On Including Part Hierarchies in Object-Oriented Languages, with an Implementation in Smalltalk.

Edwin Blake & Steve Cook

Department of Computer Science, Queen Mary College,
London University, Mile End Road, London, El 4NS.
Email: edwin@qmc-cs.UUCP

§ 1. Introduction.

In very many situations we consider objects to be constructed from parts. The parts in their turn may consist of smaller, simpler parts. And so forth. In engineering design we speak of assemblies and subassemblies; in science we describe things in terms of their components. However, when we want to model objects consisting of parts in Smalltalk, and many other object-oriented languages, we are confronted with a dilemma: either sacrifice the data encapsulation properties of the language or utterly flatten the whole-part hierarchy.

In this paper we describe what a part hierarchy is and why it is so important (§l.1). We then show how many object-oriented languages fail to provide this hierarchy (§1.2). After reviewing previous work (§1.3) we sketch how we added the structure to the Smalltalk (§2). The simple implementation details are outlined in an appendix. We discuss where in our experience the part hierarchy was useful and where it fails (§3). Lastly we stand back and examine the broader implications of adding a part hierarchy, and how a part hierarchy interacts with a single and multiple inheritance class hierarchy (§4).

It may roughly be said, to give a taste of the conclusions, that parts and wholes address a different dimension of generalization from that addressed by classes. Classes, like sets, provide a partial ordering structure which can be read as "is-a-kind-of" or "belongs-to", while parts and wholes provide a much more complicated "is-a-part-which-fits- there" structure.

1.1. What is meant by a Part Hierarchy and Why is it Important?

Things are often described in terms of parts and wholes; the way the division into parts is made depends on the purpose of the analysis. A part is a part by virtue of its being included in a larger whole. A pan can become a whole in itself, which can then be split into further parts. In this way we build up a hierarchy of parts and wholes, which we have called the *part hierarchy.* Rather than attempt a formal description of part-whole relations [Smith, 1982] we shall present a series of illustrative examples.

We distinguish between a mere *collection,* or additive whole, or heap, (e.g., a bag of marbles, a pile of electronic components) and a more *structured whole* (e.g., an animal, a wired-up electronic circuit). To the former we apply set theory, to the latter a part hierarchy.

It is also useful to distinguish between extensive *parts* — components, fragments, constituents, pieces — and non-extensive *attributes*—features, aspects, moments. For example: a table is made up out of a fiat top and four identical legs placed in the corners; these are the parts. A table also has a colour, which is an attribute rather than a part. We shall be more concerned with parts than attributes.

Part-whole analysis is crucial to science and technology. Pirsig [1974] provides a very good example of the hierarchical description of the assemblies and subassemblies used in engineering:

> A motorcycle may be divided for purposes of classical rational analysis by means of its component assemblies and by means of its functions.
>
> If divided by means of its component assemblies, its most basic division is into a power assembly and a running assembly.
>
> The power assembly may be divided into the engine and the power-delivery system. The engine will be taken up first.

The engine consists of a housing containing a power train, a fuel-air system, an ignition system, a feedback system and a lubrication system.

The power train consists of cylinders, pistons, connecting rods, a crankshaft and a flywheel.

The ignition system consists of an alternator, a rectifier, a battery, a high-voltage coil and spark plugs. ... etc.

That's a motorcycle divided according to its components. To know what the components are for, a division according to functions is necessary

Parts are also met in those branches of computation where physical objects are represented, for example, *model-based computer vision* and *computer graphics*. We shall be testing our implementation with an example which straddles these two fields: a stick figure [Marr & Nishihara, 1978] (see figure 1). The particular stick figure we shall discuss, called "joe", has numerous parts arranged hierarchically. For example, joe's legs have feet which have toes, and toes consist of phalanges.

Model-based vision draws upon the work on knowledge representation in artificial intelligence [e.g., Brooks 1981). *Frame-based* representation [Fikes & Kehler 1985] has similarities to the object-oriented approach. Frames describe parts and attributes by means of slots.

One of the standard texts on computer graphics [Foley & van Dam 1982] devotes a chapter to "Modelling and the Object Hierarchy". The proposed new graphics standard PE{IGS (Programmer's Hierarchical Interactive Graphics System) [1986] organizes objects in a structure hierarchy. Both *structure hierarchy* and *object hierarchy*, as used above, are synonyms for our part hierarchy.

