Chapter 3

A Framework for Object-Based Programming Languages

3.1 Introduction

In this and the next chapter we consider all the different aspects of a framework for the implementation of object-based and object-oriented programming languages (OOPL). As is the case for frameworks in general, a framework for object-oriented programming languages, is a skeleton implementation of an object-oriented programming language. It represents a theory for how to design and implement object-oriented programming languages.

A framework is more than a mere implementation. Extensibility must be taken into account. The framework reifies the important constituents of an OOPL, such as objects, in an abstract way. Furthermore during the design of the framework we were led by the above discussed principles of compositionality and full abstraction.

A framework for a programming language in general, and for an OOPL in particular, must incorporate, in principle, all the different aspects associated with the implementation of that programming language. The different components, i.e. compiler, evaluator, program browser, … of an entire programming environment must be taken into consideration. We restrict our attention to the representation of programs and their evaluation. Since for object-oriented programming languages the result of evaluating a program is an object we will also consider the representation of objects.

The proof of the pudding is in the eating. This is especially so for object-oriented frameworks. We will test the framework by specialising it to different OOPL and by incorporating different language features. In this chapter we will confine ourselves to a very simple object-based programming language. Extension to a full-fledged object-oriented programming language will be done in the next chapter.

We will proceed as follows. We will first analyse the different design issues concerning OOPL. For our purpose we need a coherent framework of concepts that clearly delineates a design space of languages that can be called object-oriented. As we said in the introduction, examples can be found of different coherent frameworks of language concepts that are each called object-oriented. Our point of departure will be the notion of strongly encapsulated polymorphic objects that have a well-defined behaviour and interface. Inheritance will be briefly discussed, just enough to motivate that a thorough discussion on inheritance can be deferred to the next chapter. The argument for this approach is that at least for one particular kind, inheritance can be fully encapsulated in the internal implementation of objects. Furthermore this allows us to build a framework based on objects alone (no classes, no inheritance, no delegation), that can be specialised later on to include inheritance.

The next step will be the introduction of object-oriented frameworks. We will discuss what language concepts in the object-oriented paradigm support the construction of reusable programs and what must be added to construct open system. Reusability in object-oriented programs is partly based on inheritance. Since we deferred a thorough investigation of inheritance to the next section, only a simple form of inheritance will be used in our treatment of reusability.

A simple object-based programming language (Simple) is then presented for which the semantics is given in the form of a calculus for primitive objects. This calculus is discussed. It conforms to all of the previously adopted criteria. Although not fully formalised yet, it is mature enough to serve our purposes here. Similar calculi are being proposed in the literature [Abadi&Cardelli94] [Dami93] .

An initial set of abstract classes that form a framework is proposed. This initial framework incorporates our strong notion of objects. It is used to implement Simple. While trying to extend Simple, the limitations of this initial proposal are shown. The framework is adapted accordingly.

3.2 Design Issues in Object-Oriented Programming Languages

In the object-oriented research community there is still confusion as to what is essential for object-oriented programming, or even what it means to be object-oriented. There is no consensus about the central notion of what an object is, or should conform to. Different camps can be identified.

The Operations and State Camp

Probably the most popular conception of an object is that of a collection of operations that share a changeable state. The set of operations defines the interface of the object, and an object can only be accessed via the interface. The state of an object is hidden for all users of the object, but shared by all operations of the object, i.e. state changes by one operation may be seen by subsequent executed operations. The operations are called methods, the hidden state is mostly realised by instance variables. The concept of a hidden state is a particular form of the concept of encapsulation.

An important correlated notion is that of *object identity*. In an OOPL that supports identity each object is assigned a unique identity. This identity is independent of the values of the object's hidden state. Identity is kept over state changes and can be used to uniquely refer to an object. Given two objects it can be tested whether they have the same identity. Sometimes objects can explicitly ask to change identity without changing state.

The definition of objects employed in this camp is a very operational one. It is too much directed towards imperative programming languages. The more abstract definition that will be employed in our work captures it as a special case.

The Classes+Inheritance Versus the Objects+Delegation Camp

One of the most debated issues of object-oriented systems is *inheritance*. From a *practical* point of view inheritance may be considered one of the most important contributions object-orientation has made. The whole notion of *software reuse* and of *incremental definition* of software systems has gained wide-spread acknowledgement due to the concept of inheritance.

Inheritance and classes are closely linked since in most languages only classes (rather than objects) can be inherited from. This has led to the almost general belief that object-oriented = objects + classes + inheritance [Wegner87], i.e. a distinction is made between *object-based languages*, that do not have classes, *class-based languages*, that do have classes but no inheritance and *object-oriented languages* that have both classes and class-based inheritance.

Classless languages generally employ a *delegation* mechanism rather than an *inheritance* mechanism. If inheritance specifies behaviour sharing at the level of classes, then delegation specifies behaviour sharing at the level of objects. Delegation mechanisms may vary in the amount of flexibility that is supported, i.e. with pure delegation the sharing pattern may dynamically vary after an object has been created, in a more restricted form the delegation structure is fixed after an object has been created.

Although some work has been done to harmonise classless delegation and class-based inheritance [Stein,Lieberman&Ungar89] and proposals are made for integrating them [LaLonde,Thomas&Pugh86] [Stein87], both are still considered as fundamentally different language concepts. As we will illustrate further on, at the semantic level the differences between pure delegation and inheritance seem to confirm this feeling. Moreover we will show that neither satisfies our design criteria, and that a new notion of inheritance needs to be developed. We will show how a particular form of object-based inheritance, on the basis of mixin-methods, is the right compromise between class-based inheritance and classless delegation.

Finally note that for people in the classes+inheritance camp, the concept of objects is a very broad concept. All that is needed is that it must be possible to support some notion of inheritance. OOPL are not classified according to the nature of objects, but rather they are primarily classified according to the presence and the nature of classes and inheritance. Given the large variety of kinds of objects, all with sometimes fundamentally different properties (as will be illustrated below), inheritance is at least a suspect property for a primary classification.

The Polymorphism Camp

Objects are often associated with polymorphism. If we define the protocol of an object as the set of operations that can be applied to that object, then an object-oriented language is polymorphic if each object can be transparently substituted by all other objects that have at least the same protocol. We say that an object can be transparently substituted by another object if the latter object can be used in any program context in which the former object can be used without modification of neither objects, nor the program context. In a polymorphic programming language any object can be filled in a particular program context, if it has a protocol that subsumes the expected protocol. Take for example an operation that expects an object with a particular protocol. According to the polymorphism camp, this operation must be transparently applicable to all objects that implement this protocol.

Note that this form of polymorphism only says something about being able to substitute objects with a comparable protocol, it does not say anything about whether such substitution is *meaningful*. To decide whether a substitution is meaningful the behaviour (i.e. the abstract description of the effect of the applicable operations on an object) of the substituted objects must be taken into account. We will have more to say on this later.

The above form of polymorphism is called inclusion or *late-binding polymorphism*. In contrast with for example parametric polymorphism it is independent of type issues. With parametric polymorphism an operation can be applied to arguments of different types but typically has to rely on case analysis on the actual types of the arguments to perform its action.

Late-binding polymorphism is obtained in different ways. The first and foremost way is by means of message passing. An operation is not applied to an object (as for example in applying a function to some value), but rather an object is asked to perform an operation by sending it a message. It is the responsibility of the object itself to select the corresponding operation to perform. Obviously objects that understand the same messages can be substituted for each other.

A second popular way to achieve late-binding polymorphism is through overloaded functions, as exemplified by languages based on multi-methods. With multi-methods, all operations are overloaded with all the objects to which they are applicable, i.e. with one abstract operation (identified, for example, by its name) different concrete variants of the operation are associated. When an operation is applied to an object the correct variant is automatically selected. An object' s protocol consists of all those operations in which the object is used as a determinant for overloading (things might be a bit more complicated since an operation may be overloaded on different arguments, but we will ignore that for the moment). Again, objects that share the same protocol can be substituted freely.

The Data Abstraction Camp

As already mentioned the idea of an object as a set of operations that share a state is a very popular one. This view on objects is a particular case of the more general idea about objects as data abstractions, i.e. data-representations that are only accessible through a separately defined set of operations. The particular form of data abstraction used in object-oriented programming languages is also called encapsulation.

The problem here is that many different forms of data abstraction exist. As will be discussed data abstraction, as found in abstract datatype programming languages, and encapsulation, as found in object-oriented languages, are different forms of data abstraction. Furthermore, within the object-oriented paradigm itself, different forms of encapsulation exist. The two most notable are *class-based encapsulation* such as can be found in for example C++ [Ellis&Stroustrup90] and *object-based encapsulation* such as can be found in for example Smalltalk [Goldberg&Robson89]. We find it important to make a clear distinction between them.

The Object-Oriented Typing Camp

Type-systems are an important issue in understanding object-orientation. They are prototypical for what kind of static information can be attached (in case of type checking) or gathered (in case of type inference) to object-oriented programs.

Almost all, if not all, type-systems for object-oriented languages are based on object interfaces (or protocols), i.e. objects are annotated with interface specifications. It is primordial for the typing of object-oriented programs that it can be checked whether an object conforms to its formally declared interface specification. Therefore the interface of an object must be explicit in its definition, i.e. the interface must be formally derivable from an object definition. This must also be true when inheritance is introduced. The inheritance mechanisms must be such that the interfaces of newly created objects can be derived.

Interfaces are not typical for object-oriented programming alone. They play an equally important role in abstract datatypes. Nor are polymorphism and data encapsulation alone typical for object-orientation. It is typical for OOPL that explicit interfaces are intimately connected with late-binding polymorphism and object-based encapsulation to form the intuitive notion of an object as a self contained entity that has a well-defined behaviour and responds to a well-defined set of messages. It is this notion of objects that will be explored.

This section is an overview of the design issues involved while designing an OOPL. This is not an easy task. There have been an abundant amount of proposals for object-oriented languages and language features since the conception of object-orientation.

We will use explicit interfaces as a first yardstick. We feel that although the notions of classes and inheritance are important, the emphasis should be put on objects and on how new objects can be derived from old ones in a fashion that interfaces are derivable also. Up to the present there is a dichotomy between class-based languages that use inheritance (interfaces can be derived) and prototype-based languages that use delegation (interfaces can not be derived) as code reuse mechanisms. We will explore the essential differences between the two and formulate an alternative, object-based inheritance, for which interfaces are derivable and explore the consequences.

Encapsulation and polymorphism will be used as a second yardstick. The distinction between object-based (typical for OOPL) and module-based (typical for abstract datatypes) encapsulation will be elaborated upon. We will explore some variations on encapsulation.

3.2.1 Objects, Interfaces, Messages and Encapsulation

The goal of this section is to clearly delineate our notion of an object. It is the intention to give a more elaborate definition of the intuitive idea of an object as a self contained entity that has a well-defined behaviour and responds to a well-defined set of messages. We will show that the notions of *explicit interfaces*, *object-based encapsulation* and *late-binding polymorphism* are intimately connected. This informal definition of objects will then be used as a yardstick to evaluate possible design decisions in the construction of an object-oriented language. The definition of objects employed in this text is:

> **Object**: Objects can only be operated on by sending messages. An object responds to a finite set of messages. A message consists of a receiver and a selector. Selectors are abstract distinguishable entities. The result of sending a message to an object is again an object. Message passing is an atomic operation: for the sender of a message the result only depends on the combination of the receiver object and the selector, and the way a receiver implements its response to a message is entirely hidden. For each given object the set of messages it responds to is known. Two objects are equal when they respond to the same messages with the same results.

Objects that conform to the above definition will be called *substitutable objects*. In the remainder of this section we illustrate and elaborate on this definition. We will elaborate on the notion of selectors, the atomicity of message passing, the way objects implement their response to a message and how this all goes together with the notion of encapsulation.

The next figure shows an example person object in a graphical notation. Each node corresponds to an object, the messages an object responds to are represented as outgoing arrows, the selector of the message is placed above the arrow, the value is the target of the arrow. In a textual form message expressions are represented as 'o.x' or simply 'ox' or 'o(x)' for sending a message with selector 'x' an object 'o'. It should be noted that the graphical notation is an informal notation of a person object.

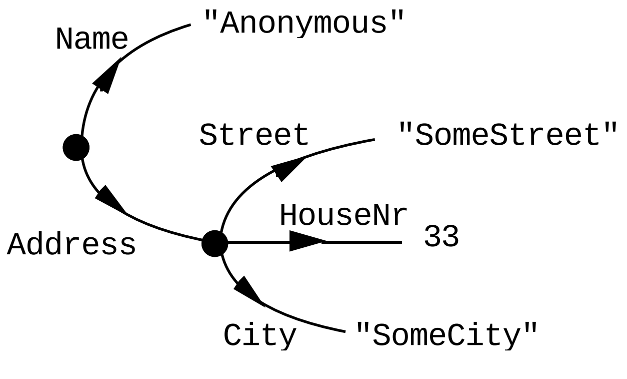

Figure 3.1

Selectors are abstract (syntactic, noncomputed) distinguishable entities. Selectors are not first class values (or objects). Stated in programming language terms the message part (the x in the message 'o.x') of a message expression is not evaluated. Selectors need to be distinguishable since they are used to identify the different messages an object responds to.

The collection of messages an object responds to is traditionally called its *interface*. It is important that objects have an *explicit interface* that is part of their definition. That is to say, the interface of an object is determined totally by the object's definition and an object should always respond to the same messages in a given context. An object's interface should not be determined by the context in which the object is used.

Two objects are *extensionally equal* if they both respond to the same messages with the same results. The interface of an object, together with a description of how an object responds to the messages that are sent to it, is commonly referred to as an object's *behaviour*. We can restate the above definition as: two objects are extensionally equal if they both have the same behaviour.

Objects can be extensionally defined as a finite collection of named attributes, i.e. mappings of selectors (the name of the attribute) to objects (the value of the attribute). Message passing then corresponds to attribute selection. The result of a message is the object associated with the attribute with the name that corresponds to the selector of the message. Extensional definitions of objects are reminiscent of the more conventional record structures rather than our intuitive idea of objects. It is no surprise that records are extensively used to model certain features of object-oriented programming. Examples can be found in type systems of OOPL and modelling of inheritance in class-based OOPL. We will come back to this in the next section.

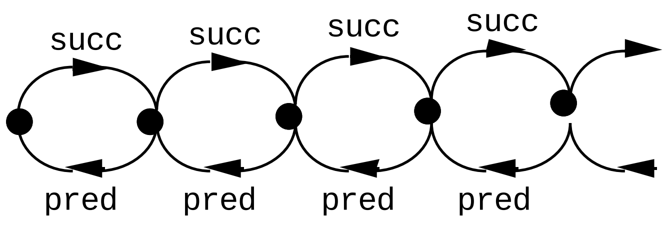

Figure 3.2

An extensional definition of an object is but a mere list of how each name in the objects interface is mapped onto its attribute value (that can be an object definition again, of course). Since extensionally defined objects must directly map each message to a resulting object (without a computation for example), extensional definitions are limited in their ability to express the more interesting object structures. Essentially they are limited in the possibility to define recursive object structures (such as the natural numbers in the above figure).

Intentional definition of objects

An *intentional definition* of an object defines for each name how its value is computed. Attributes can be defined in different ways. An attribute can directly contain a stored value i.e. the attribute is defined by a simple value; or its definition can depend on other attribute values. The former are conventionally referred to as instance variables, the latter are usually referred to as methods.

Message passing is an 'atomic' operation. When a message is sent to an object we say that the corresponding attribute is selected. This means that the corresponding attribute's definition is looked up and evaluated and the result is returned. We will refer to the former as attribute lookup and the latter as attribute evaluation. For example, for a method this corresponds to what is conventionally called *method lookup* and *method invocation*. From the stand-point of the sender of a message method lookup and invocation are unobservable parts of the atomic message, i.e. they form an indivisible whole.

An attribute, either public or private, is selected by name. The result of this can be, among others, a computed value or a side effect (e.g. in the case of a method) or simply a stored value (e.g. in the case of an instance variable) or a combination of these, all depending on the type of attribute. Each type of attribute can have its own *attribute selection rule*.

Encapsulation
The way and the means to compute an attribute's value, or even the fact that it is a computed value rather than a 'stored' value, remains hidden for the sender of a message. This is what is commonly referred to as encapsulation. We will refer to it as *object-based encapsulation* to make a distinction with other forms of encapsulation. For example, some object-oriented languages employ a form of *class-based encapsulation*. With class-based encapsulation attributes can be declared private (as before), but in contrast with object-based encapsulation, all objects of the same class can invoke or access each others private attributes. Class-based encapsulation is more akin to the more general form of *module-based encapsulation*.

We will call all the objects that send messages to some distinct object, the *instantiating clients* of this distinct object (the terminology might seem bizarre here but will become clear later on). Object-based encapsulation then means that an object is free to use 'private resources' to realise its behaviour, that the instantiating clients have no access to. Conversely this also means that an object can only realise its behaviour by making use of its own attributes and private resources. The private resources available to an object will be called its *private or encapsulated attributes*.

Encapsulation of Acquaintances
Encapsulation is based on how an object can gain access to other objects. In the spirit of Actor languages [Agha86], we will call all the objects that a distinctive object has knowledge of, or can directly refer to, the *acquaintances* of that object. We will call an object the owner of its acquaintances. Two complementary aspects of encapsulation can be identified. The one side has to do with to what extent an instantiating client can gain access to the acquaintance of the object of which it is instantiating client. The other side has to do with how an object can gain new acquaintances.

Object-based encapsulation is the consequence of the fact that an object has total freedom in the way it realises its behaviour. This ensures that the implementation of an object remains hidden for its instantiating clients. This includes the fact that an object can use, in its implementation, other objects or acquaintances that are not directly accessible by instantiating clients.

With object-based encapsulation instantiating clients are given access to an object's acquaintances at the initiative of the owner object. An owner object gives access to one of its acquaintances (whether already in existence or newly created) as a result of message passing, i.e. by returning an acquaintance as result of a message. An owner object that does not grant access to any of its acquaintances can be called an *autistic object*, since no information can be retrieved from this object. An object that grants access to all of its acquaintances is a *nonencapsulated object*. In OOPL where acquaintances are stored in named instance variables, encapsulation is tantamount to restricting (read) access to these instance variables. Notice however that this does not exclude *public instance variables* as long as a distinction can be made between instance variables that are public, and instance variables that are encapsulated.

The complementary sort of encapsulation is based on how an object can gain references to other objects. Once again the term autistic applies to objects that can not gain any new acquaintances, and nonencapsulated applies to objects that can gain new acquaintances without restriction. The set of acquaintances of an object can be restricted such that it is determined at any time by the initial set of acquaintances (the ones that where available when the object was created) and the acquaintances gained as arguments of message passing.

