the two inheritance mechanisms are different.

3.1 Sharing in class-based systems is name sharing for variables and property sharing for methods

Classes, variables and methods. In class-based systems a distinction is made between state properties (variables) and behavioural properties (methods). The declarations of variable and method names and the definitions of method values[9] are done at the class level. Given an object o, let us note $Class(o)$ the class of o, and given a class α, $\mathcal{VN}_\alpha$ the set of the variable names declared for instances of α and $\mathcal{MN}_\alpha$ the set of method names declared for instances of α. According to our previous definitions (see § 2), $\mathcal{N}_o = \mathcal{VN}_\alpha \cup \mathcal{MN}_\alpha$ for any object o.

Class-inheritance. Classes are organized in a class-inheritance graph which we note $G_\mathcal{I} = (\mathcal{C}, \mathcal{I})$ where $\mathcal{C}$ is the set of classes and $\mathcal{I}$ is the inheritance relation. We note $<_\mathcal{I}$ the transitive closure of $\mathcal{I}$, and $\leq_\mathcal{I}$ the transitive and reflexive closure of $\mathcal{I}$. Given two classes α and β, if $\alpha <_\mathcal{I} \beta$ (respectively $\alpha \mathcal{I} \beta$) then α is said to be a subclass (respectively a direct subclass) of β.

Sharing in class-based system is variable name sharing. Class-inheritance is used to determine the $\mathcal{VN}_\alpha$ set for any given class α. It ensures that any variable name of which the declaration is held by α or one of the α's superclasses is an element of $\mathcal{VN}_\alpha$:

$$\forall(\alpha, \beta) \in \mathcal{C}^2,\ \alpha <_\mathcal{I} \beta \Rightarrow \mathcal{VN}_\alpha \supseteq \mathcal{VN}_\beta$$

Therefore, it is easy to prove the existence of a variable name sharing relation $\mathcal{S}_\mathcal{I}$ between objects, which can be defined by:

$$\forall(o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S}_\mathcal{I} o_2 \Leftrightarrow \left\{ \begin{array}{c} Class(o_1) = Class(o_2) \\ or \\ Class(o_1) \mathcal{I} Class(o_2) \end{array} \right.$$

[9] Method values can be thought of as lambda-expressions.

In other words, each instance of a class α shares the existence of its variables with any other instance of α and moreover with any other instance of any α's subclass. Of course, each instance will own a proper value for each variable declared in its class. Note that there is neither property nor value sharing for variables.

Sharing in class-based systems is method property sharing. As for variable name declarations, class-inheritance is used to determine the $\mathcal{MN}_\alpha$ set of any given class α:

$$\forall(\alpha, \beta) \in \mathcal{C}^2,\ \alpha <_\mathcal{I} \beta \Rightarrow \mathcal{MN}_\alpha \supseteq \mathcal{MN}_\beta$$

It is straightforward to establish that $\mathcal{S}_\mathcal{I}$ is also a method name sharing relation.

Furthermore, class-inheritance is also used for method activation. Methods that can be activated by sending a message to a given object o are those the value of which is defined in $Class(o)$ or in one of $Class(o)$'s superclasses. We distinguish between them by calling $\mathcal{MN}^{Class(o)}$ the set of the former (the latter being in the set $\mathcal{MN}_{Class(o)} - \mathcal{MN}^{Class(o)}$).

Given an object o_1, when a message is sent to activate the method m named n of o_1, a lookup for m's value definition is performed which is started at o_1's class and eventually continued along the inheritance links to be finally found in a certain class β. β can hold only **one** value definition for the method named n, this implies that n identifies a unique method within the context of β, and this method is m. As we could have chosen any object o_2 among the instances of any class α, such that $Class(o_1) \leq_\mathcal{I} \beta$ and $\alpha \leq_\mathcal{I} \beta$, instead of o_1 for the lookup to terminate in β, we can conclude that:

$$\begin{array}{l} \forall(o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{S}_\mathcal{I} o_2 \ \Rightarrow \\ \qquad (\forall n \in \mathcal{MN}_{Class(o_1)} - \mathcal{MN}^{Class(o_1)}, \\ \qquad\qquad Prop(o_1, n) = Prop(o_2, n)) \end{array}$$

Thus class-inheritance induces property sharing for methods. In other words, all the instances of a class β have the same behavioural properties (and thus the same behaviour) and, given any α, subclass of β, all of these properties which are not

redefined in α are also some behavioural properties of α's instances.

3.2 Delegation achieves (variable and method) property sharing

We characterize here the kind of sharing achieved by delegation. In a delegation-based system, property name declarations and property value definitions are directly done at the object level, on a per-object basis [26], and inheritance directly occurs between objects. Objects are linked together by delegation (or parent) links in a delegation graph $G_{\mathcal{D}} = (\mathcal{O}, \mathcal{D})$, the vertices of $G_{\mathcal{D}}$ are the objects of the system. $\mathcal{D}$ is the delegation relation, we note $<_{\mathcal{D}}$ its transitive closure and $\leq_{\mathcal{D}}$ its transitive and reflexive closure. If, given two objects o_1 and o_2, $o_1 <_{\mathcal{D}} o_2$ (respectively $o_1 \mathcal{D} o_2$, i.e. there is a delegation link from o_1 to o_2) then o_1 is said to be a descendant (respectively a child) of o_2, and o_2 an ancestor (respectively the parent) of o_1.