PHIGS also has the novel concept of inheritance on a part hierarchy where attributes of the whole are inherited by the parts. E.g., the legs of the table could inherit the colour of the whole. The requirement is not quite as general as it at first appears. This "inheritance" is only used when the whole structure is traversed from the root down in order to display it, thus the wholes are always accessed before the parts and the attributes can therefore be stacked.

We adopt the policy that information is stored in the part hierarchy at its corresponding logical level: information about the whole is not stored in the parts, information about the parts which is not modified by the whole remains with the parts. Ideally the whole knows the parts but the parts do not know of the whole.

1.2. The Dilemma Posed by Parts in Object-Oriented Languages.

Data abstraction is a fundamental aspect of object-oriented languages. Data abstraction can be summarized as meaning explicit interface protocols and a hidden local state. When an object is assembled from its parts these parts are no longer independent. A part belongs to the local state of the whole and the interface is mediated by the owner.

If this requirement is strictly interpreted the existence of the parts should become invisible to the users of the whole. The whole protocol which a part understands, some of which it may implement perfectly adequately, will have to be reimplemented as the protocol of the whole. The net result is that the part hierarchy is replaced by a single monolithic whole as far as the external world is concerned.

On the other hand, if we did not include parts in the local state of the whole, external users could explicitly request the part and then modify it. The resulting changes could violate the integrity of the whole. The response of a part would also be independent of its position in the whole.

This is the dilemma. We would like the parts to remain visible, while at the same time access is mediated by the owner.

1.2.1. An Example of the Dilemma in Smalltalk.

Parts can be identified with a particular kind of instance variable that can be accessed via messages to read and assign the parts. In standard Smalltalk the parts can be accessed by putting together the message selectors of the various parts.