Obviously, it is not a violation of encapsulation that an object can gain new acquaintances by creating new objects on the one hand and by sending messages to its acquaintances and retrieving the result on the other hand. The other way around, an object must be given an initial set of acquaintances when it is created. The initial set of acquaintances is given by the object that creates. Similarly, new acquaintances are gained during message passing. An instantiating client can explicitly pass a set of acquaintances as arguments of a message.

In contrast with the former sort of encapsulation, this latter form is generally not accounted for in present day OOPL. A simple case where this encapsulation is violated is in OOPL that have global variables. In that case the objects contained in the global variables are acquaintances of all the objects present in the system. It is obvious that the set of acquaintances of a particular object can unrestrictedly change.

Encapsulation of Methods
It is obvious that apart from the ability to encapsulate acquaintances, it should also be possible to encapsulate methods. Again it is important to make a distinction between module-based and object-based encapsulation of methods. Present day OOPL either lack the possibility to encapsulate methods (Smalltalk [Goldberg&Robson89]) or employ a form of module-based encapsulation of methods (Self [Ungar&Smith87], C++ [Ellis&Stroustrup90]). The latter usually takes the form of privacy attributes that are attached to methods. All objects of the same class can invoke each others private methods, objects of different classes can only invoke each others public methods. So called private methods and their usage will be discussed in the section on scoping.

It is possible to employ object-based encapsulation for methods. Questions that must be answered are: how are these methods invoked, can private methods be overridden, what about visibility of encapsulated methods that are declared in an ancestor ? These questions will be answered in the section on scoping.

Languages that provide objects with a uniform access to both state and behaviour are called *slot-based languages* or also *languages that blend state and behaviour* [Ungar&Smith87]. Such languages typically feature uniform access to private and public methods and private and public instance variables; all of which are accessed through message passing.

Encapsulation of Inheritance
Not much has been said about inheritance yet. The reason is that the fact whether an object has been created as an instance of a class and whether this class inherits from another class or whether an object is created as a copy of another object or whether it directly inherits or delegates to another object should be encapsulated in that object. For an instantiating client only the behaviour of an object is important, not how this behaviour is realised. In subsequent sections a second kind of clients will be introduced. Clients that *do* care about how an object' s behaviour is realised. These are the so called *inheriting clients*, i.e. classes or objects that inherit from the class or object they are client of.

Late-binding Polymorphism
Late-binding polymorphism is the result of the fact that the same method name can be used by different objects. Each object can associate a different body (definition) to this method name. When a message is sent the appropriate method body is selected according to the receiver-object. This leads to polymorphic code since objects that implement the same protocol can be intermixed.

Late-binding polymorphism is inherent to the objects as defined above. Since each object has total freedom in the way and the means to compute an attribute's value (encapsulation), in case of methods, each object can associate different method bodies to the same method name. So, in our definition of objects, polymorphism and object-based encapsulation are two sides of the same coin !

3.2.2 Alternative Object Models

In order to further delineate our view on OOPL we will sketch four alternative approaches to object-oriented programming and we will show in what respect they do not correspond to our notion of objects.

Multi-Methods
Multi-methods form the basis for a popular class of object-oriented programming languages including CLOS [Moon89]. Multi-methods depart from the idea that messages are passed to a single distinct receiver. The goal is to construct a more powerful form of message passing where multiple 'objects' can participate in method lookup. Multi-methods were introduced, at first, to integrate object-oriented concepts into a functional language (Lisp), but were also inspired by the observation that message passing to a single distinguished receiver is in some cases awkward. This is the case with e.g. most hybrid binary arithmetic operations.

In a typical language employing multi-methods an object is defined by giving a list of its instance variables. Methods are declared separately. They are defined as (runtime) overloaded functions [Ghelli91]. With each method name different definitions can be associated, each distinguished by the number of formal arguments and a specification for each formal argument on which collection of objects a particular method definition is applicable. So, with each method name different method definitions can be associated. Which exact definition is to be used when a method is invoked depends on the actual arguments.

```
CartesianPoint = Object x,y:Real EndObject;
PolarPoint = Object rho,theta:Real EndObject;

Method sum (p1:CartesianPoint; p2:CartesianPoint)
  ^CartesianPoint(p1.x + p2.x, p1.y + p2.y)

Method sum (p1:PolarPoint; p2:PolarPoint)
  ^PolarPoint(…)

Method sum (p1:CartesianPoint; p2:PolarPoint)
  ^…

Method sum (p1:PolarPoint; p2:CartesianPoint)
  ^…

cPoint:CartesianPoint(1,1) ; pPoint:PolarPoint(2, pi);
sum(cpoint, pPoint)
```

At first sight multi-methods seem to be a generalisation of objects and message passing. If we were to use a function notation for message passing, then the function name would correspond to the message name and the first parameter of the function call to the receiver of the message. Furthermore, the receiver would play a special role since it is used to determine the exact function or method definition to be used. The overloading that goes with multi-methods and the late-binding polymorphism that was discussed in our definition of objects above are seemingly similar notions. Whereas with pure message passing only the receiver determines the exact method to be selected, with multi-methods all parameters can be used in determining the exact method definition that is to be selected. This gives multi-methods a gain in expressiveness. With the gain in expressiveness, however, comes a loss in object-encapsulation.

In contrast with the single receiver approach where each method truly belongs to one object, each method in the multi-method approach belongs to all of its arguments. As such the implementation details of none of the arguments is hidden in a method's definition. As a result programming languages supporting multi-methods typically do not support encapsulation. One notable counter-example to this is Cecil [Chambers92]. The solution, in fact, exists in combining overloading with encapsulation. All arguments in the multi-method that are overloaded (i.e. that play a role in determining the exact method-body that will be invoked) are regarded as non encapsulated for that method; all other arguments are encapsulated. So the encapsulation problems with multi-methods can be amended.

The lack of support for explicit interfaces is a more fundamental difference with the model of objects as put forward in the previous section. Intuitively, objects defined in a language employing multi-methods do not have the flavour of self-containedness typically ascribed to objects. This stems from the fact that in such languages the set of messages an object responds to is determined by the context in which an object is used, rather than by the object's definition.

Consider the point example from above. It is easy to construct an alternative example where two entirely different sets of methods are defined that are both applicable to e.g. polar points. Each set of methods is defined in a different context (e.g. in some local scope or for example in a 'package' or 'module'). Depending on the context in which a certain polar point is used it will respond to two entirely different sets of messages.

Although multi-methods are certainly useful, they do not conform to our notion of object-orientedness. The lack of support for encapsulation has already been noted, but appropriate solutions to provide encapsulated multi-methods have been provided in other work [Chambers92]. Essentially the lack of explicit interfaces, or rather that the interface of an object is not determined by the object's definition but by its surrounding context, seems to us a more profound difference with what is intuitively called object-oriented.

Objects with an Explicit Method Dispatcher
Objects with an explicit *method dispatcher* can be found in languages for concurrent object-oriented programming [Agha86][America87], but also (and again) in attempts to embed object-oriented features into Lisp-like languages [Abelson&Sussman84]. The central idea is that each object has a 'main' part or body. This main part is responsible for 'method dispatching'. Message passing has more or less the conventional form, that is, a message expression is composed of a distinct receiver and a message. The receiving side, however, takes quite a different form.

The programmer of the receiving object is responsible for deciphering the messages that have been sent and for deciding what action to take accordingly. A process that is normally part of the implementation of objects, and thus invisible for the programmer. The part of the receiving object that is responsible for this deciphering is usually called the 'method dispatcher'. For the purpose of method dispatching a message is explicitly constructed out of a message name and arguments. A typical method dispatcher is nothing but a simple procedural implementation of a mapping of a message name to an associated internally declared procedure (for example a simple Pascal-like case statement).

```
(define (CartesianPoint x y)
  (define (sum p1 p2)
      (CartesianPoint (+ x (p2 'x)) (+ y (p2 'y))))
  (lambda msg
    (case (car msg)
       ('(sum) (sum (cadr msg) (caddr msg)))
       ('(x) x)
       ('(y) y))))

(define p1 (CartesianPoint 1 2))
(define p2 (CartesianPoint 3 2))
(p1 'sum p2)
```

Objects can have encapsulated variables and methods. In the example encapsulation is achieved by making use of the specific scope rules of the underlying language.

Objects with an explicit method dispatcher don't have an explicit interface. Since the method dispatcher is algorithmically defined it is not possible, in general, to determine an object's interface on inspection of its definition.

Associated with explicit method dispatchers is the phenomenon of first-class message names (although this is not always the case). In the above example it is apparent that message names are first-class values, they can be passed as arguments for instance. In the example this is essential for the method dispatcher to work since the dispatcher needs to compare message names.

Restrictions can be put on the form of the method dispatchers (such as in the above example where the dispatcher is essentially a case statement) and message expressions. All such restrictions will have the goal of making the accepted message interface more explicit.

It is trivial to notice that for example object-oriented type checking[1] of this sort of objects is impossible. It is, in general, not possible to determine whether a message sent to an object will result in an error or not. In fact it is in general not possible to determine what message is sent to an object. Neither is it possible to determine whether two objects are substitutable.

[1] It is possible to type check objects with explicit dispatchers, but type checking will reduce to checking e.g. function domains of the functions that represent the objects, which is obviously not what is intended in this case.

Lambda calculus with records
Lambda calculus extended with a record datatype can be used to model certain features of object-oriented programming. Objects are simply modelled as records. Once again the particular scope rules of lambda calculus can be used to achieve encapsulation. Instance variables are modelled as record fields containing a value, or directly as variables in lambda calculus. Methods are modelled as record fields containing a lambda function. In the example we use the notation `{x1.e1 ; … ; xn.en}` for a record with field labels `x1` to `xn` and values `e1` to `en`.

```
(define (CartesianPoint x y)
  {x.x ;
   y.y ;
   sum.(lambda (p2) (CartesianPoint (+ x p2.x) (+ y p2.y)))})

(define p1 (CartesianPoint 1 2))
(define p2 (CartesianPoint 3 2))

(p1.sum p2)
```

Message passing, however, is not an atomic operation. The fact whether an attribute of some object is implemented as a stored value versus a computed value can not be hidden. Whereas a message retrieving a stored value is mere record selection, a message that invokes a method must be explicitly constructed as a combination of record selection and function application.

The fact that this construction must be made by the sender of the message also compromises object encapsulation in a way. It is possible to retrieve a method without invoking it. This method can subsequently be stored for later invocation gaining direct access to an object's encapsulated parts without passing through its interface. It is not so much the fact that this is possible but rather that the means provided to define methods can not preclude this.

Modules and Abstract Datatypes
OOPL share their concern about interfaces and encapsulation with abstract datatypes (ADT). The essential difference is that encapsulation in ADTs is a result of the type system employed. The fact that the implementation of an ADT can only be accessed through its procedural interface is enforced by the type system. This leads to a different sort of encapsulation (i.e. module-based encapsulation).

The essential difference with ADT encapsulation is that a function in an ADT's interface can access the implementations of all the ADT's elements it has knowledge of, e.g. a function that is part of an ADT, that has two arguments that are both typed as elements of the ADT, has direct access to the implementation details of both arguments (see the sum operation in the point example below). This leads us to say that with ADTs the implementation details are mutually visible to all members of the ADT. In contrast a method of some object has only access to the encapsulated part of ONE object: the receiver.

Even if a method has indirectly obtained a reference to the receiver object it can not access the receiver's implementation details via this reference. To put it otherwise: an object has no direct access to the implementation details of any of its acquaintances, even if it has itself as acquaintance. With ADTs a function activation of a function in the ADT has direct access to the implementation details of all the elements of the ADT that are visible in that activation.

```
Module CartesianPointModule
  Interface
    Type CartesianPoint

    Function sum(p1, p2) Result CartesianPoint

  Implementation
    Type
      CartesianPoint = Record x:Real y:Real End

    Function sum(p1, p2) Result CartesianPoint
      c:CartesianPoint
      Begin
        c.x := p1.x + p2.x
        c.y := p1.y + p2.y
        ^c
      End
End
```

Traditionally, ADTs also lack the late-binding polymorphism that is associated with objects. Polymorphism can be added orthogonally to ADTs.

The essential difference between object-based encapsulation and ADT encapsulation is in all aspects similar to the two complementary encapsulation mechanisms of procedural abstraction and type abstraction as identified by Reynolds in [Reynolds75]. In a similar vein the differences between objects and ADTs have been amply discussed in [Cook90]. Although the terminology used (Cook discusses the difference between what he calls abstract datatypes and procedural data abstraction) is different, and we do not share his opinion that the essence of object-orientation is procedural data abstraction, we agree with his argumentation on how ADTs support a different notion of encapsulation and are restricted in their overloading capabilities.

3.2.3 Operations on Objects

In practical object-oriented programming languages objects are defined in different ways. In class-based languages all objects are instantiated from a set of templates — called classes — in a cookie cutter way [Stein,Lieberman&Ungar89]. In prototype-based languages idiosyncratic objects or new objects can be made by copying old objects.

Furthermore different mechanisms exist to construct new objects out of (a combination of) old objects. In general this takes the form of inheritance, which is an incremental modification mechanism on classes, or delegation [Lieberman86], which is a control structure similar to message passing, or a more modular mechanism where two classes or objects are combined with a primitive operator to form new objects [Bracha92][Bracha&Lindstrom92].

The emphasis of this and the next four sections is, on the one hand, on how new objects can be constructed so that interfaces are derivable, and on the other hand, on how we can derive *objects* from already existing *objects* (with the emphasis on *objects*). Unrestricted unanticipated delegation is unacceptable in this context. In the section on prototypes it is shown that unrestricted unanticipated delegation is in contrast with the constraint of explicit interfaces. Pure class-based approaches are unacceptable in their need to have an explicit class construct in order to derive new objects. Class-based inheritance is not an operation on objects.

The purpose is to come up with an acceptable form of object-based inheritance, whereby object-based inheritance is an inheritance mechanism for objects (such as delegation) in which interfaces are derivable. We will first show how classes can be represented as generator functions. Exactly these generator functions will form the basis of a novel mechanism of object-based inheritance. This mechanism is based on mixin-methods. Note that classes will not be discarded entirely, in the section on prototypes versus classes we will show how classes can be reintroduced in an OOPL that employs object-based inheritance.

The second goal is to show that at least one other operator exists to derive new objects. We will show an operator that derives a new object by encapsulating one object in another. It will be shown that essentially two forms of this encapsulation operator exist. Furthermore an example will be given to show that one of both forms is a possible candidate as a basis for introducing user defined control structures in an OOPL in replacement of the often used higher order functions.

In spite of the considerations that were made in the previous section, objects will be modelled as records. Much of what follows in this and the next four sections is derived from work in denotational semantics of object-oriented programming languages [Cardelli88] [Cook89] [Bracha92] [Hense92] [Ghelli90] [Kamin88] in which records are almost unanimously used to model objects. It is our opinion that the argumentation is, to a sufficient extent, independent of the employed object model.

3.2.4 Classes and Class-based Inheritance

Design Issues in class-based languages
In class-based languages objects are grouped into classes. The class concept is heavily overworked. Classes also play many different roles in different OOPL.

A class is usually defined as a *template* that contains both instance variable definitions and method definitions. This template can be *instantiated* to create new objects, called *instances* of the class or template, that conform to this template. An object conforming to some template has exactly the number and names of instance variables as defined in the template and responds to messages according to the appropriate method definitions from the template. The exact value of the instance variables can vary from object to object.

Class-based inheritance is usually characterised by its *strict use of templates*. That is to say, objects instantiated from some template will stay conform to this template during their entire lifetime. This means, for example, that an instance of some class can not be extended by adding extra methods.

Another characteristic property of class-based languages is the *strict separation of templates and instances*. Concretely this means that the operations allowed on templates are restricted to instantiation and on the other hand instances can not be used as templates. Furthermore, class-based languages require in general that all objects are instance of some class, i.e. no other means are provided to create objects than by class instantiation.

Classes can be organised into class hierarchies. We will see that *class-based inheritance strictly involves incremental modifications of templates*. This view is supported by the fact that objects in a class-based language have *strong identity* (as discussed in the section on object identity).

Classes as a classification mechanism
Objects are classified by classes; classes are classified by class taxonomies. Keeping this in mind, an object belongs to more than one class: the class from which it is instantiated and all the superclasses from that class (or more formal all the classes into which this class is classified). We will call the class from which an object is instantiated the object's class. In traditional class-based languages an object has exactly one class and an object can not be reclassified dynamically. Languages exist, though, where an object can have more than one class [Hamer92] and where objects can be reclassified dynamically. We will have a closer look at the latter case.

Two different sorts of *dynamic reclassification* exist depending on the 'feasibility' of the reclassification. An object is closely linked to its class since a class describes the template ('layout') of all the objects that are instantiated from it (and in fact dynamic reclassification is more or less a violation of the 'strict' use of templates that is typical for class-based languages).

An object can not easily be reclassified to a class that defines a totally different template for its instances. Taking this into consideration, the 'feasible ' reclassifications of an object are those where an object is reclassified to a superclass of the object's class or one of the possible subclasses of the object's class. The 'unfeasible' reclassifications of an object are those where an object is reclassified to a class that is unrelated to the object's class. Although the latter sort of reclassification has been studied in the context of for example object-oriented databases the former sort of reclassification seems more useful as a concept in programming languages.

Reclassification is called monotonic if an object is reclassified to one of the possible subclasses of its class. *Monotonic reclassification* is interesting since it allows an object to gain attributes during its lifetime.

Classes as a module mechanism
Classes are in some cases used to introduce an extra form of encapsulation. Existing languages mostly employ an object-based encapsulation, i.e. the per object encapsulation of private attributes that is inherent to object-based programming. Classes can be used to introduce a sort of *module-based encapsulation* [Ungar,Chambers,Chang&Hölzle91]. The idea is to let all objects belonging to the same class have privileged access to each other. This is reminiscent of, the already discussed, abstract datatype encapsulation.

In its ideal form this sort of module-based encapsulation should be an extra form of encapsulation on top of the already existing object-based encapsulation, this is not always the case however (e.g. C++ exclusively uses module-based encapsulation). Module-based encapsulation is realised by declaring in each class which of the attributes are visible only for the objects belonging to the class or vice versa, which of the attributes are visible for all objects not belonging to the class. In its ideal form this extra restriction on the visibility of attributes only applies to those attributes that are not made invisible by the object-based encapsulation.

Object-based encapsulation alone is in some cases too restrictive. Examples where module-based encapsulation is desirable overlap largely the examples where multi-methods are desirable, e.g. arithmetic operations. Module-based inheritance is indeed useful, but it has been rightfully argued that classes and modules are separate concepts [Szyperski92], i.e. that this sort of encapsulation should not be strictly coupled to classes but rather that modules should be provided as an explicit language construction.