Delegation is a message forwarding mechanism which ensures that: when an object o_1 is asked for the value of one of its properties named n and it can not answer by itself (i.e. $n \notin \mathcal{N}^{o_1}$), the question is forwarded to the ancestors of o with the task for the nearest ancestor which is able to answer, let's call it o_2, (i.e. $n \in \mathcal{N}^{o_2}$) to perform the answer in the value context of o.

$\mathcal{D}$ is also a name sharing relation. Indeed, for any object o a value can be computed for any property name which is declared for any ancestor of o, either this value is explicitly defined for o, or it is inherited. Therefore any property name which is the name of a property of at least one of the ascendants of o is also a property name for o:

$$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 <_{\mathcal{D}} o_2 \Rightarrow \mathcal{N}_{o_1} \supseteq \mathcal{N}_{o_2}$$

$\mathcal{D}$ is also a property sharing relation. Indeed, consider the lookup performed at the reception of a message sent to an object o_1 asking for the value of a property p named n. Let's call o_2 the object where this lookup ends. o_2 is the object holding the definition of p's value, and since o_2 can only hold **one** definition of the value of the property named n, n also identifies p within the context of o_2 (i.e. $Prop(o_1, n) = p = Prop(o_2, n)$). Note that we could have done the same reasoning with any object o_3, such that $o_1 \leq_{\mathcal{D}} o_3$ and $o_3 \leq_{\mathcal{D}} o_2$. We thus deduce that:

$$\forall (o_1, o_2) \in \mathcal{O}^2,\ o_1 \mathcal{D} o_2 \Rightarrow \\ (\forall n \in \mathcal{N}_{o_1} - \mathcal{N}^{o_1},\ Prop(o_1, n) = Prop(o_2, n))$$

In other words, an object shares each of its properties with its descendants which do not hold a definition of the value of this property.

Remark: As there is exactly the same kind of sharing for both variables and methods in delegation-based systems, they can be (but are not necessarily) embedded in the unique notion of property (as done with the "slots" of SELF [1]).

4 Variable value sharing and delegation semantics

The important result of the previous section is the characterization of delegation as a mechanism entailing property sharing for both variables and methods and class-inheritance as one entailing also method property sharing but only variable name sharing. From this perspective, the main difference between the two mechanisms is that delegation induces property sharing for variables. We focus on this characteristics to bring to the fore the semantic issues raised by the existence of a delegation link between two objects. For this purpose, we recall in § 4.1 the intrinsic interest of variable property sharing and in § 4.2 the problems it raises.

4.1 Variable property sharing is useful

Consider some objects representing a person — say *Joe* — in a delegation-based system as shown in Fig. 1 (such an example is also considered in [3, 9, 20]). Suppose we first want to consider *Joe* as a simple person, we create the object `JoePerson` with the variables `address`, `age`, `name` and `phone`, and a method `growOld`. Then the object `JoeSportsman` is created with the variables `stamina` and `weight` as a child of `JoePerson`

in order to be able to deal with *Joe* as a sportsman. The delegation link from `JoeSportsman` to `JoePerson` ensures that all the properties of the latter are shared by the two objects. Note that representing *Joe* by these two objects allows us to deal with:

- *Joe* as a sportsman (whose properties are `weight`, `stamina`, `address`, `age`, `name`, `phone` and `growOld`),
- *Joe* as person (without the `weight` and `stamina` properties).

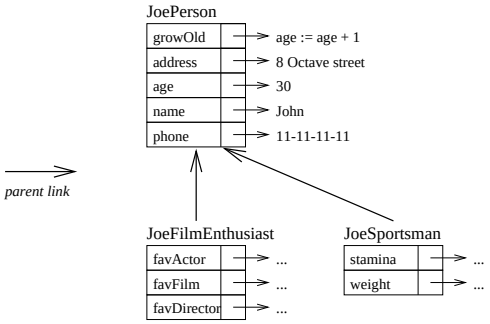

Figure 1: A representation of *Joe* split in three objects.

We also keep the ability to create other children of `JoePerson` along other lines. We do it when creating the `JoeFilmEnthusiast` object, having `JoePerson` as parent, with the definitions of the variables `favouriteActor`, `favouriteFilm` and `favouriteDirector`.

Note that in this example we clearly want the three objects to denote the same entity of the real world, the real person *Joe*. Thus the `address` variable[10] owned by `JoePerson` is also intended to be the `address` variable of `JoeSportsman` and `JoeFilmEnthusiast`. For that reason, variable property sharing achieved by delegation is **in this case** welcomed because any message attempting to affect the `address` variable will result in the modification of the `address` variable in the `JoePerson` object and thus be effective for all the three objects.

[10]The `address` variable is here considered as an example, we could have made the same remarks for any other variable owned by `JoePerson`.

