Constructing an Object-Oriented Program in a Non Object-Oriented Language

N. Lévy

CRIN-CNRS & INRIA LORRAINE
BP. 239, F-54506 Vandœuvre-les-Nancy Cedex (France)

Abstract: Mastering software development is a challenge. Up to now, all the propositions done have come out with new languages. In this paper, we present a development framework in which different approaches can be supported and the program is a component. Within the environment, development operators enable the incremental construction and modification of programs, separating the use of design concepts from the technical details of how they are captured in the programming language. They also offer flexibility since it is possible to define a library of operators capturing alternative definitions of particular concepts and strategies.

The focus of this paper is to show how object-orientation is introduced without a new language. A set of development operators enabling object-oriented software development is defined and then used in a simple case study.

1 Introduction

Looking at its history, it may seem that computer science is devoted to define languages. Any new approach, method or system is associated with a new language. But, are we sure that mastering software development needs to always define new languages? Using object-orientation [Mey88, Boo90, RBP+91] has proven to be successful [DEKW94]. The object model is a good means to clarify and understand the application that is to be implemented. But again, new languages [GR83, Str86, Mey91] have come express these new concepts.

We believe that to master software development, we have to be able to make clearer the difference between the concepts to be used and the syntax. The expression of the taken decisions is important and will facilitate going back on them. An incremental library of decisions could help to reuse not the programs (the syntax), but the reasoning (the semantics).

In this paper, we present a development framework [SL93, Sou93a] in which different approaches can be supported and the program is a component. Within the environment, development operators enable the incremental construction and modification of programs. The operators enable the programmers to develop programs in an intuitive fashion by separating the use of design concepts from the technical details of how they are captured in the programming language. They also offer flexibility since it is possible to define a library of operators capturing alternative definitions of particular concepts and strategies.

The development framework is the basis of an interactive programming development environment [BS93] implemented in the Esprit project ICARUS. This environment has two main features.
Firstly, it is language-independent [LS94]. Therefore, it can be used with existing languages. Secondly, it records the design decisions taken during the development. This information represents a deeper understanding of the system than the product alone. It facilitates constructing and making evolve the programs and may also aid the steps of refinement and maintenance [Smi94].

The focus of this paper is to show how object-orientation is possible without introducing a new language. A set of development operators enabling object-oriented software development are defined and then used in a simple case study. The non-object-oriented programming language used is Isetl. Isetl [BDL89, Lev90], based on SETL [SDDS86], is a procedural language containing a rich set of expressions. Let us note that in Isetl the variables are not typed: only some predefined data types are available.

Section 2 describes the development framework. Sections 3 present the operators for object-oriented development. These operators are illustrated in section 4 by the development of the class DATE.

2 Development Framework

This section describes the proposed development framework [SL93, Sou93b]. The development of a program can be viewed as a sequence of steps where each step corresponds to the application of a development operator to the development state. The development state consists of a workplan, denoting the history of goals and development decisions, and a product, denoting the program. Links associate the workplan and product.

2.1 Workplan

The workplan consists of a collection of tasks describing goals to be achieved in order to produce the required program. A task is reduced by the application of a development operator. A reduction consists of decomposing the task into several subtasks which together achieve the same goal as the original task. For example, Figure 1 shows the reduction of the task Define DATE, by the development operator Defining Class, introducing the subtasks Define Attributes of Class DATE, Define Invariant of Class DATE and Define Operations of Class DATE.

![Diagram](image)

Figure 1: Workplan Corresponding to the first step of the Construction of the class DATE

In general, there may be several possible reductions for any given task. A task may, therefore, be reduced by a number of different development operators.

Often it is desirable to decompose a task into its subtasks progressively, i.e. to apply an operator which produces only one, or a limited number, of subtasks at a time. For example,
the task Define Operations of Class DATE could be reduced to a single subtask Define Operation Next Day of Class DATE and then later, after the operation is completed, a subtask Define Operation Next Week of Class DATE could be added. To enable this, a reduction can either be partial (represented by a non-shaded circle) or complete (represented by a shaded circle as in Figure 1). A partial reduction is one which allows the list of its subtasks to be extended.