Similar observations to the above can be made concerning the use of 'shared' or global variables. Most class-based languages associate with a class a set of variables that are shared by, and directly visible to, all instances of that class. These so called *class-variables* play the role of global variables and can be used to share information among instances of one and the same class. Apart from these class-variables other kinds of shared variables are provided (in Smalltalk for example there are so called global variables, pool variables and class variables). Here again one can wonder whether the use of shared variables should be strictly related to classes and if not whether it should perhaps be better to provide a separate mechanism to share variables.

The use of *nested classes* is related to both of the above issues. Depending on the language nested classes are used to implement shared variables or to provide a restricted form of a module mechanism. This will be elaborated upon in the section on scoping.

Meta-Classes
Classes themselves can be considered as first-class objects. The advantages are that classes can be manipulated as any other object and that instantiation can be realised by message passing. An object is created by sending an instantiation message to the class of which an instance is required. This has several advantages. First, no special operation is required for instantiation; secondly a class can now provide different instantiation methods. The task of an instantiation method is to create a new instance and to do the necessary initialisation.

In a pure class-based language (one in which every object must have a class) it is impossible to consider a class as a first class object without then considering the meta-class of this class (i.e. the class of which an object's class is an instance) and its meta-class and its meta-class and so on … .

It is apparent that the *class concept is heavily overloaded*. This is also apparent in the relatively large number of proposals to unravel the different functionalities covered by the class concept. Witness of this are proposals to differentiate classes from types (interfaces), classes from modules and to provide sharing mechanisms that are independent from the class construct.

Class-Based Inheritance as an Incremental Modification Mechanism
The idea of class-based inheritance is that a new class is defined by specifying how it differs from an already existing class. This latter process is also referred to as *incremental modification* [Wegner&Zdonik88]. Incremental modification is considered to be one of the most innovative concepts of OOPL. The goal of it is to realise small system changes by small specification changes, or, to make extensions or modifications to a system's behaviour without modifying existing code but rather by adding new code.

Viewed as an incremental modification mechanism inheritance takes the form of a parent P (the superclass) that is transformed with a modifier M to form a result R = P + M (the subclass); the result R can subsequently be used as a parent for further incremental modification.

In our model of objects the parent, result and modifier are collections of named attributes. From the viewpoint of the result R, the attributes defined in the parent P are referred to as the *inherited attributes*, attributes defined in the modifier M are referred to as the *proper attributes* of R.

The result R is truly an extension of the parent P (in contrast with e.g. aggregation). Access to the inherited attributes in R is exactly the same as access to the proper attributes of R, though the proper attributes of R take precedence over the inherited attributes in case of name clashes.

The above incremental modification model of inheritance is a simplification. In most common object-oriented languages, modifiers themselves, also, have access to the attributes of the parent with which they are composed. For example, a subclass can invoke operations defined in the superclass (hereafter called parent operations). To model this, a modifier M is parameterised by a parent P that can be referred to in the definitions of the attributes of M. The actual parent is supplied to a modifier when a modifier is composed with a parent. Composing a parent P and a modifier M now takes the form P Δ M = P + M(P), where the modifier M is no longer a simple set of attributes, but is now a function from a parent to a set of attributes.

Modifiers can be made concrete in lambda calculus. In the following example we assume that collections of attributes are represented as records. We assume that records can be added by means of the "+" operator. As an example a modifier is given that extends a point object with a `sumtwice` method. The point object is assumed to understand a sum message.

```
PointModifier =λsuper.{sumtwice.λp2.super.sum(p2.sum(p2))}
ExtendedPoint = Point Δ PointModifier
```

It is obvious from the previous that inheritance as modelled above is an asymmetric operation. Modifiers are different from objects. An object is combined with a modifier to form a new object. It is not possible to combine two objects directly. It is possible, however, to define an operator that combines two existing modifiers into a new modifier:

```
M1 & M2 = λP. (P Δ M1) Δ M2
```

Further argumentation for the asymmetric treatment of inheritance will be given in the section on multiple inheritance.

Method Overriding, Late Binding of 'self'

The above model of inheritance is still a simplification. No treatment is given of recursion or self-references. The expressiveness and the modelling power that is ascribed to inheritance owes much to the special treatment of recursion in OOPL. Recursion emerges when, in response to some message, the invoked method refers to the receiver object. For this purpose a so called *'self' pseudo-variable* is provided. The self pseudo-variable contains a reference to the receiver object. It can be used in the implementation of an object to define recursive methods, for example.

Recursive method invocations take a special form when the base class that invokes one of its methods recursively is modified with a modifier that overrides the method under consideration. In this case the recursive call will result in an invocation of the *overriding method* definition. This is illustrated in the following picture.

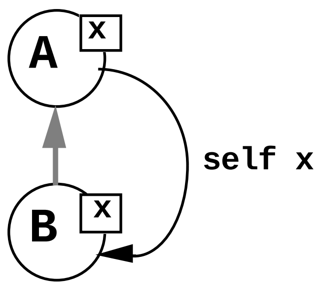

Figure 3.3

The special treatment of recursive invocations in the inherited attributes is referred to as *late binding of self*. The fact that inherited attributes from a parent are more essentially part of inheritors than attributes that are merely invoked can be explained entirely by the late binding of self in recursive invocations in inherited attributes.

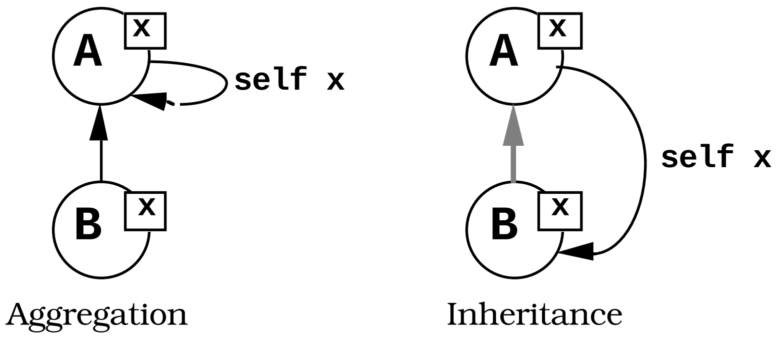

Figure 3.4

Late binding of self is the source of much of the power of inheritance. It is also part of the good programming practice that is associated with inheritance. It allows code to be factored out in a common parent. The parent can rely on its possible inheritors to 'fill in' the unspecified details. A parent that invokes methods that are to be defined in inheritors is called an *abstract parent* [Bracha&Lindstrom92]. Abstract parents are an essential part in structuring inheritance hierarchies and creating *object-oriented frameworks*. We refer to the section on that topic.

Generator Functions and Classes

The most widely accepted definition of inheritance is given in terms of its operational semantics. The operational semantics describes the way a method is looked up in the inheritance chain of a receiver object [Goldberg&Robson89]. This process is referred to as *'method lookup' semantics*. A first denotational semantics in which the importance of recursion is recognised, was given in [Reddy88]. A denotational semantics in terms of fixed points is given in [Cook&Palsberg89][Cook89] and has been used by several other authors [Bracha92] [Hense92].

This analysis is based on the fact that recursive (function) definitions, in a mathematical treatment, can be expressed as fixed points. For example the factorial function can be expressed (in lambda notation) as:

```
fac = FAC(fac)
where FAC = λf.λx.if x=0 then 1 else x*f(x-1)
```

The function FAC of which the factorial function is the fixed point is called the generator function. The self-reference in the factorial function is explicitly captured by the first argument to this generator function (i.e. ' f' in the above). The fixed point of a *generator function* is constructed by applying a fixed point operator to it. The standard fixed point operator in lambda calculus is the Church or Y fixed point combinator [Revesz88], i.e.:

```
fac = Y(FAC) where Y(f)= f(Y(f))
```

An example of how a point object can be modelled using fixed points is given below. The point object is self referential in its `distFromOrigin` method. The `distFromOrigin` method needs access to the point object itself. This self-reference is resolved by constructing the resulting point object as the fixed point of a generator function that expects a point object as argument. This generator function generates objects in which this self-reference is resolved. The idea is of course that the argument of the generator function is bound to the object that is being generated: a point object is the fixed point of the generator function. The `INST` function, in the example, is the same as the Church or Y fixed point combinator.

```
pgen(a,b) =
  λs. { x.a,
        y.b,
        distFromOrigin.√(s.x² + s.y²),
        closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)
      }

p = INST(pgen(1,2))
p.distFromOrig

INST(x) = x(INST(x))                    -- INST(x) = Y(x)
```

Another way to look at generator functions is that they are objects in which the self is not yet bound. In that respect they are comparable with classes. Taking the fixed point of a generator function is the equivalent of instantiation. Hence the name of the fixed point combinator in the example.

Inheritance can be modelled as an operator on generator functions. The fact that in a generator function, the self is not yet bound provides exactly the necessary functionality to model the late binding of self. This is illustrated in the following example. Manhattan points are constructed by overriding the `distFromOrigin` method from point. Generators can be combined to form new generators with the `INHERIT` combinator. The result of the `INHERIT` combinator distributes its self argument (the result is a generator) over its two constituent generators. The result of applying the two constituent generators are two objects that can be combined with mere record combination. Instantiation of a Manhattan point is done by taking the fixed point of the combined generator.

```
mpgen = λs.{distFromOrigin.(s.x + s.y)}

mppgen(a,b) = INHERIT(pgen(a,b), mpgen)

mpp = INST(mppgen(3,4))

INHERIT(G1,G2) = λs.G1(s)+G2(s)
```

The techniques found in this section and the ones found in the previous section can be combined. This gives a full account of inheritance as found in class-based languages.

3.2.5 Classless Delegation

In *prototype-based languages* (also referred to as *delegation-based languages*) objects are not organised into classes. The basic mechanisms provided in prototype-based languages are object creation and delegation, apart from message passing, of course. This results in a much simpler view of object-oriented programming. Still some design issues remain to be discussed. The discussion below follows to a certain degree the discussion of prototype-based languages in [Dony,Malenfant&Cointe92].

Object Creation
In the absence of classes, objects must be created by another means than template instantiation. Two alternatives exist. One is that objects are created ex nihilo; the other alternative is that objects are created by *cloning* (copying) an already existing object.

Objects that are created *ex nihilo* can be either created as empty objects that must be 'filled up' afterwards, or an object is created by listing its public and private attributes (both methods and instance variables) and the initial values for instance variables. For similar reasons as in [Dony,Malenfant&Cointe92] the former option is ruled out. Creation of empty objects presumes the existence of primitives to dynamically change the structure of an object, i.e. to add instance variables and methods. This is not only a dangerous feature but is also in contradiction with the fact that an object's interface should be explicit in its definition.

The second alternative to make new objects is by cloning existing objects. It is apparent that this is a less primitive way to create objects. With a cloning primitive alone no objects with a new structure can be constructed. In fact cloning closely resembles instantiation in class-based languages. Cloning can be interpreted as taking an existing object as template to create a new object. Additionally to the creation of a new instance, which is the purpose of instantiation, this new object has an initial state that is a copy of the state of the object used as template. Different sorts of cloning exist according to the amount of control a programmer has over the copying process with respect to which part of the state of an object has to be copied (e.g. deep copying versus shallow copying).

To sum up: whereas in class-based languages objects with an idiosyncratic structure are created by first defining a template (ex nihilo) and then instantiating this template. And whereas in class-based languages objects of the same kind are created by taking different instances of one and the same template. We can say that in prototype-based languages, on the contrary, idiosyncratic objects can be created directly (ex nihilo) and objects of a same kind are created by cloning an existing object with the extra advantage that state, also, is copied into the newly created object.

Delegation
The second characteristic mechanism for prototype-based languages is delegation. Different forms of delegation (according to the flexibility and anticipation of the delegation structure) exist. After discussing delegation in its purest form, i.e. explicit unanticipated delegation, we will review some variants of delegation.

Delegation allows the behaviour of an object to be defined in terms of the behaviour of another object. In its most general form delegation was introduced [Lieberman86] as a message forwarding mechanism. A message that can not be handled directly by the receiving object is forwarded (or delegated) to another object that responds on behalf of the delegating object.

Delegation is a special form of message passing. A message is delegated to an object. What differentiates delegation from ordinary message passing is the interpretation of recursive method invocations (i.e. the interpretation of the self pseudo variable). As illustrated, the difference between message passing and delegation is similar to the difference between aggregation and inheritance. The difference between inheritance and delegation is that the inheritance structure of an object is fixed, whereas the delegation structure can vary.

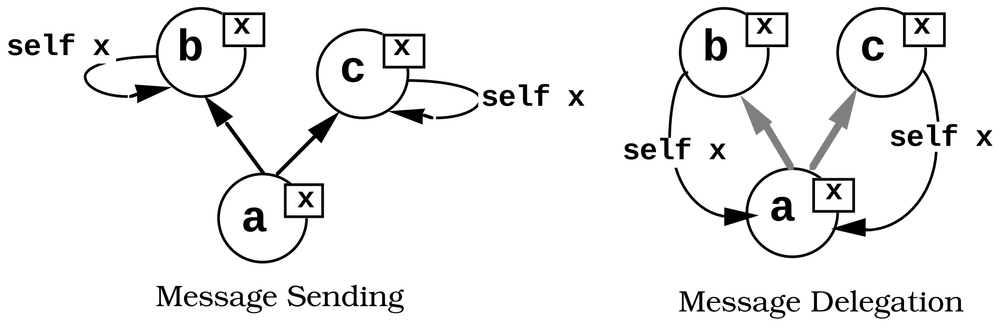

Figure 3.5

In the above model of class-based inheritance objects were modelled as fixed points of generators. An object' s self was bound at instantiation time. In a model of classless delegation all objects must have an unbound self. At any point in time a message can be delegated to an object forcing all self-references in that object to be redirected to the delegating object. An example can be found below.

```
makePoint(a,b) =
  { x.a, y.b,
    distFromOrigin.λself.√(self.x^2 + self.y^2),
    closerToOrigin.
      λself.λa.(self.distFromOrigin(self) < a.distFromOrigin(self)) }

makeManhattan(p)
  = p + { distFromOrigin.λself.self.x + self.y) }

p1 = makePoint(1,2)
mp1 = makeManhattan(p1)
p2 = makePoint(2,2)
mp2 = makeManhattan(p2)

p1.closerToOrigin(p1)(p2)            --- true
mp1.closerToOrigin(mp1)(mp2)         --- true
mp1.closerToOrigin(mp1)(p1)          --- false
```

In this encoding of objects all methods have access to the receiver as an explicit argument. Each time a message is sent the receiver must be passed along explicitly as message argument. This allows considerable freedom. Delegation is characterised by the fact that the receiver that is passed as argument is not the same as the receiver of the delegated message. Consider the following example where ' mp1' delegates a message to ' p2' . ' p2' is the explicit receiver of the delegated message, but all self-references are directed to ' mp1' .

```
p2.distFromOrigin(mp1)               --- √3
```

Variants of Delegation
A first design alternative to be considered is whether delegation must be explicit or implicit. With *explicit delegation* an object can explicitly delegate a message (an operation that syntactically resembles message passing but has a different meaning in its late binding of self) to any other object it has knowledge of (e.g. instance variables). Since an object has to decide on a message per message basis whether this message has to be delegated, explicit delegation is only relevant for objects with an explicit method dispatcher. So, although explicit delegation provides great flexibility it is in contradiction with the notion of explicit interfaces. Therefore implicit delegation is favoured.

With *implicit delegation* an object can explicitly designate another object as its parent. The parent is used to delegate messages to that are not found in the interface of the receiving object. Once again two design choices can be made. One in which the delegation structure can be dynamically changed, the other in which an object can not change parent. In the latter case the parent object must be assigned when the object is created and can not be reassigned during the object' s life-time. We adopt the terminology from [Stein,Lieberman&Ungar89]: the former is called *unanticipated delegation*, the latter is called *anticipated delegation*.

In prototype-based languages where the delegation structure can be dynamically changed the parent of an object is typically stored in some specially identified instance variable (e.g. Self [Ungar&Smith87]). This instance variable can be consulted and also modified, thus changing an objects parent. Dynamically changing an object' s parent is flexible (useful examples are given in [Ungar,Chambers,Chang&Hölzle91]), but again in contradiction with the notion of explicit interfaces.

Implicit anticipated delegation can also be interpreted as an incremental modification mechanism. Here again a parent (the parent object) is incrementally transformed with a modifier M (the definitions in the delegating object) to form a result R = P + M (the resulting object). This insight underlies the argument in [Stein87] in which the author argues that delegation is inheritance. Oversimplifying the main result one could say that the argument boils down to the fact that both delegation and inheritance are both in essence based on an incremental modification mechanism. Whereas in class-based languages inheritance involves incremental modification of stateless templates, implicit anticipated delegation can be considered the same as incremental modification of objects with state.

In the remainder we will call the form of implicit delegation where the delegation structure is anticipated *object-based inheritance* (versus class-based inheritance). This name stems from the fact that this sort of delegation is very similar to inheritance in class-based languages. The same term has been used in [Canning,Cook,Hill&Olthoff89] for a similar notion. Furthermore we will call an object that is being extended the *prototype* (the superclass in class-based terminology) and the extended object will be called the derived object (the subclass in class-based terminology).

3.2.6 Mixin-Method Based Inheritance

The difference between class-based inheritance and delegation can be recast in terms of to what degree the self of an object is encapsulated in that object. With class-based inheritance all objects have a totally *encapsulated self* — an instantiating client of an object can not reassign the self of that object. With delegation-based inheritance all objects have an *unencapsulated self* — all instantiating clients of an object can reassign the self of an object. The dilemma of object-based inheritance is that normal message passing requires objects that have an encapsulated self, and inheritance requires objects that have a nonencapsulated self.

Moreover, generator functions can be seen as a motivation for class-based languages. Since the late binding of self that is essential for inheritance is so intimately connected with generator functions, it could be argued that generator functions and consequently classes are an essential ingredient of object-oriented programming. Generator functions must be first class values to model class-based inheritance with the above approach.

In this section we will show that an alternative approach to inheritance can be devised that combines the advantages of class-based inheritance with the advantages of delegation and avoids the above pitfalls. The flavour of delegation that is so obtained corresponds to the already mentioned object-based inheritance. On the semantic level a new technique is proposed, based on *encapsulated generator functions*, to explain this new kind of inheritance.

Generator functions can be thought of as objects with an unassigned self. The essence of our solution to the above dilemma is that an object encapsulates a nonencapsulated version of itself and that an object must be asked to extend itself; and only an object can extend itself.