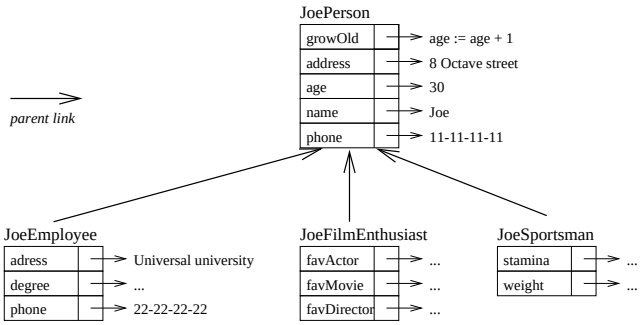

Figure 2: Another representation of *Joe* completed with an employee aspect.

Furthermore any property can be redefined in a child of `JoePerson`. For example, `JoeEmployee` is created as a new child of `JoePerson`, denoting *Joe* as an employee, in which a new `degree` variable is defined and the two variables `address` and `phone` are redefined (see Fig. 2). As far as redefined properties are not shared, *Joe* is represented as having different address and phone number while he is at his work.

There is no simple way to obtain an equivalent representation in a class-based language since class-inheritance simply does not achieve variable property sharing.

4.2 Variable property sharing raises the problem of object identity

We recall here how variable property sharing breaks the frontiers of objects and subsequently breaches encapsulation [27]. More generally we raise the issue of the interpretation of the delegation link.

Consider another example in a delegation-based system. A point at 5@10 is represented by an object `Point1`, and a turtle at 10@10 and heading to 90 by an object `Turtle1` (see Fig. 3-a). Because a turtle object is like a point object having one more variable (`heading`) and two more methods (`rotate` and `forward`), and more specifically because `Turtle1` has the same value (at least at creation time) for the `y` variable and a different

value for the `x` one as `Point1`, `Turtle1` is made a child of `Point1`. If now `Turtle1` is asked to move at 10@14, its `y` value has to be changed. A value definition of `y` is not found in `Turtle1` but in `Point1` where the modification is performed (see Fig. 3-b). As delegation basically achieves property sharing, the parent link not only grants a read access but also a write access for `Turtle1` to the `y` variable owned by `Point1`.

Consequently, if we only expect an object to be an entity which is able to receive messages, then `Turtle1` and `Point1` can be considered as two different objects but this become false as soon as we also expect an object to be an independent individual entity.

In most of the delegation-based languages, there is no prerogative to create an object as a child of another. Full access to an object's properties can then be gained in an unexpected manner by creating a child of it. Either property sharing basically achieved by delegation should be restricted to value sharing (see § 2 for definition), or a set of objects connected by delegation links have to be considered as the parts of the representation of only one global entity (as was done in our previous example given in § 4.1). To choose between these two alternatives amounts to choose between two different but founded use of delegation.

5 A first semantics: individual objects and default values

A first sound semantics for delegation can be found in some actors systems (e.g. Act1 [17]), frame-based ones (e.g. Y3 [10]) but also prototype-based ones (e.g. the KR language of Garnet [22]). In these systems, the properties of an object can not be modified by sending an affectation message to one of its descendants. An object has at least as many properties as its parent, each of these properties is identified by the same name within the context of any of the two objects and has the same default value[11].

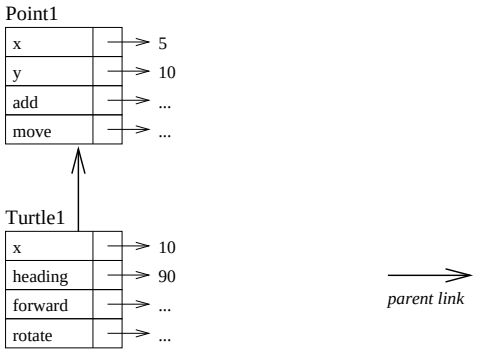

Figure 3-a: before the modification of `y`.

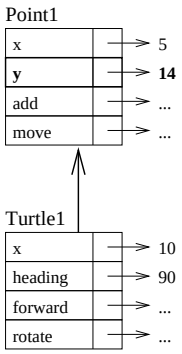

Figure 3-b: after the modification of `y` with property sharing.

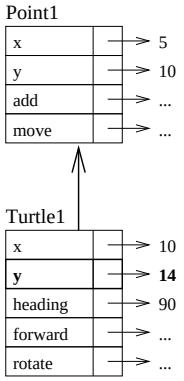

Figure 3-c: after the modification of `y` with value sharing.

Figure 3: `Point1` and `Turtle1` share the `y` variable. The modification of `y`'s value of `Turtle1` leads to the case of 3-b with property sharing and to the case of 3-c with value sharing.