Subtasks of a given task may also be related by a precedence relation indicating that certain of them should be reduced before others. A precedence relation is a strict partial order on tasks represented graphically by a dashed arrow between the tasks as in Figure 1. It acts as a hint to the programmer as to which subtask to reduce first.

2.2 Product

The product consists of formal text in a given programming language. During the development of a program, an incomplete part of the product (i.e. a part not yet defined), is represented by a typed placeholder which is linked to a single non-reduced task in the workplan. The type of the placeholder corresponds to the kind of program text it can be replaced with. A placeholder is denoted between { }. When the product represented by some placeholder is a list, it is denoted between *{ }* (for example *{ ITEM}* denotes the placeholder for a list of items). In order to define the part of a program a placeholder represents, a development operator is applied to the associated task and its subsequent subtasks. When this occurs, the links between the workplan and product are maintained so that each completed part of the product is linked to the tasks in the workplan which produced it.

The Programming Language Isetl

Isetl [BDL89, Lev90] is an interactive implementation of SETL [SDDS86], a programming language built around mathematical notations. It contains the usual collection of statements common to procedural languages, as Pascal, but a richer set of expressions. Isetl is provided with some predefined types like sets and tuples. To test the type of an expression, Isetl provides the operations is Type (e.g. is integer). But it is not possible to define new types nor classes neither to type the used variables.

2.3 Development Operators

Development operators work simultaneously on the workplan and product to reduce tasks and construct or modify the associated product text. They obtain parameters interactively from the specifier and from the existing workplan and product. Parameters will be denoted by words in upper-case (like NAME and PARAM in Figure 2). The parameters obtained from the workplan are given in the name of the operator (NAME in Figure 2 is such a parameter). The parameters obtained from the specifier are preceded in the operator definition by a message which is printed by the environment asking the specifier for a value of a given class (PARAM in Figure 2 is such a parameter).

Operators may introduce subtasks that will in turn, be reduced by the application of development operators. Such subtasks are associated with placeholders (such as ITEM1 in Figure 2) in the product text. The precedence relationship between these subtasks is indicated by the clause precedence.

Development operators consist primarily of a language-independent section which describes the action on the workplan, followed by a language-dependent section. The general form of an
operator Defining NAME
status complete or partial
interactive parameter
  Message to enter the parameter: PARAM
subtasks
  1: first subtask defines \langle ITEM_1 \rangle
  2: second subtask defines \langle ITEM_2 \rangle
precedence for example: 1 < 2
lsetl product text
\langle lsetl text to be defined \rangle \rightarrow lsetl text defined with NAME, PARAM, \langle ITEM_1 \rangle, \langle ITEM_2 \rangle

Figure 2: General Schema of a Development Operator

The language-independent section introduces the name (Defining), the status of the operator, the parameters (NAME and PARAM), the subtasks it introduces and the precedence relation between these subtasks.

The language-dependent product part introduced by an operator is composed of keywords of the programming language, parameters (e.g. NAME), parts introduced manually by the user as interactive parameters (e.g. PARAM), placeholders (e.g. \langle ITEM_1 \rangle) associated with the subtasks and parts computed by product operators. The use of such product operators will be illustrated in Section 3.1.

To be applicable to a task, an operator must modify the kind of product text associated with the task. The placeholder \langle ANY \rangle is the most general representing any kind of product text.

2.4 Basic Operators

The construction process involves resolving (or reducing) a task, either by introducing new subtasks or finishing the work assigned to the task. This process is repeated, step by step until there are no more unreduced tasks. Some operators are totally method- or language-independent such as the ones presented below.

To specify an item in a list, the operator Introducing Item is first applied to introduce the corresponding placeholder.

operator Introducing Item of NAME
status partial
interactive parameter
  Name of the Item: ITEM_NAME
subtasks
  1: Define ITEM_NAME of NAME defines \langle ANY \rangle
lsetl product text
\langle ANY \rangle \rightarrow \langle ANY \rangle \ast \langle ANY \rangle \ast