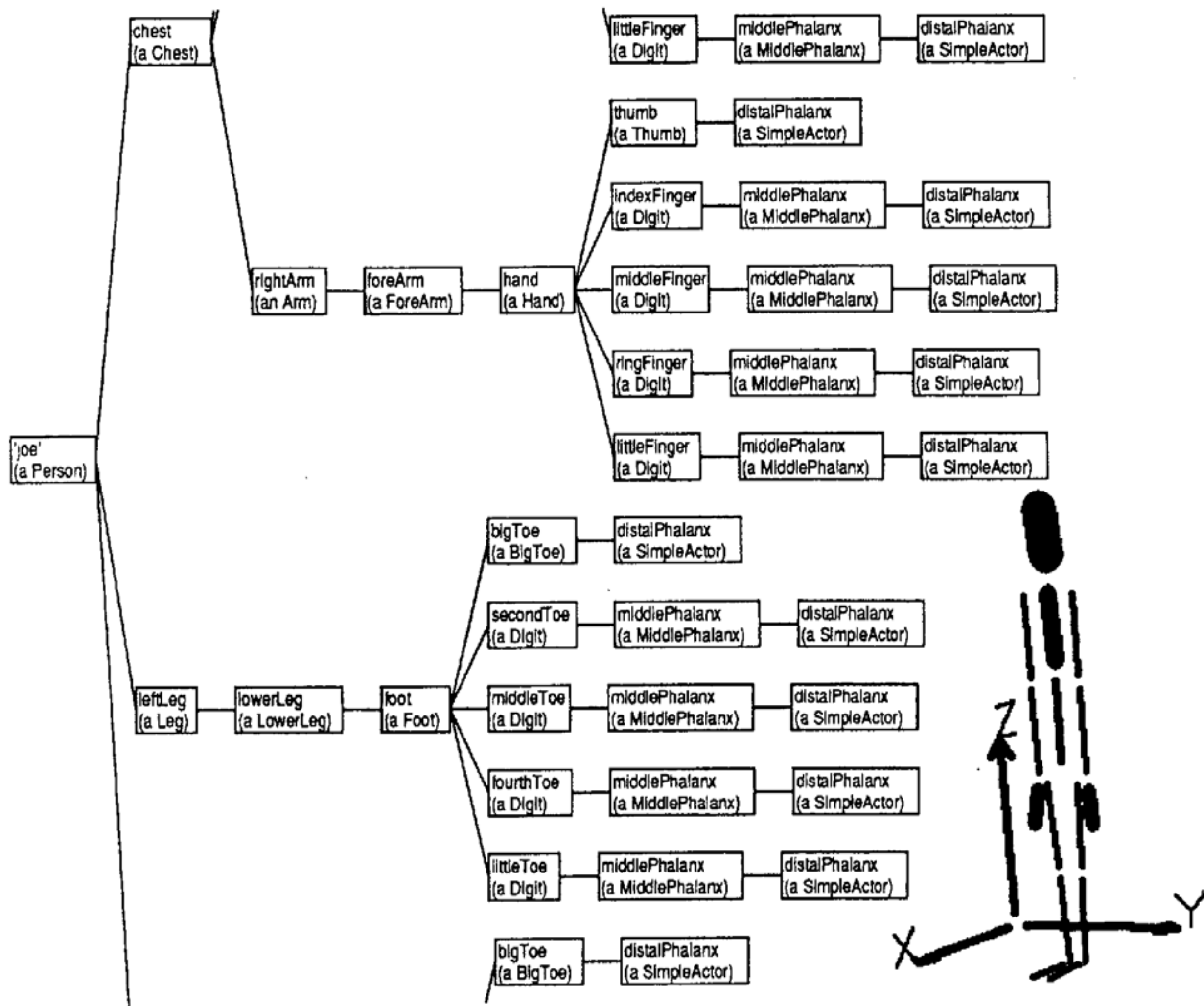

Figure 1. The part hierarchy for a stick figure implemented in Smalltalk. The boxes contain the name of the part and the default class to which it belongs. The figure itself is rendered on the right (without fingers and toes).

In our example of a stick figure (figure 1) we define the following classes:

Person with instance variables: chest, leftLeg, etc.

Foot has instance variables: bigToe, secondToe etc.

then:

joe ← Person new. "Create an instance of Person called 'joe"

joe leftLeg. "Return contents of the left leg instance variable."

joe leftLeg lowerLeg foot. "Return the left foot."

joe leftLeg lowerLeg foot bigToe wiggle. "Wiggle left big toe."

This standard syntax has the disadvantage that the actual instance variable objects themselves are handed out and the sender of the message can modify them at will. This subverts the idea that objects can hide and control their local state.

The alternative is to disallow the direct part access messages; but then the top level object needs to implement the whole message protocol for all its pans. For example, to wiggle the left big toe the class Person would need a message like 'leftLegLowerLegFootBigToeWiggle' in its instance message protocol. This collapsed message completely flattens the part hierarchy and so removes the conceptual advantages of factoring that knowledge in an intuitive manner.

One could attempt to intermix direct access to instance variables in "safe" cases, and use collapsed messages (e.g., leftLegLowerLegFoot) when access must be controlled, in an *ad hoc* manner. This is

really worse, because the external world needs to know when to send collapsed messages and when not. If 'joe' in our example above, gets his foot encased in plaster then a collapsed message would be needed instead of direct access and all callers would have to know this.

1.2.2. Class Hierarchies Do Not Provide Part Hierarchies.

Neither single nor multiple class inheritance seem to have any bearing on the part-whole dilemma mentioned above. Classes deal with specializations of objects. In single inheritance we have a partial ordering which provides the same structure as can be expressed by sets and subsets. Even with multiple inheritance only one set of instance variables can be inherited for each superclass. Thus, even if the class "Table" inherited legs four times over along different paths, it would still only have the characteristics of one leg.

If multiple inheritance were extended so that the instance variable data were somehow kept separate, the parts would still be defined at the same level as the whole. That is, the same object would contain both the complete part protocols and the protocols corresponding to their structured interaction.

1.3. Parts in Object-Oriented Programming.

The standard Smalltalk system provides for objects to have dependants. The normal operation is that parts are given backpointers to their owners, and the owner can then be informed of changes. However:

- Parts have to know which changes are significant to the owner. (i.e., high level knowledge at lower levels).
- If parts forward all changes indiscriminately to the owner object then the owner must again contain the equivalent of the protocols of all the parts.
- It creates circular structures, with attendant maintenance problems.