Consider again our point-turtle example (see Fig 3-a). The reason why the `Turtle1` object is made a child of the `Point1` object is clearly in this case the reuse of the definition of `Point1`'s properties. We do not want `Point1`'s properties to be the very properties of `Turtle1`, but rather `Point1` to provide default values for the `Turtle1`'s variables which are not redefined. In order to ensure this an `y` affectation message sent to `Turtle1` should not

[11] Lieberman also suggested objects as default behaviour and value repositories for their children in [18].

result in `Point1`'s `y` value modification as shown in Fig. 3-b but rather in the definition of the `y` variable in `Turtle1` as shown in Fig. 3-c. Such an interpretation of variable affectation amounts to restrict, by a separate mechanism, property sharing (basically achieved by delegation) to value sharing. Either this mechanism can test for the existence of a definition in some ascendant for the variable to be affected before adding a new local definition (as in Y3 [10]. Or it can systematically consider the affectation of a non locally defined variable as the creation of a new initialized variable (as in KR [22]).

Delegation links can in this case be intended as "is-like-a" links. Delegation then grants read access to variables but no more write access to the parent properties. The frontiers between objects are then clear: in our example `Point1` and `Turtle1` are really two different objects and can evolve on their own. The only way to modify the value of the `Point1`'s `y` variable is to explicitly send a message to `Point1`. Although this will also modify the `y` value for `Turtle1`, this isn't unexpected since this value is only intended to be a default one.

6 A second semantics: viewpoints within an object

We showed in the previous section that property sharing is restricted to value sharing in some existing systems in order to get rid off the problem of object identity. But we also pointed out in § 4.1 the usefulness of property sharing by producing an example of the representation of a person *Joe*. We introduce in this section a second sound semantics which retains property sharing.

Consider again the representation of *Joe* which is split in four objects (see Fig. 2): `JoePerson`, `JoeEmployee`, `JoeFilmEnthusiast` and `JoeSportsman`. What does this mean exactly? According to Ferber, objects denoting the same entity (coreferent objects) denote viewpoints of this entity [13]. It should be clear that `JoePerson` denotes *Joe* as a person, `JoeSportsman` *Joe* as a sportsman, `JoeFilmEnthusiast` *Joe* as a film enthusiast and `JoeEmployee` *Joe* as an employee.

To split a representation in several objects in a delegation hierarchy is simply a natural way of representing viewpoints. As in a description hierarchy, the most general viewpoints are those denoted by the objects near the top of the hierarchy whereas the most specific viewpoints are those denoted by the objects which are leaves of the hierarchy. In our example, person is a more general viewpoint on *Joe* than either employee, sportsman or film enthusiast.

We however still face some problems because we can not deal with the whole representation of *Joe*. We can not send messages to it but only to one of the objects denoting a viewpoint of *Joe*. We neither deal with the whole representation as a structure in order to duplicate it for example. The problem of object identity is still not resolved but we now know that the frontier to be put on the split representation is one around it all, that is around the four objects in our example. A solution to these problems is to give the object status to the whole representation and to remove that status from its four parts. Indeed, we can no more consider `JoeEmployee` (for example) as an object because there would be a conflict between its frontier and the one of the whole representation. The inadequacy between the system and the real-world should be recovered: a one-to-one correspondence between objects in the system and entities of the real world should be ensured. We define *split objects* for this purpose in the next section.

7 Split objects

We have shown in the previous section that delegation can be used to achieve a per-viewpoint representation of a single entity of the real world. We also pointed out that a solution to some problems raised by these representations relies on a one-to-one correspondence between objects in the system and entities of the world being represented. We propose in this section a model for *split objects* in which this correspondence is respected.