An object is, in this approach, a wrapper around its own generator function. This wrapper fixes and encapsulates the self for the wrapped generator function. Since an object has an explicit reference to its own generator function it can extend itself. An instantiating client of this object however has a view of the object in which the self is encapsulated. This is illustrated in the following figure.

```
mpgen = λg.λs.{distFromOrigin.(s.x + s.y)}
pgen(a,b) = λg. λs.
    { x.a,
      y.b,
      distFromOrigin.√(s.x² + s.y²),
      closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)
      mp.WRAP(EXTEND(g,mpgen))
    }

p = WRAP(pgen(1,1))
p.distFromOrigin                        -- = √2
mp = p.mp
mp.distFromOrigin                       -- = 2

WRAP(x) = INST(x(x))
EXTEND(G1,G2) = λg.INHERIT(G1(g), G2(g))
```

Generator functions now have two arguments. One argument is used, as before, to contain a reference to the object itself. The other argument contains a reference to the generator itself. Instantiation takes a slightly different form. An object must be provided with its generator function when it is instantiated. This is done in the new instantiation function WRAP. Notice also how generator functions are combined.

Objects can only be extended by selecting an appropriate attribute. In the example the point object is extended to a manhattan point by selecting the `mp` attribute. In anticipation of a thorough discussion on *mixin-based inheritance*, methods (attributes) that return an extension of their receiver object will be called *mixin-methods* (*mixin-attributes*).

The result is a different notion of prototype-based programming. In pure delegation-based languages any object can delegate to any other object it chooses. Object-based inheritance with encapsulated generators still has the notion of extending objects rather than classes. It also conforms to what is generally considered important for prototype-based languages in its nonstrict use of templates. However, it is not possible for an object to delegate messages to any other object it chooses. Each object has control over how it will be extended.

Delegation is more powerful — and, as was shown, too powerful — than object-based inheritance with encapsulated generator functions. Indeed the latter one can be mimicked with the former. It suffices to define a method for each corresponding mixin method that extends the receiver object by delegating to it. Object-based inheritance with encapsulated generator functions is in its turn more powerful than class-based inheritance in its ability to extend objects.

The above new inheritance mechanism puts inheritance in a different perspective. In class-based languages as well as in delegation-based languages, inheritance is realised by explicit operators to combine classes (respectively objects), making inheritance a fundamental operation. In contrast, inheritance as in the above proposal, is an operation that is internal to objects. Witness of this is that all generator functions can be encapsulated in objects.

The above model of inheritance is the basis of mixin-methods in *Agora*. We will not pursue the above line of reasoning in a formal way (the usage of encapsulated generators in a denotational semantics of mixin-methods is currently under investigation). The practical use of mixin-methods will be explored in the section on multiple inheritance.

Can classes and prototypes coexist ?
In the previous sections we showed that both classes and prototypes contribute to object-oriented languages. Still, we argued that each, in combination with respectively inheritance and delegation, has unacceptable drawbacks. An intermediate form — i.e. prototypes with mixin-based inheritance — was opted for. We now turn to the question whether we can reintroduce some of the advantages of classes into a possible OOPL featuring prototypes with mixin-based inheritance.

As argued in other work [Stein,Lieberman&Ungar89] the difference between classes and prototypes is a difference in flexibility. On the one hand classes and class hierarchies are an important structuring mechanism, on the other hand it is sometimes important to have objects with an idiosyncratic behaviour or to be able to temporarily extend an object's behaviour.

There are two notable attempts to integrate classes and prototypes. One attempt is the integration of class-like behaviour in a prototype-based language through the use of 'traits' objects in Self [Ungar,Chambers,Chang&Hölzle91]. The other is the introduction of objects with an idiosyncratic behaviour in a class-based language in a hybrid language [Stein,Lieberman&Ungar89]. Traits objects in Self have been shown [Dony,Malenfant&Cointe92] not to provide the right concept for this task. The hybrid approach starts from classes as the key concept and handles prototypes as a 'special case' taking away much of the simplicity that is typical for prototype-based languages. It is our opinion that an integrated approach should start from prototypes and consider classes as an extra structuring mechanism on top of the already existing prototypes.

As was shown in the two previous sections the underlying principles of class-based languages are (ex nihilo created) templates, template instantiation and incremental modification of templates; the underlying principles of prototype-based languages are (ex nihilo created) objects, object cloning and incremental modification of objects.

The difference between, and consequently the integration of, classes and prototypes can now best be illustrated by seeing templates as a special case of objects. This view is supported by [Stein,Lieberman&Ungar89], where respectively templates and empathy are considered the fundamental principles underlying both inheritance and delegation. The main difference between classes and prototypes is how strictly templates are used and how strict the separation of templates and instances is.

To introduce classes in a language with only objects we must separate out special objects that will act as templates. In order to be able to regard these special objects as true templates some restrictions must be put on the use of these objects. We will call these special objects *template objects*, all other objects will be called *instance objects*.

First, a strict distinction must be made between those messages that are sent to create a new instance, called *instantiation messages*, and all other messages, called *'ordinary' messages*. In a class-based language the former ones are the instantiation messages defined for classes, in a prototype-based language these are the cloning messages defined on all kinds of objects. So, template objects should be restricted in such a way that they only respond to cloning messages, and vice versa, a strict use of templates implies that all instance objects should be restricted in such a way that instance objects don't respond to cloning messages. Note that since instance objects are cloned versions of some template objects, the above restriction truly is a restriction on the interfaces of both the template object and the derived instance objects. Both objects include the entire protocol (i.e. cloning and all other messages). Declaring an object as a class, however, implies that the protocol of this object is restricted to cloning messages. Making an object as instance of some class implies that the protocol of this object is restricted to all but the cloning messages.

Secondly, a restriction must be put on the incremental modification of instance objects. The incremental modification of template objects corresponds exactly to inheritance. A strict use of templates implies that instance objects can not be incrementally modified. In general it is difficult to enforce such a restriction. However, in the special case of mixin-method based inheritance where an object must be asked to extend itself, this restriction is, again, a restriction on the interface of objects.

Summing up: in order to introduce classes in a prototype-based language some objects must be restricted in the way they are used; we introduce two kinds of these restricted objects, i.e. template objects, that will play the role of classes (and can be called classes from now on!), and instance objects, that will play the role of instances of classes. Template objects can only be cloned (instantiated) and incrementally modified, instance objects can not be cloned nor incrementally modified.

Some important remarks should be made here. Not all objects must be either template objects or instance objects, i.e. it is still possible to have objects with an idiosyncratic behaviour that are not classes. That is to say, classes and prototypes can happily live together. Moreover, we can loosen the restrictions put on instance objects. In most class-based languages it is possible to copy instances (a closely related operation to cloning), it is in most cases not possible to incrementally modify instances. If we allow instance objects to be incrementally modified, then template objects will serve as *minimal templates* [Stein,Lieberman&Ungar89] rather than strict templates. Note that incrementally modifying an instance object is closely related to monotonic reclassification in pure class-based languages.

Finally note that, even when ignoring the above remarks, this approach does not lead to what has been called, in the section on classes, a pure class-based language. Not all objects are instances of a class: classes are not. Classes are objects with an idiosyncratic behaviour and class inheritance is expressed as incremental modification of objects. This is much in the spirit of [Stein87]. We find this a very satisfactory situation. Although meta-classes have some use, meta-classes are conceptually too complex. A symptom of this is the distinction between class-methods and instance-methods in pure class-based languages with meta-classes. The concept of class-methods (and also class instance variables) is mostly considered as concept that is 'hard' to understand. By modelling classes as objects with an idiosyncratic behaviour and instantiation as object cloning, this distinction between class-methods and instance methods, is nothing but a restriction on the protocols of the template object (the class) and the instance object (its instances).

3.2.7 Encapsulation as an Explicit Operation on Objects and Generators

Up until now nothing much was said about the form of the encapsulated attributes. In all of the above discussions encapsulation was a consequence of the nested scoping of the underlying lambda calculus and encapsulated variables were encoded as variables in the underlying calculus. Rather than being just a collection of unrelated attributes, the encapsulated attributes can be grouped into an object. Encapsulation then becomes an operation on objects where one object is encapsulated in another object, or alternatively an operation on generator functions. Although unconventional, encapsulation operators in this form are helpful in modelling private methods or nested objects for example. The combination with inheritance gives rise to a variety of different possibilities. This has much to do with the question of encapsulated versus nonencapsulated inheritance as will be discussed in the next chapter. We will not try to be complete in covering all possible forms of encapsulation operators. We will just point out some possibilities.

In its simplest form an encapsulated part is an object, but is defined locally to the generator function. This can be a good way to structure the encapsulated part of an object. For example the encapsulated part can inherit independently from the object of which it is part.

```
pgen =
  λs. let e = INST(λs.{x.0, y.1})
   in {distFromOrigin.√(e.x² + e.y²)
       closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)}

p = INST(pgen)
```

The above form of encapsulating attributes fails in modelling private methods as can be found in current day OOPL. What is typical for private methods is that they inherit the ' self' from the object of which they are part, in contrast with the above example where the encapsulated part is an object that has its own self reference. The difference between the two is that in the latter case self-references in a private method are directed to the encapsulated part only, whereas in the former they are directed to the entire object. This can be solved with an encapsulation operator that works on generator functions. The idea is that the generator function of the encapsulated part must be used rather than its instantiated form. The self argument of this generator function can then be bound to whatever is necessary (e.g. it is bound to the entire object when modelling true private methods).

Below is an example of the usage of such an encapsulation operator on generator functions (ENCAPSG). This encapsulation operator encapsulates one object into another such that they share the same self-reference. In this example it is used to model a colour point that inherits from a point class in an encapsulated way. The x and y encapsulated point attributes are not visible to the methods defined in the colour point (encapsulated inheritance will be discussed later on in the text).

```
cpgen = λe.λs.{color.e.color}
pgen = λe.λs.{ distFromOrigin.√(e.x$^2$ + e.y$^2$),
               closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)}

mppgen = λe1.λe2.INHERIT(ENCAPSG(e1,pgen),ENCAPSG(e2,mpgen))

p = INST(ENCAPSG(λs.{x.0,y.1},pgen))
mpp = INST(mppgen(λs.{x.3,y.4})(λs.{d.5}))

INST(x) =  Y(x)
ENCAPSG(e,x) =  λs.x(e(s))(s)
INHERIT(G1,G2) = λs.G1(s)+G2(s)
```

The encapsulation operator that works on generator functions serves the purpose of structuring the encapsulated part of an object mainly with respect to the inheritance hierarchy. We will come back on this later when discussing mixin-methods and their implementation.

Alternatively an encapsulation operator on objects (after instantiation) can be devised. Consider the following example. After instantiation a point object still expects an encapsulated part. In the example the expected encapsulated part of p is bound to a simple object containing x and y variables.

```
pgen =λs.λe.{ distFromOrigin.√(e.x$^2$ + e.y$^2$),
               closerToOrigin.λp.(s.distFromOrigin < p.distFromOrigin)}
egen = λs.{x.0,y.s.x+1}

p = ENCAPS(INST(egen), INST(pgen))

INST(x) =  Y(x)
ENCAPS(e,x) = x(e)
```

It should be noticed that the ENCAPS operator is one that encapsulates attributes into an object after its instantiation. It is an example of an operator that enables an object to gain acquaintances after instantiation. The most important advantage of an explicit operator for encapsulating one object in another, is that it allows to define control structures in a pure message passing style, thus giving an interesting alternative to higher order functions as a basis for user defined control structures.

Consider the following example. The boolean objects true and false are defined. A conditional expression is constructed by encapsulating an object with two methods that correspond to the branches of the conditional into either the true or false object.

```
truegen =λs.λe.{if.e.true}
falsegen =λs.λe.{if.e.false}
true = INST(truegen)
false = INST(falsegen)

 … λb.λx.λy.ENCAPS({true.x, false.y}, b).if  …

INST(x) =  Y(x)
ENCAPS(e,x) = x(e)
```

Other very similar encapsulation operators can be envisaged depending on their interaction with the different inheritance operators. Especially the interaction with encapsulated generators could prove to be very interesting. Here again, we will not investigate further the different forms of encapsulation operators in a formal setting. Nor will we develop a complete set of operators that explores the different combinations between the more primitive inheritance and encapsulation operators.

We saw two forms of encapsulation operators. One that operates on generator functions and that is mainly used to structure the encapsulated part of an object with respect to inheritance. The other, more interesting one, operates on objects. One object can be encapsulated in an other object. This form of encapsulation offers an object to gain acquaintances after instantiation. It is thus comparable to argument passing. An interesting application of this encapsulation operator is user defined control structures. Both kinds of encapsulation operators will be illustrated in the remainder of this text. The encapsulation operator on objects will be used in the construction of a calculus of objects. An encapsulation operator similar to the one on generators will be used in the construction of a programming language with mixin-based inheritance where mixins can be nested.

3.2.8 Objects with State, State Changes and Object Identity

In most practical object-oriented languages objects have a state that can be changed. State changes are most often realised by having so called instance variables that can be assigned values. An instance variable is associated with an object.

It should be evident from above that, although we find state and state changes important issues in OOPL, we do not find state and state changes as one of the determining features for 'object-orientation'. That is, in our definition of what an object is, an object must not have state to be called an object. We value objects for their abstraction capabilities, more or less comparable to abstract datatypes. Encapsulation does not necessarily mean 'an encapsulated state' (although the notion of encapsulated state is by now part of the 'folklore' of the object-oriented community), but rather that an object can use its own private resources to realise its behaviour. Vice versa an object can also have a 'public state', i.e. variables that are part of the public interface of an object, provided that variables and methods are accessed in a uniform way (e.g. variable access with accessor methods).

Object Identity, Strong versus Weak Identity

An important correlated notion is that of *object identity*. In an OOPL that supports identity each object is assigned a unique identity. This identity is independent of the values of the object's attributes. Identity is kept over state changes and can be used to uniquely refer to an object. Given two objects it can be tested whether they have the same identity. Object identity also is an important issue in defining different copying (object cloning) strategies.

We will call an OOPL where there is a one to one mapping of objects to object identities an OOPL with *strong object identity*. Thus, in a system with strong object identity with each identity corresponds one object, and each object is assigned one identity. Class-based OOPL typically are languages with strong object identity.

The latter constraint is relaxed in OOPLs that allow some form of object-based inheritance (or delegation). With object-based inheritance an existing object — and consequently one that already has an identity — is extended to form a new object. The newly created object is assigned a new identity. Still, the object that is being extended is part of the extended object.

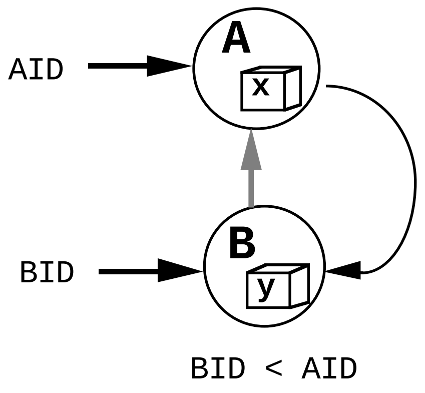

Figure 3.6

On the one hand the original object is a part of the extended object since its interface and implementation are used directly as part of the interface and implementation of the extended object. In this sense it is different than just any acquaintance of the extended object. The extended object can use acquaintances to realise its behaviour, but their interface and implementation does not become a part of the extended object.

On the other hand the original object is a part of the extended object since some of the changes to the extended object are identical to changes to the original object. This is so for changes due to messages sent to the extended object that are implemented in the original object. This phenomenon is normally referred to as inheritance of state. The state of the original object can be accessed through two paths: the identity of the original object and the identity of the extended object.

The above discussion leads us to say that OOPLs with object-based inheritance have objects with *weak identity*. There is a relation between the identity of the original and the extended object. The identity of the original object is a subidentity of the identity of the extended object, i.e. apart from an operator to test whether two objects have equal identity, it makes sense to provide an operator to test whether one object has an identity that is a subidentity of the identity of another object. Similar observations have been made in

[Dony,Malenfant&Cointe92] where prototype-based OOPLs are classified, amongst other criteria, according to the extent to which they support 'split objects'.

Changing Identity
An important question related to object identity is in how far an object can change identity or in how much the object that is associated with some identity can change. The latter can vary from being restricted to state changes only, to having an explicit operator that swaps the identities of two arbitrary objects (such as the become operator in Smalltalk).

It seems natural that due to the importance of an object's interface, identity changes should be restricted to objects with the same interface. This ensures that all objects referenced through one identity have the same interface.

On the other hand in some cases one can observe major changes in the behaviour of an object due to (small) state changes (i.e. a person object becomes a PhD person due to graduation). Practical solutions such as *dynamic reclassification* [Hamer92], unanticipated delegation or more recently predicate classes [Chambers93] have been proposed to cover such phenomena. It should be kept in mind, however, that these mechanisms have a substantial impact on the relation between an object and its identity.

3.3 Object-Oriented Frameworks

Inheritance is a powerful technique for structuring the code in one' s program. It allows common program-fragments to be factored out in superclasses. This ' factoring out' ultimately leads to skeleton classes that define only an abstract implementation. Such an *abstract class* can be reused to build a variety of concrete subclasses. A concrete subclass tailors the abstract class to its specific needs by filling in the methods used, but not implemented in the abstract class. *Object-oriented frameworks* grew out of the practical application of this sort of object-oriented techniques while building computational systems. They grew out of the observation that inheritance and late-binding polymorphism are powerful abstraction mechanisms, and that programs expressed in an object-oriented programming language can be reused by incrementally adapting them to different needs. Among the earliest examples of object-oriented application frameworks was the Smalltalk Model/View/Controller framework [Goldberg&Robson89].

According to [Wirfs-Brock90], an object-oriented framework is a skeleton implementation of an application or application subsystem in a particular problem domain. It is composed of concrete and abstract classes and provides a model of interaction or collaboration among the instances of classes defined by the framework. An important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in co-ordinating and sequencing application activity. This inversion of control gives the framework the power to serve as extensible skeleton. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application [Johnson&Foote91].

Frameworks are an emerging technique; there is still much to be learned concerning the design, configuration, and architecture-level description of frameworks [Opdyke92]. They have been investigated mainly in the context of design and code reuse. There is not yet an agreed definition of what exactly is a framework. Furthermore, most frameworks are defined informally; the constraints that are imposed on the components of a framework are informal constraints [Kiczales&Lamping92]. Work on formally expressing constraints can be found in [Holland92][Helm&al90].

Although frameworks are typically explored in the context of reusability, it is our intention to use frameworks as a means to express open systems. It has already been noted that: a framework is a way of representing the theory of how one should solve the problems in a particular area [Johnson90]. Therefore it shares much of the properties of an open system. The relation with open implemented systems has already been noted in [Holland92].

In this section we will explore the notions of abstract classes and object-oriented frameworks. More importantly we will show how a framework can be used to express an open system.