This system of storing knowledge of the whole in a part which is then handed out, is useful where the part hierarchy is shallow and where it is simpler to hand out parts to the external world. An example of such a "part" is the *model* in the Smalltalk model-view-controller mechanism.

Part-whole relations are not often catered for in object-oriented languages: Loops' *composite objects* [Stefik & Bobrow, 1985] being one of the few examples. ThingLab [Borning 1979, 1981] is a Smalltalk based system for simulating physical objects (e.g., geometric shapes, bridges, electrical circuits, documents, calculators). In ThingLab, objects consist of parts. The major contribution is a system for representing and satisfying constraints which exist between the parts. Multiple class inheritance hierarchies, and *part-whole hierarchies* are used to describe the objects and their interrelations. Parts are referred to symbolically by means of *paths* that name the nodes to be visited in proceeding down the part hierarchy. Our work extracts, analyses, and refines the concept of a part hierarchy first encountered in ThingLab and draws a clearer distinction between class hierarchies and part hierarchies.

1.3.1. Prototypes and Delegation.

As will be seen in section 2, our *implementation* of a part hierarchy makes use of message forwarding. Message forwarding is a powerful general notion which can also be used to implement delegation, which has long been used in actor languages in preference to the notion of class [Agha 1986]. This means that an object's message protocol includes those messages which can be delegated to prototypes or exemplars [Borning 1986; Lieberman 1986; LaLonde, Thomas & Pugh 1986]. For example, if we were modelling horses, the Platonic ideal horse would be a prototype and a particular nag in the field would delegate the responses to some of its messages to that ideal horse.

It is in this sense that multiple inheritance was implemented by means of parts in ThingLab. This creates a rather blurred distinction between a part hierarchy and a class hierarchy which we want to avoid in this paper. The notion of class is retained, as is the possibility of multiple class inheritance.

§ 2. A Mechanism for Modelling Objects with Parts.

We have given a concrete example of incorporating parts using standard Smalltalk (§1.2.1). This provides the motivation for making a small change to the standard syntax (§2. 1) which does allow a full part hierarchy to coexist with data encapsulation. The implementation of the extended language (§2.2) is achieved by a rather simple modification to the standard system.

For the rest of this section we shall be concerned with Smalltalk, but many of the remarks will apply to other object-oriented languages.

2.1. A Syntax and Semantics for Manipulating Parts in Smalltalk.

We want to recognize that parts are objects in their own right — active first class objects; but *without violating the principle of encapsulation*. A whole is allowed to hide its parts, even pretend it has parts which do not in fact exist as such. The whole has additional properties arising from the interaction of the parts, which should not be stored in the parts.

Our solution uses the notion of censored access. The whole forwards messages, possibly censored, to parts, and returns answers, also possibly censored, from parts. This necessitates a new kind of message selector: a *compound message selector* and a new concept: *message forwarding* of possibly censored messages.

Let us call the message selectors of standard Smalltalk simple selectors, whether they be unary, binary or keyword selectors. A compound selector is then a <path>+“.”+<simple-selector>. The path is a series of one or more part names also separated by full stops (“.”). Part names always begin with a lower case letter. The first part name in a path (its head, if it were a list) is called the prefix. A path provides a way of referring to a part further down the part hierarchy.

Our example (§l.2.1) then becomes:

```
joe leftLeg. “Return the left leg instance variable.”
joe leftLeg.foot.bigToe. “Return the left toe.”
joe leftLeg.foot.bigToe.wiggle. “Wiggle left big toe.”
```

The semantics associated with the construct is one of message forwarding. Either a class understands a full compound message selector or it does not. If it does then the method corresponding to the compound selector is executed and the result returned; this may involve sending a censored version of the message to the part.

If the compound message is not part of the protocol of the object then the prefix is stripped off. The rest of the (possibly compound) message is forwarded to the instance variable named by the prefix. In that case the answer to the forwarded message is also the answer to the compound message as a whole. This definition is clearly recursive.

If the Smalltalk convention of *private* messages is accepted as providing adequately strict encapsulation, then parts can always be accessed by means of automatically generated private messages. The message is then forwarded not to an instance variable, but to the *answer* returned by sending a part access message. This means that the parts need not necessarily be instance variables. They can be generated as and when needed. This could provide multiple views of the same object without storing all versions explicitly. This further extends the principle of information hiding.