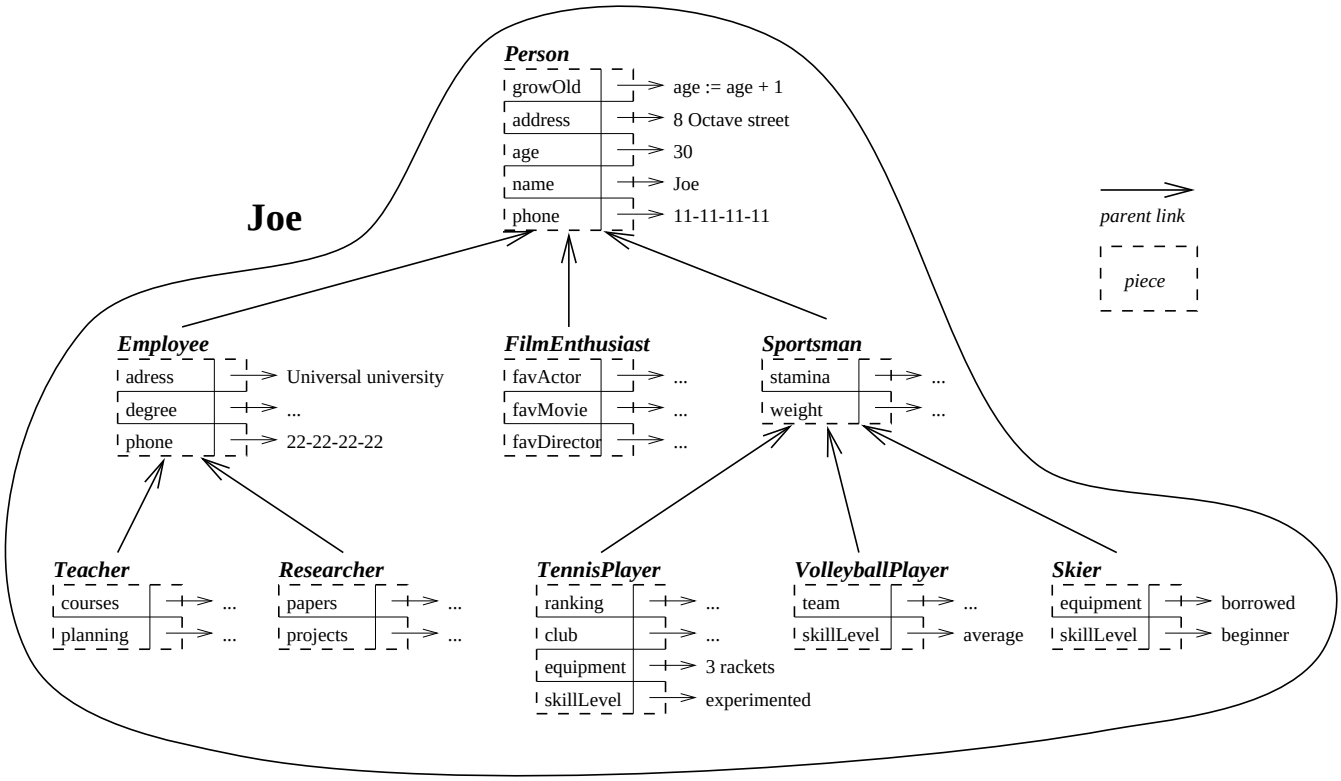

Figure 4: A split object representing a person, each piece denotes a viewpoint on this person.

7.1 A basic model for split objects

A *split object* is defined as a collection of *pieces*. The properties of a split object are stored into its pieces. Pieces are organized within an object in a delegation hierarchy (a property sharing one). A split object denotes a single entity of the real world and its pieces denote viewpoints of this entity. **Pieces do not have an object status, whereas split objects do**. To illustrate what a split object is, consider our new completed representation of *Joe* shown in Fig. 4. *Joe* is now represented by a single split object `Joe` which contains nine pieces. **`Person`, `Employee`, `FilmEnthusiast`, and `Sportsman` are no more objects but only pieces of `Joe`**. We detail below the basic features that an object-centered language with cloning should include to provide for split objects. The reader is also referred to [3] for a more complete description of this model and to [20] for a denotational semantics of a similar one.

Naming and accessing
As far as split objects are first class entities, they are directly accessible. This is not the case of pieces, which can only be accessed through the encompassing split object by specifying a piece name.

Creation
Split objects are created by cloning. A special case of creation is creation ex-nihilo: it is achieved by first cloning *the empty split object* (a predefined object of the language having no pieces) and then performing as many piece additions as required.

Cloning
Cloning a split object results in a new split object, initially composed of the same set of pieces holding the same properties. Cloning a split object thus consists in deep copying the pieces hierarchy in which each piece is shallow copied.

Modification
Split objects can be modified on a per piece basis.

One can add to, delete from or modify a piece of a split object.

- Adding a piece is performed by creating a new empty piece as a child of an existing one. Identifiers of both the existing and the new pieces should be specified.
- Piece deletion also implies the deletion of all the descendant pieces in the hierarchy: recall that a piece denotes a viewpoint and descendant pieces denote more specific viewpoints. If *Joe* is no more a sportsman, it is clear that he can no more be a skier, a volleyball player or a tennis player.
- Piece modification consists in either adding a property, deleting a property, setting a new value to a variable, or changing the body of a method stored in a piece of the split object.

Basic message sending
A split object's property values are accessed by message sending. Since methods and variables are owned by the pieces and since pieces denote viewpoints, messages are sent on a per viewpoint basis. When a message is sent to an object, a viewpoint must be specified by giving the identifier of the piece denoting it. A lookup is then performed, starting at this piece and eventually continued in its ascendant pieces, in order to compute and return the reply. Message sending can be extended to support messages sent to an object without specifying any viewpoint. This will be further discussed in § 7.2.

Pseudo-variables
As in all object-oriented languages, a pseudo-variable *self* is bound, during a method's activation, to the current receiver, i.e. in our case a split object. For the particular case where one wants to send a message to *self* from the same viewpoint as the message currently being evaluated, a second pseudo-variable *thisViewpoint* is also provided. During the evaluation of a message, *thisViewpoint* is bound to the name of the piece denoting the current viewpoint.

Structure coherence controls
At modification time, it is possible to check that the tree structure of the piece graph of a split object is preserved.

7.2 Extending message sending to address more viewpoints

We have presented in previous subsection a basic model in which the user can deal with each of the viewpoints denoted by a piece in a split object. An interesting issue is now to know whether there are only as many denoted viewpoints as there are pieces in a split object. We show in this section that there are more viewpoints than pieces. We give some clues on how messages can be sent from all these viewpoints, including a particular one which can be thought of as the "reunion" of all the others.