When using frameworks to express open systems two aspects of frameworks need to be emphasised. First of all the distinction between a *framework' s external interface* (corresponding to the object-level interface) and the *frameworks internal interface* (corresponding to the meta-level interface) must be made more explicit. Secondly special attention must be paid to those transformations on the framework that preserve the design of the framework. Particularly *refinement*, (partial) *concretisation*, and extensions to a framework will be discussed.

For the question how frameworks are developed by generalising a set of concrete applications, and vice versa, how they are tested and refined for reusability by building applications that use it, we refer the reader to [Opdyke&Johnson90] [Johnson&Russo91].

3.3.1 Reusability in Object-Oriented Programs

Programs expressed in an object-oriented programming language are, by the very nature of object-orientation, reusable. This does not mean however that every object-oriented program constitutes a full-fledged framework. Before discussing the properties that distinguish a framework from a run of the mill object-oriented program, we will first show in what ways reusability is supported by the object-oriented paradigm.

Late-binding Polymorphism

A polymorphic function or method is more reusable than a monomorphic one. It can be reused with different types of arguments. All arguments of a method in an object-oriented program can be used in a polymorphic way. Every object that conforms to the correct protocol can be used as actual argument. Due to late-binding, a method can rely on the fact that all messages sent to its arguments will invoke the right methods. A method is, as such, reusable for different types of arguments.

It should be noted that *late-binding polymorphism* is a more reusable form of polymorphism than the one that can be found in more conventional languages (e.g. ADA). In the latter polymorphism is more a typing issue, and polymorphic functions are less reusable since they must rely on case analysis of their polymorphic arguments to work. In this respect late binding is more comparable to the overloading facilities that can be found in such languages.

Moreover, in object-oriented programming languages where bounded polymorphism is used, it is possible to constrain the type of polymorphic arguments. Such a constraint is ideally expressed as a behaviour specification to which all the polymorphic arguments must conform. This gives rise to a subtype relation that is based on the principle of *substitutability* [Wegner&Zdonik88] [Liskov87]:

> ***Principle of Substitutability**: An instance of a subtype can always be used in any context in which an instance of a supertype was expected.*

In practice, constraints in bounded polymorphism are not formally expressed as behavioural specifications. They are either based on signatures or purely on protocols (i.e. the names of the public attributes of an object). Constraints are then enforced by a type checking algorithm.

Incremental Modification (Inheritance)
Inheritance lets an inheritor reuse the code of its parent in the form of inherited attributes. Consequently all inheritable entities (classes and objects) have two roles and, accordingly, two interfaces. A class, for example, is used for instantiation[2], i.e. it has a protocol for creating instances; and it is used for deriving new (inheriting) classes, i.e. it has an inheritance protocol.

The underlying mechanisms of inheritance have been amply discussed. In practice, inheritance can serve two different purposes. It can be used as a mere code reuse mechanism, i.e. there is no (conceptual) relation between an inheritor and its parent. Or, it can be used as a classification mechanism, in which inheritance is restricted to those cases where a certain relation between parent and inheritor exist. For example inheritance can be restricted such that only *behaviourally compatible inheritance* [Wegner90] is allowed, i.e. the inheritor must be a behaviourally compatible subtype of its parent. With respect to polymorphism, behaviourally compatible inheritance means that an inheritor may be used in any polymorphic context where the parent can be used. Other restrictions to inheritance exist, of which the restriction to *signature compatible inheritance* [Wegner90] is the most common one. In the latter case inheritance can also be used for type checking purposes. The distinction between inheritance as a mere code reuse mechanism and inheritance as a classification mechanism is an important one (especially in relation to polymorphism). We will come back to this later.

Genericity: Generic Classes, Type Substitution, Object Factories
In object-oriented programming languages a whole range of *genericity mechanisms* has been explored. Examples abound: generic classes (Eiffel [Meyer88]), templates (C++ [Ellis&Stroustrup90]), type substitution [Palsberg&Schwartzbach90], object factories [Gamma&al.93] and so on. In general, genericity allows a generic class to be instantiated with a set of classes and types. As we will see, it is an important way to make a class more reusable. Since the concept of genericity is less well-explored as a code reuse mechanism in object-oriented languages than e.g. inheritance (it is considered less germane to object-orientation), we find it important enough to give a brief overview of some of the mechanisms that introduce genericity in object-oriented languages. Especially since we agree with [Palsberg&Schwartzbach90] that, in a special

[2] Only concrete classes are considered here, we will have more to say on abstract classes later on.

way, it can be considered as a complementary inheritance mechanism.

A *generic class* is a class that is parameterised (typically with a set of types). The generic parameters can be used in the definition of the generic class. A generic class is instantiated, to an actual class, by providing a set of actual arguments for the generic parameters. Typically generic classes are associated with bounded polymorphism, i.e. the formal types of the generic parameters put bounds on the possible actual generic parameters. Below is an example[3] of a generic list class, and its instantiation to a list of integers.

```
generic class List (NodeType:Object)
  instance variables
    head:NodeType
    tail:List
  methods
    head returns NodeType
      ^head
    tail returns List
      ^tail
endclass

class ListOfInteger is List(Integer)
```

In a statically typed language a generic class can be reused by adapting it to different types. Parametric polymorphism, which is typically associated with generic classes, is generally considered as a form of polymorphism that is orthogonal to late-binding polymorphism [Palsberg&Schwartzbach90]. Also in weakly and dynamically typed languages, generic classes can play an important role for reusability. A class is client of other classes of which it creates instances. A class can be made more reusable by making it generic with respect to the classes of which it creates instances. In the remainder of this text we will see plenty of examples of this. For example, in the implementation of object-oriented languages, an important kind of expressions are expressions that create, so called, slots (i.e. representations of methods and instance variables). These sorts of expressions can be made more reusable by making them generic to the kind of slots that are created. Another good example (example from [Gamma&al.93]) can be found in the usage of user interface toolkits. A toolkit requires different controls such as scroll bars, buttons, or text editors. An application should not hard code dependencies on the look-and-feel: it should be made generic with respect to which toolkit is used.

Generic classes can not be reused in the same flexible way as classes can be reused through inheritance. They require forethought about what is listed in the generic parameters. Furthermore, generic classes differ from ‘ordinary’ classes. In general it is not possible to inherit from a generic class. Nor is it possible to gradually instantiate a generic class, i.e. it is not possible to further specialise a generic parameter after the generic class has been instantiated. This limits generic classes as a code reuse mechanism.

Alternatives to generic classes exist that are more suitable for code reuse. In [Palsberg&Schwartzbach90] a mechanism called *type substitution* is introduced as an alternative to generic classes. Type substitution is a subclassing mechanism that complements inheritance. It allows all types used in a class to be gradually specialised, without the need of listing these types as generic parameters. As such it avoids the limitations associated with generic classes. In

[3] The above 'pseudo' language is used for those examples and other class descriptions where the description language is irrelevant. This pseudo language is for the larger part self-explanatory, just note that its message passing syntax is derived from Smalltalk's message passing syntax [Goldberg&Robson89].

[Palsberg&Schwartzbach90] it is shown that type substitution and inheritance form an orthogonal basis for code reuse in the definition of classes. Below is the same generic list example of above but with type substitution[4]. It is shown how a type can be substituted after inheriting from the original list class. Furthermore it is shown how types can be gradually specialised.

```
class List
  instance variables
    head:Object
    tail:List
  methods
    head returns Object
      ^head
    tail returns List
      ^tail
endclass

class DoubleLinkedList extends List
  instance variables
    prev:DoubleLinkedList
  methods
    prev returns DoubleLinkedList
      ^prev
endclass

class DLListOfNumber extends DoubleLinkedList[Object<-Number]
class DLListOfInteger extends DLListOfNumber[Number<-Integer]
```

Type substitution was explored in a statically typed language. Obviously only "type consistent" substitutions are allowed. As is the case with generic classes, a mechanism similar to type substitution can also be used in dynamically and untyped languages. In that case this mechanism is used to make classes more reusable with respect to the classes of which instances are created.

[Madsen,Magnusson&Møller-Pedersen90] investigate the use of "virtual classes" (we will use the term *virtual class attributes*, in our opinion this term reflects better the meaning of the concept) as a means to express genericity. Virtual class attributes are a typical language feature of BETA [Kristensen&al.87]. A virtual class attribute is an attribute much in the style of an instance variable, but that contains a class. This attribute can be modified in later subclasses. Below one can find the list example with virtual class attributes.

```
class List
  virtual class attributes
    NodeType:Object
  instance variables
    head:NodeType
    tail:List
  methods
    head returns NodeType
      ^head
    tail returns List
      ^tail
endclass
```

4 Notice that in the example recursive class definitions are used and that this recursion must be 'extended' in inheritors. We will not deal with this issue in this or later examples. See [Palsberg&Schwartzbach90] for a discussion.

```
class DoubleLinkedList extends List
  instance variables
    prev:DoubleLinkedList
  methods
    prev returns DoubleLinkedList
      ^prev
endclass

class DLListOfInteger extends DoubleLinkedList
  virtual class attributes
    NodeType:Integer
endclass
```

Virtual class attributes share many of the advantages of type substitution over generic classes. They require forethought in the list of virtually declared class attributes however. A class is only generic with respect to the classes that are listed as virtual class attributes. This need not be the case however. In slot-based languages where slots are accessed through message passing (e.g. Self [Ungar&Smith87]), it is generally so that all attributes can be overridden, including instance variables. In that case virtual classes and type substitution are equivalent code reuse mechanisms.

In analogy with abstract methods, and for symmetry reasons that will become apparent in the next section we will use *abstract class attributes* to express genericity. We will see that abstract class attributes are comparable to virtual class attributes for an untyped language. We opted for a different name since virtual class attributes are so much associated with their realisation in BETA, and because it makes the symmetry with abstract methods apparent in the terminology.

Finally, *object factories* [Gamma&al.93] are worth noting as a technique to introduce genericity in classes. Unlike the above three techniques object factories can be used in almost any object-oriented programming language. It consists of a set of conventions that one imposes on one's self. It grew out of a practical concern that hard coding the names of classes of which one creates instances drastically reduces the reusability of one's code. Rather than directly referring to a class name to create a new instance, one refers to a, so called, object factory to create a new instance. It is possible to have different object factories in a single program. Each factory groups related classes. It is possible to override the creation methods that are defined on the factory objects in order to "install" new classes. Object factories are especially useful when the classes of which instances must be created change dynamically. This is not well-covered by the above techniques.

Encapsulation
The implementation of an object's behaviour is not visible to the clients of that object. Due to this all objects that implement the same behaviour can be reused transparent of their implementation.

As will amply be discussed in the sections on multiple inheritance, encapsulation of inheritance, also, is important for reuse potential. In all cases where the inheritance structure of an object is exposed, and where users of that object depend on this, reuse is seriously hampered.

In case of statically typed programming languages this is an important issue. Typing should only be based on the behaviour of objects, not on their implementation. In the case of inheritance this means that subtyping and inheritance are to be considered as different mechanisms, a generally acknowledged fact in the object-oriented community [Shan&al.93]. All languages that confuse subtyping and inheritance have less code reuse potential.

3.3.2 Reusability in Object-Oriented Frameworks

Not every object-oriented program is a framework, regardless of the fact that object-oriented programs are “de facto” reusable. What distinguishes a framework from a merely reusable application is that the major design issues of the application are made explicit. The kind of reuse that is made possible in a program without further additions can best be typified by code reuse, i.e. no relation exists between the program that is reused and the program that reuses. What we intended was *design reuse*, i.e. the program that reuses must be able to know and respect all the major design issues of the program that is reused.

In a framework the major design issues are made explicit by means of *abstract classes*. Abstract classes form the skeleton of an object-oriented framework. Therefore they will be discussed next.

Furthermore the framework must be reused in such a way that the major design choices are respected. That is to say, even when we know what the major design issues are, it still is possible to (code) reuse the framework in such a way that these design issues are violated. This is discussed in the section on operations on abstract classes.

3.3.3 Abstract Classes

In general terms, an abstract class is a class that is only partially implemented. Before making use of the abstract class it must be made concrete by “filling in” the missing details in the implementation. Conventionally only classes with abstract methods are called abstract classes. We will extend the notion of an abstract class to classes that have an abstract acquaintance.

Abstract Methods
The first kind of abstract class that is considered is a class that implements one set of methods, called the *template methods* [Johnson&Russo91], in terms of another set of unimplemented methods, called the *abstract methods* (*or virtual methods*) . Abstract methods are in most cases public methods, but can be private also. Instances of classes with abstract methods can not be used, since their implementation is incomplete. The different kinds of methods can be defined as below.

> ***An abstract method*** *is a method that has no implementation, and is formally declared as such.*
>
> ***A template method*** *is a method that has an implementation but that calls either directly or indirectly an abstract method. Thus, a method that calls another template method is itself a template method, since it will indirectly call an abstract method.*
>
> ***A concrete method*** *is a method that has an implementation and that does not rely on abstract or template methods.*

An abstract method can be made concrete in a subclass by overriding it with a concrete method. The template methods are reused through inheritance. The question under what conditions template methods can be overridden will be discussed later. Abstract methods are a design issue. Programming language support for abstract methods is not available in all object-oriented programming languages. Although concretisation is generally supported (through inheritance and overriding), no support is generally available for indicating that a particular method is a template method or an abstract method. Let alone, that restrictions on the rules for overriding template and abstract methods are enforced.

An abstract class that uses abstract methods provides design information by reifying the algorithmic decomposition of template methods.

Abstract Acquaintances
The second kind of abstract classes are classes that use other unknown classes in their implementation. In particular, classes that create instances and that have not yet decided which concrete class these instances must be instantiated from. These are called classes with an *abstract acquaintance* (where the abstract acquaintance is the referenced class). Again, instances of classes with an abstract acquaintance can not be used, since their implementation is incomplete.

> ***An abstract class attribute*** *is an attribute that can be used in the implementation to refer to a class, but contains no reference to a class and must be overridden in a subclass to do so.*
>
> ***A template method*** *is a template method in the sense of the previous definition or is a method that refers either directly or indirectly to an abstract class attribute to create instances from.*

A class with abstract acquaintances can be made concrete, by substituting all abstract acquaintances that are used in the implementation, by concrete classes. Here again, few object-oriented programming languages provide support for this kind of concretisation. Language support varies from the above discussed generic classes [Meyer88] or type substitution [Palsberg&Schwartzbach90] to ad hoc solutions such as factory objects [Gamma&al.93]. Here we will study abstract class attributes as a means to express abstract acquaintances.

Abstract class attributes are dual to abstract methods. The implementation of a class is expressed in terms of a set of class attributes that are declared abstract. In the same way that there is no implementation associated with an abstract method, there is no concrete class associated with an abstract class attribute. An abstract class attribute can be made concrete in a subclass by overriding it with a concrete class.

An abstract class that uses abstract acquaintances, provides design information by reifying the aggregation structure of instances of this abstract class.

3.3.4 Operations on Abstract Classes

The external interface of an abstract class is, obviously, its list of publicly visible methods. The internal interface is directed towards inheritors of the abstract class [Deutsch87]. It specifies which, and the constraints under which method and class attributes can be overridden. In this section we will look at this in greater detail. We will discuss how an abstract class can be reused such that the design of the abstract class is "respected".

The idea is to see what the effect is of, and what constraints should be in effect when overriding abstract class attributes with concrete class attributes and abstract, template and concrete methods by abstract, template and concrete methods. First we will restate our extended definition of an abstract class.

> ***An abstract class*** *is a class that contains at least one abstract method or an abstract class attribute. It can not be instantiated since its implementation is not complete. A class that is not abstract is a concrete class.*

In the following example, methods x, and y are abstract, methods a, b, and c are template methods and method f is a concrete method. P is an abstract class attribute.

```
class Example
  abstract class attributes
    P
  methods
    abstract x:anArgument
    abstract y
    template a
      self x:3
    template b
      self a
    template c
      ^P x:4 y:8
    concrete f
      ^3
endclass
```

Concretisation of the Abstract Class
An abstract class can be made more concrete by overriding its abstract methods or its abstract class attributes. An abstract method can be overridden with either a template method or a concrete method. All other methods in the abstract class are inherited.

A concretisation of an abstract class can be either a concrete class, or, again, an abstract class. In this latter case the concretisation is partial. A concretisation can be partial due to the fact that not all abstract (methods and class) attributes are overridden, or that an abstract method is overridden with a template method. In the case where an abstract method is overridden with a template method, this template method can invoke already existing abstract methods (class attributes) or newly introduced abstract methods (class attributes). This latter will give rise to layered abstract classes, and in a larger context to layered frameworks, as in the following example. The abstract method “y” is overridden with a template method. This template method invokes a newly added abstract method “v”.

```
class SubExample extends Example
  methods
    abstract v
    template y
      self v
endclass
```

In absence of type information, not all concretisations give desirable results. It is possible to observe “message not understood” errors as could be the case in the following obvious example where the abstract method x is made concrete. Although it is possible to observe from the template method ' a' , that in at least one case the method ' x' is given an integer argument, in the concretisation of ' x' this argument is sent a ' push' message (a message that is probably not defined for integers).

```
class ErrorExample extends Example
  methods
    concrete x:anArgument
      anArgument push:4
endclass
```

In order to avoid e.g. message not understood errors, it needs to be specified more formally what possible concretisations are allowed for a certain abstract attribute. Commonly, for this purpose, the arguments of the abstract methods are typed and a type constraint for the abstract class attributes can be given. In the ideal situation, the constraints to what the concretisations of an abstract attribute should conform should be formally specified (see e.g. [Helm&al90] [Holland92]). For an abstract class attribute this constraint can be expressed as a behaviour specification, or constraints on the behaviour of the instances of the possible concretisations of the abstract class attribute.

In any case, the following terminology can be introduced (referred to as the *substitutability rule* for concretisation later on in the text). A concretisation is *substitutable* for an abstract attribute if it produces no (runtime) errors in the case no formal constraints are specified for the abstract attribute or it conforms to the constraints that are specified for the abstract attribute.

Refinement of the Abstract Class
An abstract class can be refined in different ways. First of all, if the abstract class contains concrete methods, then the abstract class can be refined by overriding existing concrete methods with new concrete methods. Secondly, and more importantly, an abstract class that contains template methods, can be refined by overriding a template method with a new template method or a concrete method. In both cases all other existing template, abstract, and concrete methods are inherited from the abstract class. A refinement of an abstract class is, again, an abstract class (the amount of abstract method does not decrease by refinement). It is especially important to notice the difference between a refinement of an abstract class and a partial concretisation of an abstract class. Although both result in an abstract class they do so in a different way.