For any natural number n (n can be equal to 0), an operator Reducing to n Items can be defined that introduces n subtasks and whose status is complete. For example, for n equal to two:
operator Reducing to 2 Items of NAME
   status complete
   interactive parameters
   Name of the first Item : ITEM_NAME_1
   Name of the second Item : ITEM_NAME_2
subtasks
   1 : Define ITEM_NAME_1 of NAME defines (ANY_1)
   2 : Define ITEM_NAME_2 of NAME defines (ANY_2)
Isetl product text
   *(ANY)* —> (ANY_1) (ANY_2)

Sometimes, a placeholder is introduced that can be resolved by nothing. This happens for all optional parts of a formal text. The operator Defining by Nothing allow placeholders to be removed in such cases.

operator Defining by Nothing
   status complete
   Isetl product text
   (ANY) —>

At any time during a development, the specifier may want to directly define a piece of the program, without using an operator, i.e. with just the text editor. In such cases, the operator Terminating can be applied. This operator is always applicable, i.e. at any time and to any task. It increases the flexibility of the approach. Naturally, the interactively given text must be syntactically-correct with respect to the chosen programming language, here Isetl.

operator Terminating
   status complete
   interactive parameter
   Value : STRING
   Isetl product text
   (ANY) —> STRING

3 Operators for Object-Oriented Programming

The object-oriented approach to programming adopted in this paper, views a system as a class defined in terms of objects of other classes. Generally, a class consists of a structure that is defined by a set of attributes and integrity constraints (called invariant) and a set of operations associated to the class. A class can also be defined by inheritance but, because of a lack of space, this will not be studied here. A class is composed of some operations to construct, modify and observe objects of the class. In this paper, we will just define the operators to be used to define the constant Init and the operations to modify the value of an attribute.

3.1 Defining a Class

A Class describes some object model by its static data structure. This structure is defined by a set of attributes. The attributes of an object are some typed values. Two operations are needed
for each attribute: one to access to it (in fact it is the attribute), and one to modify its value. An initial object named Init, is also to be defined for each class, so an initial value must be provided for each attribute.

In Isetl, the variables are not typed, so to model object-orientation, the objects defined must carry them-self the name of their class. The operator Defining Class defines an object in Isetl as a couple of its class and structure. In order to allow inheritance and polymorphism, that is the fact that an object can be of several classes, the class of an object will be a sequence (or a tuple in Isetl) of its classes, from the lower in the hierarchy to its farwest ancestor.

```
operator Defining Class NAME 
  status complete 
  subtasks 
    1: Define Attributes of Class NAME defines *(<ATT_NAME>(n) <TYPE>(n) 
                                           <VALUE>(n) <INVARINT>(n))_n
    2: Define Invariant of Class NAME defines <INVARINT>
    3: Define Operations of Class NAME defines *(<OPERATION>)* 

Isetl product text
  <ANY> ——>
    is_NAME := func(lower(NAME));
    return is_tuple(lower(NAME)) and #lower(NAME) = 2 
    and is_tuple(lower(NAME)(1)) and lower(NAME)(1)(1) = "NAME"
    *(and is_<TYPE>(n)(Access..<ATT_NAME>(n)(lower(NAME)))) and <INVARINT>(n))_n
    and <INVARINT>
  end func;
  Init := func();
  return [["NAME"], [[]<VALUE>(n)]_n]]
  end func;
  *(Access..<ATT_NAME>(n) := func (lower(NAME));
    if is_NAME(lower(NAME))
      then return lower(NAME)(2)((n))
      else return "ERROR"
    end if
  end func;
  Modif..<ATT_NAME>(n) := func (lower(NAME), new_value);
    local lower(NAME);'
    if is_NAME(lower(NAME)) and is_<TYPE>(n)(new_value)
      then lower(NAME')(2)((n)) := new_value;
        if is_NAME(lower(NAME'))
          then return lower(NAME)
        else return "ERROR"
      end if
    else return "ERROR"
  end if
  end func;
  *(<OPERATION>)*
```

The operation is_NAME has a parameter denoted as lower(NAME). Here, we invent a variable name: the name of the class in small letters (computed by the product operator lower). For example, if the class to be defined is DATE, the variable will be date.
Let us note that a placeholder to be defined by a subtask, can be used several times in the product. For example \( \langle \text{TYPE}\rangle(n) \) is used in the operation \( \text{is.NAME} \) and in \( \text{Modif.}_n \langle \text{ATT.NAME}\rangle(n) \). Naturally, when defined, every occurrence of the placeholder will be replaced by its value.