The split object `Joe` (see Fig. 4) is a collection of nine pieces and each of them denotes a viewpoint on *Joe*. `Joe` thus denotes at least nine viewpoints. We are able to deal with any of these nine viewpoints by sending messages. We can for example ask *Joe* as a researcher which are the papers he has written by sending a message to `Joe`, indicating the `Researcher` piece and the selector `papers`. But how can we get this information if we do not know that the `papers` variable of `Joe` is stored in the `Researcher` piece? Shouldn't we be able to deal with *Joe* as a whole, the global viewpoint on *Joe* from which any of the properties defined in `Joe`'s pieces can be a priori accessed? We would like to answer yes to this question but there is no piece which denotes *Joe* as a whole: the global viewpoint is only implicitly denoted in `Joe`, and messages can not a priori be sent from it.

If we allow pieces to have more than one parent, a naïve solution to this problem would be to create a piece to denote *Joe* as a whole. This piece could be an empty one made the child of each of the leaves of the hierarchy in order to gain access to any of the properties defined for `Joe`. But this solution is only applicable in a system providing for multiple delegation.

However, a key remark is that *Joe* as a whole

can be considered as the composition of other viewpoints of *Joe*, denoted by pieces of different branches of the sharing hierarchy. We can also consider that all possible combinations of such pieces denote interesting implicit viewpoints on the entity denoted by a split object. For example, we could want to deal with the teacher-and-researcher viewpoint on *Joe* (denoted by the combination of the `Teacher` and `Researcher` pieces). But we also need to consider all possible compositions of viewpoints, since even a combination of some pieces of the same branch does not denote the same viewpoint as the one denoted by the more specific of these pieces. Just consider for example the person-and-employee viewpoint (denoted by the combination of the `Person` and `Employee` pieces): it is not the same viewpoint as the employee one because asking the address of *Joe* as an employee is an unambiguous question whereas asking the address of *Joe* as a person-and-employee is ambiguous.

When counting the viewpoints (implicitly or explicitly) denoted in a split object, one must thus count as many as there are non empty subsets in the whole set of pieces. In our example, `Joe` has 9 pieces and denotes $2^9 - 1$, that is 511, viewpoints on *Joe*. We conclude that creating an empty piece to explicitly denote each viewpoint implicitly denoted is not a practically applicable solution with respect to the memory consumption.

Message sending can be extended to support messages along any of the viewpoints denoted in a split object. Implicitly denoted viewpoints can be specified by a list of piece identifiers. It is also possible to systematically consider a message sent with no viewpoint specification as a message sent from the global viewpoint. Properties of a split object can then be activated without knowing in which pieces they are stored and full support for encapsulation is thus provided.

As there can be ambiguous messages, a lookup strategy should be chosen, which at least detects ambiguities. It seems reasonable to expect also this strategy to not perform any lookup in pieces which are not related (either ascendant or descendant of) the pieces specified as a viewpoint. We have proposed such a strategy in [3] which ensures an uniformly defined semantics of messages sent to either explicitly or implicitly denoted viewpoints.

8 Viewpoints in other systems

Our primary goal in this work was not to design a new model for viewpoints but rather to fully understand delegation and its semantics. The notion of viewpoints nevertheless came naturally out when we thought about split representations. Many other works have been done around this notion in the object-oriented field of research. We discuss them in a short comparison with our split objects.

The notion of perspectives appears in Loops [4, 5]. Perspectives, which are reified by independent objects with separate name spaces, are clustered into some special composite objects and are interpreted as different views on the same conceptual entity. Each perspective on an entity can be accessed from any other perspective on the same entity.

Ferber developed a theory of viewpoints based on the coreferentiality [12, 13], where viewpoints of an entity are represented by coreferent objects. Translation operations and coreference rules between coreferent objects can be defined to ensure the coherence of the whole representation of the application domain.

The main differences between these two approaches split objects lies in the fact that viewpoints on a split object are not reified. Viewpoints are also clustered as in Loops, but they are not independent. Some kind of coherence between viewpoints is also ensured by the sharing relation between pieces.

ROME [7, 8] emphasizes fragmentation of the definition of properties in a inheritance hierarchy (principle of *multiple and evolving representation of objects*). Viewpoints (denoted by classes) are also organized in a generalization/specialization hierarchy in which lookup can be restricted to some classes during the evaluation of *as-expression*

messages. In this respect, the functionalities of split objects with extended message sending are very similar to those of ROME.

The viewpoint notion can also cover viewpoints on the entire application domain, rather than solely viewpoints on individual entities. Indeed, in Tropes [21, 24] a viewpoint corresponds to a whole hierarchy of classes. Such hierarchies, the root of which are called concepts are used for classification purposes: an object is the instance of only one concept, which totally describes its attributes names, but can be classified within a viewpoint in the concept's subclasses, which specialize descriptions of instances in term of attribute values.

Harrison and Ossher proposed a model of subjects [14], where a subject also specifies a whole hierarchy of classes. A subject merely corresponds to the view of the world one can have from the context of a particular application. Subjects can be activated (eventually more than once) in order to associate some data and behaviour to any of the object identities of the system. An object identity can be associated to different data and behaviour by different subject activations. Subjects can also be composed together, in which case a *composition rule* then defines how they cooperate in respect with various topics including message sending, coherence of representations, object creations, etc. (see [23] for two possible composition rules).