Not all refinements respect the design of the abstract class. An example of such a refinement is found below. A template method encodes, for its inheritors, design information about the decomposition of an algorithm. The refinement of class `Example` below does not respect the decomposition of the “a” template method.

```
class BadRefinement extends Example
  methods
    concrete a
      self c
endclass
```

Whether a refinement respects the design of an abstract class can be stated more formally as follows:

Refinement Constraint for Abstract Classes*: A refinement of an abstract class respects the design of that class, if each concretisation of the refinement of the abstract class is substitutable for the same concretisation of the abstract class.*

In other words, when making an abstract class concrete, it is transparent whether one makes the abstract class concrete or whether a refinement that respects the design of this abstract class is made concrete.

Extension of the Abstract Class

An abstract class can be extended by adding new abstract attributes or, template, or concrete methods in its inheritors. It is possible to distinguish two sorts of extensions: extensions that depend on abstract attributes or template methods of the abstract class, and extensions that are independent of the abstract attributes and template methods of the abstract class. We will concentrate on the former kind of extensions.

```
class ExtendedExample extends Example
  methods
    template q
      self x:"a"
endclass
```

In this case we must see to it that an extension that refers to an abstract attribute does not expect more from this attribute than the original abstract class did.

> ***Interoperability Constraint for Extension of Abstract Classes:*** *Insofar that a concretisation is substitutable for a particular abstract attribute in the abstract class, this concretisation must be substitutable for this abstract attribute in the extension of the abstract class.*

Stated otherwise, an extension of an abstract class must not presume, in its reference to abstract attributes, any constraints that are not explicit in the abstract class.

Abstraction of the Abstract Class

An abstract class can be made more abstract by e.g. overriding a concrete or template method with an abstract method. This kind of operations has the intention of generalising the framework rather than making it more concrete. This sort of transformation of a framework lies more in the realm of refactoring [Opdyke92] or iterating over the design of the framework with the intention to broaden its applicability. This is beyond the scope of this text.

3.3.5 Role of Abstract Classes in Frameworks

In the previous section we discussed the operations and constraints on these operations that allow us to reuse the design of an abstract class. Without this sort of constraints abstract classes can only be reused in the sense of code reuse.

An abstract class is the design for a single object. A framework is the design of a set of objects that collaborate to carry out a set of responsibilities. Such a set of objects is called an *ensemble* [Johnson&Foote88]. We now turn to the question of how all of the above can be extended to full-fledged frameworks, and what the role of abstract classes is therein.

Forming Ensembles

In the previous section we saw that not all concretisations are allowed for a certain abstract attribute. Two possible ways to specify constraints were pointed out, i.e. typing abstract methods, and class attributes, or, in the particular case of an abstract class attribute, giving a true behaviour specification. In this section we consider a third alternative: that of using a concrete or abstract class (concrete or template method) as a constraint for an abstract class attribute (abstract method).

Based on the *substitutability rule* of before, a (concrete) class specifies an entire set of behaviours: it specifies all behaviours of objects that are substitutable for its instances. So, a concrete class can be used to constrain the possible concretisations of an abstract class attribute. Only those concretisations are allowed of which the instances are substitutable for the instances of the constraining concrete class.

Similarly an abstract class can be used as a concretisation constraint for an abstract class attribute. Only those concretisations are allowed that are refinements, concretisations, or extensions of the constraining abstract class. Consider our abstract class example from before. The abstract class attribute P can be constrained to being a derivation of some abstract point class, forming a contract between the users of the abstract class attributes (only the external point interface may be used), and the specialisers of the abstract class attribute (only specialisations of the abstract point class may be used to make P concrete). This is how in practice frameworks are created and documented.

```
class Example
  abstract class attributes
    P:AbstractPoint
  methods
    abstract x:anArgument
    abstract y
    template a
      self x:3
    template b
      self a
    template c
      ^P x:4 y:8
    concrete f
      ^3
endclass
```

Two notes must be made here. The first note is that one could be tempted to conclude that if all abstract class (method) attributes are constrained by abstract or concrete classes (template or concrete methods), necessarily ending at the leaf nodes with concrete classes, then we end up with a plain concrete class. This is not so. The difference is that now the contracts between the major parts of the implementation of the concrete class are made explicit (for a good discussion on this in the specific case of methods see [Lamping93]). And moreover they may not be violated ! This is what we wanted. Both a plain concrete class and the abstract class that is made concrete through its concretisation constraints define an entire design space of (substitutional) behaviours. The difference between the two is that in the latter, the design of the class is made explicit.

The second note is about how ensembles are created. Consider an abstract class with a set of abstract class attributes (constrained by abstract classes), and a set of template methods that create instances of and use these abstract class attributes. Such a configuration specifies an ensemble. The template methods specify how the objects in the ensemble work together.

3.3.6 Frameworks, Conclusion

Although much of the facilities that come with object-oriented programming help in expressing reusable systems, not every reusable object-oriented program forms a framework. In order to study frameworks in the context of building systems with an open design two aspect of frameworks were emphasised. The first is the emphasis on the distinction between the external interface of the framework, and the internal interface of the framework. These interfaces are inherited from the abstract classes that constitute the framework. Secondly emphasis was put on the constraints that surround the derivation of systems or new frameworks from an existing framework. These constraints guard against using the internal interface of the framework without respect for the major design issues that were made explicit. For concretisation of the framework in particular, we saw that the constraints on abstract class attributes can go beyond mere interface or protocol specifications. In a practical setting we can rely on concrete, or again abstract classes, and the substitutability principle to define more fine-grained constraints on alternative concretisations of abstract class attributes. An abstract class that makes use of a collection of abstract class attributes in its template methods, specifies an ensemble.

3.4 A Simple Object-based Programming Language

The programming language that will be presented here is a very elementary object-based programming language that still carries the essential object-oriented features in it. It is directly motivated by the analysis of object-oriented features in the previous section.

The thus obtained language is so simple that its semantics can be given in the form of a calculus. In this section we will explain this calculus by giving the rewrite rules and examples. In the following sections we will interpret this calculus as a programming language, by giving an implementation in the form of an evaluator for it. Essentially the calculus and the programming language differ in the order of evaluation of subexpressions. In the former evaluation order is arbitrary, in the latter evaluation order is fixed. To emphasise the difference between the two the calculus will be referred to as *OPUS* (Object-based Programming calculUS), the programming language will be referred to as *Simple*.

The calculus that will be presented here is still under development [Steyaert92] [Mens,Mens&Steyaert94]. Still, for our purpose here, it has reached a sufficient level of maturity. Only a part of OPUS will be explained, for issues such as recursion, inheritance, classes, etc. the reader is referred to the above references. A rough understanding of OPUS suffices so that the basic concepts of implementational structures of object-based programming languages can be explained.

3.4.1 A Calculus for Object-based Programming

OPUS is designed as an elementary calculus to express object-orientation. It models, in a direct way, the crucial features of object-based programming, i.e. objects, encapsulation and, message passing. OPUS has strongly encapsulated objects, unary methods, instance variables and unary message passing. The objects in the calculus can not change state. Inheritance is not included. No provisions are made for recursion (e.g. “self” sends), although it can be shown how recursive objects can be constructed (but we will not do so here, see [Steyaert92] [Mens,Mens&Steyaert94]).

First of all, OPUS explicitly uses names for message passing. As already discussed in the literature, the use of names greatly simplifies the modelling of object-oriented systems. Examples of this are: the lambda calculus augmented with records (cf. [Cardelli88], [Cardelli&Mitchell89]), Milner' s π-calculus for describing concurrent computations (cf. [Milner91]) and the lambda-calculus augmented with names, combinations and alternations, as presented in [Dami93a], which is shown to be more expressive than the lambda-calculus with records.

Secondly it has an explicit encapsulation operator. An important aspect of the calculus is that both the public part and the encapsulated part of an object are objects in their own right; unbound variables in public methods are seen as a form of private method invocation. The public part of an object can contain instance variables (public instance variables); the encapsulated part can contain methods (private methods).

We first proceed with a brief introduction to the calculus. The purpose is to get an idea of the meaning of expressions in this calculus.

Concrete Grammar
An OPUS-expression is *well-formed* if it is an element of the language generated by the following context free grammar.

OPUS concrete grammar:

```
Non-Terminal labels   = { Expression Application Abstraction
                          BaseObject CompoundObject Association
                          MethodAssociation VariableAssociation Literal }
Terminal labels       = { Name }
Start label           = Expression

Expression           -> Application | Abstraction | Name | Literal
Application          -> "(" Expression ")" Name
Abstraction          -> BaseObject | CompoundObject
CompoundObject       -> "<" Expression "," Expression ">"
BaseObject           -> "[" [ Association {";" Association}] "]"
Association          -> MethodAssociation | VariableAssociation
VariableAssociation  -> Name "." Expression
MethodAssociation    -> Name "#" Expression
Literal              -> "0" | "1" | … | "9"
```

Objects and Message Passing in the Calculus
OPUS' s base objects and message passing are comparable to records and record selection. They behave as follows:

object	message expressions	result
[]	([])x	normal form
[x.1; y.2; z.3]	([x.1; y.2; z.3])y	-> 2
[x.a; y.b]	([x.a; y.b])y	-> b

Base objects of this kind are constructed entirely of 'instance variable associations'. Free variables are bound via lexical scoping (as is hinted at in the last example). The rule for message passing to such base objects is:

Rule 1a: Instance variable selection in a base object (lexical scoping)

$$([\ldots ; x_i.e_i ; \ldots]) \, x_i \rightarrow e_i, \text{ if } \forall \, j<i: x_j \neq x_i$$

In order to model an object we introduce two extra concepts: 'method associations' and 'compound objects'. A compound object is the composition of a public part (mostly a base object with method associations) and an encapsulated part (mostly a base object with instance variable associations); we use the following notation: <*Public,Private*>.

The selection of instance variable associations in compound objects is a straightforward extension of Rule 1a.

Rule 1b: Instance variable selection in a compound object (lexical scoping)
$(< [\ldots ; x_i.e_i ; \ldots] , d >)\ x_i \rightarrow e_i,\ \text{if } \forall\, j<i : x_j \neq x_i$

Method associations (x#e) differ from instance variable associations (x.e) in their scoping: the free variables of a method association in the public part of a compound object, are bound via the encapsulated part of that compound object. An example:

```
(<[x#a; y#b], [a.3]>)x
-> {[a.3]}a                 evaluating the body of method x (i.e. "a")
                            in the encapsulated part (i.e. [a.3])
= ([a.3])a
-> 3
```

We will use the notation {c}d to denote the evaluation of an expression d in a context c. The rules for evaluating an expression in a context will be explained below. Given this notation we can now express the selection of methods.

Rule 2a: Method selection in a base object
$(\ [\ldots ; x_i\#e_i ; \ldots]\ )\ x_i \rightarrow \{[]\}e_i,\ \text{if } \forall\, j<i : x_j \neq x_i$

Rule 2b: Method selection in a compound object
$(\ < [\ldots ; x_i\#e_i ; \ldots], d >\ )\ x_i \rightarrow \{d\}e_i,\ \text{if } \forall\, j<i : x_j \neq x_i$

In these rules we use a notation {c}d that is not formally a part of the calculus, i.e. it is not a well-formed expression. It must be considered as some kind of meta-definition for the *evaluation of an expression d in a context c*. Formally, the definition of {c}d can be given inductively as follows:

If d is a literal	**then** {c}d = d
If d is a name	**then** {c}d = (c)d
If d is an application (e) x	**then** {c}d = ({c}e) x
If d is a compound object <e , f>	**then** {c}d = <{c}e, {c}f>
If d is a base object $[\ldots x_i.e_i;\ldots y_i\#f_i;\ldots]$	**then** $\{c\}d = [\ldots x_i.\{c\}e_i;\ldots y_i\#f_i;\ldots]$

Remark that evaluating a base object in a context c yields a new base object where *only the instance variables are evaluated in the context*, while the methods remain unaltered. In principle evaluating an expression in some context distributes over all subexpression, except for method associations (method associations are evaluated in the encapsulated part of the object to which they belong, according to rule 2). Evaluating an unbound variable (a name) in a context is, simply, sending that name to the context. Note that the context, in which an expression is evaluated, can be an arbitrary object.

Two examples will illustrate these rules.

```
(<[x#[k.a; j.b]], [a.3; b.4]>)x
  -> {[a.3; b.4]}[k.a; j.b]                              (2b)
  =  [k.([a.3; b.4])a; j.([a.3; b.4])b]
  -> [k.([a.3; b.4])a; j.4]                              (1a)
  -> [k.3; j.4]                                          (1a)

(<[x#<[m#a], [a.b]>], [a.3; b.4]>)x
  -> {[a.3; b.4]}<[m#a], [a.b]>                          (2b)
  =  <[m#a], [a.([a.3; b.4])b]>
  -> <[m#a], [a.4]>                                      (1a)
```

These examples show how objects can be returned as a result of "method invocation", and the effect on the binding of free variables. We can also pass objects (as "arguments") to methods; as an example of this we show the construction of boolean values. The method 'foo' expects as input a condition:

```
[if#true]                                           true object
[if#false]                                         false object

[foo#(<condition,[true.1;false.2]>)if]          object with one
                                                    method: foo

(<[foo#(<condition,[true.1;false.2]>)if], [condition.[if#true]]>)foo
-> {[condition.[if#true]} (<condition, [true.1;false.2]>)if      (2b)
=  (<[if#true],[true.1;false.2]>)if
-> {[true.1;false.2]}true                                        (2b)
=  1
```

For modelling objects the most important advantage of the calculus over lambda-calculus with records is that both parts of a compound object can be, again, compound objects. This is essential on the one hand to model private methods, and on the other hand to have some form of curried binding of instance variables. The key insight is that an unbound variable in a selected method is seen as a message with an implicit receiver: the encapsulated part of the compound object of which the method was a part. The interaction between the encapsulated part and the public part of a compound object, is that a selected method from the public part sends its unbound variables to the encapsulated part and conversely that the encapsulated part is an object that contains methods for the unbound variables in a selected method from the public part.

Although in the previous rules and examples it was shown that the private part of a compound object could be a compound object again, the public part was always a base object. So, we need to consider the case were the public part also is a compound object. This will be called curried binding of private attributes. The rule for curried binding is simply as follows:

Rule 3: currying
<<e, f>, g> $\rightarrow$ <e, <f, g>>

This rule can be shown to be consistent with the previous rules. It is essential for modelling objects to which the private attributes are bound in different stages. To give an example the foo method from the boolean example above can also be constructed as follows:

```
<[foo#(<condition,[true.1;false.2]>)if], [condition#condition]>
```

Sending the message foo, after binding an actual condition to the receiver object, as before, will give the same result. Not only does this make more clear what the expected instance variables (or arguments) are for the foo method, but also, now, the object to which the foo method belongs can have its own private attributes (see below). Argument passing could be modelled this way: arguments (the condition) are bound to an object in supplement to the already bound instance variables (the variable a).

```
<[foo#(<condition,[true.a;false.2]>)if], [a.3; condition#condition]>
```

3.5 Definition of the Framework

The above calculus defines an — albeit operational — semantics for our Simple programming language. Before proceeding with an implementation of Simple, we will first give a initial approximation of a framework for object-based programming languages. This initial approximation will be used to express an evaluator for Simple. More importantly, we will illustrate its shortcomings and, accordingly, improve the framework.

3.5.1 Representation of Programs and Compositionality

The representation of programs will be based on *abstract grammars*. For the syntactic aspects of programming languages there is a well-developed theory of formal languages that supports both the description of the syntax of programming languages and the representation of programs in programming languages. The theory of context free grammars [Chomsky56] provides the basis for most, if not all, aspects on defining the concrete syntax of programming languages. These grammars are used in most programming language implementations to "parse" the textual representation of a program into an internal program representation. This program representation is then used by the other components of the programming environment.

For the (internal) representation of programs there is a well developed theory of *abstract syntax trees* (AST's) [De Hondt93][Madsen&Nørgaard88] that originated in the Mentor project [Donzeau-Gouge&al.80]. Abstract syntax trees are also used in the definition of the semantics of programming languages [Schmidt86]. Whereas the concrete syntax concentrates on "parsability" issues such as precedence rules, and the exact textual representation of the terminal symbols in the grammar, the abstract syntax concentrates on the hierarchical structure of the grammar, i.e. on how compound expressions are composed of subexpressions.

Different forms of abstract grammars exist imposing more or less structure on the form of the grammar definition. We take the simplest form. Below is a definition of an abstract grammar for Simple. It is an abstract grammar that corresponds to the already given concrete grammar for OPUS (OPUS and Simple share the same concrete grammar). It takes the form of a set NT of non terminal nodes, a set T of terminal (or leaf) nodes, a collection of expansion sets, a set of production rules, and a root set.

The nonterminal nodes account for *composite expressions*, i.e. expressions that are compositions of subexpressions. For each composite expression a production rule exists, that gives the number and type of subexpressions. A composite expression can have either a fixed number or a variable number of subexpressions. Each subexpression can be again an expression of a limited kind, i.e. limited to an expansion set[5] of expressions. In the description of the production rules below, the sets ExpressionSet, PatternSet, etc. are used for this purpose. Terminal nodes account for expressions that have no further subexpressions. The root set is the expansion set of root expressions. Below, the set of possible start nodes (or root nodes) is ExpressionSet.

Simple Abstract Grammar

```
NonTerminal Labels      = { Application BaseObject CompoundObject
                            MethodAssociation VariableAssociation }
Terminal Labels         = { Name Literal }
Start Set               = Expression

ExpressionSet           = { Application } + AbstractionSet + PatternSet
PatternSet              = { Name }
AbstractionSet          = { BaseObject } + { CompoundObject }
AssociationSet          = { MethodAssociation } + { VariableAssociation }

Application               -> ExpressionSet x PatternSet
CompoundObject            -> ExpressionSet x ExpressionSet
BaseObject                -> AssociationSet*
VariableAssociation       -> PatternSet x ExpressionSet
MethodAssociation         -> PatternSet x ExpressionSet
```

Abstract grammars are easily transformed into a class hierarchy [Hedin89] [Madsen&Nørgaard88]. Expansion sets are translated into abstract classes. Each abstract class has as subclasses the classes that arise due to the translation of the elements of its expansion set. Each composite expression is translated into a concrete class having the subexpressions that comprise the composite expression as instance variable. The (expansion set) abstract classes are used for the typing of the components of a composite expression. Primitive expressions are translated into concrete classes that are not further aggregated.

The following class hierarchy is such a straightforward translation of a part of the abstract grammar for Simple. This class hierarchy contains the abstract classes AbstractExpression, AbstractionExpression, AssociationExpression and AbstractPattern that arise from the respective expansion sets ExpressionSet, AbstractionSet, A ssociationSet and P atternSet (the abstract classes are underlined). The concrete classes U naryMessageExpression, CompoundObjectExpression, BaseObjectExpression, MethodAssociationExpression and VariableAssociationExpression are the translation of the respective nonterminal nodes Application, CompoundObject, BaseObject, MethodAssociation and VariableAssociation. Finally, the terminal node Name is translated to the concrete class Pattern. Literals are not considered in the remainder of the text.