2.2. Implementing the Part Hierarchy (see also appendix).

To implement the part hierarchy the following changes are needed:

1) Change the compiler to allow compound messages.
2) Extend the message selector class (Symbol) to provide access to the various parts of the compound messages.
3) Implement the mechanism for forwarding messages to parts.

4) (Optional) Add a class initialization method to add private messages automatically to access all parts.

The extended Smalltalk with multiple inheritance [Borning and Ingalls, 1982] is widely distributed. This provides (1) and most of (2) above, while (4) is an elementary exercise.

It is (3) which is the heart of the system. It uses a powerful general mechanism: message forwarding. When a compound message is initially sent it is only understood if the object needs control over that aspect of its part structure which the message accesses. If the compound message is not understood then the part named by the message prefix can safely handle the message. A method is then compiled just to forward the message. If the same message is sent again the code already exists and the message will be understood.

The forwarding of messages can be overridden in a completely natural way if it is later decided that an object *does* need control over that aspect of its parts. The controlling method is simply added with the compound selector which names the part, and it replaces the forwarder.

§ 3. Experience with Using the Part Hierarchy.

The importance of a part hierarchy in computer graphics and vision has been mentioned in the introduction, as was the example of a stick figure. The part hierarchy described the structured figure rather well (§3.1) but it is not designed for a collection of stick figures (§3.2).

3.1. Results from Modelling a Single Stick Figure.

A moving stick figure and camera was implemented using the part hierarchy defined above. The following points arose from this experiment:

1) Parts are compatible with the existing multiple inheritance implementation.
2) For deep hierarchies the message selectors become rather lengthy.
3) Some, more or less elaborate, form of typing is required for the parts of an object.
4) The class inheritance hierarchy can interact in unexpected ways with the part hierarchy.

The second point means we might require some more syntactic sugar by way of abbreviations. In the case of parts which are common to a number of subclasses it is, at present, possible to define an abbreviation explicitly in the superclass. For example, all limbs of the stick figure have a proximal joint which implements the local coordinates. A common message to the proximal joint is to alter its orientation, thus the superclass of limbs implemented the abbreviation “orientation:” for the compound message “proximalJoint.orientation:”.

The third point, concerning typing, uncovers an active area of current research, which is beyond the scope of this paper. The need for an instance variable type arises when a new object is created and its parts have to be instantiated. In order to instantiate the parts their classes have to be known. This can be coded as an initialization message to the new instance, but a more general solution is to associate a default class with each instance variable. When a new object is then created, the creation message is also forwarded to these classes in order to create the parts.

The fourth point concerns the interaction between the class inheritance hierarchy and the part (or modelling) hierarchy. Message forwarders are treated as normal methods, they apply to a whole class and are inherited by subclasses. This means that all instances in the same class are forced to have the same relationship to their subparts. Another problem arises when message forwarders are created in the class of the receiver of the compound message and not in the (super) class where the instance variable is actually defined. If the superclass is then modified to intercept the compound message it will have no effect in those subclasses which have already acquired message forwarders. In our implementation this problem was alleviated by making message forwarders visible to the user in a special category of message. A fairly effective solution to this problem would be to place the message forwarder in the method dictionary of the highest class in the hierarchy which possesses the relevant instance variable, rather than in the method dictionary of the original recipient of the compound message.

3.2. Elements of a Collection are not Parts.

In our experimental implementation of stick figures the one situation where the part hierarchy was at a distinct disadvantage was with collections of stick figures. This is perhaps not too surprising since wholes are meant to be more structured than sets. The collection itself has an associated coordinate system in which the figures are embedded, it may even have a graphical appearance. Thus the collection is a kind of "Actor", and is in some ways a whole just like a single stick figure.

The problem with using parts is two fold:

- The representation of a variable number of pans is rather difficult.
- The requirement that pans have explicit names, otherwise such an advantage, is here a liability. Objects in a collection are nameless, or at least their names change.

Representing a variable number of parts can be achieved in a number of more or less messy ways: a class for each cardinal number, allowance for empty part slots, or having indexed anonymous parts. All these options were attempted but had to be rejected. If classes are dispensed with and prototypes used instead, some of these objections might be met. Each particular collection with a fixed number of elements could perhaps have its own pan access messages.