It is possible to compare split objects and extended message sending with subjects, because for example viewpoints can be composed in the two models. Semantics of viewpoints composition are not determined by the same factors: this issue does not rely in our model on some composition rule but rather on the lookup strategy chosen for implicitly denoted viewpoints and the property sharing relation imposed on the pieces of a split object.

Viewpoints can be considered in object-oriented application design techniques. Role modeling [2] emphasizes separation of concern between different aspects at possibly different level of details in the design of more or less independent applications. A role model is a behaviour description of two or more entities termed roles. Roles describe the requirements that objects have to satisfy in order to achieve the behaviour described by the role model. An important operation on role models is synthesis where some roles are sometimes projected together into a new aggregated role to obtain a new role model.

Considering that requirements of a role can be clustered into a piece, and that projection of roles can then be thought of adding pieces to a split objects, our model may perhaps be a well-adapted tool for the implementation of role models.

Finally, Us [25] also uses delegation to express perspectives. Definitions of objects in Us are done through layers: each layer contains exactly one piece (eventually an empty one) of each object. Objects can inherit from each other by delegation, and so do layers in a separate but orthogonal delegation hierarchy. A layer considered together with its chain of layer parents is a perspective from which messages can be sent to objects. The *receiver-perspective symmetry principle* governs evaluation of messages: layer parent links and object parent links have the same semantics.

The symmetry principle is quite well respected, but is weakened by the lookup algorithm which composes the layer parent hierarchy first. This is the main difference with our model in which a split object must first be considered before viewpoints (our viewpoints are also viewpoints on objects rather than viewpoints on the whole system). There are nevertheless strong similarities between split objects and Us: the notion of piece, the use of delegation and the generalization/specialization relation between perspectives.

9 Conclusion and future works

In this paper we have introduced a formalism in which we defined the notions of *name sharing*, *property sharing* and *value sharing*. This allowed us to precisely qualify the kind of sharing achieved in class-based systems as *variable name sharing* and *method property sharing*, and the one achieved

by delegation as *property sharing*.

We have then shown how property sharing can be used to obtain per viewpoint representations of objects but also explained that a secure use of delegation relies on a correct semantics of delegation links. As a first sound semantics we highlighted the restriction property sharing to value sharing done in some existing systems, using delegation to achieve default life-time value sharing between independent objects. As a second semantically founded use of delegation, we proposed a new model of split objects in which property sharing no more occurs between objects but rather between one object's viewpoints. We have finally briefly compared the split object model with other systems providing viewpoints.

We discussed how split objects can be manipulated in a class-less language, creation of new objects is basically achieved by cloning. We however believe that there exists a corresponding class-based model: one can think of classes in which the declaration of variables and the definition of methods are partitioned along a set of piece names. Indeed, delegation occurs between pieces within split objects and we did not put any requirements either on inheritance between split objects, or on the organization of their world. We currently are investigating what are the pros and cons of choosing between a class-based system or an object-centered one to implement split objects. Topics pertinent to this choice are including:

- the possible solutions for factorization of common properties: code reuse can be achieved by class-inheritance in class-based systems, but it remains a problem in prototype-based languages [3, 9];
- the way viewpoints are expressed in split objects, whether piece names of an object are defined at the class-level or directly at the object-level;
- creation and dynamic modification of split objects.

Another important issue to be discussed is encapsulation. As far as it is possible to put frontiers between split objects, some support for encapsulation is provided by our model, but we do not give details on how encapsulation can actually been achieved. Moreover there may be two levels of encapsulation: encapsulation of properties (the "classical one") and encapsulation of pieces.

Acknowledgements

This work began in collaboration with Jacques Malenfant, we would like to thank him for his numerous and relevant comments. We also would like to thank Bernard Carré, Roland Ducournau and Gilles Vanwormhoudt for many fruitful discussions on viewpoints and delegation. Thanks to Allister Cockburn for his help in finalizing this paper.

References

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, and M. Wolczko. *The* SELF *4.0 Programmer's Reference Manual*. Sun Microsystems Inc, Stanford University, 1995.

[2] E.P. Andersen and T. Reenskaug. System Design by Composing Structures of Interacting Objects. In *Proceedings of the 6th European Conference on Object-Oriented Programming (ECOOP '92), Utrecht NL, Lecture Notes in Computer Science 707*, pp. 133–152, 1992.

[3] D. Bardou and C. Dony. Propositions pour un nouveau modèle d'objets dans les langages à prototypes. In *Actes de LMO '95 (Langages et Modèles à Objets), Nancy, France*, pp. 93–109, 1995.

[4] D.G. Bobrow, M. Stefik. *The LOOPS Manual*. Memo KB-VLSI-81-13, Xerox Palo Alto Research Center, 1983.

[5] D.G. Bobrow and M. Stefik. Object-Oriented Programming: Themes and Variations. In *The AI Magazine (6)4*, pp. 40–62, American Association for Artificial Intelligence, 1986.