[5] The name ' expansion set' is derived from the fact that an expansion set contains the types of the nodes that can be used in the ' expansion' of a subexpression.

Simple Expression Class Hierarchy (first try)

```
AbstractExpression
  AbstractionExpression
   CompoundObjectExpression (publicPart:AbstractExpression,
                             privatePart:AbstractExpression)
   BaseObjectExpression (associations: Sequence(AssociationExpression))
   UnaryMessageExpression (receiver:AbstractExpression,
                           pattern:AbstractPattern)
   AbstractPattern
     Pattern(name:String)

AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
  VariableAssociationExpression
  MethodAssociationExpression
```

This simple translation scheme only takes the aggregation structure of expressions into account. A better, but less trivial, translation must take the expected behaviour of each of the elements of the abstract grammar into account. For example, when evaluation behaviour is defined for the expression classes, it can be observed that most, but not all, expressions need an evaluation method. This can be illustrated in the above class hierarchy. Syntactically there is no problem in having a common syntactical representation for patterns used directly as expressions (i.e. as identifiers, an expression that when evaluated looks itself up in the context) and the patterns that are part of a message expression. So, a trivial translation of an abstract grammar in which both kinds of patterns have the same representation would map both onto the same class. Furthermore this class would be a subclass of the abstract expression class, since patterns are elements of the expression expansion set.

Patterns, used as expressions, can be evaluated; their evaluation is looking them up in the context. On the other hand, the pattern part of a message expression is merely used as a unique identifier, and may not be evaluated.

The class hierarchy that results from the trivial translation does not give a hierarchy that is suited for adding evaluation behaviour. In the case of a pattern that is used directly as an expression, an evaluation method is needed. In the case of a pattern that is part of a message expression, one needs to be able to test whether two patterns are the same. An uncoupling of the classes for both kinds of patterns is in order. Both play a different role in the evaluator and accordingly will lead to different objects with different protocols.

Accordingly a new class hierarchy of expressions can be constructed. Remark the uncoupling of both forms of patterns.

Simple Expression Class Hierarchy (second try)

```
AbstractExpression
  AbstractionExpression
   CompoundObjectExpression (publicPart:AbstractExpression,
                             privatePart:AbstractExpression)
   BaseObjectExpression (associations:Sequence(AssociationExpression))
  UnaryMessageExpression (receiver:AbstractExpression,
                          pattern:AbstractPattern)
  PatternExpression (pattern:AbstractPattern)

AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
  VariableAssociationExpression
  MethodAssociationExpression

AbstractPattern
  Pattern(name:String)
```

Remark also that association expressions are not part of the expression class hierarchy. The association expansion set is not a subset of the expression set. At first sight this could easily be amended. The syntax could be extended to allow expressions such as `x.3`, which would be a syntactically simpler construct equal to `[x.3]`. This would unnecessarily complicate the explanation of the implementation of Simple.

In general, an uncoupling of classes that have the same aggregation structure, but play different roles in the evaluation process suffices in order to create a "good" class hierarchy. Further examples will be given in the remainder of the text.

Evaluation is expressed as a method that is defined on expression objects. Evaluation is done in a context (which for Simple is an object again). The result of evaluating an expression is an object. Compositionality is achieved by encapsulating the compositional structure of each expression, such that a composite expression can only rely on the evaluation methods of its subexpressions to express its own evaluation.

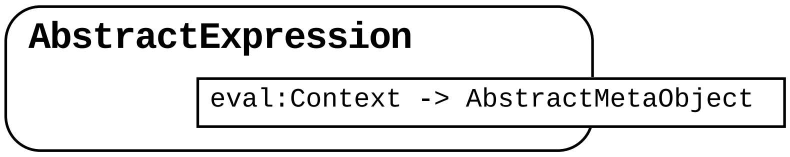

Figure 3.7

The abstract class "AbstractExpression" of which all expression classes will be derived contains one abstract method.

Abstract Class for Expression Objects

```
class AbstractExpression
  methods
   abstract eval:context result AbstractMetaObject
endclass
```

From the viewpoint of the evaluator adding a new expression "simply" involves extending the evaluator in a compositional way. From the viewpoint of, for example, the parser, adding a new expression class is a more complicated matter. For the time being however this problem will be skirted and we will presume that expression objects can be freely added. We will come back on this issue.

3.5.2 Representation of Objects and Full Abstraction

From our discussion about objects in the first part of the text it should be apparent that there exists a plethora of different kinds of objects, e.g. objects that have a class, idiosyncratic objects, objects that have an encapsulated state, objects that don't have an encapsulated state, objects that are defined as a specialisation of another object, primitive objects, objects that can be used as templates, … and so on. What is common to all these types of objects is that they can receive messages. A message consists of a pattern part and an arguments part. For the time being we will leave it open what form the arguments part can take (for Simple this is not very important since only unary messages are sent). We will first take a look at object representations that are too operational.

Objects could be implemented as, for example, sets of slots (named attributes) that can be searched. A hypothetical implementation is given below. Such an implementation is not abstract enough since message passing is not encoded as an atomic operation. Message passing needs to be encoded by the evaluator as looking up a slot and evaluating the body of this slot in some context.

```
class SlotObject
  instance variables
    slots:Set(Slot)
  methods
    concrete lookup:pattern result Slot
      slot := slots findSlot:pattern
      if slot found
        then [^slot]
        else [… raise an error …]
endclass
```

Although, it is presumable that slots will play an important role in the implementation of objects, it is a violation of the encapsulation principle if slots need to be exported for message passing. This representation of objects is not fully abstract. The encapsulation principles that govern at the level of the programming language are not respected in the representation of objects. In particular we will say that such an implementation has a non-encapsulated representation of objects.

So, the implementation of objects must be such that it encodes an atomic message passing operator, and the internal structure (e.g. inheritance structure) of objects remains hidden in the implementation of each object. The abstract class `AbstractMetaObject`[6] (see figure 3.8) presents the protocol that all concrete implementations of objects must have to ensure this.

Objects are represented as instances of concrete subclasses of the `AbstractMetaObject` class. The protocol of this class will include at least a subprotocol to send messages; i.e. each object representation includes at least a send method. The send method should implement message handling for the represented object. The send method accepts a pattern argument and an optional client argument. Accordingly, an actual call of the send method will include the message pattern, and optionally any additional information that must be transferred from the sender object (e.g. arguments of the message, or encapsulated parts in the case of Simple). The actual role of *client objects* will be discussed in a following section.

[6] Apart from speaking about expression objects (objects that implement expressions), context objects (objects that implement contexts), etc., the term meta-objects will be used for objects that implement objects, hence the name of this class.

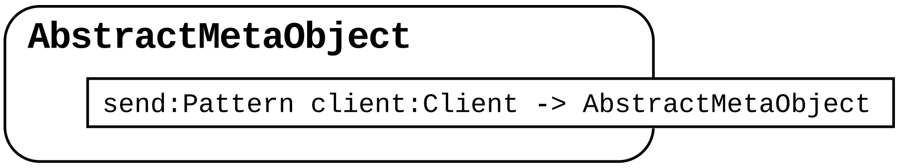

Figure 3.8

Internally, objects can be implemented as, for example, sets of slots, that can be searched. This implementation remains hidden for the user of an object. If this is respected, then objects are represented fully abstract. Stated otherwise, even at the implementation level it is not possible to directly access the private attributes of an object: encapsulation of objects is preserved at the implementation level. This will be called a fully encapsulated implementation of objects.

The class "AbstractMetaObject" of which all meta-object classes are to be derived has one abstract method.

Abstract Class for Meta-Objects

```
class AbstractMetaObject
  methods
   abstract send:pattern client:client result AbstractMetaObject
endclass
```

3.5.3 Message Passing

Given the two abstract classes above, it is possible to express how message passing proceeds. This is encoded in the class "UnaryMessageExpression" as found below (we presume the existence of an empty client object "EmptyClient").

Template Class for Message Passing

```
class UnaryMessageExpression extends AbstractExpression
  instance variables
   receiver:AbstractExpression
   pattern:AbstractPattern
  methods
   concrete eval:context result AbstractMetaObject
     ^(receiver eval:context) send:pattern client:EmptyClient
endclass
```

This class ties together evaluation of expressions and sending messages to objects. Especially because message passing plays such an important role, and because this concrete class ties together two important abstract classes, the expression class for message passing will play an important role in the definition of the framework and it is the intention to be able to inherit it in all extensions to the framework.

Furthermore the above message passing class must be complemented with an abstract class for patterns. Patterns are used as unique identifiers in the implementation of message passing. They essentially implement an equality test.

Abstract Class for Patterns

```
class AbstractPattern
  methods
   abstract = pattern result Boolean
endclass
```

3.6 Concretisation to a Simple Object-based Language

We can now show how the above abstract classes can be made concrete in order to express an evaluator for Simple programs.

3.6.1 Abstraction Expressions and Object Structures

Calculus-objects are created by the evaluation of either the CompoundObjectExpression or the BaseObjectExpression. Both expression classes are concretisations of the abstract expression class. Their evaluation is simply composed of the evaluation of their components and the creation of an according meta-object (i.e. instances of CompoundObject and BaseObject, the code of which will be listed in the next paragraph).

To illustrate all the following class descriptions, they will be preceded by a short explanation. In this explanation underlined terms are syntactic objects that can be evaluated. For example "<a, b>" is a compound object expression. Patterns — instances of the AbstractPattern class — will be represented as #x. All other terms are evaluated objects that can, for example, be sent messages. For example "<a,b>" is a meta-object of type compound object. Note that these descriptions are only illustrative.

```
<publicPart,privatePart> eval:context
  -> <publicPart eval:context, privatePart eval:context>
```

```
class CompoundObjectExpression extends AbstractionExpression
  instance variables
   publicPart:AbstractExpression, privatePart:AbstractExpression
  methods
   concrete eval:context result AbstractMetaObject
     ^CompoundObject
          publicPart:(publicPart eval:context)
          privatePart:(privatePart eval:context)
endclass
```

```
[…; x_i.e_i; … ; y_i#f_i; …] eval:context
  -> […; x_i.e_i eval:context; … ; y_i#f_i eval:context; …]
```

```
class BaseObjectExpression extends AbstractionExpression
  instance variables
   associations:Sequence(AssociationExpression)
  methods
   concrete eval:context result AbstractMetaObject
       local variables slots:Sequence(Slot)
     for each association in associations
        slots add:(association eval:context)
     ^BaseObject slots:slots
endclass
```

More interestingly are the so created meta-objects listed below. Both meta-object types conform to the above given abstract meta-objects, and implement the `send:client:` method. The role of client objects becomes clear in the following code. Client objects are used in the implementation of a compound object to accumulate its encapsulated part with the already existing client object, i.e. the currying rule of the calculus has an almost literal counterpart in the implementation of message passing to compound objects. In the implementation of a base object this client object is then used as a context in which to interpret the body of a selected method. So, the client object is used to carry information from the sender object to the receiver.

```
[…; x_i.e_i; …] send:#x_i client:client
  -> x_i.e_i valueIn:client
```

```
class BaseObject extends AbstractMetaObject
  instance variables
   slots:Sequence(Slot)
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^(slots findSlot:pattern ifAbsent:[^ERROR]) valueIn:client
endclass
```

```
<publicPart, privatePart> send:#x client:client
  -> publicPart send:#x client:<privatePart, client>
```

```
class CompoundObject extends AbstractMetaObject
  instance variables
   publicPart:AbstractMetaObject
   privatePart:AbstractMetaObject
  methods
   concrete send:pattern client:client result AbstractMetaObject
     ^publicPart
        send:pattern
        client: (CompoundObject publicPart:privatePart
                                privatePart:client)
endclass
```

3.6.2 Association Expressions and Slots

Methods and instance variables are added to an object by evaluating association expressions. An association expression is not a true expression. It is not a subtype of the `AbstractExpression`. Notice that the evaluation method is overloaded on `AssociationExpression`. Its evaluation returns a slot rather than an `AbstractMetaObject`.

Abstract Class for Association Expressions

```
class AssociationExpression
  instance variables
   pattern:AbstractPattern
   value:AbstractExpression
  methods
   abstract eval:context result Slot
endclass
```

```
pattern.value eval:context
  -> pattern.(value eval:context)
```

```
class VariableAssociationExpression extends AssociationExpression
  methods
   concrete eval:context result Slot
     ^VariableSlot key:pattern value:(value eval:context)
endclass
```

```
pattern#value eval:context
  -> pattern#value
```

```
class MethodAssociationExpression extends AssociationExpression
  methods
   concrete eval:context result Slot
     ^MethodSlot key:pattern value:value
endclass
```

The class `Slot` encodes instance variable and method slots in an object. They associate the pattern that identifies a slot, to its value in a given context. The class `Slot` is an abstract class that is newly introduced.

Abstract Class for Slots

```
class Slot
  instance variables
   key:AbstractPattern
  methods
   abstract valueIn:context result AbstractMetaObject
   concrete key result AbstractPattern
     ^key
endclass
```

The class Slot has two concrete subclasses for representing instance variables and methods. Their implementation is straightforward.

```
(key.value) valueIn:context
  -> value
```

```
class VariableSlot extends Slot
  instance variables
   value:AbstractMetaObject
  methods
   concrete valueIn:context result AbstractMetaObject
     ^value
endclass
```

```
(key#value) valueIn:context
  -> value eval:context
```

```
class MethodSlot extends Slot
  instance variables
   value:AbstractExpression
  methods
   concrete valueIn:context result AbstractMetaObject
     ^value eval:context
endclass
```

3.6.3 Message Passing

Messages are sent to objects by evaluating expressions of the type UnaryMessageExpression, or P atternExpression. The class UnaryMessageExpression has already been presented. Identifier lookup is interpreted as sending an according message to the context object.

```
pattern eval:context
  -> context send:#pattern client:EmptyClient
```

```
class PatternExpression extends AbstractExpression
  instance variables
   pattern:AbstractPattern
  methods
   concrete eval:context result AbstractMetaObject
     ^context send:pattern client:EmptyClient
endclass
```

Finally, the concrete pattern class can be given, as the last class in our implementation of the calculus. Essentially patterns can be compared for identity. In our encoding of patterns we use overloading on the pattern argument, i.e. the actual class of the pattern argument determines, together with the receiver, what exact equality test method is chosen. In the general case the equality test is applied to the commuted arguments. In the case where two patterns of the same concrete pattern class are compared, the names of the patterns determine equality.

```
class Pattern extends AbstractPattern
  instance variables
    name:String
  methods
    concrete = (pattern:AbstractPattern) result Boolean
      ^pattern = self
    concrete = (pattern:Pattern) result Boolean
      ^name = pattern name
    concrete name result: String
      ^name
endclass
```

3.6.4 Implementation of Simple, Summary

To implement Simple, concretisations of the abstract classes `AbstractExpression`, `AbstractMetaObject` and `AbstractPattern` were given. These classes represent the abstract concepts of expressions, meta-objects, and patterns. The expression class for message passing played an important role in tying all these abstract classes together. These three abstract classes and the expression class for message passing will form the kernel of our framework. As was shown in the previous section, for a first approximation these abstract classes are sufficiently detailed for the derivation of an implementation of a simple object-based language. In the next section we will show that they can be improved.

Furthermore, specifically for the implementation of Simple we introduced two new class hierarchies: that of `AssociationExpression`, and `Slot`. Although they are not a part of the basic framework, we will see that in practice the concept of slots plays an important role in the implementation of object-oriented programming languages. For this reason slots will be made part of the framework. The concept of slots can also be found in [Mulet&Cointe93].

Below are listed the key abstract classes in the implementation of Simple.

Abstract Class for Meta-Objects

```
class AbstractMetaObject
  methods
    abstract send:pattern client:client result AbstractMetaObject
endclass
```

Abstract Class for Expression Objects

```
class AbstractExpression
  methods
    abstract eval:context result AbstractMetaObject
endclass
```

Abstract Class for Patterns

```
class AbstractPattern
  methods
    abstract = pattern result Boolean
endclass
```

Abstract Class for Slots

```
class Slot
  instance variables
   key:AbstractPattern
  methods
   abstract valueIn:context result AbstractMetaObject
   concrete key result AbstractPattern
     ^key
endclass
```

3.7 Improving the Framework

The previous implementation of our Simple object-based programming language is limited with respect to extensibility. Although some of its major design decisions are made explicit (i.e. expressions, expression evaluation, objects, and message passing), we will show in what respects it is a "closed" implementation.

3.7.1 Reifier Methods

The first and most obvious question that is left open in the above implementation is the question of how new expression classes can be added to our Simple object-based programming language, and what the effect of this is on the rest of the environment. As we already said, from the viewpoint of a parser, adding a new expression class is a more complicated matter than it is from the viewpoint of an evaluator. With each new expression class a syntax must be associated from which this new expression class will be generated. Since it is the task of the parser to parse program text into a suitable internal representation, this involves extending the parser such that it recognises this new syntactic construct. This brings us into the realm of languages with an *extensible syntax*. In its most general form in a language with an extensible syntax, arbitrary syntactic constructs can be added to the language. This could for example be supported by a table driven parser.

We will take an approach that can better be termed as the *generic syntax* approach. Rather than allowing arbitrary new syntactic constructs, a few generic syntactic constructs, that will be termed reifier classes, or reifier methods, will be provided that can be instantiated. This approach is very similar to the concept of special forms in Lisp-like languages. It is inspired by reifier functions [Smith82] as found in procedural reflective languages; hence the name. It is also related to structured grammars as found in [De Hondt93].

From the viewpoint of adding new expression classes, compound expressions are the most important. We can identify two sorts of compound expressions: those that have a fixed number of subexpressions of possibly heterogeneous type (*compound expressions*), and those that have a variable number of subexpressions of homogeneous type (*aggregate expressions*). For both a generic syntax can be given. For compound expressions, the syntax is made generic in its use of the keywords that identify the syntactic construct. For aggregate expressions the syntax is made generic in its use of delimiters. In the latter the number of delimiters that can be used is fixed. Other delimiters could be devised, of course. What is important is that keywords and delimiters are separately recognisable lexical symbols.

```
GenericCompoundExpression
  -> [ BoldKeyword Expression { BoldKeyword Expression } ]

GenericAggregateExpression
  ->  LeftAggregateSymbol
       [ Expression { ";" Expression } ]
      RightAggregateSymbol

BoldKeyword            -> BoldIdentifier ":"
BoldIdentifier         -> BoldCharacter { BoldCharacterOrDigit }
BoldCharacter          -> "a" | "b" | …
LeftAggregateSymbol    -> "[" | "{"
RightAggregateSymbol   -> "]" | "}"
```

In a similar way a generic syntax can be given for non-compound expressions.

```
GenericPrimitiveExpression -> BoldIdentifier
```

A *generic expression* is instantiated by giving a concrete set of keywords, or delimiters. With respect to the evaluator, each such instance must be associated with an evaluation method, or stated otherwise it must be associated with an expression class that contains an evaluation method. The translation of an instance of a generic expression to its corresponding expression class can be done either by the parser (statically) or by the evaluator itself (dynamically). In view of our later ambitions of constructing a reflective language we will examine the latter case.