### 3.2 Defining an Attribute of a Class

The structure of the objects of a class is defined by the attributes. An attribute is characterised by a name (of the access function), a type and an initial value. Its value can also be restricted by an invariant.

```plaintext
operator Defining Attribute ATT_NAME of Class NAME
  status complete
  interactive parameters
    Class of the attribute : TYPE
    Initial value of the attribute : VALUE
  subtask
    1 : Define Property Required for the Attribute ATT_NAME of Class NAME
defines 〈INARIANT〉

Isel product text
〈TYPE〉 → TYPE
〈VALUE〉 → VALUE
〈INARIANT〉 → 〈INARIANT〉
```

This operator puts the interactively entered datas (TYPE and VALUE) to their right places in the program. The attribute invariant is left undefined: it can be either defined by application of the operator Defining Invariant or, if there is no special restriction to indicate, deleted by the operator Defining by Nothing.

### 3.3 Defining an Invariant

A class invariant restricts the possible values of the attributes. It is be expressed by a boolean formula. Let us note that the formula given by the programmer must be syntactically correct with respect to Isel. In Isel, the invariant will be used as an additional test made by the operation \( \text{is.NAME} \). The operator Defining Invariant will be applied for defining a general invariant and the property required for an attribute.

```plaintext
operator Defining Invariant of NAME
  status complete
  interactive parameter
    Formula of the Invariant : INARIANT

Isel product text
〈INARIANT〉 → INARIANT
```

### 4 Case Study: Define the Class DATE

To illustrate our approach, we will show some steps of the construction of the class describing dates. We consider a date as being composed of a day, a month and a year.
In order to define the class DATE, we first introduce the task Define DATE and we reduce it by application of the operator Defining Class. This operator introduces three subtasks:

1: Define Attributes of Class NAME
2: Define Invariant of Class NAME
3: Define Operations of Class NAME

Figure 1 shows the worplan obtained after this first step of the development. To define the three attributes of the class DATE (the day, the month and the year), we first have to introduce each of them: either all together by application of the operator Reducing to 3 Item or one after the other by applying three times the operator Introducing Item and once Defining by Nothing in order to complete the reduction (prohibiting the addition of new attributes). In these two cases, the names of the attributes (Day, Month, Year) are given as interactive parameters and the resulting program is the same.

Then to define the type and initial value of the attributes, we apply the operator Defining an Attribute three times with respectively the following interactive parameters for the type and the initial value of each attribute:

Attribute Day: INTEGER and 1; a day is an integer and we initialise it to 1;
Attribute Month: INTEGER and 1; a month is an integer and we initialise it to 1;
Attribute Year: INTEGER and 1994; a year is an integer and we initialise it to 1994.

The operator Defining an Attribute introduces a subtask Define Property Required for the Attribute. We reduce the three subtasks introduced by application of the operator Defining an Invariant with the following formulas as interactive parameters.

Attribute Day: Access.Day (date) in [1..31]; a day belongs to the interval [1..31];
Attribute Month: Access.Month (date) in [1..12]; a month belongs to the interval [1..12];
Attribute Year: Access.Year (date) in [1994..]; a year is greater than 1994.

The constraint that we have 30 days in April etc, will be given as invariant of the class with the following formula as interactive parameter:

\[(\text{Access.Day}(\text{date}) \neq 31 \text{ or } \text{Access.Month}(\text{date}) \in \{1, 3, 5, 7, 8, 10, 12\})
\]
\[\text{and } (\text{Access.Month}(\text{date}) \neq 2 \text{ or } \text{Access.Day}(\text{date}) \notin \{30, 31\})\]
\[\text{and } (\text{Access.Month}(\text{date}) \neq 2 \text{ or } \text{Access.Day}(\text{date}) \neq 29 \text{ or } \text{Access.Year}(\text{date}) \mod 4 = 0 )\]