However, an appropriate solution is provided by multiple inheritance of both a *Collection* to provide access to the objects and a basic class of figure *(Actor)* to convey the appearance and position of the collection. This was the strategy adopted. The individuals in the collection were accessed with the normal messages used on collections.

§ 4. Implications of the Part Hierarchy for Object-Oriented Languages.

At this stage of the paper we have established the general importance of part hierarchies in natural philosophy and engineering. We have shown how a simple extension of Smailtalk can provide, fairly elegantly, a part hierarchy mechanism. This mechanism has been illustrated and criticized by means of examples drawn from three-dimensional modelling. We are now going to characterize the results of this effort.

4.1. Part Hierarchies and Class Inheritance Hierarchies.

When it comes to a single inheritance language, the question is really how orthogonal the concepts of part hierarchy and class hierarchy are. That the *implementations* can have mutual dependencies has already been seen.

The class hierarchy can be taken to express an "IS-A" relation [Brachman, 1983]. Thus an integer IS-A number. More specifically in our view, if the inheritance hierarchy is anything more than an implementation tool then it exemplifies the "IS-A-KIND-OF" relation: integer IS-A-KIND-OF number. The part hierarchy on the other hand expresses an "IS-A-PART-OF' relation, or more accurately since the part does not know its owner it expresses the reversed "HAS-A-PART" relation.

If we adopt this view then the two concepts are orthogonal. The class hierarchy provides generalization and specialization, this has nothing to do with structured building up of objects from parts, except that general objects may sometimes have fewer parts than their specialized subclasses.

It has already been said that delegation provides an inheritance mechanism. Message forwarding, which we use to access parts, can be used for delegation. Therefore message forwarding can be used to implement multiple inheritance [Borning, 1986]. If we retain the notion of class (or maintain a distinction between prototypes and other objects), then we can regard this as an implementation issue and not a fundamental relation.

Multiple inheritance is preferred when objects are unstructured collections. When objects have a fixed structure, or when parts are subservient to the emergent properties of the whole, then a part hierarchy is better. A part hierarchy is thus a valuable extension to both single and multiple inheritance object-oriented languages.

4.2. Part Hierarchies and Encapsulation.

At first sight it might appear that part hierarchies violate the principle of data encapsulation and information hiding, because they provide access to the parts of an object. In our view the opposite is actually the case. Part Hierarchies provide better control over access to the parts than is found in many object-oriented languages.

It is true that the internal structure of the object is made visible. But this is strictly mediated by messages. These messages allow full control and even allow the whole to pretend it has parts, by providing messages which mimic the behaviour of parts without actually implementing them explicitly.

The need to see structure of an object is similar to the need for “grey-boxes” rather than “black-boxes” in engineering. Depending on the level at which we are designing we need more or less of the structure of our wholes to be visible.

§ 5. Conclusion.

Part hierarchies have been incorporated in object-oriented languages in a way which strengthens the data abstraction properties of these languages. Such part hierarchies are most useful in simulation, where we are concerned with structured physical objects.

Part hierarchies provide facilities not provided by class hierarchies. This is because class hierarchies provide less structure than part hierarchies, they provide the structure of sets, and sets contain no notion of relative position or multiplicity. Part hierarchies are less useful for describing sets and collections and multiple inheritance may be needed to deal with these. Part hierarchies can be regarded as being an orthogonal notion to class inheritance hierarchies.

5.1. Further work: Multiple Views.

Part hierarchies can implement multiple representations of knowledge without requiring there to be an independent object for each representation. For example: consider a point defined in either Cartesian coordinates or in polar coordinates. If we wish to implement both views then the polar coordinates can always be calculated from the Cartesian coordinates, even though no such “parts” are actually stored [Borning, 1979, “virtual parts”].

Multiple views are used in certain kinds of knowledge representation. Philosophers [e.g., Pirsig 1974] emphasize that the division into parts is arbitrary and depends on the purpose of the analysis. It does seem that a small number of alternate views (e.g., polar vs. Cartesian coordinates) can be accommodated, but whether it is a general extensible mechanism requires further investigation.

§ 6. Appendix: Implementing a Part Hierarchy in Smalltalk.