A possible dynamic implementation would translate generic expressions to an internal representation that corresponds to the following abstract grammar for generic expressions. The evaluation method for the components of this abstract grammar is responsible for dispatching to the right evaluation method according to the component's associated keywords, or delimiters.

Generic Abstract Syntax for Compound Expressions

```
ExpressionSet           = …  + { GenericCompoundExpression
                                  GenericAggregateExpression
                                  GenericPrimitiveExpression } + …

GenericPrimitiveExpression     -> IdentifierSet
GenericCompoundExpression      -> (IdentifierSet x ExpressionSet)+
GenericAggregateExpression     -> DelimiterSet x ExpressionSet*
```

Expression Class Corresponding to the Above Abstract Grammar

```
class GenericCompoundExpression extends AbstractExpression
  instance variables
   keywords:Sequence(Identifier)
   subexps:Sequence(Expressions)
  methods
   abstract eval:StandardContext result AbstractMetaObject
     … dispatch to the appropriate evaluation method according
     … to keywords
endclass
```

In principle the entire syntax of Simple can be recast in terms of instances of generic expressions. The syntax of expressions for the construction of base objects, for example, can be seen as an instance of a generic aggregate expression with the "[" and "]" symbols as delimiters. This is, with the current generic expressions, not possible however. In an ordinary abstract grammar, expansion sets introduce hierarchical structuring capabilities, i.e. they are used to put constraints on what kinds of subexpressions can be used in a compound expression. This capability is lacking in the above generic expressions. All subexpressions of a generic compound expression can be arbitrary expressions. We will see how the introduction of evaluation categories amends this situation.

Finally, we note that a more object-oriented view on generic expressions is possible. In this view a compound generic expression is encoded as a special message, the receiver and arguments being expressions. Such an encoding will be called a *reifier message*. Correspondingly, an expression class then has a set of associated *reifier methods* that implement the evaluation for the corresponding reifier messages. In a similar vein generic expressions can be seen as instantiation messages of corresponding expression classes. If this is the case then we talk about *reifier classes*. Both approaches will be elaborated upon in the implementation of Agora.

3.7.2 Extra Indirection Needed in Context and Client Objects

As we saw before, evaluation of expressions is done in a context. In the preceding implementation this context is being built up during method lookup, i.e. method bodies are evaluated in a context that contains information that is local to the object in which the method is being looked up. This is a direct consequence of the encapsulation principle. On the other hand, sometimes it is necessary to transfer information from the sender object to the receiver object (otherwise objects would become autistic). Client objects are used for this purpose.

In the previous implementation, client and context objects were restricted to the encapsulated parts of compound objects. As such, they were directly implemented as instances of either `BaseObject` or `C ompoundObject` (i.e. subclasses of `AbstractMetaObject`). In general, however, this is not possible since contexts and clients must contain other information as well.

Consider adding a self expression to the above calculus with the standard meaning of evaluating to the current receiver. During message passing, the change of receiver must be recorded. Furthermore this information must be accessible when this newly added self expression is evaluated. Rather than extending the evaluator with an extra "self" argument, it seems better to encode this "receiver" information as part of the client and context objects. So, client and context objects must be encoded as aggregates of all the different components that make up the context and the client. Contexts serve as an aggregate for all the information needed by the evaluator; clients serve as an aggregate for all the information passed from sender to receiver.

Below we can find the definitions for the context and client objects that can be used in the above implementation of the calculus. The encapsulated part object that formerly served as client and context is now an instance variable (called `private`) of the explicit client and context objects. The encapsulated part is a *public instance variable* since it can be read and set freely by all users of client and context objects. The evaluator of Simple must be adapted in order to take these definitions into account. This involves replacing all references to a client or a context as object, with respectively "`client private`" and "`context private`", and, creating a client and context object the moment an encapsulated part is turned into a client or context.

```
class StandardContext
 public instance variables
  private:AbstractMetaObject
endclass
```

```
class StandardClient
 public instance variables
  private:AbstractMetaObject
endclass
```

The obvious advantage is extensibility of contexts and clients without having to adapt the entire evaluator. For example, for the introduction of a self expression, clients and context can be extended with an extra field. Furthermore the evaluator must be adapted such that 1) objects, on reception of a message, fill in the current receiver in the client; 2) the current receiver is copied from the client to the context when a selected method body is evaluated, and 3) the self expression is added, with the straightforward evaluation method of returning the current receiver from the context.

```
class ContextWithSelf extends StandardContext
 public instance variables
  currentReceiver:AbstractMetaObject
endclass

class ClientWithSelf extends StandardClient
 public instance variables
  currentReceiver:AbstractMetaObject
endclass
```

The class `SelfExpression` has the following form:

```
class SelfExpression
 methods
  concrete eval:(context:ContextWithSelf) result AbstractMetaObject
    ^context currentReceiver
endclass
```

Two notes should be made here. One is about the apparent parallel between the context and client class hierarchies. Due to this parallel one could be tempted to eliminate one of both. In the implementation of Agora, where objects have a more complex internal structure, we will show that clients and contexts do serve different purposes, and need not parallel each other.

The second is about the (lack of) compatibility between the existing object structures and the extension of the evaluator with a self expression. One part of this extension involves overriding the send method defined on objects such that the current receiver is added to the client object. Only the objects that have this overridden send method can use the self expression in their implementation. In some way this should be reflected in the adapted class hierarchy for expressions. The self expression can not be substituted in all program contexts where an abstract expression is expected[7]. It can be used only in those program contexts where an object that fills in the current receiver, is defined. A mechanism to control this is needed. This will be defined in the following section with the introduction of evaluation categories.

[7] The issue of typing has been avoided up until now, but another way to look at the above problem is that the self expression can not be made a subtype-subclass from `AbstractExpression`. Whereas the evaluation method for `AbstractExpression` has a context argument from `StandardContext` type, and, whereas the evaluation method for `SelfExpression` has a context argument with type `ContextWithSelf`, and whereas `ContextWithSelf` is a subtype of `StandardContext`, it can be concluded that due to the contravariance rule on method arguments, `SelfExpression` is not a subtype of `AbstractExpression`.

3.7.3 Evaluation Categories and Category Patterns

In the calculus and its implementation only side-effect free expressions are considered. When extending the calculus with side effects, one must also consider statements. The essential difference between an expression and a statement is that the former returns a result and the latter does not. Correspondingly, they have different evaluation methods.

Consider extending the calculus with side-effects. This involves adding statements such as a compound and an assignment statement, but also the imperative variant of message expressions.

```
class AbstractStatement
  methods
    abstract eval:context
endclass

class CompoundStatement extends AbstractStatement
  instance variables
    statements:Sequence(AbstractStatement)
  methods
    concrete eval:context
      for each statement in statements
        statement eval:context
endclass

class AssignmentStatement extends AbstractStatement
  instance variables
    pattern:AbstractPattern
    value:AbstractExpression
  methods
    concrete eval:context
      context assign:pattern value:(value eval:context)
endclass

class UnaryMessageStatement extends AbstractStatement
  instance variables
    receiver:AbstractExpression
    pattern:AbstractPattern
  methods
    concrete eval:context
      (receiver eval:context) send:pattern
endclass

class PatternStatement extends AbstractStatement
  instance variables
    pattern:AbstractPattern
  methods
    concrete eval:context
      context send:pattern
endclass
```

Secondly, the object structures must be extended such that imperative messages can be sent. Furthermore a mechanism must be provided to assign a new value to an instance variable attribute of an object. For the time being this latter is resolved in an ad hoc manner; in the framework for Agora a more definitive solution will be considered. Since imperative objects must also be able to accept functional messages, they inherit the standard message passing behaviour from the already defined standard objects. Only the abstract class for imperative objects is listed, all others are straightforwardly implemented.

```
class AbstractImperativeObject extends AbstractMetaObject
  methods
    abstract send:pattern client:client
    abstract assign:pattern value:AbstractMetaObject
endclass
```

Also the slot hierarchy must be adapted. Only the abstract class for imperative slot is listed, in the implementation of the concrete slots the value of an instance variable slot may be reassigned, the value of a method slot not.

```
class ImperativeSlot extends Slot
  methods
    abstract valueIn:context
    abstract value:newValue
endclass
```

The class hierarchy of statements is not related to the class hierarchy of expressions. The drawback of having different class hierarchies is that the according evaluation methods are not related to each other. Moreover, as is the case now, only the refinements of the abstract class of the (side-effect free) expressions are part of the framework. In the same vein, unary message expressions, as documented in the framework, are limited to messages that return a result. In practice other such *expression kinds* can be expected, that are essentially different in the types of the arguments, and result type of the evaluator. In fact, in the implementation of the calculus, the hierarchy of `AssociationExpression` is such an example. Similarly, the self expression of the previous section can be regarded upon as an expression kind that can only be evaluated in a context that keeps track of the current receiver.

To capture the existence of different expression kinds, we will introduce the notion of evaluation categories. Rather than overloading the evaluation method on unrelated expression hierarchies whereby each evaluation method has a possibly different signature, the evaluation method will be overloaded on the context argument also. Much in the style of multi-methods, the selection of the evaluation method does not only take the receiving expression object into account, but also the class of the context. The logic behind this is that, in practice, each expression kind is evaluated in its own particular context class. For example self expressions are evaluated in a context in which the receiver is recorded, statements are evaluated in a context that allows sequencing of expressions, ordinary expressions are evaluated in a standard context, etc. This gives rise to different evaluation categories. When these evaluation categories are encoded by means of overloading all different expression kinds can be part of the same expression hierarchy. Furthermore, one and the same expression can be evaluated in different evaluation categories. An example of this latter is the evaluation of message expressions. Formerly, two, and in the general case an unlimited number of, classes were needed for message expressions. After the introduction of pattern categories we will see how this can be encoded in one and the same class.

Evaluation categories can also be used to reintroduce the hierarchical organisation of syntactic structures in a system employing a generic grammar. Consider a generic compound expression. In principle the subexpressions of this compound expression are not constrained. In some cases they ought to be, however. For example in an extension of Simple with statements, the subexpressions of a statement sequence, must be again statements. To ensure this it suffices that the evaluation method associated with a statement sequence evaluates its subexpression in the appropriate 'statement' evaluation category.

The abstract class for expressions is extended to take overloading on the context into account. The context argument is annotated with a class (i.e. StandardContext). In the annotation of the context argument the + superscript indicates the fact that the evaluation method can be overloaded, in addition to being overridden, in later subclasses. In a concrete subclass the argument of a method that is overloaded on this argument is represented as a couple "(formal argument name: overloaded class name)". An example can be found in the adapted implementation for self expressions.

Rather than introducing a separate evaluation method for the evaluation of imperative expressions (i.e. one that does not return a result), we will expect all imperative evaluations to return a dummy result. In theory a more general solution could be adopted whereby the result of an evaluation is returned in a specially defined result aggregate (much in the style of context and client aggregates). For simplicity reasons we adopt the more ad hoc solution of returning dummy results.

Abstract Expressions with Overloading on the Context Argument

```
class AbstractExpression
  methods
   abstract eval:StandardContext+ result AbstractMetaObject
endclass
```

The self expression of the previous section now takes the following form:

```
class SelfExpression extends AbstractExpression
 methods
  concrete eval:(context:StandardContext) result AbstractMetaObject
    ERROR("the self expression must be evaluated in an appropriate
context")
  concrete eval:(context:ContextWithSelf) result AbstractMetaObject
    ^context currentReceiver
endclass
```

Similarly to evaluation categories, message pattern categories are introduced, although the introduction of different message kinds, such as imperative and side-effect free messages, occurs less often. The class AbstractMetaObject is extended such that the possibility is left open to override the send method on the type of the pattern argument.

Abstract Meta-Object with Overloading on the Pattern Argument

```
class AbstractMetaObject
  methods
   abstract send:AbstractPattern+ client:StandardClient
       result AbstractMetaObject
endclass
```

Since pattern equality is dependent on the classes these patterns belong to, pattern categories will mainly be used for overloading one and the same pattern in different categories. For example, a functional and an imperative method can be given the same pattern name, they will be differentiated by the categories of their respective patterns. So, in general, pattern categories will not be used to overload the send method for objects, but rather to differentiate, in looking up a pattern, patterns from different categories.

Finally, the class UnaryMessageExpression can be adapted so that it is compatible with all of the above. This class is a typical example of an expression class that can be evaluated in different evaluation categories. The one thing that must be given special attention is that the pattern that is used in sending the message to the evaluated receiver object inherits the category from the evaluation context in which the entire message expression is evaluated. For this purpose the asCategory method is introduced on patterns. This is an ad hoc solution, in the section on classifiers and traces we will show a better solution for this problem.

Message Passing with Evaluation and Pattern Categories

```
class UnaryMessageExpression extends AbstractExpression
  instance variables
   receiver:AbstractExpression
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandardContext+) result AbstractMetaObject
     ^(receiver eval:context)
         send:(pattern asCategory:context)
         client:EmptyClient
endclass
```

Adapted Abstract Pattern Class

```
class AbstractPattern
  methods
   abstract = pattern result Boolean
   abstract asCategory:StandardContext+ result AbstractPattern
endclass
```

```
--- example of adapted concrete pattern
class Pattern
  instance variables
    name:String
  methods
    …
    concrete asCategory:(context:FunctionalContext)
        result AbstractPattern
      ^FunctionalPattern name:name
    concrete asCategory:(context:ImperativeContext)
        result AbstractPattern
      ^ImperativePattern name:name
    …
endclass
```

3.7.4 Making the Layered Structure Explicit

Up until now the layered structure in the implementation of Simple is not made explicit. Some of the concrete classes of this implementation can be made more reusable by the introduction of abstract class attributes. An example is given below.

Final, More Reusable Version of Compound Object Expressions

```
class AbstractCompoundObjectExpression extends AbstractionExpression
  abstract class attributes
   ACompoundObject
  instance variables
   publicPart:AbstractExpression, privatePart:AbstractExpression
  methods
   concrete eval:context result: AbstractMetaObject
     ^ACompoundObject
        publicPart:(publicPart eval:context)
        privatePart:(privatePart eval:context)
endclass
```

Concretisation to 'Standard' Compound Object Expressions

```
class CompoundObjectExpression extends AbstractCompoundObjectExpression
  concrete class attributes
   ACompoundObject:CompoundObject
endclass
```

All expression classes in the hierarchy must be adapted accordingly, i.e. the classes BaseObjectExpression, VariableAssociationExpression, and MethodAssociationExpression must each be constructed as an abstract class with an abstract class attribute for the creation of instances of respectively BaseObject, VariableSlot, and MethodSlot.

An implementation in which all (meta-) objects are created by means of abstract class attributes is more open-ended than one in which the classes of the meta-objects are 'hard-coded'. Typical extensions of the evaluator that make use of this are debuggers. Rather than making the expression class hierarchy concrete with the standard implementations of meta-objects, the expression class hierarchy is made concrete with meta-objects with debugging facilities.

Other example usages of this layered structure can be found in optimisation of object representations, i.e. all meta-object classes can be overridden with meta-object classes that represent objects more efficiently.

3.8 Conclusion

To summarise we list the key classes involved in the improved framework used to implement and extend Simple.

The class AbstractExpression was extended such that different sorts of evaluation can be dealt with. Furthermore, context objects are introduced to bundle all the arguments of the evaluator. Context object are also instrumental for overloading the evaluation function to obtain different *evaluation categories*. A special notation was introduced for overloaded arguments.

Abstract Expressions with Overloading on the Context Argument

```
class AbstractExpression
  methods
    abstract eval:StandardContext+ result AbstractMetaObject
endclass
```

Standard Context Object for Simple, Grouping All Evaluation Arguments

```
class StandardContext
 public instance variables
  private:AbstractMetaObject
endclass
```

The notion of generic expressions was introduced so that the framework can be extended with new kinds of expressions. Three sorts of generic expression were discussed.

It was shown that the expression class hierarchy must be extended with abstract classes with abstract class attributes for capturing the layered structure of the implementation of Simple. The adapted class hierarchy is shown below. Although the so obtained abstract classes are specific to the implementation of Simple (not all object-based languages need to have the notion of e.g. compound objects), a good rule of thumb can be distilled from this experience. This rule of thumb says that in an implementation all meta-objects must be created by means of abstract class attributes. This rule will be followed in the next sections.

Simple Expression Class Hierarchy

```
AbstractExpression
  AbstractionExpression
    AbstractCompoundObjectExpression (ACompoundObject)
      CompoundObjectExpression (publicPart:AbstractExpression,
                                privatePart:AbstractExpression)
    AbstractBaseObjectExpression (ABaseObject)
      BaseObjectExpression (associations:Sequence(AssociationExpression))
  UnaryMessageExpression (receiver:AbstractExpression,
                          pattern:AbstractPattern)
  PatternExpression (pattern:AbstractPattern)
AbstractAssociationExpression (AAssociation)
  AssociationExpression (pattern:AbstractPattern, value:AbstractExpression)
    VariableAssociationExpression
    MethodAssociationExpression
AbstractPattern
  Pattern(name:String)
```

The class `AbstractMetaObject` was extended such that the send method can be overloaded on the pattern type. Furthermore, client objects are introduced to bundle all the arguments of the send method.

Abstract Meta-Object with Overloading on the Pattern Argument

```
class AbstractMetaObject
  methods
    abstract send:AbstractPattern+ client:StandardClient
        result AbstractMetaObject
endclass
```

Standard Client Object for Simple, Grouping all Send Arguments

```
class StandardClient
 public instance variables
  private:AbstractMetaObject
endclass
```

The class `UnaryMessageExpression` was adapted to take evaluation and pattern categories into account. `UnaryMessageExpression` is an expression type that can be evaluated in any possible evaluation category. The pattern class is extended so that evaluation categories can be 'inherited'.

Message Passing with Evaluation and Pattern Categories

```
class UnaryMessageExpression extends AbstractExpression
  abstract class attributes
   EmptyClient
  instance variables
   receiver:AbstractExpression
   pattern:AbstractPattern
  methods
   concrete eval:(context:StandardContext+) result AbstractMetaObject
     ^(receiver eval:context)
         send:(pattern asCategory:context)
         client:EmptyClient
endclass
```

Adapted Abstract Pattern Class

```
class AbstractPattern
  methods
   abstract = pattern result: Boolean
   abstract asCategory:StandardContext result: AbstractPattern
endclass
```