[6] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism.

In *ACM Computing Surveys (17)5*, pp. 472–522, 1985.

[7] B. Carré. The Point of View Notion for Multiple Inheritance. In *Proceedings of the OOPSLA/ECOOP Conference, Ottawa, Canada, ACM SIGPLAN Notices (25)10*, pp. 312–321, 1990.

[8] B. Carré, L. Dekker and J.M. Geib Multiple and Evolutive Representation in the ROME Language. In *Proceedings of TOOLS2, Paris*, pp. 101–109, 1990.

[9] C. Dony, J. Malenfant, and P. Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. In *Proceedings of the 7th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '92), Vancouver, British Columbia, ACM SIGPLAN Notices (27)10*, pp. 201–217, 1992.

[10] R. Ducournau. Y3/YAFOOL*: Le langage à objets*. Sema Group 1989.

[11] R. Ducournau, M. Habib, M. Huchard, M.-L. Mugnier, and A. Napoli. Le point sur l'héritage multiple. In *Techniques et sciences informatiques (14)3*, pp. 309–345, 1995.

[12] J. Ferber and P. Volle. Using Coreference in Object Oriented Representations. In *Proceedings of the 8th European Conference on Artificial Intelligence*, pp. 238–240, 1988.

[13] J. Ferber. *Objets et agents: une étude des structures de représentation et de communications en Intelligence Artificielle*. Thèse d'informatique, Université Pierre et Marie Curie, Paris 6, 1989.

[14] W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of the 8th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '93), Washington, DC, USA, ACM SIGPLAN Notices (28)10*, pp. 411–428, 1993.

[15] W.R. LaLonde, D. Thomas, and J.R. Pugh. An Exemplar Based Smalltalk. In *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), Portland, Orergon, ACM Sigplan Notices (21)11*, pp. 322–330, 1986.

[16] W.R. LaLonde. Designing Families of Data Types Using Exemplars. In *ACM TOPLAS (11)2*, pp. 212–248, 1989.

[17] H. Lieberman. *A preview of Act1*. AI memo No 625, Massachusetts Institute of Technology, 1981.

[18] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. In *Proceedings of the 1st Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '86), Portland, Oregon, ACM SIGPLAN Notices, (21)11*, pp. 214–223, 1986.

[19] J. Malenfant. On the Semantic Diversity of Delegation-Based Programming Languages. In *Proceedings of 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), Austin, TX, USA, ACM SIGPLAN Notices (30)10*, pp. 215–230, 1995.

[20] J. Malenfant. *Split Objects: Taming Value Sharing in Object-Oriented Languages*. Rapport de recherche IRO-968, Département d'Informatique et de Recherche Opérationnelle, Université de Montréal, 1995.

[21] O. Mariño. *TROPES*. Thèse d'informatique, Université Joseph Fourier, Grenoble 1, 1993.

[22] B.A. Myers, D. Giuse, R.B. Dannenberg, B. Vander Zanden, D. Kosbie, E. Pervin, A. Mickish and P. Marchal. Garnet: Comprehensive Support for Graphical, Highly-Interactive User Interfaces. In *IEEE Computer, 23(11)*, pp. 71-85, 1990.

[23] H. Ossher, M. Kaplan, W. Harrison, A. Katz and V. Kruskal. Subject-Oriented Composition Rules. In *Proceedings of the 10th Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '95), Austin, TX, USA, ACM SIGPLAN Notices (30)10*, pp. 235–250, 1995.

[24] F. Rechenmann, O. Mario and P. Uvietta. Multiples Perspectives and Classification mechanism in Object Representation. In *Proceedings of the 10th European Conference*

on Artificial Intelligence, Stockholm, pp. 425–430, 1990.

[25] R.B. Smith and D. Ungar. A Simple and Unifying Approach to Subjective Objects. To appear in *TAPOS special issue on Subjectivity in Object-Oriented Systems (2)3*, 1996.

[26] L.A. Stein, H. Lieberman, and D. Ungar. A Shared View of Sharing: The Treaty of Orlando. In *Object-Oriented Concepts, Applications and Databases, W. Kim and F. Lochovosky eds., Addison-Wesley*, 1988.

[27] P. Steyaert. and W. De Meuter. A Marriage of Class- and Object-Based Inheritance Without Unwanted Children. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP '95), Aarhus, Denmark, W. Olthoff ed., LNCS 952, Springer-Verlag*, pp. 127–144, 1995.

[28] A. Taivalsaari. *A Critical View of Inheritance and Reusability in Object-Oriented Programming*. Thèse d'informatique, Université de Jyväskylä, No 23, Finlande, 1993.

[29] D. Ungar and R.B. Smith. SELF: The Power of Simplicity. In *Proceedings of 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), Orlando, FL, ACM SIGPLAN Notices (22)12*, pp. 227–242, 1987.

[30] P. Wegner. Dimensions of Object-Oriented Language Design. In *Proceedings of 2nd Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87), Orlando, FL, ACM SIGPLAN Notices (22)12*, pp. 168–182, 1987.