We assume that the Smalltalk has been extended allow multiple inheritance [Borning & Ingalls, 1982]. This provides the changes to the parser which allows compound messages and the basic modification to the class Object’s instance method for the message “doesNotUnderstand:”, This sends a message to the class of the receiver with the selector “tryCopyingCodeFor:”.

The class method “tryCopyingCodeFor:” is the one we override in the super (meta) class of all objects which are to have parts. If the message sent was not compound or was sent as part of the multiple inheritance implementation we treat it as before. If it is a compound selector and the prefix starts with a lower case letter (excluding “super” and “all”) we compile a message forwarder.

The message forwarder is simply a method which has as selector the complete compound message with all the keywords and arguments. The method sends the prefix to self and then the rest of the compound message. The answer of the method is that result. For example:

To forward:

```
joe leftLeg.lowerLeg.foot
```

the method is:

```
↑self leftLeg lowerLeg.foot
```

These methods are synthesized and compiled without saving the source code by using the class message "compileUnchecked:". They are classified as 'message forwarders' for access in the browser. It is not essential for message forwarders to be visible in the browser, but if they are not it can make the behaviour of programs rather non-obvious to the programmer. The appropriate class messages are "organization classify:under:".

The remaining class methods are equally simple. These are methods to compile code to access parts (instance variables) by name automatically. In this case it is useful to save the source code, since fast compilation is no longer important. A dictionary which associates instance variable names with default classes should also be created and stored as a class variable. This dictionary can be used when parts belonging to a whole have to be instantiated.

§ 7. References.

Agha, G. (1986) SIGPLAN Notices **21**, 10 58-67. "An overview of actor languages."

Borning, A.H. (1979) ThingLab: A constraint-oriented simulation laboratory. Xerox Palo Alto Research Center report SSL-79-3, **a revised version of** Stanford University PhD. thesis, Stanford Computer Science Department Report STAN-CS-79-746.

Borning, A.H. (1981) ACM trans. Programming Languages and Systems **3**,4 353-387. "The programming language aspects of ThingLab, a constraint-oriented simulation laboratory."

Borning, A.H. (1986) IEEE/ACM Fall Joint Computer Conf., Dallas, Texas, Nov 1986. pp. 36-40 "Classes versus prototypes in object-oriented languages."

Borning, A.H. & Ingalls, D.H.H. (1982) Proc. Nat. Conf. Artificial Intelligence, Pittsburgh, PA. pp. 234-237. "Multiple inheritance in Smalltalk-80."

Brachman, R.J. (1983) Computer **16**, 10 30-36. "What IS-A is and isn't: an analysis of taxonomic links in semantic networks."

Brooks, R.A. (1981) Artificial Intelligence **17** 285-348. "Symbolic reasoning among 3-D models and 2-D images."

Fikes, R. & Kehler, T. (1985) Comm. ACM **28** 904-920. "The role of frame-based representation in reasoning."

Foley, J.D. & van Dani, A. (1982) Fundamentals of Interactive Computer Graphics. Addison-Wesley. Reading, Massachusetts.

LaLonde, W.R., Thomas, D.A. & Pugh, J.R. (1986) OOPSLA'86: SIGPLAN Notices **21**, 11 322-330. "An exemplar based Smalltalk."

Lieberman, H. (1986) OOPSLA'86: S1GPL4N Notices **21**, 11 214-223. "Using prototypical objects to implement shared behavior in object- oriented systems."

Man, D. & Nishihara, H.K. (1978) Proc.R.SocLond. B **200** 269-294. "Representation and recognition of the spatial organization of three- dimensional shapes."

Moon, D.A. (1986) OOPSL4'86: SIGPLAN Notices 21, 11 1-8. "Object-oriented programming with Flavors."

PHIGS (1986) Programmer's Hierarchical Interactive Graphics System. ISO PHIGS revised working draft, 13 June 1986.

Pirsig, R.M. (1974) Zen and the art of motorcycle maintenance: An enquiry into values. The Bodley Head.

Smith, B. (1982) Parts and Moments. Studies in Logic and Formal Ontology. Philosophia Verlag. Miinchen.

Stefik, M. & Bobrow, D.G. (1985) The Al Magazine. **6**, 4 40-62. "Object-oriented programming: themes and variations."