The workplan resuming the approach followed is given in Figure 3. The Isel program constructed is as follows:

```isel
is_DATE := func(date);
    return is_tuple(date) and #date = 2
        and is_integer(date(1)) and date(1)(1) = "DATE"
        and is_INTEGER(Access.Day(date))
        and Access.Day(date) in [1..31]
        and is_INTEGER(Access.Month(date))
        and Access.Month(date) in [1..12]
        and is_INTEGER(Access.Year(date))
        and Access.Year(date) in [1994..]
        and (Access.Day(date) \neq 31 \text{ or } Access.Month(date) \in \{1, 3, 5, 7, 8, 10, 12\})
        and (Access.Month(date) \neq 2 \text{ or } Access.Day(date) \notin \{30, 31\})
        and (Access.Month(date) \neq 2 \text{ or } Access.Day(date) \neq 29 \text{ or } Access.Year(date) \mod 4 = 0 )
end func;
```

1238
Figure 3: Workplan of the First Steps of the Construction of Class DATE

Init := func();
    return [["DATE"], [1, 1, 1994]]
end func;

Access_Day := func (date);
    if is_DATE(date)
        then return date(2)(1)
        else return "ERROR"
    end if
end func;

Modif_Day := func (date, new_value);
    local date';
    if is_DATE(date) and is_INTEGER(new_value)
        then date'(2)(1) := new_value;
            if is_DATE(date')
                then return date
                else return "ERROR"
            end if
        else return "ERROR"
    end if
end func;
Access_Month := func (date);
  if is_DATE(date)
    then return date(2)(2)
    else return "ERROR"
  end if
end func;

Modify_Month := func (date, new_value);
  local date';
  if is_DATE(date) and is_INTEGER(new_value)
    then date'(2)(2) := new_value;
      if is_DATE(date')
        then return date
        else return "ERROR"
      end if
    else return "ERROR"
  end if
end func;

Access_Year := func (date);
  if is_DATE(date)
    then return date(2)(3)
    else return "ERROR"
  end if
end func;

Modify_Year := func (date, new_value);
  local date';
  if is_DATE(date) and is_INTEGER(new_value)
    then date'(2)(3) := new_value;
      if is_DATE(date')
        then return date
        else return "ERROR"
      end if
    else return "ERROR"
  end if
end func;

"(OPERATION)"

5 Conclusion

This paper has presented a language-independent framework for the development of programs. The framework is based on that proposed in the Esprit project ICARUS concerned with the incremental construction of specifications. Development operators are used to simultaneously construct a workplan and a product, denoting the program itself. The workplan denotes the history of goals or tasks to be achieved and the corresponding applications of operators to these goals, also called reductions of tasks.

The object-oriented approach was reflected by the definition of a little set of operators which were instantiated for the programming language Isel.

The development operators consist of a language-independent section which describes the action on the workplan and of a language-dependent section describing the product text defined by
the operator. This language-dependent text is composed of keywords of the programming language, the operator parameters, parts introduced manually by the user as interactive parameters, placeholders associated with subtasks and parts computed by product operators. The operators effectively hide the technical details of how object-oriented concepts are captured in the programming language, making program development more intuitive. Therefore, certain choices as to how the concepts are to be captured are made by the operators. However, other operators reflecting different choices could also be defined.

It is possible to extend the defined operators to be used with other languages. The set of operators defined must also be completed by other operators, for example to define a class by inheritance or to introduce polymorphism.

The practicality of the proposed approach is dependent on an environment which manages the workplan and product and the links between them. Such an environment is under development within the Esprit project ICARUS. A prototype of the environment is already implemented and allows simple operators to be chosen via a menu which is dynamically defined according to the current task type. A more flexible notion of operator and an interface to facilitate the implementation of such operators is presently being developed.

References


[GR83] A. Goldberg and D. Robson. *Smalltalk 80: The Language and its Implementation*. Addison-Wesley, 1983.


Curriculum:

Nicole Lévy is born in 1958 in Mexico City where she lived until 1976. She is graduated from the University of Paris VII and has a doctorate from the University of Nancy I in 1984 on tools for constructing and transforming algebraic specifications of abstract data types. She is assistant professor (*maître de conférences*) at the University of Nancy I since 1984. Her research work is centered around the tools to help constructing, modifying and reusing formal specifications and programs. She participated, among other projects to the European Esprit project ICARUS and collaborated to the definition of the framework and environment on which is based the present paper.

Main References


